UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MARCO DE TRABAJO DE RENDERING NO FOTORREALISTA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

EDUARDO NICOLÁS GRAELLS GARRIDO

PROFESORA GUÍA:
SRA. MARÍA CECILIA RIVARA ZÚÑIGA.

MIEMBROS DE LA COMISIÓN:
SR. MAURICIO PALMA LIZANA
SR. BENJAMÍN BUSTOS CÁRDENAS

SANTIAGO DE CHILE
ABRIL DE 2008

Resumen

En Computación Gráfica uno de los conceptos más importantes es el *Rendering*, definido como el proceso que a partir de los datos de una escena tridimensional genera una imagen en una ventana de visualización en 2 dimensiones. El rendering siempre se ha enfocado en generar imágenes fotorrealistas, aunque para diversas aplicaciones y escenarios se desea una imagen abstracta y/o estilizada. El proceso que puede generar ese tipo de imágenes es llamado *Rendering No Fotorrealista*, o NPR por sus siglas en inglés (*Non Photorealistic Rendering*). El NPR integra conceptos de Computación Gráfica y de Geometría Diferencial, y puede aplicarse en visualización científica e ilustración técnica, con el fin de destacar áreas o contornos de los objetos que son de interés para el observador; en la producción de imágenes que imiten medios tradicionales como la pintura o la acuarela; y en la generación de imágenes que imiten dibujos animados, entre otras áreas.

En esta memoria se diseña e implementa un marco de trabajo (*framework* en inglés) para Computación Gráfica que permite desarrollar técnicas NPR y de rendering tradicional. El diseño del marco de trabajo contempla el estudio de los puntos comunes de diferentes técnicas de NPR y de las áreas asociadas, y, siguiendo el paradigma de la Orientación a Objetos, define distintas componentes internas que pueden ser extendidas para implementar técnicas de NPR y de rendering tradicional. La implementación se realiza en el lenguaje de programación `C++` utilizando bibliotecas de código abierto o compatibles con código abierto: se utilizan la biblioteca gráfica OpenGL y el lenguaje Cg para comunicarse con los procesadores de la tarjeta de vídeo (GPU).

El marco de trabajo incluye técnicas ya implementadas: para el rendering tradicional se implementaron graficadores de Gouraud y de Phong, mientras que para las técnicas NPR se implementaron graficadores de dibujado animado (Cel-Shading) y de modelos de tonos (Gooch Shading). Se implementaron extractores de líneas de interés (Siluetas y Contornos Sugestivos), así como una clase abstracta que permite definir nuevos tipos de líneas a extraer. La demostración de estas implementaciones se realiza mediante una aplicación de prueba que permite cargar modelos tridimensionales estáticos y animados; los resultados obtenidos se muestran mediante imágenes de los modelos tridimensionales cargados en la aplicación de prueba.

Finalmente se realiza una discusión sobre el trabajo futuro, sobre las posibles líneas de investigación a seguir en estudios de post-grado y sobre el uso del marco de trabajo para implementar técnicas de rendering no NPR. Se concluye que el marco de trabajo se puede utilizar para el desarrollo de técnicas NPR y de rendering tradicional, que su desempeño es adecuado para el rendering en tiempo real para escenas tridimensionales de tamaño medio y que puede ser utilizado en aplicaciones reales.

AGRADECIMIENTOS

"Cualquier destino, por largo y complicado que sea, consta en realidad de *un solo momento*: el momento en que el hombre sabe para siempre quién es."
Jorge Luis Borges – Biografía de Tadeo Isidoro-Cruz

"Sea el límite del tiempo o del espacio, no hay nada que inspire mayor horror que un final."
Yukio Mishima – Nieve de Primavera

Mishima siempre supo cuándo, cómo y dónde sucedería *su* final. Lo cierto es que Mishima fue un gran hombre, un verdadero genio. Por otro lado, nosotros, los hombres comunes, no sabemos cuándo sucederá *ese* final tan temido, o más bien, no sabemos ni podemos determinar cuántos finales nos quedan. Una única cosa es cierta: el tiempo que nos queda no es ilimitado. Creemos tener toda la eternidad por delante, aunque solamente tenemos un puñado de días que se deslizan por nuestra vida a una velocidad incontrolable.

Estas dos citas pueden resumir lo que siento respecto a esta memoria. Este trabajo marca en final de una etapa y a la vez un nuevo comienzo. Los estudios que he realizado, incluyendo todos los tropiezos y errores, y unos pocos aciertos, y la gente que he conocido en mi camino, ha definido ese único momento del cual consta mi destino. Por ello, quiero dedicar mi trabajo a mi familia, siempre tan cariñosa y comprensiva, ha sabido querer a un joven siempre misterioso y problemático; y al amor de mi vida, y nuestra familia de Pajaritos de colores, con quienes decidimos construir un mismo hogar y seguir un mismo camino; asimismo, quiero agradecer a los profesores María Cecilia Rivara y Ricardo Baeza-Yates, dos personas que, sin duda alguna, han sido (son) modelos a seguir, siempre dispuestos a aconsejar y guiar; y mis amigos de Ryuuko y Ficciones, con quienes hemos emprendido una larga aventura juntos.

También dedico este trabajo a la memoria de mi abuelo, Francisco de Paula Graells del Campo. El tiempo ha borrado poco a poco la imagen que tengo de él, pero no el cariño ni los buenos recuerdos.

Índice general

Capítulo 1

Introducción

Desde sus inicios, la Computación Gráfica ha tenido entre sus objetivos lograr que la representación de una escena en tres dimensiones sea lo más realista posible. El proceso que genera esa representación es llamado *rendering*, y este trabajo de título se enmarca dentro de un sub-área del rendering llamada *rendering no fotorrealista*, o *NPR* por sus siglas en inglés (*Non Photorealistic Rendering*).

El NPR recibe como entrada una escena tridimensional y la grafica de un modo que no es necesariamente realista, sino más bien *abstracto*, con el fin de acentuar detalles, características o formas de los objetos. Esto se logra mediante el trazado de líneas, la formulación de modelos de iluminación y el pintado interior de los objetos emulando medios tradicionales. Entre sus aplicaciones se encuentra la ilustración y animación estilo dibujo animado o caricatura, la simulación de imágenes generadas con medios tradicionales, la ilustración técnica, la impresión de planos, visualización geográfica, algunas técnicas de visualización científica, entre otras.

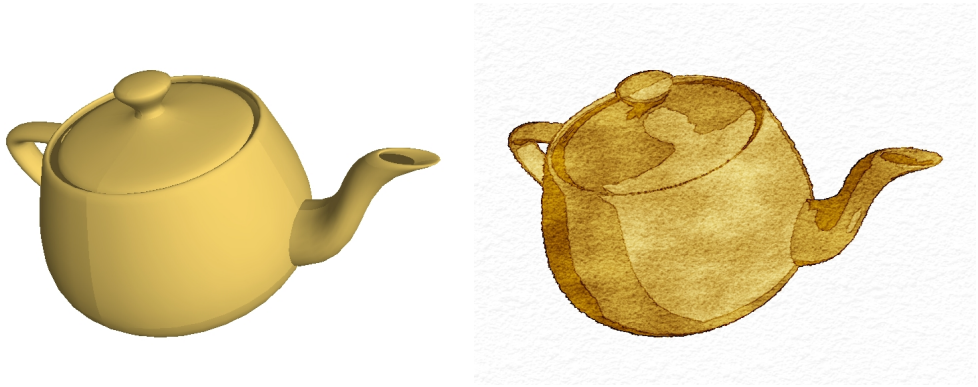


Figura 1.1: *La tetera graficada tradicionalmente y en estilo acuarela. Fuente: [BKTS06].*

En la Figura 1.1 se puede apreciar una técnica NPR aplicada a uno de los objetos más conocidos en Computación Gráfica, la tetera de Utah. Se muestra su graficación original y una graficación que simula ser una acuarela. Un punto importante, implícito en la imagen, es el siguiente: una técnica NPR es *independiente* del objeto que se está graficando, lo que quiere decir que las técnicas NPR se pueden aplicar a cualquier modelo tridimensional.

Aunque el campo del rendering no fotorrealista existe desde comienzos de la Computación Gráfica, solamente en la década de los '90 se vio una explosión en el interés de numerosos investigadores, debido a que recién en esos años los procesadores fueron capaces de llevar a cabo los cálculos necesarios para poder generar imágenes estilizadas en un tiempo razonable. Estos cálculos suelen tomar mucho más tiempo que los asociados al rendering tradicional, debido al post-proceso extenso de los objetos que se están graficando. En la actualidad, en las conferencias y congresos de Computación Gráfica y Animación por Computador siempre hay publicaciones dedicadas a NPR; no obstante, no existen herramientas que permitan trabajar con NPR de una forma generalizada. Esta situación ha motivado la proposición de un marco de trabajo para NPR, construido en base a técnicas y conceptos del rendering tradicional, que integre e implemente las técnicas y conceptos necesarios para poder desarrollar técnicas de NPR.

1.1. Clasificación de Técnicas NPR

NPR puede ser muy útil a la hora de crear y visualizar contenidos en tres dimensiones, ya que muchas veces la gráfica fotorrealista no comunica en su completitud el mensaje que se desea entregar o no muestra explícitamente la información que se requiere. Existen dos grandes áreas en NPR: la extracción y graficación de líneas y contornos a partir de una superficie, y la graficación abstracta y/o estilizada de los objetos.

1.1.1. Líneas y Contornos

Los dibujos que consisten solamente en líneas o trazos suelen transmitir fielmente al observador las formas más importantes de la figura que se está observando, y permiten indicar zonas de interés o extraer información implícita en los objetos. En esta área de NPR se busca identificar, a partir de una superficie, las curvas que permitan marcar zonas visuales de interés. Debido a ese propósito se utilizan conceptos de geometría diferencial, ya que la extracción de líneas debe definirse formalmente a partir de las propiedades geométricas de la superficie con la que se está tratando.

En esta memoria se trabaja con modelos tridimensionales poligonales, por lo que se requiere una forma de aproximar una superficie diferenciable con una malla de triángulos, junto con algoritmos para tratar las propiedades geométricas de esas mallas. De ese modo se pueden responder las dos preguntas esenciales de esta área de NPR: **¿qué extraer?** y **¿cómo extraerlo?**

Por ejemplo, para la ilustración técnica, muchas veces se desea dejar de lado el fotorrealismo y apreciar el objeto de forma simplificada, resaltando los bordes de interés que están frente al observador y también detrás del objeto. La Figura 1.2, generada por el proyecto *FreeStyle* [GTDS04], muestra este tipo de aplicación.

1.1.2. Graficación Abstracta y/o Estilizada

La segunda área es la graficación abstracta y/o estilizada, que busca representar un objeto o una escena de una forma que acentúe o destaque ciertas cualidades de los objetos. Para realizar la graficación se pueden utilizar todo tipo de técnicas que no necesariamente dependen de las propiedades de los objetos, sino más bien del estilo de graficación que se desea obtener. Por ejemplo, en la Figura 1.3 se muestra una escena con tres personajes graficada de manera estilizada. Se observa que la estilización afecta a todos los elementos de la escena de modo similar sin importar las propiedades de los objetos que la componen: los colores utilizados no son típicos de una escena real, y las intensidades de iluminación no siguen el patrón difuso de la realidad, sino que son graficadas utilizando una leve técnica de achurado. Los rasgos de los personajes tampoco son fotorrealistas, lo cual se complementa con la graficación estilizada.

En el caso de la visualización científica, se puede querer destacar algunas áreas críticas del objeto que se está graficando. Dichas áreas se pueden representar con intensidades de color que no correspondan a un modelo de iluminación fotorrealista, pero sí cumplirían con la premisa de llevar la atención del observador a esas áreas. Para la visualización geográfica, se suele combinar una graficación estilizada de un terreno en conjunto con la extracción de líneas de interés.

Otra aplicación en esta área del NPR es la imitación de medios tradicionales, es decir, la graficación de una escena tridimensional de modo que parezca una obra realizada con acuarela, pintura (graficación mediante pinceladas), lápiz grafito, dibujo achurado, carboncillo, etc. Existen muchas técnicas en la literatura, aunque no todas son simulaciones en estricto rigor, debido a lo costoso que puede resultar, y a que la imitación en base a heurísticas suele lograr resultados convincentes.

1.2. El Problema a Resolver y la Solución Propuesta

A pesar de que existe una gran cantidad de conocimiento disponible en las publicaciones electrónicas, en las conferencias y en la literatura de Computación Gráfica, no hay disponibilidad de herramientas para trabajar con NPR, por lo que quien desee implementar una técnica debe hacerlo desde cero. Sí existen algunas implementaciones de técnicas en particular, pero son implementaciones de demostración que difícilmente sirven como base para un trabajo más extenso.

Para solventar la falta de herramientas para NPR en general, se propone como tema de esta memoria el diseño e implementación de un marco de trabajo para NPR en tiempo real, es decir, para aplicaciones que generan sus imágenes al momento de ejecutarse. Este marco de trabajo debiese permitir a sus usuarios crear aplicaciones gráficas teniendo ya una base con estructuras de datos, algoritmos, graficadores tradicionales y NPR que se encuentren implementados, las cuales pueden ser utilizadas directamente o extendidas para satisfacer necesidades específicas.

Considerando esto, se vuelve necesario el estructurar el marco de trabajo como si fuera una biblioteca gráfica de uso general, de la cual el principal uso es desarrollar y utilizar técnicas de NPR. Así, el núcleo del marco de trabajo presentado se puede utilizar en cualquier tipo de

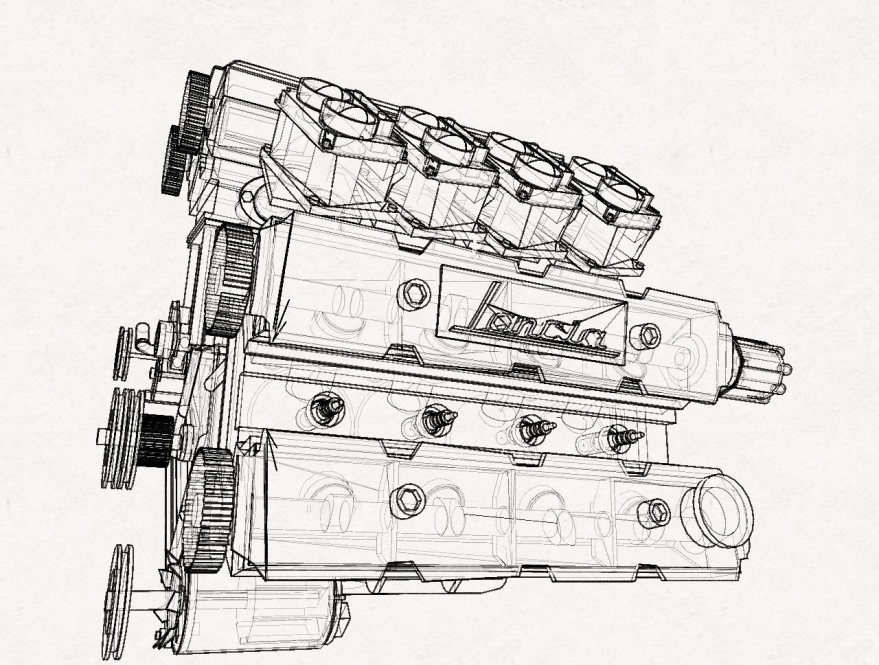


Figura 1.2: *Un modelo tridimensional de motor graficado como ilustración técnica, acentuando sus bordes. Se grafican también los bordes de interés que se encuentran ocultos al observador. Fuente: [GTDS04].*

Figura 1.3: *Imagen de un videojuego actual, "Valkyrie of the Battlefield", de la consola PlayStation 3. Los personajes no tienen rasgos fotorrealistas, los colores son similares a los de la ilustración infantil, y el sombreado de los objetos y personajes no sigue un patrón fotorrealista, incluso tiene detalles que asemejan un pintado a mano alzada. Fuente: Revista Game Watch de Japón.*

aplicación de Computación Gráfica que trabaje con modelos tridimensionales, sin importar si será utilizado para NPR o no. La justificación de este enfoque para abordar el problema se encuentra en los siguientes puntos:

- Gran parte de las técnicas de NPR requieren el conocimiento y entendimiento del rendering tradicional, y a su vez utilizan herramientas y conceptos del rendering tradicional. En la literatura no siempre queda en evidencia qué es lo necesario para implementar este tipo de técnicas. Lo cierto es que muchas de ellas se pueden implementar utilizando técnicas fotorrealistas, por lo que para implementar un marco de trabajo de NPR se puede tener como base el rendering tradicional.
- Se desea implementar el proceso en tiempo real, por lo que se debe utilizar un proceso similar o que extienda el proceso tradicional. Algunas técnicas de NPR, en particular las de simulación de medios, producen muy buenos resultados pero su implementación es demasiado costosa para ser aplicada en tiempo real. Sin embargo, si se consideran técnicas que se implementan en base al rendering tradicional, solamente hay que tener cuidado de no sobrecargar el desempeño al agregar las extensiones necesarias al proceso de rendering tradicional.

El lenguaje elegido para la implementación es `C++`[S+97] por ser de alto nivel y soportar orientación a objetos, facilitando un diseño generalizado y extensible; ser compilado y de alto rendimiento, ya que se busca desempeño en tiempo real. La metodología elegida, en base a lo planteado, es buscar componentes genéricas que sean comunes a los distintos temas, algoritmos y técnicas que se estudiarán en los siguientes Capítulos, sobre las cuales un usuario del marco de trabajo pueda construir aplicaciones especializadas de acuerdo a sus necesidades.

Para demostrar el uso del marco de trabajo, se propone la inclusión de una aplicación de prueba que grafica modelos tridimensionales con distintos estilos gráficos realistas y no fotorrealistas.

1.2.1. Objetivos Generales

Habiendo definido entonces el problema a resolver, se procede a enunciar los objetivos detrás de esta memoria. Los objetivos generales son:

- Diseñar e implementar toda la funcionalidad necesaria que permita desarrollar técnicas no fotorrealistas para graficar un modelo tridimensional dado.
- El marco de trabajo debe poder ser usado en la mayor cantidad de sistemas operativos y configuraciones posibles, es decir, debe ser *portable*.
- Proveer una aplicación de prueba que permita observar a grandes rasgos las capacidades de desempeño y extensión del marco de trabajo.

1.2.2. Objetivos Específicos

Los siguientes objetivos desglosan en puntos específicos los objetivos generales. Por parte del marco de trabajo se considera:

- Diseñar e implementar una estructura de datos que permita trabajar con modelos tridimensionales. Se debe implementar una estructura de datos flexible para representar distintos tipos de modelos tridimensionales, que a la vez permita la ejecución óptima de algoritmos geométricos y gráficos.
- Diseñar e implementar toda la funcionalidad necesaria para graficar modelos tridimensionales de manera tradicional.
- Proveer las herramientas para extraer líneas de interés desde de un modelo tridimensional.
- Proveer las herramientas que faciliten la definición de un tipo de línea de interés arbitraria, con el fin de implementar tipos de líneas que no se encuentren implementadas en el marco de trabajo.
- Proveer las herramientas para desarrollar estilos de NPR para el graficado estilizado y/o abstracto de un modelo tridimensional. Debe existir una base sobre la cual se puedan añadir las características propias de cada estilo, sin necesidad de re-implementar conceptos comunes a todos ellos.
- Proveer las herramientas para trabajar con los procesadores gráficos. Para el desarrollador es importante tener una conexión entre sus propias estructuras de datos, en este caso las del marco de trabajo, y las propias de la GPU.
- El marco de trabajo debe ser diseñado considerando orientación a objetos, de modo de ser extensible y reutilizable.
- El marco de trabajo debe utilizar bibliotecas gráficas y utilitarias estándares, con disponibilidad para múltiples sistemas operativos y de código abierto (o compatibles con licencias de código abierto).

Para la aplicación de prueba, los objetivos son los siguientes:

- Cargar diversos modelos tridimensionales y graficarlos con estilos gráficos realistas y no realistas variados.
- Permitir variar los parámetros con los cuales se grafican los modelos.
- Ser una demostración del uso del marco de trabajo en aplicaciones gráficas.

1.3. Contenido de la Memoria

Esta memoria está compuesta de tres partes. La primera está conformada por cuatro Capítulos que contienen los antecedentes necesarios para entender el trabajo propuesto y su implementación, incluyendo referencias a fuentes de información relevante y actual. Sus Capítulos son los siguientes:

Capítulo 2, Consideraciones Preliminares: Se especifican los usuarios del marco de trabajo y los requisitos que guiaron el trabajo realizado en la memoria.

Capítulo 3, Conceptos de Computación Gráfica: Se definen los conceptos gráficos utilizados en la memoria, en particular los correspondientes a representación de modelos 3D, rendering en tiempo real y curvas paramétricas, entre otros.

Capítulo 4, Conceptos Geométricos: Se definen los conceptos geométricos necesarios para abordar la teoría e implementación de NPR. Estos conceptos corresponden a geometría diferencial y la extracción de curvas de nivel.

Capítulo 5, Rendering No Fotorrealista – NPR: Se describen técnicas de NPR, comentando sus implementaciones y aplicaciones. También se incluye una breve reseña histórica sobre la abstracción.

La siguiente parte, de diseño e implementación, contiene el trabajo realizado. Se compone de un único Capítulo:

Capítulo 6, Diseño e Implementación: Se muestra el detalle del trabajo realizado, desglosado en componentes.

La última parte consiste en la exhibición y discusión de los resultados obtenidos:

Capítulo 7, Resultados: Se muestran los resultados visuales obtenidos con el marco trabajo a través de la graficación de varios modelos tridimensionales con distintas características. Se realiza un análisis de desempeño en tiempo real.

Capítulo 8, Discusión y Conclusiones: Se discuten las decisiones tomadas al llevar a cabo el trabajo realizado, así como un posible futuro del marco de trabajo. Se enumeran las conclusiones de esta memoria.

Además, se adjuntan dos Apéndices:

Apéndice A, Graficadores en Cg: Se muestra el código fuente de los graficadores implementados que utilizan la GPU.

Apéndice B, Extracción y Graficación de Líneas: Se muestra el código fuente de la operación de extracción y graficación de dos tipos de líneas de interés en un modelo tridimensional.

Capítulo 2

Consideraciones Preliminares

Este Capítulo resume las bases sobre las cuales se construyeron el diseño y la implementación del marco de trabajo. Lo constituyen las siguientes Secciones:

Sección 2.1, Definición de Marco de Trabajo: Se define formalmente el concepto "marco de trabajo" en el contexto del desarrollo de software.

Sección 2.2, Identificación de Usuarios: Se identifican dos tipos de usuario que guían la especificación de requerimientos para el marco de trabajo. Estos dos usuarios son el desarrollador de aplicaciones gráficas y el investigador de técnicas NPR.

Sección 2.3, Requerimientos: Se enumeran los requerimientos extraídos a partir de los objetivos de la memoria, de los tipos de usuario identificados y de las técnicas NPR mencionadas.

2.1. Definición de Marco de Trabajo

En el Capítulo anterior se habló de *marco de trabajo* pero no se realizó una definición formal de lo que es verdaderamente un marco de trabajo o *framework*. De acuerdo a Wikipedia[1], el concepto "marco de trabajo" se define como:

"*Un marco de trabajo de software es un diseño reusable para un sistema (o subsistema) de software. Se expresa como un conjunto de clases abstractas y a través de la forma en la que deben interactuar dichas clases. Un marco de trabajo puede estar diseñado mediante orientación a objetos. Aunque un marco de trabajo no necesariamente puede ser implementado en un lenguaje orientado a objetos, usualmente sí son implementados en ellos, por lo que son considerados como el equivalente orientado a objetos de las bibliotecas de software. Un marco de trabajo puede incluir programas de apoyo, bibliotecas de código, lenguajes interpretados, u otro software que ayude a desarrollar y conectar las diferentes componentes de un proyecto de software. Las diversas partes del marco de trabajo pueden ser expuestas a través de una API (*Application Programming Interface*).*

[1] `http://en.wikipedia.org/wiki/Software_framework`

2.2. Identificación de Usuarios

Todo marco de trabajo debe tener claro quiénes serán sus usuarios; en el caso del marco propuesto en esta memoria, se distinguen dos: aquel que desea desarrollar una aplicación gráfica, utilizando los recursos que ya provee el marco de trabajo, y aquel que desea desarrollar técnicas de NPR[2]. Se muestran los casos de uso para estos usuarios en la Figura 2.1, y a continuación se describe con más detalle el rol específico de cada usuario para el marco de trabajo.

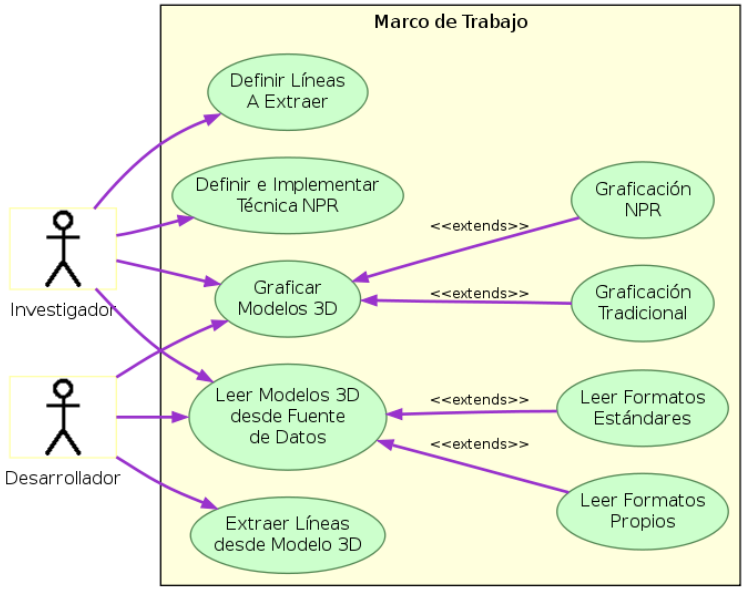


Figura 2.1: *Casos de uso para los usuarios del marco de trabajo.*

2.2.1. Desarrollador de Aplicación Gráfica

Por desarrollador de aplicación gráfica se entiende un usuario que tiene datos que desea visualizar de alguna manera, es decir, quiere que sus datos pasen por un proceso cuyo resultado sea una imagen. En este caso, esos datos son la especificación de un modelo tridimensional, y la imagen es la generada por un proceso de rendering, que bien puede ser realista o NPR.

Las siguientes son algunas de las dificultades y tareas con las que se puede encontrar este tipo de usuario al desarrollar una aplicación gráfica. Para cada dificultad se indica la solución propuesta por el marco de trabajo:

Estructuras de datos para las mallas de triángulos Ya que necesita utilizar una biblioteca gráfica que probablemente trabaje con mallas de triángulos, tendrá que escribir sus propias estructuras que se acomoden a los datos que posee. Entre los desafíos que plantea este punto está la implementación de una estructura que sea óptima con el

[2]Eventualmente podría existir un tercer tipo de usuario que mezcla a los dos usuarios mencionados, pero no es necesario analizarlo.

uso de memoria, que presente buen desempeño y no tenga problemas de programación (como fugas de memoria o acceso a punteros nulos), lo que toma mucho tiempo para programar y verificar. El marco de trabajo debe proveer estas estructuras.

Lectura de modelos y escenas Además de tener las estructuras de datos necesarias, se requiere alguna manera de poblarlas a partir de modelos y escenas tridimensionales almacenadas en disco o desde otra fuente de datos. El marco de trabajo debe proveer mecanismos para poder leer los datos desde la fuente y llevarlos a las estructuras de datos ya implementadas.

Extracción de Líneas El desarrollador quiere extraer líneas de interés a partir de los modelos que posee. El marco de trabajo debe

Graficación Tradicional y NPR El desarrollador quiere graficar los datos (tanto de un modo tradicional como NPR), y probablemente requiera que la graficación de ellos sea lo más directa posible. El marco de trabajo debe incluir graficadores tradicionales y NPR, por lo que solamente se requiere variar sus parámetros y elegir el más adecuado de acuerdo a los datos que se poseen.

Recursos y parámetros El desarrollador usualmente debe preocuparse de las complicaciones de cargar texturas u otros recursos, así como de los formatos especiales de los parámetros de la biblioteca gráfica.

La complejidad de estas dificultades depende de la aplicación que se esté desarrollando, pero independientemente de esa complejidad, se vuelve necesaria una herramienta que simplifique las tareas a realizar. El desarrollador debe tener acceso a un marco de trabajo que contenga componentes que tomen el control de la aplicación, de modo que solamente deba implementar las características específicas de su aplicación.

A partir del diagrama de casos de uso, y de las tareas enunciadas para este usuario, se puede concluir que el desarrollador de aplicaciones gráfica tiene un uso práctico del marco de trabajo, lo que quiere decir que lo utiliza como base para la aplicación que está construyendo. Lo que hace el desarrollador es crear las componentes específicas que requiere, las cuales conecta al marco de trabajo con el fin de cumplir sus tareas.

Por ejemplo, este tipo de usuario suele tener sus datos en un formato propietario que puede no ser reconocido por el marco de trabajo. En ese caso, se debe desarrollar un cargador para ese formato, de modo que el cargador conecte la fuente de datos con las estructuras implementadas.

Se puede asumir que el desarrollador ya sabe como quiere graficar los datos que posee. Conoce cuales tipos de línea extrae el marco de trabajo, así como los graficadores (NPR y tradicionales) que incluye. Simplemente los utiliza directamente ya que las estructuras de datos del marco de trabajo ya contienen sus datos.

2.2.2. Investigador de Técnicas NPR

El otro usuario específico del marco de trabajo es aquel investigador que desea implementar técnicas de NPR. Ante esto, es posible que el investigador tenga un conjunto de datos

de prueba, y que su mayor preocupación sea el poder implementar sus ideas sin preocuparse de detalles de programación que no tengan relación directa con la técnica que desea implementar, o que sean redundantes con técnicas ya implementadas.

Las tareas que puede enfrentar este tipo de usuario son las siguientes:

Lectura de modelos y escenas Como se describió anteriormente, este tipo de usuario suele tener un conjunto de datos determinado para probar sus técnicas. Estos datos consisten en modelos y escenas tridimensionales en formatos que son prácticamente estándares de facto (por su uso extendido), por lo que el marco de trabajo debe permitir la lectura inmediata de ellos sin necesidad que el usuario implemente la carga de datos.

Definición de líneas de interés El usuario quiere implementar la definición de un nuevo tipo de línea de interés. Para facilitar esta tarea, el marco de trabajo debe proveer un algoritmo de extracción de líneas de interés, así como entregar una interfaz que permita definir formalmente la línea de interés que desea extraer.

Graficación de nuevos estilos NPR No solamente se requiere la graficación tradicional de los modelos, o de las técnicas ya implementadas, también puede ser que el investigador desee implementar una nueva técnica, por lo que requiere que dicha implementación sea lo más directa posible. Esto quiere decir que el investigador debiese contar con una base sobre la cual añadir las características del estilo que quiere plantear, en vez de re-implementar características propias de las técnicas ya disponibles.

Un marco de trabajo que simplifique estas tareas permitirá que el investigador se preocupe directamente de la implementación de su tema de investigación.

2.3. Requerimientos

En base al estudio de los tipos de usuario y de los objetivos de esta memoria, se han definido los siguientes requerimientos, que se han desglosado en cuatro áreas: Rendering en Tiempo Real, o requerimientos que tienen relación con el desempeño de la aplicación; Rendering Tradicional y NPR, o requerimientos que apuntan a características específicas del proceso de generar una imagen; Generalidad, requerimientos de índole general para todo el marco de trabajo; y Extensibilidad, requerimientos específicos para las características que se desean hacer extensibles fácilmente.

2.3.1. Rendering en Tiempo Real

Estos requerimientos se relacionan con el desempeño del marco de trabajo. En particular, se requiere lo siguiente:

Cálculos Eficientes Los cálculos que se realicen en los algoritmos implementados deben ser lo más eficiente posible. En ocasiones será necesario sacrificar precisión con el fin de obtener un desempeño más alto.

Uso Óptimo de Memoria La memoria en la que son almacenadas las estructuras de datos debe estar organizada de modo que administrarla sea sencillo y que recorrerla en busca de datos sea óptimo en rendimiento. Se deben evitar las redundancias a menos que ayuden a mejorar el desempeño.

Biblioteca Gráfica Se debe implementar una interfaz de alto nivel para la biblioteca gráfica.

Uso de la GPU Se debe implementar una interfaz de alto nivel para conectar las estructuras de datos de la GPU con las estructuras de datos del marco de trabajo, para ser utilizada en la graficación de los modelos tridimensionales.

Recursos gráficos Se debe facilitar la carga de recursos gráficos, en particular las texturas.

2.3.2. Rendering Tradicional y NPR

Estos requerimientos se relacionan con las capacidades del marco de trabajo y los procesos de rendering tradicional y NPR:

Lectura de Datos Se debe proveer una interfaz de lectura que datos que permita leer datos de modelos tridimensionales en formatos estándares.

Mallas de Triángulos Se deben proveer las estructuras de datos necesarias para representar modelos tridimensionales en base a triángulos.

Extracción y Definición de Contornos Se deben proveer los algoritmos necesarios para, dada la definición de una línea de interés, extraer las líneas que cumplen la definición a partir de la malla de triángulos del modelo.

Graficación Tradicional Se deben proveer graficadores de rendering tradicional para graficar los modelos cargados en el marco de trabajo.

Graficación de Líneas Se deben proveer graficadores que, tomando un conjunto de líneas extraídas desde un modelo, dibujen esas líneas en pantalla.

Graficación Abstracta Se deben proveer graficadores que, dado un modelo tridimensional, lo grafiquen con técnicas NPR de tipo abstracto o estilizado.

2.3.3. Generalidad

Estos requerimientos se refieren a la forma de implementar y trabajar en el marco de trabajo:

Estructuras flexibles Las estructuras de datos deben ser lo más flexible posible, en cuanto si se desea agregar alguna característica, los cambios en las aplicaciones que utilicen el marco de trabajo deban ser mínimos a menos que deseen acceder a las nuevas características.

Componentes Genéricas Se deben tener componentes genéricas, en conformidad con lo visto en los Capítulos anteriores.

Uso de herramientas multiplataforma y de código abierto Solamente se deben utilizar bibliotecas de código abierto y de licencia GPL (General Public License), o compatibles con ella. Asimismo, solamente se utilizarán herramientas multiplataforma.

Uso de código externo En caso de utilizar código ajeno, en las siguientes versiones del marco de trabajo se debe intentar realizar implementaciones propias.

2.3.4. Extensibilidad

Estos requerimientos se refieren a las extensiones específicas que se espera que realicen los desarrolladores en el marco de trabajo:

Adición de nuevas fuentes de datos Se deben proveer las herramientas para que la adición de nuevas fuentes de datos (sean a partir de archivos, de input de usuario o procedurales) sea directa y sin redundancias.

Creación de graficadores y estilos gráficos Se deben proveer las herramientas para la creación directa y sin redundancias en código de nuevos estilos gráficos y de graficadores que reemplacen el *fixed pipeline*.

Creación de nuevos tipos de contornos Se deben proveer las herramientas para la especificación y extracción de nuevos tipos de contornos sin tener que modificar los algoritmos de extracción ya presentes en el marco de trabajo.

Patrón de Estrategia para los algoritmos Se sugiere que todos los algoritmos parametrizables implementados en el marco de trabajo sigan un patrón *Strategy*, con el fin de permitir al desarrollador utilizar sus propias versiones de los procedimientos.

Capítulo 3

Conceptos de Computación Gráfica

Este Capítulo busca definir y explicar algunos conceptos claves de Computación Gráfica para poder diseñar, implementar y entender una implementación de NPR. Contiene las siguientes secciones:

Sección 3.1, Representación de Modelos 3D: Se estudia la representación de objetos en tres dimensiones, tanto estáticos como objetos con animaciones en el tiempo, utilizando mallas de triángulos.

Sección 3.2, Rendering en tiempo real: Se estudia, con un grado medio de detalle, el proceso de representación gráfica de la información en tres dimensiones estudiada en la Sección 3.1, con énfasis en el tiempo real, es decir, en la gráfica que se genera en el momento de ser visualizada.

Sección 3.3, Luces y Modelo de Iluminación de Phong: Se estudia la representación de las fuentes de luz y un modelo heurístico de iluminación que presenta resultados que buscan ser realistas, llamado *modelo de iluminación de Phong*. Este modelo es el más utilizado en computación gráfica en tiempo real.

Sección 3.4, Técnicas de Sombreado: Se estudian las técnicas de sombreado, definidas como los métodos y técnicas utilizadas para evaluar un modelo de iluminación determinado en una malla de triángulos.

Sección 3.5, Interpolaciones y Curvas Paramétricas: Se estudian, desde el punto de vista gráfico, técnicas de reconstrucción de información a partir de datos conocidos. Se incluye la definición de dos curvas paramétricas.

Sección 3.6, OpenGL: Se describe la biblioteca gráfica utilizada para implementar los conceptos estudiados en este Capítulo y los siguientes.

Sección 3.7, GPUs – Unidades de Procesamiento Gráfico: Se describen brevemente las capacidades de los procesadores gráficos actuales. Se analiza el lenguaje de alto nivel Cg, que permite implementar algoritmos y técnicas gráficas directamente en estos procesadores.

3.1. Representación de Modelos 3D

Figura 3.1: *La tetera de Utah representada como una malla de triángulos.*

A lo largo de las páginas de esta memoria, se ha hablado de modelos y escenas tridimensionales sin especificar realmente de lo que se está hablando, pues se da por entendido que el lector tiene una noción de lo que es una escena y de lo que es un modelo tridimensional. Sin embargo, es necesario dar una definición, quizás no totalmente formal, de qué es un modelo y una escena tridimensional. En general, una escena está compuesta de diversos elementos: entre ellos los objetos o modelos 3D, pero también tiene una gran cantidad de atributos, como pueden ser otros elementos visuales, entre ellos las fuentes de luz, e incluso elementos abstractos o no tangibles como la temperatura, la hora o el viento.

De todos esos elementos, los que son de interés para este trabajo son los *visuales*, en particular los modelos tridimensionales y sus propiedades gráficas. En esta Sección se busca definir lo que es un modelo tridimensional y su representación computacional.

3.1.1. Definición de Modelo 3D

En la vida real, los objetos, en su mayoría, son sólidos, con un volumen y densidad determinada. De todos los objetos que existen se puede extraer la siguiente clasificación:

Objeto con cáscara o carcasa e interior Un pedazo de madera tiene una carcasa o cáscara que es visible y palpable, pero también tiene un interior, es un objeto *sólido*.

Objeto solamente con cáscara o carcasa Una tetera no tiene interior, más bien su interior es llenado con otra cosa, en general agua. Este tipo de objetos tiene volumen, pero en su interior es *hueco*. En la Figura 3.1 se aprecia el modelado 3D de la *tetera de Utah*.

Objeto sin cáscara o carcasa, pero con interior difuso El humo indudablemente tiene volumen, pero no tiene cáscara ni una superficie que permita determinar sus bordes. Estos objetos no son sólidos ni huecos, más bien son *híbridos*.

Visualmente, las dos primeras clasificaciones son muy similares, ya que a priori es imposible determinar sólo con la vista si un objeto tiene interior o no. Es decir, basta solamente con graficar la superficie que define la cáscara de un objeto para representarlo visualmente de manera efectiva o reconocible.

En Computación Gráfica en general se trabaja con superficies en tres dimensiones, es decir, los objetos pertenecientes a las dos primeras clasificaciones se pueden graficar directamente a partir de su representación computacional[1]. En base a ello, se define un **modelo tridimensional o 3D**, como una *superficie en tres dimensiones representada computacionalmente*.

Ahora bien, de manera analítica, una superficie S se puede representar de tres formas:

Representación Explícita La superficie S se expresa como una función F que al ser evaluada en un dominio entrega los puntos que la componen: $S = F(x, y, z)$.

Representación Implícita La superficie S se expresa como el lugar geométrico que cumple $F(x, y, z) = 0$.

Representación Paramétrica La superficie se denomina $S(s, t)$, donde (s, t) es un punto de un espacio de dos dimensiones. Es común ver un modelo tridimensional que une dos o más superficies paramétricas, en este caso llamadas *parches*. A cada parche le corresponde un dominio (s, t) distinto.

Las dos primeras representaciones, si bien son convenientes en términos análiticos, presentan problemas para representar superficies que no siguen un patrón matemático. Debido a esto la tercera forma es común en el modelamiento de objetos tridimensionales arbitrarios en las aplicaciones CAD y de modelamiento 3D.

Independientemente de la forma en la cual se exprese una superficie, graficar la representación es un problema aparte. Evaluar una función, sea implícita o explícita, es costoso, y también lo es la evaluación de superficies parametrizadas. Entonces, ¿cómo representar un modelo tridimensional, de forma arbitraria y probablemente compleja, de una manera que lo represente fielmente pero que además facilite la graficación en tiempo real? La respuesta a esta interrogante son las *mallas de triángulos*.

3.1.2. Mallas de Triángulos para Computación Gráfica

Las mallas de triángulos permiten representar modelos tridimensionales de superficies suaves (como una esfera) como de superficies que no son suaves (como un poliedro). En el contexto de esta memoria, una malla de triángulos es la *triangulación de una superficie*, es

[1]La tercera clasificación no puede ser representada con una superficie, pero existen técnicas y aproximaciones para hacerlo.

decir, la *discretización* de una superficie en un conjunto de triángulos que cumplen condiciones de coherencia:

- Las aristas de dos triángulos sólo se pueden intersectar en sus extremos (no pueden cruzarse).
- Una arista sólo puede ser compartida por dos triángulos.
- No puede haber triángulos sueltos, a menos que la malla sea compuesta de un único triángulo.

En Computación Gráfica estas condiciones no siempre son necesarias, ya que en la mayoría de los casos sólo se trabaja con resultados visuales donde cada triángulo se procesa independientemente de los demás. No obstante, la coherencia de los datos ayuda a representarlos mejor y a almacenarlos eficientemente en memoria, además de ser necesaria para aplicar algoritmos geométricos.

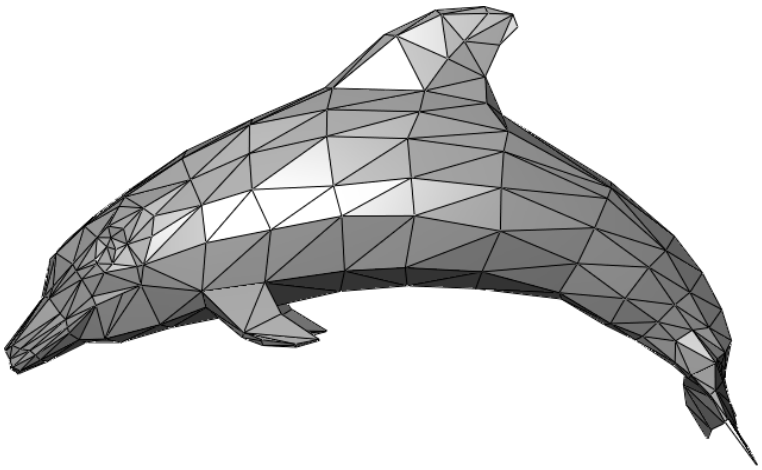


Figura 3.2: *Una malla de triángulos que modela a un delfín.*

La Figura 3.2 muestra un delfín representado con una malla gruesa. Se dice que una malla es gruesa cuando un triángulo representa una gran porción del área de la superficie, posiblemente perdiendo información debido a la aproximación de la malla, aunque de igual modo se puede percibir la forma de la superficie.

Una pregunta que surge ante la necesidad de representar mallas de triángulos es: ¿qué estructuras de datos se necesitan? Una primera respuesta es *una lista de vértices* donde el orden de ellos define los triángulos. Si solamente se desea graficar el conjunto de triángulos no es necesaria más información, pero si se desean ejecutar algoritmos sobre la malla probablemente se requerirá *información estructural* de la malla, es decir, información de vecindad entre los triángulos y otras propiedades que puedan ser de interés.

A continuación se define la información necesaria en una estructura de datos que represente un modelo tridimensional.

Vértices: Posición y Vector Normal

En el contexto de una malla de triángulos que discretiza una superficie, un triángulo es una parte de ella encerrada entre tres vértices. Cada vértice tiene una **posición en tres dimensiones**, y un **vector normal** que nos permite conocer hacia dónde está orientada la superficie en esa posición.

Colores

Si bien existen numerosos *espacios de color*, un color se suele representar en el *espacio o modelo de color RGB*. Este modelo representa los colores como una mezcla *aditiva* de componentes rojo (R), verde (G) y azul (B). Así, el negro se representa como la ausencia de los tres colores y el blanco como la presencia total de los tres colores principales. En la Figura 3.3 se observa la descomposición de una imagen en las tres componentes mencionadas.

Figura 3.3: *Una imagen con su descomposición en tres sub-imágenes, una para cada componente del espacio RGB.*

Aunque el espacio RGB no puede representar todos los colores posibles, a lo que se suma la imposibilidad de representar todos los números reales, es el más utilizado debido a que las pantallas CRT y LCD lo utilizan para desplegar imágenes. Otros modelos conocidos con CMYK (utilizado en las impresoras), YUV (utilizado en vídeo), entre otros.

Propiedades de Material

Aunque con la posición y el vector normal pueden bastar para una representación básica de la superficie, se necesitan atributos gráficos que indiquen el modo de asignarles color. Debido a esto, en cada vértice se la malla se indican las **propiedades de material** de la superficie, es decir, los **colores** de la luz que se reflejan en el objeto. Si no se considera interacción con alguna fuente de luz, simplemente se especifica el color con el que se quiere graficar la superficie.

Por ejemplo, una bola metálica de color azulado tiene distintas propiedades reflectivas que una bola de cuero color café. Ambas tienen colores definidos, pero reaccionan de manera distinta ante la luz: la bola metálica refleja de un modo mucho más claro la luz, mientras que la bola de cuero lo hace de un modo mucho más difuminado. Este tipo de comportamiento se especifica mediante las propiedades de material, que indican el color y la intensidad con que se refleja cada color ante cada componente de la luz.

Texturas y Coordenadas de Textura

Si bien los detalles de rugosidad de una superficie se pueden representar en la malla de triángulos, esto puede significar que la malla sea más fina de lo necesario, con el consiguiente costo computacional en memoria y en procesamiento. Una forma de solucionar este problema son las **texturas**, que a grandes rasgos son imágenes que se superponen en la superficie al momento de graficarla. Esta técnica es llamada **mapeado de texturas** (*texture mapping* en inglés) y se puede ver en la Figura 3.4.

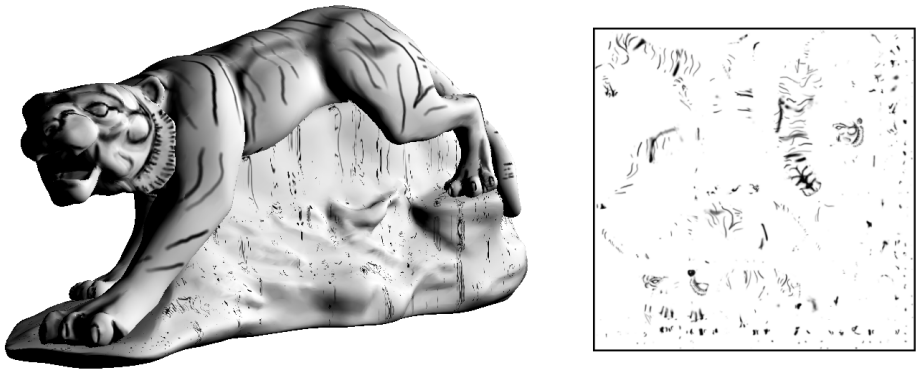


Figura 3.4: *Un modelo tridimensional de un tigre, a la izquierda, con mapeado de texturas. La textura aplicada se observa a la derecha.*

El mapeado de texturas no es una técnica automática, es decir, a partir de una imagen no se puede determinar automáticamente cómo aplicarla en la superficie. Debido a ello, cada vértice de la malla debe indicar **coordenadas de textura**. Una coordenada de textura es una posición dentro del sistema de coordenadas de la textura correspondiente (que puede ser 1D, 2D o 3D), de este modo las tres coordenadas de textura de un triángulo definen un espacio dentro de la imagen que será superpuesto en la graficación del triángulo. Ya que el área de ese espacio no es necesariamente la misma del triángulo, se vuelve necesaria una técnica que permita estirar o encoger la imagen. También cómo interactuará la textura con la información de color y material del modelo, si es que existe: la textura puede reemplazar los colores o bien se pueden mezclar los colores especificados y los correspondientes a la textura.

Información Estructural y de Vecindad

Con todo lo indicado anteriormente es posible graficar una malla de triángulos. Ahora bien, si se desea aplicar algoritmos o realizar operaciones en la superficie, probablemente se requiera información sobre la estructura de la malla.

Algunas operaciones que pueden ser requeridas por la aplicación que se esté construyendo son las siguientes:

- Determinar los dos triángulos que comparten una arista (o equivalentemente, determinar el triángulo vecino a través de una arista).
- Determinar los triángulos que comparten un vértice.

Una estructura de datos para mallas de triángulos que permita ejecutar algoritmos sobre la malla debe proveer estas asociaciones. Esta estructura de datos permite implementar un gran número de operaciones en un modelo, tanto las operaciones gráficas que se verán a continuación en este Capítulo como las operaciones geométricas mencionadas en el Capítulo 4.

Una estructura bastante probada y estudiada que permite estas operaciones es *Winged Edge*[Bau72], aunque es difícil de implementar y no es lo suficientemente adecuada para operaciones gráficas, ya que el elemento principal de la estructura son las aristas y en Computación Gráfica se trabaja principalmente con los vértices.

3.1.3. Animaciones y Deformaciones

Los datos necesarios que se han mencionado parecen ser suficientes para representar y graficar objetos estáticos. Pero ¿qué sucede con los modelos que tienen animaciones o deformaciones de su geometría a través del tiempo?¿Sirve el enfoque seguido hasta este punto?

Lo primero que se debe realizar es analizar las formas de animar o deformar un modelo tridimensional. En la actualidad, existen dos formas de hacerlo. La primera, llamada **Vertex Animation**, tiene ciertas similitudes con la animación tradicional, ya que utiliza cuadros de animación para la malla. La segunda, **Vertex Blending**, trata a los modelos como un esqueleto cuyo movimiento es definido mediante transformaciones y deformaciones en lugares claves del esqueleto.

Vertex Animation

En la animación tradicional, la animación de un objeto se produce superponiendo dibujos que lo representan en distintos instantes de tiempo. Cada uno de esos dibujos es llamado un *cuadro* (*frame* en inglés) en la animación. Éstos, al ser superpuestos rápidamente, generan la sensación de animación en el dibujo.

En el caso de un modelo tridimensional, se suele tener la información del modelo en distintos instantes de tiempo, de modo de poder superponer cada cuadro de la animación que le corresponde. Se define que la estructura de la malla es siempre la misma; lo que se deforma o cambia a través del tiempo son las posiciones de los vértices y sus vectores normales. En la Figura 3.5 se observan los cuadros de la animación de un modelo tridimensional que representa a un personaje que está corriendo.

Cada uno de los cuadros definidos para la malla es llamado **keyframe**. A diferencia de la animación tradicional, al tener la información de la estructura de la malla y no solamente una imagen de ella, es posible reconstruir el estado de la malla en el tiempo entre un keyframe y otro. En la Figura 3.5 los cuatro cuadros pueden ser insuficientes para representar fielmente la animación del personaje corriendo, por lo que se reconstruyen numerosos cuadros auxiliares entre los *keyframes* utilizando técnicas de interpolación. Esto es factible de realizar porque, al imponerse que la estructura de la malla deba ser siempre la misma, sólo es necesario interpolar posiciones y vectores.

De este modo, para almacenar un modelo tridimensional que presente animaciones de este

Figura 3.5: *Una animación de un modelo tridimensional con cuatro* keyframes.

tipo, además de la información estática y de estructura de la malla, se requiere almacenar:

1. Las posiciones y normales de cada vértice para cada *keyframe*.
2. El intervalo de tiempo entre un *keyframe* y el siguiente.

De lo cual se concluye que las estructuras de datos para un modelo animado extienden a las de un modelo estático. Adicionalmente, los costos en memoria de tener un modelo animado son directamente proporcionales a la cantidad de *keyframes* y de vértices del modelo.

Vertex Blending

Esta técnica también es llamada *Skeletal Animation* [LCF00], ya que se construye un **esqueleto** que se asigna a la malla. Toda animación o deformación se aplica al esqueleto, que está formado por huesos conectados mediante articulaciones. Cada vértice de la malla está asociado mediante un peso a una articulación. Así, la deformación que se aplica al esqueleto se aplica a la malla de acuerdo a esas asociaciones.

En la Figura 3.6 se observa un modelo tridimensional con esqueleto, que visiblemente es mucho más sencillo que la malla. Esto permite una mayor flexibilidad a la hora de hacer animaciones, e incluso a partir de un modelo con esqueleto se pueden extraer diferentes *keyframes* si sólo se cuenta con una implementación de la técnica anterior.

Se puede concluir entonces que un modelo con esqueleto es, a grandes rasgos, un modelo estático con la información adicional del esqueleto, ya que mantiene la estructura de la malla. El costo asociado a este tipo de animación es más alto en términos de cálculo que en términos de memoria, ya que en cada cuadro de la animación se deben recalcular las posiciones y vectores normales de los vértices en base a las articulaciones del esqueleto. Debido a ello, es más difícil de implementar que la técnica anterior, pero sin duda es una técnica mucho más extensible a la hora de modificar y agregar nuevas animaciones a un modelo.

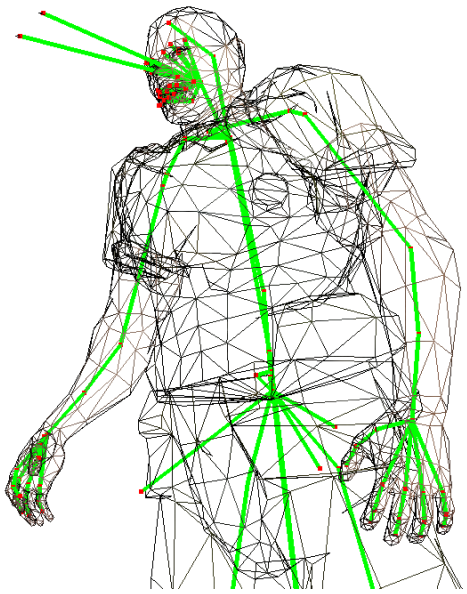


Figura 3.6: *Un modelo tridimensional del juego* Doom 3 *con esqueleto. Las líneas negras demarcan los triángulos de la malla, y las líneas verdes el esqueleto asociado a ella.*

3.2. Rendering en tiempo real

Se llama *rendering* al **proceso de generar una imagen** a partir de datos que describen una escena que se desea graficar. Este proceso puede llevarse a cabo en **tiempo real** y en **tiempo no-real**. Cuando la imagen es generada *on-line*, es decir, en el mismo instante en que se está visualizando, se dice que se está graficando en tiempo real. En tiempo no-real las imágenes son generadas *off-line*, por lo que pueden ser visualizadas tiempo después. Ejemplos de rendering en tiempo real son los videojuegos, cuya gráfica se está generando mientras se juega, y de rendering en tiempo no real son las películas animadas, que fueron generadas con anterioridad a su exhibición en los cines.

Una pregunta que surge respecto al tiempo real es: *¿cuál es el tiempo mínimo para el cual se puede considerar que la imagen se está generando al momento de ser visualizada?* Ciertamente un proceso de rendering puede tomar mucho tiempo, y una imagen que se genera en 10 segundos, aunque se genera *en el momento* no lo hace lo suficientemente rápido como para decir que es una generación *instantánea*. No obstante, una generación *realmente* instantánea es imposible. Entonces, ¿cómo se puede definir el rendering en tiempo real?

En [AMMH02] se define rendering en tiempo real como *"generar imágenes que den la sensación de movimiento e interacción"*. Esto quiere decir que se habla de rendering en tiempo real cuando se pueden generar al menos **15 imágenes en un segundo**, de modo que la rápida superposición de ellas produzca las sensaciones mencionadas.

La graficación de modelos tridimensionales en tiempo real abarca desde la lectura de datos, como puede ser la interpretación de los formatos de modelos ya mencionados, hasta su posterior aparición en pantalla. Este proceso no es fijo, depende de factores como la implementación de la biblioteca gráfica que se esté utilizando y el hardware gráfico que se

esté utilizando.

Para el rendering en tiempo real, el proceso de rendering debe ejecutarse una y otra vez con el fin de generar las imágenes que producen la animación que se despliega en la pantalla. Las diversas bibliotecas gráficas que existen ya proveen al desarrollador de un proceso predeterminado, un **fixed pipeline**, que no se puede modificar, pero al que sí se le pueden entregar opciones, y está programado "en duro" en el hardware y en la biblioteca gráfica. Sus etapas pueden verse en la Figura 3.7.

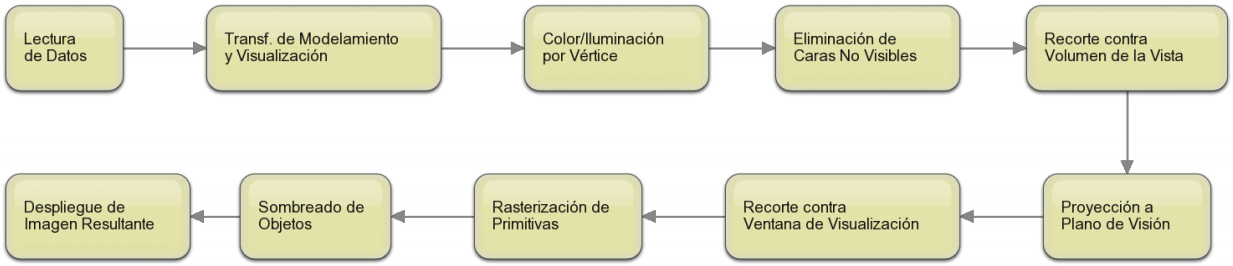


Figura 3.7: *Proceso de rendering típico.*

En el contexto de esta memoria, varias de esas etapas tienen relevancia, pues el marco de trabajo debe trabajar sobre ellas. A continuación se indican estas etapas, en conjunto con una descripción de lo que realizan dentro del proceso de rendering.

3.2.1. Lectura de Datos y Sistemas de Coordenadas

La escena que se desea graficar es leída desde una fuente de datos y almacenada en una estructura de datos en memoria. Esta estructura puede almacenar sus datos de la forma en que lo desee, pero tiene la restricción de entregar sus datos al siguiente paso del proceso de rendering como una malla de triángulos.

Cada objeto o modelo tridimensional que se ha cargado en memoria puede ser especificado en el *sistema de coordenadas de la escena*, o bien en un *sistema de coordenadas propio*. El sistema de coordenadas de la escena tiene un origen fijo, respecto al cual se especifican los vértices del modelo, mientras que el sistema de coordenadas propio tiene un origen fijo para el objeto, pero que no es necesariamente el mismo de la escena o de otros objetos. En este último los vértices se especifican de acuerdo al sistema propio del objeto.

La lectura de datos debiese permitir el tener objetos en sistemas de coordenadas propios y absolutos. Sin embargo, para la biblioteca gráfica el saber si un sistema es absoluto o propio es imposible, por lo que el desarrollador de la aplicación debe identificar el sistema de cada objeto.

Independientemente del sistema de coordenadas utilizado, la posición de los vértices de una malla pertenecen a $\mathbf{R}^3$, pero su representación interna se hace en **coordenadas homogéneas**, definidas como un punto en $\mathbf{R}^4$ donde las tres primeras componentes (x, y, z) corresponden a la posición original, y la cuarta (w) a un factor de normalización. Este factor es 1 por omisión si no se especifica en la posición de un vértice.

Cuando se tiene un punto en coordenadas homogéneas, su conversión a coordenadas tradicionales se realiza como sigue:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_h/w \\ y_h/w \\ z_h/w \end{bmatrix}$$

El uso de coordenadas homogéneas permite uniformar la representación de las matrices de proyección y transformación, así como especificar posiciones en el infinito (con $w = 0$).

3.2.2. Transformaciones de Modelación y Visualización

Cuando ya se tienen todos los objetos cargados en memoria, listos para ser graficados, se debe "uniformar" el sistema de coordenadas de cada uno de ellos, ya que es muy probable que no todos tengan el mismo sistema, de acuerdo a lo visto en la Sección anterior. La forma de realizar dicha uniformación es la *transformación* de los sistemas mediante las **transformaciones de modelación y visualización**, que, a grandes rasgos, se pueden clasificar como operaciones de *traslación*, *rotación* y *escalamiento* que se aplican a los sistemas de coordenadas.

La definición de las tres transformaciones mencionadas es la siguiente:

Transformación de Traslación Una transformación de traslación cambia una posición especificada, trasladándola en una cantidad determinada. Al aplicarse a todos los vértices de un objeto, se cambia la posición del objeto completo. Su formulación es la siguiente:

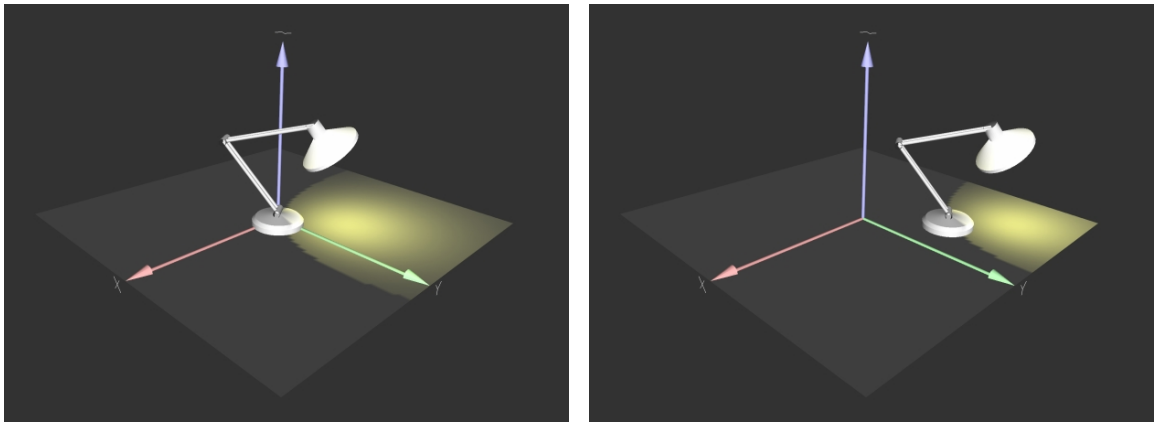
$$T(\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La variable $\mathbf{t}$ es un vector de tres dimensiones que representa la dirección en la que se quiere trasladar la posición que se está premultiplicando por la matriz.

Transformaciones de Rotación Para rotar una posición se debe primero elegir un eje de rotación. Las siguientes matrices representan la rotación de un objeto en torno a los ejes cartesianos:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

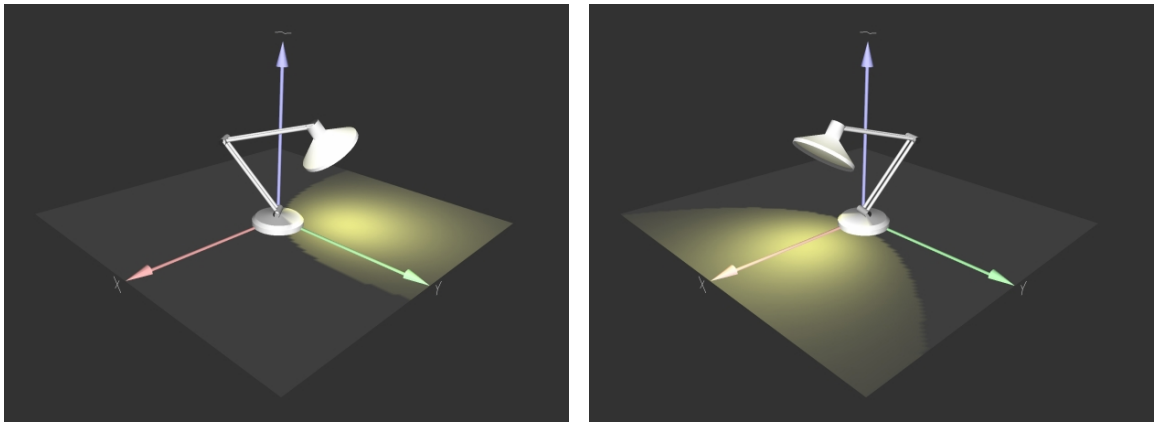


(a) Posición Original (b) Trasladada

Figura 3.8: *Traslación de un modelo tridimensional de una lámpara.*

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La rotación sobre un eje arbitrario se puede construir de diversas maneras. La más típica es realizar una descomposición del eje arbitrario en ejes cartesianos, y realizar las rotaciones que correspondan de acuerdo a la descomposición. También existen otras formas de expresar las matrices de rotación, como pueden ser las *transformaciones de Euler* o los *Quaterniones*, pero escapan del contexto de este capítulo.



(a) Posición Original (b) Rotación

Figura 3.9: *Rotación de un modelo tridimensional de una lámpara.*

Transformación de Escalamiento Para escalar una posición (es decir, multiplicar todas sus componentes por un factor de escalamiento) se debe premultiplicar la posición por

la siguiente matriz:

$$S(\mathbf{s}) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La variable **s** es un vector de tres dimensiones que representa los escalamientos para cada componente de la posición que se está premultiplicando por la matriz. El factor de normalización de la posición se mantiene intacto luego del escalamiento.

El uso de coordenadas homogéneas permite expresar todas estas transformaciones como matrices de 4×4. Al ser expresadas de manera uniforme, es posible combinar distintas matrices mediante multiplicación y premultiplicación.

En general, cualquier matriz de 4×4 puede utilizarse como una matriz de transformación. Por lo tanto, existen otras matrices aparte de las elementales, que pueden ser definidas arbitrariamente por el desarrollador o bien ser composiciones de las transformaciones ya mencionadas. En particular, las transformaciones de rotación y escalamiento se definen respecto al origen del sistema de coordenadas actual al momento de efectuar las transformaciones. Por ejemplo, si se tiene un objeto en coordenadas de la escena, y se desea que rote sobre su propio centro y no sobre el origen del sistema de coordenadas actual, se debe realizar una composición de transformaciones:

1. Trasladar el objeto hacia el origen.
2. Realizar las transformaciones de rotación necesarias.
3. Trasladar el objeto nuevamente hacia su posición original.

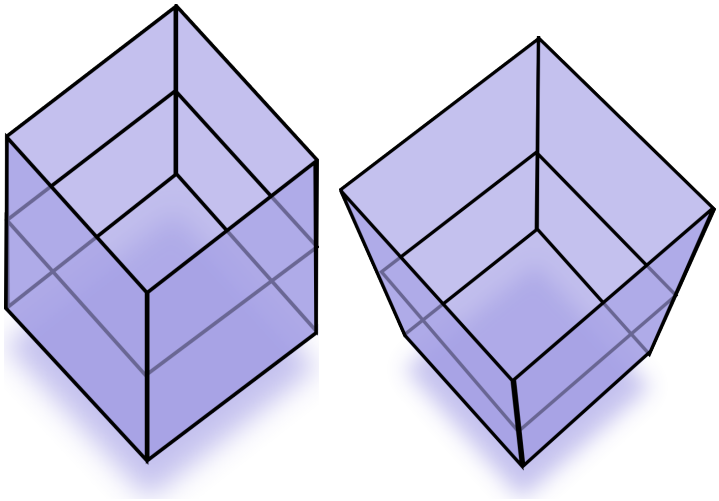
Este procedimiento genera una matriz que permite rotar al objeto respecto a su propio eje, sin importar el sistema de coordenadas en que se encuentre.

3.2.3. Proyección

En esta etapa los triángulos que constituyen los objetos son proyectados al plano de visión, que a su vez contiene al *viewport* o ventana de visualización. Esta proyección es definida por el usuario, aunque las bibliotecas gráficas suelen tener definidas las proyecciones paralelas y en perspectiva, de las cuales sólo se requiere entregar algunos parámetros. En la Figura 3.10 se observan los dos tipos de proyección en conjunto con sus volúmenes de vista respectivos. Las características de estas dos proyecciones son las siguientes:

Proyección En Perspectiva : Simula la perspectiva natural del ojo humano, en la cual el volumen de vista tiene forma de cono con la punta cortada. Este cono se aproxima con una pirámide. En esta proyección, a medida que los objetos se encuentran más lejos, se ven más pequeños.

Proyección Paralela : Simula una perspectiva ortogonal, en la cual dos líneas paralelas, cualesquiera sea su orientación, se mantienen paralelas al ser proyectadas. En esta proyección, aunque los objetos se encuentren lejos, siempre se observa su tamaño original. Es muy utilizado en los sistemas de diseño asistido por computador.



(a) Proyección Paralela (b) Proyección en Perspectiva

Figura 3.10: *Un paralelogramo graficado con dos tipos de proyecciones: Ortogonal o Paralela y en Perspectiva.*

Del mismo modo que las transformaciones, para proyectar una posición es necesario premultiplicarla por la matriz correspondiente. Y, a diferencia de las transformaciones, para cada tipo de proyección existe una gran cantidad de matrices distintas, pero con efectos similares (en general, cada biblioteca gráfica implementa sus propias matrices de proyección).

3.2.4. Rasterización y Sombreado

El término *rasterización* proviene de los dispositivos *raster*, que utilizan una matriz de *píxeles* para desplegar una imagen. Un píxel es la unidad mínima de imagen y puede contener **un** color. La cantidad de elementos de la matriz es llamada *resolución*: una resolución típica es 1280×720, lo cual quiere decir que la imagen tiene 1280 píxeles de ancho por 720 píxeles de alto.

Por rasterización (en inglés *rastering*) se entiende el discretizar las primitivas que se tienen en el plano de la ventana de visualización. Ya que la matriz de píxeles tiene una cantidad determinada de elementos, los triángulos de interés que han llegado hasta esta etapa deben transformarse en conjuntos de píxeles. Existen diversos algoritmos que llevan a cabo esta tarea, siendo el más conocido el *Algoritmo de Bresenham* [Bre77]. Un ejemplo de la aplicación de este algoritmo muestra la rasterización de una línea en una grilla de píxeles en la Figura 3.11.

Como toda discretización de una señal continua, es posible que el muestreo de los datos sea insuficiente para que la representación de los valores discretizados represente fielmente los

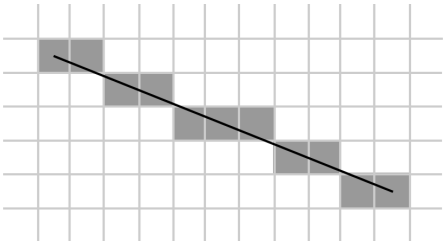


Figura 3.11: *Rasterización de una línea. Se aprecia claramente el escalonado producto de la discretización de la curva.*

datos originales. Este fenómeno es llamado *aliasing*, y en Computación Gráfica se manifiesta mediante un escalonamiento en las líneas y en los bordes de las líneas y triángulos. Existen técnicas de *antialiasing*, pero escapan del contexto de este trabajo.

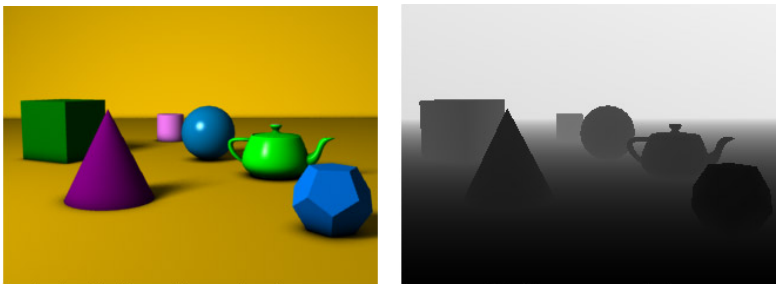
Cada triángulo de la malla está definido por el espacio contenido entre esos tres vértices, pero después de la rasterización se tiene un arreglo de píxeles que aproxima el área de ese triángulo. El proceso de rastering se ejecuta para cada triángulo que se proyecta en la ventana de visualización y que pasa la etapa de *clipping* (en la cual se descartan y recortan triángulos que, luego de ser proyectados al plano de visión, no aparecen en la ventana de la aplicación). Como se conoce la información de los vértices del triángulo, es necesario interpolar la información en cada píxel para poder sombrearlo (en inglés se utiliza el término *shade* de *shading*).

Esto quiere decir que a cada píxel se le asigna una interpolación de los valores asociados a los vértices del triángulos. Así, un píxel tiene, antes de ser pintado, una posición, un vector normal, una coordenada de textura, una intensidad de iluminación y otros valores que interesen al desarrollador. Esta información permite utilizar una técnica de sombreado y definir el color que será almacenado en el píxel. Existen tres modelos primarios de sombreado, descritos en la Sección 3.4.

3.2.5. Despliegue de la Imagen

Una vez sombreados todos los triángulos, se pueden desplegar en pantalla copiando el valor de color de los píxeles de cada triángulo en la matriz de despliegue del dispositivo de visualización. Ahora bien, no siempre es conveniente desplegar todos los píxeles directamente, porque en ninguna etapa del proceso se realizó un ordenamiento de los triángulos respecto a su profundidad. Como consecuencia, si se tiene un triángulo que tapa a otro, pero el que estaba oculto es procesado después del que lo tapaba, y no se realiza un chequeo de la profundidad asignada a los píxeles, entonces la imagen no será coherente porque el triángulo que estaba oculto sí será graficado.

Existen dos formas de resolver este problema. La primera es ejecutar algoritmos de eliminación de caras no visibles, y la segunda es realizar un test de profundidad en los píxeles que se están dibujando. Esta segunda solución, conocida como **Z-Buffer**, es la más común y fácil de implementar.



(a) Escena 3D (b) Z-Buffer

Figura 3.12: *Escena 3D ya graficada y su representación en el* Z-Buffer.

El *Z-Buffer* es una matriz, del mismo tamaño que el *viewport*, que en cada elemento almacena la profundidad del último píxel desplegado en pantalla. De modo que, si ese píxel debe ser sobreescrito, se comprueba primero si el nuevo candidato tiene una profundidad menor. En ese caso se sobreescribe, en caso contrario se descarta. Se puede decir que el Z-Buffer representa una imagen en escala de grises con los valores de profundidad, como se ve en la Figura 3.12(b).

Cuando ya se han procesado todos los píxeles de todos los triángulos, la imagen ya se encuentra en pantalla y el proceso de rendering ha terminado, como puede observarse en la Figura 3.12(a).

3.3. Luces y Modelo de Iluminación de Phong

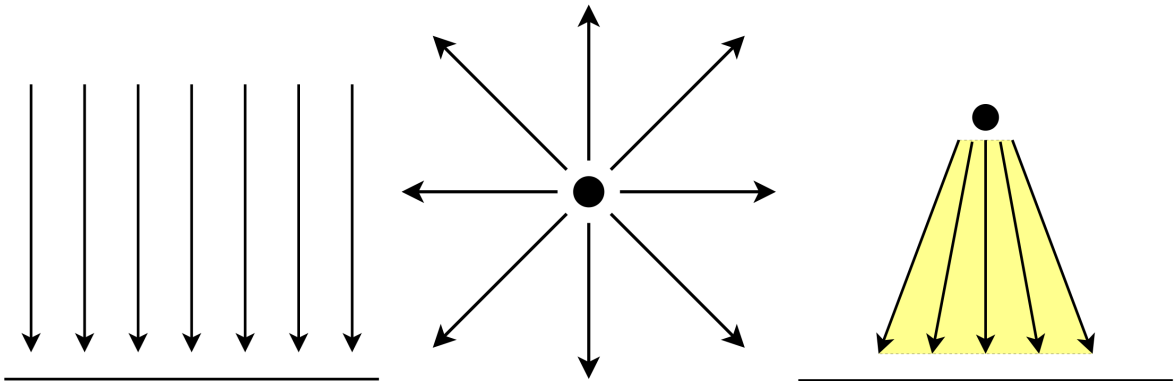
En Computación Gráfica se requiere que los objetos tengan una representación geométrica coherente con lo que modelan, como en el caso de una esfera. Se sabe que cualquier discretización en mallas de triángulos de una esfera nunca la representará perfectamente, pero se puede lograr que, geométricamente, las propiedades de la malla aproximen las propiedades del objeto real. En el rendering fotorrealista se requiere que, además de la representación geométrica, la apariencia visual de los objetos sea lo más similar posible a la imagen real. Esto incluye el considerar diversas fuentes de luz que permiten observar los objetos, en los cuales la luz cambia de intensidad en sus superficies.

En una escena, tanto real como ficticia, existen diferentes fuentes de luz. Por ejemplo, en una habitación puede haber una ampolleta encendida en el techo; quizás entra también un poco de luz por las ventanas, y si hay un monitor o una televisión, sus pantallas también emiten luz. Cada una de estas entidades *emisoras* de luz es llamada **fuente de luz**.

En la vida real, la luz llega a una superficie pero no siempre se refleja completamente. Una luz blanca puede llegar a un objeto del cual sólo se refleja su componente azul, por lo que al ojo humano ese objeto es, precisamente, azul. En Computación Gráfica ese comportamiento de la luz es aproximado con las propiedades de material y las propiedades de la fuente de luz. Un **modelo de iluminación** toma las propiedades de material, de la luz y de la superficie y *evalúa* cuál es el color final, o *intensidad de iluminación*, que corresponde a la superficie en un punto dado.

Los modelos de iluminación se pueden clasificar en *globales* y *locales*. Los modelos globales son los más completos, ya que consideran las interacciones entre los objetos y el comportamiento de la luz en una escena. Los modelos locales, a su vez, al evaluar la intensidad de iluminación en una superficie, solamente consideran las propiedades de la superficie y de las luces que se están evaluando. Debido a esto, para rendering en tiempo real el uso de modelos globales es demasiado restrictivo, por lo que se utilizan modelos locales, en particular el **modelo de iluminación de Phong**.

3.3.1. Fuentes de Luz



(a) Luz Direccional (b) Luz Puntual (c) Linterna

Figura 3.13: *Tipos de Fuentes de Luz en Computación Gráfica.*

Existen tres tipos de fuentes de luz en Computación Gráfica: las **luces direccionales**, las **luces puntuales** y las **linternas**. En la Figura 3.13 se muestra una representación gráfica de ellas, que se definen como sigue:

Luz Direccional (Figura 3.13(a)). Como su nombre lo dice, esta luz incide en cada punto de una superficie desde una misma dirección. Un análogo a este tipo de fuente de luz es el Sol visto desde la superficie de la Tierra: en cada punto del planeta, desde el cual el Sol es visible, la luz de la estrella incide casi con la misma dirección (para efectos prácticos, es la misma). Para representar una luz direccional, aparte de su dirección, se dice que está situada *en el infinito*.

Luz Puntual (Figura 3.13(b)). Una luz puntual es una fuente de luz ubicada en un punto determinado del espacio que emite luz en todas las direcciones. Nuevamente, un ejemplo de esto es el Sol, pero en el contexto del sistema solar: la estrella emite luz en todas las direcciones posibles desde una posición particular.

Linterna (Figura 3.13(c)). Una linterna es una luz puntual pero acotada por una dirección en la que debe apuntar y un ángulo de corte.

En general, a lo largo de los años ha bastado con este tipo de fuentes de luz para simular un gran número de escenas. Se podría decir que, más que tener fuentes de luz más realistas

y menos idealizadas, es más crítica la evaluación de un modelo de iluminación global y completo.

3.3.2. Modelo de Iluminación de Phong

Bui Tuong Phong, en 1973, formuló el modelo de iluminación que lleva su nombre. Mezclando conceptos físicos y consideraciones heurísticas, descompone la iluminación de una escena en tres componentes: la parte *ambiental*, que llega a toda la escena con la misma intensidad; la parte *difusa*, que depende del ángulo de incidencia (se utiliza el modelo de reflexión de Lambert); y la parte *especular*, que define cuánto de la luz se refleja en la superficie como si fuese un espejo. Debido a esta descomposición las propiedades de material de una superficie incluyen esas tres componentes.

Entonces, el modelo de iluminación evalúa la intensidad de iluminación en una superficie de la siguiente manera:

$$I(\mathbf{p}) = I_a K_a + \sum_{luces} \left[I_d K_d * N(\vec{\mathbf{p}}) \cdot L(\vec{\mathbf{p}}) + I_s K_s * \left(R(\vec{\mathbf{p}}) \cdot V(\vec{\mathbf{p}}) \right)^n \right]$$

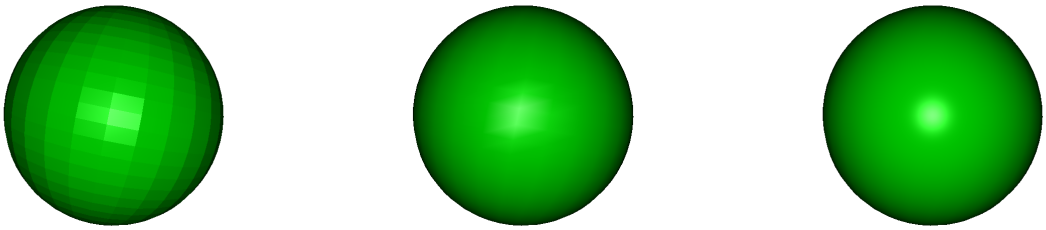
Donde los valores I corresponden a las componentes de la luz (I_a es su componente ambiental, I_d es su componente difusa e I_s es su componente especular). Los valores K corresponden a sus análogas propiedades de material. Los otros valores de la ecuación son:

- $\mathbf{p}$, la posición en la cual se está evaluando el modelo de iluminación.
- $N(\vec{\mathbf{p}})$, la normal (normalizada) de la superficie en ese punto.
- $L(\vec{\mathbf{p}})$, el vector normalizado con la dirección desde $\mathbf{p}$ hasta la posición de la fuente de luz.
- $R(\vec{\mathbf{p}})$, dirección (normalizada) en la que se refleja la luz en la superficie de acuerdo a la Ley de Snell.
- $V(\vec{\mathbf{p}})$, el vector normalizado con la dirección desde $\mathbf{p}$ hasta la posición de la vista (depende del tipo de proyección).

Se aprecia que el modelo de Phong es un modelo *local*; su fórmula no considera la interacción entre los objetos.

3.4. Técnicas de Sombreado

Sombreado se define como asignar un color a un píxel. Ahora bien, una vez definido el modelo de Phong, ¿cómo evaluarlo para saber cómo sombrear los píxeles? Existen tres formas de evaluar el modelo de iluminación: en el triángulo (Sombreado Plano), en los vértices (Sombreado de Gouraud) y en los píxeles (Sombreado de Phong). Estas técnicas se pueden apreciar en la la Figura 3.14.



(a) Plano (b) Gouraud (c) Phong

Figura 3.14: *Comparación de técnicas de sombreado en un modelo de esfera con una triangulación medianamente gruesa. Se observa que el sombreado plano delata el grosor de la malla, que el sombreado de Gouraud presenta un sombreado difuso de calidad pero que en la componente especular la interpolación es de mala calidad. El sombreado de Phong, por otro lado, presenta resultados que no delatan el grosor de la malla, al ser evaluado por píxel.*

3.4.1. Sombreado Plano

Inicialmente, en el rendering en tiempo real el poder de procesamiento no era suficiente para poder simular la apariencia suave de la iluminación. Lo mejor que se podía hacer, aparte de tener modelos con mallas gruesas, era elegir un color único para cada triángulo basado en la iluminación que recibía. De este modo, cada píxel asociado al triángulo es pintado con ese color. Debido a esto, esta técnica de sombreado es plana, ya que dentro de un triángulo no se aprecian variaciones en la intensidad de la luz incidente.

El Algoritmo 3.1 muestra como se realiza este cálculo para un triángulo, considerando algunas simplificaciones como la omisión de la normalización de los vectores. Se observa que el modelo de iluminación se evalúa una sola vez por cada triángulo, aunque existe una variación de este algoritmo que lo evalúa tres veces (una en cada vértice) y luego obtiene una intensidad promedio para todo el triángulo.

Algoritmo 3.1: Algoritmo de Sombreado Plano para un triángulo.

Input: triángulo t
Input: conjunto de píxeles p
Output: conjunto de píxeles p
$I_t \leftarrow$ evaluar_iluminación(t.centroide, t.normal, t.material);
foreach *píxel* p_i *en* p **do**
  asignar_color(p_i, I_t);
end

3.4.2. Sombreado de Gouraud

Ya que cada triángulo es discretizado en un conjunto de píxeles, lo ideal es evaluar el modelo de iluminación elegido en cada uno de ellos. Sin embargo, cuando todavía no se contaba con el poder de procesamiento suficiente para realizar dicha tarea, se utilizaba una técnica de sombreado que presentaba resultados de buena calidad y que era de bajo costo

computacional. Esta técnica es el Sombreado de Gouraud. Inicialmente, en el rendering en tiempo real no se podía calcular la intensidad de la luz para cada punto de la superficie, por lo que solamente se calculaba en los vértices de los triángulos y luego se interpolaba en los píxeles que correspondían al objeto. Este tipo de interpolación es llamado *Sombreado de Gouraud* y se puede observar en el Algoritmo 3.2.

Algoritmo 3.2: Algoritmo de Sombreado de Gouraud para un triángulo.

Input: triángulo t
Input: conjunto de píxeles p
Output: conjunto de píxeles p
$I_{t_1} \leftarrow$ evaluar_iluminación(t.v1, t.v1.normal, t.v1.material);
$I_{t_2} \leftarrow$ evaluar_iluminación(t.v2, t.v2.normal, t.v2.material);
$I_{t_3} \leftarrow$ evaluar_iluminación(t.v3, t.v3.normal, t.v3.material);
foreach *píxel* p_i *en* p **do**
    $I_t \leftarrow$ interpolar(t.v1, t.v2, t.v3, I_{t_1}, I_{t_2}, I_{t_3}, p_i.posicion);
    asignar_color(p_i, I_t);
end

Esta técnica de sombreado evalúa tres veces el modelo de iluminación elegido. Sin embargo, realiza otras operaciones matemáticas que pueden ser costosas (aunque no tanto como evaluar el modelo en cada píxel), ya que es necesaria una interpolación de las intensidades de iluminación calculadas para cada vértice (función `interpolar` en el algoritmo). Dicha interpolación se puede implementar de forma ingenua, utilizando distancia euclidiana, o se puede implementar una aproximación de la interpolación utilizando una interpolación bilineal. La función especificada en el algoritmo recibe las tres posiciones de los vértices, los tres valores que se interpolarán, y la posición en la cual se desea obtener el valor interpolado.

A partir de esto se puede deducir que los resultados gráficos del sombreado de Gouraud dependen, primero, de la implementación, y después de la finura de la malla que se está graficando. Si la malla es gruesa se puede obtener un sombreado que no es coherente con la información de los triángulos, ya que la intensidad de luz en los vértices no suele ser representativa de la intensidad con la que llega al interior de ellos. Un ejemplo es el siguiente: si con una linterna se enfoca al centro del triángulo, de modo que no llegue luz a los vértices, el triángulo se dibujará totalmente opaco, cuando en realidad sí le llegaba luz. La única forma de resolver esta situación es utilizando una malla lo suficientemente fina para que no se pierda el efecto de la luz.

3.4.3. Sombreado de Phong

El sombreado de Phong soluciona los problemas que presenta el sombreado de Gouraud a costa de un mayor cálculo. En vez de calcular la intensidad de la luz en cada vértice, esta técnica de sombreado interpola las normales de cada uno de ellos, con el fin de obtener una aproximación de la normal de la superficie en el píxel que se va a pintar y así poder evaluar el modelo en cada píxel resultante de la rasterización. El Algoritmo 3.3 muestra el pseudo-código de esta técnica de sombreado.

Algoritmo 3.3: Algoritmo de Sombreado de Phong para un triángulo.

Input: triángulo t
Input: conjunto de píxeles p
Output: conjunto de píxeles p
foreach *píxel* p_i *en* p **do**
  $\mathbf{n}_{p_i} \leftarrow$ interpolar(t.v1, t.v2, t.v3, t.v1.normal, t.v2.normal, t.v3.normal, p_i.posicion);
  $m_{p_i} \leftarrow$ interpolar(t.v1, t.v2, t.v3, t.v1.material, t.v2.material, t.v3.material, p_i.posicion);
  $I_{p_i} \leftarrow$ evaluar_iluminación(p_i.posicion, $\mathbf{n}_{p_i}$, m_{p_i});
  asignar_color(p_i, I_{p_i});
end

El poder de cálculo necesario para utilizar el Sombreado de Phong es evidente. No solamente se evalúa el modelo de iluminación en cada píxel, sino que se deben interpolar dos valores: la normal y las propiedades de material de los vértices (si es que son diferentes). Sin embargo, sus resultados son los más exactos posibles, ya que no se puede refinar más el algoritmo, al menos para las mallas de triángulos, donde la información sólo se encuentra en los vértices. Para obtener mejores resultados visuales se debe conocer información por píxel, porque es posible que la normal interpolada no sea representativa de la superficie en el punto de interpolación.

Al analizar el Sombreado de Gouraud se dijo que para mejorar su precisión visual se debía refinar la malla. Pues bien, refinar la malla es aproximar el Sombreado de Phong. Si la malla fuese tan fina que en cada píxel se dibuja un triángulo, los resultados visuales de ambas técnicas de sombreado serían idénticos.

3.5. Interpolaciones y Curvas Paramétricas

Una gran cantidad de técnicas y algoritmos de Computación Gráfica utilizan datos que deben ser interpolados, es decir, deben ser *reconstruidos a partir de datos ya conocidos*. En las secciones anteriores se han mencionado situaciones que requieren interpolación de datos.

Por ejemplo, en el Sombreado de Gouraud y de Phong se habló de la *interpolación bilineal*, que consiste en interpolar datos realizando dos interpolaciones a través de dos rectas. En un triángulo esto es muy útil ya que se tienen tres datos conocidos, y establecer una única fórmula para interpolar podría ser costoso. Es mejor realizar más interpolaciones pero más sencillas que probablemente estén soportadas por hardware. En el caso del sombreado, primero se realiza una **interpolación lineal** a través de las aristas, y luego se vuelve a interpolar considerando una línea de barrido entre dos aristas que resulten de interés.

Una *interpolación lineal* consiste en reconstruir un dato desconocido entre dos datos conocidos mediante una recta:

$$P(t) = t * P(0) + (1 - t) * P(1)$$

Donde $P(0)$ y $P(1)$ son los dos datos conocidos, llamados **puntos de control**, y $P(t)$ la interpolación de ellos. El **parámetro** t permite decidir dónde se desea interpolar, un valor

más cercano a 0 entrega un dato más cercano a $P(0)$, mientras que un valor más cercano a 1 entrega un valor más cercano a $P(1)$.

Se puede replantear una interpolación como una mezcla entre distintos puntos de control. La interpolación lineal se plantea como:

$$P(t) = F_0(t) * P(0) + F_1 * P(1)$$

Donde las funciones F son consideradas **funciones de mezcla** y deciden cuánto de $P(0)$ y cuánto de $P(1)$ se usa para obtener $P(t)$ en base al parámetro t. La suma de la evaluación de ambas funciones debe ser 1 para cumplir con el requisito de interpolar los dos puntos conocidos.

En general, se puede utilizar cualquier tipo de función para llevar a cabo una interpolación: existen interpolaciones polinomiales y trigonométricas, entre otras. En la Figura 3.15 se observa una comparación entre una interpolación lineal, que gráficamente une cada par de puntos seguidos con una recta, y una interpolación de polinomios cúbicos, que presenta una interpolación más suave y probablemente más representativa. Sin embargo, una interpolación cúbica, para ser más representativa, requiere información extra. Se necesitan cuatro puntos de control para interpolar: el punto por el cual ya pasó, los dos por los cuales debe pasar la curva, y el punto por el cual pasará después.

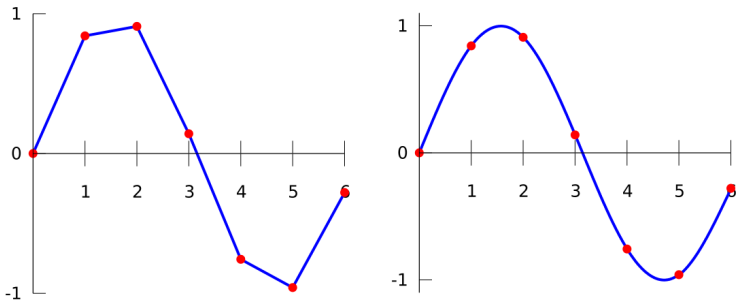


Figura 3.15: *Dos tipos de interpolación para el mismo conjunto de puntos:* lineal *(izquierda),* spline *(derecha).*

La interpolación es una técnica que se aplica para cada conjunto de datos de forma singular, es decir, la interpolación y las fórmulas obtenidas para un conjunto de datos generalmente no sirven para interpolar otro conjunto. Debido a esto surge la necesidad de expresar una interpolación de forma generalizada, de modo que se pueda aplicar la técnica a un conjunto de datos arbitrario. Las **curvas paramétricas** realizan precisamente eso: una curva paramétrica recibe un valor de parámetro t y entrega la evaluación de ese parámetro de acuerdo a diferentes puntos de control especificados para la curva. En la interpolación se recibe como entrada un *conjunto de funciones de mezcla* y un *conjunto de puntos de control*; en las curvas paramétricas las funciones de mezcla *ya existen* y están *parametrizadas* por los puntos de control que se indiquen.

Aunque el planteamiento depende de la curva paramétrica con la que se esté trabajando, en el caso de las curvas cúbicas, que son las que interesan en el contexto de esta memoria, si se tiene un conjunto de puntos $K = [P_0, P_1, \ldots, P_n]$ se itera para todo $i \leq n$ mediante secuencias

$[P_{i-1}, P_i, P_{i+1}, P_{i+2}]$ para reconstruir la curva entre los puntos P_i y P_{i+1}, utilizando una función de mezcla determinada. Cada una de estas reconstrucciones entre P_i y P_{i+1} es llamada *parche*, y dependiendo de la curva paramétrica utilizada los parches tendrán diferentes condiciones de continuidad.

Para cada conjunto de puntos de control se evalúa el parámetro t con valores en el intervalo $[0, 1]$. Así, la suavidad gráfica de la curva resultante depende del número de veces que es evaluado el parámetro. Esto provee una gran flexibilidad que permite decidir si se quiere una mejor calidad en la interpolación a costo de un mayor tiempo de cálculo.

A continuación se estudiarán dos curvas paramétricas cúbicas. Estas curvas son de gran interés en esta memoria por las propiedades que poseen y porque sus funciones de mezcla se puede expresar utilizando una matriz de 4×4.

3.5.1. B-Splines

Las B-Splines en realidad no son verdaderas curvas de interpolación, ya que no pasan por todos los puntos de control indicados. Esto no es necesariamente malo, porque en algunos casos el muestreo de datos tiene perturbaciones de ruido o errores de muestreo que no son despreciables. En dichos casos las B-Splines son útiles porque "suavizan" la curva.

Otras dos propiedades importantes son las siguientes: cada parche de la curva tiene continuidad C^2 con los parches anterior y siguiente, y la curva final se encuentra en el cierre convexo de los puntos de control. Esto último es particularmente útil si se desea realizar algún test de intersección con la curva.

La función de mezcla de las B-Splines es la siguiente:

$$B(\mathbf{t}) = \left[\begin{array}{cccc} (t-t_i)^3 & (t-t_i)^2 & (t-t_i)^1 & 1 \end{array} \right] \frac{1}{6} \left[\begin{array}{cccc} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{array} \right] \left[\begin{array}{c} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{array} \right]$$

$$, t_i < t < t_{i+1}, 3 < i < m$$

Donde m es el total de puntos de control.

Es posible forzar la interpolación de algunos puntos de control. Basta repetirlos en la lista de puntos el número de veces necesario para que la curva se acerque a esos puntos de interés.

3.5.2. Splines de Catmull-Rom

Otro tipo de spline son las curvas de Catmull-Rom. Entre sus propiedades se encuentran la continuidad C^1 entre parches y la interpolación de todos los puntos de control; aunque la curva no se encuentra dentro de la cerradura convexa definida por dichos puntos. Por lo tanto, este tipo de curvas permite realizar un recorrido suave a través de los puntos en el conjunto de datos que se desea aproximar. Un ejemplo de aplicación idónea de curvas de Catmull-Rom es la interpolación de animaciones con *keyframes*.

La función de mezcla de esta curva es la siguiente:

$$B(\mathbf{t}) = \begin{bmatrix} (t-t_i)^3 & (t-t_i)^2 & (t-t_i)^1 & 1 \end{bmatrix} \begin{bmatrix} -\tau & 2-\tau & \tau-2 & \tau \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 0 & \tau & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

$$, t_i < t < t_{i+1}, 1 < i < m$$

Donde τ es un parámetro de tensión que permite definir la rigidez de la curva (usualmente tiene un valor de 0,5). La curva pasa por los puntos P_i y P_{i+1}.

3.6. OpenGL

Implementar los conceptos enunciados, desde sus aristas más básicas hasta las más complejas, requiere un profundo manejo de hardware que solamente entorpecería el trabajo de esta memoria, ya que su foco no es implementar una biblioteca gráfica, sino más bien implementar los conceptos gráficos y geométricos necesarios para cumplir los objetivos enunciados utilizando una biblioteca gráfica actualizada, estable y de código abierto. Por ello, se ha decidido utilizar una biblioteca que cumple con estos requerimientos.

`OpenGL` [SWND03] es la biblioteca elegida en esta memoria para desplegar gráficos y comunicarse con la GPU. Su nombre significa Open Graphics Library, es multiplataforma y está en constante actualización.

Esta biblioteca ofrece diferentes **primitivas gráficas** para desplegar los gráficos. Permite recibir puntos (*points*), triángulos especificados como listas de triángulos (*triangles*), como tiras de triángulos (*triangle strips*) y como abanicos de triángulos (*triangle fans*); también soporta cuadriláteros (*quads*) y tiras de cuadriláteros (*quad strips*, así como polígonos convexos sin huecos. Además soporta líneas (*lines*), tiras de líneas (*line strips*) y loops de líneas (*line loop*).

OpenGL por sí sola es una biblioteca que solamente se encarga del despliegue de los gráficos. No maneja ventanas, ni entrada del usuario, ni sonidos, ni red, y en general nada que no sea el despliegue de gráficos. Debido a esto siempre es necesario utilizarla en conjunto con una biblioteca que se encargue de todo lo demás.

OpenGL funciona como una **máquina de estados**. Una máquina de estados es un autómata finito que tiene una cierta cantidad de estados y transiciones. Cuando el desarrollador desea dibujar en la pantalla, le indica a OpenGL que desea dibujar algo y el nombre de la primitiva con la cual quiere dibujar. Luego, le envía una lista de vértices, que es interpretada de acuerdo al estado seleccionado.

3.6.1. Graficación de Objetos Tridimensionales

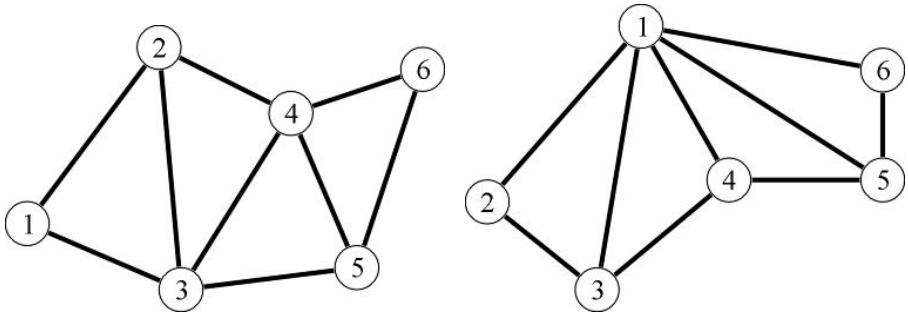
La manera más sencilla, entonces, de graficar un modelo tridimensional es activar el estado de dibujo con una primitiva de *lista de triángulos*. Luego se entrega la lista de vértices de

cada modelo tridimensional, en el orden en que éstos definen a los triángulos del modelo. Cada vértice incluye toda la información necesaria para el rendering que se ha visto en este capítulo, es decir, posiciones, vectores normales, coordenadas de textura y otros.

Ahora bien, ¿es esto lo óptimo realmente? En un modelo que tenga, por ejemplo, 1.000 triángulos, se deben enviar 3.000 vértices. A cada uno de ellos se les aplican transformaciones, proyecciones, se les calcula la intensidad de iluminación, entre otros. Pero muchas veces, sobretodo en una superficie suave, los vértices de triángulos vecinos tienen, además de la misma posición, los mismos vectores normales y coordenadas de textura. En esos casos es mejor utilizar primitivas que permitan indicar esa información compartida, de modo que se deban enviar menos vértices y se ahorren cálculos repetidos. Es aquí donde entran en juego las tiras y los abanicos de triángulos:

Tiras de Triángulos (Figura 3.16(a)): permiten indicar triángulos que son vecinos entre sí, es decir, comparten una arista. Primero se entregan los vértices del primer triángulo. Del segundo triángulo solamente se entrega un vértice: se asume que ese segundo triángulo lo constituyen los dos últimos vértices del primero y el nuevo vértice que se ha entregado. Para el tercer triángulo se repite el mismo proceso. Esto permite entregar n triángulos especificando $n-2$ vértices. A medida que el número de triángulos en la tira crece, la relación de triángulos por vértices tiende a 1.

Abanicos de Triángulos (Figura 3.16(b)): permiten indicar una serie de triángulos con una punta o vértice en común. Por ejemplo, son muy útiles para especificar una circunferencia: primero se entrega el centro de la circunferencia, y luego uno por uno los vértices que aproximan el círculo que la delimita. Además son muy útiles para triangularizar polígonos convexos. También permiten entregar n triángulos especificando $n-2$ vértices.



(a) Triangle Strip (b) Triangle Fan

Figura 3.16: *Primitivas de triángulos en OpenGL:* triangle strips *y* triangle fans. *Los números en los vértices indican el orden en el cual se entregan a OpenGL. En ambos casos se especifican 4 triángulos utilizando 6 vértices.*

Si bien las mejoras en el rendimiento al utilizar estas primitivas gráficas son importantes, un problema importante que surge es el hecho de *encontrar* las tiras y los abanicos dentro de un modelo 3D. Usualmente un modelo se almacena como un conjunto de triángulos sin información de vecindad, por lo que estas primitivas deben ensamblarse y reconstruirse cada

vez que se carga el modelo en memoria. Lamentablemente no es una tarea fácil, ya que encontrar las tiras de triángulos óptimas para un modelo 3D es un problema NP-Completo (llamado *k-stripability*). Sin embargo, existen heurísticas que permiten encontrar tiras de calidad (lo más largas posibles), como la propuesta en [EMX02].

3.7. GPUs – Unidades de Procesamiento Gráfico

Como se ha visto en este Capítulo, una gran cantidad de técnicas gráficas hacen un uso extensivo de operaciones matriciales y vectoriales para las cuales los procesadores tradicionales no están preparados. Hoy, a pesar de tener CPUs multi-núcleo con velocidades que sobrepasan los GHz de magnitud, dichas operaciones siguen siendo muy costosas debido al diseño *escalar* de los procesadores. Entonces, se hace necesario el contar con hardware extra que realice todos estos cálculos vectoriales y especializados. Este hardware es la *tarjeta de vídeo*.

Las tarjetas de vídeo tradicionales tenían programadas en hardware algunos de los algoritmos que utiliza el *fixed pipeline*. A medida que progresaba la tecnología se disponía de mejores algoritmos y de mejor calidad de imagen, pero el funcionamiento de las tarjetas estaba limitado a lo que había sido programado en duro en ellas. En la actualidad esto no es así. Los procesadores que incluyen las tarjetas de vídeo, llamados *GPUs* o *Unidades de Procesamiento Gráfico*, pueden ser programados al ejecutar ciertas etapas del proceso de rendering. Por ejemplo, se puede utilizar el sombreado de Phong en vez del sombreado de Gouraud, o reemplazar el modelo de iluminación de Phong por otro, que puede ser más sencillo o más complejo.

Las GPUs son multi-núcleo: una GPU contiene procesadores especializados que trabajan en paralelo, de modo de procesar concurrentemente vértices y píxeles cuando corresponda. Por ejemplo, el modelo *Geforce8800 Ultra*[2] de *nVIDIA* tiene 128 núcleos que corren a 1.5GHz cada uno. El saber aprovechar estas características permite generar efectos visuales de alta complejidad en tiempo real, evaluar modelos de iluminación más realistas e incluso utilizar los procesadores para computación no gráfica, mediante lo que es llamado *General Purpose Computation on GPUs* (*GPGPU*).

3.7.1. El lenguaje Cg

Como todo procesador, una GPU puede ser programada utilizando un lenguaje de máquina o ensamblador. Existen lenguajes de alto nivel, en su mayoría inspirados en C, que permiten escribir programas que pueden ser compilados para distintos modelos de tarjetas. Esto se vuelve totalmente necesario ya que, a diferencia de los procesadores tradicionales, cada GPU tiene un conjunto de instrucciones propio que no tiene por qué ser compatible con las líneas anteriores de GPUs o con las instrucciones de la competencia.

El lenguaje `Cg`[MGAK; FK03] es un lenguaje de programación de alto nivel para procesadores gráficos creado por *nVIDIA*[3]. Su nombre es la abreviación de *C for graphics*. Entre

[2] `http://www.nvidia.com/page/geforce8.html`
[3] `http://developer.nvidia.com/page/cg_main.html`

sus cualidades se encuentran:

- Es multiplataforma: está disponible para los sistemas operativos más comunes (Windows, Linux, MacOS).
- Permite ejecutar programas en GPUs actuales y de generaciones pasadas.
- Los programas se pueden compilar *on-line* dentro de la ejecución de una aplicación u *off-line* antes de la ejecución de la aplicación.
- Se puede utilizar desde OpenGL.

En la actualidad se pueden escribir tres tipos de programa para GPUs:

Vertex Program Estos programas se ejecutan una vez por cada vértice de las mallas que se están graficando. Recibe como entrada el vértice en coordenadas del objeto, por lo que es labor de este programa el aplicar las matrices de transformación y proyección, así como evaluar el modelo de iluminación que se desee si es que se va a utilizar sombreado de Gouraud (ya que en este sombreado basta que se calcule la iluminación en los vértices).

Ya que este programa está en control de la GPU, es posible ejecutar operaciones que la CPU no conoce. Por ejemplo, se puede enviar una grilla plana para ser deformada en el vertex program. La información, una vez que llega a la GPU, no vuelve a la CPU (a menos que la CPU lo requiera).

Además, un vertex program le puede enviar parámetros a los fragment program que se ejecuten. Por cada triángulo se ejecuta tres veces el mismo vertex program; los parámetros de cada uno de ellos se interpolan para definir los parámetros que recibirán los fragment program que se ejecutarán.

Fragment Program Un *fragment program* es invocado después de la etapa de rasterización, y su función es determinar si se va a pintar un píxel de un objeto. En el caso afirmativo, debe calcular el color que le corresponde en base a su entrada, interpolada desde los valores enviados por los vertex program.

Para implementar el sombreado de Phong es necesario utilizar un fragment program. Debe recibir de entrada la posición del píxel, la posición de la luz, el vector normal en el píxel, y la posición de la cámara. La posición y la normal requerida la envían los vertex program asociados al triángulo, mientras que la posición de la luz y de la cámara no varían para el objeto, por lo que las puede enviar la CPU.

Geometry Shader La última generación de GPUs permite programar *geometry shaders*, que solucionan algunas de las limitantes de los programas ya mencionados, en particular su incapacidad de crear elementos geométricos. Un vertex program recibe un vértice y entrega un vértice, un fragment program pinta o descarta un píxel, pero ninguno de los dos puede generar triángulos u otros elementos.

Es necesario considerar que un programa en Cg *sólo se ejecuta en la GPU*, es decir, no tiene acceso a disco, memoria RAM u otros dispositivos, ya que por sí solo no es una aplicación, así como Cg no es un lenguaje de programación con todas las características de C.

Capítulo 4

Conceptos Geométricos

En este Capítulo se revisan diferentes conceptos geométricos que son necesarios para algoritmos que permitan trabajar con modelos tridimensionales de una forma más completa que la simple graficación. Contiene dos secciones:

Sección 4.1, Geometría Diferencial: Se estudian brevemente las estructuras diferenciales de primer, segundo y tercer orden de una superficie suave. Se incluyen algoritmos para estimar estas estructuras en mallas de triángulos.

Sección 4.2, Curvas de Nivel: Se estudia la definición y extracción de curvas de nivel desde una superficie suave. Se define un algoritmo para extraer curvas de nivel desde una malla de triángulos.

Tanto la geometría diferencial como la extracción de curvas de nivel son esenciales a la hora de expresar formalmente algunas técnicas de rendering no fotorrealista vistas en el Capítulo siguiente.

4.1. Geometría Diferencial

En el contexto de esta memoria, un modelo tridimensional es una malla de triángulos. Se puede asumir que en algunos casos la malla es una aproximación discreta de una superficie suave, es decir, diferenciable. A continuación se enuncian estructuras diferenciales de primer, segundo y tercer orden para superficies suaves así como algoritmos para estimar dichas estructuras en mallas de triángulos. Tal estimación es necesaria pues el cálculo exacto y/o analítico es difícil y costoso computacionalmente.

Un estudio profundo de esta materia se puede encontrar en [dC76], mientras que un estudio breve enfocado a rendering no fotorrealista, en el cual se ha basado el contenido de esta Sección, se puede encontrar en [RDF05].

4.1.1. Estructuras diferenciales de Primer Orden

La primera aproximación que se puede tener de una superficie en un punto $\mathbf{p}$ es el plano tangente en dicho punto (ver Figura 4.1(a)), que se puede definir en términos de la normal $\mathbf{n}$ a la superficie en $\mathbf{p}$. En el caso de una superficie parametrizada $S(s,t)$ la normal se define como:

$$\mathbf{n}(\mathbf{p}) = \frac{\partial S(\mathbf{p})}{\partial s} \times \frac{\partial S(\mathbf{p})}{\partial t}$$

Mientras que en el caso de una superficie implícita $F(\mathbf{p}) = 0$ la normal se define como:

$$\mathbf{n}(\mathbf{p}) = \nabla F(\mathbf{p})$$

Una dirección en el plano tangente en $\mathbf{p}$ puede describirse en base a dos vectores base $\{\mathbf{s}_u, \mathbf{s}_v\}$ contenidos en él. Entonces, un vector tridimensional $\mathbf{x}$ que se ubique en el plano tangente puede ser descrito en base a $\mathbf{s}_u$ y $\mathbf{s}_v$ como $\mathbf{x} = u\mathbf{s}_u + v\mathbf{s}_v$.

4.1.2. Estimación de normales en una malla de triángulos

Usualmente un modelo tridimensional no incluye información sobre las normales en cada vértice, y si es que la incluye, las normales de dos vértices que comparten la misma posición suelen ser diferentes. Esto no es extraño ni erróneo, un ejemplo donde puede suceder es en un poliedro. Estas mallas no pueden ser aproximaciones de superficies suaves ya que presentan discontinuidades. Por lo tanto, en una malla de triángulos que aproxime una superficie suave se requiere que todos los vértices que comparten una posición tengan una misma normal.

Entonces, el problema a resolver es: *¿cómo estimar la normal en una posición dada, para una malla de triángulos?* Si se conoce la parametrización o la representación implícita de la superficie simplemente se deben evaluar las expresiones vistas anteriormente. Sin embargo, en el escenario típico se tienen mallas arbitrarias de las cuales no se conoce más información que los vértices de cada triángulo. Así, es necesario un enfoque que permita estimar las normales de la superficie en cada vértice a partir de la poca información que se tiene.

El procedimiento a llevar a cabo es el siguiente: primero se estiman las normales de cada triángulo por separado, como se observa en la Figura 4.1(b). Luego, en cada punto de la malla se suman con ponderación las normales de los triángulos que contienen a ese punto en sus vértices:

$$n(\mathbf{p}) = \sum_{triángulos} [w_i * \mathbf{n}_i]$$

donde w_i es el peso asignado a la normal del triángulo i. La normal $\mathbf{n}_i$ se calcula a partir del producto cruz de dos aristas del triángulo, ya que en ese triángulo el plano tangente a la superficie es el triángulo mismo. Se puede elegir dos aristas cualesquiera, siempre y cuando tengan un extremo en común y se tome en cuenta la orientación de los puntos. Una elección de aristas típica es la siguiente:

$$n_i = (p_1 - p_0) \times (p_2 - p_1)$$

Los puntos p_0, p_1 y p_2 son las posiciones asignadas a los vértices del triángulo.

Si se considera w_i como el recíproco del total de triángulos que comparten el punto donde se está calculando la normal, entonces $\mathbf{n}(\mathbf{p})$ será simplemente el promedio entre las normales. Este resultado puede ser aceptable para mallas regulares, pero para mallas irregulares no da buenos resultados. Esto vuelve necesario la utilización de pesos que minimicen la distorsión provocada por la irregularidad de la malla.

Una forma de medir tal irregularidad es estimar las normales para una malla de triángulos que modele una esfera: es fácil comprobar la exactitud de las estimaciones porque siempre es posible comparar la estimación con el valor real. En el caso del promedio simple de las normales, el valor estimado depende de la forma de los triángulos. Existe un enfoque que entrega resultados exactos para una esfera, independiente de la forma y distribución de los triángulos, propuesto por [Max99]: se define w_i como el área del triángulo i dividida por los cuadrados de las aristas que comparten el punto $\mathbf{p}$.

El Algoritmo 4.1 muestra el pseudo–código de lo planteado, incluyendo los pesos recomendados.

4.1.3. Estructuras diferenciales de Segundo Orden

La derivada direccional $D_{\mathbf{x}}\mathbf{f}$ de una función $\mathbf{f}$ definida en la superficie muestra cómo esa función varía al avanzar en una dirección tangente $\mathbf{x}$.

Una función que se podría considerar es $\mathbf{n}(\mathbf{p})$, la normal de la superficie. Como su derivada direccional debe ser perpendicular, sus derivadas deben estar dentro del plano tangente definido en $\mathbf{p}$. Por lo tanto, la derivada direccional en la dirección $\mathbf{x}$ de $\mathbf{n}(\mathbf{p})$ se puede escribir como:

$$D_{\mathbf{x}}\mathbf{n} = n_u\mathbf{s}_u + n_v\mathbf{s}_v,$$

donde:

$$\begin{bmatrix} n_u \\ n_v \end{bmatrix} = \begin{bmatrix} L & M \\ M & N \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

Los valores L, M y N dependen de la geometría de la superficie y se pueden obtener evaluando analíticamente $D_{\mathbf{x}}\mathbf{n}$.

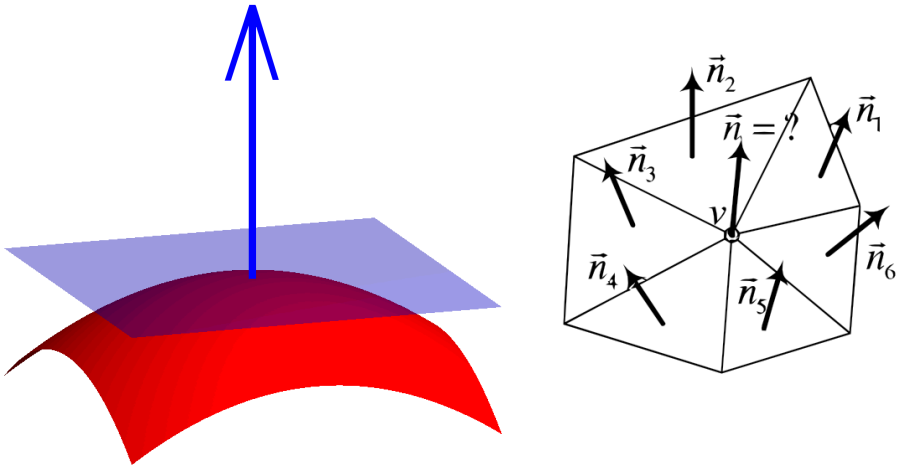
Si se tienen dos vectores tangentes $\mathbf{x}_1 = u_1\mathbf{s}_u + v_1\mathbf{s}_v$ y $\mathbf{x}_2 = u_2\mathbf{s}_u + v_2\mathbf{s}_v$, se define la *segunda forma fundamental* $\mathbf{II}$ en $\mathbf{p}$ como:

$$\begin{aligned} \mathbf{II}(\mathbf{x}_1, \mathbf{x}_2) &= (D_{\mathbf{x}_1}\mathbf{n}(\mathbf{p})) \cdot \mathbf{x}_2 = (D_{\mathbf{x}_2}\mathbf{n}(\mathbf{p})) \cdot \mathbf{x}_1 \\ &= \begin{bmatrix} u_1 & v_1 \end{bmatrix} \begin{bmatrix} L & M \\ M & N \end{bmatrix} \begin{bmatrix} u_2 \\ v_2 \end{bmatrix} \end{aligned}$$

Como $\mathbf{II}$ es simétrica, se utiliza $\mathbf{II}(\mathbf{x})$ como una abreviación para la derivada direccional de la normal. Así, $\mathbf{II}(\mathbf{x}) = D_{\mathbf{x}}\mathbf{n}$ y $\mathbf{II}(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{II}(\mathbf{x}_1) \cdot \mathbf{x}_2 = \mathbf{II}(\mathbf{x}_2) \cdot \mathbf{x}_1$.

La *curvatura normal* de una superficie S en un punto $\mathbf{p}$ mide la curvatura de la superficie en el plano tangente en una dirección $\mathbf{x}$, y es definida en términos de la segunda forma fundamental:

$$\kappa_n(\mathbf{x}) = \frac{\mathbf{II}(\mathbf{x}, \mathbf{x})}{\mathbf{x} \cdot \mathbf{x}}$$



(a) Plano Tangente a la superficie (b) Estimación de la normal en un vértice

Figura 4.1: *Estructuras diferenciales de primer orden: la aproximación del plano tangente en un plano mediante la normal, y la estimación de la normal en un vértice de la malla de triángulos que aproxima una superficie.*

Algoritmo 4.1: Estimación de normales por vértice para una malla de triángulos.

Data: malla de triángulos m
foreach *triángulo* t_i *en* m **do**
  $p_0, p_1, p_2 \leftarrow$ puntos de los vértices de t_i;
  $n_i \leftarrow (p_1 - p_0) \times (p_2 - p_1)$;
  $normalizar(n_i)$;
end
foreach *punto* $\mathbf{p}$ *en* m **do**
  $\mathbf{n}(\mathbf{p_n}) \leftarrow (0, 0, 0)$;
  foreach *triangulo* t_i *que contiene a* $\mathbf{p}$ **do**
    $a_{i_1}, a_{i_2} \leftarrow$ aristas de t_i que contienen a $\mathbf{p}$;
    $\mathbf{n}(\mathbf{p}) \leftarrow \mathbf{n}(\mathbf{p}) + \dfrac{area(t_i)}{||a_{i_1}||^2||a_{i_2}||^2} \times n_i$;
  end
  $normalizar(\mathbf{n}(\mathbf{p}))$;
end

En una superficie suave, si se consideran todas las curvas que pasan por el punto $\mathbf{p}$ se puede observar que la curvatura de todas ellas se encuentra entre los valores κ_1 y κ_2, llamadas *curvaturas principales* en $\mathbf{p}$. Las direcciones tangentes a $\mathbf{p}$ en las cuales se encuentran estas curvaturas son las *direcciones principales* $\mathbf{e}_1$ y $\mathbf{e}_2$. Estas direcciones se pueden apreciar en la Figura 4.2. Como propiedad, $|\kappa_1| \geq |\kappa_2|$.

Si se utilizan las direcciones principales como la base $\{\mathbf{e}_1, \mathbf{e}_2\}$ del plano tangente, se dice que se está trabajando en *coordenadas principales*. Al utilizar coordenadas principales, la matriz de la segunda forma fundamental se vuelve diagonal:

$$\begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix}$$

Esto se debe a que las curvaturas principales son los valores propios de la matriz de la segunda forma fundamental. A su vez, las direcciones principales son los vectores propios de dicha matriz.

Dado $\begin{bmatrix} u & v \end{bmatrix}^T = \begin{bmatrix} \cos\phi & \sin\phi \end{bmatrix}^T$, donde ϕ es el ángulo entre la dirección de interés y $\mathbf{e}_1$, se obtiene la fórmula de Euler para la curvatura normal:

$$\kappa_n(\phi) = \kappa_1 \cos^2\phi + \kappa_2 \sin^2\phi$$

Además, en base a las curvaturas principales se definen la curvatura gaussiana $K = \kappa_1\kappa_2$ y la curvatura media $H = (\kappa_1 + \kappa_2)/2$.

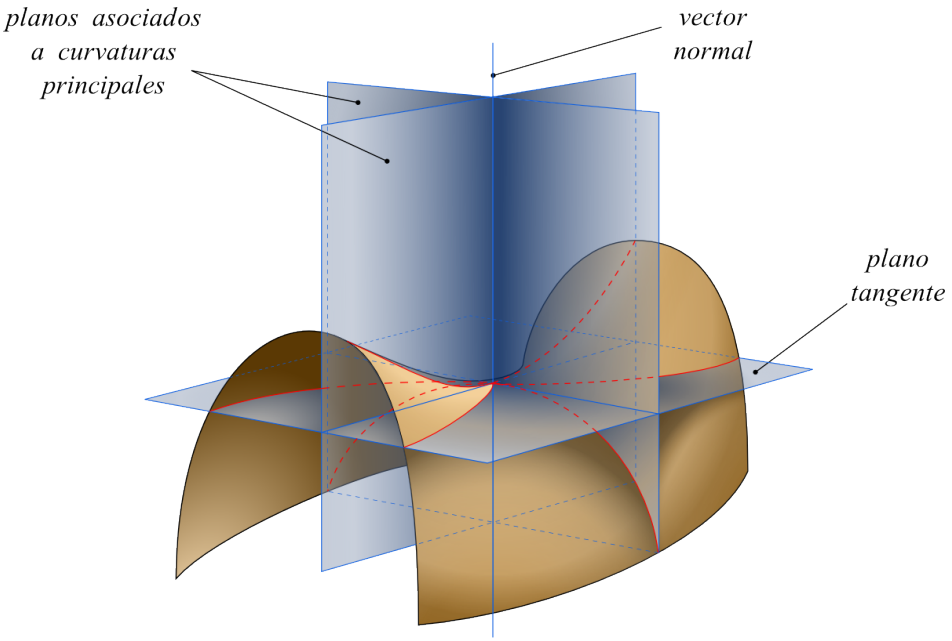


Figura 4.2: *Estructuras diferenciales de segundo orden (curvaturas) para una superficie suave.*

4.1.4. Estimación de curvaturas en una malla de triángulos

Si bien la segunda forma fundamental se puede evaluar analíticamente, es muy difícil implementar una aproximación numérica de ese valor, sobretodo para superficies que prácticamente es imposible parametrizar o expresar implícitamente. Se vuelve necesario entonces

un procedimiento para estimar **II** en mallas de triángulos arbitrarias. En [Rus04] se plantea uno de los enfoques más robustos, aplicando la misma filosofía que se aplica en la estimación de las normales: se estima la segunda forma normal para cada triángulo y luego se ponderan los valores obtenidos para cada posición.

Para estimar **II** en cada triángulo se realiza un cálculo de diferencias finitas. Al conocerse las normales en cada vértice es posible calcular la diferencia entre ellas a través de las aristas. Estas diferencias imponen una serie de condiciones para **II**, con las cuales se puede realizar una estimación mediante mínimos cuadrados:

$$
\begin{aligned}
\mathbf{II}\begin{pmatrix} a_0 \cdot \mathbf{s}_u \\ a_0 \cdot \mathbf{s}_v \end{pmatrix} &= \begin{pmatrix} (n_2 - n_1) \cdot \mathbf{s}_u \\ (n_2 - n_1) \cdot \mathbf{s}_v \end{pmatrix} \\
\mathbf{II}\begin{pmatrix} a_1 \cdot \mathbf{s}_u \\ a_1 \cdot \mathbf{s}_v \end{pmatrix} &= \begin{pmatrix} (n_0 - n_2) \cdot \mathbf{s}_u \\ (n_0 - n_2) \cdot \mathbf{s}_v \end{pmatrix} \\
\mathbf{II}\begin{pmatrix} a_2 \cdot \mathbf{s}_u \\ a_2 \cdot \mathbf{s}_v \end{pmatrix} &= \begin{pmatrix} (n_1 - n_0) \cdot \mathbf{s}_u \\ (n_1 - n_0) \cdot \mathbf{s}_v \end{pmatrix}
\end{aligned}
$$

Donde a_0 es la arista formada por los puntos p_1 y p_2, con normales n_1 y n_2 (a_1 y a_2 se definen de manera análoga). De las ecuaciones se desprende que para cada vector que representa a una arista del triángulo, aplicar la segunda forma normal a ese vector arista debe dar la variación de la normal a través del vector, en concordancia con lo visto en la sección anterior. Los vectores $\mathbf{s}_u$ y $\mathbf{s}_v$ se refieren a un sistema de coordenadas ortonormal propio de cada triángulo, que puede ser calculado a partir de las aristas y de la normal del triángulo.

Una vez que se ha estimado la segunda forma normal para cada triángulo, incluyendo la diagonalización de la matriz, se realiza un procedimiento análogo al de la estimación de normales por vértice. Para cada posición se calcula un promedio con pesos de todas las estimaciones pertenecientes a los triángulos asociados a esa posición. Sin embargo, hay que considerar dos factores que difieren del procedimiento de las normales:

- Los pesos no son los mismos. El peso propuesto por [Rus04] es el recíproco del área del triángulo asociada al vértice, una especie de "área de Voronoi". Este peso permite que las curvaturas estimadas para una esfera sean correctas independiente de la distribución de los triángulos en su superficie.
- Cada triángulo tiene un sistema de coordenadas ortonormal propio. Por consiguiente, en cada punto asociado a los vértices del triángulo en cuestión se debe estimar un nuevo sistema ortonormal al cual se deben proyectar los sistemas de los triángulos. Ahora bien, como los triángulos en general no son coplanares, deben realizarse rotaciones entre ellos para poder realizar la proyección sin perder la proporción entre las diferentes contribuciones de los triángulos.

El Algoritmo 4.2 muestra el pseudo–código asociado este procedimiento.

4.1.5. Estructuras diferenciales de Tercer Orden

Por estructuras diferenciales de tercer orden se entenderán la derivada direccional de la curvatura normal $D_{\mathbf{x}}\kappa_n$ en la dirección $\mathbf{x}$ y el gradiente $\nabla\kappa_n$ de la curvatura normal. Al ser operaciones conocidas, su cálculo analítico se desprende de las definiciones de derivada direccional y de gradiente.

Cuando se usan coordenadas principales, el cálculo se estas estructuras se vuelve más simple. Encontrar las derivadas de la curvatura normal equivale a encontrar la derivada direccional de **II** en una dirección tangente $\mathbf{x}$. El resultado es una *forma trilineal simétrica*, denominada por notación $\mathbf{C}$, que permite calcular el cambio en la curvatura a través de la superficie. Está compuesta por un tensor de $2 \times 2 \times 2$ (una "matriz de matrices") cuyas componentes dependen de las terceras derivadas de la superficie:

$$\mathbf{C} = \begin{pmatrix} D_{\mathbf{s}_u}\mathbf{II} & D_{\mathbf{s}_v}\mathbf{II} \end{pmatrix}$$

Esta matriz trilineal se puede escribir con dos o tres argumentos, lo que indica el número de veces que un vector $\mathbf{x}$ es multiplicado por el tensor. Así, $\mathbf{C}(\mathbf{x}, \mathbf{x})$ es un vector (la derivada direccional de **II** en la dirección $\mathbf{x}$) y $\mathbf{C}(\mathbf{x}, \mathbf{x}, \mathbf{x})$ un escalar (la derivada de la curvatura normal en la dirección $\mathbf{x}$). Al trabajar en coordenadas principales, el tensor que describe $\mathbf{C}$ tiene cuatro componentes:

$$\mathbf{C} = \left(\begin{pmatrix} P & Q \\ Q & S \end{pmatrix} \begin{pmatrix} Q & S \\ S & T \end{pmatrix} \right)$$

Donde $P = D_{\mathbf{e}_1}\kappa_1$, $Q = D_{\mathbf{e}_2}\kappa_1$, $S = D_{\mathbf{e}_1}\kappa_2$ y $T = D_{\mathbf{e}_2}\kappa_2$. Así, el gradiente de la curvatura normal se expresa como sigue:

$$\nabla\kappa_n(\mathbf{x}) = \frac{\mathbf{C}(\mathbf{x}, \mathbf{x})}{\mathbf{x} \cdot \mathbf{x}} = \frac{g_u\mathbf{e_1} + g_v\mathbf{e_2}}{\mathbf{x} \cdot \mathbf{x}}$$

Donde:

$$\begin{bmatrix} g_u \\ g_v \end{bmatrix} = \begin{bmatrix} Pu^2 + 2Quv + Sv^2 \\ Qu^2 + 2Suv + Tv^2 \end{bmatrix}$$

Por su parte, la derivada direccional se puede expresar como:

$$\frac{D_{\mathbf{x}}\kappa_n(\mathbf{x})}{||\mathbf{x}||} = \frac{\mathbf{C}(\mathbf{x}, \mathbf{x}, \mathbf{x})}{||\mathbf{x}||^3} = \frac{Pu^3 + 3Qu^2v + 3Suv^2 + Tv^3}{||\mathbf{x}||^3}$$

4.1.6. Estimación de derivadas de curvatura en una malla de triángulos

La estimación de las derivadas se realiza de forma análoga a la estimación de las curvaturas, como se observa en el pseudo–código del Algoritmo 4.3. Se estima $\mathbf{C}$ para cada triángulo y luego para cada vértice en la malla se suman las contribuciones de los triángulos que lo comparten. La estimación en cada triángulo se realiza del mismo modo anterior, pero en vez de evaluar la diferencia de las normales a través de las aristas, se aproximan las diferencias de **II** a través de ellas.

Algoritmo 4.2: Estimación de curvaturas para una malla de triángulos.

Data: malla de triángulos m
foreach *punto* p *en* m **do**
  construir un sistema de coordenadas (u_p, v_p) inicial en el plano tangente de p;
end
foreach *triángulo* t_i *en* m **do**
  calcular diferencias $\triangle n$ en cada arista;
  estimar **II** utilizando mínimos cuadrados;
  foreach *punto* p *que pertenece a* t_i **do**
    re-expresar **II** en términos de (u_p, v_p);
    $w_{t_i,p} \leftarrow$ área de t_i asociada a p;
    añadir $\mathbf{II} \times w_{t_i,p}$ a la curvatura asociada a p;
  end
end
foreach *punto* p *en* m **do**
  dividir **II** por la suma de los pesos acumulados (normalización);
  calcular curvaturas y direcciones principales (calculando valores y vectores propios de **II**;
end

Algoritmo 4.3: Estimación de estructuras diferenciales de tercer orden (derivadas de curvatura) para una malla de triángulos.

Data: malla de triángulos m
foreach *punto* p *en* m **do**
  construir un sistema de coordenadas (u_p, v_p) inicial en el plano tangente de p;
end
foreach *triángulo* t_i *en* m **do**
  calcular diferencias $\triangle \mathbf{II}$ en cada arista;
  estimar **C** utilizando mínimos cuadrados;
  foreach *punto* p *que pertenece a* t_i **do**
    re-expresar **C** en términos de (u_p, v_p);
    $w_{t_i,p} \leftarrow$ área de t_i asociada a p;
    añadir $\mathbf{C} \times w_{t_i,p}$ a la curvatura asociada a p;
  end
end
foreach *punto* p *en* m **do**
  dividir **C** por la suma de los pesos acumulados (normalización);
end

4.2. Curvas de Nivel

En una superficie suave se pueden evaluar diferentes funciones arbitrarias, como pueden ser la altura o las curvaturas ya definidas. Ahora bien, se desea plantear el problema de obtener una o más curvas dentro de la superficie para las cuales la evaluación de la función en cada punto de la curva entregue el mismo resultado. La solución a este problema son las *curvas de nivel*, *isolíneas* o *isocontornos*, y se definen como el lugar geométrico para el cual $f(\mathbf{p}) = n$, siendo n el *nivel* de la curva y f la *función generadora* de la curva de nivel. Ciertamente para algunas funciones en vez de obtenerse curvas se podrían obtener parches de la superficie, pero en general quien desee obtener curvas de nivel elije funciones apropiadas.

4.2.1. Ejemplos

A continuación se enumeran algunos tipos conocidos de curvas de nivel que permiten ilustrar este concepto, visibles en la Figura 4.3.

(a) Mapa Topográfico (b) Curvas Parabólicas (c) Isofotas

Figura 4.3: *Ejemplos de curvas de nivel en un mapa, en un busto del Apollo de Belvedere dibujadas por Felix Klein (fuente: [HCV52]), y en un modelo tridimensional.*

Altura

La altura es un tipo de curva presente en los mapas geográficos, o en general en cualquier tipo de mapa, que permite conocer la altura asociada a la geografía. Esto es muy útil ya que generalmente los mapas son ilustraciones que grafican un lugar desde arriba, de modo que se pierde la noción de altura en los datos. Por ello se vuelve necesario el poder representar esa información de manera anexa en el mapa, y es así como se utilizan, además de otros recursos como el color, las curvas de nivel de la altura para indicar los diferentes niveles que alcanza el lugar representado. En la Figura 4.3(a) se observa un mapa topográfico con una gran cantidad de curvas de nivel que representan la altura. La función generadora es, entonces, $f(\mathbf{p}) = \mathbf{p}_z$.

Curvaturas

En la sección anterior se definieron las curvaturas normal, gaussiana y media en una superficie suave. Ahora bien, ¿qué tipo de curvas de nivel se puede extraer desde esas definiciones? Al contrario de la altura, en general no se desea obtener una serie de curvas de nivel para estas curvaturas, sino que más bien se desea extraer *una* curva de nivel 0. Estas curvas son llamadas curvas parabólicas y representan los lugares en los cuales la superficie tiene curvatura cero, es decir, se puede considerar plana localmente. Usualmente se extraen a partir de la curvatura gaussiana. Es decir, la función generadora es $f(\mathbf{p}) = K(\mathbf{p})$.

En la historia el matemático Felix Klein intentó buscar el sentido estético que tendrían estas líneas, tomando una reproducción del Apollo de Belvedere y graficando las curvas parabólicas sobre la escultura. Lamentablemente las líneas parecían no tener sentido estético y tampoco demarcaban zonas de interés aparente. En la Figura 4.3(b) se observa una fotografía de la escultura con las curvas parabólicas dibujadas en ella.

Isofotas

Las isofotas son curvas de nivel asociadas con la intensidad de iluminación en una superficie. Esto permite segmentar la superficie de acuerdo a la intensidad de iluminación incidente. A diferencia de la altura y de las curvaturas, las isofotas suelen ser curvas que cambian en el tiempo, ya que en la mayoría de los escenarios la geometría y/o la o las fuentes de luz tiene posición variable.

Si bien la intensidad de la iluminación depende del modelo de iluminación elegido, usualmente se utiliza la intensidad de Lambert, es decir, la función generadora es $f(\mathbf{p}) = \mathbf{n}(\mathbf{p}) \cdot \mathbf{l}(\mathbf{p})$, siendo $\mathbf{l}$ el vector unitario que va desde $\mathbf{p}$ hasta la posición de la fuente de luz. Este tipo de curvas de nivel permite marcar las zonas del objeto que están siendo iluminadas, como se observa en la Figura 4.3(c).

Otros ejemplos

En general, cualquier tipo de función que se pueda calcular en una superficie es candidata a generar curvas de nivel. Esto quiere decir que siempre que se tenga una superficie junto con datos a evaluar se puede extraer una curva de nivel. En campos como la meteorología se suelen extraer curvas de nivel para la temperatura, la velocidad del viento, la presión o incluso el grado de precipitaciones. En un modelo tridimensional se puede anexar prácticamente cualquier tipo de información a la malla, por lo que las posibilidades no se limitan a los ejemplos vistos en este Capítulo.

4.2.2. Discretización en Mallas de Triángulos

En la Figura 4.2.2 se observan las curvas parabólicas en la parte superior de un modelo tridimensional de un ángel. Se ha posicionado la cámara de modo de poder comparar las curvas obtenidas en su rostro con las del Apollo de Belvedere. La pregunta que surge es:

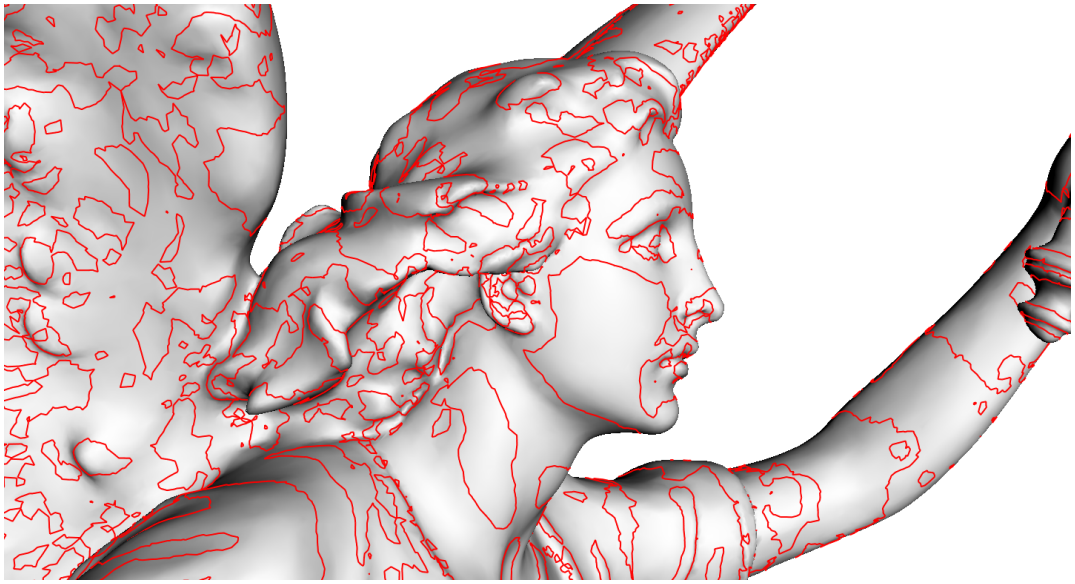


Figura 4.4: *Curvas de nivel* 0 *para la curvatura gaussiana en una modelo tridimensional.*

¿cómo extraer las curvas deseadas a partir de una malla de triángulos? La respuesta depende de la cantidad de datos que se poseen sobre la malla. Como usualmente se posee información solamente en los vértices, se debe plantear un algoritmo que permita extraer las curvas utilizando sólo esa información.

Para plantear un algoritmo de extracción de curvas se deben tener las siguientes consideraciones:

Discretización de una curva: Polilínea Al estar trabajando con discretizaciones, es prácticamente imposible obtener una parametrización o una representación implícita para una curva de nivel. Por lo tanto las curvas también deben discretizarse, es decir, expresarse como una polilínea o una sucesión de puntos en orden de recorrido. La función que se desea utilizar como generadora debe poder evaluarse en cada vértice de la malla. Entonces

Evaluación en todos los puntos La función generadora debe poder evaluarse en todos los puntos de la malla.

Inserción de puntos en la polilínea Se debe determinar cómo se seleccionarán los puntos que formarán la curva. Es muy probable que nunca un punto de la malla sea parte de la curva (por diversos motivos, siendo el más recurrente la precisión en la representación de los números).

La última consideración es sumamente importante debido a que usualmente sólo se trabaja con los vértices y las aristas de los triángulos. En la Figura 4.2.2 se aprecia que las curvas parabólicas tienen una apariencia tosca y cuadriculada, lo que se debe no solamente a la discretización, sino a la presencia de ruido en la malla y a la forma en la que se insertan los puntos en las polilíneas que finalmente serán las curvas de nivel. El procedimiento más común para insertar los puntos es verificar que dentro de una arista se encuentre el nivel buscado,

es decir, para las curvas parabólicas un extremo de la arista debe cumplir $K < 0$ y el otro $K > 0$. Así, se determina que dentro de esa arista existe un punto que debe insertarse en la polilínea. El punto se calcula mediante interpolación lineal para encontrar una aproximación de la posición en la cual K vale 0.

El problema es que son muy pocas las ocasiones en las cuales la función varía linealmente entre un vértice y otro. Una solución es aplicar otro tipo de interpolación o utilizar los puntos de la polilínea como puntos de control en una curva paramétrica.

Una vez que se obtiene un punto de la polilínea, se debe encontrar otro punto para ingresar en la curva: se "recorre" la malla siguiendo la curva de nivel para obtener más puntos representativos que la discreticen. Este procedimiento que recorre la malla se puede realizar de diversas maneras, a continuación se plantea un algoritmo de recorrido completo de la malla que solamente visita una vez cada triángulo. Luego se comenta un algoritmo aleatorio que tiene resultados menos completos (puede omitir algunas curvas) pero puede presentar mejoras en la velocidad de extracción para mallas muy grandes.

4.2.3. Algoritmo de Recorrido de la Malla

Para extraer las curvas de un cierto nivel desde una malla de triángulos, lo primero que se debe hacer es encontrar en la lista de triángulos de la malla un triángulo en el cual se tenga un punto de partida para la curva. Una vez que ya se encontró un triángulo de interés, mediante la búsqueda en aristas ya mencionada, se elige como siguiente triángulo a revisar el vecino del triángulo de interés a través de la arista que contiene el punto inicial de la curva. En ese nuevo triángulo de interés se revisa si en sus otras aristas la curva de nivel continúa. Si se encuentra uno, se repite el proceso; si no, se sigue buscando en la lista de triángulos por otro punto de partida para una nueva curva.

Es importante el mantener un registro de los triángulos visitados, con el fin de evitar revisar un triángulo dos veces. Esto asegura que cada triángulo de la malla es visitado una sola vez. Por otro lado, en mallas muy grandes el proceso puede tomar demasiado tiempo.

El Algoritmo 4.4 presenta el pseudo–código de este procedimiento. Se recalca que sólo se consideran triángulos que tienen dos aristas de interés porque, en el caso de tener una, no hay por dónde seguir la curva de nivel, y el caso de tres aristas de interés es imposible. Se ha omitido el chequeo del nivel exacto en los vértices puesto que sólo complicaría el planteamiento del algoritmo.

4.2.4. Algoritmo Aleatorio

El algoritmo anterior entrega todas las líneas de contorno requeridas de una malla, pero su ejecución puede ser muy costosa en mallas de gran tamaño. Dicho costo puede no ser aceptable para el caso del rendering en tiempo real, donde muchas veces se requiere extraer curvas de nivel para funciones cuyos valores dependen del momento en el que se ejecutan, por lo que en cada cuadro de la animación del programa es necesario volver a extraer las curvas. En este escenario el algoritmo anterior sólo es apropiado para mallas pequeñas.

Algoritmo 4.4: Extracción de curvas de nivel que recorre toda la malla de triángulos.

```
Input: malla de triángulos m
Input: función generadora f
Input: nivel n
Output: lista de curvas de nivel curvas
curvas ← lista vacía de curvas de nivel;
foreach punto p en m do
    evaluar f(p);
end
foreach triángulo t_i en m do
    encontrar dos aristas que tengan un punto de nivel n;
    if existen dos aristas de interés y t_i no se ha visitado then
        c_actual ← nueva polilínea;
        a_inicio ← arista inicial de interés;
        a_interés ← segunda arista de interés de t_i;
        insertar punto de nivel n de a_inicio en c_actual;
        insertar punto de nivel n de a_interés en c_actual;
        marcar t_i como visitado;
        t_k ← vecino de t_i por a_interés;
        while t_k no ha sido visitado do
            if t_k no tiene dos aristas de interés then
                marcar t_k como visitado;
                break;
            end
            a_interés ← segunda arista de interés de t_k;
            insertar punto de nivel n de a_interés en c_actual;
            marcar t_k como visitado;
            t_k ← vecino de t_k por a_interés;
        end
        insertar c_actual en curvas;
    else
        marcar t_i como visitado;
    end
end
```

Como toda técnica de tiempo real, a veces es necesario sacrificar la certeza o la completitud en la extracción con el fin de obtener un mejor desempeño. Así, se puede utilizar un algoritmo aleatorio, propuesto por [MKG+97], que no recorre todas las caras y presenta resultados satisfactorios. Este algoritmo utiliza la misma idea de recorrer la malla siguiendo un camino a través de las aristas, pero sólo considerando un subconjunto del total de los triángulos para iniciar el recorrido. Es decir, en vez de buscar en toda la malla por triángulos en los cuales encontrar un punto de partida de la curva, se busca sólo en un conjunto inicial.

En un comienzo se pueden perder muchas curvas debido a la elección de los triángulos iniciales, pero se puede recurrir a otras técnicas para mejorar el resultado. Una de ellas es guardar los triángulos que fueron de interés en la ejecución anterior, ya que con alta probabilidad volverán a serlo.

Capítulo 5

Rendering No Fotorrealista – NPR

Este capítulo está dedicado por completo al Rendering No Fotorrealista, o NPR.

Sección 5.1, La Abstracción como fundamento de NPR: Se analiza la abstracción, o *"no fotorrealismo"* en un contexto fuera de la Computación Gráfica. Se define NPR de un modo más formal.

Sección 5.2, Trazado y Extracción de Líneas: Se analiza el NPR desde el trazado y dibujado de líneas para representar modelos tridimensionales.

Sección 5.3, Graficación Abstracta de los Objetos: Se analiza el NPR desde el pintado interior de los modelos.

Sección 5.4, Coherencia Temporal: analiza la coherencia temporal, un término que norma la graficación de estilos no fotorrealistas para animaciones y deformaciones en los objetos.

Sección 5.5, Bibliotecas y Aplicaciones: Se describen aplicaciones que demuestran algunas técnicas de NPR para el trazado de líneas.

5.1. La Abstracción como fundamento de NPR

El término *rendering* está completamente ligado a la Computación Gráfica, y su aplicación fuera del contexto de esa área no tiene mucho sentido. Sin embargo, el *no fotorrealismo* puede ser llamado **abstracción** en un contexto artístico. Según la Real Academia Española, abstracción es *el efecto de abstraer*, y *abstraer* se define como "separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción".

La abstracción ha tenido un gran papel en la vida del hombre, tanto en el arte como en la ingeniería u otras áreas, sean científicas o humanistas. En esta Sección se hará un análisis breve del fotorrealismo y la abstracción en el arte y en la ingeniería, con el fin de enlazar esos conceptos con la Computación Gráfica.

Parte de esta Sección se ha basado en la charla "Realistic or Abstract Imagery: The Future of Computer Graphics?" de Pat Hanrahan[1]

5.1.1. Abstracción en el Arte

El término *Trompe-l'œil* significa, en francés, "engañar al ojo". Ese término se aplica a las pinturas que no solamente quieren retratar algo, sino más bien hacer creer que son una realidad. Es típico ver este tipo de pinturas en el techo de las capillas europeas; se dice que la obra más importante de este tipo es "La gloria de San Ignacio", presente en la Iglesia de San Ignacio en Roma e ilustrada en la Figura 5.1. Al centro de la iglesia existe un punto desde el cual la imagen adquiere un realismo acentuado por la perspectiva y la distancia al techo del lugar.

Figura 5.1: *"La gloria de San Ignacio", de Andrea Pozzo, 1685–1694.*

A lo largo de la historia los ejemplos de pinturas y obras realistas son tan numerosos como impresionantes. No se puede negar que no tendría sentido tener obras abstractas en los techos de las iglesias, o a la hora de hacer retratos (los mejores retratos además de fotorrealistas expresan mucho), pero se vuelve necesaria la abstracción para contar otras cosas y mostrar otros momentos. Surge la siguiente pregunta: ¿hasta dónde puede llegar el fotorrealismo?

El fotorrealismo sólo puede avanzar en una dirección: *imitar la realidad*. La abstracción, por otro lado, tiene un camino mucho más libre: puede trabajar con el *lenguaje* o con la *representación pictórica*, como describe [McC98] y lo ilustra en la Figura 5.2. Así, aparte de los fotorrealistas, existen artistas que se han decantado totalmente por la representación pictórica, mientras que otros lo han hecho solamente por el lenguaje, y los demás han mezclado las tres direcciones, lo cual junto a las distintas técnicas y medios para *realizar* una obra han dado por resultado una gran cantidad de estilos y tendencias. Esto confirma el énfasis en *comunicar* del que se hablaba en la Introducción de esta memoria.

[1]Visible en `http://www.graphics.stanford.edu/~hanrahan/`.

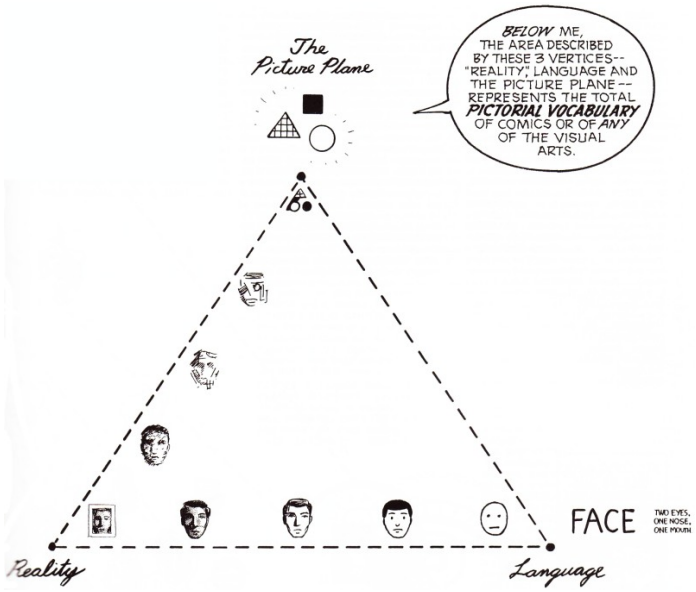


Figura 5.2: *Relaciones entre la realidad, el lenguaje y lo pictórico: los diferentes caminos que puede tomar la abstracción. Fuente: [McC98].*

Figura 5.3: *Obra de Vincent Van Gogh,* "A Starry Night"*, 1889.*

Un ejemplo de abstracción y expresión es la pintura "La Noche Estrellada" de Vincent Van Gogh, en la Figura 5.3, de la cual Carolina Edwards, pintora chilena, dice lo siguiente [Edw07]: "A Van Gogh le atraía especialmente la noche, cuando la luz de la luna y las estrellas parece ser capaz de despertar la naturaleza adormecida e inmóvil. En su obra, la naturaleza lo envuelve todo, dejando al ser humano pequeño e indefenso, abrumado ante una fuerza tan grandiosa, imposible de controlar".

Ciertamente esta apreciación del arte es subjetiva, pero la expresividad y su vigencia es innegable. Ahora bien, no solamente hay que considerar la abstracción como un punto importante, también hay que pensar y cuestionar *qué* es realmente una **representación** y *cuál* es su relación con lo *representado*. En esta postura se encuentra a René Magritte que, siendo fotorrealista, logra un cuestionamiento inigualable, en particular con una de sus pinturas más conocidas, titulada "La traición de las imágenes"(*La trahison des images*, Figura 5.4), donde el espectador ve una pipa que bien podría ser la suya, y sin embargo hay un mensaje que dice, célebremente, "*esto no es una pipa*".

Figura 5.4: *"La traición de las imágenes", de René Magritte, 1928–1929.*

El tener presente esa relación entre un objeto, una sensación, una emoción o un sentimiento, y su representación, es lo que permite a un artista plasmar su visión en una obra. El fotorrealismo es una forma de hacerlo, tan válida como la abstracción, pero con distintos propósitos y posibilidades. En la Computación Gráfica el fotorrealismo todavía no alcanza el nivel de perfección que tiene en el arte tradicional, donde incluso se utilizan fotografías. Lamentablemente, la evolución de la tecnología rápidamente deja obsoleto lo que en un instante se conoce como fotorrealismo en Computación Gráfica: lo que en los años '90 era la máxima expresión del fotorrealismo, hoy, en el siglo 21, es una representación básica e ineficiente. La abstracción, por otro lado, es *a-temporal*. La obra de Van Gogh que se ha mostrado es del año 1889 y su expresión, su valor y su calidad han perdurado, y lo seguirán haciendo, en el tiempo y en la memoria de los hombres.

5.1.2. Abstracción en la Ciencia

James Watt fue un ingeniero en la revolución industrial con grandes aportes a los motores de vapor. Pero además de ello propuso una nueva forma de diseñar máquinas, mediante la ilustración técnica. En esa época era usual que los mismos ingenieros ensamblaran sus motores, lo cual era poco eficiente. Watt tuvo la idea de diseñar las piezas de sus máquinas en papel, para que otros las construyeran de acuerdo a sus especificaciones y así poder armarlas después. La idea fue tan buena que su empresa se especializó en el diseño ilustrado de máquinas.

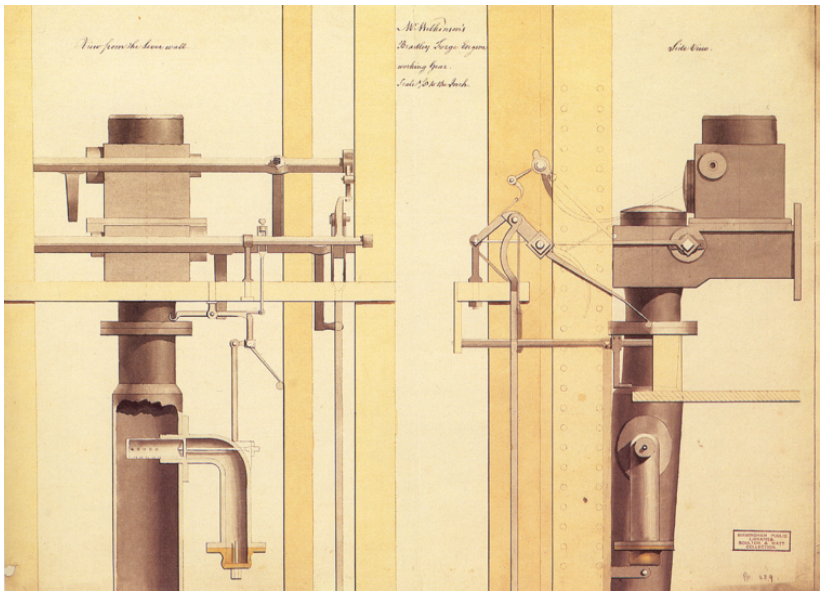


Figura 5.5: *Ilustración técnica de una parte del motor Bradley Forge de John Wilkinson, realizada por James Watt el año 1792.*

En la Figura 5.5 se observa una ilustración de James Watt realizada el año 1792. Se observa que, a pesar de tener más de 200 años, la ilustración perfectamente podría ser actual: se utilizan técnicas vigentes hasta el día de hoy. Ciertamente la ilustración no es fotorrealista, pero sin embargo logra comunicar a la perfección cómo es la pieza que se debe construir: con los colores se indican los materiales, y las dos perspectivas utilizadas permiten dimensionar la pieza, que se encuentra a escala.

La ilustración técnica no solamente se usa en la especificación de piezas mecánicas o motores. En la arquitectura las aplicaciones de CAD no suelen utilizar gráficos realistas, más bien son simplificaciones que permiten tener una idea de los espacios y de las condiciones de los lugares, pero no previsualizar su apariencia bajo las condiciones de luz reales. En las publicaciones médicas se suelen encontrar ilustraciones de órganos y partes del cuerpo humano que no son realistas, precisamente porque no buscan dar a conocer esos órganos visualmente, sino más bien, acentuar las características importantes: abstraer la información.

5.1.3. Definición de NPR

Una vez definidos los conceptos elementales del rendering tradicional y los conceptos geométricos necesarios, se puede comenzar a definir formalmente lo que es el *rendering no fotorrealista*, *NPR*. El NPR toma todos los conceptos anteriores y, a través de ellos, en un *proceso de rendering que extiende o reemplaza al tradicional, genera imágenes con algún nivel de abstracción a partir de una escena tridimensional.*

La investigación y técnicas de NPR se pueden dividir en dos grandes áreas, mencionadas en el Capítulo 1. La primera es el **trazado y extracción de líneas a partir de modelos tridimensionales**, y la segunda es la **graficación abstracta de los objetos**.

5.2. Trazado y Extracción de Líneas

El trazado y extracción de líneas es un área en NPR que no solamente se dedica a *graficar* de manera abstracta un modelo tridimensional, sino que también extrae *información* de ellos. Las líneas por sí solas son suficientes para que el ser humano comprenda la forma y las cualidades de un objeto, y permiten establecer áreas de interés, de modo de segmentar el objeto e indicar límites que, posiblemente, no sean visibles directamente en una imagen fotorrealista.

La problemática principal en esta área es, entonces, *¿cuáles líneas dibujar y extraer?* En esta Sección se describirán algunos tipos de líneas propuestos por la literatura. A fin de compararlos y graficarlos claramente, para cada uno de ellos se mostrará el mismo modelo tridimensional desde dos puntos de vista diferentes. La versión original, con iluminación utilizando reflexión de Lambert, se puede observar en la Figura 5.6.

Figura 5.6: *Modelo tridimensional graficado con reflexión Lambertiana. Se muestran dos vistas parciales del modelo.*

Los siguientes contornos están definidos en su mayoría para superficies suaves, y su extracción se puede realizar con los algoritmos vistos en el Capítulo anterior. El estudio de estos contornos permite establecer características comunes que definan la modelación en clases dentro del marco de trabajo de un tipo de contorno genérico que sirva de base para definir un tipo de contorno específico.

5.2.1. Siluetas

La silueta de un objeto es la primera línea de contorno importante que se puede definir, ya que es la más sencilla de calcular y extraer. Se define como las curvas de nivel 0 de la función $f(\mathbf{p}) = \mathbf{n}(\mathbf{p}) \cdot \mathbf{v}(\mathbf{p})$, donde $\mathbf{n}(\mathbf{p})$ es la normal en el punto $\mathbf{p}$ y $\mathbf{v}(\mathbf{p})$ es el llamado *vector de vista*, un vector unitario que se dirige desde $\mathbf{p}$ hasta la posición $\mathbf{c}$ de la cámara o punto de vista, si se utiliza proyección en perspectiva (en caso de utilizar proyección ortogonal se utiliza el vector al plano de visión).

Gráficamente, las siluetas son las curvas que se encuentran en el límite entre las áreas visibles y las no visibles de un objeto. Debido a esta definición tan simple, es posible extraerlas en superficies no suaves, como puede ser un poliedro, verificando en cada arista si los dos triángulos que la comparten tienen distinta visibilidad (uno visible y el otro no).

Las siluetas, si bien ayudan a comprender el perfil de un objeto, y en caso de objetos con concavidades, algunos detalles de su interior, en general no proveen información suficiente sobre las formas interiores de un objeto. En la Figura 5.7 se observan las siluetas de un modelo tridimensional, marcadas con color verde.

Figura 5.7: *Silueta de un modelo tridimensional.*

5.2.2. Crestas y Valles

O *Ridges & Valleys* [OBS04]. Corresponden a **máximos** y **mínimos** locales de la curvatura normal, respectivamente. A diferencia de las siluetas, estas curvas no dependen del punto de vista, por lo que basta calcularlas una única vez. Precisamente ésa es su mayor desventaja para la *graficación*, ya que una línea que aporta información desde un punto de vista no necesariamente lo hace desde otro. Por otro lado, estos tipos de contorno son utilizados en la segmentación de superficies con buenos resultados. En las Figuras 5.8 y 5.9 se observan las crestas y valles de un modelo tridimensional.

Para extraer las crestas y valles, se debe extraer la curva de nivel 0 para la función $D_{\mathbf{e_1}}\kappa_1$, con la condición $\kappa_1 > 0$ en el caso de las crestas y $\kappa_1 < 0$ en el caso de los valles.

Figura 5.8: *Crestas de un modelo tridimensional.*

Figura 5.9: *Valles de un modelo tridimensional.*

5.2.3. Contornos Sugestivos

O *Suggestive Contours* [DFRS03]. Corresponden a líneas que pueden ser siluetas desde otros puntos de vista cercanos al actual, a líneas que extienden las siluetas actuales, y a puntos de inflexión en la superficie. Se definen como las curvas de nivel 0 para la **curvatura radial** $\kappa_r(\mathbf{p})$. La curvatura radial es la curvatura normal evaluada en la dirección $\mathbf{w}$, definida como la proyección de $\mathbf{v}(\mathbf{p})$ en el plano tangente a la superficie en $\mathbf{p}$.

Como la curvatura radial forma loops cerrados, que no necesariamente tienen un significado estético, se aplica un test de corte: $D_{\mathbf{w}}\kappa_r > 0$. Este test refuerza el concepto de que se dibujen solamente puntos de inflexión positiva en la superficie. Se puede utilizar un *umbral de tolerancia* al realizar este test. En la Figura 5.10 se observan los contornos sugestivos en un modelo tridimensional, marcados con color azul.

Figura 5.10: *Contornos Sugestivos de un modelo tridimensional.*

5.2.4. Líneas de Destacado

O *Highlight Lines* [DR07], donde se definen las líneas de destacado sugestivo (*Suggestive Highlights*) y las líneas de destacado principales (*Principal Highlights*), que de cierto modo complementan a los contornos sugestivos. Su propósito es remarcar zonas de interés visualmente.

Las líneas de destacado sugestivo se definen como las curvas de nivel 0 de κ_r, con el test de corte $D_{\mathbf{w}}\kappa_r < 0$ si $\mathbf{n}\cdot\mathbf{v} > 0$ y $D_{\mathbf{w}}\kappa_r > 0$ si $\mathbf{n}\cdot\mathbf{v} < 0$. Se pueden observar en la Figura 5.11.

5.2.5. Crestas Aparentes

O *Apparent Ridges* [JDA07]. Estos contornos son similares a los contornos sugestivos, ya que consideran la curvatura de la superficie y son dependientes de la vista. Sin embargo, tienen un enfoque totalmente distinto: en las Crestas Aparentes no se trabaja directamente con la curvatura de la superficie, sino que se considera la llamada *curvatura dependiente de la vista*. Estas líneas aparecen en los lugares donde, de acuerdo al punto de vista, se producen los máximos en la variación de la curvatura. Se pueden observar en la Figura 5.12.

Si se define la máxima curvatura dependiente de la vista en un punto de la superficie como q_1, y la dirección principal de la máxima curvatura como $\mathbf{t}_1$ (es decir, los análogos de κ_1 y $\mathbf{e}_1$), se definen como **Crestas Aparentes** las curvas de nivel 0 para la función $D_{\mathbf{t}_1} q_1$.

5.2.6. Extracción de Aristas de Interés

Exceptuando las siluetas, todos estos contornos están definidos para superficies suaves. Entonces, ¿qué sucede con las superficies no suaves, en particular con objetos como poliedros? Se pueden extraer las aristas cuyos triángulos formen un ángulo mayor a un umbral definido por el desarrollador, lo cual puede aproximar las crestas y valles de la superficie. En general, esta técnica no da buenos resultados.

5.2.7. Graficado de Líneas

Una vez que ya se tienen las líneas y contornos de interés, se puede proceder a graficarlas de acuerdo a alguna técnica especial. Después de la extracción, se posee un conjunto de polilíneas, donde, a su vez, cada una de ellas es un conjunto de puntos que debe recorrerse en el orden (estos puntos pertenecen a las aristas de la malla del modelo) que fueron extraídos.

Cada polilínea se puede graficar con las primitivas de líneas de OpenGL, o bien pueden ser utilizadas con curvas paramétricas, de acuerdo a lo que se desee obtener. En el caso de las siluetas, se recomienda utilizar curvas de Catmull-Rom, mientras que en los otros tipos de contorno se recomienda utilizar B-Splines debido a la presencia de ruido en las mallas, que se ve acentuado en las derivadas de segundo orden. Luego, los puntos ya interpolados se pueden graficar con las mismas primitivas de líneas.

Ahora bien, en ocasiones no solamente se desea graficar las líneas utilizando las primitivas disponibles, sino también *darles estilo*, como puede ser la emulación de un pincel, de un trazo entintado a mano alzada o con lápiz grafito. Todo ello es posible, aunque hay que tener en cuenta que la extracción de las líneas es costoso, por lo que probablemente no hay mucha libertad en la estilización.

Para ello existen varios enfoques:

- Se puede asumir un ancho constante del trazo, de modo de aplicarle una textura que puede simular distintos medios [NM00].
- En distintos puntos de las curvas se puede definir un grosor para la línea. En [GVH07] ese grosor se define de acuerdo a la distancia a las isofotas en la imagen. Los resultados tienen una apariencia visual muy similar a la de un entintado de cómic.
- Una técnica más sencilla pero similar es la de dar grosor a las líneas de acuerdo a la curvatura de ellas (no confundir con la curvatura de la superficie). Esta técnica se propone y define en [SKCN03].

En general, la estilización de líneas puede quedar a total libertad del desarrollador, con la condición de saber elegir bien cómo hacerlo sin afectar el desempeño del programa.

Figura 5.11: *Líneas de Destacado Sugestivo de un modelo tridimensional.*

Figura 5.12: *Crestas Aparentes de un modelo tridimensional.*

5.2.8. Problemas a resolver

Un problema abierto es la detección de contornos para geometría que se deforma en el tiempo, es decir, para modelos animados. En [DFR04] se dan algunas nociones para los contornos sugestivos en el caso del movimiento de cámara, lo cual se puede aplicar a una geometría que se deforma pero con una cámara fija.

Por otro lado, el dibujo de líneas no está limitado a las superficies. También se puede aplicar a datos de volumen: una investigación al respecto se puede encontrar en [BKR+05].

5.3. Graficación Abstracta de los Objetos

Se podría decir que la extracción de líneas se acerca bastante al vértice del lenguaje en la pirámide de la abstracción. En NPR existen otras técnicas que se acercan más al vértice del plano pictórico que al del lenguaje. En esta Sección se revisarán algunas técnicas que se mueven entre estos dos vértices. Se mostrará que, en gran parte, se implementan utilizando técnicas de rendering tradicional o modificando conceptos tradicionales, por lo que es factible implementarlas en tiempo real.

5.3.1. Cel-Shading: dibujos animados

El nombre **cel-shading** [LMHB00] proviene de las micas de plástico que se utilizan en la animación tradicional, llamadas *cels*. Esta técnica también es llamada *toon shading*, debido a que el sombreado de la intensidad de iluminación está inspirado en el pintado característico que recibían las micas.

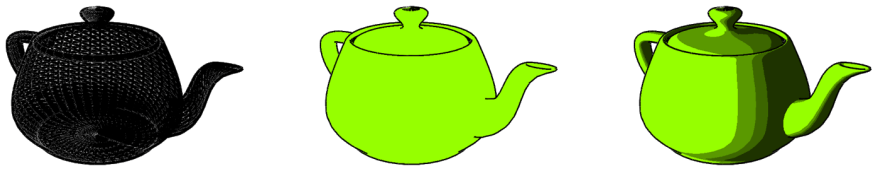


Figura 5.13: *Ejemplo de aplicación de* Cel-Shading *al modelo de la tetera de Utah.*

En la realidad, la intensidad de iluminación varía suavemente desde un punto de una superficie hasta otro, a menos que haya alguna discontinuidad en la superficie, lo cual se puede interpretar como "la intensidad de iluminación en una superficie es una función continua". En *toon shading* se discretiza la intensidad de iluminación que se aplica al color, utilizando una función por pasos (*step function*) en vez de una función continua con valores entre 0 y 1. El modelo de iluminación que se utiliza es el de Phong y se puede utilizar sombreado de Gouraud o de Phong.

La discretización de la intensidad de iluminación aplicada al color se puede interpretar como colorear con la misma intensidad las áreas de la superficie que se encuentren entre las

isofotas de los niveles definidos en la función por pasos.

Además, los dibujos animados tienen remarcados sus contornos. En general, en *toon shading* se grafican solamente las siluetas, pero los contornos sugestivos o las crestas aparentes pueden ser un buen complemento. En la Figura 5.14 se observan varias alternativas aplicadas al conejo de Stanford.

Esta técnica se puede implementar tanto utilizando *Cg* (u otro lenguaje para programar la GPU) como en el *fixed pipeline*. Sin embargo, utilizar la GPU es mucho más intuitivo y eficaz, ya que la segunda opción realiza cálculos redundantes y tiene un mayor costo en memoria. Sólo se indicará la implementación de la primera alternativa.

Lo primero que se debe hacer es codificar la función por pasos en una textura 1D. Una textura 1D es el equivalente a una imagen 2D con un píxel de altura, y se utiliza frecuentemente para guardar una serie de valores, como en este caso. Esta textura es adicional a las posibles texturas del modelo, y no se necesitan coordenadas de textura especificadas en cada vértice, ya que la aplicación de la textura no es fija. Lo que se hará es *evaluar* el modelo de iluminación y utilizar ese resultado como una coordenada de textura para así obtener el valor adecuado en la función por pasos. Una textura 1D de ejemplo se observa en la Figura 5.15.

Se pueden utilizar dos texturas 1D, una para la componente difusa de la iluminación y otra para la componente especular. La coordenada de textura se calculará del siguiente modo:

$$\begin{aligned} t_d &= N \cdot L \\ t_s &= R \cdot V \end{aligned}$$

Donde t_d es la coordenada de la textura difusa y t_s es la coordenada de textura especular. Se asume que los vectores están todos normalizados.

Una vez que ya se haya evaluado el modelo de iluminación de ese modo, la intensidad en un punto se calcula como [FK03]:

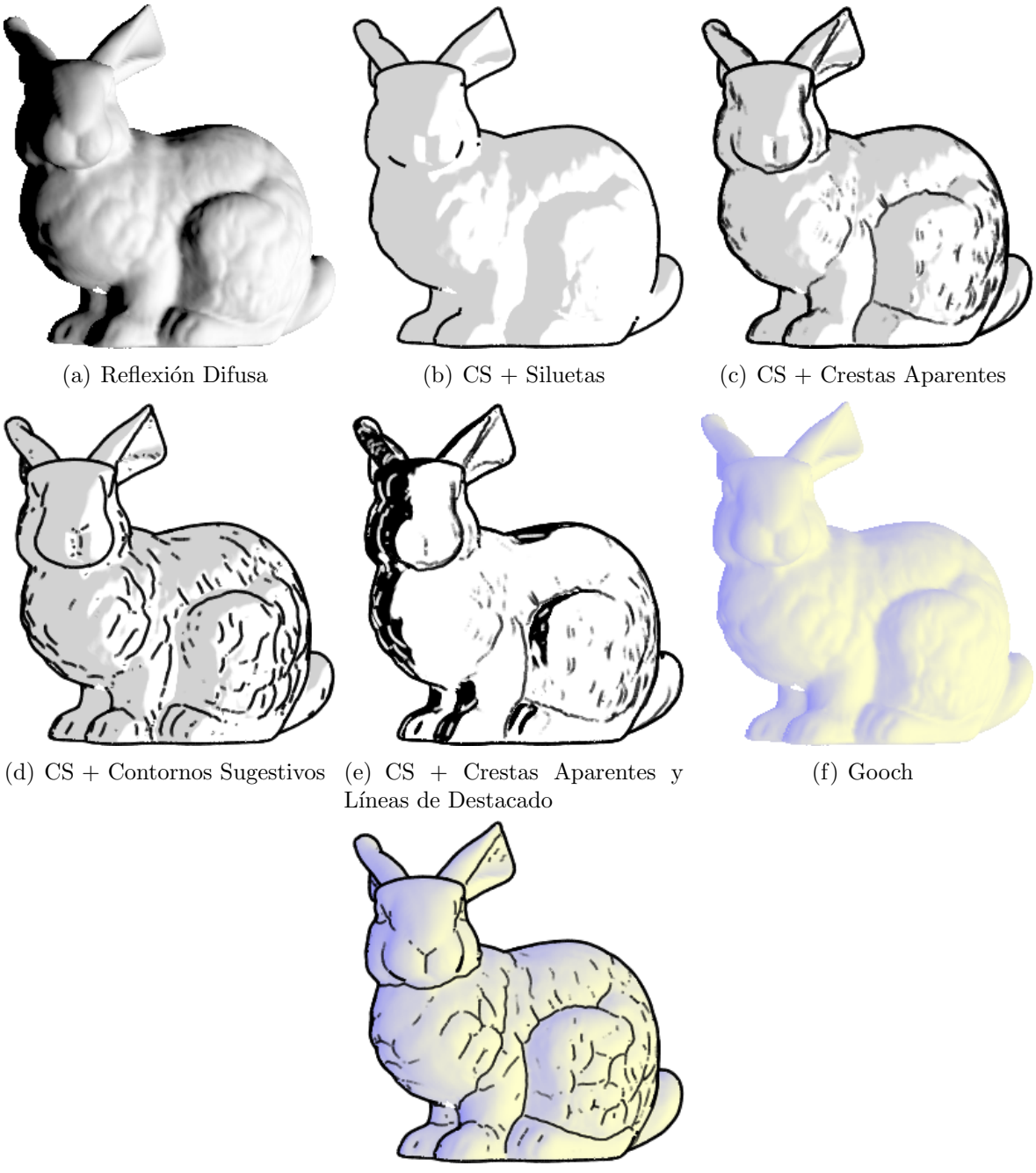
$$I(\mathbf{p}) = I_a K_a + \sum_{luces} [I_d K_d * \texttt{tex1D}(dif, t_d) + I_s K_s * \texttt{tex1D}(spec, t_s)^n]$$

La función `tex1D` recupera el valor de una textura 1D a partir de la coordenada dada. Los parámetros dif y *spec* son las texturas 1D. Esta función funciona del siguiente modo, considerando como ejemplo una textura con valores [0,3, 0,3, 0,6, 0,6, 1,0]:

- Si la coordenada de textura es (0,5) el valor que se obtendrá es 0,6.
- Si es (0,0), el valor es 0,3.
- Si es un valor intermedio, como puede ser (0,9), se realiza una interpolación entre los valores de la textura.

De ello se concluye que la iluminación que se utiliza en *toon shading* es un modelo de iluminación de Phong simplificado.

El siguiente paso es dibujar las líneas sobre el modelo. Existen diferentes alternativas, en orden de complejidad:



(a) Reflexión Difusa (b) CS + Siluetas (c) CS + Crestas Aparentes

(d) CS + Contornos Sugestivos (e) CS + Crestas Aparentes y Líneas de Destacado (f) Gooch

(g) Gooch + Valles

Figura 5.14: *Estilos gráficos de* cel-shading *(CS, en tonos grises, dando un dibujo claro, y negros, dando un dibujo estilo cómic) y* gooch shading *para el conejo de Stanford. Imágenes obtenidas con el software RTSC (Sección 5.5.1).*

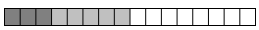


Figura 5.15: *Textura 1D en su representación visual. Cada casilla corresponde a un píxel. También se le puede ver como un arreglo de valores flotantes entre* 0 *y* 1.

- Dibujar dos veces el modelo. La primera, se escala por un factor mayor a 1 y se grafica todo el modelo de color negro. La segunda vez que se dibuje, las siluetas quedarán marcadas y su grosor dependerá de la escala aplicada en el primer paso. Este método es rápido y muy fácil de implementar, pero el resultado visual puede ser de mala calidad.
- Al evaluar el modelo de iluminación, verificar el valor de $N \cdot V$. Si es cercano a 0, descartar la intensidad de iluminación y pintar con negro el píxel. Esto remarcaría las zonas del modelo que sean cercanas a las siluetas, pero sólo funciona correctamente para superficies suaves. El resultado visual depende de la forma del modelo, pero se implementa fácilmente y no se requieren graficaciones extras de la geometría.
- Extraer el *Z-Buffer*, que puede ser interpretado como una imagen en escala de grises a la cual se le aplican filtros para detectar bordes buscando discontinuidades. Esto entrega más que las siluetas y visualmente tiene buenos resultados, pero como contraparte, aplicar filtros de imagen no es barato y no se trabaja con información de la geometría, por lo que no se puede realizar un post-proceso de las líneas. El resultado se puede observar en la Figura 5.16.
- Extraer los contornos adecuados para el modelo tridimensional que se está graficando utilizando las técnicas mencionadas en la Sección anterior, mediante los algoritmos de extracción de curvas de nivel. Esta técnica es la más costosa pero a su vez es la que permite mayor flexibilidad, además de permitir obtener los mejores resultados visuales.

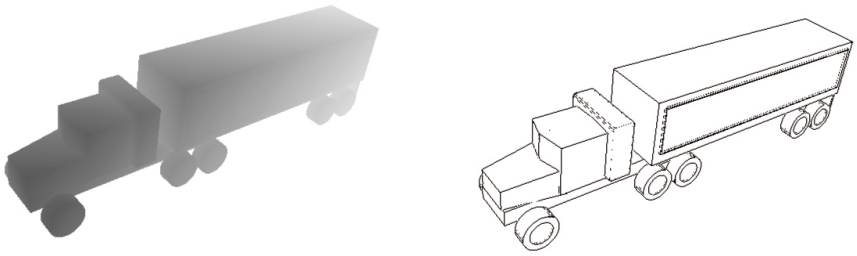


Figura 5.16: *Imagen de Z-Buffer para un modelo tridimensional y la extracción de bordes mediante un filtro de imagen.*

La última alternativa puede entregar los mejores resultados, pero no siempre es la más adecuada. La decisión debe depender de las características de lo que se quiera graficar, del contexto (¿el propósito es calidad visual o velocidad en el rendering, o ambos?) o de condiciones sobre la geometría (¿el modelo tiene animaciones?¿será estático siempre?¿la cámara cambiará de posición constantemente?). Independientemente de la elección, esta técnica se caracteriza por generar imágenes que son agradables y llamativas, por lo que son ideales para animaciones, videojuegos e ilustraciones no técnicas.

5.3.2. Modelos de Tonos

La técnica anterior es un replanteamiento del modelo de iluminación, complementado con el dibujo de líneas. Otra posibilidad que se ha planteado es la de reemplazar el modelo de iluminación en vez de modificarlo, no con el fin de iluminar un objeto, sino más bien no perder información sobre él y aún así dar a conocer la forma que éste tiene (si se suprime la iluminación se pierde gran parte de la información sobre la forma de su superficie).

Inspirados por la ilustración técnica, en [GGSC98] se propone esta idea, donde en vez de un modelo de iluminación se utiliza un modelo de tonalidades para representar zonas frías y cálidas de un objeto en base a la iluminación. Además se complementa con el dibujo de líneas de interés, en particular siluetas, crestas y valles, aunque éstas últimas son extraídas a partir de aristas. En la Figura 5.17 se observa una pieza mecánica graficada con esta técnica, conocida como **Gooch Shading** en honor a sus creadores. En la Figura 5.14 se muestra esta técnica aplicada al conejo de Stanford.

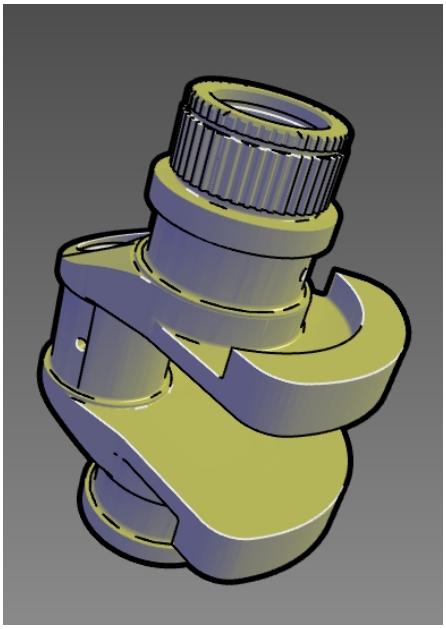


Figura 5.17: *Ilustración de una pieza mecánica utilizando un modelo de tonos y destacando líneas de interés en el modelo.*

El modelo de tonos propuesto consiste en la siguiente evaluación del modelo de iluminación:

$$I(\mathbf{p}) = \left(\frac{1 + N \cdot L}{2}\right) k_f + \left(1 - \frac{1 + N \cdot L}{2}\right) k_c$$

Donde k_f corresponde a la tonalidad fría y k_c a la tonalidad cálida. Los autores proponen definir estas tonalidades del siguiente modo:

$$\begin{aligned} k_f &= k_b + \alpha k_d \\ k_c &= k_y + \beta k_d \end{aligned}$$

Donde k_b y k_y corresponden a los colores azul y amarillo, respectivamente, modulados de acuerdo a las necesidades del usuario. Los valores α y β son parametrizables e indican cuánto

deben tomar estas tonalidades frías y cálidas del color difuso del modelo de acuerdo a sus propiedades de material (también podría utilizarse una coordenada de textura si el modelo tuviese una).

La implementación de esta técnica es directa si se ha implementado el modelo de iluminación de Phong o *cel-shading*. Además, para la extracción de líneas se pueden considerar las alternativas vistas en la técnica anterior.

5.3.3. Rendering de Acuarela (Watercolor)

Una técnica más complicada es la simulación de una pintura en acuarela a partir de un modelo o escena 3D. La dificultad de esta técnica radica en que la acuarela no solamente tiene un "modelo de iluminación propio", sino que también presenta un comportamiento físico que hay que simular o por menos representar de un modo gráfico, consistente en la interacción de los pigmentos, diluidos en agua, con el papel. Un estudio detallado de este comportamiento, con énfasis en computación gráfica, se puede encontrar en [CAS+97].

Sin embargo, el estudio referenciado está enfocado a la simulación, por lo que no es adecuado para tiempo real. Pero existe un una técnica que intenta imitar visualmente la acuarela, propuesta en [BKTS06]. Esta técnica, pensada en tiempo real y en animaciones, se basa en el cel-shading y en otros conceptos para lograr su propósito, y consta de un algoritmo de dos pasos. El primero, de procesamiento del modelo tridimensional, busca crear la primera abstracción del modelo. Para ello crea una imagen mediante *cel-shading*, pero sin dibujar contornos. La segunda etapa es de procesamiento de la imagen, aplicando diversos filtros: un filtro para dar el aspecto irregular de la tintura, otro para darle más fuerza a la intersección entre dos pigmentos distintos, y otro para darle textura de papel a la imagen. Estos filtros son comunes en el área de visión computacional y por lo tanto no es difícil implementarlos o encontrar una implementación de ellos.

El resultado de esta técnica se puede ver en la Figura 5.18. Se observa que hay dos resultados, uno con la técnica aplicada directamente en el modelo y otra con un suavizamiento de normales del modelo. Este suavizamiento se aplica para eliminar detalles de la imagen resultante, debido a que no se busca una imagen perfectamente detallada.

La dificultad de implementar esta técnica está en el análisis de lo que sucede al aplicarla dentro de una animación, sea con una cámara fija y un modelo animado, un modelo fijo y una cámara en movimiento, o una cámara en movimiento y un modelo animado. Aquí surge el problema de la coherencia temporal, que será abordado en la siguiente Sección, y que se refiere a la continuidad y coherencia de lo graficado a través del tiempo. En particular, los problemas se dan al aplicar los filtros con texturas de ruido y de papel, que se asume que utilizan texturas fijas en 2D. La solución es reproyectar dichas texturas en el modelo de acuerdo al movimiento que se esté llevando a cabo.

5.3.4. Otras técnicas

Para estilos pictóricos, en especial aquellos que utilizan pinceladas o brochazos, [Mei96] propone una técnica que consiste en crear un sistema de partículas a partir de la parte visible

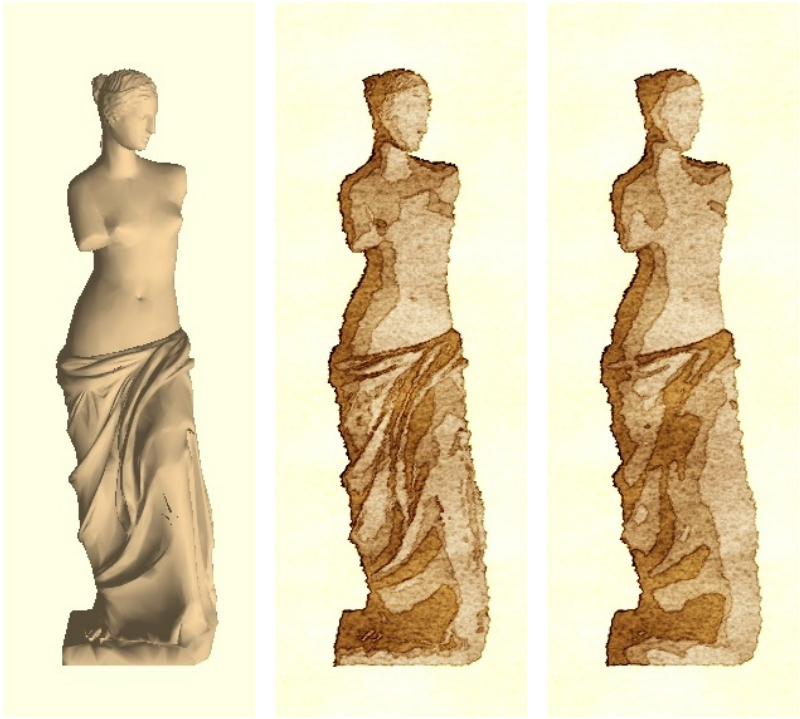


Figura 5.18: *Rendering tipo acuarela. Se muestra el modelo original con sombreado de Gouraud, un rendering tipo acuarela preservando el detalle y otro suavizando las normales del modelo, con el fin de eliminar detalles en la imagen final.*

del modelo 3D (una implementación con algunas mejoras se puede apreciar en [HS04]). En este sistema, cada partícula representa un brochazo, y cada brochazo adquiere su color a partir de una imagen de referencia, que simplemente es la graficación tradicional del modelo que se está dibujando. Este enfoque del problema es bastante parametrizable, ya que basta cambiar la forma y distribución de las partículas para tener un nuevo estilo pictórico, pero no es necesariamente el más adecuado, ya que las partículas suelen ser muy similares, provocando que la imagen creada no sea del todo convincente. La mayor dificultad es generar un buen conjunto de partículas, en superficies implícitas y paramétricas [WH05] propone métodos para obtener puntos representativos de la superficie. Para modelos tridimensionales arbitrarios, [KP] proponen un método para extraer puntos representativos, considerando animaciones y cambios de cámara. La diferencia entre estas técnicas y las de acuarela es que en esta última los brochazos se fusionan en el papel, mientras que en los estilos pictóricos las pinceladas y brochazos son claramente distinguibles.

Una técnica diferente, más enfocada a las imágenes que a los modelos tridimensionales, es la propuesta por [Her98]: se toma una imagen de referencia, se obtienen las posiciones y direcciones de los brochazos (usualmente segmentando la imagen mediante diagramas de Voronoi) y luego se grafica cada uno utilizando texturas y colores. Se puede aplicar a un modelo tridimensional si como imagen de entrada se utiliza el rendering tradicional, pero no es recomendable para tiempo real.

Otra técnica es el dibujo achurado, llamado *hatching* en inglés. Una superficie ilustrada de esta forma presenta un achurado fuerte en sus zonas más oscuras y representativas, mientras

que en sus zonas más iluminadas o de poco interés no se dibuja nada. En [HZ00] el achurado lo determinan las direcciones principales, que son suavizadas con el fin de volverlas más uniformes. Por otro lado, en [LKL06] se trabaja con texturas predefinidas, que contienen los achurados, que se mezclan de acuerdo a la intensidad de iluminación.

Existen otros trabajos que no buscan imitar medios tradicionales, pero sí destacar algunas características del objeto. Un ejemplo es [RBD06], donde en vez de modificar el pintado y sombreado del objeto, modifican las condiciones de la escena, en particular las propiedades de la luz para diferentes partes del objeto, de modo de exagerar la iluminación en sus puntos más interesantes y así destacar características de interés.

5.4. Coherencia Temporal

Aunque no existe una definición estricta de coherencia temporal, se refiere a la continuidad y coherencia de una animación a través de sus cuadros. Cuando se quiere entregar al usuario o espectador la *sensación de movimiento* entre un cuadro de una animación y otro, es necesario que no se muestren discontinuidades, es decir, si aparece una nueva línea, esa línea debe mantenerse o evolucionar durante un tiempo determinado y no tener una aparición abrupta y de corta duración, de modo que realmente aporte a la imagen y no sea solamente ruido.

Por otro lado, dos imágenes de un modelo tridimensional producidas en tiempos distintos pero bajo las mismas condiciones (misma configuración de fuentes de luz, de punto de vista, y de estilo gráfico) deben ser iguales. Por lo tanto, técnicas que requieran números aleatorios, o que realicen cálculos que no dependen de la geometría pero sí de factores externos que cambian arbitrariamente, son susceptibles a tener problemas de coherencia temporal. Como ejemplo se puede mencionar la técnica de rendering de acuarela descrita, que aplica texturas de ruido a un modelo tridimensional graficado como un dibujo animado. En ese caso no se puede utilizar un generador de texturas realmente aleatorio, porque en ese caso no existiría coherencia temporal. Se necesita utilizar un generador de texturas *pseudo-aleatorio*, que tenga apariencia aleatoria pero que en realidad tenga resultados definidos y predecibles por el desarrollador.

Dicho generador existe y se llama **Ruido de Perlin** [EM03]. Este "ruido" genera números aleatorios en 1, 2, 3 y 4 Dimensiones a partir de un número en el número de dimensiones requerido. Por definición, el Ruido de Perlin entrega siempre el mismo resultado para el mismo parámetro, por lo que es idóneo para este tipo de situaciones.

Para la extracción y dibujado de líneas, en [DFR04] se enumeran técnicas que permiten mejorar la coherencia temporal para los contornos sugestivos, pero son aplicables a otros tipos de contornos. En particular, se sugiere darle transparencia a las líneas en sus extremos, de modo que sea suave su movimiento y su aparición.

5.5. Bibliotecas y Aplicaciones

En la red se encuentran dos aplicaciones que permiten estudiar y probar la mayoría de las técnicas descritas en este capítulo. Una, **Real Time Suggestive Contours**, permite

trabajar con la extracción y dibujado de líneas en tiempo real. La otra, **Freestyle**, es un marco de trabajo para el trazado de líneas, que solamente considera aristas de silueta y de triángulos que formen un ángulo considerable entre sí, pero presenta una gran flexibilidad a la hora de estilizarlas. Ambas son de código abierto y multiplataforma.

5.5.1. Real Time Suggestive Contours – RTSC

`RTSC`[2] [DFR04] es una aplicación de demostración que implementa diferentes técnicas de detección de contornos en modelos tridimensionales. Implementa Contornos Sugestivos, Siluetas, Líneas de Destacado, Crestas Aparentes, Crestas y Valles, Curvas Gaussianas, sombreado de Gooch y algunas tonalidades específicas de Cel-Shading. A pesar de ello es una aplicación que difícilmente puede ser extendida para algo más generalizado que no sea el dibujo de líneas y de esos sombreados en particular, pues cada técnica que implementa está escrita en duro en el código sin tener relación con las otras técnicas. Además, solamente carga modelos estáticos, sin información de material ni textura. Sin embargo, es una gran fuente de conocimiento, ya que presenta un desempeño en tiempo real incluso para mallas grandes de cientos de miles de triángulos. Utiliza C++.

5.5.2. Freestyle

`Freestyle` [3] [GTDS04], es un marco de trabajo para el rendering estilizado de líneas a partir de modelos tridimensionales. Está programado en C++ pero con una interfaz para el usuario/desarrollador desde Python, que permite implementar fácilmente nuevos estilos para las líneas.

La limitación de Freestyle, en el contexto de esta memoria, es estar enfocado a tiempo no real. Tampoco tiene un conjunto de contornos muy extenso, ya que solamente extrae aristas de silueta y aquellas cuyos triángulos forman un ángulo considerable. A pesar de ello presenta muy buenas ideas en su diseño e implementación, y las imágenes que genera son de buena calidad.

[2] `http://www.cs.princeton.edu/gfx/proj/sugcon/`
[3] `http://freestyle.sourceforge.net/`

Capítulo 6

Diseño e Implementación

Este Capítulo contiene el diseño y los detalles de implementación de las distintas componentes del marco de trabajo. Se presentan las siguientes Secciones:

Sección 6.1, Componentes del Marco de Trabajo: Se realiza una descripción de alto nivel de las componentes del marco de trabajo, mostrándose la relación que existe entre ellas.

Sección 6.2, Componente Genérica: Contiene las clases genéricas que son utilizadas en las otras componentes.

Sección 6.3, Componente Geométrica: Contiene las clases y estructuras de datos de mallas de triángulos.

Sección 6.4, Componente de Modelos 3D: Contiene las clases y estructuras de modelos 3D estáticos y animados.

Sección 6.5, Contornos y Extractores de Curvas de Nivel: Contiene las clases y algoritmos que extraen y modelan diferentes tipos de contornos.

Sección 6.6, Componente de Visualización: Contiene las clases que permiten especificar algunos atributos de escena, graficar y visualizar modelos tridimensionales, así como crear nuevos tipos de graficadores y técnicas de NPR. También contiene las clases que funcionan como interfaz de alto nivel entre la GPU y las estructuras de las otras componentes.

Para cada componente se analiza el diseño y su respectiva implementación utilizando diagramas de clases y describiendo lo que hace cada clase de cada componente.

6.1. Componentes del Marco de Trabajo

El marco de trabajo está compuesto de cinco componentes principales, que serán descritas con detalle en las Secciones siguientes. Sin embargo, en esta Sección se presenta una visión

de alto nivel de estas componentes, de modo de clarificar las relaciones que existen entre ellas y sus propósitos.

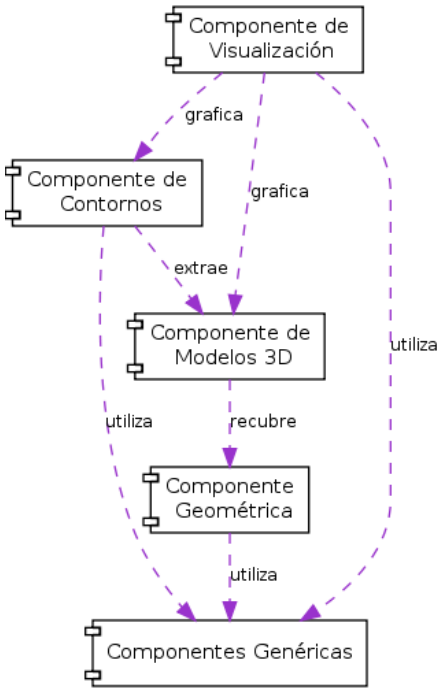


Figura 6.1: *Visión de alto nivel de las componentes del marco de trabajo y sus relaciones.*

En la Figura 6.1 se muestra una representación visual de las componentes y sus relaciones. En la parte inferior del diagrama se encuentran la Componente Genérica, que contiene clases parametrizadas (es decir, *templates*) que representan tipos de datos básicos y comunes en las distintas áreas que se abarcaron en los Capítulos anteriores. Debido a esto, esta componente es utilizada por las otras componentes, en particular la Componente de Modelos 3D y la Componente de Contornos.

La Componente Geométrica contiene las estructuras de datos necesarias para representar mallas de triángulos. Se puede decir que esta componente es de bajo nivel en términos gráficos. Por otro lado, la Componente de Modelos 3D es de alto nivel, ya que recubre la Componente Geométrica para poder representar modelos tridimensionales estáticos y animados. Este recubrimiento incluye la asociación de las distintas clases de la Componente Geométrica para tener una malla de triángulos, así como la carga de datos y la modificación de éstos, a través de los estimadores de estructuras diferenciales. También se incluyen las clases que modelan los atributos gráficos de un modelo tridimensional.

La Componente de Contornos tiene dos funciones: la primera es definir tipos de contornos o líneas de interés, la segunda es extraer esos contornos a partir de un modelo tridimensional dado. Para la definición de contornos provee una interfaz que define la función en un vértice de la malla de triángulos, y para la extracción provee una interfaz para implementar algoritmos de extracción. Incluye la implementación del algoritmo que recorre toda la malla visto en el Capítulo 4.

La Componente de Visualización contiene las clases para poder visualizar modelos tridi-

mensionales y, en caso de ser necesario, los contornos que han sido extraídos de ellos. Para esta visualización se proveen clases que modelan los atributos gráficos de una escena, las clases que conectan las estructuras de datos del marco de trabajo con la GPU y las interfaces de graficadores, que son la base para implementar graficadores tradicionales y técnicas de NPR.

6.2. Componente Genérica

En los Capítulos 3 y 4 se observó que muchas técnicas y algoritmos hacen uso de operaciones vectoriales y matriciales. Por este motivo, se optó por implementar clases genéricas de **Posiciones** y **Matrices** que permitan parametrizar posiciones en espacios de cualquier dimensión y de matrices de cualquier tamaño. Del mismo modo, también se implementaron **Interpoladores** lineales y cúbicos.

6.2.1. Posiciones

La clase genérica `Posicion<T,N>` permite expresar una posición en un espacio de N dimensiones de tipo `T`. En la Figura 6.2 se puede observar que todas las instancias de esta clase trabajan, hasta el momento, con posiciones en espacios de punto flotante, de dimensiones 2, 3 y 4.

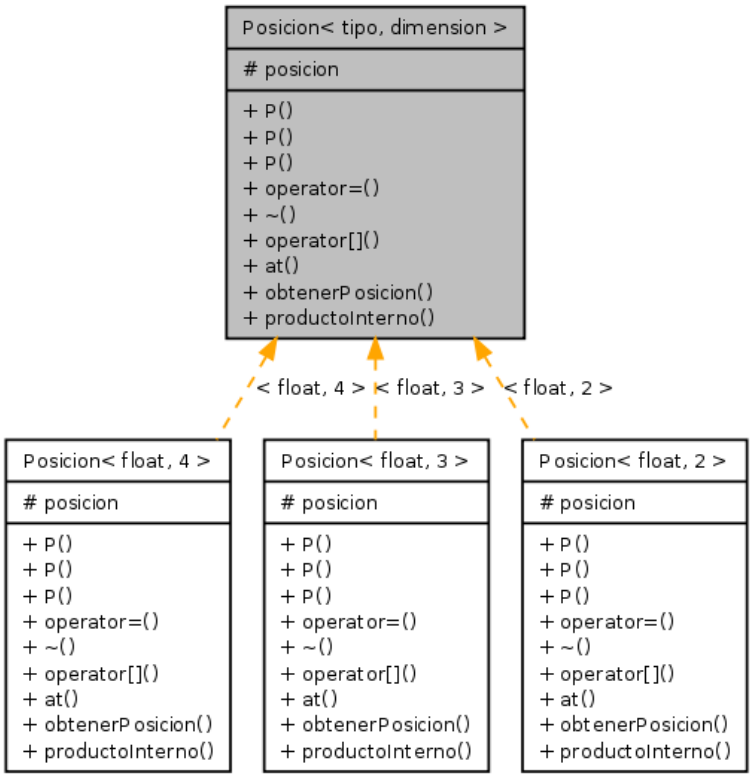


Figura 6.2: *Clase Genérica* `Posicion` *y derivadas.*

La clase `Posicion` funciona como un arreglo de elementos en memoria, pero añadiendo operaciones aritméticas ($+$, $-$, $*$ y $/$) y de producto interno ($\cdot$). Así, se vuelve fácil trabajar con posiciones genéricas ya que se facilitan las operaciones típicas que se efectúan con ellas.

6.2.2. Matrices

La clase genérica `Matriz<T,M,N>` permite expresar una matriz de elementos de tipo `T` de tamaño $M \times N$. En la Figura 6.3 se observan las operaciones implementadas para estas matrices genéricas, así como las instancias de la clase. Se ve también la clase `MatrizDeTransformacion`, que hereda de `Matriz<float,4,4>` en concordancia con lo visto en el Capítulo 3.

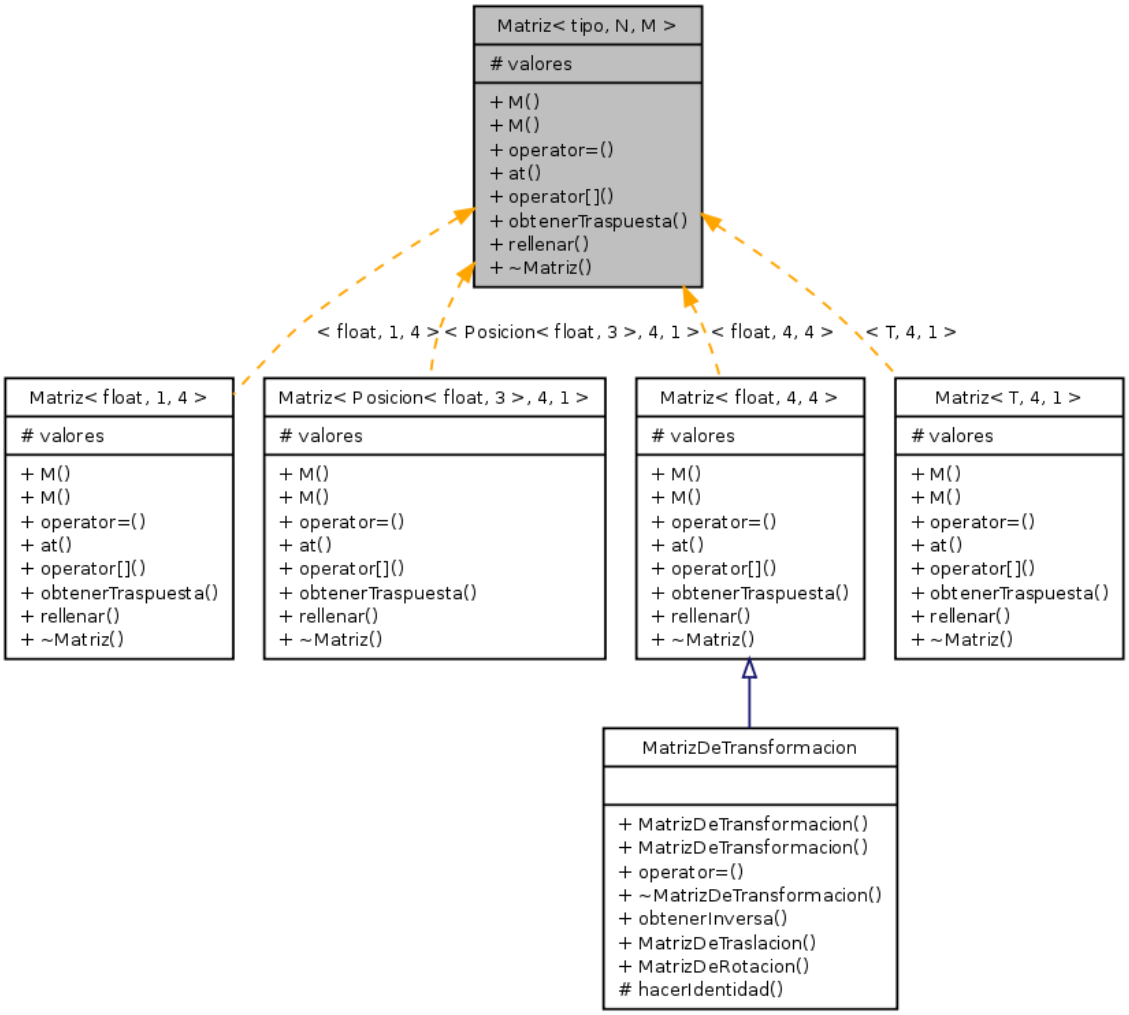


Figura 6.3: *Clase Genérica `Matriz` y derivadas, incluyendo `MatrizDeTransformacion`.*

La clase `Matriz` implementa operadores aritméticos ($+$, $-$, $*$ y $/$) entre matrices. Además permite la multiplicación entre ellas con elementos de distinto tipo (si y solo si esos tipos implementan los operadores $+$ y $*$ entre sí). Se observa que se puede calcular la inversa de una matriz; para la clase `MatrizDeTransformacion` no se calcula realmente una matriz inversa

según procedimientos matemáticos, sino que se está utilizando una heurística que permite obtener la inversa de una matriz que es composición de traslaciones y rotaciones.

6.2.3. Interpoladores

Como se vio en el Capítulo 3, las interpolaciones cúbicas se pueden expresar a partir de matrices de 4×4, por lo que su implementación es trivial si se cuenta con matrices y posiciones genéricas. Un punto importante que debe tenerse en cuenta es que la matriz de 4×4 contiene valores de punto flotante, con coeficientes conocidos, mientras que el vector con puntos de control no necesariamente contiene números, ya que un punto de control puede ser definido como una posición, un vector o cualquier otro tipo de elemento que desee interpolarse. Debido a esto las clases de interpolación también son genéricas, siendo la clase `Interpolador<T>` la interfaz base para interpolar puntos de control de tipo `T`.

Los interpoladores implementados son `InterpoladorLineal<T>`, y los interpoladores cúbicos `CatmullRom<T>` y `BSpline<T>`. En la Figura 6.4 se observa el diagrama de sus clases.

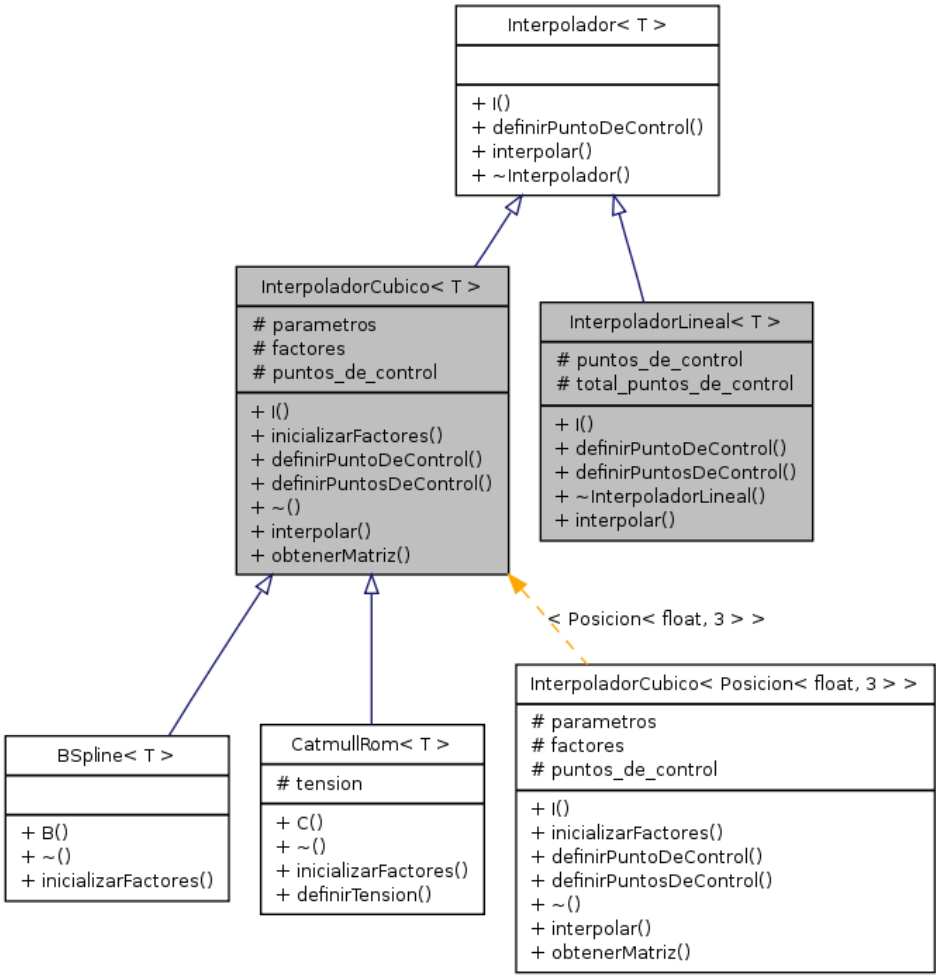


Figura 6.4: *Clase Genérica `Interpolador` y derivadas: `InterpoladorLineal` e `InterpoladorCubico` (de la cual derivan `Catmull-Rom` y `B-Spline`).*

La clase `Matriz<Posicion<float,3>,4,1>` que se observa en la Figura 6.3 corresponde a la instanciación de la clase `Matriz` realizada en el interior de los interpoladores cúbicos para posiciones. Esta matriz de 4×1 contiene los cuatro puntos de control que definirán la interpolación.

6.3. Componente Geométrica

Esta componente contiene clases que implementan las estructuras de datos de mallas de triángulos, haciendo un fuerte uso de la componente genérica. Un desarrollador no debería acceder directamente a esta componente, ya que ha sido pensada desde un comienzo para representar las primitivas necesarias para el proceso de rendering. Estas primitivas difícilmente cambian con el tiempo, por lo que no debería ser necesario modificar estas clases a menos que se esté implementando un algoritmo que modifique los datos de un modelo tridimensional.

Las clases de esta componente tienen un grado medio de **cohesión** entre sí, justificado por dos razones:

- Razón conceptual: las definiciones de los conceptos modelados por estas clases son, de por sí, cohesionados.
- Razón de diseño: es improbable que se requiera cambiar las estructuras internas de las mallas de triángulos, por lo que en vez de buscar una solución más extensible, menos específica y optimizada, se decidió utilizar clases fuertemente conectadas entre sí de modo de optimizar el uso de memoria y la comunicación entre ellas.

Las clases que pertenecen a esta componente no son flexibles en el contexto de las mallas de triángulos ni de los atributos de una escena, más bien lo son respecto al tipo de información que contienen y a las formas de acceder a ella. Las clases geométricas necesarias para representar mallas de triángulos se definieron de acuerdo a los requerimientos enunciados en el Capítulo 3 y se pueden apreciar en la Figura 6.5. El diagrama fue realizado utilizando un modelo de resortes[1], que ha identificado como componentes centrales a las clases `Punto` y `Vector`.

6.3.1. Clases `Punto`, `Vector` y `Vertice`

La clase `Punto` representa una posición en tres dimensiones, expresada en coordenadas homogéneas. Además de sus atributos de posición, contiene atributos de curvatura y de derivada de curvatura. Implementa las operaciones aritméticas necesarias para la interpolación y métodos para:

- Determinar la distancia entre dos instancias de `Punto`.
- Trabajar con la información auxiliar, incluyendo información de curvaturas e índices (cada `Punto` tiene un índice asignado dentro de la malla).

[1]Se utilizó el programa `neato` del paquete *graphviz*, descargable en `http://www.graphviz.org`.

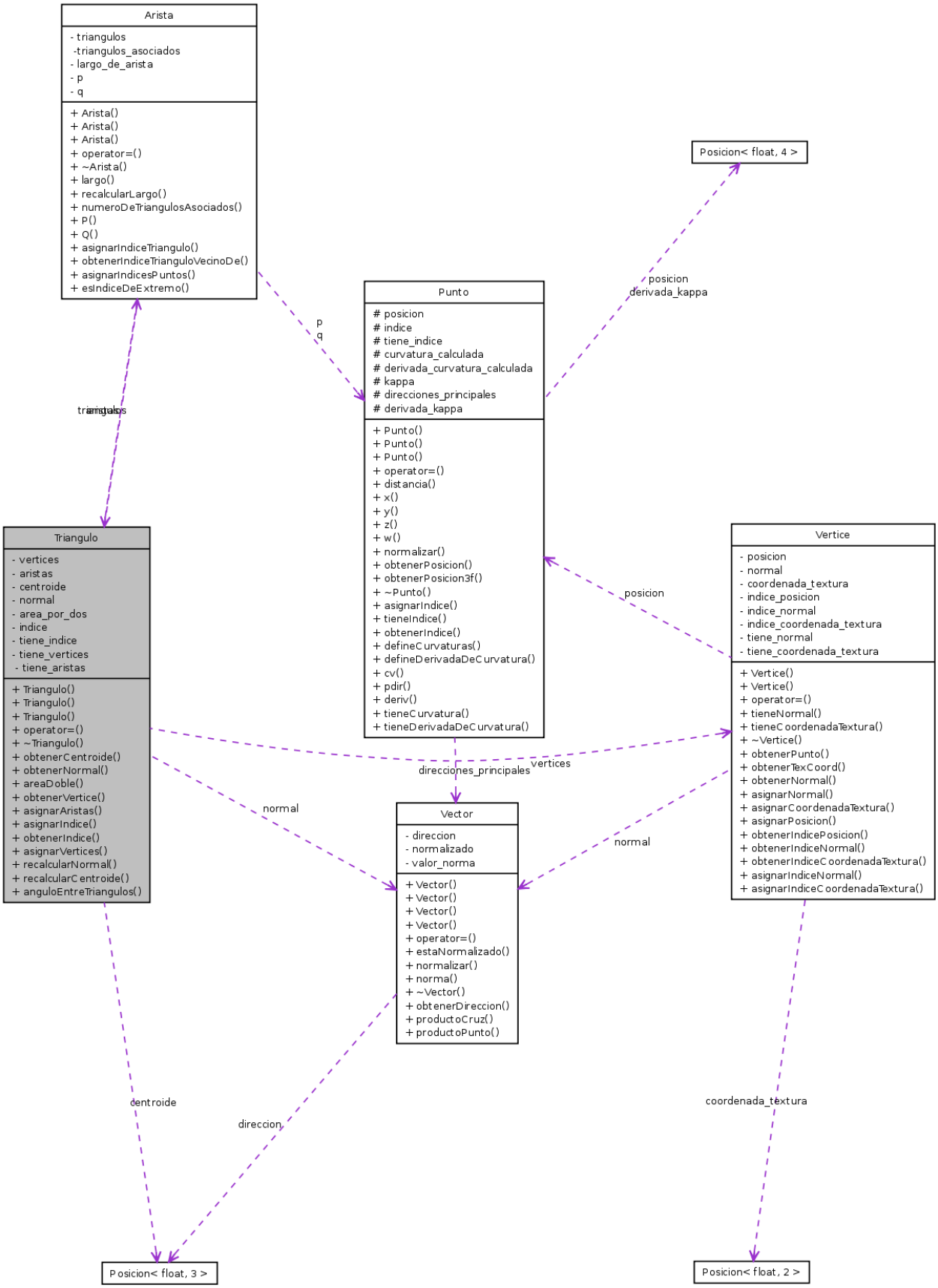


Figura 6.5: *Diagrama de las clases* `Punto`, `Vertice`, `Vector`, `Arista` *y* `Triangulo`.

- Determinar si tiene o no calculada su información de curvatura.

La clase `Vector` representa una dirección en el espacio. No utiliza coordenadas homogéneas, aunque puede ser de largo arbitrario. Implementa:

- Operaciones de *producto punto* ($\cdot$) y *producto cruz* ($\times$).
- La normalización del largo del vector.
- Operadores aritméticos ($+$, $-$, $*$ y $/$)

La clase `Vertice` permite representar un vértice de un triángulo. Contiene punteros al `Punto` que representa su posición, al `Vector` que representa su vector normal y a la `Posicion` que representa su coordenada de textura, si es que posee una. Una instancia de esta clase no tiene el significado geométrico de un vértice en Geometría Computacional, más bien es un *placeholder* o un contenedor para la información de un vértice dentro de un triángulo. Dos vértices que tienen la misma posición no necesariamente comparten el vector normal, y si lo hacen, pertenecen a dos triángulos distintos, por lo que son dos vértices *diferentes*.

6.3.2. Clases `Arista` y `Triángulo`

La clase `Arista` representa una arista definida por dos instancias de `Punto`. Contiene punteros a los triángulos que la comparten, de modo que se pueden conocer los vecinos de un triángulo preguntándole a sus aristas.

La clase `Triangulo` contiene:

- Punteros a sus vértices y aristas.
- Un `Vector` que representa la normal del triángulo.
- Una `Posicion` que representa su centroide (calculado como el promedio simple entre las posiciones de sus vértices).
- Otros valores auxiliares como su índice y el doble de su área.

El que ambas clases se referencien entre sí aumenta la cohesión de esta componente. Esta decisión de diseño responde a la problemática que se desea resolver: ser la base de una malla de triángulos que posea más información que una simple lista de triángulos, pero sin tener estructuras demasiado complejas que afecten el desempeño de la aplicación.

6.4. Componente de Modelos 3D

Un modelo 3D no es solamente una malla de triángulos, aunque las mallas ciertamente son la base de lo que es un modelo 3D. También se deben considerar los atributos gráficos

que tienen los modelos, en particular los materiales y las texturas. Esta componente contiene clases que modelan distintos tipos de modelos 3D, incluyendo sus atributos gráficos, contiene clases que permiten leer datos desde el disco y algoritmos de estimación de estructuras diferenciales de primer, segundo y tercer orden. Así, se puede decir que esta componente tiene dos tipos de clases:

- Las relacionadas con las estructuras de datos de un modelo tridimensional: aquellas que interactúan con la malla de triángulos, pudiendo ser tanto cargadores de datos (por ejemplo, leen los modelos desde una fuente de datos) como modificadores de ellos (por ejemplo, estiman normales y curvaturas para el modelo).
- Las relacionadas con los modelos tridimensionales en sí: es decir, las clases que representan la malla de triángulos y sus atributos gráficos.

Existen variados formatos de modelos tridimensionales, tanto en la forma de almacenamiento como en los tipos de animaciones, por lo que se estas clases uniforman lo más posible la representación en memoria de ellos.

6.4.1. Modelos Estáticos

La clase `Modelo3D` es la clase base para representar un modelo tridimensional. Corresponde a un modelo estático que implementa las estructuras mencionadas en el Capítulo 3. En la Figura 6.6 se observa el diagrama de clase de `Modelo3D`. Como se dijo anteriormente, la componente de modelos tridimensionales recubre la componente geométrica, lo que se aplica directamente en la clase `Modelo3D`, que, a grandes rasgos, contiene una malla de triángulos y sus atributos gráficos.

Como se tienen conjuntos de puntos, de vectores, aristas, triángulos, vértices y materiales, se utiliza el contenedor `std::vector`, que permite acceso aleatorio a los datos y que asegura que todos están en posiciones contiguas en memoria.

La información de vecindad se almacena siguiendo un modelo de *listas de adyacencia*, mediante instancias de la clase `std::list`, lo que permite recorrer fácilmente las listas. La utilización de listas tan simples no incide en la eficiencia de la estructura porque *no se requiere acceso aleatorio* a los elementos de ellas. Sólo se requiere iterar en la lista elemento por elemento.

Existen dos listas de adyacencia, una para almacenar los índices de los vértices asociados a una posición y otra para los triángulos asignados a una posición, lo cual sumado a la información de vecindad de los triángulos y aristas completa la información requerida conceptualmente por las estructuras de datos vistas en el Capítulo 3.

6.4.2. Modelos Animados con Keyframes

La clase `ModeloAnimado` deriva de `Modelo3D`, por lo que se puede considerar un modelo 3D especializado. Esta especialización consiste en la adición de las animaciones con *keyframes*

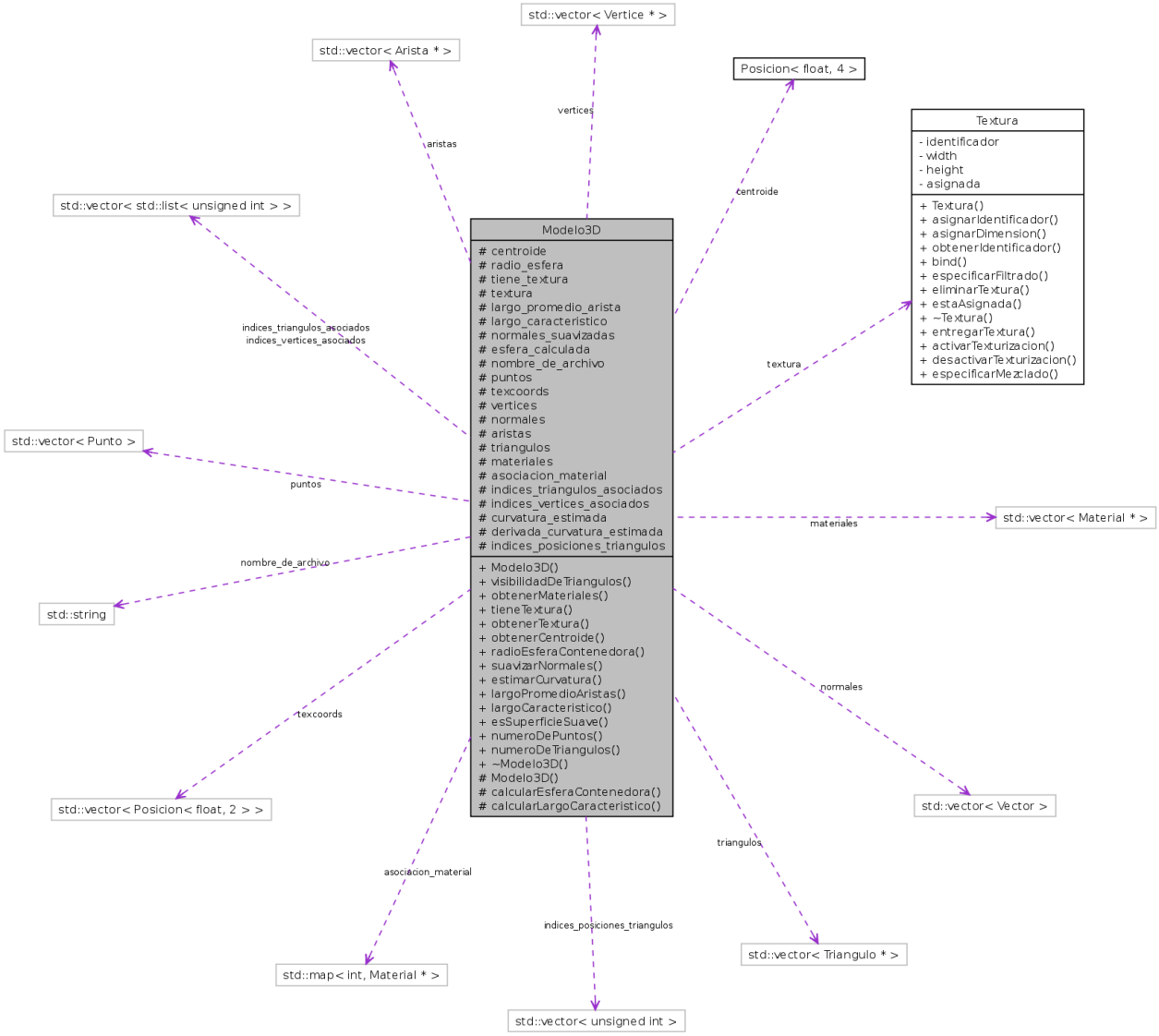


Figura 6.6: *Diagrama de clases de* `Modelo3D` *y* `Textura`.

del modelo. Estas animaciones se implementan mediante una clase `AnimacionConKeyframes`, que contiene la información de la animación: sus cuadros (clase `CuadroDeAnimacion`) y su nombre (por ejemplo "*WALKING*"). En la Figura 6.7 se observa el diagrama de estas clases.

Un `CuadroDeAnimacion` contiene las posiciones y normales correspondientes a cada vértice en la animación, de acuerdo a lo estipulado en el Capítulo 3 sobre los modelos animados. Se exige que los modelos animados presenten coherencia en la especificación de sus *keyframes*, es decir, el número de triángulos debe ser el mismo en cada cuadro.

La clase `ModeloAnimado` sigue manteniendo las estructuras de `Modelo3D`. Ya que su información se anexa a la de un modelo estático, es necesario *asignar* los valores de la animación actual (sea un *keyframe* o la interpolación entre dos *keyframes*) a las estructuras estáticas. Esto permite la utilización de `ModeloAnimado` en el contexto de `Modelo3D` sin necesidad de especificar que se está trabajando con un modelo animado.

6.4.3. Extendiendo los Modelos Implementados

Se han implementado los modelos estáticos y los modelos animados con keyframes. Pero, ¿qué sucede con los modelos con esqueleto? Ciertamente, de acuerdo a su definición, la implementación de este tipo de modelos es bastante complicada y por sí misma podría ser un tema de memoria. Ahora bien, el diseño de la clase base, `Modelo3D`, ha sido pensado en posibles extensiones como ésta, ya que un modelo con esqueleto tiene una única malla (es una especie de extensión de un modelo estático, mientras que en Vertex Animation se puede decir que son varios modelos estáticos juntos). Para implementar *Vertex Skinning*, se deben diseñar e implementar las clases que definen el esqueleto de un modelo, de forma similar a como está implementada la clase `ModeloAnimado`: un modelo con esqueleto tiene un esqueleto asociado y el esqueleto tiene un conjunto de huesos y articulaciones.

Cuando se hayan diseñado las clases que definan el esqueleto, se debe interpretar la información de movimiento y poblar las estructuras que se heredaron de `Modelo3D`, del mismo modo en que lo hace `ModeloAnimado`, con el fin de poder utilizar un modelo con esqueleto en el contexto de un modelo tridimensional estático.

6.4.4. Cargadores de Modelos

Una vez especificadas las clases que permiten representar modelos tridimensionales, es necesario poblarlas con datos provenientes de modelos reales. Para esto se ha especificado una interfaz que define la metodología para cargar los modelos. La clase base es `CargadorModelo3D`, declarada clase amiga de `Modelo3D`, por lo que tiene total libertad para manipular las estructuras de datos internas de `Modelo3D`. Para los modelos animados se provee una interfaz especializada, que hereda de `CargadorModelo3D`, llamada `CargadorModeloAnimado`. En la Figura 6.8 se observan estas clases y las derivadas que se implementaron.

Para los modelos estáticos se provee una clase `CargadorOBJ` que lee archivos en formato `.OBJ` (*Alias Wavefront*)[2]. A su vez, de `CargadorModeloAnimado` hereda `CargadorMD2`, que

[2]Los archivos .OBJ son utilizados como formato de prueba por una gran cantidad de investigadores, ya

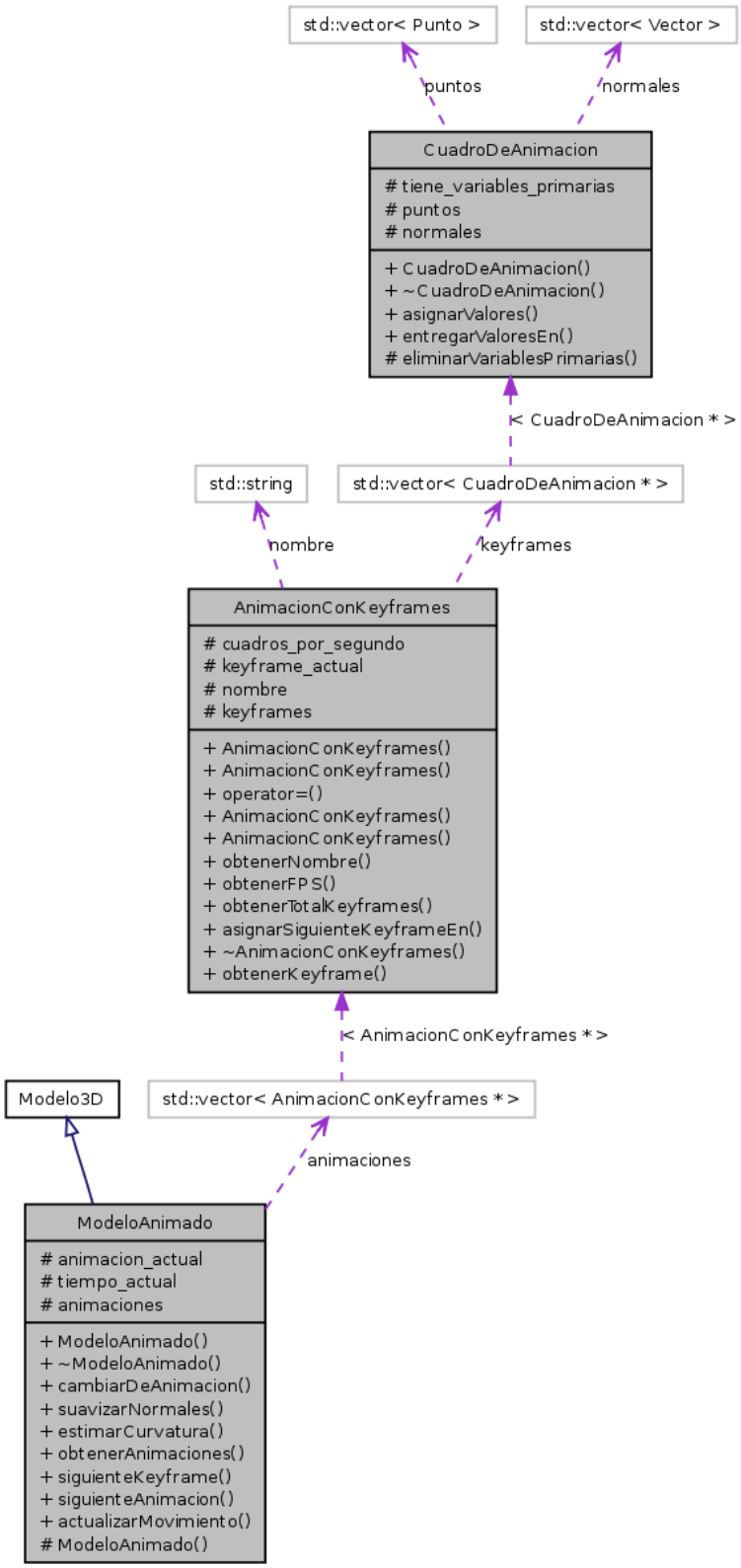


Figura 6.7: *Diagrama de clases de `ModeloAnimado`, incluyendo las clases `AnimacionConKeyframes` y `CuadroDeAnimacion`.*

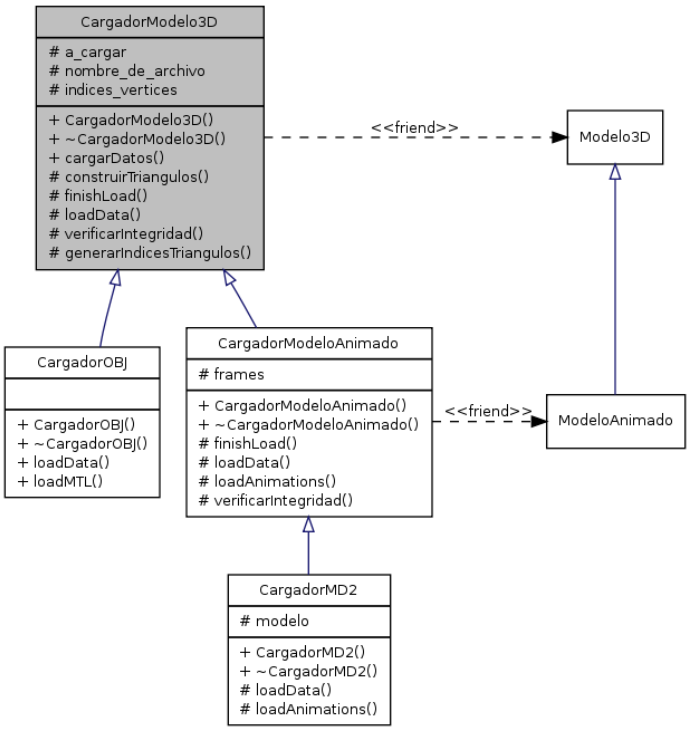


Figura 6.8: *Diagrama de clases de* `CargadorModelo3D` *y* `CargadorModeloAnimado`*, y sus derivadas* `CargadorOBJ` *y* `CargadorMD2`*.*

carga modelos en formato `.MD2` (*Quake II Model*)[3].

6.4.5. Extendiendo la Lectura de Datos

En el marco de trabajo ya se ha implementado la lectura de dos formatos de almacenamiento de modelos 3D. Sin embargo, no todos los usuarios tienen sus datos en esos dos formatos. Por lo tanto, en algunas ocasiones será necesario crear un cargador para esos formatos específicos.

Para implementar un cargador de modelos, sea estático o animado, se debe implementar la función `loadData`, que recibe como parámetros las referencias a las estructuras de un modelo. En el caso de los modelos estáticos, se debe leer la información de los vértices (construir las instancias de las clases `Punto` y `Vector`, y `Posicion<float,2>` si es que se requieren coordenadas de textura) y generar una lista de índices de vértices que indican los triángulos que se van a construir. En el caso de los modelos animados, además de todo lo anterior se construyen las animaciones y sus respectivos *keyframes*.

La clase base se encarga de construir los triángulos y reconstruir la información de vecin-

que pueden ser importados y exportados desde todas las aplicaciones de modelado 3D. Incluyen información de materiales y posibles texturas que incluya el modelo, entre otras cosas.

[3]Los modelos .MD2 incluyen una serie de animaciones, textura y modelos, aunque lamentablemente por limitaciones del formato tienen mallas muy gruesas (tienen muy pocos triángulos para modelar un objeto de alta complejidad), lo cual los hace muy aptos para los juegos pero inapropiados para aplicar algoritmos geométricos. Se eligió este formato porque la lectura de datos es sencilla de implementar.

dad, así como realizar diferentes pruebas para verificar la integridad de los datos. En el caso de implementar una nueva clase derivadada de `Modelo3D`, se debe especificar la interfaz asociada. Por ejemplo, en el caso de los modelos con esqueleto, se debe especificar también una clase `CargadorModeloConEsqueleto`, que a partir de la fuente de datos recree el esqueleto y pueble por primera vez los datos de la clase padre.

6.4.6. Estimadores de Estructuras Diferenciales

Los algoritmos de estimación de normales y de curvatura (así como de derivadas de curvatura) vistos en el Capítulo 4 se implementan en las clases que se pueden observar en la Figura 6.9. Estas clases tienen una base llamada `Estimador`, de la cual heredan `EstimadorCurvatura` y `EstimadorDeNormales`. La primera implementa los algoritmos mencionados en la Sección 4.1.4, basada en el código de `RTSC`, pero utilizando las estructuras de datos del marco de trabajo[4]. La segunda implementa los algoritmos mencionados en la Sección 4.1.2.

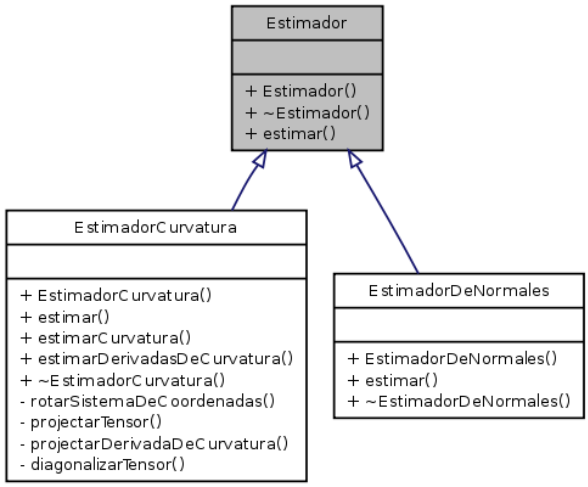


Figura 6.9: *Diagrama de clases de `Estimador` y sus implementaciones `EstimadorCurvatura` y `EstimadorDeNormales`.*

Los estimadores actúan de un modo similar a los cargadores, ya que interactúan directamente con los datos internos de las clases de modelos 3D, aunque esta vez en vez de definirse como clases amigas se implementa una versión simplificada del patrón de diseño *Visitor*, lo que implica que cada clase derivada de `Modelo3D` debe reimplementar los *métodos de visita* de los estimadores. En el caso de `ModeloAnimado`, los estimadores se ejecutan del mismo modo que en `Modelo3D`, pero las estimaciones las realizan en las instancias de `CuadroDeAnimacion` que pertenecen al modelo.

6.4.7. Atributos Gráficos: `Color`, `Material` y `Textura`

De acuerdo a lo planteado en el Capítulo 3, un color se puede representar como un punto en un espacio en tres dimensiones. Además de ello, en Computación Gráfica se trabaja con

[4]Además, de RTSC se utilizó el archivo `lineq.h`, que implementa mínimos cuadrados para el cálculo de las derivadas y de las curvaturas.

la transparencia o *canal alpha*, por lo que la clase `Color` contiene como variable interna una instancia de `Posicion<float,4>`[5], donde la cuarta componente corresponde al canal alpha del color. También se implementan operaciones aritméticas entre colores con el fin de realizar mezclas cuando sea necesario. Estas operaciones respetan el espacio de color, es decir, nunca se salen del cubo unitario del espacio RGB.

La clase `Material` contiene la especificación de sus valores K_a, K_d y K_s como instancias de la clase `Color`, y el *factor de brillosidad* como un valor de punto flotante.

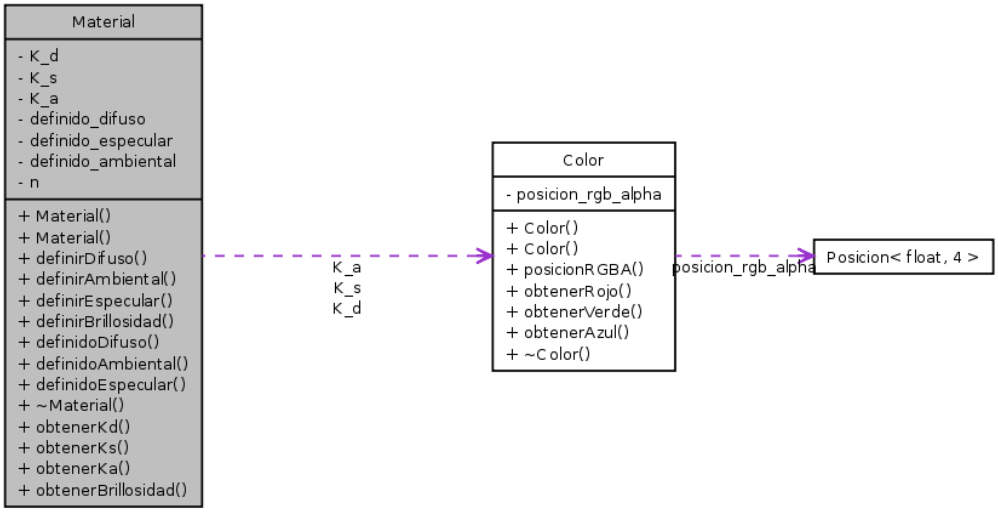


Figura 6.10: *Diagrama de clases de `Material` y `Color`.*

La clase `Textura` (ver Figura 6.6) es un contenedor de información sobre una imagen que ha sido entregada a OpenGL[6]. Contiene información sobre sus dimensiones, así como el identificador único que le es asignado por OpenGL. También permite crear una textura en 1D a partir de un arreglo de valores de punto flotante.

6.5. Contornos y Extractores de Curvas de Nivel

Esta componente contiene las clases relacionadas con la extracción de información a partir de los modelos, es decir, los extractores de líneas y las definiciones de contornos. En la Figura 6.11 se observa el diagrama de las interfaces `Contorno` y `ContourGenerator`, en conjunto con sus clases derivadas. Este conjunto de clases permite trabajar con curvas de nivel en modelos que aproximen superficies suaves. Ya que trabajan con la clase base `Modelo3D`, es posible aplicar estos algoritmos a instancias de `ModeloAnimado` o a otras posibles clases derivadas que se implementen en el futuro.

[5]A pesar de ser conceptualmente una posición en el espacio, se decidió no derivar la clase `Posicion` ya que nunca un `Color` será utilizado en el contexto de una `Posicion`.

[6]Idealmente esta imagen debiera cargarse en la misma clase `Textura`. Sin embargo, por el momento se está utilizando una clase auxiliar que realiza esta carga, llamada `CargadorQT` ya que utiliza la biblioteca `Qt4`, disponible en `http://www.trolltech.com`. Como la clase tiene un único método y desaparecerá en las siguientes versiones del marco de trabajo, no es necesario documentarla.

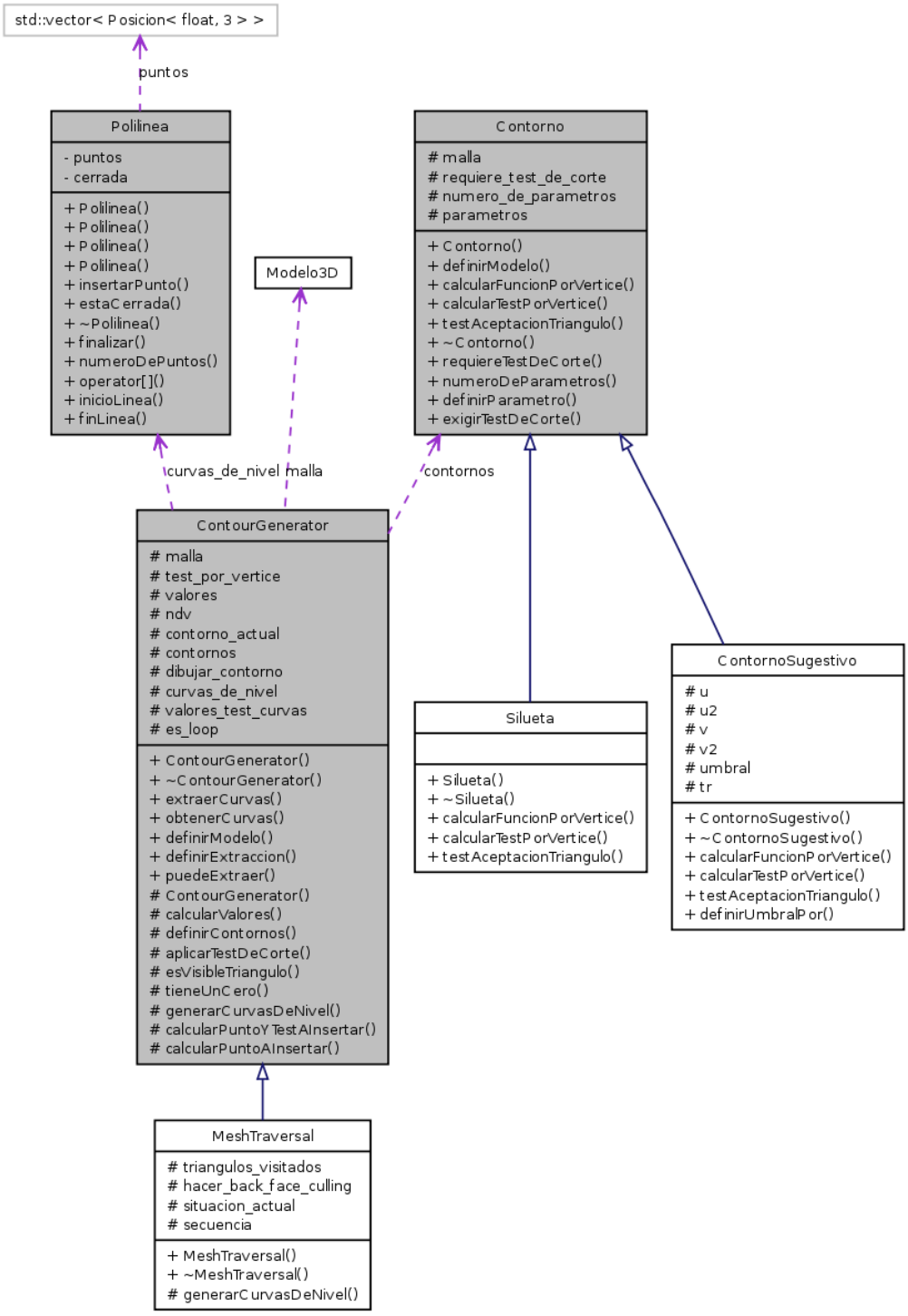


Figura 6.11: *Diagrama de clases para la componente de contornos y extractores de curvas de nivel. Se observa la interfaz de* `ContourGenerator`*, y el algoritmo de recorrido de la malla que la implementa,* `MeshTraversal`*. También se encuentra la interfaz* `Contorno`*, sus implementaciones* `Silueta` *y* `Contorno Sugestivo`*, y la clase* `Polilinea`*.*

También se observa la clase `Polilinea`, que permite representar una curva como un conjunto de puntos que deben recorrerse en orden, implementa un contenedor para una instancia de `std::vector<T>`, con `Posicion<float,3>` identificado como `T`. La clase que implemente la interfaz `ContourGenerator` recibe un conjunto de implementaciones de `Contorno`, un `Modelo3D`, un punto de vista (es decir, una `Posicion`) y entrega un conjunto de polilíneas con las curvas de nivel especificadas por los contornos.

6.5.1. Interfaz de Contornos

La clase `Contorno` define una interfaz para representar los contornos mencionados en el Capítulo 5. Esta clase tiene tres funciones principales:

- `calcularFuncionPorVertice`: evalúa el contorno en un vértice de la malla de triángulos. Recibe como parámetros las propiedades geométricas de ese vértice.
- `calcularTestPorVertice`: evalúa si el contorno cumple la condición de corte en ese vértice. Recibe como parámetros las propiedades geométricas de este vértice.
- `testAceptacionTriangulo`: evalúa si un triángulo puede ser descartado rápidamente sin necesidad de evaluar la función propia del contorno o el test de corte en cada vértice. Recibe como parámetro el triángulo a evaluar.

Cada contorno tiene otras variables internas, como un valor booleano que indica si necesita test de corte y un arreglo con posibles parámetros internos que el contorno pueda requerir.

6.5.2. Definición de Tipos de Contornos

La clase `Contorno` permite definir un tipo de contorno en base a la definición de ese contorno, sin preocuparse en cómo será extraído o qué tipo de modelo tridimensional se está utilizando, porque esas tareas las debe realizar el algoritmo de extracción. El diseño de esta clase se realizó considerando los contornos definidos en el Capítulo 5, aunque no todos ellos fueron implementados debido a que el trabajo de esta memoria no es implementar todos los contornos, sino entregar las herramientas para implementarlos.

A modo de prueba, se implementaron las siluetas (clase `Silueta`) y los contornos sugestivos (clase `ContornoSugestivo`). La clase `Silueta` por su simplicidad no presenta mayor interés, pero la clase `ContornoSugestivo`, por la complejidad del contorno que modela, sí. Esta clase implementa el test de corte indicado en los antecedentes y descarta los triángulos que tienen curvatura gaussiana positiva en todos sus vértices. Además, utiliza un parámetro de tolerancia para el test de corte. Ya que los otros contornos tienen definiciones similares, sus implementaciones deben ser análogas a la de los contornos sugestivos.

6.5.3. Algoritmos de Extracción

La clase `ContourGenerator` define una interfaz para implementar un algoritmo que extraiga curvas de nivel en base a una posición arbitraria, que en NPR suele ser el punto de

vista o la posición de la cámara. La clase `ContourGenerator` recibe como parámetros, entonces, un conjunto de punteros a instancias de la clase `Contorno` y una `Posicion`. Como los punteros deben ser creados fuera de la clase, de acuerdo a las necesidades de extracción de la aplicación, `ContourGenerator` (y sus derivadas) no necesitan conocer el tipo de `Contorno` que están calculando, ya que solamente utilizarán sus funciones de test por vértice, de cálculo de función por vértice y posiblemente de descarte rápido de triángulos. Dichas funciones pueden acceder a variables y métodos internos propios de cada tipo de contorno, evitando que alguna implementación de `ContourGenerator` requiera conocer el tipo de `Contorno` que está calculando en la malla. Esto permite la implementación de contornos tan complejos como las crestas aparentes, pues todos esos parámetros que no están contemplados por la interfaz `Contorno` se pueden especificar fuera del extractor de contornos.

La clase `MeshTraversal` implementa el algoritmo de extracción de curvas de nivel en base a lo visto en la Sección 4.2.3, siguiendo la interfaz definida por `ContourGenerator`.

6.6. Componente de Visualización

La componente de visualización se encarga de la graficación de los modelos tridimensionales. Implementa el *fixed pipeline* de modelos 3D y de polilíneas. Estas clases están diseñadas en base a la extensibilidad, con el fin de facilitar la implementación de graficadores y técnicas NPR.

Como clases de bajo nivel dentro de esta componente, se incluyen las necesarias para configurar una escena tridimensional básica (`FuenteDeLuz` y `Camara`) y para representar un programa escrito en Cg (`CgShader`). Estas clases son utilizadas dentro de los graficadores que son parte de esta componente.

Para el usuario desarrollador, las clases graficadoras son de uso directo. Simplemente les entrega el modelo tridimensional ya cargado en memoria y elige cómo dibujarlo. Para el usuario investigador, estas clases son la base de su trabajo de visualización: las clases base implementan el *fixed pipeline*, y las clases derivadas van aumentando en complejidad, incluyendo la comunicación con la GPU mediante el lenguaje Cg.

Las clases `GraficadorDeInterior` y `GraficadorDeLineas` implementan el *fixed pipeline* para los modelos y para las líneas que se desean dibujar. La primera grafica los modelos, incluyendo materiales y texturas, utilizando el sombreado de Gouraud; la segunda recibe el conjunto de curvas (lista con instancias de `Polilinea`) que entrega una implementación de `ContourGenerator` y las grafica uniendo los puntos que la definen con líneas rectas. No hay ningún procesamiento extra de los datos más que la transmisión directa a OpenGL, no así con sus posibles clases derivadas, que tienen más libertad.

De `GraficadorDeInterior` deriva otro graficador base, `GraficadorCg`. Este graficador implementa el mismo *fixed pipeline*, pero permite utilizar el lenguaje Cg para llevar a cabo la graficación. De este modo, `GraficadorCg` es una base para el desarrollo de graficadores que utilicen la GPU para graficar los modelos tridimensionales.

De `GraficadorDeLineas` deriva un graficador de líneas, llamado `Spline`. El graficador `Spline` recibe un conjunto de polilíneas y considera sus vértices como puntos de control para

una curva paramétrica. La curva paramétrica a utilizar depende del usuario, ya que se utiliza un puntero a una instancia de `InterpoladorCubico`.

Al igual que los extractores de curvas de nivel, los graficadores trabajan con instancias de `Modelo3D`, por lo que es posible utilizarlos con instancias de `ModeloAnimado` y otras posibles derivaciones de `Modelo3D`.

El diagrama de clases de esta componente, excluyendo los graficadores de líneas, se puede observar en la Figura 6.12.

6.6.1. Fuentes de Luz

Una fuente de luz es similar a un material por los valores que la definen, en este caso las intensidades I_a, I_d e I_s, que en la clase `FuenteDeLuz` son representados por instancias de `Color`. A diferencia de las propiedades de material, cuenta con dirección (una instancia de `Vector`) y posición (instancia de `Posicion<float,4>`, en coordenadas homogéneas).

La componente `posicion_transformada` es una variable auxiliar que permite conocer la posición de la luz respecto a una matriz de transformación particular. Esto es particularmente útil a la hora de trabajar con modelos 3D, que poseen coordenadas en sistemas propios y no en el sistema de coordenadas de la escena, en el cual está especificada la posición de la luz. El motivo por el que este valor está dentro de la clase es para evitar entregar más parámetros de lo adecuado a clases que utilizan `FuenteDeLuz` y que requieren sus coordenadas en el sistema propio, como es el caso de `GraficadorCg`.

6.6.2. Cámaras

Una cámara modela el punto de vista de la escena y las transformaciones de visualización. La clase `Camara` configura el punto de visión a partir de un Modelo3D "enfocado". Esta clase es muy similar a `FuenteDeLuz` en su implementación, ya que también tiene una componente `posicion_transformada` para facilitar algunas operaciones gráficas.

6.6.3. Clase `CgShader`

Previamente se definieron tres tipos de programas para GPUs que se pueden escribir en Cg. La clase `CgShader` sirve como una interfaz de alto nivel que permite cargar fácilmente *Vertex Programs* y *Fragment Programs* desde el disco, así como la entrega de parámetros desde las estructuras de datos del marco de trabajo. Esto significa que es posible indicar como parámetro a un programa de Cg una instancia de una clase del marco de trabajo. Las clases que se pueden enviar como parámetros son las siguientes: `Punto`, `Material`, `Color`, `FuenteDeLuz`, `Matriz`, `Textura` y `Posicion<float,4>`.

Esta clase no implementa ningún tipo de lógica de aplicación, solamente funciona como una interfaz de alto nivel. Probablemente un usuario avanzado puede utilizar las funciones de Cg directamente y así ganar en rendimiento al ahorrar las llamadas a las funciones de `CgShader`.

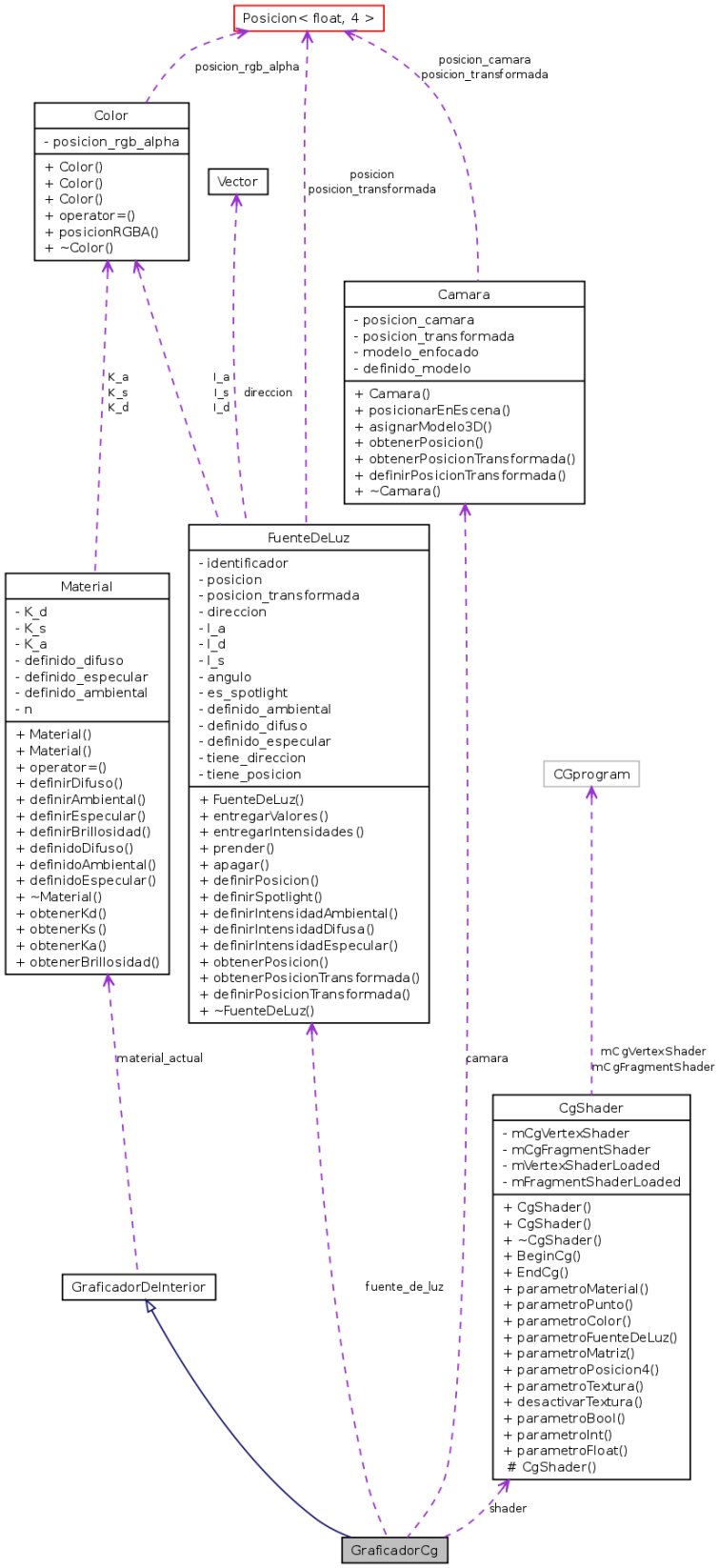


Figura 6.12: *Diagrama de clases para la componente de visualización (excluyendo los graficadores de líneas).*

6.6.4. Graficadores de Interior: Fixed Pipeline y Graficadores Cg

Al analizar la clase `GraficadorDeInterior` con más detalle, se puede encontrar que la graficación de un modelo consta de tres etapas: *preparación* (método `preparar`), *ejecución* (método `ejecutar`) y *finalización* (método `finalizar`). Lo que se realiza en cada una de ellas se puede resumir de la siguiente manera:

Preparación Se activa el mapeado de texturas si es que el modelo tiene alguno y se entregan parámetros específicos del graficador que se esté utilizando.

Ejecución Se entrega a OpenGL la información del modelo, es decir, la malla de triángulos con sus atributos gráficos.

Finalización Se desactiva el mapeado de texturas, con el fin de dejar la configuración de OpenGL del mismo modo en que estaba antes de graficar el modelo, y se liberan recursos que se pudieron adquirir en la etapa de preparación.

La clase `GraficadorDeInterior`, al implementar el *fixed pipeline*, no realiza más que lo mencionado y en realidad no necesita tal separación. Pero las clases que deriven de `GraficadorDeInterior` pueden aprovecharlo, como es el caso de `GraficadorCg`. El lenguaje Cg accede a muchas variables que se especifican durante el *fixed pipeline*, por lo que quien desee escribir un graficador utilizando dicho lenguaje no necesita re-escribir la etapa de ejecución. Solamente se requiere implementar la etapa de preparación y finalización, en las que se configuran los parámetros de los programas en Cg que se desean utilizar. Cada instancia de `GraficadorCg` (incluyendo clases derivadas) incluye sus propios programas escritos en Cg.

6.6.5. Creación de Nuevos Graficadores

La clase `GraficadorCg` contiene una instancia de `CgShader`, que se encarga de recibir los parámetros necesarios para cada programa. Se han implementado tres clases que derivan de `GraficadorCg`: `Phong`, que implementa el sombreado de Phong; `Toon`, que implementa *Cel-Shading*; y `Gooch`, que implementa `Gooch Shading`. Estas clases se pueden observar en la Figura 6.13, y sus implementaciones se pueden encontrar en el Apéndice A.

En el caso de los graficadores NPR, la clase `Toon` en su etapa de preparación entrega a `CgShader` las dos texturas 1D que representan las funciones por pasos para la intensidad de iluminación. Asimismo, desactiva dichas texturas en la etapa de finalización. La clase `Gooch` en su etapa de preparación calcula las tonalidades cálida y fría que asignará a través de la superficie del modelo a graficar. Ya que estas tonalidades dependen de las propiedades de material del objeto, deben ser recalculadas para cada modelo que se grafique. Es conveniente graficarlas en la etapa de preparación y no en la ejecución porque eso implicaría una redundancia de cálculos. La clase `Gooch` no implementa la etapa de finalización, porque solamente las texturas deben desactivarse y dicho graficador no las utiliza.

A partir de estas implementaciones se deduce que para crear un nuevo graficador puede bastar con escribir su inicialización en el constructor, su etapa de preparación y, si es que es necesario, su etapa de finalización. Algunos graficadores más avanzados, en posibles versiones

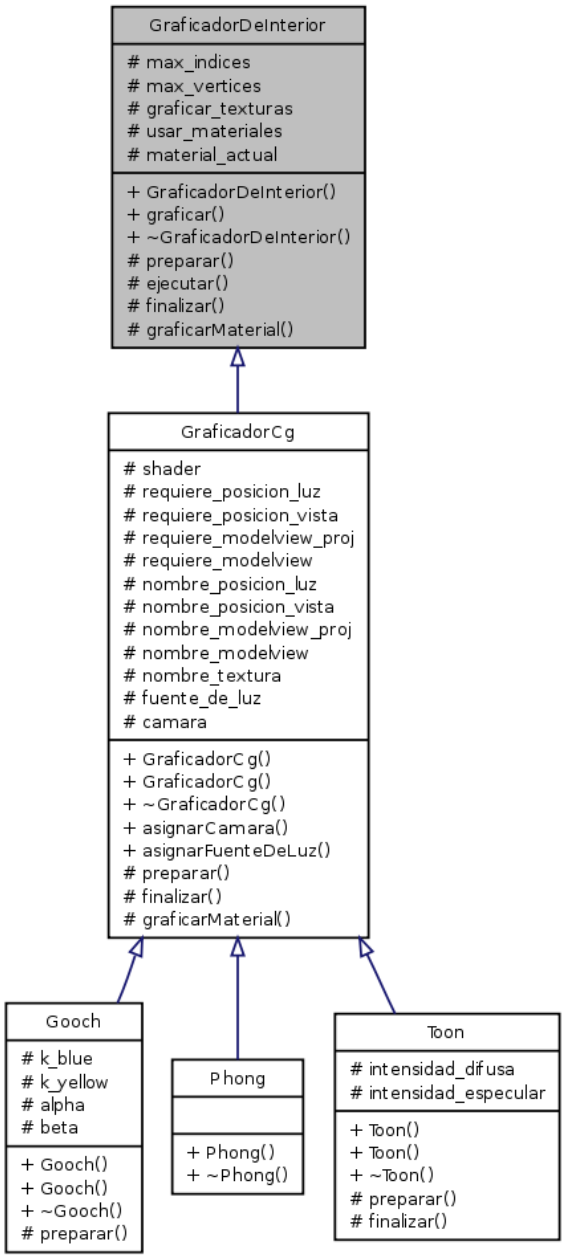


Figura 6.13: *Diagrama de clases de GraficadorDeInterior y sus clases derivadas.*

futuras del marco de trabajo, podrían reimplementar la etapa de ejecución, con el fin de realizar varios pasos sobre el modelo tridimensional.

6.6.6. Graficadores de Líneas

La clase `GraficadorDeLineas` presenta una interfaz análoga a la de `GraficadorDeInterior`, pero más sencilla, debido a que posibles extensiones de esta clase pueden ser de una naturaleza muy distinta a la clase original, y que se está trabajando con datos de menos complejidad (líneas en vez de modelos 3D). En la Figura 6.14 se observa el diagrama de clases para `GraficadorDeLineas` y la clase derivada que se implementó, `Spline`.

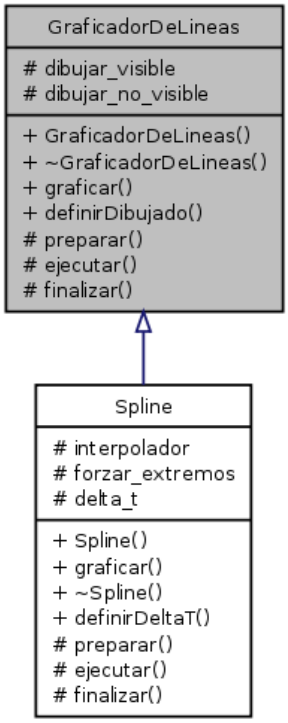


Figura 6.14: *Diagrama de clases de `GraficadorDeLineas` y su clase derivada `Spline`.*

6.6.7. Creación de Nuevos Graficadores de Líneas

El proceso para crear un nuevo graficador de líneas es similar a la creación de un nuevo graficador de modelos tridimensionales. Por ejemplo, la clase `Spline`, al inicializarse, recibe como parámetro una instancia de un puntero a una clase derivada de `InterpoladorCubico<T>` (con `T` especificado como `Posicion<float,3>`), que considera como puntos de control los puntos que definen las polilíneas que se reciben al momento de realizar la graficación (en concreto, una lista de instancias de `Polilinea`). De este modo, se pueden instanciar diferentes graficadores de `Spline` para diferentes interpoladores, permitiendo la aplicación de la curva paramétrica adecuada para los distintos tipos de contornos extraídos, tal como se muestra en el Apéndice B.

Una vez que se ha recibido la lista de polilíneas a graficar, se lleva a cabo una etapa de preparación, una de ejecución, y una de finalización. En el caso de `Spline` solamente se ha reimplementado la etapa de ejecución, en la que entre par de puntos de control se evalúa el interpolador cúbico un número de veces determinado por la variable `delta_t` de la clase, que indica la separación entre un punto y otro (la separación entre dos puntos de control es 1).

Una clase hipotética que se puede plantear es un graficador que realice interpolaciones cúbicas pero mediante la GPU. Esto es posible porque los puntos de control y las matrices de interpolación se pueden entregar a la GPU mediante la clase `CgShader`.

Capítulo 7

Resultados

Este Capítulo enumera los resultados de trabajo realizado a través de las siguientes Secciones:

Sección 7.1, Aplicación de Prueba: Se muestra la aplicación de prueba, cargando modelos tridimensionales estáticos y animados, con mallas de distintas características.

Sección 7.2, Análisis de Desempeño: Se analiza el desempeño de la aplicación de prueba en la extracción de líneas, en la graficación tradicional y en los graficadores implementados.

Todas las imágenes de este Capítulo han sido producidas utilizando la aplicación de prueba, exceptuando la imagen de la aplicación de prueba en sí.

7.1. Aplicación de Prueba

Si bien el resultado de la memoria es el mismo trabajo realizado en la memoria, la existencia de una aplicación de prueba permite verificar el funcionamiento del marco de trabajo de acuerdo a los términos planteados en los capítulos anteriores.

La aplicación de prueba es un visualizador de modelos tridimensionales en formatos .OBJ (estáticos) y .MD2 (animados), cuya interfaz gráfica se puede observar en la Figura 7.1. Funciona de la siguiente manera: una vez que se carga el modelo tridimensional, se procede a configurar una escena sencilla que contiene el modelo ya cargado y una fuente de luz. El modelo puede ser rotado y se puede cambiar el ángulo del campo de visión de la escena (se configura la matriz de proyección con esto), lo que equivale a escalar la graficación del modelo. Para el caso de los modelos animados se cargan todas sus animaciones, pero no se muestra la animación como tal, sino que se puede apreciar cada keyframe de manera estática. Se puede seleccionar uno de los graficadores implementados para visualizar el modelo.

Respecto a sus características técnicas, la aplicación corre sobre la biblioteca `Qt4`[1] para

[1]Disponible en `http://www.trolltech.com`.

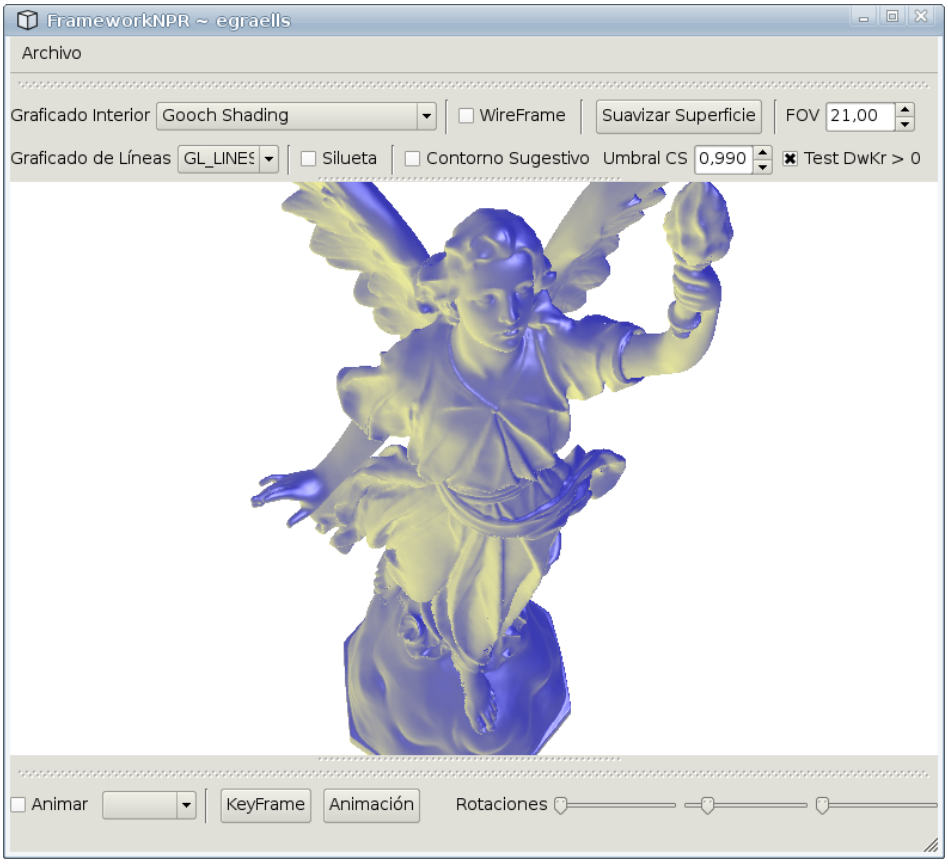


Figura 7.1: *Interfaz de la aplicación de prueba del marco de trabajo. Se muestra un modelo tridimensional de alta complejidad utilizando sombreado de Gooch.*

los elementos de interfaz y la gestión de la ventana. Esta biblioteca es de código abierto y multiplataforma.

7.1.1. Descripción de los Graficadores

De acuerdo al Capítulo 6, existen cuatro graficadores, dos de ellos tradicionales: el primero implementa el fixed pipeline, y el segundo implementa sombreado de Phong en la GPU, mediante el lenguaje Cg. También se entregaron dos graficadores NPR, que también utilizan la GPU: Cel-Shading y Gooch Shading.

En la aplicación de prueba estos cuatro graficadores están presentes. En particular existen dos instancias de Cel-Shading, una que recibe una textura 1D con dos tonos, y otra que recibe una textura con tres tonos. De Gooch Shading sólo existe una instancia, pero los colores de las tonalidades dependen de las propiedades de material del modelo que se esté visualizando.

7.1.2. Descripción de la Extracción de Líneas

En el caso de la extracción de líneas, ésta no se lleva a cabo si la superficie no es suave. Como inicialmente ningún modelo 3D lo es, el botón "Suavizar Superficie" realiza las estimaciones de las estructuras diferenciales en la malla del modelo.

Se utilizan los dos tipos de contorno implementados (siluetas y contornos sugestivos). Ambos contornos, si el usuario decide extraerlos, se dibujan mediante polilíneas de color negro. El usuario también puede decir aplicar una curva paramétrica a la polilínea, pudiendo elegir entre las dos implementadas (Catmull-Rom y B-Splines). En el caso de los contornos sugestivos se puede desactivar el test de corte.

7.1.3. Cargando Modelos 3D Estáticos

En la aplicación de prueba se cargaron los siguientes modelos estáticos:

Stanford Bunny – LOW Figura 7.2. Fuente: repositorio de escaneo 3D de la Universidad de Stanford[2]. Contiene 4.970 triángulos.

Se observa que se pueden extraer contornos sugestivos, a pesar del grosor de la malla del modelo, y extraer características de la superficie. Sin embargo, debido al muestreo insuficiente de datos, los contornos presentan mucho ruido.

Stanford Bunny – HIGH Figura 7.3. Fuente: repositorio de escaneo 3D de la Universidad de Stanford. Contiene 69.451 triángulos.

Se observa que la extracción de contornos sugestivos entrega muchas más líneas, ya que la superficie del modelo presenta una gran cantidad de concavidades. Asimismo, el fino muestreo evita que exista una gran cantidad de ruido.

Klein Bottle Figura 7.4. Generado utilizando la biblioteca `trimesh2`[3]. Contiene 120.000 triángulos.

La famosa botella de Klein es una superficie que no tiene interior ni exterior, aunque en su representación como modelo tridimensional no es *realmente* una botella de Klein, sino más bien solamente su representación gráfica, teniendo un exterior e interior determinados. A pesar de ello, resulta interesante su proceso con los algoritmos de contornos y su graficación.

Tigre Figura 7.5. Encontrado en la red[4]. Contiene 61.770 triángulos.

Este modelo, a diferencia de los anteriores, presenta propiedades de material (color rojo) y una textura que se aplica a la superficie, de color blanco con rayas negras. En casos como éste, no se recomienda el trazado de contornos sugestivos, ya que se mezclan con las rayas que posee la textura y se sobrecarga demasiado la representación visual.

[2] `http://graphics.stanford.edu/data/3Dscanrep/`

[3] RTSC está construido sobre esta biblioteca. Fuente: `http://www.cs.princeton.edu/gfx/proj/trimesh2/`

[4] `http://www.sharecg.com/v/13103/3d-model/feng-shui-tiger`

Se observa que la especificación del material rojo modifica las tonalidades que utiliza el sombreado de Gooch.

Tetera de Utah Figura 7.6 Encontrado en la red[5]. Contiene 26.026 triángulos.

Una de las tantas representaciones de la tetera de Utah. En este caso se utilizó una versión fina de la malla, siendo ésta la única que se pudo encontrar en la red que tuviese una calidad aceptable. La mayoría de las versiones de este modelo que se pueden encontrar son de mala calidad: o son demasiado gruesas o sus normales no son consistentes.

Knot Figura 7.7. Generado utilizando la biblioteca `trimesh2`. Contiene 20.000 triángulos.

Por su forma, esta superficie también es interesante geométricamente, aunque no tiene las propiedades de una botella de Klein. Sólo se le extraen siluetas porque los contornos sugestivos no son significativos.

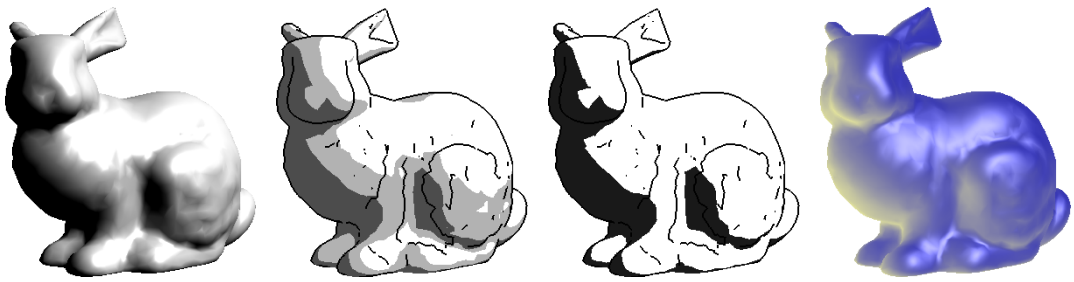


Figura 7.2: *Modelo estático: Stanford Bunny - LOW. De izquierda a derecha: sombreado de Phong, Cel-Shading con 3 tonos, Cel-Shading con 2 tonos y sombreado de Gooch.*

Figura 7.3: *Modelo estático: Stanford Bunny - HIGH. De izquierda a derecha: Cel-Shading con 3 tonos, Cel-Shading con 2 tonos y sombreado de Gooch.*

Estos modelos presentados son de diferentes naturalezas. Los dos primeros provienen de una fuente de datos real, es decir, de un escaneo tridimensional, y además el primero es una versión simplificada del segundo: es el resultado de un proceso aplicado a la malla. Los

[5] `http://www.sharecg.com/v/7584/3d-model/utah-teapot,-hires-obj-model`

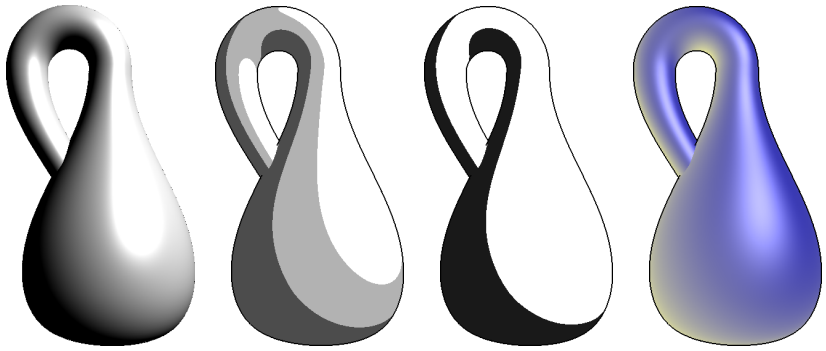


Figura 7.4: *Modelo estático: Klein Bottle. De izquierda a derecha: sombreado de Phong, Cel-Shading con 3 tonos, Cel-Shading con 2 tonos y sombreado de Gooch.*

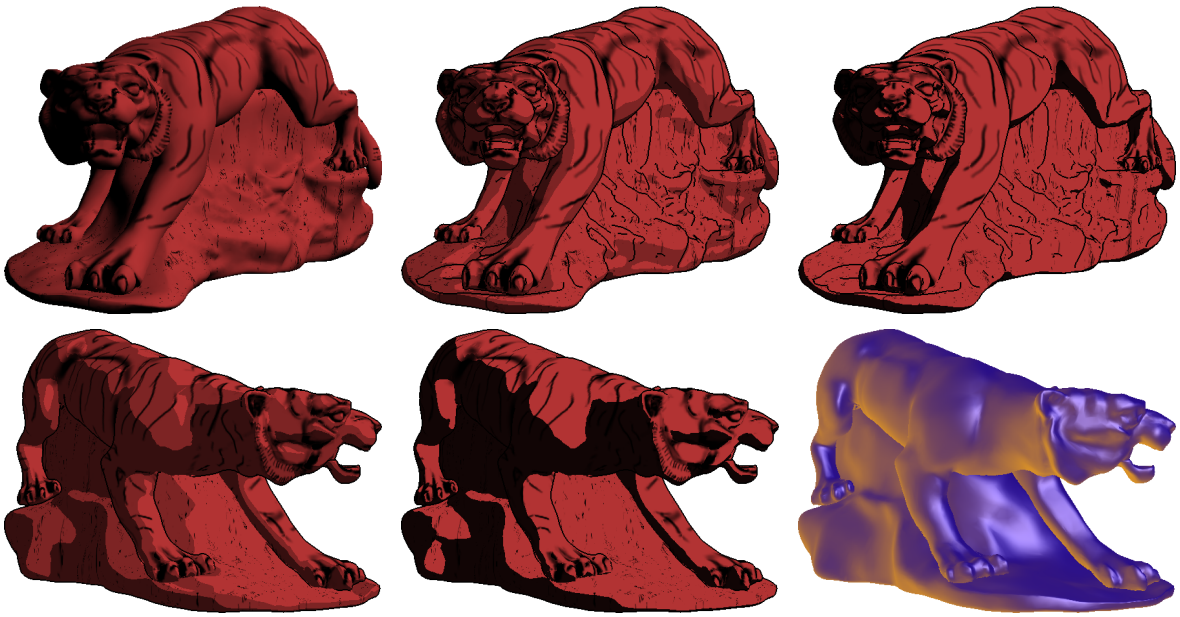


Figura 7.5: *Modelo estático: Tigre. En la fila superior: sombreado de Phong, Cel-Shading con 3 y 2 tonos. En la fila inferior: Cel Shading con 3 y 2 tonos, y sombreado de Gooch. El modelo incluye textura y propiedades de material rojo.*

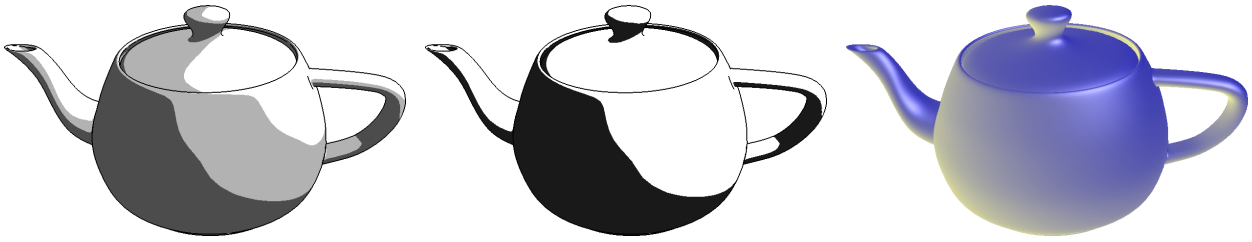


Figura 7.6: *Modelo estático: Tetera de Utah. De izquierda a derecha: Cel-Shading con 3 y 2 tonos, sombreado de Gooch.*

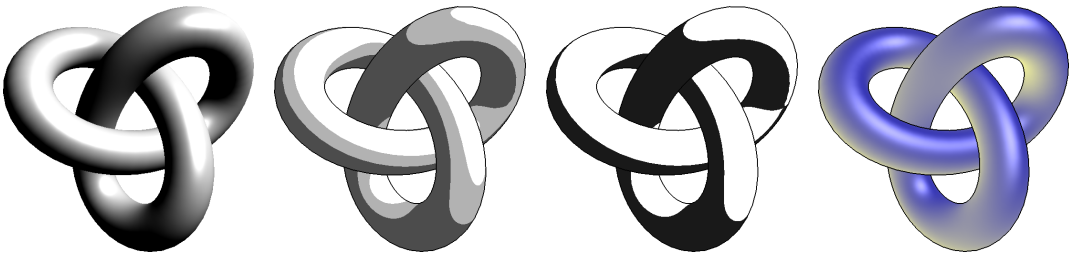


Figura 7.7: *Modelo estático: Knot. De izquierda a derecha: sombreado de Phong, Cel-Shading con 3 y 2 tonos, sombreado de Gooch.*

modelos del tigre y de la tetera fueron encontrados en la red y presentan algunos problemas de consistencia (en particular de aristas compartidas por más de dos triángulos) que el cargador de modelos .OBJ pudo identificar y pasar por alto. Los modelos de la botella de klein y del knot, a pesar de ser visualmente simples, tienen mallas difíciles de manejar por el concepto geométrico que representan.

A pesar del gran número de triángulos que estos modelos poseían, las estructuras presentadas en los Capítulos anteriores no colapsaron ni presentaron problemas: los graficadores implementados y los algoritmos de extracción de contornos, como muestran las figuras, se pudieron aplicar a todos ellos.

7.1.4. Cargando Modelos 3D Animados

En el caso de los modelos animados, solamente se implementó un cargador para el formado .MD2, ya que es el más sencillo de implementar. Esto, sin embargo, tiene sus consecuencias, ya que por limitaciones del formato el número máximo de triángulos en un modelo es 4,096, lo que no quiere decir que los modelos tengan ese número de triángulos: en la mayoría de las ocasiones el número es mucho menor, ya que al utilizarse en juegos, mientras menos triángulos poseían los modelos mejor era el desempeño.

En la aplicación de prueba se han cargado dos modelos animados:

Perelith Knight [6] Figura 7.8. Contiene 634 triángulos.

Este es un modelo *profesional*, ya que fue realizado por una compañía para un videojuego. Finalmente fue desechado porque cambiaron el motor gráfico. El modelo, en vez de pasar al olvido, fue liberado a la comunidad, por lo que es de uso libre y gratuito. Se ha utilizado en algunas publicaciones que utilizan animaciones y modelos tridimensionales como un modelo de prueba.

Marvin el Marciano [7] Figura 7.9. Contiene 787 triángulos.

A diferencia del modelo anterior, éste ha sido realizado por un usuario del juego y contiene inconsistencias en las aristas (aristas compartidas por más de dos triángulos).

[6] `http://planetquake.gamespy.com/View.php?view=ModeloftheWeek.Detail&id=22`
[7] `http://planetquake.gamespy.com/View.php?view=ModeloftheWeek.Detail&id=13`

Figura 7.8: *Modelo animado: Perelith Knight. Se muestran dos cuadros de animación.*

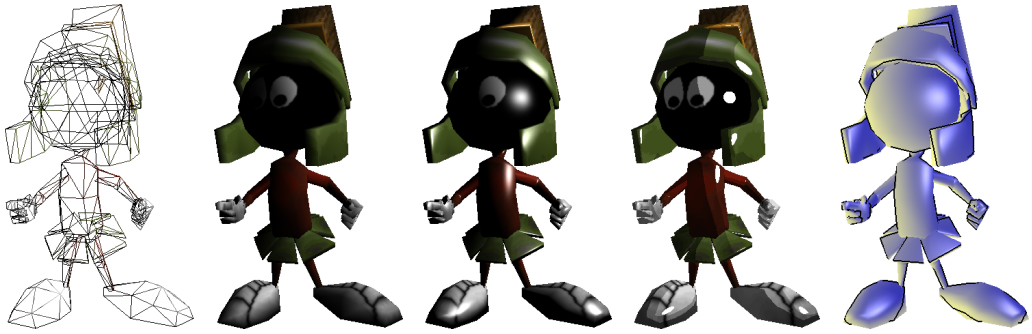


Figura 7.9: *Modelo animado: Marvin el Marciano. Se muestra el mismo cuadro de una animación, graficado con (de izquierda a derecha): modelo de alambres, sombreado de Gouraud, sombreado de Phong, Cel-Shading con 3 tonos, y sombreado de Gooch.*

La cantidad de triángulos no es necesariamente un índice de la finura de la malla. Si la forma del objeto a modelar es sencilla, entre 600 y 700 triángulos puede ser una cantidad aceptable de triángulos (sobretodo si se está trabajando con poliedros). Pero, en estos dos casos, las formas que se están representando son complejas, ya que son cuerpos con forma semi-humana, con extremidades, cabeza y movimientos, por lo que las mallas se pueden considerar muy gruesas. Las consecuencias de este grosor en las mallas son visibles: el sombreado de Gouraud es muy opaco, ya que la luz que llega a los vértices no es representativa de la luz que llega al interior de los triángulos; y por otro lado, la extracción de líneas no genera buenos resultados, porque la estimación de las estructuras diferenciales no posee suficiente información: las siluetas extraídas al modelo de Marvin no siempre representan un lugar exacto de silueta, como se observa en la Figura 7.9.

7.1.5. Muestra de Curvas Paramétricas

En la Figura 7.10 se observa el modelo de baja calidad del conejo de Stanford, con sombreado de Gooch, siluetas dibujadas con el *fixed pipeline* y contornos sugestivos utilizando el graficador de splines con los interpoladores de Catmull-Rom y B-Splines. Para cada graficador

se adjunta una imagen de los contornos sugestivos sin aplicar el test de corte.

Se observa que la graficación de contornos sugestivos con el *fixed pipeline* se ve perjudicada por la mala calidad de la malla, ya que adquieren una trayectoria de *zig-zag* que les resta significado y atractivo. Por otro lado, utilizar splines de Catmull-Rom tampoco ayuda, ya que si bien suaviza la curva, ésta sigue pasando por los puntos *zigzagueantes*, de lo cual se concluye que el problema no es la interpolación de la curva, sino que lo es el muestreo de datos a partir de las aristas. La graficación con B-Splines genera un resultado más cercano al deseado, en concordancia con las propiedades de dichas curvas.

7.2. Análisis de Desempeño

Uno de los objetivos del marco de trabajo es que sus componentes tengan desempeño en tiempo real. Para medir este desempeño, al comienzo de la función que realiza el proceso de rendering se inició un contador de milisegundos, y como última instrucción de aquella función se detuvo el contador y se calcula la siguiente fórmula:

$$fps = \frac{1000}{t}$$

Donde t es el tiempo transcurrido entregado por el contador. Así, se aproxima el número de cuadros por segundo a partir de la graficación de *un* cuadro. En el Cuadro 7.1 se aprecian los resultados de los distintos graficadores en los modelos estáticos, promediando la graficación de 100 cuadros de animación. Las pruebas se realizaron en un computador portátil con procesador Dual Core de 1,8 GHz, 1 GiB de memoria RAM, sistema operativo Ubuntu Linux 7.10, y tarjeta de vídeo nVidia 7400 GS.

Modelo	Tri.	Fixed P.	Phong	Toon	Gooch	Siluetas	Sil. y Cont. S.
BunnyL	4.970	125	125	125	125	89,33	47,62
BunnyH	69.451	14,49	14,49	14,49	14,49	6,06	3,28
Tetera	26.026	38,46	37,04	37,04	38,46	16,13	9,80
Knot	20.000	55,56	52,63	52,63	52,63	23,26	13,70
Tigre	61.770	13,33	13,70	13,89	15,15	6,25	3,37
Klein	120.000	9,35	9,09	9,26	9,35	4,13	2,53

Cuadro 7.1: *Rendimiento en* cuadros/segundos *de la aplicación de prueba para los diferentes modelos estáticos probados. En el caso de Cel-Shading (Toon), se obtuvo el mismo rendimiento para las dos instancias del graficador. En el caso de los Contornos Sugestivos, se realizó el test de corte.*

A partir de la información del cuadro se puede inferir lo siguiente:

- La graficadores que utilizan Cg entregan mejores resultados visuales sin sacrificar más tiempo. Ahora bien, el marco de trabajo no ha sido optimizado, por lo que el desempeño podría mejorar al usar la GPU.
- Un modelo de hasta 70.000 triángulos, sin textura, puede ser graficado en tiempo real sin extracción de líneas. Asimismo, si se utiliza una textura para el modelo, se pueden graficar hasta 60.000 triángulos con una tasa cercana a los 15 cuadros por segundo.

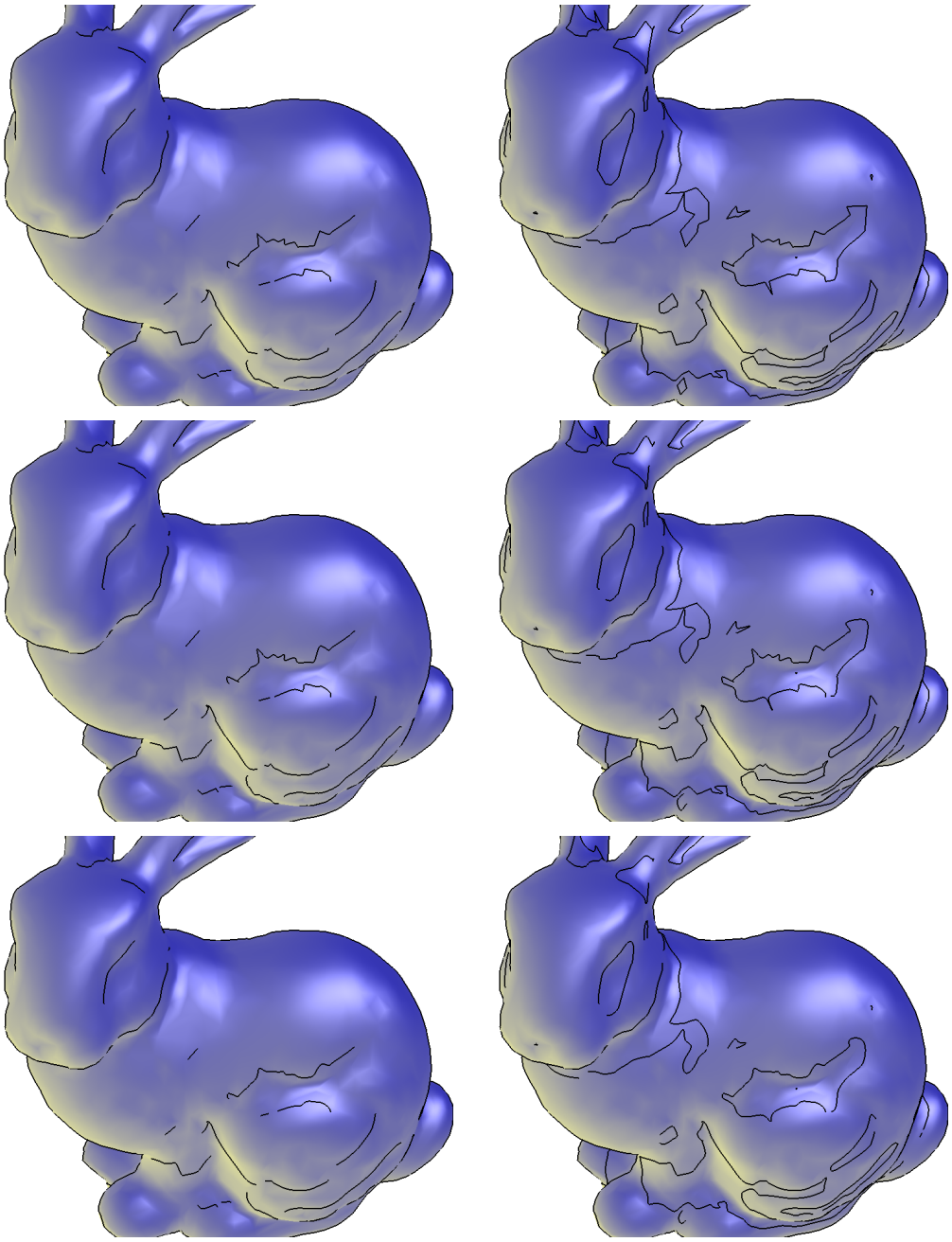


Figura 7.10: *Modelo estático con líneas de siluetas y contornos sugestivos. Las imágenes a la izquierda aplican el test de corte en los contornos sugestivos; las imágenes a la derecha, no. La fila superior muestra los contornos sugestivos graficados con el* fixed pipeline*; la fila central, con splines de Catmull-Rom; la fila inferior, con B-Splines.*

- Un modelo de hasta 20.000 triángulos puede ser graficado en tiempo real, con extracción de siluetas y contornos sugestivos, utilizando el algoritmo de recorrido completo de la malla.
- En el caso de Cel-Shading, las dos instancias del graficador obtuvieron el mismo resultado. Es decir, se pueden utilizar diversas texturas 1D con las intensidades de la iluminación y el desempeño será similar en todas las instancias.

Las cifras obtenidas son aproximadas, ya que dependen del contexto y de la configuración de la máquina que se esté utilizando. La resolución, el procesador gráfico, los procesos que se estén corriendo en segundo plano y el sistema operativo inciden en los resultados. Pero, a pesar de ello, estas cifran dan una idea del rendimiento que se puede obtener con el marco de trabajo en su primera versión.

Capítulo 8

Discusión y Conclusiones

En este Capítulo se analizan los resultados del trabajo realizado. Las Secciones que componen este análisis son las siguientes:

Sección 8.1, Cumplimiento de Objetivos: Se analiza el cumplimiento de los objetivos presentados en la memoria.

Sección 8.2, Puntos a Mejorar: Se destacan áreas del marco de trabajo que, a pesar de ser funcionales, requieren de mayor trabajo para ser óptimas.

Sección 8.3, Trabajo Futuro: Se enumeran ideas para completar el marco de trabajo en el futuro. Estas ideas no proponen realmente nuevas áreas de investigación, más bien complementan lo que ya se ha realizado con el fin de abarcar completamente la base teórica entregada en esta memoria.

Sección 8.4, Líneas de Investigación: Se proponen líneas de investigación para el futuro. A diferencia de la sección de Trabajo Futuro, las líneas propuestas no han sido abarcadas por completo por la investigación actual en el tema.

Sección 8.5, El Marco de Trabajo en Otras Áreas: Se comentan posibles extensiones y usos del marco de trabajo en contextos ajenos a NPR.

Sección 8.6, Conclusiones: Se realizan las conclusiones finales de esta memoria.

8.1. Cumplimiento de Objetivos

En el Capítulo 1, se enumeraron los objetivos de esta memoria. En este Capítulo se vuelven a enumerar, pero esta vez se comenta el grado de éxito en su cumplimiento.

8.1.1. Objetivos del Marco de Trabajo

Los objetivos del marco de trabajo son los siguientes:

- *Diseñar e implementar una estructura de datos que permita trabajar con modelos tridimensionales. Se debe implementar una estructura de datos flexible para representar distintos tipos de modelos tridimensionales, que a la vez permita la ejecución óptima de algoritmos geométricos y gráficos*: Cumplido. Se demostró que se puede trabajar con modelos tridimensionales estáticos y animados. Además, para los modelos estáticos las estructuras funcionan con un gran número de triángulos, y para cantidades razonablemente grandes se puede trabajar en tiempo real tanto con el *fixed pipeline* como con técnicas NPR y extracción de líneas.

- *Diseñar e implementar toda la funcionalidad necesaria para graficar modelos tridimensionales de manera tradicional*: Cumplido. Las estructuras de datos han sido diseñadas con los graficadores del *fixed pipeline* como base del proceso de rendering.

- *Proveer las herramientas para extraer líneas de interés desde de un modelo tridimensional*: Cumplido. Se implementó un algoritmo que extrae contornos o líneas de interés a partir de una malla de triángulos. Además, el algoritmo de extracción de recorrido de la malla puede ser reemplazado por nuevas estrategias de extracción, de manera transparente para los contornos definidos.

- *Proveer las herramientas que faciliten la definición de un tipo de línea de interés arbitraria, con el fin de implementar tipos de líneas que no se encuentren implementadas en el marco de trabajo*: Cumplido. El diseño de la componente de extracción de líneas y contornos permite expresar nuevos tipos de contornos sin necesidad de intervenir en el algoritmo de extracción. Como prueba, se implementaron dos tipos de contorno: uno sencillo (las siluetas) y otro complejo (los contornos sugestivos).

- *Proveer las herramientas para desarrollar estilos de NPR para el graficado estilizado y/o abstracto de un modelo tridimensional. Debe existir una base sobre la cual se puedan añadir las características propias de cada estilo, sin necesidad de re-implementar conceptos comunes a todos ellos*: Cumplido. Se implementaron graficadores que, teniendo como base el proceso de rendering tradicional, permiten implementar técnicas de NPR agregando la funcionalidad específica de cada técnica. Como prueba, se implementaron dos técnicas de NPR, que no reimplementan el proceso de graficación, sino que solamente añaden sus propios datos al proceso.

- *Proveer las herramientas para trabajar con los procesadores gráficos. Para el desarrollador es importante tener una conexión entre sus propias estructuras de datos, en este caso las del marco de trabajo, y las propias de la GPU*: Cumplido. La implementación de los graficadores, vista en el Apéndice A, da un ejemplo de como entregarle parámetros a la GPU directamente desde las estructuras de datos del marco de trabajo.

- *El marco de trabajo debe ser diseñado considerando orientación a objetos, de modo de ser extensible y reutilizable*: Cumplido. El diseño y la implementación de las componentes de visualización y extracción de líneas y contornos tiene como enfoques la extensibilidad y la reutilización.

- *El marco de trabajo debe utilizar bibliotecas gráficas y utilitarias estándares, con disponibilidad para múltiples sistemas operativos y de código abierto*: Cumplido. En los Capítulos anteriores se referenciaron todas las bibliotecas utilizadas: cada una de ellas

permite trabajar en múltiples sistemas operativos y todas son compatibles con la licencia GPL, incluyendo el lenguaje Cg.

Se concluye entonces que, a nivel general, el marco de trabajo cumple los objetivos planteados en esta memoria.

8.1.2. Objetivos de la Aplicación de Prueba

Los objetivos de la aplicación de prueba son:

- *Cargar diversos modelos tridimensionales y graficarlos con estilos gráficos realistas y no realistas variados*: Cumplido. En el Capítulo anterior se han cargado varios modelos, de diferentes características y distinta calidad. El marco de trabajo ha funcionado con todos ellos.
- *Permitir variar los parámetros con los cuales se grafican los modelos*: Cumplido. En el caso de los contornos sugestivos, se puede variar el umbral de tolerancia en la aplicación del test de corte. Sin embargo, en la implementación realizada, su variación no produce resultados muy notorios. También para los contornos sugestivos se pueden utilizar distintas curvas paramétricas para apreciar el comportamiento de la interpolación.
- *Ser una demostración del uso del marco de trabajo en aplicaciones gráficas*: Cumplido. Si bien a la aplicación de prueba le hacen falta características que le quiten su estado "de prueba", de modo de ser una aplicación completa, el uso de una biblioteca orientada a objetos y de código abierto hace que sea sencilla y directa su extensión hacia una aplicación real.

Se concluye entonces que, a nivel general, la aplicación de prueba cumplió los objetivos necesarios para ser una aplicación de demostración del trabajo realizado.

8.2. Puntos a mejorar

El haber cumplido los objetivos no quiere decir que el marco de trabajo sea óptimo y no se pueda mejorar. Ciertamente hay puntos mejorables, de los cuales se indican algunos a continuación:

- Se propone implementar **mejoras al algoritmo de extracción de curvas de nivel**. En su estado actual, es una implementación bastante ingenua que puede ser optimizada. Existen dos caminos posibles en este mejoramiento: el primero es implementar el **algoritmo de extracción aleatoria** [MKG+97], considerando que su base puede ser el algoritmo de recorrido de la malla. El segundo es agregar características a los puntos de las polilíneas extraídas, como puede ser un criterio de determinación de grosor de la línea. Así, se pueden implementar trazos de línea como los propuestos en [GVH07].

- Se propone implementar un graficador de líneas llamado **GraficadorDeLineasCg**. En particular, las interpolaciones mediante curvas paramétricas se pueden realizar en la GPU de modo eficiente considerando que se utilizan matrices de 4×4 y puntos de control pertenecientes a $\mathbf{R}^3$.

- Se propone implementar **filtros de imagen utilizando la GPU** [F+04]. Esto permitiría, en primer lugar, tener una alternativa a la detección de contornos, así como facilitar la implementación de **rendering tipo acuarela** [BKTS06].

- Respecto a la programación en el marco de trabajo, se propone **implementar el manejo de excepciones** para que los posibles usuarios puedan detectar y controlar errores en los modelos y escenas que se carguen en el marco de trabajo. En su estado actual, el marco de trabajo no maneja excepciones, solamente utiliza códigos de error y verificaciones antes de realizar acciones potencialmente peligrosas para la estabilidad de la aplicación.

- Se propone implementar la **extracción de aristas de contorno**. Esto no solamente ayudará a trabajar con técnicas NPR para modelos tipo poliedros, también servirá para la implementación de algoritmos de sombras (ya que éstos trabajan con las siluetas de los objetos) y para la visualización de contornos en modelos de mallas gruesas.

- Se propone **completar la animación de modelos con keyframes en la aplicación de prueba**, con el fin de ver en movimiento y no cuadro por cuadro las animaciones de los modelos. Los interpoladores entre cuadros ya están implementados, y solamente es necesario definir un criterio para pasar desde una animación a otra.

8.3. Trabajo Futuro

Este marco de trabajo se propone como extensible, lo que quiere decir que es factible y directo el añadir nuevas funcionalidades. De acuerdo al diseño presentado esto es posible siguiendo una metodología orientada a objetos. Ahora bien, ¿qué funcionalidad debería poder agregarse?¿Qué funcionalidad se sugiere agregar? Se sabe que no pueden dejarse de lado tareas como la **implementación todos los tipos de contornos mencionados** en esta memoria, así como otras técnicas de NPR, como el **achurado** y **painterly rendering**, pero estas tareas no son las únicas que se pueden sugerir. A continuación se enumeran otras tareas para ser implementadas en el corto plazo:

- Los algoritmos geométricos que se han implementado son sensibles a las mallas de poca calidad y al ruido. Sería interesante implementar **suavizamiento y refinamiento de los modelos tridimensionales**, con el fin de reducir los problemas de calidad de la malla. Se propone utilizar la información de las curvaturas para el refinamiento, y utilizar algunos criterios de área de triángulos, test de círculo y ángulo mínimo/máximo. Su implementación es similar a la de los estimadores ya presentes, que tienen acceso directo a las estructuras de datos de los modelos. En [BPK+07] se encuentran referencias y descripciones de las técnicas más importantes en esta área.

- Se propone implementar **Vertex Skinning**, así como la lectura de modelos en formatos animados más recientes. Se pueden seguir dos caminos: el primero es como construir un esqueleto a partir de una malla que no tenga uno [JT05], y el segundo es implementar estrategias para que el movimiento de la malla sea fiel al del esqueleto [LCF00], pero sin perder coherencia con el tipo de objeto que se está representando. Un ejemplo de estrategia para este problema es [KCvO07].

- Con el fin de mejorar el desempeño en tiempo real, se propone implementar heurísticas para **determinar tiras de triángulos (triangle strips)** a partir de las mallas. También simplemente se podrían reordenar los triángulos del modelo con el fin de aprovechar el caché de vértices de las GPU.

8.4. Líneas de investigación

Como trabajo futuro se sugieren las siguientes líneas de investigación:

- Se propone implementar la **extracción de isosuperficies** a partir de datos de volumen. Las isosuperficies se pueden graficar como modelos tridimensionales, por lo que su visualización ya está implementada: lo que falta es la lectura de los datos de volumen y la aplicación de un algoritmo tipo *Marching Cubes*. Precisamente ése es el enfoque aplicado en [BKR+05], donde a partir de la isosuperficie extraen siluetas y contornos sugestivos.

- Estudio de **técnicas NPR en modelos animados con coherencia temporal**. En general, cuando se trabaja con modelos animados bajo cambios de cámara, solamente se utiliza Cel-Shading en conjunto con el graficado de siluetas, pero calculadas por arista (como en un poliedro) y no en una superficie suave. Se propone estudiar metodologías para trabajar con otros tipos de contorno en mallas de triángulos que se deforman, considerando cambios de cámara. En [DFR04] se aplican conceptos de conherencia temporal para contornos sugestivos en condiciones de movimiento de cámara. Este enfoque puede ser extendido para considerar deformación de la geometría a través del tiempo.

- Diseño y creación de una **herramienta de diseño e implementación de técnicas NPR mediante un lenguaje intermedio o interpretado**, que permita trabajar con técnicas NPR sin necesidad de volver a compilar el marco de trabajo o crear nuevas clases. Además, esta herramienta podría presentar una **interfaz para trabajar visualmente en el diseño de las nuevas técnicas**. Esta aplicación se puede describir como una mezcla entre la propuesta de [KMM+02] y de [GTDS04]. La primera referencia permite definir estilos de graficación directamente en la pantalla al graficar el modelo; la segunda presenta una API para definir estilos de graficación de líneas.

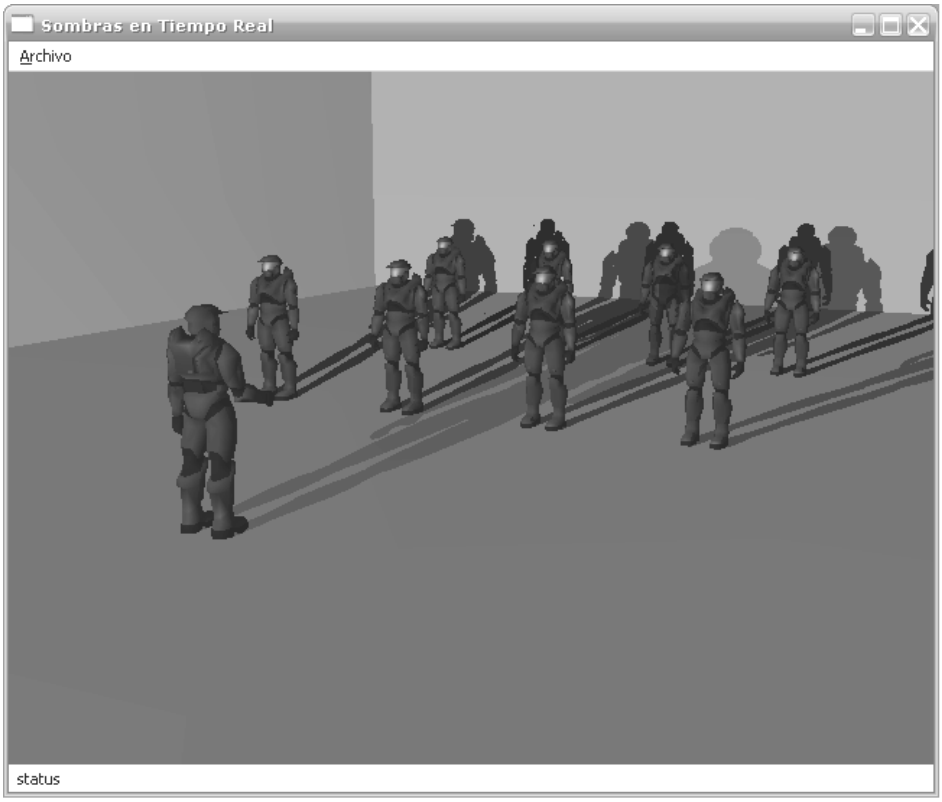


Figura 8.1: *Proyecto de Computación Gráfica de Simón Paredes, Semestre Primavera 2007: Implementación de algoritmo de sombras en tiempo real.*

8.5. El Marco de Trabajo en Otras Áreas

El trabajo futuro y líneas de investigación sugeridos se pueden enmarcar en el área de NPR, o bien, estar muy relacionados con NPR, pero el diseño del marco de trabajo permite trabajar en rendering tradicional. En el futuro, se propone implementar diversos efectos gráficos para visualizar los modelos cargados con el marco de trabajo. Entre estos efectos se encuentran los algoritmos de sombras, de reflexión y refracción, de profundidad de campo, entre otros.

En el Semestre de Primavera del año 2007, Simón Paredes[1], alumno del curso de Computación Gráfica (código CC52B), realizó una implementación de un algoritmo de sombras utilizando una versión preliminar del marco de trabajo. Su proyecto fue exitoso y obtuvo la nota 7.0, y puede observarse en la Figura 8.1.

El uso que dio el alumno al marco de trabajo confirma que es una base para poder implementar técnicas de Computación Gráfica, incluso no NPR, sobre la cual se puede construir una aplicación agregando la funcionalidad necesaria. El alumno sólo requirió concentrarse en las dificultades específicas del problema que tuvo que resolver. Además, realizó su proyecto en el sistema Microsoft Windows, mientras que el marco de trabajo ha sido desarrollado en Linux, lo que muestra su uso en múltiples plataformas.

[1]Contacto: `sparedes@dcc.uchile.cl`.

8.6. Conclusiones

En Computación Gráfica no solamente el diseño interno de una aplicación y su desempeño son importantes. La imagen producida en pantalla es tan o más importante que esos aspectos de una aplicación. Sin embargo, en base a las imágenes vistas en el Capítulo 7, se puede plantear la siguiente pregunta: ¿son atractivas estas imágenes? Ellas cumplen con el objetivo de ser una abstracción de lo que están representando, pero, ¿son abstracciones eficientes?

Ante esa pregunta, es necesario enfatizar que el objetivo global de esta memoria no es implementar graficadores de NPR ni configurar escenas tridimensionales que sean atractivas al ser graficadas con NPR, pues la graficación NPR es sensible a las escenas que se están graficando. Esto podría contradecirse con lo dicho en el Capítulo 1, "una técnica NPR es *independiente* del objeto que se está graficando", pero lo cierto es que no hay tal contradicción. La técnica funciona para cualquier modelo o escena, pero para producir una imagen eficiente, abstracta pero a la vez atractiva, es necesario adecuar algunas condiciones. Se puede establecer una analogía con una cámara fotográfica: puede sacar fotos *de cualquier persona o lugar*, pero es necesario que un fotógrafo sepa utilizarla. Las personas o los lugares no necesitan *adecuarse a la foto*, sino que al revés: el fotógrafo debe saber manipular las condiciones de la escena y de su cámara para obtener la mejor imagen posible, sea ésta abstracta o no.

El objetivo global del marco de trabajo es tener una base sobre la cual crear y configurar graficadores y escenas 3D. Para ello, primero fue necesario un estudio de las materias pilares del rendering no fotorrealista: en primer lugar la Computación Gráfica, vista en el Capítulo 3, y en segundo lugar la Geometría Diferencial y sus aplicaciones en Computación Gráfica, vista en el Capítulo 4. También fue necesario ver el estado actual de NPR: se describieron sus fundamentos históricos y luego se describieron las técnicas actuales, en conjunto con sus ideas y detalles principales; ello se vio en el Capítulo 5. Este estudio teórico, que puede ser considerado extenso para una memoria, es absolutamente necesario si el trabajo que se desarrolla requiere una base teórica fuerte. Sólo con esa base se pudieron tomar los elementos comunes de cada área que participa en NPR y unificar el trabajo en ellas. También se identificó un modo de extender esos elementos comunes con el fin de realizar implementaciones específicas.

El trabajo realizado planteado en el Capítulo 6 cumplió con los objetivos planteados en la memoria, como se concluyó en la Sección 8.1. Ante esto cabe preguntarse ¿cómo se puede considerar efectivo un marco de trabajo como tal? La implementación de un marco de trabajo, independiente de su naturaleza y propósito, tiene como mayor dificultad el tener un diseño que sea efectivamente extensible y reutilizable para diferentes tareas. Un marco de trabajo que sirve solamente para una tarea, cuando en realidad se planteó para más de una, es inefectivo y no sirve. Por lo tanto, no basta con cumplir los objetivos en una memoria como ésta para poder considerar que el trabajo, o al menos su etapa inicial, ha finalizado exitosamente. Los resultados expuestos, tanto por la aplicación de prueba como el uso por parte de un alumno de Computación Gráfica, permiten concluir que, en efecto, se ha desarrollado un marco de trabajo que apoya el desarrollo en NPR. La aplicación de prueba ha demostrado que la solución implementada es usable en una aplicación real, el diseño presentado tiene componentes extensibles que siguen la orientación a objetos, y el desempeño para modelos con una gran cantidad de triángulos es adecuado para tiempo real hasta una cota superior de 20.000 triángulos que, si bien no es una malla extremadamente grande, sí es suficiente para una gran cantidad de aplicaciones actuales.

Entonces, se podría decir que la conclusión final es que esta memoria ha finalizado con éxito. Se cumplieron los objetivos planteados, se demostró su cumplimiento, y se plantearon nuevos caminos por seguir. Pero el ciclo de vida de un marco de trabajo es mayor al de esta memoria, y el éxito de su implementación depende del uso y de la extensión que se haga de él en el futuro. Sólo el tiempo puede decir si el trabajo realizado en esta memoria ha sido verdaderamente exitoso, porque un marco de trabajo que no es utilizado, tanto por su autor como por otros desarrolladores, a pesar de tener una buena implementación, no tiene razón de ser. Así, es el tiempo quien tiene la última palabra y quien podrá juzgar si esta memoria ha llegado a buen término.

Bibliografía

[AMMH02] T. Akenine-Moller, T. Moller, and E. Haines. *Real-Time Rendering*. AK Peters, Ltd. Natick, MA, USA, 2002.

[Bau72] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[BKR+05] Michael Burns, Janek Klawe, Szymon Rusinkiewicz, Adam Finkelstein, and Doug DeCarlo. Line drawings from volume data. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 512–518, New York, NY, USA, 2005. ACM.

[BKTS06] Adrien Bousseau, Matthew Kaplan, Joëlle Thollot, and François Sillion. Interactive watercolor rendering with temporal coherence and abstraction. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*. ACM, 2006.

[BPK+07] Mario Botsch, Mark Pauly, Leif Kobbelt, Pierre Alliez, Bruno Lévy, Stephan Bischoff, and Christian Rössl. Geometric modeling based on polygonal meshes. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 1, New York, NY, USA, 2007. ACM.

[Bre77] Jack Bresenham. A linear algorithm for incremental digital display of circular arcs. *Commun. ACM*, 20(2):100–106, 1977.

[CAS+97] C.J. Curtis, S.E. Anderson, J.E. Seims, K.W. Fleischer, and D.H. Salesin. Computer-generated watercolor. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 421–430, 1997.

[dC76] M.P. do Carmo. *Differential geometry of curves and surfaces*. Prentice-Hall Englewood Cliffs, NJ, 1976.

[DFR04] Doug DeCarlo, Adam Finkelstein, and Szymon Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 15–145, New York, NY, USA, 2004. ACM.

[DFRS03] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 848–855, New York, NY, USA, 2003. ACM.

[DR07] Doug DeCarlo and Szymon Rusinkiewicz. Highlight lines for conveying shape. In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pages 63–70, New York, NY, USA, 2007. ACM.

[Edw07] Carolina Edwards. Aprendiendo a mirar, *Especiales EMOL*. Ediciones El Mercurio, 2007.

[EM03] D.S. Ebert and F.K. Musgrave. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[EMX02] Regina Estkowski, Joseph S. B. Mitchell, and Xinyu Xiang. Optimal decomposition of polygonal models into triangle strips. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, pages 254–263, New York, NY, USA, 2002. ACM.

[F+04] R. Fernando et al. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics*. Addison-Wesley, 2004.

[FK03] R. Fernando and M.J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[GGSC98] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452, New York, NY, USA, 1998. ACM.

[GTDS04] Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François Sillion. Programmable style for npr line drawing. In *Rendering Techniques 2004 (Eurographics Symposium on Rendering)*. ACM Press, june 2004.

[GVH07] Todd Goodwin, Ian Vollick, and Aaron Hertzmann. Isophote distance: a shading approach to artistic stroke thickness. In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pages 53–62, New York, NY, USA, 2007. ACM.

[HCV52] D. Hubert and S. Cohn-Vossen. *Geometry and the Imagination*. Chelsea Publishing Co., New York, 1952.

[Her98] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 453–460, New York, NY, USA, 1998. ACM.

[HS04] Michael Haller and Daniel Sperl. Real-time painterly rendering for mr applications. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 30–38, New York, NY, USA, 2004. ACM Press.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[JDA07] Tilke Judd, Frédo Durand, and Edward Adelson. Apparent ridges for line drawing. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 19, New York, NY, USA, 2007. ACM.

[JT05] Doug L. James and Christopher D. Twigg. Skinning mesh animations. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 399–407, New York, NY, USA, 2005. ACM.

[KCvO07] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 39–46, New York, NY, USA, 2007. ACM.

[KMM+02] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. Wysiwyg npr: drawing strokes directly on 3d models. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 755–762, New York, NY, USA, 2002. ACM.

[KP] J. Kautz and S. Pattanaik. Dynamic point distribution for stroke-based rendering.

[LCF00] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[LKL06] Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. Real-time pencil rendering. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 37–45, New York, NY, USA, 2006. ACM.

[LMHB00] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 13–20, New York, NY, USA, 2000. ACM.

[Max99] Nelson Max. Weights for computing vertex normals from facet normals. *J. Graph. Tools*, 4(2):1–6, 1999.

[McC98] S. McCloud. Understanding Comics: The Invisible Art. *IEEE TRANSACTIONS ON PROFESSIONAL COMMUNICATION*, 41(1):67, 1998.

[Mei96] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM.

[MGAK] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *International Conference on Computer Graphics and Interactive Techniques*, pages 896–907.

[MKG+97] Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. Real-time nonphotorealistic rendering. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[NM00] J. D. Northrup and Lee Markosian. Artistic silhouettes: a hybrid approach. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 31–37, New York, NY, USA, 2000. ACM.

[OBS04] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. Ridge-valley lines on meshes via implicit surface fitting. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 609–612, New York, NY, USA, 2004. ACM.

[RBD06] Szymon Rusinkiewicz, Michael Burns, and Doug DeCarlo. Exaggerated shading for depicting shape and detail. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1199–1205, New York, NY, USA, 2006. ACM.

[RDF05] Szymon Rusinkiewicz, Doug DeCarlo, and Adam Finkelstein. Line drawings from 3d models. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 1, New York, NY, USA, 2005. ACM.

[Rus04] Szymon Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Symposium on 3D Data Processing, Visualization, and Transmission*, 2004.

[S+97] B. Stroustrup et al. *The C++ programming language*. Addison-Wesley Reading, MA, 1997.

[SKCN03] S. Saito, A. Kani, Y. Chang, and M.Ñakajima. Generation of varying line thickness. *Computer Graphics International, 2003. Proceedings*, pages 294–299, 2003.

[SWND03] D. Shreiner, M. Woo, J.Ñeider, and T. Davis. The OpenGL Programming Guide (Red Book), 2003.

[WH05] A.P. Witkin and P.S. Heckbert. Using particles to sample and control implicit surfaces. *International Conference on Computer Graphics and Interactive Techniques*, 2005.

Apéndices

Apéndice A

Graficadores en Cg

Este Apéndice contiene el código fuente de los tres graficadores que derivan de `GraficadorCg`. Se incluyen sus definiciones de clase, implementaciones de clase, y programas en Cg.

Cada clase inicializa sus variables internas en su constructor. Además, indica a `GraficadorCg` las variables de OpenGL que le enviará a sus respectivos programas. En particular, los parámetros que se pueden utilizar son:

- La posición de la Fuente de Luz de la escena. Se entrega por omisión y su nombre de parámetro es `lightPosition`.
- La posición de la Cámara o Punto de Vista. Se entrega por omisión y su nombre de parámetro es `eyePosition`.
- La matriz de modelación y visualización (*modelview matrix*). No se entrega por omisión, su nombre de parámetro por omisión es `modelView`.
- La matriz de modelación y visualización premultiplicada por la matriz de proyección. Se entrega por omisión y su nombre de parámetro es `modelViewProj`.
- El nombre asignado a la textura que tiene el modelo (por omisión es `textura`) y un valor booleano que indica si el modelo tiene textura (por omisión es `tiene_textura`).

Los parámetros se entregan a Cg mediante la variable interna `shader` de `GraficadorCg`. Las clases que derivan de ella definen en su constructor (puede ser en el método `preparar` también, de acuerdo a la implementación) valores booleanos y nombres que definen si requieren cierto parámetro y bajo cuál nombre lo requieren. Para mostrar el uso de estos parámetros, en el código del sombreado de Phong se definen los parámetros con sus valores originales. Además, la técnica Gooch Shading no utiliza texturas ni materiales en la GPU, por lo que esos parámetros se desactivan en el constructor.

A.1. Sombreado de Phong

A.1.1. Código de clase

Phong.h

```
#include "../GraficadorCg.h"

class Phong : public GraficadorCg {
public:
        Phong();
        virtual ~Phong();
};
```

Phong.cpp

```
#include "Phong.h"

Phong::Phong():
GraficadorCg(
"Framework/Visualizacion/Graficadores/DeInterior/Phong/vertex_program.cg",
"Framework/Visualizacion/Graficadores/DeInterior/Phong/fragment_program.cg"
){
        requiere_posicion_luz = true;
        nombre_posicion_luz = "lightPosition";

        requiere_posicion_vista = true;
        nombre_posicion_vista = "eyePosition";

        requiere_modelview_proj = true;
        nombre_modelview_proj = "modelViewProj";

   nombre_textura = "textura";
}

Phong::~Phong(){}
```

A.1.2. Vertex Program

```
void main( float4 position     : POSITION,
           float3 normal       : NORMAL,
           float2 texcoord     : TEXCOORD0,
           out float4 oPosition : POSITION,
           out float3 objectPos  : TEXCOORD0,
           out float3 oNormal : TEXCOORD1,
           out float2 oTexcoord  : TEXCOORD2,
```

```
            uniform float4x4 modelViewProj
){
        oPosition = mul(modelViewProj, position);
        objectPos = position.xyz;
        oNormal = normal;
        oTexcoord = texcoord;
}
```

A.1.3. Fragment Program

```
void main( float3 position          : TEXCOORD0,
           float3 normal            : TEXCOORD1,
           float3 texcoord          : TEXCOORD2,
           out float4 color         : COLOR,
           uniform float4 lightPosition,
           uniform float4 eyePosition,
           uniform float3 Kd,
           uniform float3 Ks,
           uniform float3 Ka,
           uniform float shininess,
           uniform sampler2D textura,
           uniform int tiene_textura
){
   float3 P = position;
   float3 N = normalize(normal);
   float3 L = normalize(float3(lightPosition) - P);
   float3 V = normalize(float3(eyePosition) - P);
   float3 H = normalize(L + V);

   float3 lightColor = float3(1.0, 1.0, 1.0);

   float diffuseIntensity = max(dot(N, L), 0);
   float3 diffuse = Kd * lightColor * diffuseIntensity;

   float specularIntensity = pow(max(dot(N, H), 0), shininess);
   if (diffuseIntensity <= 0) specularIntensity = 0;
   float3 specular = Ks * lightColor * specularIntensity;

   float4 texture_color = float4(1.0, 1.0, 1.0, 1.0);
   if (tiene_textura > 0) texture_color = tex2D(textura, texcoord);
   color.xyz = diffuse * texture_color.xyz + specular;
   color.w = texture_color.z;
}
```

A.2. Toon Shading

A.2.1. Código de clase

Toon.h

```
#include "../GraficadorCg.h"

class Toon : public GraficadorCg {
public:
        Toon();
        Toon(const Textura &difusa, const Textura &especular);
        virtual ~Toon();
protected:
        void preparar(Modelo3D*);
        void finalizar(Modelo3D*);

        Textura intensidad_difusa;
        Textura intensidad_especular;
};
```

Toon.cpp

```
#include "Toon.h"

Toon::Toon():
GraficadorCg(
"Framework/Visualizacion/Graficadores/DeInterior/Toon/vertex_program.cg",
"Framework/Visualizacion/Graficadores/DeInterior/Toon/fragment_program.cg"
){
        float diffuse_ramp[] =
         {0.3f, 0.3f, 0.3f, 0.7f, 0.7f, 0.7f, 1.0f, 1.0f};
        float specular_ramp[] =
         {0.0f, 0.0f, 0.0f, 1.0f};
        intensidad_difusa = Textura(diffuse_ramp, 8);
        intensidad_especular = Textura(specular_ramp, 4);
}

Toon::Toon(const Textura &difusa, const Textura &especular):
GraficadorCg(
"Framework/Visualizacion/Graficadores/DeInterior/Toon/vertex_program.cg",
"Framework/Visualizacion/Graficadores/DeInterior/Toon/fragment_program.cg"
){
        intensidad_difusa = difusa;
        intensidad_especular = especular;
}
```

```
Toon::~Toon(){
        intensidad_difusa.eliminarTextura();
        intensidad_especular.eliminarTextura();
}

void Toon::preparar(Modelo3D* modelo){
        this->GraficadorCg::preparar(modelo);
        shader.parametroTextura("diffuseRamp", intensidad_difusa);
        shader.parametroTextura("specularRamp", intensidad_especular);
}

void Toon::finalizar(Modelo3D* modelo){
        shader.desactivarTextura("specularRamp");
        shader.desactivarTextura("diffuseRamp");
        this->GraficadorCg::finalizar(modelo);
}
```

A.2.2. Vertex Program

```
void main( float4 position     : POSITION,
           float3 normal       : NORMAL,
           float2 texcoord     : TEXCOORD0,
           out float4 oPosition : POSITION,
           out float3 objectPos  : TEXCOORD0,
           out float3 oNormal : TEXCOORD1,
           out float2 oTexcoord  : TEXCOORD2,
           uniform float4x4 modelViewProj
){
        oPosition = mul(modelViewProj, position);
        objectPos = position.xyz;
        oNormal = normal;
        oTexcoord = texcoord;
}
```

A.2.3. Fragment Program

```
void main( float3 position          : TEXCOORD0,
           float3 normal            : TEXCOORD1,
           float3 texcoord          : TEXCOORD2,
           out float4 color         : COLOR,
           uniform float4 lightPosition,
           uniform float4 eyePosition,
           uniform float3 Kd,
           uniform float3 Ks,
           uniform float3 Ka,
           uniform float shininess,
```

```
            uniform sampler2D textura,
            uniform sampler2D diffuseRamp,
            uniform sampler2D specularRamp,
            uniform int tiene_textura
){
    float3 P = position;
    float3 N = normalize(normal);
    float3 L = normalize(float3(lightPosition) - P);
    float3 V = normalize(float3(eyePosition) - P);
    float3 H = normalize(L + V);

    float3 lightColor = float3(1.0, 1.0, 1.0);

    float3 diffuseIntensity =
       tex2D(diffuseRamp, float2(max(dot(N, L), 0), 0.5));
    float3 diffuse = Kd * lightColor * diffuseIntensity;

    float3 specularIntensity =
       tex2D(specularRamp, float2(pow(max(dot(N, H), 0), shininess), 0.5));
    float3 specular = Ks * lightColor * specularIntensity;

    float4 texture_color = float4(1.0, 1.0, 1.0, 1.0);
    if (tiene_textura > 0) texture_color = tex2D(textura, texcoord);
    color.xyz = diffuse * texture_color + specular;
    color.w = texture_color.w;
}
```

A.3. Gooch Shading

A.3.1. Código de clase

Gooch.h

```
#include "../GraficadorCg.h"
#include "../../../Color.h"

class Gooch : public GraficadorCg {
public:
        Gooch();
        Gooch(Color &k_c, Color &k_w);
        virtual ~Gooch();
protected:
        void preparar(Modelo3D*);
        Color k_blue;
        Color k_yellow;
        float alpha;
        float beta;
```

```
};
```

Gooch.cpp

```
#include "Gooch.h"

Gooch::Gooch():
GraficadorCg(
"Framework/Visualizacion/Graficadores/DeInterior/Gooch/vertex_program.cg",
"Framework/Visualizacion/Graficadores/DeInterior/Gooch/fragment_program.cg"
){
        graficar_texturas = false;
        usar_materiales = false;
        k_blue = Color(0.0f, 0.0f, 0.5f, 1.0f);
        k_yellow = Color(0.5f, 0.5f, 0.0f, 1.0f);
        alpha = 0.2f;
        beta = 0.6f;
}

Gooch::Gooch(Color &k_b, Color &k_y):
GraficadorCg(
"Framework/Visualizacion/Graficadores/DeInterior/Gooch/vertex_program.cg",
"Framework/Visualizacion/Graficadores/DeInterior/Gooch/fragment_program.cg"
){
        graficar_texturas = false;
        usar_materiales = false;
        k_blue = k_b;
        k_yellow = k_y;
        alpha = 0.3f;
        beta = 0.7f;
}

Gooch::~Gooch(){}

void Gooch::preparar(Modelo3D* modelo){
        this->GraficadorCg::preparar(modelo);
        Color k_cool =
         k_blue + alpha * modelo->obtenerMateriales()[0]->obtenerKd();
        Color k_warm =
         k_yellow + beta * modelo->obtenerMateriales()[0]->obtenerKd();
        shader.parametroColor("k_warm", k_warm);
        shader.parametroColor("k_cool", k_cool);
}
```

A.3.2. Vertex Program

```
void main( float4 position     : POSITION,
           float3 normal       : NORMAL,
           float2 texcoord     : TEXCOORD0,
           out float4 oPosition : POSITION,
           out float3 objectPos  : TEXCOORD0,
           out float3 oNormal : TEXCOORD1,
           out float2 oTexcoord  : TEXCOORD2,
           uniform float4x4 modelViewProj
){
        oPosition = mul(modelViewProj, position);
        objectPos = position.xyz;
        oNormal = normal;
        oTexcoord = texcoord;
}
```

A.3.3. Fragment Program

```
void main( float3 position          : TEXCOORD0,
           float3 normal            : TEXCOORD1,
           float3 texcoord          : TEXCOORD2,
           out float4 color         : COLOR,
           uniform float4 lightPosition,
           uniform float4 eyePosition,
           uniform float3 k_warm,
           uniform float3 k_cool
){
   float3 P = position;
   float3 N = normalize(normal);
   float3 L = normalize(float3(lightPosition) - P);
   float3 V = normalize(float3(eyePosition) - P);
   float3 H = normalize(L + V);

   float diffuseIntensity = dot(N, L);

   float3 I_d = (1.0 + diffuseIntensity) / 2.0
      * k_cool + (1 - (1 + diffuseIntensity) / 2.0) * k_warm;

   color.xyz = I_d + pow(max(dot(N, H), 0), 16.0)
      * float3(0.5, 0.5, 0.5);
   color.w = 1.0f;
}
```

Apéndice B

Extracción y Graficación de Líneas

Este Apéndice contiene el código que muestra el proceso de extracción y graficación de líneas de contornos a partir de un modelo tridimensional.

Lo primero que debe tenerse en cuenta es que el algoritmo de extracción, definido en la interfaz `ContourGenerator`, recibe como parámetros un modelo tridimensional, mediante un puntero a una instancia de la clase `Modelo3D`, y un contenedor del tipo `std::vector` con punteros a instancias de la clase `Contorno`. Por ejemplo, una inicialización del vector de contornos podría ser la siguiente:

```
std::vector<Contorno*> contornos;
contornos.resize(2);
contornos[0] = new Silueta();
contornos[1] = new ContornoSugestivo();
```

A su vez, una inicialización del extractor de contornos podría ser:

```
ContourGenerator *extractor = new MeshTraversal(contornos);
```

Y el modelo tridimensional con el que se va a trabajar se podría cargar del siguiente modo:

```
Modelo3D *modelo = new Modelo3D("stanford_bunny.obj");
```

Para configurar la extracción se debe indicar a los contornos y al extractor el modelo que se utilizará:

```
extractor->definirModelo(modelo);
contornos[0]->definirModelo(modelo);
contornos[1]->definirModelo(modelo);
```

Para configurar un contorno en particular, como puede ser el umbral de extracción de los contornos sugestivos (umbral para el test $D_{\mathbf{w}}\kappa_r > 0$, se debe llamar a una función propia de la clase `ContornoSugestivo`. Esto se puede realizar fácilmente mediante un `dynamic_cast` desde el *scope* de la aplicación. No se ejecuta dentro del algoritmo de extracción.

```
ContornoSugestivo *cs = dynamic_cast<ContornoSugestivo*>(contornos[1]);
```

```
cs->definirUmbralPor(modelo->largoCaracteristico(), tolerancia_contorno);
```

Cuando ya se encuentren configurados tanto el extractor como los contornos,se necesita una instancia de `GraficadorDeLineas` para poder desplegarlos en pantalla. En este ejemplo, se utiliza un graficador de *fixed pipeline* y un graficador `Spline` con curvas de Catmull-Rom:

```
GraficadorDeLineas *gfp = new GraficadorDeLineas();
GraficadorDeLineas *spline = new Spline(
        new CatmullRom<Posicion<float,3> >()
        );
```

Además, se necesita un punto de vista para los contornos dependientes de la vista. Esta posición, instancia de `Posicion<float,4>` se debe especificar en coordenadas del modelo. En este ejemplo dicha posición se llama `posicion_camara_transformada`.

```
if (modelo->esSuperficieSuave()){
    extractor->extraerCurvas(posicion_camara_transformada);
    std::list<Polilinea> siluetas = extractor->obtenerCurvas(0);
    gfp->graficar(siluetas);
    std::list<Polilinea> c_sugestivos = extractor->obtenerCurvas(1);
    spline->graficar(c_sugestivos);
    }
}
```