Universidad de Granada
Departamento de Lenguajes y Sistemas Informáticos

Optimización de Algoritmos Geométricos Básicos Mediante el Uso de Recubrimientos Simpliciales

Doctorando

Carlos Javier Ogayar Anguita

Directores

Rafael Jesús Segura Sánchez
Francisco Feito Higueruela

Memoria presentada para la obtención del grado de Doctor en Informática

Granada, Mayo de 2006

Editor: Editorial de la Universidad de Granada
Autor: Carlos Ogayar Anguita
D.L.: Gr. 1178 - 2006
ISBN: 978-84-338-4005-9

La memoria titulada ***Optimización de Algoritmos Geométricos Básicos Mediante el Uso de Recubrimientos Simpliciales***, que presenta D. Carlos Javier Ogayar Anguita para optar al grado de Doctor en Informática, ha sido realizada en el Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada bajo la dirección de los doctores D. Francisco R. Feito Higueruela y D. Rafael Jesús Segura Sánchez.

Granada, Mayo de 2006.

El doctorando

Carlos Javier Ogayar Anguita

Los directores

Rafael Jesús Segura Sánchez

Franciso R. Feito Higueruela

Aviso Legal

3ds Max® es una marca registrada de AutoDesk, Inc.

Alias|WaveFront® es una marca registrada de Alias|WaveFront, una división de Silicon Graphics Ltd.

AltiVec™ es una marca registrada de Motorola, Inc.

AMD® es una marca registrada de Advanced Micro Devices, Inc.

Apple®, **Mac**® y **Macintosh**® son marcas registradas de Apple Computer, Inc.

Arial® es una marca registrada de The Monotipe Corporation plc.

ATI Rage® y **Radeon**® son marcas registradas de ATI Technologies, Inc.

Cinema4D™ es una marca registrada de MAXON Computer.

Cortona® es una marca registrada de Parallel Graphics, Ltd.

E&S® es una marca registrada de Evans & Sutherland Computer Corporation.

Graphics Blaster® es una marca registrada de Creative Technology Ltd.

IBM® y **PowerPC**™ son marcas registrada de IBM Corporation.

Intel®, **Pentium**® y **Pentium**® **4** son marcas registradas de Intel Corporation.

Java™ y **Java3D**™ son marcas registrada de Sun Microsystems Inc.

Linux® es una marca registrada de Linus Torvalds.

Microsoft®, **Windows**®, **Visual Studio**®, **Visual Basic**®, **MSDN**®, **Xbox**®, **DirectX**® y **Direc3D**® son marcas registradas de Microsoft Corporation.

NVIDIA®, el logo de NVIDIA, **3dfx**®, **Voodoo**®, **RIVA TNT**™, **TNT**™, **CineFX**® y **GeForce**® son marcas registradas de NVIDIA Corporation.

RenderMan® es una marca registrada de Pixar Animation Studios.

S3 Savage® es una marca registrada de VIA Technologies, Inc.

SGI®, **OpenGL**®, **GL**® y **Maya**® son marcas registradas de Silicon Graphics Ltd.

Times® y **Palatino**® son marcas registradas de Heidelberger Druckmaschinen AG, disponibles en Linotype Library GmbH.

Unix® es una marca registrada de The Open Group.

VideoLogic® y **PowerVR**™ son marcas registradas de VideoLogic Ltd.

El resto de marcas registradas y copyrights son propiedad de sus respectivos dueños.

Algunos modelos 3D utilizados en este trabajo han sido descargados de 3DCafe (http://www.3dcafe.com) y del repositorio 3D de la Universidad de Stanford (The Stanford 3D Scanning Repository; http://graphics.stanford.edu/data/3Dscanrep/). Los modelos *AngelWing* y *Console* utilizados en algunas imágenes han sido realizados por Sparky (The Singularity; http://www.merlin.net.au/~sparky).

Agradecimientos

Para completar este trabajo ha sido necesario emplear muchas horas de trabajo. Después de varios años dedicando prácticamente todo el tiempo disponible a la investigación, por fin esta tesis reúne las condiciones necesarias para ser finalizada.

Casi todos los trabajos de este tipo se realizan bajo la influencia de muchas personas, y este no es la excepción. Son muchos los que han aportado algo bueno (y a veces malo) a todo este proceso, de forma directa o indirecta. Probablemente olvide a alguno en estas líneas, pero a todos les debo mi más sincero agradecimiento.

Para empezar, quiero agradecer especialmente a Pedro Cano y a Juan Carlos Torres las oportunidades que me ofrecieron hace algún tiempo, y que me influenciaron para comenzar con una actividad investigadora que culmina con este trabajo.

Quiero agradecer la dirección, asesoramiento y continua ayuda de mis directores de tesis, Rafael Segura y Francisco Feito, que me han proporcionado una línea de investigación productiva y me han ayudado desde el principio a mejorar en todos los aspectos. También tengo que agradecerles las oportunidades que me han ofrecido y su *insistencia* para quedarme en el ámbito universitario un poco más y completar mi formación académica. También quiero agradecer a Francisco Velasco, tutor de esta tesis, su ayuda en las primeras fases del trabajo.

Tengo que agradecer a todos los revisores de las publicaciones sus críticas y comentarios, ya que me han orientado positivamente para trabajar cada vez mejor. Muy especialmente tengo que agradecer a Robert Joan-Arinyo y a Juan Carlos Torres las exhaustivas revisiones preliminares que hicieron de este trabajo, y que permitieron obtener un resultado de mayor calidad.

Me gustaría agradecer a todos mis amigos y a mis compañeros de trabajo su apoyo y colaboración en todo momento: Ángel Luis García, Francisco de Asís Conde, Antonio Rueda, Juanjo Jiménez, Juan Ruiz y tantos otros que me han ayudado incluso sin saberlo.

Por último y más importante, quisiera agradecer a mis padres y hermanos su apoyo incondicional durante todo el tiempo. A ellos les dedico esta tesis y mis futuros trabajos.

Parte del trabajo presentado en esta memoria ha sido financiado por la Unión Europea (fondos ERDF) y el Ministerio de Educación y Ciencia de España a través del proyecto de investigación TIN2004-06326-C03-03.

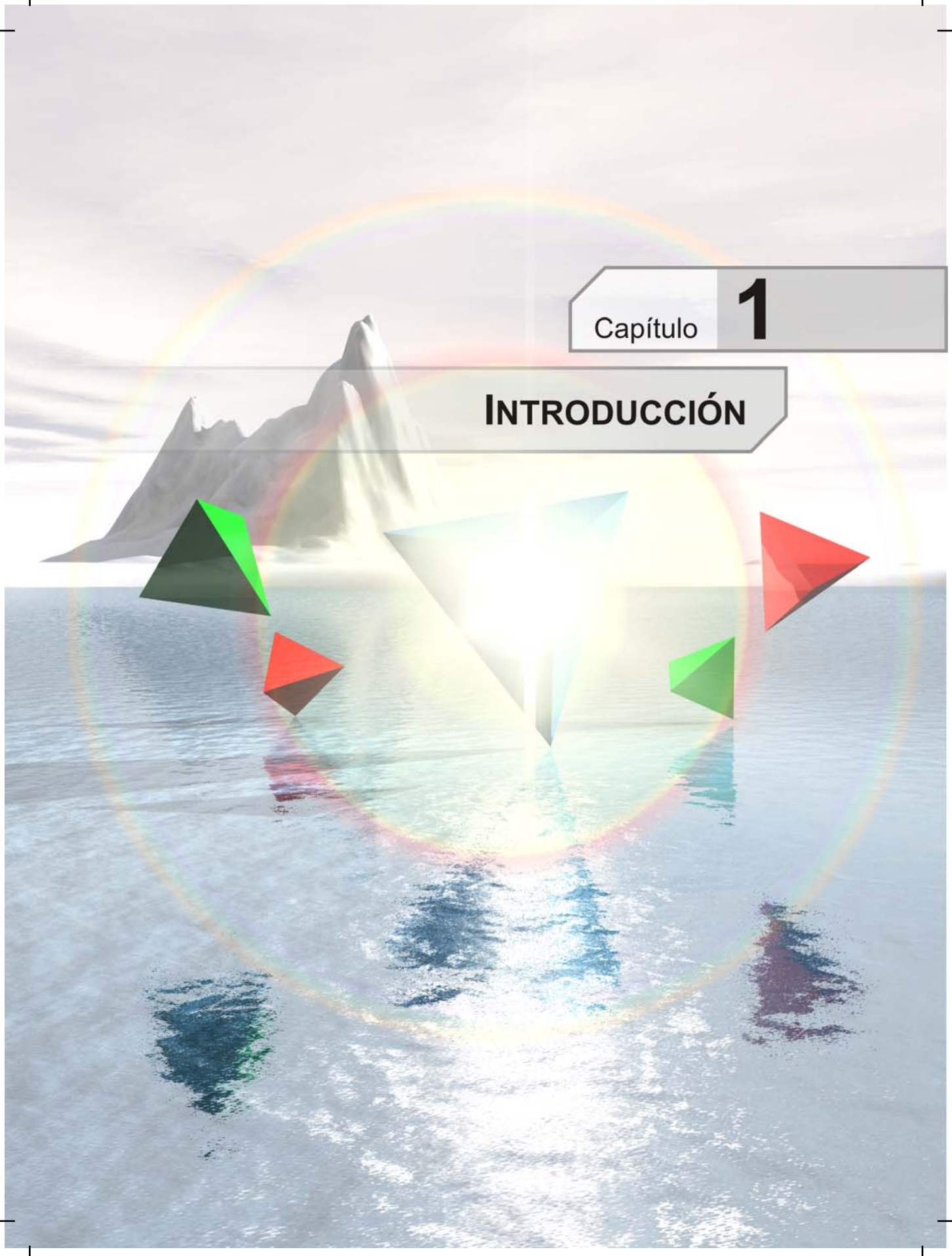
A mi familia

Contenido

Lista de figuras

Lista de tablas

Lista de Algoritmos



Capítulo 1

INTRODUCCIÓN

1

En este Capítulo se realiza una breve introducción al trabajo realizado que sirve para presentar las motivaciones que originaron la investigación, así como los objetivos planteados. También se expone un pequeño resumen del contenido de cada Capítulo para proporcionar un esquema de la estructura general de la tesis.

1.1 Presentación

El tema de este trabajo queda contenido dentro del ***Modelado de Sólidos***, que puede definirse como el conjunto de herramientas y técnicas destinadas a la representación y manipulación de objetos sólidos. Esta representación puede partir de un objeto real o virtual, y en cualquier caso permite obtener un modelo utilizando una estructura de datos procesable por un ordenador.

Las representaciones de sólidos permiten realizar un sinfín de operaciones sobre los modelos como si se efectuaran sobre los objetos reales o virtuales a los que representan. Esto permite la creación y manipulación de sólidos con el fin de extrapolar los resultados de todo tipo de simulaciones a objetos del mundo real. Por esta razón, el modelado de sólidos es una herramienta básica en el diseño y fabricación asistidos por ordenador.

Además de en el diseño industrial, el modelado de sólidos es una herramienta fundamental en la creación de objetos virtuales destinados a simulación, realidad virtual, videojuegos, etc. Todos los objetos que se definen en estos ámbitos tienen su base en actividades de modelado, ya sean manuales o procedurales.

Dentro del modelado de sólidos existe una gran cantidad de métodos de representación de objetos y de algoritmos para procesarlos. La conversión entre esquemas de representación de objetos es uno de los problemas básicos, así como la adaptación de los algoritmos que se aplican sobre los mismos. De este conjunto de herramientas, los algoritmos geométricos tienen una gran importancia, ya que sirven como base para otros más complejos y de mayor nivel.

Los algoritmos geométricos utilizados en modelado de sólidos cubren un amplio espectro de aplicación. Entre los más básicos destacan los métodos de inclusión e intersección de entidades geométricas, trazado de curvas, muestreo de superficies, etc. Hay otros problemas de mayor complejidad cuya resolución implica la utilización de las técnicas básicas. Uno de estos problemas es la evaluación de operaciones booleanas entre sólidos, que constituyen una herramienta básica en la construcción de objetos.

1.2 Motivación

La teoría de recubrimientos simpliciales presentada en los trabajos de Feito, Torres y Segura [Feito95b, Feito97, Segura01] permite representar objetos gráficos basándose en la descomposición de los mismos en símplices. Esta forma de representación puede aplicarse a objetos poliédricos ofreciendo un nuevo enfoque para la resolución de problemas relacionados con los mismos.

Los recubrimientos simpliciales son el fundamento de todos los algoritmos que se proponen en esta tesis. En este trabajo se presentan las bases que formalizan esta técnica. Más adelante se verá con más claridad su utilización práctica. Todo lo presentado aquí está basado en los trabajos de Feito, Torres y Segura [Feito95b, Feito97, Segura01].

Parte de los resultados obtenidos en esta tesis ya se presentaron en un trabajo anterior [Ogayar04], que sirve como punto de partida a todo el proceso de investigación. La utilización de recubrimientos simpliciales ofrece un nuevo enfoque para la resolución de algunos problemas geométricos del modelado de sólidos. Aunque en trabajos anteriores se habían presentado soluciones a estos problemas, realmente no se habían desarrollado implementaciones eficientes ni versiones optimizadas. Además, con el auge actual del hardware gráfico programable surgen nuevas posibilidades de

implementación de los métodos basados en recubrimientos simpliciales, ya que estos algoritmos se adaptan muy bien al procesamiento vectorial que ofrecen en parte las GPUs modernas.

1.3 Objetivos

Teniendo en cuenta todo lo anterior, los objetivos que se plantearon inicialmente para este trabajo son, por orden de prioridad, los siguientes:

- Estudio, implementación y optimización de varios algoritmos geométricos básicos utilizando la representación de sólidos mediante recubrimientos simpliciales. Entre estos problemas se encuentran la inclusión de puntos en sólidos, la voxelización de objetos y la realización de operaciones booleanas.

- Adaptar y optimizar los algoritmos desarrollados a mallas de triángulos, ya que esta forma de representación está muy extendida en casi todas las aplicaciones recientes.

- Comparar las técnicas desarrolladas con otras similares ya existentes. Se procurará que los métodos objeto de comparación sean de los más extendidos y referenciados en la literatura clásica y reciente.

- Adaptar los algoritmos desarrollados al hardware 3D existente en la medida de lo posible.

- Aplicar los algoritmos desarrollados a problemas específicos de mayor nivel dentro de la Informática Gráfica.

Cabe destacar que los objetivos principales se han cumplido en el plazo de tiempo esperado. Algunas tareas secundarias como las aplicaciones a otras áreas quedan pendientes como trabajos futuros.

1.4 Organización de la memoria

Este documento presenta de forma concreta el trabajo realizado durante varios años, así como los resultados obtenidos y las correspondientes conclusiones. Parte de la presente tesis ha sido publicada en diversas revistas y congresos nacionales e internacionales. El resto está en periodo de revisión para su publicación.

El Capítulo 2 presenta unos conceptos acerca del modelado de sólidos con objeto de enmarcar formalmente el resto del trabajo. Ya que esta disciplina es demasiado amplia, tan sólo se exponen los conceptos más cercanos a la temática del trabajo, no sólo por su ámbito sino también por las similitudes que puede presentar con algunos aspectos, incluyendo estructuras de datos, algoritmos, etc.

En el Capítulo 3 se exponen los fundamentos del modelado de sólidos basado en recubrimientos simpliciales. Ya que todos los algoritmos de este trabajo están basados en este tema, es importante ofrecer una introducción que permita entender la base formal de los algoritmos.

En el Capítulo 4 se propone una solución completa para el problema de la inclusión de puntos en sólidos. Se presenta el algoritmo basado en recubrimientos simpliciales junto con una serie de optimizaciones, además de una implementación en GPU novedosa. También se muestran los resultados de un estudio comparativo que incluye varios algoritmos de uso extendido. Este estudio ha sido realizado utilizando un software que establece las mismas condiciones para todos los métodos con el fin de obtener resultados concluyentes [Ogayar05].

El Capítulo 5 presenta varias soluciones basadas en recubrimientos simpliciales para el problema de la voxelización de sólidos. También se presentan diversos métodos para aumentar el rendimiento y varias implementaciones basadas tanto en el uso de operaciones optimizadas como en el uso del hardware gráfico actual. Al igual que en el Capítulo 4, los resultados son contrastados con implementaciones eficientes de otros métodos referenciados en la literatura actual.

El Capítulo 6 presenta un esquema completo para la evaluación de operaciones booleanas con mallas de triángulos. Este método está basado en trabajos anteriores. Sin embargo, la implementación y optimizaciones realizadas utilizando los nuevos enfoques presentados en capítulos anteriores ofrecen unos resultados muy superiores en términos de rendimiento a los obtenidos en trabajos previos. El algoritmo de evaluación de operaciones booleanas es un caso práctico de utilización del método de inclusión de puntos en sólidos del Capítulo 4. También se presenta un estudio comparativo que contrasta los resultados con soluciones comerciales.

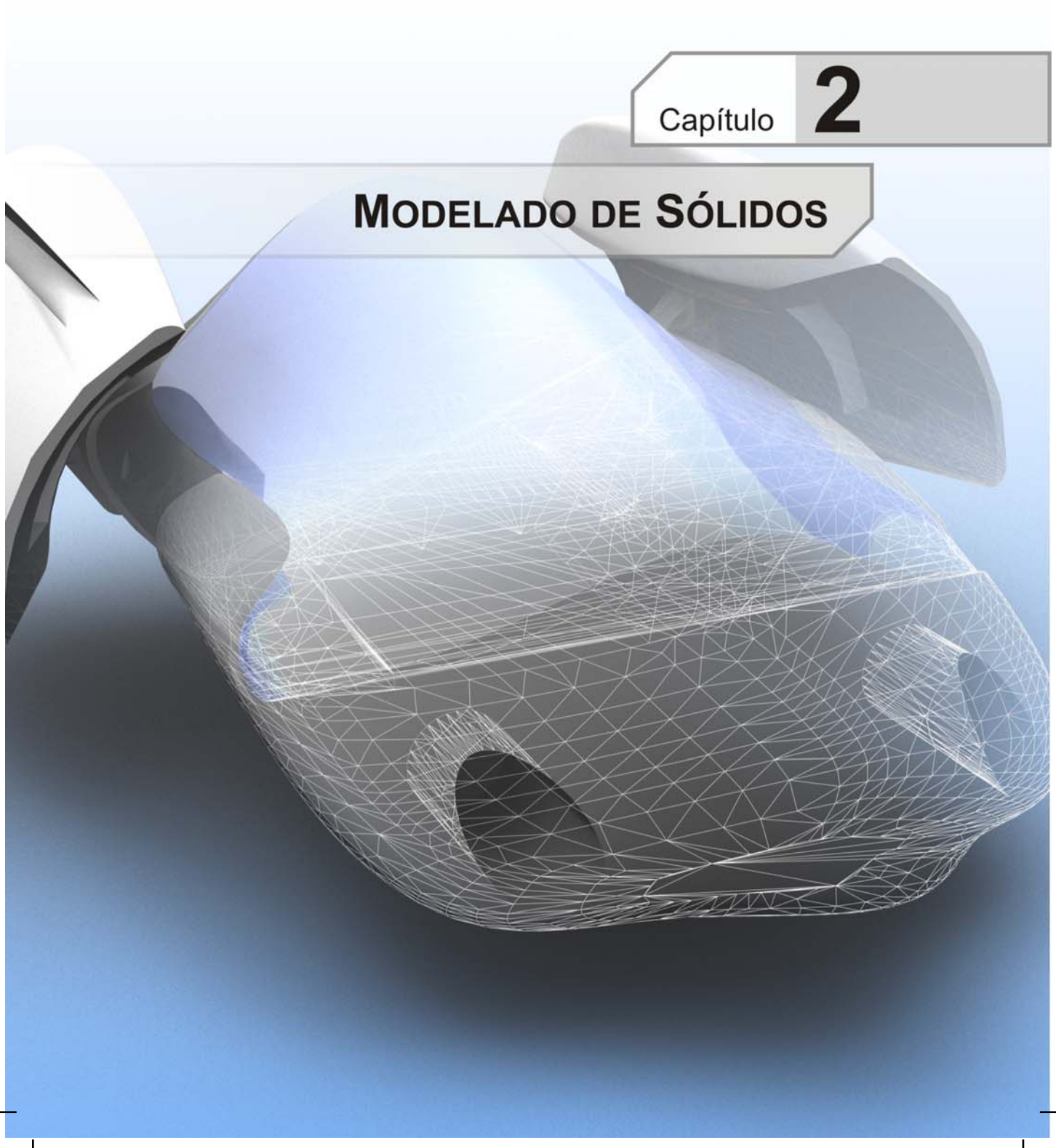
El Capítulo 7 presenta las conclusiones generales y resume las principales aportaciones de este trabajo. También se presentan posibles mejoras que pueden realizarse en el futuro, así como varias líneas de investigación que quedan abiertas para trabajos posteriores.

Por último, en los apéndices se presentan algunos aspectos del hardware gráfico programable y del software desarrollado. El objetivo es mostrar ciertos detalles

sobre la evolución y el estado actual del hardware gráfico, y de cómo ha influido en el desarrollo de este trabajo. También se exponen algunos aspectos sobre la estructura del software desarrollado y las características de los medios utilizados para las pruebas.

Capítulo 2

MODELADO DE SÓLIDOS



2

En este Capítulo se presenta un resumen de los aspectos más importantes del modelado de sólidos, que es el campo en el que se desarrolla este trabajo. Se describirán los esquemas de representación más extendidos, así como los problemas y soluciones más habituales, resaltando en especial los temas en los que se basa esta tesis.

2.1 Introducción

Un ***modelo*** puede definirse como un objeto artificial o virtual construido a semejanza de otro objeto real o virtual con el fin de realizar ciertas operaciones sobre el mismo. La construcción de un modelo supone una abstracción de la realidad y puede ser de dos tipos [Torres92]:

- **Modelos computacionales**. Son los que se representan utilizando un lenguaje de programación.
- **Modelos abstractos**. Son los que para el proceso de modelado se utiliza un lenguaje abstracto.

El concepto de ***modelado*** se aplica en muchos ámbitos de la computación. Básicamente consiste en realizar una representación de la realidad para simularla. En muchas ocasiones se realiza un modelo virtual de algo real para realizar operaciones sobre la representación artificial y extrapolar los resultados al mundo real. Lo que interesa en este trabajo es el proceso de abstracción de objetos o sólidos del mundo real mediante el modelado computacional.

El ***modelado geométrico*** es el campo de la Informática Gráfica que se encarga de estudiar la entrada, representación y consulta sobre objetos geométricos [Requicha88]. Un ***modelador geométrico*** es el software encargado de realizar las tareas propias del modelado geométrico empleando una interfaz de usuario para entrada de datos, estructuras de datos para las representaciones de los objetos y los algoritmos necesarios para la resolución de las operaciones que el usuario desee realizar sobre el modelo.

Dentro del modelado geométrico, el ***modelado de sólidos*** puede definirse como el conjunto de herramientas y técnicas destinadas a la representación y manipulación de objetos sólidos. Esta representación puede partir de un objeto real o virtual, y en cualquier caso permite obtener un modelo utilizando una estructura de datos procesable por un ordenador. Entre las operaciones más habituales que se realizan sobre los sólidos destacan el procesamiento de la superficie del modelo y de su interior, las operaciones entre distintos objetos, la visualización, el cálculo de propiedades, etc.

El modelado de sólidos es un campo muy estudiado [Baer79, Requicha80, Requicha82, Requicha83, Mortenson85, Hoffmann89, Brunet92, Rosiggnac99]. Las dos definiciones más interesantes son las siguientes:

- Según Requicha [Requicha83] el modelado de sólidos es "*El conjunto de teorías, técnicas y sistemas orientados a la representación completa en cuanto a la información de sólidos. Dicha representación debe permitir, al menos en un principio, calcular automáticamente cualquier propiedad bien conocida de cualquier sólido almacenado*".

- Según Mortenson [Mortenson85] un modelo geométrico de un sólido es una "*Representación matemática, no ambigua y completa, de la forma de un objeto físico tal que dicha representación pueda se procesada por un ordenador*".

La construcción de un modelo puede enfocarse desde tres niveles distintos de abstracción [Requicha80, Mäntylä88]:

1. **Objetos físicos**: No se puede percibir un objeto real con toda su complejidad, y por lo tanto tampoco se pueden representar todos los detalles en un ordenador para poder hacer cálculos sobre sus propiedades.

2. **Objetos matemáticos**. Son una idealización adaptada al objeto real. Esta abstracción de las características del objeto permite simular su comportamiento según unas hipótesis. Permiten establecer una conexión simple con el mundo real para su manipulación. Los objetos matemáticos son muy útiles para las simulaciones.

3. **Representación**. Una vez establecido el modelo matemático es necesario asignar una representación adecuada para el ordenador.

Por último, antes de introducir los conceptos teóricos del modelado de sólidos hay que mencionar los ***sistemas de modelado de sólidos***. Estos sistemas también denominados modeladores de sólidos son programas que proporcionan métodos para la creación, edición y visualización de sólidos a través de la gestión de diversas estructuras de datos que dan soporte a uno o varios esquemas de representación de sólidos. Un modelador suele disponer de una interfaz de usuario que permite realizar todas las operaciones posibles, o bien de una interfaz de programación (API) que permite hacer uso de la funcionalidad del modelador en forma de programa.

2.2 Fundamentos teóricos

A continuación se presentan algunos fundamentos teóricos que ayudan a establecer una base formal sobre la que se basa este trabajo.

En primer lugar hay que definir formalmente el concepto de sólido. El modelo matemático de un sólido tiene dos enfoques distintos [Mäntylä88]:

1. Se puede definir un objeto sólido en el espacio euclídeo mediante la *topología puntual*.

2. Se puede definir un objeto por su contorno mediante la *topología algebraica*.

2.2.1 Topología puntual

Según este enfoque un sólido es un subconjunto S, cerrado y acotado, del espacio euclídeo E^3 [Mäntylä88].

Para que un subconjunto de puntos cerrado y acotado sea considerado sólido debe satisfacer las siguientes condiciones [Requicha80]:

a) **Rigidez**. Un sólido debe ser invariante ante transformaciones rígidas, esto es, translaciones y rotaciones.

b) **Regularidad**. Todo el sólido debe estar compuesto por materia sin tener caras, aristas o puntos aislados. Esto puede formalizarse a partir del concepto de *regularización de un conjunto de puntos A*, *r(A)*, que se define como:

$$r(A) = clausura(interior(A))$$

El conjunto *A* es *regular* si $A = clausura(A)$. Un conjunto regular y acotado se denomina ***r-set***.

c) **Representación finita**. Para poder representar la información en el ordenador ésta debe ser finita.

Todo lo anterior implica que los modelos sólidos válidos son subconjuntos de R^3 limitados, cerrados, regulares y semianalíticos, lo que implica que la frontera debe estar bien definida [Requicha88]. Como se mencionó anteriormente, estos conjuntos se denominan *r-sets*.

2.2.2 Topología algebraica

Este enfoque intenta definir el sólido mediante información sobre su superficie [Mäntylä88]. Este modelo asume que el sólido tiene un interior homogéneo y no se almacena información sobre el mismo. Por tanto, la única información que se almacena es la que representa a la superficie del sólido que lo delimita del exterior.

Para que la representación sea correcta deben satisfacerse una serie de condiciones que garanticen una topología correcta. La superficie representada del sólido debe ser continua, completa y cerrada, no debe intersectar consigo misma y debe separar perfectamente el interior del exterior. A esta superficie se la conoce como la *frontera* del sólido [Mäntylä88].

Mediante este enfoque se realiza una definición matemática del sólido en función de la superficie (2D) en lugar del volumen (3D). La representación resultante es muy conveniente desde el punto de vista computacional, tanto para el almacenamiento como para el procesamiento del sólido [Kaufman94].

2.2.3 Definición de sólido

Se pueden aunar los dos enfoques anteriores (el puntual y el algebraico) para especificar las características que debe tener un objeto para ser considerado como sólido. Así, definiremos ***sólido*** [Requicha80] como un subconjunto del espacio euclídeo R^3 que cumple las siguientes condiciones:

1. **Rigidez**. El sólido tiene una forma fija e invariante que es independiente de su posición y orientación.

2. **Homogeneidad**. Todo el sólido está compuesto por material sin presentar puntos, aristas o caras aisladas, esto es, debe ser un conjunto *regular*.

3. **Finitud**. Debe ocupar una porción finita en el espacio.

4. **Representable de forma finita**. El sólido debe ser susceptible de ser representado mediante una secuencia finita de datos.

5. **Invariante**. Debe ser invariante respecto a transformaciones geométricas.

6. **Delimitado por el contorno**. Debe existir siempre una frontera que delimite el interior del sólido del exterior.

2.2.4 Operaciones regularizadas

Ya que los sólidos pueden combinarse mediante operaciones booleanas [Requicha85], pueden aparecer casos en los que no se cumpla la condición de regularidad. Por tanto, es necesario utilizar ***operaciones booleanas regularizadas*** (notadas por $\cup^*,\cap^*,-^*$) que sí garantizan que los resultados de las operaciones entre sólidos siguen siendo sólidos válidos [Foley96]. La Figura 2-1 muestra un ejemplo de operación regularizada.

Con las definiciones vistas hasta ahora, se puede obtener una operación regularizada como:

$$A\ op^*\ B = clausura\ (\ interior\ (\ A\ op\ B\ )\ )$$

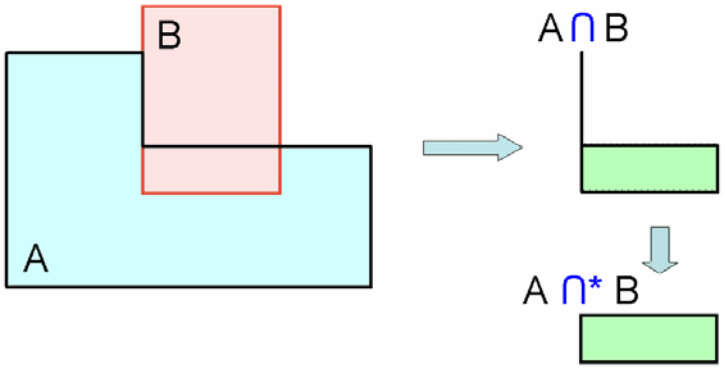


Figura 2-1. Ejemplo de intersección no regularizada (∩) y regularizada (∩) en 2D.*

2.3 Esquemas de representación

Una vez que se han determinado los elementos sobre los que trata el modelado de sólidos, se presenta a continuación la forma de representación de los objetos. El concepto de ***representación*** está vinculado al concepto de abstracción mencionado con anterioridad. En la Figura 2-2 puede verse que dentro del espacio de objetos R^3, sólo un subconjunto de ellos es válido dentro del modelado de sólidos. De igual forma, sólo una parte de los objetos representables en un ordenador serán una representación de los objetos del conjunto *D*.

Formalmente, un esquema de representación puede definirse como una relación $S : M \rightarrow R$, siendo *R* el conjunto de expresiones sintáctica y semánticamente correctas que definen el modelo en términos computacionalmente. De todos los esquemas de representación posibles, sólo deben considerarse aquellos que son únicos y no ambiguos, lo que quiere decir que:

$$\forall m \in M, \exists v \in R / S(m) = v$$

Pueden enunciarse múltiples formas de representar los sólidos, sin embargo estas representaciones deben cumplir ciertas propiedades formales que están directamente relacionadas con las propiedades que deben cumplir los sólidos. El esquema de representación elegido para el modelado de sólidos debe cumplir las siguientes condiciones [Requicha80]:

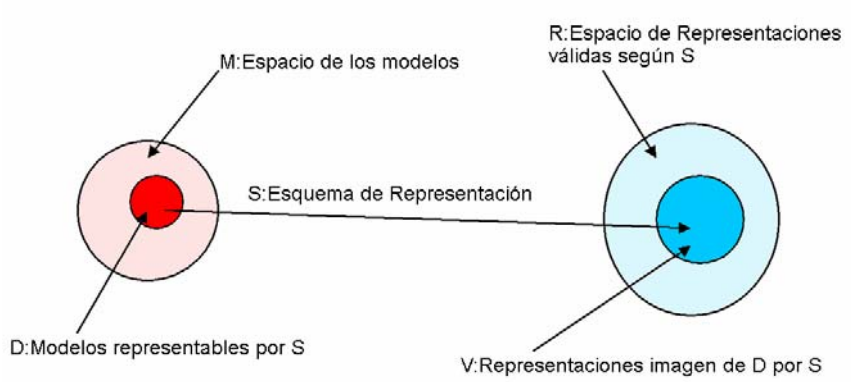


Figura 2-2. Dominio y rango de un esquema de representación.

- **Dominio**. El dominio *D* es el conjunto de objetos representables por el sistema, es decir, determina la capacidad descriptiva del esquema de representación. Cuanto mayor sea el conjunto de objetos, mejor será el esquema de representación. En cualquier caso, es la aplicación la que deberá determinar el conjunto de sólidos posibles y el esquema de representación acorde.

- **Validez**. El conjunto de todas las representaciones válidas del sistema *R* debe coincidir con el rango de la relación *S*. Además, debe ser el propio sistema de modelado utilizado el que se encargue de comprobar en cada momento la validez de los objetos representados.

- **Ausencia de ambigüedad**. Una representación debe definir un único objeto. No es conveniente utilizar esquemas de representación que permitan el uso de una misma representación para varios objetos.

- **Unicidad**. Un objeto del espacio de los modelos debe tener una única representación posible utilizando el esquema elegido. Esto es muy difícil de conseguir en la práctica, ya que objetos iguales es posiciones diferentes tendrán la misma representación, lo que se soluciona mediante el uso de transformaciones geométricas para complementar la representación de las instancias del objeto. Si no se verifica la unicidad no será posible establecer criterios de igualdad o desigualdad entre objetos.

Estos principios son los mismos que se aplican en la abstracción de datos a la hora de construir tipos de datos abstractos. Esto se debe a que el modelado de sólidos

es una especialización del proceso general de modelado computacional. Además de las propiedades necesarias enunciadas con anterioridad, es conveniente que el esquema de representación elegido cumpla con una seria de criterios adicionales [Requicha80]:

- **Concisión**. La representación debe ser lo más compacta posible evitando la redundancia en los datos. Esto es difícil, ya que en la mayoría de los casos es necesario tener información extra y redundante para optimizar y acelerar diversas operaciones que se realizan sobre los modelos.

- **Facilidad de edición**. Habitualmente los datos son introducidos mediante un proceso guiado o realizado por completo por un ser humano, por lo que se hace necesario un subsistema de entrada de datos accesible.

- **Eficacia**. Cada esquema de representación tiene un campo de aplicación concreto, por lo que debe ser eficaz en ese campo de aplicación. La eficacia se valora en términos de capacidad de representación de los sólidos y la eficiencia de las operaciones que se realizan sobre los mismos.

Los modeladores de sólidos deben realizar multitud de operaciones sobre los objetos. Esto conlleva la utilización de esquemas de representación adecuados a las operaciones a realizar. Por tanto, además de definir varios sistemas de representación, también hay que establecer mecanismos de conversión entre representaciones.

Teniendo en cuenta todo lo anterior, los esquemas de representación a ser empleados dentro del modelado de sólidos se dividen en varios grupos. Según Requicha [Requicha80] son:

1. *Esquemas ambiguos*. Como por ejemplo dibujos o proyecciones 2D de diseños industriales.

2. *Instanciación de primitivas*.

3. *Enumeración espacial*.

4. *Descomposición en celdas*.

5. *Geometría Constructiva se Sólidos (CSG).*

6. *Barrido y traslación.*

7. *Representación basada en fronteras (B-Rep).*

Además, es posible obtener esquemas de representación mixtos según las necesidades del modelador utilizado. Esta clasificación puede simplificarse agrupando varias categorías utilizando los dos modelos matemáticos de sólidos establecidos [Shapiro01].

- **Representaciones implícitas y constructivas**: establecen las reglas para comprobar qué puntos pertenecen al conjunto y cuáles no; tales representaciones son naturalmente soportadas por el modelo de sólidos de conjuntos de puntos continuos. En este grupo se incluyen la instanciación de primitivas, la Geometría Constructiva de Sólidos y las operaciones de barrido y traslación.

- **Representaciones enumerativas y combinatoriales**: especifican las reglas para los puntos generados en el conjunto (y no otros puntos). Estas representaciones se acercan más a los modelos de sólidos combinatoriales. En este grupo se incluyen los esquemas de enumeración espacial y de descomposición de celdas, así como la representación basada en fronteras (B-Rep).

A continuación se presentan los esquemas de representación más interesantes, prestando especial atención a los utilizados en este trabajo.

2.3.1 Instanciación de primitivas

Mediante la ***instanciación***, los sólidos se definen como un conjunto de primitivas parametrizadas predefinidas que son replicadas variando sus parámetros. Las propiedades se calculan mediante procedimientos propios de cada primitiva en función de los parámetros de definición. No hay mecanismos de combinación entre objetos, por lo que el dominio está completamente limitado al conjunto de primitivas de partida. La validez del modelo queda asegurada por los parámetros que lo definen. Es un esquema no ambiguo y no único.

2.3.2 Operaciones de mezcla y barrido

Existen varios sistemas de creación de superficies y sólidos basados en algoritmos que buscan la creación de sólidos de manera fácil y rápida. Entre estos sistemas destacan las operaciones de ***mezcla*** [Woodwark87], ***barrido*** [Hui94] y las ***operaciones de Minkowski*** [Elber99]. Todas tienen aplicaciones en el diseño mecánico e industrial, sin embargo, no siempre garantizan la validez del sólido resultante y además son difíciles de integrar en otros esquemas muy utilizados como el CSG.

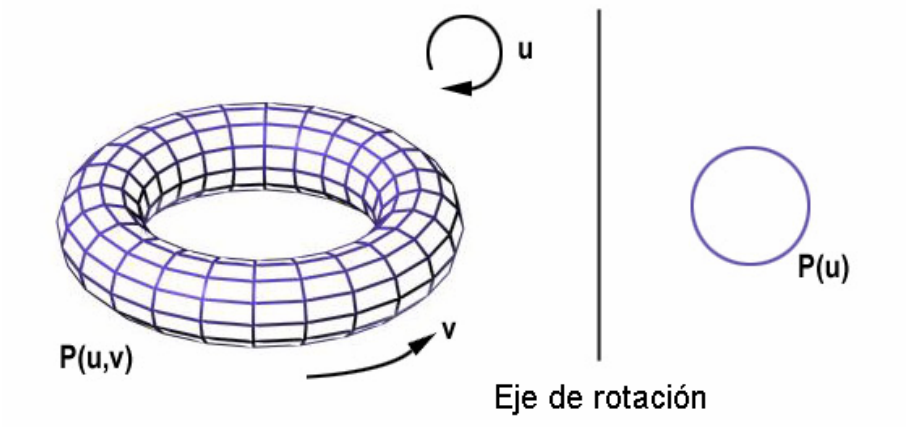


Figura 2-3. Ejemplo de operación de barrido rotacional.

La más popular de estas construcciones es la representación por barrido. Consiste en definir un sólido mediante una superficie bidimensional a lo largo de una trayectoria tridimensional. Los tipos de representación por barrido más destacados son el barrido traslacional y el rotacional. El primero consiste en desplazar una superficie bidimensional a través de la perpendicular al plano en el que está contenida hasta un punto determinado. En el segundo caso, se desplaza la superficie bidimensional siguiendo una trayectoria circular alrededor de un eje de rotación. Una extensión del barrido consiste en hacer que el tamaño y la orientación de la superficie cambien a lo largo de la trayectoria, obteniendo objetos con estrechamientos y torceduras (twist). Además, se pueden realizar barridos generales en los que tanto la trayectoria como la superficie utilizada son arbitrarias. La Figura 2-3 muestra un ejemplo de barrido rotacional.

La principal ventaja de este tipo de representación es la facilidad para la creación de modelos, ya que basta con definir una curva o superficie y una trayectoria. Sin embargo, el dominio de este tipo de representaciones está limitado a aquellos sólidos que presentan algún tipo de simetría rotacional o traslacional. Por tanto, más que un esquema de representación, estos sistemas son considerados como herramientas para la generación de sólidos basados en otros esquemas más completos como B-Rep o CSG.

2.3.3 Geometría Constructiva de Sólidos

La ***geometría constructiva de sólidos*** (CSG) es la representación constructiva más extendida por su versatilidad. Es un esquema que representa al sólido a través de la combinación de operaciones regularizadas de conjuntos de sólidos que normalmente suelen ser muy elementales. A estos sólidos básicos se les llama *primitivas* [Requicha78].

Las primitivas básicas utilizadas normalmente son sólidos parametrizados de figuras simples como cubos, cilindros, conos, etc., predefinidos internamente como semiespacios limitados. De este modo no se pueden construir objetos no limitados. Las primitivas pueden ser instanciadas múltiples veces, con valores distintos de sus parámetros, además de su posición, rotación y escalado en el espacio.

Las instancias transformadas podrán ser combinadas mediante las operaciones booleanas regularizadas unión, intersección y diferencia. Estas operaciones se calculan según la teoría de conjuntos y transforman el resultado en un r-set aplicando la clausura del interior al conjunto obtenido. Siempre debe obtenerse un sólido válido, que puede ser el conjunto vacío. Destacan sobre este tema los trabajos [Requicha80, Mortenson97].

La estructura de datos utilizada para representar un modelo CSG suele ser un árbol binario (árbol CSG). En las hojas del árbol se almacenan las primitivas (cubo, cilindro, esfera, etc.) y en cada nodo interno del árbol cualquiera de las operaciones booleanas permitidas, esto es, unión, intersección o diferencia, que se aplica a los dos sólidos representados por los dos subárboles que cuelgan de dicho nodo interno. La Figura 2-4 muestra un ejemplo de árbol CSG.

Las representaciones CSG son concisas, siempre válidas en el dominio de los modelos r-sets y fácilmente parametrizables y editables. Muchos algoritmos de modelado de sólidos trabajan directamente sobre representaciones CSG basándose en la estrategia *divide y vencerás*, debido a la estructura de los árboles CSG que permite que los resultados calculados sobre las hojas sean combinados hacía la raíz del árbol de acuerdo a las operaciones asociadas a los nodos internos del mismo.

El problema que plantea la representación CSG es que no proporciona explícitamente información sobre la geometría y la conectividad del sólido modelado. Esto es necesario a la hora de visualizar el resultado o de extraer la frontera para una conversión a B-Rep, por ejemplo. En estos casos hay que utilizar métodos adicionales y normalmente costosos. Para la visualización suele emplearse trazado de rayos, y para la extracción de una representación B-Rep se utilizan técnicas de evaluación de fronteras. La mayoría de los modeladores utilizan este esquema de representación de forma conjunta con otro que permita realizar las operaciones deseadas de forma eficiente.

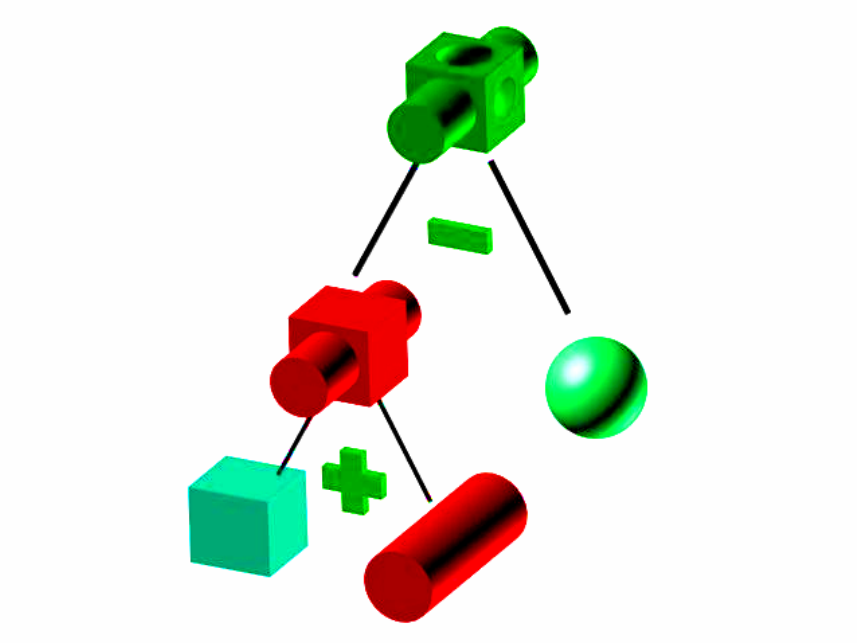


Figura 2-4. Ejemplo de árbol CSG.

Pese a los inconvenientes, la representación CSG es muy fácil de utilizar y pueden construirse modelos muy complejos en términos paramétricos de alto nivel que cualquier usuario puede manejar, lo que explica su amplia presencia en el entorno académico y del diseño industrial.

2.3.4 Descomposición espacial

Una ***estructura de datos espacial*** [Mäntylä88, Kaufman93] es aquella que organiza alguna geometría en un espacio n-dimensional. El concepto es aplicable a cualquier dimensión, pero aquí nos centramos en el caso 3D. Estas estructuras suelen utilizarse, además de como esquema de representación de sólidos, para acelerar consultas sobre elementos en el espacio, lo que incluye tareas como operaciones de ocultación (culling), tests de intersecciones y de inclusión y detección de colisiones. Cualquiera de las estructuras de enumeración espacial sirve para representar un sólido, sin embargo en algunos casos la representación es incompleta.

La organización espacial de los datos suele ser jerárquica, lo que significa en la mayoría de las situaciones que la representación es un árbol. La razón principal para organizar de este modo la información usada es la aceleración de las consultas [Glassner84], que suelen pasar de $O(n)$ a $O(log(n))$. Por supuesto, la construcción de las

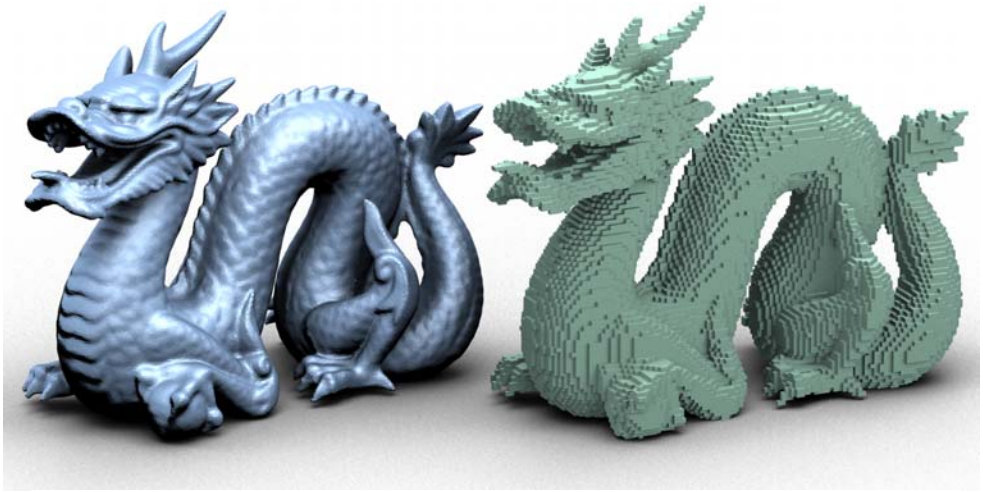


Figura 2-5. Representación B-Rep de un sólido (izquierda) y un espacio de voxel asociado (derecha).

estructuras de descomposición espacial es costosa tanto en espacio (memoria) como en tiempo de procesamiento [Ogayar04, Ogayar05].

2.3.4.1 Enumeración espacial

El esquema de ***enumeración espacial*** consiste en la subdivisión uniforme del espacio, y es la forma de división espacial más sencilla. Un ***voxel*** (de Volume Element) es una especie de *cuboide* [Glassner89], o prisma rectangular alineado con los ejes, y es la unidad fundamental creada por un proceso de particionamiento del espacio. El término en sí connota la extensión del elemento básico de una imagen 2D (pixel) a 3D, lo que constituye un elemento volumétrico. En un espacio de voxels éstos no tienen por qué seguir un patrón determinado en su localización y tamaño, si bien lo normal es que presenten una distribución uniforme e igual tamaño, lo que constituye la subdivisión uniforme del espacio (grid). Este último aspecto es importante en la definición de un sólido, ya que si se busca almacenar elementos volumétricos de distinto tamaño lo más adecuado sería el uso del octree (cualquier espacio uniforme de voxels puede ser representado mediante un octree sin sufrir pérdidas de información).

La división del espacio en voxels puede ser considerada como la operación más básica de subdivisión y discretización espacial. Todo el espacio que ocupa el sólido se divide en voxels, que quedan etiquetados según su inclusión en dicho sólido. El espacio de voxels más simple consiste en una matriz 3D de bits que quedan activados (a 1) si se incluyen en el objeto, y desactivados (a 0) si ocurre lo contrario. Si el voxel está parcialmente ocupado por el sólido es necesario establecer algún criterio de

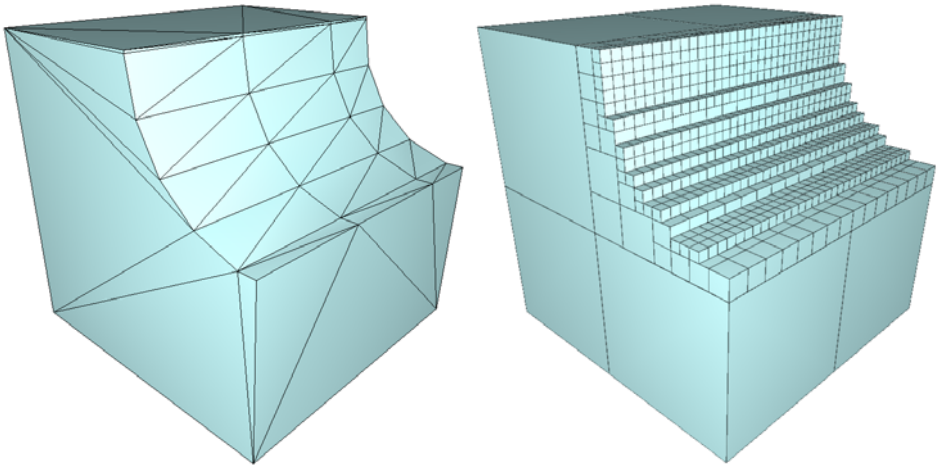


Figura 2-6. Representación B-Rep de un sólido (izquierda) y su correspondiente octree de profundidad 6 (derecha). En el octree sólo se muestran los nodos hoja.

clasificación que permita considerarlo como perteneciente o no al objeto. La Figura 2-5 muestra un ejemplo de voxelización.

Las ventajas de esta representación son la simplicidad y la facilidad para el cálculo de las operaciones booleanas y de ciertas propiedades como el volumen de un sólido. Es un método aproximado, es único y no ambiguo. El problema principal radica en que no es posible la representación exacta de muchos sólidos, debido a que no existe el concepto de ocupación parcial, por tanto la mayoría de los sólidos (especialmente si tienen caras curvas) únicamente pueden ser aproximados. Se puede aumentar la precisión del modelo disminuyendo el tamaño de los voxels, aunque esto supone un uso considerable de memoria. Se han desarrollado varios esquemas de representación basados en este sistema que redefinen de forma adaptativa el tamaño o la forma de los voxels, como son el octree, y el BSP (Binary Space Partition) de planos alineados con los ejes.

2.3.4.2 Octree

Para reducir la ocupación de memoria de una enumeración espacial básica o espacio de voxels, es necesaria otra forma de organización espacial. Una de las más utilizadas es el ***octree***, que es una estructura de datos jerárquica que describe cómo se distribuyen los elementos en el espacio tridimensional [Samet88, Samet88b, Watt00]. De la misma forma que un voxel puede considerarse como la extensión 3D de un pixel, el octree es la extensión tridimensional del quadtree [Samet84]. El octree aprovecha la

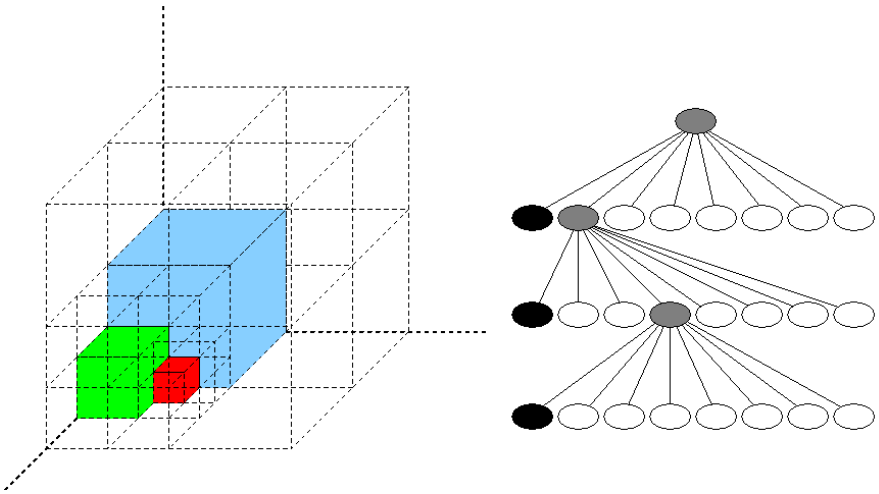


Figura 2-7. Ejemplo de construcción de un octree.

coherencia espacial que surge en la enumeración espacial entre voxels con igual pertenencia al sólido, y que por tanto pueden agruparse.

Un octree es un árbol con un máximo de 8 descendientes por cada nodo, aunque este número suele ser fijo. Esta estructura es ideal para organizar el espacio en cubos de distinto tamaño, lo que permite ahorrar memoria. En la estructura jerárquica del octree hay dos tipos de nodos: los nodos intermedios y los nodos hoja. Los nodos intermedios almacenan información ordenada sobre otros subnodos contenidos dentro del mismo. Cuando un cubo representado por un nodo se divide, lo hace en 8 octantes del mismo tamaño cada uno. En cada nodo hoja se almacena información sobre el sólido. La Figura 2-6 muestra la representación aproximada de un sólido mediante un octree.

En un octree que representa un sólido hay 3 tipos de nodos hoja: blancos si el espacio que representa el nodo está dentro del sólido, negro si está fuera y gris si la clasificación del espacio representado es indeterminada. Puede considerarse todo nodo no hoja como gris, lo cual resulta obvio, ya que el estado de inclusión lo determinarán los descendientes.

El problema de este esquema de representación es que, al igual que ocurre con la enumeración espacial, el resultado es sólo una aproximación del sólido en la mayoría de los casos. Al ser una representación en un espacio discreto aparece el problema del aliasing (con lo que el resultado puede presentar un perfil semejante al de unos dientes de sierra, muy común en imágenes 2D).

La construcción del octree comienza con una caja envolvente (normalmente un cubo) que se ajusta lo más posible al sólido a descomponer. Cada nodo generado contendrá los siguientes elementos:

- **Caja envolvente**. Es el espacio delimitador que engloba el nodo. Aunque normalmente se utilizan cubos para describir el espacio, no es una condición necesaria, y por tanto se pueden utilizar prismas de base rectangular, lo que determinará la forma global del octree (todos los nodos tendrán las mismas proporciones; su volumen dependerá de la profundidad en el árbol). Toda esta información sirve para acelerar los cálculos, ya que puede obtenerse a partir de la posición del nodo en el árbol y de la caja envolvente total del octree.

- **Nodos hijos**. Si un nodo se subdivide siguiendo los criterios establecidos, y por tanto deja de ser un nodo hoja, mantendrá punteros a los subnodos. Se puede tomar la determinación de eliminar (o de no crear) un subnodo que estará vacío (y su puntero desde el nodo padre será nulo).

Cada nodo del octree (comenzando por el nodo raíz) se subdivide en ocho octantes. Para cada nodo hay que calcular la inclusión en el sólido siguiendo algún criterio. La subdivisión del octree finalizará al alcanzar una profundidad determinada. A veces es interesante para procesos de inclusión ofrecer la opción de que los nodos que quedan en la frontera del sólido tengan un estado indeterminado (ni dentro ni fuera), en lugar de utilizar alguna heurística para aproximar su clasificación (lo que siempre produce problemas de aliasing).

La profundidad del árbol dependerá de uno o más criterios establecidos. Lo más habitual es establecer una profundidad máxima en cada una de sus ramificaciones. Lógicamente un nodo que queda por completo fuera del sólido no se subdivide más. La Figura 2-7 muestra un diagrama de la estructura de un octree.

Las operaciones booleanas con esta representación son muy simples, ya que no requieren de cálculos geométricos. Deben compararse los nodos que correspondan al mismo espacio. El problema es que los escalados del modelo quedan restringidos a múltiplos de 2 y las rotaciones a múltiplos de 90°.

2.3.4.3 Octree extendido

Para mejorar al octree básico se han propuesto varios esquemas jerárquicos que buscan un mejor aprovechamiento de la memoria así como conseguir una representación exacta del modelo. Las mejoras se realizan principalmente en los nodos

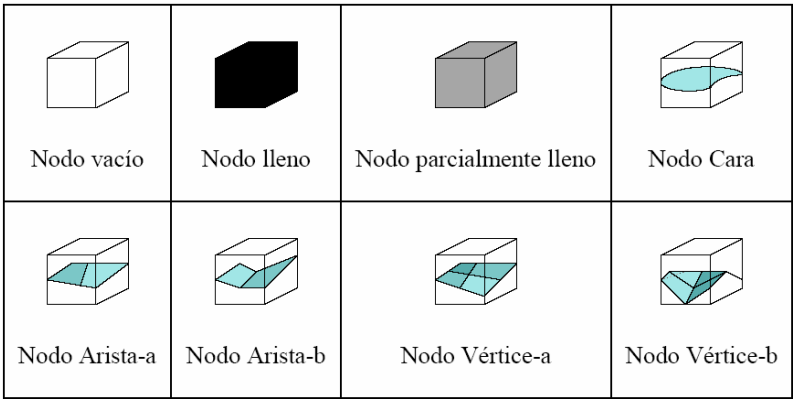


Figura 2-8. Tipos de nodos en el octree extendido.

hoja, donde se almacena información adicional acerca de la frontera del sólido [Ayala85, Brunet85, Navazo87, Brunet90]:

- Nodo *cara*. Es un nodo atravesado por una cara plana, sin aristas ni vértices.
- Nodo *arista*. Almacena parte de una arista que lo atraviesa e información de las caras que comparten dicha arista.
- Nodo *vértice*. Almacena un vértice de la superficie, así como las caras y aristas que lo referencian.

Según del tipo de nodos que se utilicen existen los *Face Octrees* [Brunet90b], Face and Edge Octrees [Navazo89] y los Octrees Extendidos o PM-Octrees [Ayala85, Durst89]. La Figura 2-8 muestra los tipos de nodos del octree extendido.

Todos estos esquemas utilizan menos espacio en memoria, ya que necesitan menos nodos para realizar la representación, sin embargo, las operaciones booleanas son mucha más complicadas al introducir cálculos geométricos.

2.3.4.4 BSP

Un método extremadamente popular para ordenar información espacial es el ***BSP*** (Binary Space Partitioning), por el cual el espacio se fragmenta recursivamente en subespacios convexos mediante hiperplanos. Siendo n la dimensión del espacio, para n=2, la estructura divisora es una línea, y para n=3 es un plano. Un árbol BSP es la

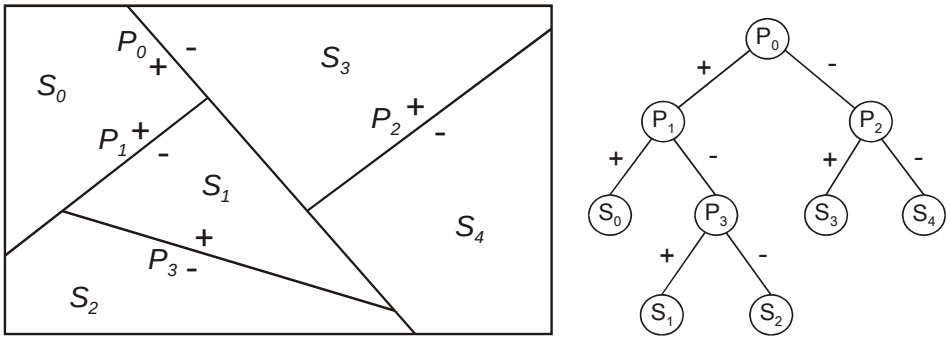


Figura 2-9. Particionamiento binario de un espacio bidimensional.

estructura de datos utilizada para representar la división realizada del espacio. Para n=3, el nodo raíz representa todo el espacio, y contiene el plano que dividirá al mismo en dos subespacios (considerados subconjuntos). De dicho nodo dependen dos ramas del árbol, cada una asociada a cada porción del espacio situada en referencia a la parte positiva o negativa del plano divisor. De esta forma, si el plano es $N \cdot X - d = 0$, la rama izquierda reúne los puntos que cumplen la condición $N \cdot X - d > 0$, y la derecha, los puntos que cumplen $N \cdot X - d < 0$. Los puntos situados sobre el plano $N \cdot X - d = 0$ se considerarán como positivos o negativos según un criterio determinado (por supuesto utilizando el mismo para todo el árbol). La Figura 2-9 muestra un ejemplo de BSP en 2D.

Hay dos tipos de BSP muy utilizados en informática gráfica; los alineados con los ejes (*axis-aligned*) también conocidos como ***bintrees*** y los alineados con polígonos (*polygon-aligned*). En ambos casos, la estructura espacial sirve para navegar de forma ordenada según un criterio establecido, ordenando la geometría de forma aproximada en el alineado con los ejes y de forma exacta en el alineado con polígonos.

El ***BSP alineado con los ejes*** comienza a construirse utilizando la caja englobante de la geometría (lógicamente alineada con los ejes). Cada nodo, comenzando por el inicial, se subdivide en dos partes iguales utilizando un plano alineado con uno de los tres ejes coordenados. Si el plano divide al espacio en dos partes de igual volumen, el árbol es un *bintree*, y si el plano divisor tiene una posición arbitraria es un ***kd-tree***. El resultado es la división de cada caja (asociada a un nodo del árbol) en otras dos. Para representar un objeto en el árbol se marca cada subespacio de cada nodo como blanco, negro o gris, como ocurre con los nodos de los octrees. Al no ajustarse los planos de división a la superficie del sólido, este esquema proporciona una representación aproximada. Cualquier representación obtenida utilizando un bintree puede convertirse a un octree (no siempre con el kd-tree).

El ***BSP alineado con polígonos*** [Gordon91, Abra97] utiliza un polígono arbitrario para la subdivisión del espacio. En cada subdivisión el espacio queda dividido en dos partes (positiva y negativa en referencia al plano divisor). Esta estructura nos permite representar de forma exacta un sólido [Thibault87, Naylor90, Paterson90]. Para crear un árbol BSP eficiente se requiere bastante tiempo, y además presenta el problema del equilibrado, es decir, es conveniente tener un árbol en el que la profundidad de todos los nodos hoja sea más o menos la misma. Un árbol BSP mal equilibrado puede ser ineficiente. Consultar [Fuchs80, James99] para obtener heurísticas de construcción de árboles BSP. El BSP orientado con las caras del sólido es un esquema exacto para objetos poliédricos, no ambiguo y no único.

El árbol BSP es una estructura de clasificación espacial más general que los octrees y espacios de voxels, ya que permite una segmentación espacial sin restricciones en la orientación de los planos; es más, cualquiera de ellos puede ser representado utilizando un BSP. El problema es que para conjuntos complejos de entidades geométricas, la memoria consumida y el tiempo de procesamiento puede llegar a ser excesivo. En cuanto a las operaciones booleanas, éstas se realizan mediante intersección de planos de un árbol con otro [Thibault87, Naylor90b].

2.3.4.5 SP-Octree

Para corregir los inconvenientes del octree y del BSP, así como para aunar las ventajas de ambos, surge la representación ***Space Partition Octree*** (SP-Octree) [Cano04]. Este esquema se basa en la modificación del octree clásico para almacenar información extra en todos los nodos, tanto en los intermedios como en los nodos hoja. En estos últimos se almacena información sobre la frontera del sólido, con lo que la representación del sólido será exacta. En los nodos internos también aparece información sobre la frontera, lo que permite optimizar algunas operaciones, ya que no es necesario recorrer la estructura hasta llegar a un nodo terminal. Además, permite fácilmente la transmisión progresiva de sólidos en red.

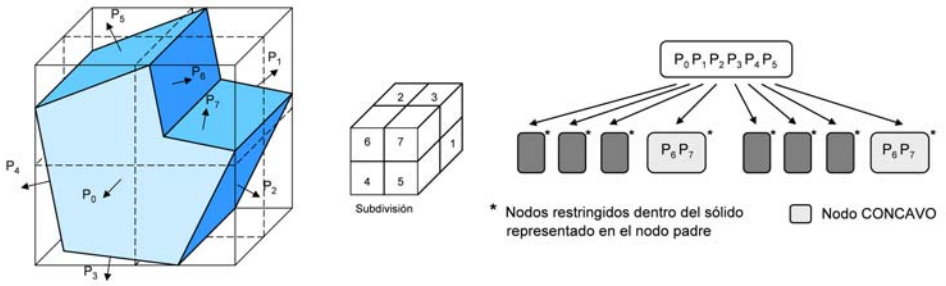


Figura 2-10. Un nodo gris, y el orden establecido para los nodos hijos.

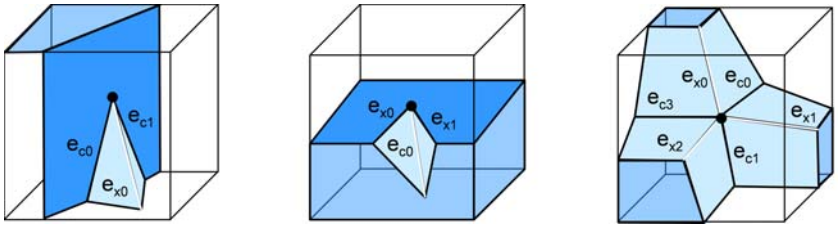


Figura 2-11. Nodos vértice.

En el SP-Octree se utilizan distintos tipos de nodos. Utilizando la misma estructura de árbol octal que en octrees clásicos y en octrees extendidos, se define un grupo de tipos de nodos que permiten incluir parte de la información de la frontera tanto en los nodos terminales como en los intermedios. El objetivo es clasificar cada nodo en función de las características (convexidad y concavidad) de los planos que aparecen en él. Además, en los nodos internos se almacena la información de los planos cuya configuración se mantiene igual en los nodos descendientes, por lo que no es necesario almacenarlos en ellos. Esta información permite acotar la zona del espacio en que está definido el sólido dentro del nodo (Figura 2-10).

- Los ***nodos blancos y negros*** sirven para representar volúmenes completos que están dentro o fuera del sólido respectivamente, al igual que ocurre con los octrees convencionales (Figura 2-12).

- Cuando la intersección del sólido con un nodo del árbol es convexa, se utiliza un ***nodo convexo***. En este caso, se incluye en el nodo el conjunto de planos que definen los semiespacios cuya intersección forma el objeto convexo (Figura 2-12).

- Si la intersección del nodo y el sólido representado es cóncava, se utiliza un ***nodo cóncavo***. El nodo cóncavo se define como la diferencia del volumen envolvente del nodo con la intersección de los semiespacios incluidos en el mismo, invirtiendo la orientación de los planos que los definen (Figura 2-12).

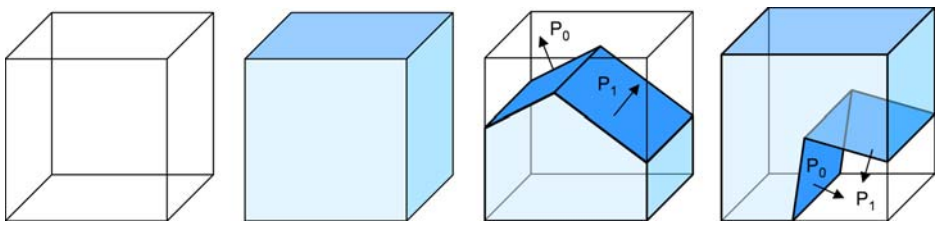


Figura 2-12. Nodos blanco, negro, cóncavo y convexo de un SP-Octree.

- Si existen tanto concavidades como convexidades, el nodo se clasifica como ***gris***, y debe ser subdividido; aun así, mantiene la información de los planos (Figura 2-10).

- Para completar la representación se definen los ***nodos vértice*** que sirven para representar partes del sólido donde no es posible utilizar ninguno de los nodos anteriores. Un nodo vértice sólo contiene un vértice del sólido en el que convergen tanto aristas cóncavas como convexas. Este nodo almacena, además del vértice, la información de los planos de las caras que lo referencian, así como la configuración de las aristas que convergen en dicho vértice (Figura 2-11).

Esta estructura aporta los beneficios del octree y del BSP, haciendo la representación exacta. Sin embargo, al igual que ocurre con los PM-Octrees, las operaciones booleanas son mucha más complicadas debido a la variedad de tipos de nodos.

2.3.4.6 Algoritmos y optimizadores basados en descomposición espacial

En Informática Gráfica existe una gran cantidad de algoritmos geométricos que utilizan estructuras de enumeración espacial [Samet89]. En este trabajo se han implementado muchas de ellas, ya que además de servir como esquemas de representación de objetos, son muy útiles como optimizadores espaciales para la aceleración de algunos algoritmos. A continuación se presentan algunas de las estructuras utilizadas para las labores de optimización de algoritmos.

Enumeración espacial. Mediante un espacio de voxels pueden almacenarse elementos, como polígonos, segmentos, etc., que intersectan cada elemento volumétrico, de forma que las consultas sobre entidades próximas a un espacio se aceleran drásticamente, siendo de orden constante. La división del espacio en voxels (*grid*) puede ser considerada como la operación más básica de organización espacial. Esta estructura de datos suele consumir mucha memoria.

Jerarquía de volúmenes envolventes (BVH). La jerarquía de volúmenes es una estructura jerárquica donde los nodos hoja son los volúmenes envolventes de los elementos clasificados (por ejemplo, triángulos). Los nodos intermedios constituyen la mínima caja envolvente que abarca los volúmenes envolventes de los nodos descendientes. Con esta estructura las consultas se aceleran y además es extremadamente sencillo de construir, lo que es beneficioso en caso de clasificar entidades geométricas complejas.

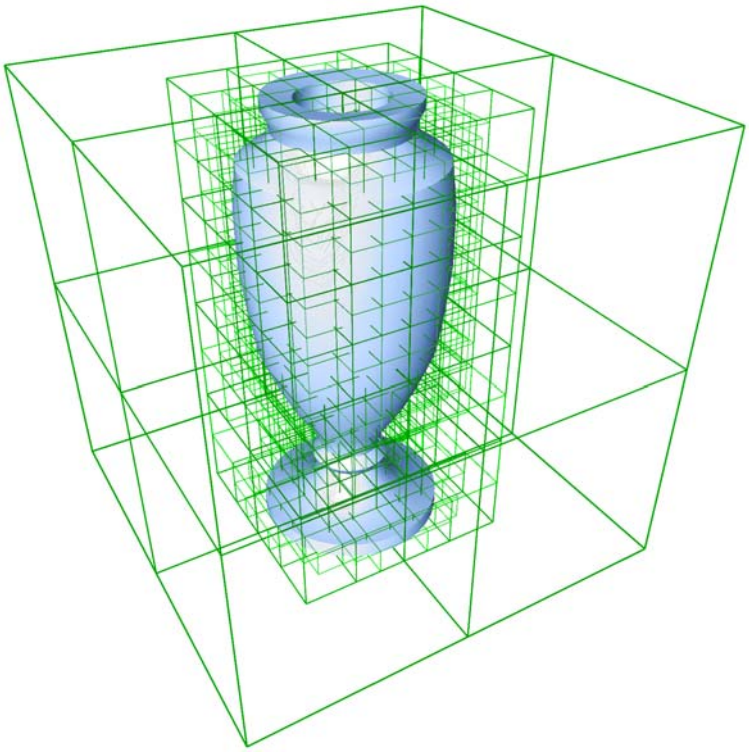


Figura 2-13. Cálculo de un octree de profundidad 5 sobre un modelo poligonal. Los nodos hoja almacenan los identificadores de los polígonos incluidos.

Octree. De la misma forma que se utiliza el octree para representar sólidos, también se puede emplear como optimizador espacial. Permite descartar volúmenes completos de espacio en operaciones como el cálculo de optimización del rendering* de escenas 3D, la determinación de visibilidad de subespacios, la selección de objetos, la detección de colisiones entre todo tipo de estructuras, además de la inclusión de puntos en sólidos. La estructura del octree clasificador es la misma que la del octree que representa a un sólido, pero en esta ocasión, la información de los nodos hoja es distinta. En este caso, la estructura contiene referencias a entidades geométricas, y cada nodo puede estar asociado con varias de ellas. En cada nodo puede almacenarse la siguiente información, además de las referencias a los nodos descendientes:

---

* *El término **render** se utiliza en lugar de dibujar, interpretar o representar por su gran utilización en el ámbito de la Informática Gráfica y porque en este contexto suele aplicarse de forma concreta a la representación de información 3D. De igual forma, se utilizan a lo largo del texto términos como rendering, renderizar, rasterizar, texel, voxel, etc., que aunque no existen en castellano, no tienen una traducción fácil y por tanto se utilizan como una simplificación. En cualquier caso, son vocablos habituales en el argot informático.*

- *Lista de elementos*: Cada nodo almacena los elementos que intersectan o están totalmente incluidos en el volumen descrito por el nodo. Para ahorrar memoria esta lista debe existir sólo en los nodos hoja.

- *Vecinos*. Cada nodo puede almacenar información de los octantes vecinos, que dependiendo de la conectividad considerada en dicho nodo serán 6 vecinos (caras), 18 vecinos (caras y aristas) o 24 vecinos (caras, aristas y vértices). Esta opción es muy interesante para cálculos de colisión. Para los algoritmos implementados en este estudio no se ha utilizado.

Si el octree se utiliza para clasificar los polígonos de un sólido, cada nodo almacenará una lista de caras que lo intersectan o quedan incluidas. La lista de caras debe ser realmente una lista de índices para ahorrar memoria. Si un nodo queda vacío puede eliminarse y dejar en el padre una referencia nula. Si el nodo tiene un número de caras asociadas superior al mínimo establecido se subdividirá de nuevo en octantes, y se repetirá el proceso hasta que se alcance una profundidad determinada en la estructura o bien se alcance un número máximo de elementos por nodo.

El problema geométrico principal en el proceso de clasificación de polígonos en el octree es determinar cuándo un polígono intersecta un nodo de la estructura. En [Möller02] se presenta un método muy eficiente para determinar la intersección de un triángulo con un voxel, lo que es de mucha utilidad cuando se utilizan mallas de triángulos.

BSP. La subdivisión del espacio por planos tiene muchas utilidades para la optimización de algoritmos [Eberly01], entre las que destacan la eliminación de superficies ocultas, la optimización del rendering de escenas basadas en polígonos, la determinación de visibilidad de subespacios, la selección de objetos, la detección de colisiones entre todo tipo de estructuras, y la inclusión de puntos en sólidos.

2.3.5 Representación mediante fronteras (B-Rep)

El modelo de fronteras es uno de los más populares debido a su campo de aplicación, y por tanto uno de los más utilizados por la mayoría de los sistemas de modelado. Este esquema de representación se basa en el hecho de que todo sólido queda delimitado por una frontera que lo separa del exterior. Esta frontera puede modelarse utilizando vértices, aristas y caras [Baer79]. La Figura 2-14 muestra un ejemplo.

Con este esquema se representa el sólido mediante una frontera definida por un grafo donde se almacena la geometría de la superficie y la relación topológica que

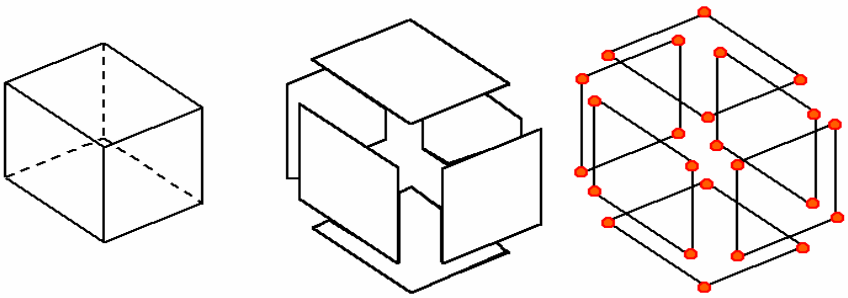


Figura 2-14. Estructura básica de un B-Rep.

existe entre todos los elementos de dicha geometría. La entidad geométrica fundamental más común en este esquema de representación es el polígono, aunque también pude emplearse cualquier tipo de superficie.

El dominio de objetos representables depende de los elementos empleados para la evaluación de la frontera. El caso más habitual, y el que se trata en este trabajo, es el que emplea caras planas, lo que permite definir sólidos poliédricos. Es un esquema no ambiguo y no único.

En el caso de utilizar caras planas para la frontera, las condiciones de validez son las siguientes:

- El poliedro debe cumplir la fórmula de Euler [Hoffmann89]:

  Vértices – Aristas + Caras = 2

  Y si tiene agujeros, la fórmula de Euler-Poincaré [Hoffmann89]:

  Vértices – Aristas + Caras – 2(1-Agujeros) = 0*

  Se puede tener más de un cuerpo y un agujero en un objeto.

  El cumplimiento de esta fórmula es condición necesaria pero no suficiente para garantizar la validez del sólido.

- Restricciones geométricas:

  a) Cada arista está delimitada por dos vértices.

b) Cada arista es compartida por dos caras.

c) Cada vértice está presente en al menos 3 caras.

d) Las caras no se intersectan salvo en vértices y aristas comunes.

2.3.5.1 Representación

Existen muchas estructuras de datos para la representación de sólidos poliédricos, como Windged-Edge [Baumgart75, Mäntylä88], Star-Edge [Karasick88] o F-Edge [Vaneck89]. La mayoría utiliza como elementos vértices, aristas y caras organizados en tablas con referencias de caras a aristas y de aristas a vértices, aunque con ciertas variaciones según el tipo de estructura. A continuación se presentan algunos detalles sobre estructuras utilizadas.

1. **Simples** [Mäntylä88].

   a) *Basada en polígonos*. El sólido se representa como un conjunto de caras, cada una formada por una lista de vértices.

   b) *Basada en vértices*. Igual que la anterior, el sólido queda representado por una lista de caras, pero en esta ocasión, los vértices compartidos no se duplican. Se almacena una lista de vértices y las caras contienen referencias a los vértices que utilizan. Es mucho más eficiente que la estructura anterior. El problema de esta estructura y de la anterior es que es complicado comprobar algunas relaciones topológicas.

   c) *Basada en aristas*. Se basa en representar cada cara como una lista de aristas que a su vez están formadas por vértices. Lo más habitual es mantener listas de vértices, aristas y caras con referencias de unos elementos a otros según proceda.

2. **Windged-Edges** [Mäntylä88].

   Para reducir el coste computacional de algunas operaciones sobre las estructuras anteriores se introdujeron otras más complejas, entre las que destaca la estructura de aristas aladas [Baumgart75]. Se basa en las características de un sólido variedad de dimensión 2, esto es, cada arista es compartida por 2 caras, y en cada vértice convergen un mínimo de 3 aristas.

La estructura se basa en las aristas y su conectividad con otros elementos. Cada arista tiene dos punteros a vértices, dos punteros a las caras relacionadas y a las 4 aristas que parten de ella.

Con esta estructura se pueden recorrer todos los elementos de forma iterativa realizando operaciones como la comprobación de la corrección topológica. Además, las consultas a los elementos permanecen en orden constante.

2.3.5.2 Características

Este esquema de representación está muy extendido ya que es fácil de construir, modificar y visualizar. Entre las ventajas e inconvenientes encontramos:

- ✓ Es muy útil para visualización. El hardware gráfico actual está orientado a este tipo de representación.
- ✓ Es fácil de manipular y las posibilidades de edición son muy amplias. Se pueden aplicar multitud de operaciones y transformaciones.
- × La creación manual es tediosa.
- × Puede llegar a utilizar mucha memoria.
- × Las operaciones booleanas son muy costosas, ya que en el peor de los casos hay que operar entre sí todos los elementos de los sólidos que intervienen en la operación.
- × Es costoso comprobar su validez y mantener la integridad del resultado de algunas operaciones.

Pese a que en principio parece que hay más inconvenientes que ventajas, muchos sistemas de modelado se basan en esta estructura debido a la importancia de su campo de aplicación, como la realidad virtual. Según parece, en la industria y en el software de consumo son mucho más importantes las posibilidades referentes a edición y visualización que las relacionadas con la corrección topológica o el ahorro de memoria. Además, cada vez aparecen más métodos y algoritmos que reducen los efectos de los inconvenientes señalados.

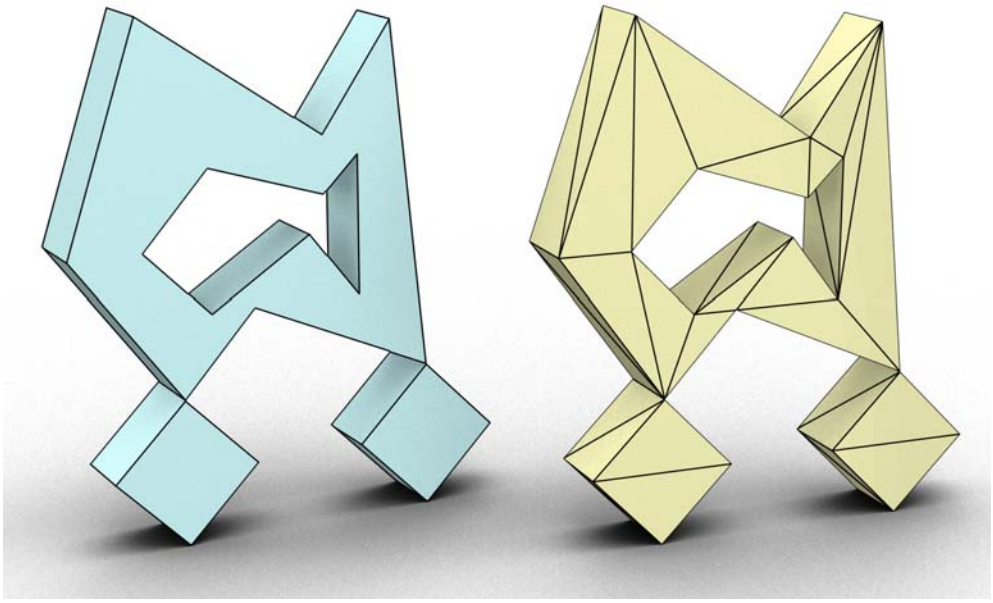


Figura 2-15. Representación de un sólido mediante B-Rep de caras planas de n lados (izquierda) y su correspondiente malla de triángulos (derecha).

2.3.5.3 Mallas de triángulos

Las mallas de triángulos son un tipo de B-Rep muy utilizado por su simplicidad. Algunos de los algoritmos presentados en esta tesis están optimizados para ser utilizados con esta estructura. La representación de la frontera del sólido se basa en triángulos, por lo que su dominio es el de los objetos poliédricos. Básicamente un B-Rep basado en caras planas puede contener polígonos de 3 o más lados; en el caso de mallas de triángulos los polígonos son sólo triángulos. Ya que cualquier polígono de más de tres lados puede descomponerse en triángulos mediante un proceso de *teselación*, cualquier sólido poliédrico general puede representarse mediante una malla de triángulos. La Figura 2-15 muestra un ejemplo.

Las estructuras de datos que se emplean para representar mallas de triángulos suelen ser mucho más sencillas que las asociadas a un B-Rep de caras de 3 o más lados. Aunque pueden utilizarse esquemas elaborados como el de aristas aladas, en la mayoría de las situaciones las estructuras utilizadas son muy sencillas. Las aplicaciones que no necesitan realizar comprobaciones topológicas complejas o con mucha frecuencia, suelen basar la representación de las mallas en un esquema simple orientado a vértices. Con este enfoque, tanto la lista de vértices como la lista de

triángulos se basan en vectores dinámicos de datos, lo que simplifica el acceso a los datos.

Las ventajas de esta representación son múltiples, ya que la gran mayoría de algoritmos de geometría computacional aplicables al modelado de sólidos se benefician de esta representación más simple. Además, todo lo referente a la visualización de los objetos queda enormemente simplificado cuando se tratan sólo triángulos. Hay que recordar que el hardware gráfico actual sólo admite triángulos o primitivas que puedan descomponerse en triángulos. Esta es la estructura de datos más utilizada en la visualización 3D en tiempo real.

2.3.6 Modelado basado en puntos

El modelado basado en puntos es una alternativa a los sistemas clásicos de modelado para la representación de superficies, tanto para el tratamiento como para la visualización de modelos complejos [Kobbelt04]. Con los modelos basados en puntos no es necesario mantener información topológica global, lo que facilita el tratamiento de sólidos cuya forma cambia dinámicamente.

Una representación basada en puntos puede ser considerada como un muestreo de una superficie continua que genera una nube de puntos p_i, opcionalmente con vectores normales, colores y otras propiedades asociadas. Una de las ventajas de esta representación es que permite obtener propiedades asociadas a un punto cualquiera a partir de los puntos circundantes pertenecientes al modelo.

Los modelos basados en puntos suelen elaborarse con scaners 3D o utilizando métodos de reconstrucción basados en imágenes. En ambos casos, los puntos muestreados contienen cierta cantidad de ruido debido a las imprecisiones de los métodos de adquisición, que puede reducirse mediante técnicas basadas en filtros.

Varios conceptos y algoritmos aplicados comúnmente a mallas de triángulos han sido adaptados a las representaciones basadas en puntos. Se ha llevado a cabo simplemente reemplazando las relaciones de vecindad entre triángulos por la proximidad entre puntos. Sin embargo, también han surgido técnicas nuevas que se aplican específicamente a nubes de puntos. Ya que no es necesario mantener información de conectividad o mantener ninguna consistencia topológica, el muestreo o la reestructuración son muchos más fáciles que con mallas de triángulos.

2.4 Algunos problemas básicos en modelado de sólidos

La mayor parte de los problemas que se presentan en el modelado de sólidos son de tipo geométrico, aunque también hay ciertas dificultades relacionadas con los esquemas de representación, y por tanto con las estructuras de datos utilizadas. De la resolución de los problemas geométricos se encarga una disciplina dentro de la Informática Gráfica, que es la **Geometría Computacional**. Problemas típicos dentro de este campo y relacionados con el modelado de sólidos son la intersección de segmentos, la determinación de envolventes convexas, etc. La Geometría Computacional es una de las herramientas de base más importantes en el modelado de sólidos [Berg97].

2.4.1 Representación de entidades geométricas

Las entidades geométricas que pueden estar presentes en un modelador de sólidos, y por tanto en la representación del modelo, son:

1. Puntos.
2. Aristas.
3. Caras.
4. Superficies.
5. Volúmenes.

Los puntos en R^3 pueden representarse fácilmente mediante sus coordenadas o como intersección de entidades de orden superior. Las aristas se suelen representar mediante sus puntos inicial y final.

La superficie del objeto puede representarse mediante caras poligonales, formadas por aristas conectadas, o por superficies expresadas de forma paramétrica, como B-Splines, Nurbs, etc. Las superficies llevan asociadas una nube de puntos de control, además de una serie de coeficientes.

En el caso de que la superficie esté representada por caras poligonales, se emplean listas de aristas conectadas y cerradas y una sucesión de vértices de la cual se pueden obtener directamente la lista de lados. Existen representaciones que utilizan una notación paramétrica.

La forma de representación del volumen suele ser la utilización de una ecuación implícita, como $x^2+y^2+z^2=1$, que define una esfera de radio unidad. También, como ocurre con las superficies, pueden representarse volúmenes de forma explícita, lo que limita la cantidad de volúmenes que pueden representarse. El siguiente ejemplo muestra una representación explícita de un cubo centrado en el origen y de dimensiones 2^3:

$x \geq 1 \wedge x \leq -1$
$y \geq 1 \wedge y \leq -1$
$z \geq 1 \wedge z \leq -1$

2.4.2 Operaciones sobre entidades geométricas

Existen muchas operaciones que pueden realizarse sobre las primitivas geométricas que aparecen en un modelador de sólidos. A continuación se muestran algunas de las más importantes.

2.4.2.1 Clasificación de puntos

La clasificación de un punto del espacio respecto de cualquier primitiva geométrica es la base de multitud de algoritmos geométricos. El problema consiste en dado un punto, determinar su posición respecto de la primitiva. Esto incluye operaciones tan importantes como punto en segmento, punto en plano, punto en polígono, y finalmente punto en sólido.

Este algoritmo es de vital importancia en el modelado de sólidos, y es uno de los desarrollados y optimizados en esta tesis [Ogayar04, Ogayar05, Ogayar05b].

2.4.2.2 Trazado de curvas

Este problema consiste en conectar un conjunto de puntos en el plano para el caso 2D y en el espacio para el caso 3D. La curva suele obtenerse mediante la aproximación a segmentos de recta que unen los puntos. Este algoritmo se emplea para algoritmos de intersección de curvas.

2.4.2.3 Muestreo de superficies

Ya que las superficies suelen estar expresadas en forma paramétrica, su visualización rápida o la conversión a otras representaciones no son triviales. Las técnicas básicas para resolver este problema se basan en el trazado de rayos y el cálculo de una malla de polígonos mediante teselación de la superficie [Ruiz97].

El trazado de rayos se utiliza ampliamente en la visualización de superficies y volúmenes. Consiste en lanzar un rayo desde la posición del observador en el espacio que pasa a través de una matriz de pixels e intersecta a los objetos existentes. Esta matriz almacenará los resultados de las intersecciones según una serie de parámetros y formará la imagen final. Esta técnica es fácil de implementar y permite conocer con exactitud la forma de la superficie, sin embargo los cálculos son muy complejos y los resultados obtenidos se definen exclusivamente en el espacio de la imagen.

En general, se suelen utilizar técnicas de teselación que convierten las superficies y volúmenes expresados en forma paramétrica ó implícita a primitivas geométricas más simples de procesar, como triángulos.

2.4.2.4 Intersección de superficies

Un problema muy habitual y complejo de resolver dentro del modelado de sólidos es la intersección entre entidades geométricas. Esta operación debe llevarse a cabo con validez dentro del esquema de representación elegido, lo que influye en la complejidad del proceso.

La intersección entre dos primitivas da como resultado nuevas primitivas geométricas que complementan o sustituyen a las que intervienen en la operación. La dificultad de la misma depende de la complejidad de las primitivas geométricas, siendo la intersección entre dos sólidos el caso más complicado. La operación de intersección sirve como base para la resolución de otros problemas más complejos. En el caso de sólidos, las operaciones booleanas son el caso más claro [Requicha85, Requicha88, Rivero00].

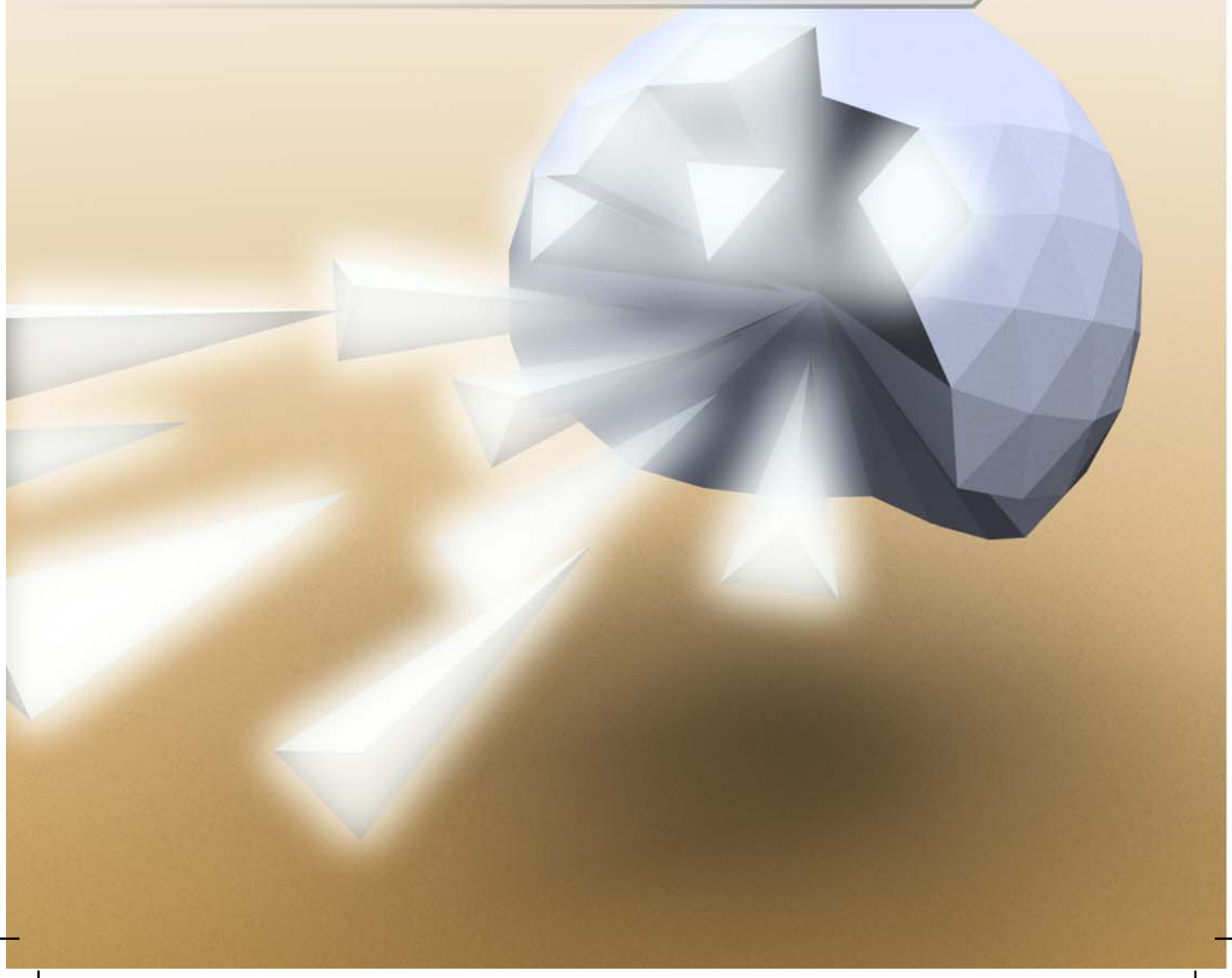
2.5 Conclusiones

En este Capítulo se han presentado las bases del modelado de sólidos y parte de su estado actual. Aunque sólo se trata de una breve exposición de una disciplina muy amplia, sirve como introducción antes de tratar los temas de este trabajo.

De la misma forma, los esquemas de representación de sólidos presentados sitúan a la presente tesis en contexto, ya que muchas de las optimizaciones de algoritmos geométricos tratados se basan en la utilización de diversos esquemas de representación, así como conversiones entre los mismos.

Capítulo 3

Representación de Sólidos mediante Recubrimientos Simpliciales



3

Los recubrimientos simpliciales son utilizados en todos los algoritmos cuyas optimizaciones se proponen en esta tesis. En este Apartado se presentan las bases que formalizan esta técnica. Más adelante se verá con más claridad su utilización práctica. Todo lo presentado aquí está basado en los trabajos de Feito y Torres [Feito95b, Feito97] y Segura [Segura01].

3.1 Introducción

Un ***símplice*** d-dimensional es la envolvente convexa de *d+1* puntos linealmente independientes, por lo que un símplice tridimensional es un tetraedro, uno bidimensional es un triángulo y uno unidimensional es una línea. Mediante un conjunto no disjunto de símplices tridimensionales puede representarse un sólido, y esta es la base de los algoritmos propuestos en este trabajo. A continuación se muestran los formalismos necesarios para la utilización de los recubrimientos simpliciales [Segura01].

3.2 Definiciones topológicas

A continuación vamos a presentar algunas definiciones topológicas referidas a los conjuntos de puntos. La importancia de estas definiciones radica en que muchos modeladores de sólidos restringen el tipo de sólidos a una clase concreta, dejando al margen algunos de estos conceptos.

Definición 3.1. Un *espacio métrico* [Munkres01, Requicha88] es un par $\{M, d\}$, donde M es un conjunto y d es una aplicación $d : M \times M \rightarrow R$ llamada *distancia* o *métrica*, de manera que $\forall x, y \in M$:

$d(x, y) \geq 0$;

$d(x, y) = 0 \Leftrightarrow x = y$;

$d(x, y) = d(y, x)$;

$d(x, z) \leq d(x, y) + d(y, z)$;

Un ejemplo de espacio métrico es el espacio euclídeo con la distancia euclídea. En lo que sigue, y cuando la definición de la distancia sea clara en el contexto, obviaremos citar la misma.

Definición 3.2. Sea M un espacio métrico, se dice que un conjunto $S \subseteq M$ es *abierto* si para todo punto s perteneciente a S, existe un entorno totalmente contenido en S.

Definición 3.3. El *interior* de un conjunto $S \subseteq M$ (que notaremos como *int(S)*) es el subconjunto abierto más grande contenido en S.

Definición 3.4. La *clausura* de S (*clau(S)*) es la unión de S y todos los puntos P tales que cualquier bola abierta de radio mayor que 0 centrada en P intersecta a S y su complementario. Por ejemplo, la clausura de un polígono abierto es el polígono mas las aristas y los vértices.

Definición 3.5. La *frontera* de S (*front(S)*) se define como la diferencia conjuntista entre la clausura de S y el interior de S.

fron(S) = clau(S) - int(S)

Definición 3.6. Un conjunto S se dice *dimensionalmente homogéneo* si todos los puntos de S pertenecen o bien al interior de S, o bien a la frontera de S. Intuitivamente,

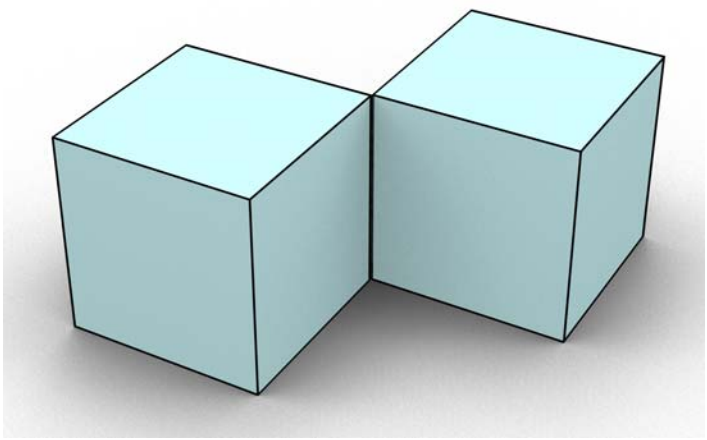


Figura 3-1. Ejemplo de sólido no-variedad.

S es dimensionalmente homogéneo si no tiene puntos, aristas, caras, etc. (depende de la dimensión), aisladas.

Definición 3.7. Una operación se dice *regularizada* si toma como operando un conjunto de puntos y devuelve un conjunto de puntos cerrado y dimensionalmente homogéneo.

Definición 3.8. Un conjunto de puntos *S* es *conexo* si para todo par de puntos de *S* existe un segmento de curva que los conecta por arcos y que está totalmente incluido en *S*. Un conjunto *S* es conexo interiormente si su interior es conexo.

Definición 3.9. Dos conjuntos son *cuasi disjuntos* si su intersección regularizada es vacía. Por ejemplo, dos cubos, uno sobre el otro compartiendo una cara, son cuasi disjuntos.

Definición 3.10. Un *sólido* es un conjunto cerrado, conexo y dimensionalmente homogéneo, definido en E^3. Además, para que un conjunto sea considerado como sólido, el conjunto debe cumplir además las condiciones de *rigidez*, *regularidad* y *representación finita*. Ver [Mäntylä88] para más detalles.

Definición 3.11. Un sólido *X* se dice que es *variedad* (manifold) si para cada punto de la superficie del sólido existe un entorno de dicho punto de radio $\delta>0$ tal que dicho entorno es homeomorfo a un disco en R^2 [Mäntylä88]. La Figura 3-1 muestra un ejemplo de sólido no-variedad.

Es fácil darse cuenta de que la operación regularizada entre dos sólidos variedad puede dar lugar fácilmente a un sólido no-variedad; así, la unión regularizada de dos cubos unidos en un vértice da lugar a un sólido no-variedad, ya que en ese vértice no es posible la construcción de ninguna esfera totalmente contenida en el sólido resultante.

3.3 Álgebra de objetos gráficos

En esta Sección se realiza un breve resumen de las conclusiones más destacadas de los trabajos de Torres [Torres92, Torres93] y Feito [Feito95b], que sirven de base teórica a la presente tesis. El trabajo de Torres supone una formalización para los objetos gráficos. A partir de dicha formalización, Feito propone un modelo teórico para el modelado de sólidos basado en el recubrimiento de los mismos mediante símplices.

Si bien existen en la literatura múltiples trabajos cuyo objeto es el modelado de sólidos, los trabajos encaminados hacia la formalización en Informática Gráfica son más escasos, teniendo especial importancia los trabajos de Duce [Duce88] y Fiume [Fiume89], encaminados hacia la formalización de los sistemas raster, y la generalización de dicho trabajo propuesta por Torres [Torres92], dirigiéndose hacia aspectos de formalización de la visualización y modelado respectivamente.

Torres propone en su trabajo una teoría matemática que puede ser utilizada como metalenguaje para el desarrollo de representaciones abstractas de sistemas gráficos, permitiendo expresar aspectos de la visualización y del modelado. Esta teoría está basada en el concepto de objeto gráfico, definido mediante dos funciones: una función de aspecto y una función de presencia. A partir de esta definición, Torres define un conjunto de operaciones (suma, producto por escalar, unión, intersección, diferencia, producto de objetos y producto circular de objetos) estableciendo una serie de propiedades sobre dicho conjunto y las operaciones definidas.

Definición 3.12. Una transformación geométrica $\tau : R^n \rightarrow R^n$, es una aplicación que cumple la siguiente propiedad:

$$\forall X, Y \in \mathrm{R}, \forall \beta \in [0, 1] \subset \mathrm{R} \Rightarrow \tau(\beta \cdot X + (1 - \beta) \cdot Y) = \beta \cdot \tau(X) + (1 - \beta) \cdot \tau(Y)$$

Definición 3.13. Una ζ-estructura sobre un cuerpo K es un conjunto W en el cual se definen las siguientes operaciones:

- Una operación interna llamada suma, y que notaremos por +, con la cual W es un grupo conmutativo.

- Una operación externa sobre el cuerpo K, y que notaremos por *, llamada producto por escalar, de manera que (W,+,*) es un espacio vectorial.

- Tres operaciones internas llamadas unión, intersección y complementación (unaria), y que notaremos por $\cup$, $\cap$ y ˜ respectivamente, de manera que W con esas operaciones es un álgebra de Boole.

- Una operación interna que llamaremos producto, y que notaremos por ×, que satisface las propiedades asociativa, conmutativa y existencia de elemento neutro.

La definición de objeto gráfico contempla dos factores: por una parte, la *presencia*, y por otra el *aspecto*. Este se refiere a propiedades del objeto gráfico desde el punto de vista visual (color, transparencia, etc.), mientras que aquella describe la ocupación de espacio por parte del objeto gráfico. Estos dos factores se definen como dos ζ-estructuras llamadas *espacio de aspecto* (o dominio de aspecto) δ y *espacio de presencia* (o dominio de presencia) π.

Definición 3.14. Un volumen V se define como un subconjunto de R^n. El volumen de un objeto gráfico es la porción de espacio ocupada por el objeto. Definimos el universo de objetos gráficos, U como una tripleta $U = (\pi, \delta, n)$, siendo n la dimensión del espacio euclídeo.

Definición 3.15. Definimos objeto gráfico en el universo U como un par (μ, α), donde μ es la función de presencia definida como $\mu : R^n \rightarrow \pi$, y α es la función de aspecto definida como $\alpha : R^n \rightarrow \delta$.

Ejemplo. Un triángulo puede definirse como:

Triángulo $(P_1,P_2,P_3)= (\mu,\alpha)$, con

$$\mu(P) = \begin{cases} 1 & si\ \exists a,b,c \in (0,1), a+b+c=1 / P = a \cdot P_1 + b \cdot P_2 + c \cdot P_3 \\ 0 & en\ otro\ caso \end{cases}$$

$$\alpha(P) = k \cdot \mu(P)$$

siendo k el valor de aspecto del triángulo.

Definición 3.16. El volumen *Vol* asociado a un objeto gráfico $O(\mu,\alpha)$ es el conjunto de puntos que cumplen que:

$Vol(O) = \{P \in R^n / \alpha(P) \neq 0 \vee \mu(P) \neq 0\}$

Torres demostró que el conjunto de objetos gráficos es una ζ-estructura, por lo que se definían todas las operaciones de las ζ-estructuras. A partir de las definiciones de objeto gráfico que acabamos de señalar, Torres desarrolló un Álgebra de Objetos Gráficos, definiendo para ello operaciones que permitían mantener la integridad de los objetos. Con dichos resultados Feito desarrolló un sistema generador de objetos gráficos y demostró su validez. Para ello, utilizó símplices originales. En lo que sigue, supondremos que todas las operaciones entre sólidos (o símplices) son operaciones regularizadas.

3.4 Sistema generador de objetos gráficos

A partir de los trabajos de Torres, Feito establece un sistema de modelado basado en el recubrimiento de los sólidos mediante símplices [Feito95b]. Para ello, en primer lugar, establece un sistema de generadores basado en objetos gráficos muy sencillos [Feito97b]. Tras ello, define un conjunto de operaciones regularizadas sobre dicho sistema de modelado, y propone finalmente una representación basada en el recubrimiento de los sólidos mediante símplices, creando una estructura en árbol similar a la de la representación basada en CSG.

Definición 3.17. Se define *clausura convexa* de $d + 1$ puntos $P_0, P_1, P_2, ..., P_d \in R^n$, y lo notaremos como S, como el conjunto de las combinaciones lineales de todos los puntos P_i, esto es:

$$S = \left\{ P / P = \lambda_0 \cdot P_0 + \lambda_1 \cdot P_1 + \lambda_2 \cdot P_2 + \ldots + \lambda_d \cdot P_d , \sum_{i=0}^{d} \lambda_i = 1 \right\}$$

Definición 3.18. Un *d-simplex* es la clausura convexa de $d + 1$ puntos linealmente independientes, es decir:

$\lambda_i \neq \lambda_j$, $i \neq j$, i, j = 0, ...d

Un *d-simplex original* es un d-simplex en el que uno de los puntos es el origen. Un *d-simplex orientado* es un d-simplex en el que los puntos están ordenados según un orden específico.

Definición 3.19. Sea x un número real. Se define la función *signo*, que notaremos como $sign(x)$, como:

$$sign(x) = \begin{cases} 1 & si\ x > 0 \\ 0 & si\ x = 0 \\ -1 & si\ x < 0 \end{cases}$$

Definición 3.20. Sean $P_1(x_1,y_1)$, $P_2(x_2,y_2)$, $P_3(x_3,y_3)$ tres puntos en R_2, se define el *área signada* del triángulo $T=P_1P_2P_3$ como:

$$Area(T) = \frac{1}{2} \cdot \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

Si los tres puntos se encuentran alineados sobre la misma recta, entonces el área del triángulo es 0. Se define el área signada de un triángulo T como el área del triángulo considerando su signo. El área signada (o simplemente signo de un triángulo T, $sign(T)$), se define entonces como:

$$sign(T) = sign(Area(T)) = [T]$$

Se puede demostrar fácilmente que si el área signada es positiva, entonces el triángulo tiene orientación positiva, es decir, los vértices se recorren en el sentido antihorario. Como puede observarse fácilmente, si uno de los puntos del triángulo coincide con el origen de coordenadas, el cálculo del determinante necesario para el cálculo del área signada se simplifica bastante, ya que en lugar de calcular un determinante 3 × 3, se obtiene un determinante 2 × 2.

Definición 3.21. Sean A, B, C y D cuatro puntos en R_3. El volumen signado [ORourque94] del tetraedro con vértices D, A, B, C, designado como [DABC] se define como:

$$[DABC] = \frac{1}{6} \cdot \begin{vmatrix} x_a - x_d & y_a - y_d & z_a - z_d \\ x_b - x_d & y_b - y_d & z_b - z_d \\ x_c - x_d & y_c - y_d & z_c - z_d \end{vmatrix} = \frac{1}{6} \cdot \begin{vmatrix} x_a & y_a & z_a & 1 \\ x_b & y_b & z_b & 1 \\ x_c & y_c & z_c & 1 \\ x_d & y_d & z_d & 1 \end{vmatrix}$$

Donde $D=(x_d, y_d, z_d)$, $A=(x_a, y_a, z_a)$, $B=(x_b, y_b, z_b)$, $C=(x_c, y_c, z_c)$. Puede probarse de forma trivial que el tetraedro tiene una orientación positiva si el volumen signado es positivo, esto es, el resto de vértices se disponen en contra de las agujas del reloj desde la parte opuesta del punto. En el caso de que los cuatro puntos sean coplanarios, el volumen del tetraedro seria 0.

Al igual que ocurría con los triángulos, se define volumen signado del tetraedro [ORourque94] de vértices *D*, *A*, *B* y *C*, denotado por [*DABC*] como el volumen del tetraedro teniendo en cuenta su signo, es decir:

[*DABC*] = *sign*(*Volumen*(*DABC*))

Al igual que con los triángulos, a partir de este momento cuando nos refiramos al signo de un tetraedro no estaremos refiriendo al volumen signado del mismo. Se demuestra fácilmente que el tetraedro tiene orientación positiva (es decir, el resto de vértices se ven en sentido contrario del reloj desde el lado opuesto a un punto) si el volumen con signo es positivo. Nuevamente, si uno de los vértices del tetraedro coincide con el origen de coordenadas, el cálculo del volumen signado del tetraedro se simplifica enormemente, pasando de un determinante 4 × 4 a un determinante 3 × 3.

Definición 3.22. El volumen signado de la pirámide *P* con vértice *V* y base *S*, está denotado por *[P]* y coincide con el volumen de *P* si la pirámide tiene una orientación positiva, y con el volumen negativo de *P* si la orientación es negativa. Si el vértice de la pirámide coincide con el origen de coordenadas se dice que es una *pirámide original*.

Definición 3.23. El signo de la cara *F* de un poliedro genérico, denominado por *[F]*, es el signo de la pirámide obtenida mediante la unión de la cara con el origen de coordenadas.

Con las definiciones anteriores, como ya hemos mencionado, Feito desarrolló un sistema generador de objetos gráficos, válido para dos, tres o más dimensiones, en el que los objetos gráficos se construyen a partir del recubrimiento del volumen del objeto mediante símplices originales. Para el espacio bidimensional, los símplices utilizados serán triángulos; para el espacio tridimensional, los símplices serán tetraedros.

Teorema 3.1. [Feito97b] Sistema generador. Sea un sólido *S* con caras $F_1F_2F_3...F_m$, dado en una orientación consistente (los vectores normales se orientan hacia fuera del sólido). Entonces:

$$S = \sum_{i=1}^{m} \left([P_i] * P_i(\mu_0, \alpha_0) \right)$$

donde P_i representa la pirámide original obtenida mediante la unión de la cara F_i con el origen de coordenadas, con la función constante de presencia μ_0 y la función constante de aspecto α_0.

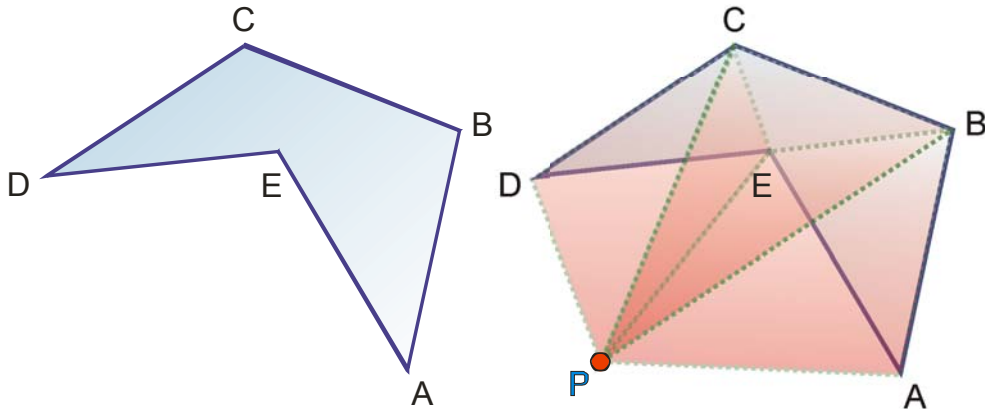


Figura 3-2. Un polígono y su recubrimiento simplicial asociado utilizando el punto adicional P. Puede verse cómo los triángulos del recubrimiento se solapan.

Demostración (ver [Feito97b]). En el caso de polígonos 2D, la ecuación del teorema 1 también es válida, aunque en lugar de pirámides se tienen triángulos OF_i, definidos entre el origen y el lado F_i del polígono, cuyo signo queda definido por su área signada. En lugar de utilizar pirámides se pueden usar tetraedros, lo que permite la simplificación de los cálculos. Como puede verse, las pirámides no tienen por qué ser disjuntas. Esto permite trabajar con los recubrimientos de los sólidos, en lugar de particiones disjuntas de los mismos. La principal ventaja de este enfoque es que el recubrimiento puede obtenerse de forma muy sencilla con un algoritmo lineal, manteniendo la representación inicial del sólido (un grafo vértice-lado-cara). Otra ventaja es que no es necesario almacenar la triangulación del sólido; tan sólo es necesario conocer los lados y un punto arbitrario, sin tener que almacenar información adicional.

Definición 3.24. Sea P un polígono, el recubrimiento de P, denotado como C_p, es el conjunto de triángulos obtenidos mediante la unión de un punto arbitrario del plano de P (por ejemplo, el centroide del polígono) con cada lado del polígono. Análogamente, sea S un sólido, el recubrimiento de S, denotado como C_s, es el conjunto de tetraedros obtenidos mediante la unión de cada triángulo del recubrimiento de cada cara de S con un punto arbitrario (por ejemplo, el centroide del sólido). La Figura 3-2 muestra un ejemplo de recubrimiento simplicial creado a partir de un polígono.

Teorema 3.2. [Feito97]. Sea un punto Q y un sólido S. Entonces Q está dentro de S si:

$$\sum_i sign(Q,T_i)\cdot[T_i]=1$$

donde $T_i \in C_S$, $[T_i]$ es el volumen signado (o el área signada en 2D) del símplice, y la función $sign(Q,T_i)$ devuelve el volumen signado del símplice formado por un punto Q y el triángulo T_i (un lado en 2D y una cara en 3D).

Corolario 1. Sea un punto Q interior a S. Entonces $\exists T_i \in C_S$, $[T_i]\geq 0$ / $Q \in T_i$.

Demostración. De forma trivial puede verse que, cuando se calcula la inclusión de un punto en un sólido, sólo se utilizan operaciones de suma algebraica. Por tanto, en cualquier momento se cumple que el signo de T_i debe ser positivo para obtener un resultado positivo. También es trivial demostrar que los puntos del sólido incluidos en un T_i negativo están también incluidos en, al menos, dos T_i positivos, ya que el resultado debe ser positivo.

Corolario 2. Sea Q un punto y un tetraedro original positivo $T=(OABC)$, entonces Q está dentro de T si:

$$[OABQ]\geq 0 \quad y \quad [OBCQ]\geq 0 \quad y \quad [OCAQ]\geq 0$$

Demostración. Puede ser probado trivialmente considerando el caso particular de tetraedros para sólidos del teorema 3. En el caso de tetraedros negativos el sentido de la comparación debe cambiarse.

El aspecto más destacable de los teoremas anteriores es que reducen la composición de cualquier sólido, variedad o no-variedad, con o sin agujeros, cóncavo o convexo, a la unión regularizada de objetos simples (triángulos en 2D o tetraedros en 3D). Este tipo de descomposición es denominado recubrimiento, no precisándose necesariamente el que los símplices utilizados a la hora de realizar el recubrimiento tengan por qué ser disjuntos. Este hecho presenta numerosas ventajas frente a otras aproximaciones basadas en la descomposición de los sólidos. Por una parte, un recubrimiento de un polígono en 2D mediante triángulos puede realizarse de manera muy sencilla mediante un algoritmo en tiempo lineal O(n), siendo n el número de vértices del polígono; en cambio, una triangulación (o división del polígono en triángulos disjuntos) del mismo polígono conlleva un algoritmo en O(n·log n). Si bien Chazelle [Chazelle91] propone un algoritmo lineal para realizar la triangulación del polígono, no se ha llevado a la práctica con éxito.

Por otra parte, la utilización de recubrimientos unifica el tratamiento que se hace de los objetos gráficos en espacios n-dimensionales, lo que permite que los algoritmos presentados en una dimensión sean fácilmente extensibles a dimensiones

superiores. De hecho, y como veremos más adelante, muchos de los algoritmos propuestos en la presente tesis, derivados en su mayoría de los propuestos por Feito, son válidos para varios espacios dimensionales, cambiando únicamente la definición de símplices según la dimensión en la que nos encontremos.

Además, y como ya hemos mencionado, el sistema generador desarrollado por Feito es válido para cualquier tipo de sólido poliédrico, ya sea variedad o no-variedad, con o sin agujeros, cóncavo o convexo, etc. Ello proporciona un mecanismo de modelado en el que los casos especiales no deben ser tratados de forma independiente, sino que los algoritmos desarrollados se muestran sencillos de implementar al no haber distinción entre los distintos tipos de sólidos. El sistema generador es válido tanto para sólidos homogéneos como para sólidos heterogéneos. Para el tratamiento de los sólidos compuestos por un número finito de componentes regulares, basta con descomponerlo en un número finito de objetos homogéneos, los cuales pueden expresarse como una combinación de los objetos gráficos del sistema generador. Una conclusión importante es que estos objetos gráficos generadores pueden ser originales. Esta propiedad del modelado propuesto por Feito lo diferencia de la mayoría de las soluciones existentes basadas en la representación de los sólidos mediante sus fronteras, y en las que el tratamiento de los casos especiales requiere tratamientos exhaustivos que en la mayor parte de las ocasiones complican enormemente los algoritmos y las estructuras de datos.

3.5 Representación de sólidos mediante recubrimientos simpliciales

Definición 3.25. Un poliedro P en 3D es un conjunto de puntos cerrado, limitado por un conjunto de caras planas, $C=\{C_0,C_1,...,C_n\}$. Cada una de las caras es un polígono de acuerdo con la definición 3.24.

Con lo definido hasta ahora se cuenta con una nueva representación de sólidos poliédricos con caras planas, independientemente de la naturaleza de las mismas [Segura01]. Para ello hay que almacenar la siguiente información:

- Como información geométrica, se almacenarán todos y cada uno de los vértices de las caras. Para evitar duplicidad en los mismos, se mantendrá una lista obtenida mediante la unión de las listas de vértices correspondientes a las caras que delimitan el poliedro. Así mismo, y para reducir los cálculos necesarios en los algoritmos, se almacenará también el centro de gravedad del poliedro, y la caja envolvente, dada mediante dos puntos que delimitan la esquina inferior izquierda y superior derecha de dicha caja envolvente.

- En cuanto a la información topológica propiamente dicha, únicamente es necesario almacenar la información referente a la conectividad de los vértices. Para ello, en lugar de almacenar tantas listas como caras tenga el poliedro, lo que se hace es mantener una única lista de índices, de manera que en dicha lista se incluyen referencias a los vértices que delimitan cada cara. Para separar la definición topológica de una cara de la siguiente se utiliza el valor *-1* como delimitador, ya que éste es un índice imposible en una lista que comienza a indexarse a partir de 0.

Con esta información se dispone ya de todos los elementos necesarios para realizar los recubrimientos de cada una de las caras mediante triángulos. Es importante destacar que si el número de vértices de la cara es menor o igual que tres no se procede a realizar el proceso de recubrimientos mediante triángulos. Esto es especialmente útil cuando los objetos que se van a representar vienen dados mediante mallas triangulares, ya que el tamaño de la información a almacenar se reduce de manera considerable.

3.5.1 Aplicación a mallas de triángulos

Ya que las mallas de triángulos son la estructura de datos principal que se va a utilizar en el resto de la tesis, basta recordar que un sólido poliédrico de caras planas siempre puede representarse mediante los recubrimientos simpliciales de sus caras formados por triángulos. Esto es aplicable a la mayoría de los algoritmos presentados en este trabajo.

De esta forma, los algoritmos se aplicarán sobre mallas de triángulos de forma directa o, cuando sea posible, sobre triángulos obtenidos de los recubrimientos de los polígonos que componen la frontera. Por esto, la estructura de datos empleada en todos los algoritmos representa de forma directa mallas de triángulos.

3.6 Determinación de propiedades de los sólidos

Con las estructuras de datos presentadas es fácil construir algoritmos para la determinación de algunas de las propiedades de los sólidos. Aparte de otras aplicaciones, el modelado de sólidos mediante recubrimientos simpliciales se ha aplicado con resultados satisfactorios en problemas de detección de colisión tanto en 2D como en 3D [Jimenez05, Jimenez06]. Este Apartado se centra en la obtención de propiedades relacionadas con la superficie y el volumen.

3.6.1 Determinación del centroide de un sólido 3D

El cálculo del centroide de un sólido poliédrico en tres dimensiones es especialmente importante en problemas de Ingeniería e Informática Gráfica, para temas como el cálculo de la esfera envolvente para la detección de colisiones. Teniendo en cuenta la estructura de datos propuesta, el centro geométrico de cada una de las caras es calculado en el momento en el que se almacena cada cara, ya que es este punto el que se ha tomado como punto de referencia a la hora de realizar el recubrimiento de los polígonos que constituyen la cara. Así pues, puede calcularse fácilmente el centro geométrico de un sólido a partir de los centros geométricos de cada una de las caras que lo constituyen, utilizando la siguiente ecuación:

$$CG = \frac{\sum_{i=0}^{n} CG_i * K_i}{\sum_{i=0}^{n} K_i}$$

Siendo K_i el número de vértices de la cara i.

3.6.2 Área de un sólido

Para calcular el área del sólido basta con realizar la suma de las áreas de las caras que lo componen. Ya que estamos trabajando con sólidos poliédricos y los recubrimientos de sus caras a triángulos, la sumatoria debe tener en cuenta el signo de cada triángulo. Esto es, para una cara de más de 3 lados puede haber triángulos negativos. Cuando se tiene directamente una malla de triángulos de partida, el signo de los triángulos es irrelevante, ya que siempre el positivo.

3.6.3 Volumen de un sólido

Para obtener el volumen total del sólido basta con sumar los volúmenes signados de los tetraedros que componen su recubrimiento simplicial en 3D. El Algoritmo 3-1 muestra el listado para realizar esta operación.

```
double solido::volumen ()
{
   double ac = 0;
   point3 o ( 0, 0, 0 );

   for ( int i = 0; i < listaCaras.numero; i++ )
   {
      for ( int c=0; c<listaCaras[i].listaTriangulos.numero; c++)
      {
         tetraedro t ( o, listaCaras[i].listaTriangulos[c] );
         ac += t.volumen (); //el volumen del tetraedro es signado
      }
   }

   return ac;
}
```

Algoritmo 3-1. Algoritmo para determinar el volumen de un sólido.

Capítulo 4

Test de Inclusión Punto en Sólido Optimizado

4

En este Capítulo se presenta un algoritmo de inclusión de puntos en sólidos B-Rep de caras planas mediante la utilización de recubrimientos simpliciales, así como una serie de optimizaciones que hacen el proceso mucho más eficiente, incluyendo una implementación en GPU programable [Ogayar06]. También se hace una comparación con otros métodos frecuentemente utilizados con el fin de hacer una reflexión sobre la conveniencia de cada técnica en cada caso, teniendo en cuenta los requerimientos de memoria, CPU y GPU [Ogayar05]. Las opciones que se han estudiado son algunas basadas en estructuras de enumeración espacial, la solución clásica basada en el Teorema de la Curva de Jordan y la propuesta variante de Feito-Torres [Feito97] aplicada a mallas de triángulos [Segura05].

4.1 Introducción

Como se mencionó en el Capítulo 2, la ***clasificación de un punto*** del espacio respecto de cualquier primitiva geométrica es el punto de partida de multitud de algoritmos geométricos. El problema consiste en, dado un punto, determinar su posición respecto de la primitiva. En el caso del ***test punto-en-sólido***, éste consiste en determinar si el punto especificado queda incluido dentro del sólido. La resolución eficiente y robusta de este problema permite el desarrollo de otros algoritmos de mayor nivel, como la detección de colisiones o las operaciones booleanas.

En el caso particular de sólidos representados mediante su frontera, la prueba de inclusión consiste en determinar si el punto está sobre la frontera, en el espacio delimitado por dicha frontera, o fuera del mismo. Según el criterio que se tome, la frontera puede considerarse o no parte del interior, por lo que el test será positivo o negativo si el punto está sobre la frontera respectivamente. En este trabajo se trata el algoritmo de inclusión con sólidos B-Rep de caras planas, esto es, el caso en el que la frontera está compuesta por polígonos.

4.2 Revisión de soluciones

A continuación se presentan las soluciones más habituales para la resolución del problema de la inclusión de un punto en un sólido utilizando una representación B-Rep de caras poligonales. Existen varias técnicas para la clasificación de puntos en sólidos poliédricos, cada una especializada para ser utilizada en unas condiciones concretas [Kalay82, Lane84, Horn89, Joan96, Ogayar03, Ogayar03b, Ogayar05]. Realmente, hay tantos algoritmos de inclusión como esquemas de representación, sin embargo, este trabajo se centrará con sus pruebas en los que presenten resultados precisos sobre representaciones B-Rep. No obstante, a continuación se muestran las soluciones que se utilizan con más frecuencia, incluyendo algunas conversiones de representación empleadas en casos donde no es necesaria la precisión.

4.2.1 Algoritmos de inclusión no exactos

En ocasiones es conveniente tener un algoritmo de inclusión muy rápido aunque los resultados no sean muy exactos. En esta línea surgen algunos algoritmos de detección de colisiones o conversión de representaciones. A continuación se exponen algunas soluciones basadas realmente en la conversión de B-Rep a otra estructura que permite una clasificación muy rápida aunque poco precisa.

4.2.1.1 Algoritmo de inclusión basado en Espacio de Voxels

El algoritmo del ***grid*** ó ***espacio de voxels*** consiste en dividir el espacio utilizando una segmentación uniforme, mediante la cual se definen celdas contenidas total o parcialmente dentro del sólido. La inclusión de cada celda debe ser calculada con algún algoritmo adicional, como el basado en el teorema de la curva de Jordan. Lo que suele hacerse es probar la inclusión del centro de cada voxel, aunque pueden utilizarse métodos más complejos, como el muestreo de varios puntos contenidos en el voxel para establecer el estado de inclusión mayoritario (inclusión total con imprecisión), o bien determinar un estado de inclusión parcial. También pueden

utilizarse métodos estocásticos y probabilísticos para determinar la inclusión usando varias muestras.

Como puede verse, la clasificación de un punto respecto del sólido vendrá determinada por el estado de inclusión del voxel donde queda contenido. Si el voxel está etiquetado como parcialmente incluido, deberá utilizarse otro método para determinar el resultado exacto. Para obtener una representación más o menos aproximada es necesario utilizar mucha resolución en la retícula 3D, así como contar con mecanismos para representar los estados de inclusión parcial. Si se utilizan voxels binarios (sólo permiten inclusión o exclusión total) suele aparecer el problema del ***aliasing***, que se acentúa en grids de baja resolución. Los principales problemas del grid son la dependencia respecto de un algoritmo adicional de inclusión para calcular el estado de los voxels, así como el espacio 3D que se desperdicia como consecuencia de una segmentación uniforme. Como ventaja, hay que destacar la rapidez de los métodos de consulta de la estructura.

4.2.1.2 Algoritmo de inclusión basado en Octree

En esta ocasión, la estructura se calcula teniendo en cuenta la distribución del sólido, de forma que se generan octantes de forma recursiva en aquellas partes donde se concentran los polígonos. Esto produce un gran ahorro de memoria, y sobre todo, elimina el número de tests de inclusión para construir la estructura, ya que habrá muchos menos octantes que voxels en el grid equivalente. Como con el espacio de voxels, aparecen problemas de aliasing y el rendimiento vuelve a estar en función del algoritmo seleccionado para la inclusión inicial de puntos pertenecientes a cada octante. En esta ocasión, el recorrido por la estructura no es tan eficiente como en el grid, aunque existen métodos paramétricos [Revelles00, Garga93] que reducen esta diferencia drásticamente.

Para atenuar el aliasing producido con el grid y el octree debe utilizarse un método de muestreo que permita obtener una clasificación aproximada de la celda (voxel u octante), esto es, utilizar varios puntos y realizar una ponderación sobre sus estados de inclusión.

La navegación por el octree es trivial y por tanto, una vez construido el octree que representa al sólido, el test de inclusión de un punto en un subespacio es tan sencillo como profundizar en el árbol hasta llegar a un nodo raíz válido, o ser descartado en el proceso, lo que reduce la complejidad de la clasificación a orden $O(\log n)$.

4.2.1.3 Algoritmo de inclusión basado en kd-tree

Este algoritmo implica la construcción de un kd-tree o un BSP alineado con los ejes. Ya que deben existir nodos del árbol cuyo estado de inclusión sea positivo y completo, debe profundizarse mucho en la creación de la estructura para evitar el aliasing, que es mucho más grave en el bintree que en el kd-tree (permite adaptar los planos de corte). En cualquier caso, y de igual forma que ocurría con el octree, la clasificación de los puntos es muy eficiente, ya que se realiza en orden *O(log n)*. Aunque el kd-tree es fácil de construir y recorrer proporciona una representación aproximada, por lo que si se busca la precisión la mejor variante de BSP es la alineada con polígonos que se presenta más adelante.

4.2.2 Algoritmo de inclusión basado en el Teorema de la Curva de Jordan

Los métodos más utilizados para la inclusión de puntos en cualquier entidad geométrica están basados en el Teorema de la Curva de Jordan [ORourque94]. Esta técnica es extremadamente popular debido a su simplicidad y a su extenso campo de aplicación. El teorema dice "*Sea C una curva simple y cerrada en el plano, entonces C separa al plano en exactamente 2 componentes de modo que dichas componentes tienen a C como frontera*" [Munkres01]. En la práctica, al lanzar un rayo desde un punto en una dirección aleatoria, se producirá un número de intersecciones con la frontera; si este número es impar, el punto queda dentro, y si es par, el punto queda fuera. Este teorema, que está definido para 2D es extensible a 3D, siempre y cuando se trate con sólidos, esto es, que exista una superficie continua que divida al espacio en dos dominios sin puntos en común, y que sea dicha superficie la frontera compartida.

Los algoritmos de inclusión basados en el Teorema de la Curva de Jordan no necesitan estructuras de datos precalculadas, de hecho, es considerada como la forma más rápida para comprobar la inclusión de un punto en un sólido sin realizar ningún preprocesamiento [Möller02], aunque como se demuestra en este trabajo, es mejorado ligeramente por el método de Feito-Torres basado en el estudio de signos [Ogayar05]. El principal problema del método de Jordan es que el proceso debe ser repetido, cambiando la dirección del rayo, cuando se detecta una intersección con un vértice o una arista, así como en el caso en que el punto está contenido en una cara o el rayo trazado es coplanar a uno de los triángulos. Como puede suponerse, la elección del método de intersección rayo-triángulo influirá decisivamente en la eficiencia y precisión del algoritmo de inclusión las elecciones más acertadas son [Badouel90] ó [Möller97], que utilizan coordenadas baricéntricas.

Una optimización conveniente para acelerar el algoritmo de intersección rayo-triángulo del método de Jordan consiste en la utilización de una estructura de

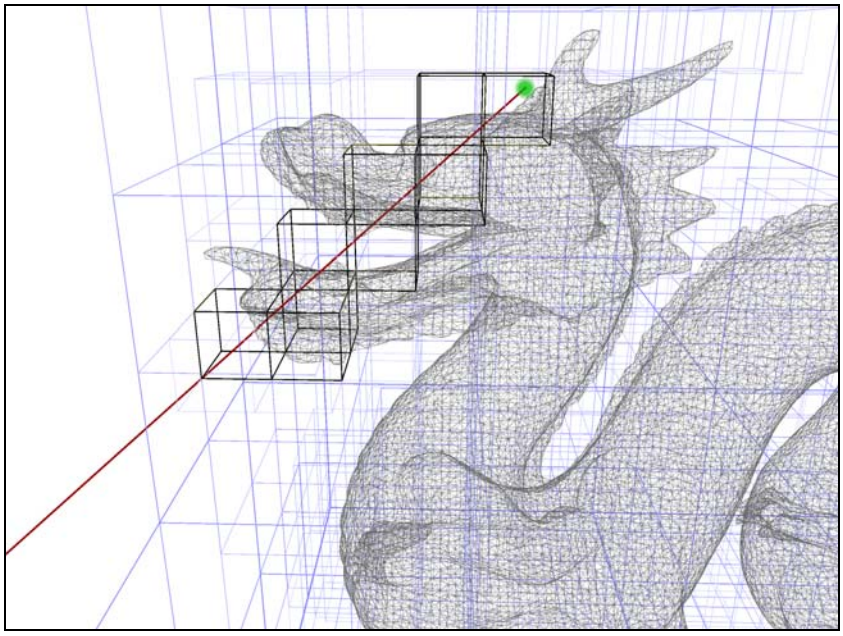


Figura 4-1. Optimización espacial basada en octree. La imagen muestra nodos hoja intersectados por un vector (en rojo) desde un punto interior (en verde). Los voxels marcados son el resultado de ejecutar Gargantini [Gargantini93] sobre el octree.

segmentación espacial, como el BSP o el octree, que permita descartar fácilmente un gran número de intersecciones. En este estudio se ha utilizado el octree, junto con un algoritmo paramétrico de recorrido del mismo [Gargantini93], aunque hay otros métodos alternativos como [Revelles00]. Usando un octree, sólo los polígonos incluidos en los voxels intersectados por el rayo son pasados al algoritmo de intersección rayo-triángulo, lo que disminuye enormemente el número de intersecciones global del rayo con los polígonos.

Nótese la diferencia entre utilizar estructuras de segmentación espacial para almacenar estados de inclusión de subespacios (voxels) y la utilización de dichas estructuras como optimizadores en procesos como en el trazado de rayos. De esta forma, no es lo mismo utilizar un octree como estructura de inclusión que como optimizador para la intersección rayo-triángulo utilizada en el algoritmo de la curva de Jordan.

4.2.3 Algoritmo de inclusión basado en BSP

El ***BSP*** es una forma extremadamente popular de organización espacial, y no es una excepción en el modelado de sólidos. Para la prueba de inclusión de puntos en objetos B-Rep utilizando esta estructura, basta con construirla según se especifica en el Apartado 2.3.4.4. Al clasificar un punto como exterior o interior utilizando esta estructura, basta con recorrer el árbol comprobando en cada nodo la posición relativa del punto respecto del plano asociado a dicho nodo, hasta llegar a un nodo hoja. Los nodos que representan cada subespacio en un nivel determinado del árbol binario están etiquetados de igual forma que el signo del subespacio al que representan: positivo y negativo. Cuando se llega a un nodo hoja, utilizando el plano asociado, se comprueba la posición del punto; si queda en el subespacio positivo el punto se considera fuera del sólido, si queda en el subespacio negativo, se considera dentro. En el caso de estar situado sobre el plano de corte, es cuestión de criterio considerar su posición; dependerá de la inclusión de las caras en el sólido, esto es, si se prefiere que las caras formen parte del interior, un punto sobre el plano de corte quedará dentro del sólido.

Para la inclusión de puntos en mallas de triángulos, el BSP ideal es el alineado con los polígonos, ya que es una representación exacta. Los nodos hoja en la estructura contienen subconjuntos convexos del sólido. El test de inclusión de puntos es muy eficiente utilizando esta estructura. El problema es la gran cantidad de memoria consumida, especialmente cuando se producen muchas divisiones de triángulos entre los distintos subespacios (esto ocurre sobre todo cuando se presentan concavidades en los sólidos), lo que obliga a calcular más ramificaciones en el árbol.

4.2.4 Algoritmo de inclusión basado en PM-Octree y SP-Octree

Tanto en los PM-Octrees como en los SP-Octrees se almacena información sobre la frontera del sólido en los nodos, con lo que la representación es exacta y además mantiene la eficiencia inherente en la clasificación jerárquica a la hora de indexar elementos.

El test de inclusión con estas estructuras es muy eficiente, sin embargo no son representaciones que puedan obtenerse fácilmente a partir de un B-Rep. Esto hace que el preprocesamiento sea muy costoso, por lo que globalmente el BSP suele ser más eficiente, aún cuando su construcción también puede resultar costosa.

4.3 Algoritmo de inclusión basado en Recubrimientos Simpliciales

El algoritmo de inclusión basado en ***recubrimientos simpliciales*** de Feito-Torres [Feito97] es un método que comprueba la inclusión de un punto en un poliedro genérico sin resolver ningún sistema de ecuaciones. Es sencillo, robusto, y puede aplicarse en cualquier caso. Trabaja con representaciones B-Rep de caras planas genéricas, alcanzando su máxima eficiencia con mallas de triángulos.

El método de Feito-Torres aplicado a sólidos poliédricos es una extensión del algoritmo de inclusión en 2D presentado en [Feito95]. En [Feito97] se presentan los fundamentos teóricos tanto del caso 2D como del caso 3D; aquí se resumen las características generales del método 3D [Ogayar05, Segura05]. Los conceptos básicos se presentan en los Apartados 3.4 y 3.5 y en [Segura01].

4.3.1 Descripción del algoritmo

Dado un sólido definido mediante una malla de polígonos y un punto para calcular su inclusión en dicho sólido, el algoritmo de Feito-Torres se basa en la descomposición del sólido en ***símplices***, que para el caso 3D resulta en un conjunto de ***tetraedros***. Este conjunto de tetraedros forma el recubrimiento simplicial del sólido. Calculando la inclusión del punto en cada uno de los tetraedros del recubrimiento puede determinarse la inclusión del punto en el sólido [Feito97].

El caso más sencillo y eficiente del algoritmo se presenta con mallas de triángulos. Cuando se procesa una estructura con polígonos de más de tres lados, cada uno de éstos se reduce a un conjunto de triángulos, que componen su recubrimiento simplicial 2D [Feito95]. De esta forma, se utilizan los recubrimientos simpliciales en dos fases del algoritmo: en primer lugar, para convertir cualquier polígono de más de tres lados en un conjunto de triángulos, y en segundo lugar en la parte principal del algoritmo, que trabaja con tetraedros.

A partir de ahora se considerará que el sólido está representado mediante una ***malla de triángulos***, aunque parte de éstos formen el recubrimiento de algún polígono de más de tres lados. Otra opción para convertir una malla de polígonos en una malla de triángulos es la ***teselación***, esto es, la conversión de cada polígono a un conjunto de triángulos disjuntos que cubren la misma superficie del polígono original. Aunque esta solución es la adecuada para el tratamiento y visualización de polígonos basada en triángulos, en este algoritmo no es necesario y, puesto que puede llegar a ser un proceso bastante costoso, no se utiliza en el algoritmo de inclusión.

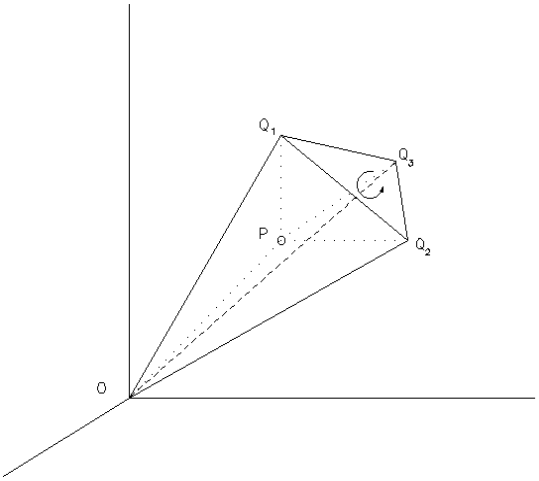


Figura 4-2. Inclusión de un punto en un tetraedro.

Cada tetraedro del recubrimiento del sólido está formado por los vértices de cada triángulo de la malla que representa al objeto y un punto adicional, común a todos los tetraedros del conjunto. Para simplificar los cálculos, y sin perder generalidad, se utiliza el origen de coordenadas como el vértice adicional para cada tetraedro. A este tipo de tetraedro, en el que uno de los vértices coincide con el origen de coordenadas se le llama ***tetraedro original***. Hay que recordar que el conjunto de tetraedros no tiene que ser necesariamente disjunto, esto es, existe la posibilidad de que se solapen, lo que ocurre casi siempre que hay concavidades en el sólido. La utilización de tetraedros originales viene motivada por la simplificación del cálculo de sus signos, que se realiza mediante determinantes.

Una vez que está formado el recubrimiento simplicial del sólido, para determinar la inclusión de un punto cualquiera, éste se clasifica respecto de cada tetraedro: si está en el interior, se acumula el signo del tetraedro en una variable. El valor final de este acumulador determina si el punto está dentro del sólido o fuera.

Donde la función *signo(x)* es la introducida en la Definición 3.19, y que devuelve *-1* para un *x* negativo, *+1* para un *x* positivo y *0* si *x* es *0*. Si el tetraedro tiene orientación negativa cada una de las desigualdades debe ser invertida. Asumiendo que no hay triángulos degenerados a la hora de calcular los determinantes, esto es, triángulos con sus tres vértices colineales, se puede determinar también si el punto está sobre una de las caras o aristas del tetraedro original.

Si uno de los signos de la condición de la ecuación del Corolario 2 de la Ecuación 3.2 es 0 significa que el punto está en una cara. Si dos de los signos son 0 el

```
Inicializar los conjuntos de vértices positivos
(PosVert) y negativos (NegVert) a vacío

acum=0;

foreach  Ti=( Qi1Qi2Qi3) del recubrimiento {
 if  (P ∈ Ti) return (TRUE);
 if (P∉ (OQi1Qi2Qi3)) continue;
 if (P está en el interior de (OQi1Qi2Qi3))
  acum+=2*sign([OQi1Qi2Qi3]);
 else if (P está en una cara de (OQi1Qi2Qi3))
   acum+=sign([OQi1Qi2Qi3]);
  else  // P está en la arista OQij
   if (sign([OQi1Qi2Qi3])>0) { Tetraedro positivo
    if (Qij ∉ PosVert) {
     insertar Qij en Posvert
     acum+=2;
    }
   }
   else {   // El tetraedro tiene signo negativo
    if (Qij ∉NegVert)  {
     insertar Qij en NegVert;
     counter-=2;
    }
   }
}

return (acum == 2);
```

Algoritmo 4-1. Algoritmo de inclusión punto-en-sólido de Feito-Torres.

punto se encuentra en una arista. Este hecho habrá de ser tenido en cuenta para los casos especiales. Así, si el punto se encuentra sobre una cara del tetraedro distinta a la base (la que coincide con el triángulo original del objeto), entonces el algoritmo acumula ½ del signo del tetraedro, ya que dicha cara estará compartida por dos tetraedros.

En el caso de que el punto se encuentre en una de las aristas originales, entonces se consulta la presencia del vértice de la arista en dos conjuntos de vértices (uno para los positivos y otro para los negativos, según el signo del tetraedro); si el punto no está en el conjunto se acumula el signo del tetraedro y se inserta el vértice en el conjunto. La idea de esta solución es considerar las aristas sólo una vez, ya que en este caso no se conoce cuantas aristas del sólido de partida comparten este vértice. El Algoritmo 4-1 muestra el listado.

El algoritmo de Feito-Torres no requiere de preprocesamiento y es totalmente independiente de la topología del sólido, si bien hay ciertas optimizaciones que aceleran drásticamente los cálculos. Además, es válido para modelos con agujeros y no variedad.

4.3.2 Optimizaciones

Las optimizaciones que admite el algoritmo de inclusión están relacionadas con el precálculo de algunos elementos, así como el empleo de indexadores espaciales para los tetraedros. Algunas optimizaciones son compatibles con otras, con lo que se combina la ganancia en rendimiento. En cualquier caso, cada una tiene sus requerimientos en cuanto a tiempo de procesamiento y memoria, por lo que pueden emplearse según la situación. Hay que recordar que el algoritmo básico no necesita ningún preprocesamiento para ofrecer resultados robustos. En este Apartado no se incluye la implementación en GPU del algoritmo, aunque pudiera verse como una optimización. Dicha versión es presentada en el Apartado 4.3.3.

4.3.2.1 Preprocesamiento básico

Esta optimización consiste en precalcular el signo de cada tetraedro original. Este proceso podría optimizarse más si se dispusiera de antemano de la normal de cada cara del sólido. En este sentido hay que destacar que muchos de los formatos de almacenamiento de mallas de polígonos incorporan esta información, por lo que no requeriría de cálculos adicionales.

4.3.2.2 Jerarquía de cajas envolventes

Con este método se precalculan las cajas envolventes de cada tetraedro original (formado por los vértices de cada triángulo y el origen de coordenadas) y se organizan de forma jerárquica. En cada nodo de la estructura se almacenan las coordenadas de la mínima caja envolvente que engloba a todas los volúmenes de los niveles inferiores. Los nodos terminales contienen el volumen envolvente de un sólo tetraedro.

Esta opción es muy fácil de construir. El principal problema es que para la mayoría de los sólidos, la caja envolvente de cada tetraedro tiene un tamaño considerable. Esto hace que, en la mayoría de los casos, para un punto a incluir no suelen descartarse demasiados tetraedros en el algoritmo de inclusión. Además, el criterio ideal para organizar los tetraedros en grupos, y por tanto construir una jerarquía, es el de cercanía de los mismos, lo cual puede ser algo complicado si no se dispone de información de conectividad entre las caras del sólido.

4.3.2.3 Indexación mediante octantes

Esta es una optimización muy económica en cuanto a memoria y se calcula muy rápidamente. Está recomendada cuando la memoria es escasa. Consiste en centrar el sólido en el origen de coordenadas, por lo que utilizando los planos coordenados, el espacio queda dividido en 8 octantes. Cada octante se define con un código de 6 bits, esto es, a cada coordenada del espacio (x, y, ó z) le corresponden 2 bits que pueden indicar positivo (*01*) o negativo (*10*). Cada cara del sólido estará incluida total o parcialmente en uno o más octantes, por lo que se le calcula su código de 6 bits que indica en qué subespacios está presente. Al utilizar el origen de coordenadas como uno de los vértices de cada tetraedro que forma el recubrimiento simplicial del sólido, puede derivarse que el estado de inclusión del tetraedro es el mismo que el de su correspondiente cara del sólido. Por tanto, clasificando los triángulos del objeto, también quedan clasificados los tetraedros.

Cuando se realiza el test de inclusión, al punto a probar se le asigna un código que se corresponde con el octante al que pertenece siguiendo el criterio anterior. Para verificar la inclusión, se realiza una operación lógica *and* entre el código del punto y el de cada tetraedro original; si el resultado es 0, no hay ningún octante en común, y se puede concluir directamente que el punto queda fuera de dicho tetraedro.

4.3.2.4 Indexación mediante segmentación uniforme

Esta optimización consiste en calcular un *grid** 3D (rejilla 3D) de voxels que se ajusta a la caja envolvente del sólido. En cada voxel se almacena una lista de tetraedros que lo intersectan. Esta estructura es muy costosa en memoria, pero el rendimiento del algoritmo aumenta en gran medida según se construye el grid con mayor resolución. Para clasificar un punto en el sólido sólo hay que iterar en el algoritmo de inclusión sobre los tetraedros que intersectan el voxel donde queda incluido el punto. Esto permite descartar la gran mayoría de tetraedros del sólido.

La construcción del grid indexador puede hacerse de dos formas. La primera consiste en calcular los tetraedros que intersectan cada voxel. Esto obliga a iterar sobre toda la lista de tetraedros para cada nodo del grid, lo que es extremadamente costoso. Además, los cálculos que hay que realizar para intersectar un tetraedro con un voxel no son inmediatos, lo que empeora aun más el rendimiento. Afortunadamente, en el Apartado 5.3 se presenta un método muy eficiente para calcular la voxelización de un sólido. Mediante este proceso se logra la conversión de B-Rep a grid. Extendiendo este algoritmo, como se indica en la Sección 5.5, se puede almacenar en cada voxel la lista de

* *Se ha preferido utilizar el término* ***grid*** *en lugar de rejilla para hacer más clara la referencia a una estructura de datos y a su algoritmo optimizador asociado.*

	Preprocesamiento básico	Jerarquía de cajas	Octantes	Segmentación uniforme
Tiempo de cálculo	Bajo	Medio	Muy bajo	Medio*
Uso de memoria	Muy bajo	Medio	Muy bajo	Muy alto
Beneficio en el rendimiento	Medio	Bajo	Medio	Muy alto

(*) Usando una voxelización del sólido

Tabla 4-1. Características de los optimizadores para el algoritmo de Feito-Torres.

tetraedros que lo intersectan durante el proceso de rasterización. De esta forma se consigue el indexador espacial en muy poco tiempo.

El grid de voxels podría compactarse formando un octree, lo que ahorraría memoria. Sin embargo en la práctica se demuestra que, debido a las dimensiones y al solapamiento de los tetraedros, no se obtiene un beneficio significativo con esta clasificación espacial. Además, el tiempo de construcción es considerable, ya que para compactar los voxels hay que comparar listas de índices, que en ocasiones pueden ser muy grandes.

4.3.2.5 Comparación de optimizaciones

Las optimizaciones presentadas anteriormente tienen distintas características en función de su coste computacional, su uso de memoria y el aumento de rendimiento producido en el algoritmo. En la Tabla 4-1 se muestra una comparación de las distintas opciones. Es necesario recordar que algunas son compatibles entre sí, y otras excluyentes. La comparativa mostrada en la Tabla 4-1 refleja una valoración subjetiva de las características estudiadas. La calidad o bonanza de cada parámetro se indica de forma textual y como valores en el rango [1-10] codificados como estrellas, donde cada unidad se corresponde con media estrella.

4.3.3 Implementación en GPU programable

En este Apartado se presenta la adaptación del algoritmo para ser implementado utilizando el hardware gráfico. La solución planteada hace uso de las características especiales de las actuales GPUs programables para resolver el problema de una manera robusta y eficiente [Ogayar05b, Ogayar06]. Como se explica en el Apartado A.8, las GPUs programables actuales pueden ser utilizadas para resolver

problemas de tipo general, o de geometría computacional como la inclusión de puntos o la detección de colisiones [Jimenez05].

Para llevar a cabo esta adaptación a la GPU, se ha optado por utilizar OpenGL y Cg de NVidia [Kilgariff05, Fernando03], lo que permite un desarrollo bastante rápido y la posibilidad de sacar el máximo provecho del hardware [Moreland03, Purcell02, Purcell03]. En cualquier caso, el algoritmo es fácilmente adaptable a cualquier entorno que permita un acceso a la parte programable de la GPU [Buck04b], como DirectX, por ejemplo. En la el apéndice B se presentan más detalles sobre el entorno software y hardware utilizados.

Básicamente, el cálculo de la inclusión de un punto en un sólido descrito en el Algoritmo 4-1 es el siguiente:

1. Se descompone el sólido en tetraedros. Para conseguirlo, con los vértices de cada triángulo y el origen de coordenadas se construye un tetraedro. El origen de coordenadas, siempre considerado como el cuarto vértice del tetraedro, se denomina vértice original.

2. Se comprueba la inclusión del punto a probar en cada uno de los tetraedros (que pueden estar solapados). Esta inclusión será -1 si el punto está fuera, y +1 si está dentro.

3. Se suma el estado de inclusión del punto a probar en cada tetraedro. Si la suma es 1, el punto estará dentro, de lo contrario estará fuera.

Cuando el punto a probar se encuentra sobre una arista original (cualquier arista asociada con el cuarto vértice del tetraedro), o sobre una cara original (cualquier cara del tetraedro asociada con el cuarto vértice del mismo), se produce un caso especial. Si el punto está en una cara original, el estado de inclusión se multiplica por ½. Si está sobre una arista original, tal como se describe en el Algoritmo 4-1, deben utilizarse dos conjuntos que estarán asociados al vértice de la malla vinculado con dicha arista.

El algoritmo se divide en dos etapas para su ejecución en la GPU. En la primera fase se ejecuta un vertex shader o programa de vértices (consultar el Apéndice A) que calcula la inclusión del punto a probar en los tetraedros del recubrimiento del sólido. El pixel shader o programa de fragmentos (consultar el Apéndice A) hará una sencilla conversión de los datos. El resultado queda almacenado en el framebuffer*. En

** El término **framebuffer** hace referencia al buffer donde se almacenan las imágenes que genera la GPU. De la misma forma se prefiere utilizar términos como **pipeline** (en lugar de segmentación ó cauce) ya que están mucho más extendidos que sus traducciones al castellano.*

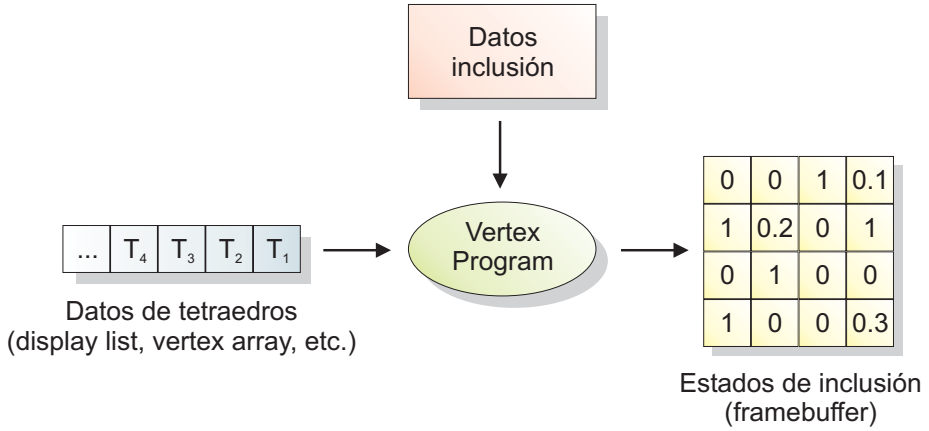


Figura 4-3. Procesamiento de datos en la primera fase del algoritmo. El vertex program se encarga de calcular el estado de inclusión de los puntos en los tetraedros de la malla. El resultado se almacena en el framebuffer.

la segunda fase otro pixel shader se encarga de realizar la sumatoria final. En esta fase no se utiliza vertex shader. La forma en que funcionan las dos etapas se explica a continuación.

4.3.3.1 Primera fase

La primera fase del algoritmo utiliza un vertex shader para realizar los cálculos de inclusión del punto a probar en todos los tetraedros del sólido. En esta fase el pixel shader sólo escribe en el framebuffer lo que le llega desde el vertex shader haciendo una conversión del rango numérico de [0,1] a [-1,1]. Lo habitual en un proceso de rendering es que el vertex shader transforme vértices y el pixel shader escriba pixels en el framebuffer según los valores interpolados de los vértices que intervienen en la primitiva que se está rasterizando. Lo que se busca en este algoritmo es que se ejecute el vertex shader una vez por cada tetraedro de la malla, y el resultado se escriba en el framebuffer. Para conseguirlo se realiza el siguiente proceso (ver Figura 4-3):

1. Se calculan los datos de los tetraedros asociados a la malla de triángulos y se convierten a un formato que la GPU pueda entender. La forma en que se ha resuelto es utilizando una display list de OpenGL para codificar los tetraedros como si fueran vértices con atributos que son enviados al hardware gráfico. Sobre estos vértices (no confundir con los puntos del tetraedro) se ejecuta un vertex shader que calcula la inclusión del punto a probar en el tetraedro representado por los atributos asociados a cada vértice procesado. De esta forma se busca una correspondencia 1 a 1 entre

cada uno de los tetraedros de la malla para el cálculo de la inclusión con un vértice con atributos a procesar en el vertex shader, y también con un pixel final del framebuffer. En el Algoritmo 4-2 puede verse la estructura *appin*, que define la semántica de los atributos de los vértices. Se emplea la normal, el color y las primeras coordenadas de textura para codificar tres puntos de un tetraedro (el cuarto es un origen de coordenadas fijo). La posición real del vértice se utiliza para situarlo en el viewport como si éste fuera una matriz 2D estándar. Además, en la coordenada z de la posición del vértice se codifica el signo del tetraedro.

2. Se rasteriza una matriz 2D de puntos en el viewport con OpenGL activando el vertex shader de inclusión, de forma que los puntos queden alineados con una matriz de n x n, siendo n potencia de 2. La razón de esto se explica en la segunda fase el algoritmo. En este paso, como programa de fragmentos se usa un pixel shader que escribe en el framebuffer el valor de inclusión que le llega desde el vertex shader multiplicado por el signo de cada tetraedro correspondiente. Esto es así debido a una limitación del vertex shader que fuerza la salida a [0,1]. También cambia el rango de [0,1] a [-1,1]. De esta forma se obtiene una correspondencia directa desde el vertex shader al framebuffer, donde cada pixel de la imagen contiene los datos de inclusión de cada tetraedro, así como su signo. El programa de pixels *convert* (ver Algoritmo 4-3) aplica el signo del tetraedro al valor de inclusión obtenido.

Ya que el framebuffer de flotantes utilizado puede ser de hasta 128 bits por texel (pixel de textura), se utilizan 4 valores en coma flotante de 32 bits para procesar en cada shader. Esto permite calcular en una sola ejecución del vertex shader el test de inclusión de 4 puntos distintos en cada tetraedro de la malla, lo que aprovecha el procesamiento vectorial natural de la GPU. El Algoritmo 4-2 muestra el listado en Cg para calcular la inclusión de puntos de 4 en 4 para cada tetraedro.

Una vez que el framebuffer contiene los estados de inclusión de todos los tetraedros, únicamente falta realizar la suma final de los datos, que constituye la segunda fase del proceso. Este puede realizarse de dos formas; copiando el framebuffer a memoria principal (necesario para el tratamiento de casos especiales) y haciendo los cálculos con la CPU, o realizando el paso completo en la GPU mediante el uso de un pixel shader.

```
#define TETRA_OUTSIDE     0
#define TETRA_INSIDE      1
#define TETRA_FACE        0.1
#define TETRA_EDGE1       0.2
#define TETRA_EDGE2       0.3
#define TETRA_EDGE3       0.4

struct appin
{
   float3 vertexData   : POSITION;
   float3 tetraVertex1 : NORMAL;
   float3 tetraVertex2 : COLOR0;
   float3 tetraVertex3 : TEXCOORD0;
};

struct vout
{
   float4 position : POSITION;
   float4 color    : COLOR0;
   float4 signs    : COLOR1;
};

vout tetraVertexProgram4 (
   appin IN,
   uniform float3 testPoint,
   uniform float3 testPoint2,
   uniform float3 testPoint3,
   uniform float3 testPoint4,
   uniform float3 v0,
   uniform float4x4 ModelViewProjectionMatrix )
{
   vout OUT;
   OUT.signs = IN.vertexData.z;
   OUT.color  = float4 (
   tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
    IN.tetraVertex1, testPoint ),
   tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
    IN.tetraVertex1, testPoint2 ),
   tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
    IN.tetraVertex1, testPoint3 ),
   tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
    IN.tetraVertex1, testPoint4 )
   );
   OUT.position = mul ( ModelViewProjectionMatrix,
    float4 ( IN.vertexData.x, IN.vertexData.y,0.0,1.0));
   return OUT;
}
```

Algoritmo 4-2. Vertex shader para el cálculo de la inclusión de 4 puntos en un tetraedro cuyos datos están especificados como un vértice con atributos.

El tratamiento de casos especiales se realiza de la siguiente manera: el estado de inclusión del punto a probar será 1 si está dentro y 0 si está fuera. Para el resto de casos se utilizan códigos especiales (pueden verse en el Algoritmo 4-2), ya que sólo se pueden utilizar números en el rango [0,1]. Estos códigos no podrán sumarse después de forma directa. Cuando en el vertex shader se detecta un caso especial, se traslada el vértice de la display list que está siendo procesado a una posición reservada del framebuffer que no se corresponda con ningún tetraedro de la malla (por ejemplo, la esquina superior derecha). Cuando termine la primera fase del algoritmo se comprueba si en esta posición especial hay algo escrito. La posición especial se comprueba con una transferencia de 1x1. Si es así, debe repetirse esta primera fase desactivando la comprobación de casos especiales, copiando el framebuffer a memoria principal y realizando la segunda fase del proceso por software. Se ha comprobado empíricamente que la frecuencia en la que se presentan casos especiales es muy baja, por lo que el algoritmo general no se ve prácticamente afectado en su eficiencia. En las tablas de resultados de este Capítulo, el método que se acaba de explicar aparece etiquetado como CPU+GPU.

4.3.3.2 Segunda fase

El objetivo del pixel shader en esta etapa es hacer la sumatoria de la matriz de números en coma flotante que se encuentra en el framebuffer. Para conseguirlo, se realizan los siguientes pasos:

1. El resultado de la primera parte del algoritmo debe estar disponible como textura en esta fase. Para ello, se copia el contenido del framebuffer resultante en una textura, o bien se utiliza en la fase anterior un rendering directo a textura. Dicha textura resultado contiene los estados de inclusión de los tetraedros, y está disponible como parámetro constante en el pixel shader de la etapa actual.

2. Se inicializa el proceso de rendering desactivando cualquier vertex shader que hubiera y habilitando la aplicación de textura.

3. Se dibuja un cuadrilátero que ocupe la cuarta parte de la textura de datos (que tiene las mismas dimensiones que el antiguo framebuffer), y situado en la esquina inferior izquierda siguiendo el sistema de coordenadas de OpenGL. Las coordenadas de textura se ajustarán al tamaño exacto de la textura. Como se usan texturas de flotantes, las coordenadas serán en pixels, no en formato [0,1] estándar. Si la textura de datos es de 512x512, la coordenada máxima será (511,511). El algoritmo es más fácil de implementar si el framebuffer es n x n, siendo n potencia de 2. Aunque

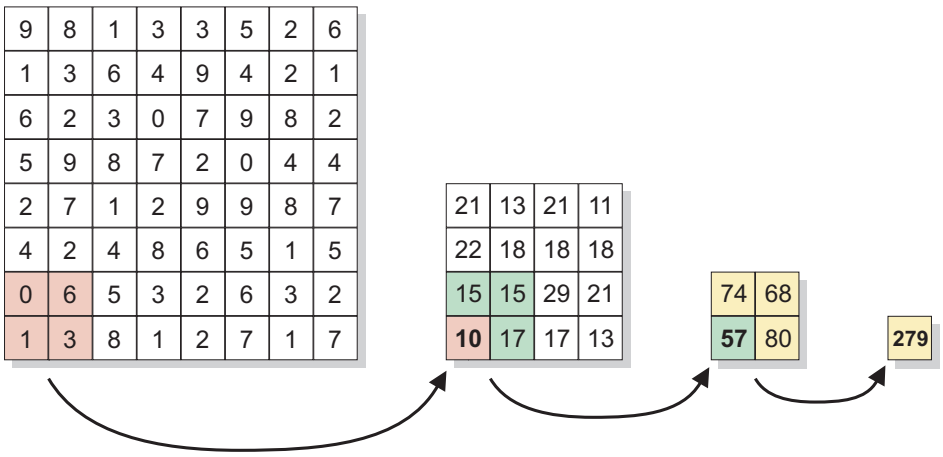


Figura 4-4. Ejemplo de sumatoria por bloques de 2x2 pixels. En cada paso se reduce la textura en un factor de 4 hasta llegar al resultado final. En algunas GPUs puede ser necesario añadir un offset (normalmente 0.5) a las coordenadas de textura utilizadas para un correcto direccionamiento de los texels.

esto suponga un desperdicio de espacio en la mayoría de las ocasiones, la velocidad de ejecución es mucho mayor, ya que el pixel shader es más simple.

4. Mediante el rendering del cuadrilátero anterior reducido en tamaño respecto de la textura de datos, se consigue que cada pixel de la imagen resultante se corresponda con un rectángulo 2x2 de la textura de datos. Por cada pixel resultado se ejecutará el pixel shader listado en el Algoritmo 4-3, que calcula la sumatoria de sus 4 valores correspondientes en la textura de origen.

5. Con lo anterior se obtiene una imagen 4 veces menor que contiene sumas parciales de la textura de datos. El resultado se copia desde el framebuffer a la textura de datos de origen para repetir el proceso hasta que la imagen resultado sea de 1x1 pixels, que será copiada a memoria principal. También puede utilizarse el rendering directo a textura utilizando buffers adicionales, estableciendo el origen de datos y el destino de forma conveniente.

El proceso de sumas parciales puede verse en la Figura 4-4. Aunque en cada iteración se reduce por 4 el tamaño de la matriz de datos, este factor puede ser variable implementando bucles dentro del pixel shader, ajustando el tamaño del cuadrilátero

```
float4 convert ( in float4 data : COLOR0, in float4
 signs : COLOR1 ) : COLOR0
{
   return data * (signs*2-1);
}

float4 accum ( float2 texcoord : TEX0,
 uniform samplerRECT img : texunit1 ) : COLOR
{
   float4 a, b, c, d;
   a = f4texRECT ( img, texcoord );
   b = f4texRECT ( img, texcoord + float2(0,1) );
   c = f4texRECT ( img, texcoord + float2(1,0) );
   d = f4texRECT ( img, texcoord + float2(1,1) );
   return a+b+c+d;
}
```

Algoritmo 4-3. Pixel shaders para la conversión de datos en la primera fase y para la sumatoria del framebuffer en la segunda fase.

dibujado y cambiando el tamaño de los buffers implicados. Es mejor mantener las operaciones en 2x2 para usar el máximo de cauces* (*pipes*) disponibles.

Hay que destacar que se utiliza un formato de textura de flotantes de 128 bits, esto es, 4 números de 32 bits cada uno que permitirán almacenar en cada texel, y por tanto para cada tetraedro, el estado de inclusión de 4 puntos. Las texturas utilizadas deben tener el mismo formato. La sumatoria de la matriz de datos resultado de la primera fase del algoritmo se realiza en $O(\log n)$ pasadas para n^2 elementos y una suma parcial de 4x4 pixels en cada ejecución del pixel shader.

4.3.3.3 Rendimiento

El algoritmo implementado en GPU presenta un coste lineal respecto al número de triángulos de la malla, al igual que la versión original del algoritmo. Sin embargo, como puede comprobarse en la Sección 4.4.3, la eficiencia del algoritmo en la GPU es mayor que en la CPU para la mayoría de los casos. Hay que tener en cuenta que en la versión básica en CPU, salvo optimizaciones adicionales, no necesita preparar ninguna estructura adicional, mientras que para la GPU hay que inicializar las display lists y preparar los buffers oportunos. Sin embargo, el procesamiento paralelo en la GPU tiende a compensar todos estos factores. Además, hay que tener en cuenta que mientras la GPU está ocupada con el algoritmo de inclusión, la CPU puede dedicarse a

* *A lo largo del texto se emplea el término **pipeline** (pipeline gráfico) para designar el conjunto de etapas del proceso de rendering. Sin embargo, se utiliza el término **cauce** en lugar de **pipe** para designar cada unidad de computación en un procesador segmentado debido a que es algo más natural.*

otras tareas. También hay que tener en cuenta que muchos optimizadores de la versión CPU también sirven para la versión GPU.

Según parece, la evolución del hardware gráfico traerá consigo una serie de mejoras y prestaciones que facilitarán el desarrollo de algoritmos de propósito más general que los actuales. Entre las posibles extensiones que son previsibles de ser implementadas, las presentadas a continuación beneficiarían al algoritmo de inclusión de puntos en varios aspectos:

- La posibilidad de interrumpir el pipeline de rendering justo después de ejecutar el vertex shader y recuperar los datos transformados en una matriz. Esto permitiría utilizar la GPU directamente como procesador vectorial, lo que se ajustaría muy bien a la filosofía de este algoritmo. Además, el formato de los datos de salida serían utilizables de forma directa y sin conversión alguna (ahora mismo el vertex shader sólo devuelve números en [0,1]). Esta nueva capacidad está relacionada con la extensión Render-To-Vertex-Array [Lefohn04].

- Si se pudiera leer desde el framebuffer a nivel de pixel shader se podrían realizar operaciones matemáticas de acumulación mucho más versátiles que las actuales, que se reducen a un blending y a operaciones lógicas a nivel raster. Esta opción permitiría, al realizar las acumulaciones sobre el framebuffer, mover todo el algoritmo a nivel de pixel shader, que según la GPU utilizada, podría resultar más rápido que la versión del vertex shader.

- La opción de tener registros acumuladores durante la fase del vertex shader permitiría sumar los estados de inclusión de cada tetraedro y recuperar el resultado final de un acumulador de la GPU. Esta opción dejaría reducido el algoritmo al mínimo y sería, sin duda, la más eficiente.

Las transferencias de GPU a memoria principal suponen un gran problema, ya que interrumpen el pipeline y no son demasiado eficientes. Lo primero no parece que tenga una solución fácil, pero las transferencias sí mejorarán en el futuro. El bus AGP utilizado en la actualidad está siendo reemplazado por el PCI-Express, que ofrece mayores prestaciones, en especial porque es simétrico (la velocidad de transferencia de CPU a GPU es igual que de GPU a CPU).

4.3.4 Implementación distribuida y paralela

Como se ha visto en la Sección 4.3.3, el algoritmo de inclusión puede adaptarse perfectamente al procesamiento vectorial, lo que sugiere que puede ser paralelizado en su forma más básica. Además del uso de la GPU para realizar el proceso, existen otros enfoques que pueden ayudar a construir una solución más eficiente aprovechando múltiples unidades de computación, es decir, CPUs y GPUs agrupadas de varias formas. En este Apartado se presentan algunas ideas de cómo conseguir un procesamiento distribuido y paralelo del algoritmo.

Recordemos que el Algoritmo 4-1 trabaja sobre la descomposición del sólido en tetraedros y el cálculo de la inclusión del punto a probar en cada uno de los mismos. De esta forma, el cálculo de la inclusión del punto en un tetraedro es independiente del cálculo de la inclusión en el resto de tetraedros. Solamente están vinculados en la suma final, que determinará el estado de inclusión del punto en el sólido. Consecuentemente, puede distribuirse el trabajo si el conjunto de tetraedros se reparte entre varias unidades de computación, para finalmente reunir las sumas parciales en el resultado final. Hay que tener en cuenta que al distribuir el procesamiento para el test de un sólo punto, pueden aparecer problemas con el tratamiento de los casos especiales, ya que se necesita de estructuras de datos adicionales que habría que sincronizar.

Siguiendo este enfoque y suponiendo distintas situaciones, a continuación se explican algunos enfoques de multiprocesamiento. La Tabla 4-2 presenta una calificación orientativa de algunas de las soluciones posibles teniendo en cuenta los factores más relevantes.

4.3.4.1 Multiprocesamiento

Las capacidades de procesamiento paralelo que poseen las CPUs y GPUs actuales funcionan de forma transparente al software implementado. De esta forma, si la CPU contiene múltiples cauces segmentados (en pipeline), o la GPU utiliza procesamiento vectorial, el paralelismo se realiza de forma natural. Lo que se presenta aquí supone la adaptación del software para controlar el reparto de los cálculos entre varias unidades de procesamiento.

En primer lugar, si se dispone de más de una CPU en el sistema, puede conseguirse un aumento proporcional del rendimiento utilizando varias tareas simultáneas. Para el test de inclusión de un sólo punto en un sólido, cada tarea o proceso debe encargarse del cálculo de la inclusión del punto en un subconjunto de tetraedros del recubrimiento simplicial. Cuando todas las tareas terminan se hace una recopilación de las sumas parciales y se obtiene el resultado final. En el caso de que el

test de inclusión se realizase sobre más de un punto, la distribución del trabajo puede realizarse a nivel de test completo de cada punto, por lo que cada proceso realiza las pruebas de un conjunto de puntos a probar sobre el sólido. Hay que destacar que con los sistemas multiprocesador, este algoritmo puede repartir el trabajo de forma natural y eficiente, por lo que la distribución supone un coste ínfimo.

Esta misma filosofía también es aplicable a las GPUs, esto es, un sistema con varias GPUs puede repartir el trabajo de la misma forma. En la actualidad existen clusters de GPUs que funcionan como una sola, lo que también proporciona algunas posibilidades de uso.

4.3.4.2 Clustering

La estrategia de clustering consiste en disponer de un conjunto de ordenadores conectados en red, de forma que pueden ejecutarse distintos procesos de forma sincronizada. Para el test de inclusión, la distribución de la carga de trabajo puede hacerse a nivel de test de un sólo punto o a nivel de conjunto de puntos, lo que determina el tipo de información a repartir. La ventaja de paralelizar a nivel de test de un sólo punto es que sólo se distribuye parte del recubrimiento simplicial del sólido. Sin embargo, cuando son muchos los puntos a probar, es mejor distribuir los tests completos, teniendo cada nodo que almacenar el sólido completo.

La estrategia de clustering se adapta muy bien a la optimización de segmentación espacial. En este caso, cada nodo del cluster se encarga de una parte del espacio, por lo que sólo almacena los tetraedros que *pasan* por esa región. Esta forma de trabajo es muy adecuada para la distribución a nivel de conjunto de puntos, y es rentable sólo cuando dicho conjunto es grande, lo que compensa la latencia de la red y el coste del reparto del trabajo.

4.3.4.3 Soluciones mixtas

Dentro de las soluciones mixtas posibles, las siguientes figuran entre las más interesantes:

- Reparto del trabajo entre CPU y GPU. Es más apropiado repartir los puntos a probar que repartir datos para un único test de inclusión, ya que las estructuras de datos necesarias para los casos especiales son más complicadas de sincronizar entre CPU y GPU.

- Cluster de ordenadores multiprocesadores. Cada ordenador trabaja con un conjunto de puntos para probar su inclusión. El reparto interno entre las

	MultiCPU y/o multiGPU	Clustering uniforme	Clustering híbrido	Clustering multiCPU y/o multiGPU
Test 1 punto	Muy adecuado	Adecuado	Adecuado	Adecuado
Test n puntos	Adecuado	Muy adecuado	Muy adecuado	Muy adecuado
Mallas pequeñas	Muy adecuado	No adecuado	No adecuado	No adecuado
Mallas grandes	Adecuado	Adecuado	Adecuado	Muy adecuado
Adaptación a segmentación espacial	Buena	Excelente	Excelente	Excelente
Coste de implantación	Medio	Medio	Muy bajo	Muy alto
Potencia de cálculo efectiva	Media	Alta	Media/Alta	Excelente

Tabla 4-2. Algunas características de las estrategias de procesamiento distribuido y paralelo aplicadas al algoritmo de inclusión en comparación con un sistema monoprocesador.

CPUs de cada ordenador es variable. Puede realizarse repartiendo puntos a probar entre los procesos o realizar cada test de forma simultanea repartiendo los tetraedros de los recubrimientos del sólido.

- Cluster de ordenadores multiprocesadores multiGPUs. Cada ordenador dispone de 2 ó mas CPUs y 2 ó más GPUs. La filosofía del reparto de trabajo es igual que la anterior, pero las combinaciones y la potencia de cálculo efectiva son mayores.

- Cluster híbrido. El sistema de reparto ideal es el que distribuye un conjunto de puntos y el sólido a probar en una red, y varios nodos se encargan de realizar los cálculos. Los ordenadores pueden tener cualquier configuración de CPU y GPU, simples o múltiples, así como de sistema operativo. Es la forma más realista y adaptable a la mayoría de las situaciones. Debe prestarse atención al comportamiento del algoritmo implementado en cada sistema, esto es, bajo las mismas condiciones y en distintos sistemas, los mismos parámetros de entrada deben producir exactamente los mismos resultados.

4.4 Comparación de soluciones

Es esta Sección se presentan los resultados obtenidos, así como un estudio comparativo con algunas de las técnicas más populares aplicadas a mallas de polígonos [Ogayar05]. En el apéndice B se describen los detalles técnicos sobre la implementación de todos los algoritmos presentados en este trabajo, en lo que se refiere al hardware, el lenguaje de programación, etc. Tan sólo hay que señalar que todos los algoritmos se han implementado bajo las mismas condiciones y de la forma más eficiente posible.

4.4.1 Implementación

En la Tabla 4-3 se muestran algunas características de las soluciones presentadas, que se encuentran entre las más utilizadas. Sólo se reflejan las propiedades teóricas de los algoritmos, como los órdenes de complejidad, ya que como se verá más adelante, las implementaciones dependen mucho del hardware utilizado y de las optimizaciones adicionales.

Los algoritmos que se han implementado y probado son Jordan, Jordan optimizado con octree, Feito-Torres, Feito-Torres optimizado, Feito-Torres en GPU y un algoritmo de inclusión basado en BSP. No se presentan resultados utilizando algoritmos basados en clasificadores espaciales (espacio de voxels y octree), ya que la inclusión de puntos no es exacta y es más interesante estudiar los métodos precisos. Las implementaciones se han optimizado en la medida de lo posible, ya que es interesante

	Grid	Octree	Polygon Aligned BSP	Jordan	Feito-Torres
Representación del sólido	Aproximada	Aproximada	Exacta	Exacta	Exacta
Complejidad de inclusión	O(k)	O(log n)	O(log n)	O(n)	O(n)
Eficiencia	Muy alta	Alta	Alta	Normal	Normal
Estructura espacial necesaria	Muy costosa	Costosa	Muy costosa	Ninguna	Ninguna
Adaptación a inclusión	Baja	Baja	Alta	Alta	Alta

Tabla 4-3. Características de los algoritmos de inclusión presentados.

contar con la mejor de las versiones; sirva como ejemplo que no se ha usado recursividad en la construcción del BSP. En cuanto a las versiones GPU, se ha utilizado la tecnología en procesamiento gráfico 3D programable más actual.

4.4.1.1 Algoritmo de inclusión basado en BSP

La implementación del BSP es bastante convencional. Para ir subdividiendo el espacio se toman los planos donde están contenidas las caras del sólido. Si un plano divisor corta algún polígono, el plano que contiene al mismo es considerado en los dos subespacios resultantes; esto produce un árbol de gran tamaño con modelos muy complejos, especialmente los que cuentan con muchas concavidades.

La versión del algoritmo de creación del BSP elimina toda recursividad mediante el uso de una pila cuyos elementos almacenan el estado de la ramificación en todo momento. Esta versión iterativa es mucho más eficiente en tiempo y en el uso de la memoria; la versión recursiva implementada en primer lugar agotaba rápidamente la memoria de la máquina. Asimismo, la navegación por el BSP también se ha implementado de forma iterativa.

4.4.1.2 Algoritmo de inclusión basado en el Teorema de la Curva de Jordan

Para aplicar el teorema de la curva de Jordan se ha implementado como algoritmo de intersección rayo-triángulo el de Möller-Trumbore [Möller97], modificado para detectar con un error determinado cuándo se produce una intersección con un vértice o una arista, y cuándo el rayo es coplanar al triángulo a intersectar. Esta variación introduce una pequeña penalización en el rendimiento, pero es necesaria para proporcionar robustez al algoritmo. Otra posible solución consiste en mover infinitesimalmente los vértices implicados en una intersección conflictiva, pero no garantiza que no vuelvan a producirse otros casos especiales tras la corrección. Cada vez que se produce un caso especial, se repite el proceso utilizando un vector aleatorio distinto. Este es, sin duda, el punto débil del algoritmo, que obliga a utilizar una gran precisión decimal.

El cuello de botella en el algoritmo de Jordan es la intersección rayo-triángulo. Para mejorar el rendimiento se ha utilizado en la versión mejorada un octree como indexador espacial de polígonos (no confundir con el esquema de representación de sólidos basado en octree). Partiendo del mínimo cubo envolvente, se generan octantes que se subdividen como máximo hasta una profundidad determinada, de forma que para cada nodo hoja hay asociada una lista de polígonos que lo intersectan. Esta es la variante del octree que propone [Glassner89] para ser utilizada como optimizador en el ray-casting.

Se ha realizado una mejora que consiste en eliminar octantes vacíos durante la construcción del octree, esto es, al subdividir un nodo, sólo se crean subnodos cuando éstos contengan parcial o totalmente algún polígono. Esto produce en ahorro promedio en memoria del 50% en la estructura de datos (no se cuentan los datos vinculados, como polígonos) comparado con el octree convencional, donde si se subdivide un octante, siempre se hace en 8 nodos.

Para recorrer el octree (traversing), se ha utilizado el algoritmo de Gargantini [Gargantini93], optimizado para visitar sólo el subconjunto de nodos que interesan, esto es, los que quedan por delante del punto siguiendo el vector de intersección y además son nodos hoja (los que contienen las listas de polígonos). Utilizando el octree y el algoritmo de recorrido paramétrico, la lista de triángulos que deben intersectarse en cada prueba de inclusión se reduce de forma drástica, dependiendo de la profundidad del octree. Con un algoritmo eficiente de recorrido de octrees, no compensa el uso de un grid optimizador frente al uso de un octree, ya que éste ocupa mucha menos memoria.

4.4.1.3 Algoritmo de inclusión basado en recubrimientos simpliciales

La implementación básica del algoritmo de Feito-Torres (ver Apartado 4.3) utiliza tan sólo una función auxiliar para comprobar la inclusión de un punto en un tetraedro original, y la función principal para comprobar la suma de los volúmenes de los tetraedros originales que incluyen el punto de prueba.

Como optimizaciones se han implementado el preprocesamiento básico (ver Apartado 4.3.2.1), la clasificación por octantes (ver Apartado 4.3.2.3) y la indexación mediante segmentación uniforme (ver Apartado 4.3.2.4). El preprocesamiento básico junto con alguno de los clasificadores proporcionan una buena combinación para alcanzar un rendimiento adecuado. La implementación en GPU, totalmente compatible con las anteriores, se ha implementado según se describe en la Sección 4.3.3. Hay que destacar que los optimizadores espaciales aplicados a la GPU funcionan con grupos de display lists por cada nodo, ya sea la clasificación por octantes o la segmentación uniforme, en lugar de hacer los descartes de inclusión a nivel de tetraedro. Además, debe tenerse en cuenta los límites de la caché de la GPU a la hora de construir cualquier display list. En el Apartado A.7.8 se describen los problemas que pueden presentarse y la forma de solucionarlos.

4.4.2 Pruebas

Para probar el algoritmo de inclusión con las distintas variantes y las opciones más interesantes, se han realizado varias pruebas con mallas de triángulos que presentan distintas características topológicas, cubriendo un rango razonable en cuanto a la complejidad poligonal. Se han utilizado sólidos con distintas formas, incluyendo todo tipo de concavidades y agujeros para resaltar los puntos débiles de los métodos de inclusión. Las láminas 1 a 9 (páginas centrales del libro) muestran imágenes de los sólidos utilizados. En el apéndice B se exponen las características del entorno software y el hardware utilizado para todas las pruebas de este trabajo. Para las pruebas de inclusión se han utilizado diez conjuntos de mil puntos arbitrarios situados dentro de la caja envolvente de cada sólido, obteniendo la media del rendimiento de los diez conjuntos.

4.4.3 Resultados

En primer lugar se presenta un estudio detallado del algoritmo de inclusión basado en recubrimientos simpliciales de Feito-Torres, incluyendo los optimizadores implementados, y cómo influyen en el algoritmo básico. Posteriormente se procede a la comparación con el resto de métodos. Antes de presentar los detalles sobre los resultados del algoritmo de inclusión se presenta un estudio detallado de los optimizadores del algoritmo de Feito-Torres, para después comparar las mejores versiones con el resto de métodos.

4.4.3.1 Algoritmo de Feito-Torres

El algoritmo de Feito-Torres admite varios optimizadores. En este Apartado se presentan los resultados del estudio de tiempos realizado para comprobar el impacto sobre el rendimiento final. Posteriormente se realizará la comparación con el resto de métodos. En la Tabla 4-4 se presentan los resultados de la construcción de un grid de tetraedros (también denominado TetraGrid) sobre cada uno de los modelos. Se indica el tiempo de construcción de la estructura así como la memoria ocupada. Como es lógico, a mayor resolución, mayor espacio en memoria. En la Figura 4-5 pueden verse las gráficas que ilustran los datos de la Tabla 4-4. El orden de complejidad de la construcción del grid de tetraedros está en $O(n^3)$, siendo n el ancho en voxels del grid, que forma un cubo de $(n \cdot n \cdot n)$ ajustado al cubo envolvente del sólido. Como puede verse en la Figura 4-5, la tendencia del algoritmo no siempre es uniforme, esto es, no siempre se ajusta a su orden de complejidad teórico.

			2^3		4^3		8^3	
	Vértices	*Triángulos*	*Tiempo*	*Kb*	*Tiempo*	*Kb*	*Tiempo*	*Kb*
Torusknot	808	1200	0,006335	8	0,008942	31	0,010332	63
Celtic Cross	1849	2366	0,014665	13	0,025891	54	0,025802	94
Vertebra	5228	10444	0,050873	75	0,074298	259	0,102772	429
Lion Head	13025	25946	0,100843	164	0,150893	534	0,180701	786
Golf Ball	23370	46205	0,229884	320	0,320815	1146	0,390914	2117
Armadillo	75002	150000	0,862091	835	1,195123	3206	1,505248	5957
Venus Sculpture	139217	277512	1,467139	1935	2,612462	6947	2,986706	9680
Happy Buddha	250007	500000	3,273651	3617	6,354165	12565	7,499158	17828
Chinese Dragon	437645	871414	7,209098	6477	14,29574	20111	19,64374	29478

			16^3		32^3		64^3	
	Vértices	*Triángulos*	*Tiempo*	*Kb*	*Tiempo*	*Kb*	*Tiempo*	*Kb*
Torusknot	808	1200	0,013407	192	0,032663	953	0,138214	6243
Celtic Cross	1849	2366	0,029454	219	0,036384	950	0,078432	5979
Vertebra	5228	10444	0,111165	850	0,207525	2418	0,492252	10258
Lion Head	13025	25946	0,210526	1395	0,314832	3356	0,718221	12080
Golf Ball	23370	46205	0,563552	4510	0,988746	11912	3,505607	39253
Armadillo	75002	150000	2,186012	12040	4,490355	29877	9,225538	92107
Venus Sculpture	139217	277512	3,677937	17038	5,147866	34708	8,801946	84953
Happy Buddha	250007	500000	9,296401	30371	13,60881	61362	23,297249	148665
Chinese Dragon	437645	871414	23,43022	48002	30,31309	89132	47,165318	198338

Tabla 4-4. Resultados de la construcción de un Grid de tetraedros con distintas resoluciones. El tiempo se indica en segundos y la memoria consumida en Kb.

		Sin optimizador	TetraGrid					
	Caras		2^3	4^3	8^3	16^3	32^3	64^3
Torusknot	1200	0,109153	0,027244	0,013119	0,004797	0,002314	0,001706	0,001527
Celtic Cross	2366	0,236108	0,048675	0,075668	0,029782	0,007813	0,003599	0,002072
Vertebra	10444	1,092031	0,253556	0,213965	0,045291	0,011001	0,004201	0,002948
Lion Head	25946	2,092121	0,507886	0,426291	0,110962	0,021413	0,007274	0,004087
Golf Ball	46205	4,532704	1,107139	0,487276	0,138828	0,030481	0,015856	0,007371
Armadillo	150000	20,785783	3,950333	3,002252	0,846587	0,210811	0,052549	0,027987
Venus Sculpture	277512	31,779722	12,346745	7,681687	6,128579	1,831241	0,280872	0,097908
Happy Buddha	500000	72,439564	25,290156	17,380956	6,863645	2,096489	0,410256	0,119525
Chinese Dragon	871414	98,630565	22,768432	19,213679	5,219548	0,754569	0,240589	0,087675

Tabla 4-5. Tiempos en segundos del algoritmo de Feito-Torres con y sin TetraGrid.

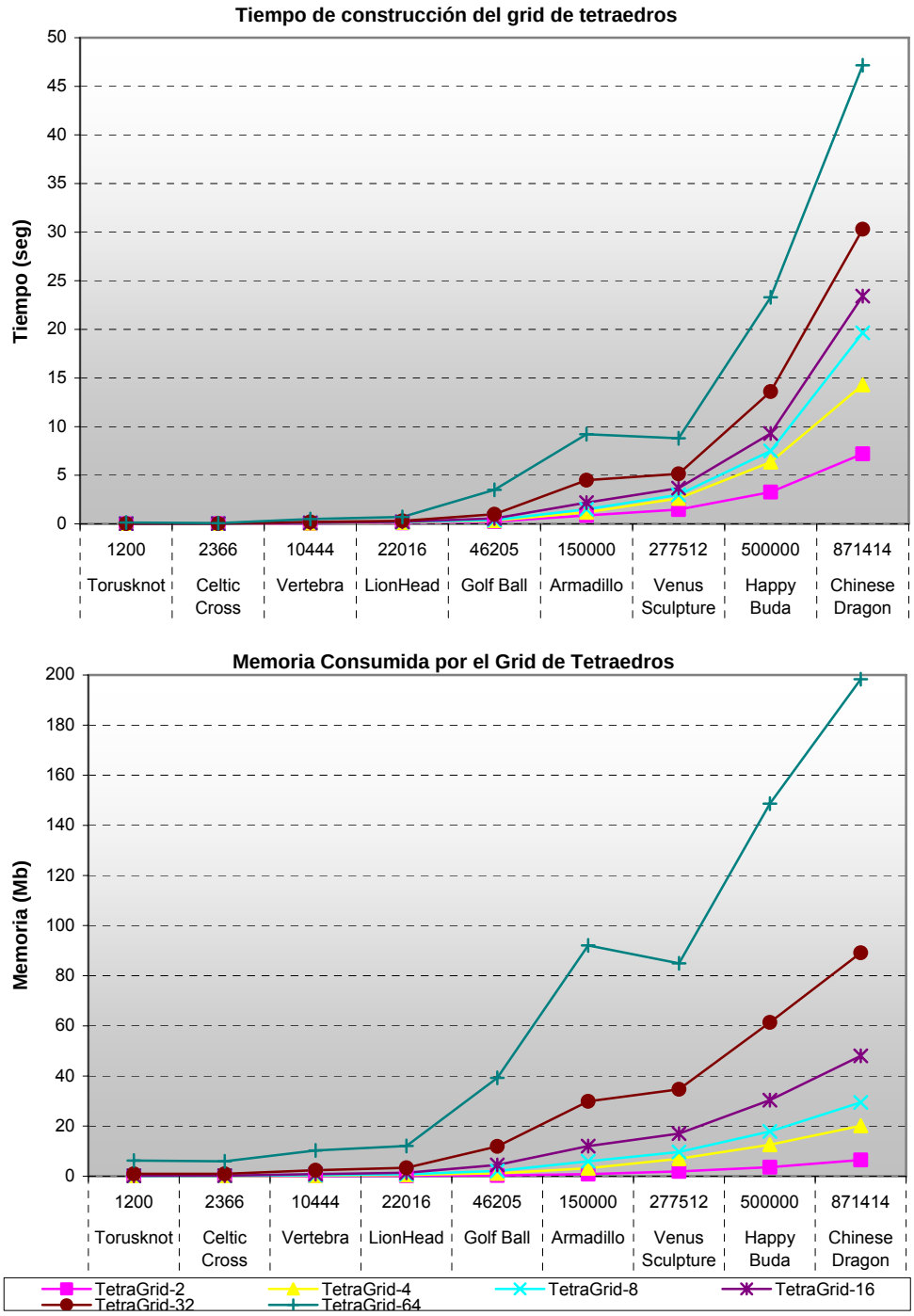


Figura 4-5. Tiempo y memoria consumidos por el grid de tetraedros.

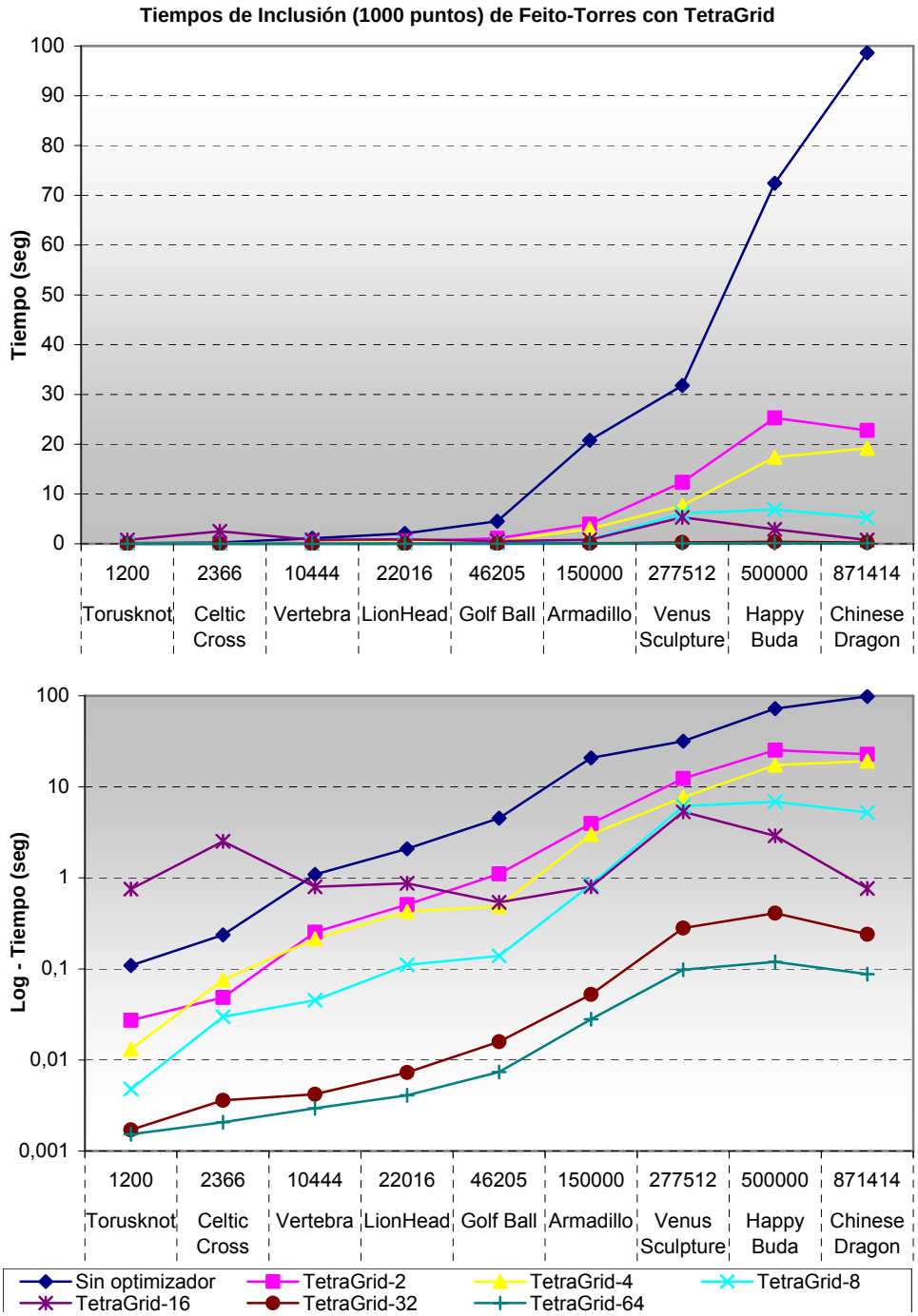


Figura 4-6. Tiempos de inclusión de Feito-Torres con y sin Grid de Tetraedros.

Lo anterior se hace notable en los casos *Armadillo* y *Venus Sculpture*, que presentan tiempos superiores e inferiores a lo esperado, respectivamente cuando la resolución del grid aumenta. Lo mismo ocurre con la memoria asociada a estos casos que, como puede verse en la Figura 4-5, también rompen la tendencia. Esto se debe a la forma de los sólidos y a su disposición en su mínimo cubo envolvente, que determina la distribución de los voxels del grid respecto del objeto. Uno de los problemas que aumentan el uso de memoria y el tiempo de proceso es el hecho de que el mismo tetraedro aparezca en muchos voxels del grid, lo que ocurre cuando la resolución es relativamente alta o los tetraedros muy grandes. El Apartado 5.5 describe la construcción de esta estructura y sus problemas asociados.

La Tabla 4-5 muestra los resultados de la ejecución del algoritmo de Feito-Torres en su versión básica comparado con la versión que utiliza el grid de tetraedros con distintas resoluciones. La Figura 4-6 muestra las gráficas de tiempos. Como puede comprobarse, el efecto del grid sobre el algoritmo está en función de la resolución utilizada, por lo que debe buscarse un compromiso entre rendimiento y memoria utilizada. En cualquier caso se ha observado que a partir de 64^3 el beneficio en el rendimiento es inapreciable; sin embargo, el gasto de memoria empieza a ser excesivo para casi cualquier sólido. En los estudios comparativos se ha determinado que una resolución de 32^3 proporciona un gran incremento en el rendimiento sin abusar demasiado de la utilización de la memoria en un sistema moderno.

La Tabla 4-6 muestra los tiempos empleados en ejecutar algunas de las mezclas más interesantes con optimizadores. Se recoge la versión básica, las implementaciones en GPU y algunas combinaciones de éstas con los optimizadores descritos en la Sección 4.3.2. La Figura 4-8 muestra las gráficas que ilustran las diferencias.

La Figura 4-7 ofrece una comparación entre la versión básica CPU del algoritmo, la implementación mixta CPU+GPU y la completa en GPU descritas en la Sección 4.3.3. Hay que recordar que la versión mixta realiza la segunda parte del algoritmo descrita en la Sección 4.3.3.2 por software para poder tratar los casos especiales de forma adecuada. Como puede observarse, los resultados en GPU son mucho más eficientes que en CPU, y esta diferencia aumenta con la complejidad del sólido. Esto es debido a que en los casos más sencillos la preparación de las estructuras para el algoritmo en GPU puede hacer que el proceso total no compense. A esto hay que sumarle las transferencias entre CPU y GPU que restan rendimiento. Sólo a partir de cierto número de tetraedros, la implementación en GPU se hace rentable. En cualquier caso, los resultados siempre dependerán de la potencia de la CPU y de la GPU.

	Caras	Básico	CPU+GPU	GPU	Octantes	TetraGrid 32	GPU+ TetraGrid-32
Torusknot	1200	0,027244	0,013119	0,004797	0,002314	0,001706	0,001527
Celtic Cross	2366	0,048675	0,075668	0,029782	0,007813	0,003599	0,002072
Vertebra	10444	0,253556	0,213965	0,045291	0,011001	0,004201	0,002948
Lion Head	25946	0,507886	0,426291	0,110962	0,021413	0,007274	0,004087
Golf Ball	46205	1,107139	0,487276	0,138828	0,030481	0,015856	0,007371
Armadillo	150000	3,950333	3,002252	0,846587	0,210811	0,052549	0,027987
Venus Sculpture	277512	12,346745	7,681687	6,128579	1,831241	0,280872	0,097908
Happy Buddha	500000	25,290156	17,380956	6,863645	2,096489	0,410256	0,119525
Chinese Dragon	871414	22,768432	19,213679	5,219548	0,754569	0,240589	0,087675

Tabla 4-6. Tiempos resultantes de ejecutar diversas versiones del algoritmo de inclusión de Feito-Torres. El tiempo se indica en segundos.

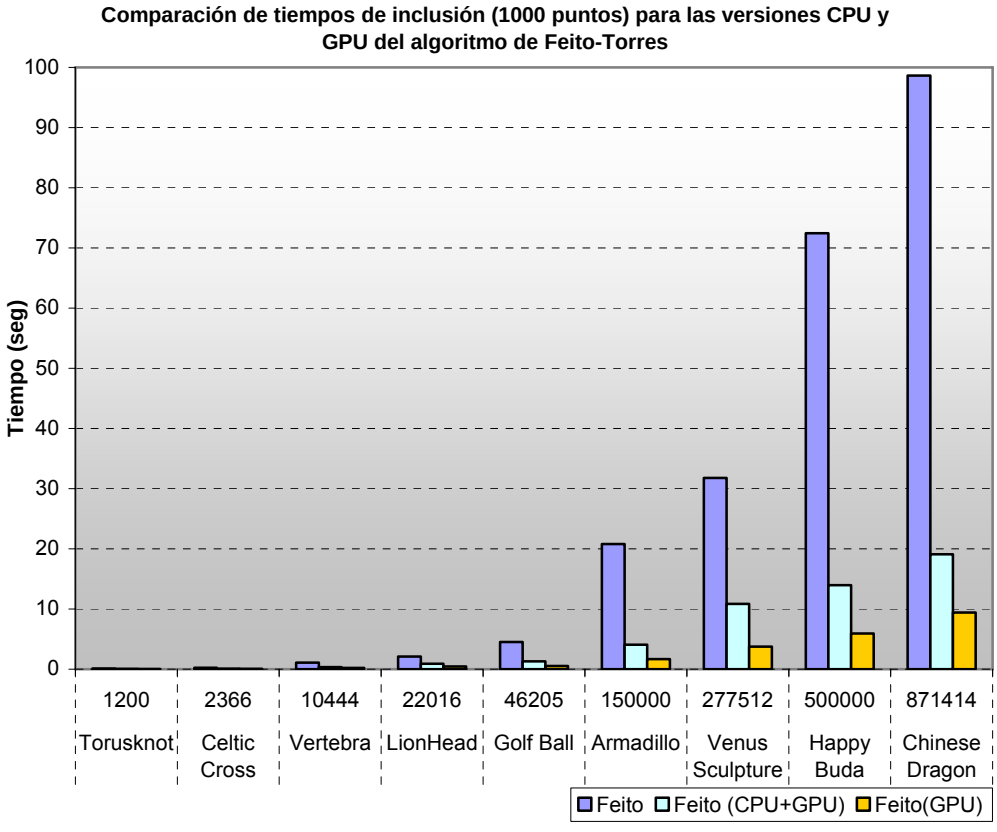


Figura 4-7. Comparativa entre la versiones CPU, CPU+GPU y GPU de Feito-Torres.

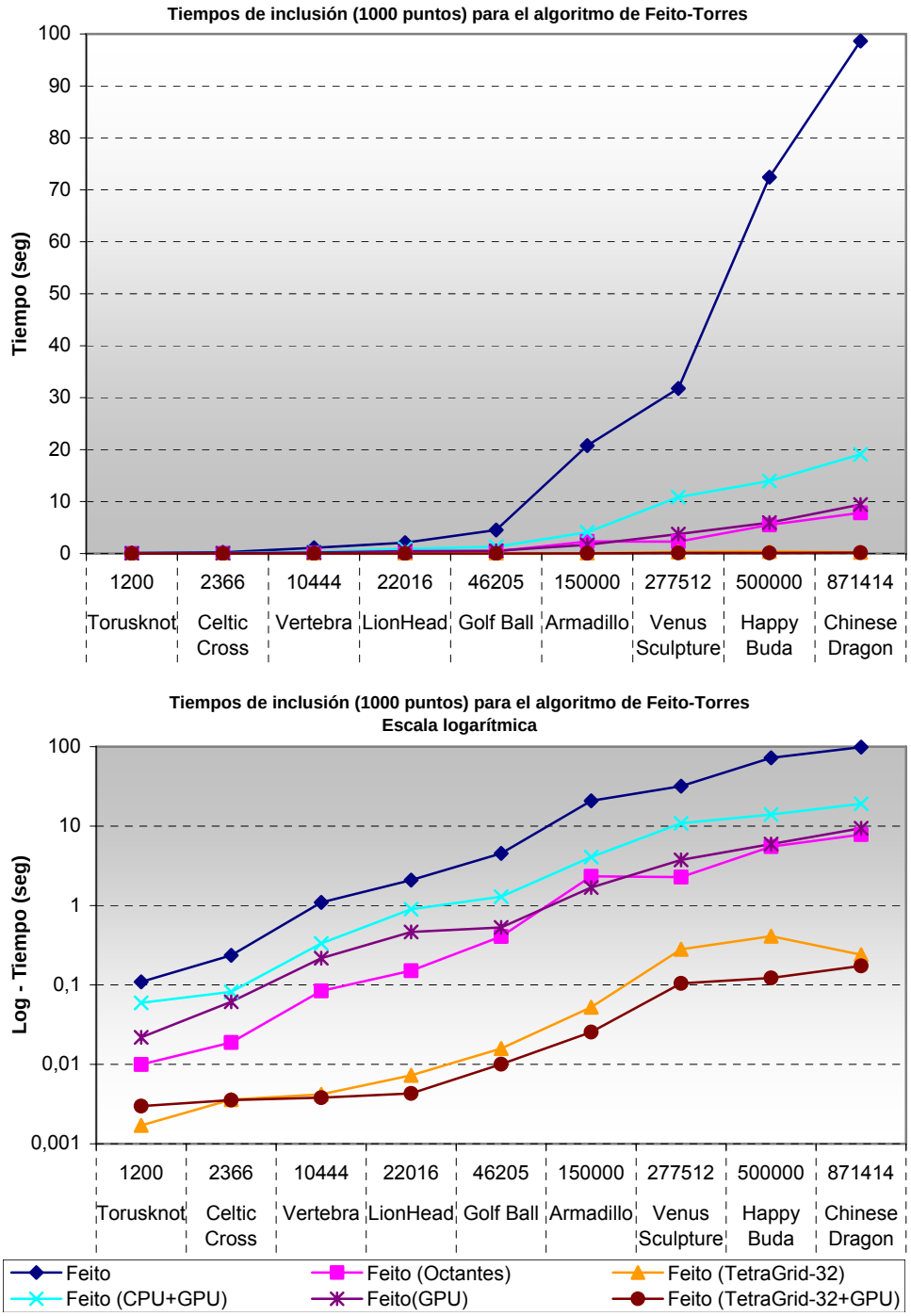


Figura 4-8. Comparativa entre varias soluciones basadas en Feito-Torres.

4.4.3.2 Comparación de soluciones

La Tabla 4-7 y la Tabla 4-8 presentan una comparativa de los recursos consumidos por los algoritmos que precisan de estructuras de datos, tanto en tiempo de procesamiento como en memoria. El algoritmo basado en BSP es el más costoso en preprocesamiento. Con los modelos más complejos presenta dificultades con el sistema de prueba, agotando los recursos con el objeto *Chinese Dragon*. El optimizador de octantes de Feito-Torres es el menos costoso, lo que lo hace una opción muy interesante, ya que, como se aprecia en la Tabla 4-9, la ganancia en rendimiento es muy buena teniendo en cuenta los pocos recursos que necesita. La Figura 4-9 muestra las gráficas de tiempo y memoria utilizados por los métodos que necesitan preprocesamiento.

La Tabla 4-9 y la Figura 4-10 muestran los datos de las pruebas de inclusión para una selección de métodos. Estos datos sirven para realizar la comparación final en rendimiento. Incluye las versiones más eficientes de Feito-Torres, el enfoque basado en BSP y dos versiones basadas en el teorema de Jordan, una sin optimizador y otra con un octree de profundidad 6.

El BSP es, sin duda, el algoritmo más rápido de clasificación espacial. Su inconveniente es el coste extremo en preprocesamiento cuando se trata con modelos complejos, tanto en tiempo de cálculo como en memoria (algunos tests colapsaron el sistema, incluso con una implementación iterativa en lugar de recursiva). El problema principal aparece cuando el plano asociado a un triángulo de la malla corta a otros polígonos, en cuyo caso hay que añadir sus respectivos planos a los dos subespacios resultantes. Esto ocasiona un crecimiento desmesurado del árbol BSP, lo que suele ocurrir cuando se presentan concavidades y agujeros en el modelo. El algoritmo basado en BSP es muy dependiente de la forma del sólido.

El algoritmo de Jordan se muestra menos eficiente cuando hay una gran densidad de vértices y polígonos, y muy especialmente cuando se presentan concavidades. En situaciones como esta, hay que repetir los cálculos de intersección variando el vector de salida del sólido hasta que deje de coincidir con vértices y aristas. Aunque este problema se reduce aumentando la precisión decimal (y reduciendo el valor de epsilon), sigue siendo patente con mallas complejas. La versión optimizada con octree mejora enormemente el rendimiento, si bien necesita un tiempo de preprocesamiento moderado y cierta cantidad de memoria. En este caso, los tiempos son sensibles al reparto de los polígonos entre los octantes del octree, que a veces puede no ser tan eficiente.

	Triángulos	Feito (Octantes)	Feito (TetraGrid-32)	BSP	Jordan (Octree-6)
Torusknot	1200	2	953	784	538
Celtic Cross	2366	3	950	399	131
Vertebra	10444	10	2418	6799	519
Lion Head	25946	22	3356	15000	216
Golf Ball	46205	46	11912	3048	842
Armadillo	150000	150	29877	48000	1047
Venus Sculpture	277512	277	34708	328000	1374
Happy Buddha	500000	500	61362	1062000	2491
Chinese Dragon	871414	871	89132	Stack overflow	4159

Tabla 4-7. Memoria en Kb utilizada por cada método de inclusión.

	Triángulos	Feito (Octantes)	Feito (TetraGrid-32)	BSP	Jordan (Octree-6)
Torusknot	1200	0,000089	0,032663	0,006995	0,061586
Celtic Cross	2366	0,000175	0,036384	0,097916	0,041236
Vertebra	10444	0,000798	0,207525	0,778738	0,123942
Lion Head	25946	0,001499	0,314832	1,906512	0,176692
Golf Ball	46205	0,003911	0,988746	53,411813	0,328452
Armadillo	150000	0,020354	4,490355	16,664282	1,140653
Venus Sculpture	277512	0,023011	5,147866	180,725821	1,609565
Happy Buddha	500000	0,056132	13,608813	1019,281251	4,062355
Chinese Dragon	871414	0,083593	30,313092	Stack overflow	7,496312

Tabla 4-8. Tiempos en segundos para la construcción de las estructuras necesarias para cada método de inclusión.

		Feito			BSP	Jordan	
	Caras	CPU	GPU	GPU+ TetraGrid-32		Normal	Octree-6
Torusknot	1200	0,109153	0,021842	0,002981	0,000001	0,187341	0,021076
Celtic Cross	2366	0,236108	0,061445	0,003556	0,000001	0,371509	0,022737
Vertebra	10444	1,092031	0,216991	0,003804	0,000006	1,709202	0,055384
Lion Head	25946	2,092572	0,465027	0,004301	0,000039	5,739875	0,056385
Golf Ball	46205	4,532704	0,530141	0,010058	0,000096	7,325025	0,058105
Armadillo	150000	20,785783	1,684518	0,025461	0,000043	28,039392	0,181457
Venus Sculpture	277512	31,779722	3,755179	0,104579	0,000016	47,419029	0,732306
Happy Buddha	500000	72,439564	5,937801	0,122512	0,000089	95,696929	1,050641
Chinese Dragon	871414	98,630565	9,417552	0,173864	--	144,543591	1,196324

Tabla 4-9. Tiempos en segundos de los tests de inclusión de 1000 puntos en los distintos sólidos.

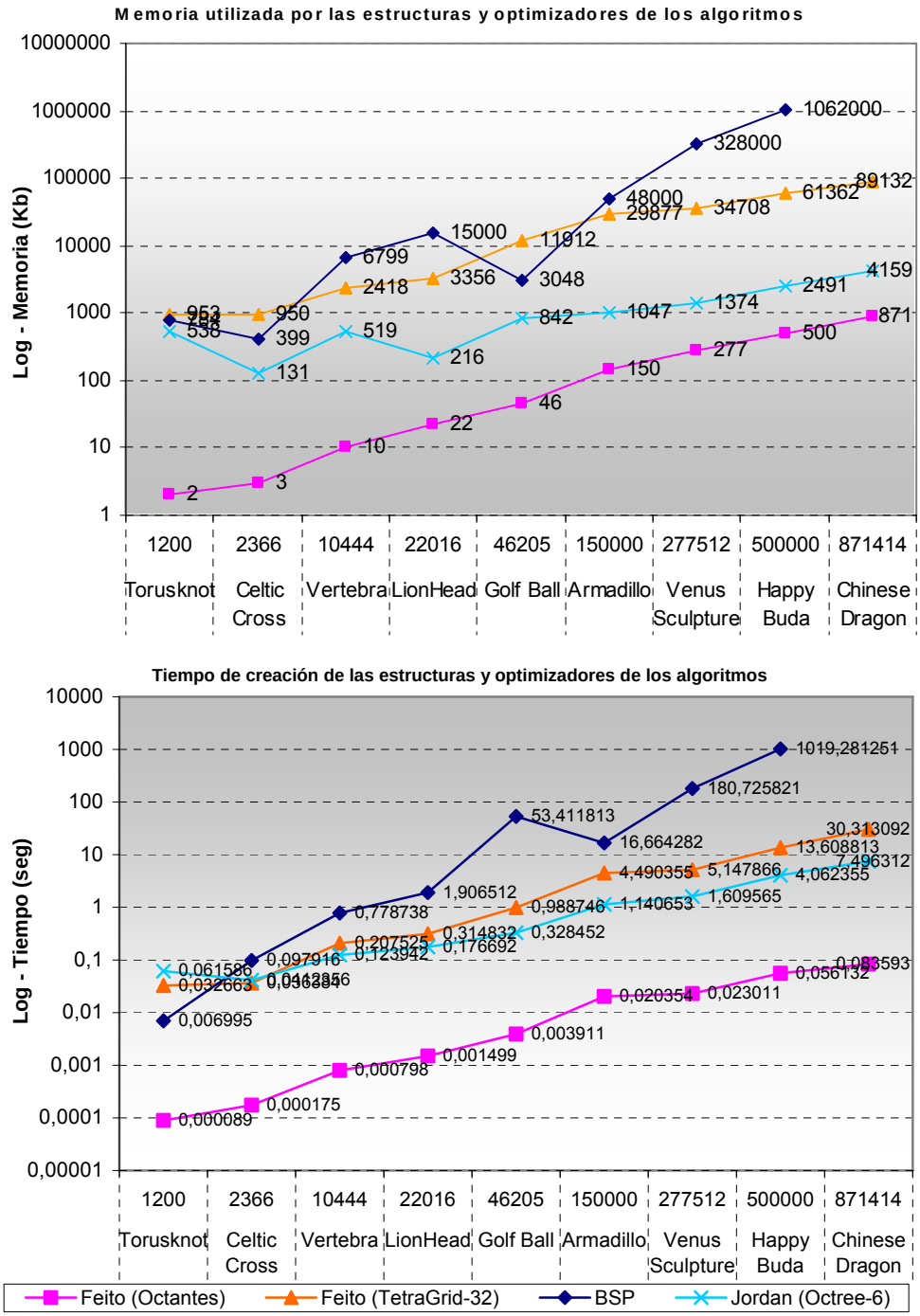


Figura 4-9. Recursos empleados en el preprocesamiento de cada método de inclusión.

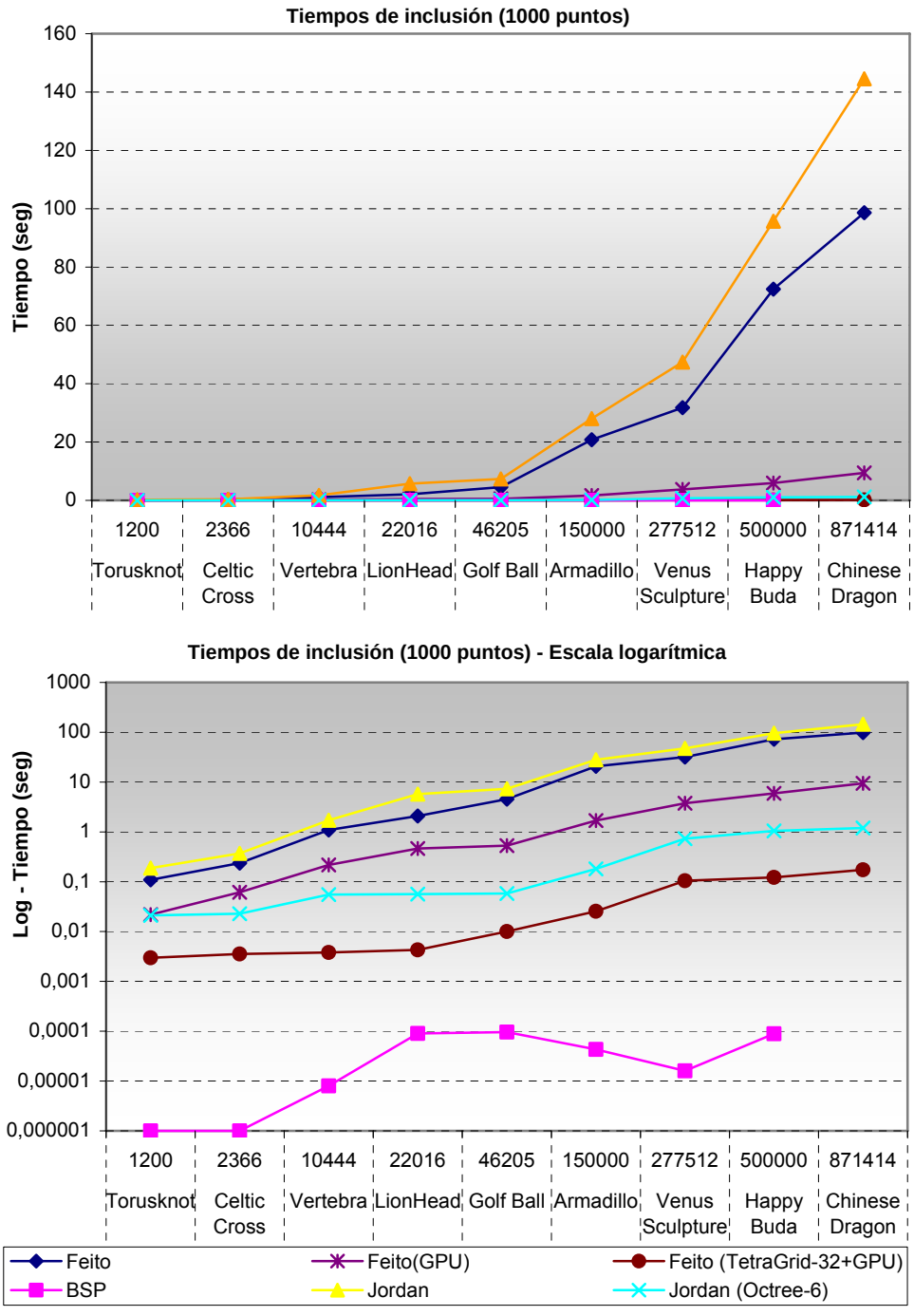


Figura 4-10. Resultados del test de inclusión (1000 puntos).

El algoritmo de Feito-Torres aparece, según los resultados de las pruebas, como el algoritmo de inclusión *sin preprocesamiento* más estable, escalable y eficiente. Los tiempos de los tests son mejores comparados con los óptimos de Jordan. En el caso de mallas complicadas, Jordan sufre los efectos de los casos especiales y repite varias veces los cálculos empeorando los resultados. Feito-Torres, sin embargo, es independiente de la forma del sólido, y mantiene una relación lineal directa con el número de polígonos, sin importar la cantidad de convexidades y agujeros que se presenten.

La eficiencia del algoritmo de Feito-Torres en la GPU es mayor que en la CPU para la mayoría de los casos. Puede verse que para mallas poco complejas, la CPU tiene cierta ventaja, ya que para la GPU hay que inicializar estructuras (las display lists) y preparar los buffers oportunos, mientras que la ejecución en la CPU es directa. No obstante, para mallas muy complejas todos estos factores tienen un peso relativo muy bajo, por lo que la versión GPU presenta unos resultados mucho mejores, llegando en ocasiones a un factor de 10 en el rendimiento. Con todo lo anterior, puede verse que el procesamiento paralelo en la GPU tiende a compensar todos estos factores. Además, hay que tener en cuenta que mientras la GPU está ocupada con el algoritmo de inclusión, la CPU puede dedicarse a otras tareas. En un sistema más complejo, como el cálculo de un árbol CSG, el test de inclusión de puntos en las mallas es sólo parte del proceso, por lo que pueden repartirse las tareas entre CPU y GPU.

Tal y como se aprecia en la Figura 4-7 y en la Figura 4-10, el beneficio obtenido por el uso del hardware gráfico es mayor cuanto mayor es el tamaño de la malla. Sin embargo aparecen algunos dientes de sierra en la curva, que coinciden con la necesidad de extender el tamaño del buffer de salida utilizado para albergar la inclusión del punto en cada tetraedro de la malla. Puede afirmarse que el beneficio alcanza su máximo cuando el número de tetraedros coincide con el número de texels de la textura.

Es importante destacar que debido a la imposibilidad de ruptura del pipeline de la tarjeta, el algoritmo no se detiene cuando aparecen casos especiales. Esto casos se producen cuando el punto se encuentra sobre una de las caras originales de algún tetraedro del recubrimiento simplicial del sólido. Al considerarse puntos aleatorios, esta circunstancia puede llegar a despreciarse. Sin embargo, en aplicaciones específicas del algoritmo de inclusión, como puede ser la detección de colisiones o el cálculo de operaciones booleanas entre sólidos, esta situación aparecerá con bastante más frecuencia, lo que obviamente no beneficia a la implementación hardware frente a la versión software.

	Feito	Feito (TetraGrid)	Feito (GPU)	Jordan	Jordan (Octree)	BSP
Eficiencia	Buena	Muy buena	Muy buena	Moderada	Muy buena	Excelente
Tiempo construcción estructura	Ninguno	Moderado[1]	Muy bajo	Ninguno	Moderado[1]	Muy alto [2]
Memoria estructura	Nada	Alta[1]	Moderado	Nada	Moderado [1]	Muy alto [2]
Dependencia de la forma del sólido	Ninguna	Ninguna	Ninguna	Moderada	Moderada	Extrema
Escalabilidad (mejor caso)	Excelente	Excelente	Excelente	Muy buena	Muy buena	Excelente
Escalabilidad (peor caso)	Excelente	Muy buena	Muy buena	Pobre [3]	Moderada [3]	Muy pobre[4]
Dificultad de programación	Muy baja	Media	Normal	Baja	Normal	Normal

[1] Depende de la profundidad de la estructura
[2] Depende de la forma del sólido
[3] Debido a casos especiales en las intersecciones
[4] Debido a concavidades y agujeros

Tabla 4-10. Características generales de cada método de inclusión implementado. Tal y como se presentan las tablas comparativas anteriores, las valoraciones son subjetivas.

4.4.4 Conclusiones

La Tabla 4-10 muestra una comparativa de las características de cada método de inclusión implementado. Las valoraciones presentadas se basan en los resultados obtenidos y en otras experiencias obtenidas durante la implementación de cada técnica. El objetivo de esta pequeña comparación es proporcionar una vista general del comportamiento de cada enfoque, así como su idoneidad para ser utilizada según el caso concreto. Esto depende de ciertos criterios como la eficiencia, el coste del preprocesamiento, la dependencia de la forma del sólido, la escalabilidad y la dificultad de implementación. Por escalabilidad se entiende la relación directa entre el aumento de la complejidad del sólido y el tiempo de cálculo del algoritmo. De esta forma, un algoritmo con buena escalabilidad aumentará sus tiempos de cálculo en función de su orden de complejidad según aumente también la complejidad del sólido tratado. Por contra, un algoritmo con mala escalabilidad presentará problemas de rendimiento conforme aumente la cantidad de datos tratados. En dicha Tabla 4-10 también se menciona la escalabilidad del peor y mejor caso; esto está pensado para algoritmos como el basado en BSP, que dependerán de varios factores para su desempeño, como por ejemplo la forma del sólido.

Como conclusión cabe destacar la conveniencia de cada algoritmo para distintas circunstancias. El mejor algoritmo sin preprocesamiento es el de Feito-Torres, en especial la versión GPU, que proporciona un excelente rendimiento sin necesidad de utilizar estructuras de datos complejas ni indexadores espaciales. Las soluciones intermedias en utilización de recursos son Feito-Torres con algún indexador y Jordan con octree (de profundidad media). Si la memoria lo permite y se realizan muchos tests de inclusión, de forma que se compensa el preprocesamiento, BSP es la mejor opción.

Lo que está claro es la relación inversa existente entre el uso de la memoria (coste de preprocesamiento en general) y el rendimiento a la hora de clasificar puntos. Los métodos más eficientes necesitan de una estructura de datos en ocasiones muy grande, mientras que los que ahorran memoria o suprimen la utilización de datos extra, no son tan rápidos. Esta relación queda muy clara en el caso de Feito-Torres y el optimizador del grid de tetraedros; con el uso de esta estructura el rendimiento aumenta de forma drástica. De igual forma, el algoritmo de Jordan y su optimización basada en octree presentan resultados dispares según se utilice dicho indexador o no. Por tanto, los requerimientos de memoria suponen una limitación importante a la hora de elegir el método de inclusión. Para la versión optimizada de Jordan se podría haber utilizado otra estructura de clasificación espacial, como el BSP alineado con los ejes. Sin embargo se ha optado por el octree porque es una estructura muy utilizada en la optimización del trazado de rayos, además de ser muy sencilla.

4.5 Conclusiones

En este Capítulo se ha presentado una solución completa para la inclusión de puntos en sólidos basada en recubrimientos simpliciales. Se ha detallado una serie de optimizaciones que permiten ajustar el algoritmo a cualquier situación, llegando en la mejor de las situaciones a un nivel de rendimiento excelente. La implementación en GPU presentada demuestra la facilidad de adaptación del algoritmo al procesamiento vectorial. Además, se han presentado unas bases teóricas para lograr un procesamiento distribuido y paralelo del método.

También se ha presentado un estudio comparativo de diversas técnicas de inclusión de puntos en sólidos, optimizaciones posibles e implementaciones en GPU, con el objeto de proporcionar los argumentos necesarios para determinar la solución más adecuada en cada situación.

Hay que destacar el uso del hardware gráfico programable. La solución planteada hace uso de las características especiales de las actuales GPUs para resolver el problema de una manera eficiente y robusta. Con la futura evolución del hardware

gráfico se espera mejorar las soluciones propuestas para hacerlas más eficientes y aprovechar de un modo más natural las capacidades de la GPU programable.

Capítulo 5

Voxelización de Sólidos Optimizada

5

La rasterización de polígonos en 2D es un problema muy conocido y tratado, existiendo numerosas formas de resolverlo, algunas de ellas óptimas e implementadas en el hardware gráfico existente. La extensión de este problema a 3D es mucho más difícil y consiste en una voxelización de los objetos representables en 3D. En este Capítulo se presentan diversas implementaciones de un algoritmo de voxelización de sólidos basado en recubrimientos simpliciales [Segura04] que permite la voxelización de sólidos poliédricos genéricos de forma eficiente y muy simple, permitiendo su implementación completa en GPU. Esta técnica se ajusta muy bien a los nuevos displays 3D, pudiendo voxelizar sólidos genéricos, tanto variedad como no variedad, y con o sin agujeros. Además, se implementan algunas soluciones existentes para resolver el problema, y se presentan los resultados de un estudio comparativo.

5.1 Introducción

La rasterización de polígonos en displays 2D convencionales es una de las operaciones básicas más comunes en un sistema gráfico. Es muy útil poder manipular de forma correcta cualquier tipo de polígono, incluyendo los polígonos cóncavos, los que tienen agujeros, los variedad y los no-variedad. Habitualmente se han utilizado dos enfoques. El primero consiste en rasterizar el polígono en su forma original mediante un algoritmo scanline [Foley96]. El segundo enfoque, consiste en

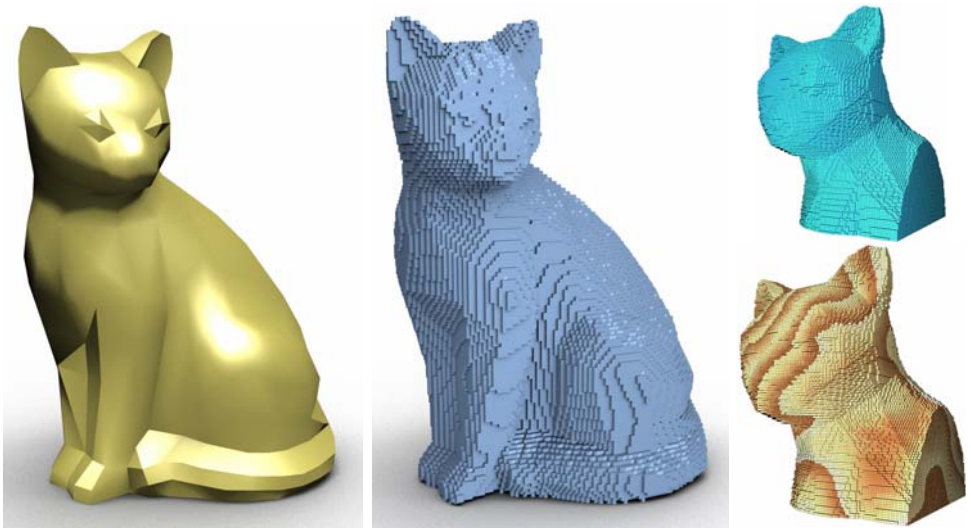


Figura 5-1. Voxelizaciones total y parciales de un sólido.

descomponer el polígono en triángulos que son rasterizados de forma más eficiente por los sistemas gráficos modernos. Con esta última técnica suele ser necesario el uso de la teselación cuando se tratan polígonos no convexos, aunque hay métodos que no requieren de dicho proceso [Rueda04].

En cualquier caso, la triangulación de polígonos genéricos es un problema importante [Bern92], pudiendo alcanzar una solución aceptable en $O(n \cdot log(n))$. La complejidad de estas soluciones las hace muy difíciles de implementar en hardware; por ejemplo, OpenGL sólo puede mostrar polígonos convexos y triángulos, pero incluye un teselador por software muy eficiente.

En los últimos años los avances en la construcción de displays 3D asequibles ha sido muy importante [Blundell00]. Este tipo de displays pueden agruparse en dos [Pastoor97]: basados en imágenes estereoscópicas, en las que se componen varias imágenes 2D para lograr un efecto 3D, y los displays reales 3D, en los que se visualiza una imagen totalmente tridimensional sin depender de la posición del observador. En esta última categoría los más prometedores son los displays de rayos cruzados (CBD) [Ebert99], basados en la excitación de iones por medio de dos lasers de diferente longitud de onda. Otra opción interesante es la basada en hologramas, muy parecida al CBD.

Aparte de los problemas técnicos, nos encontramos con la ausencia de algoritmos y librerías para manipular las grandes estructuras de datos necesarias para

visualizarlas de forma rápida y sencilla. En este sentido se ha intentado extender alguna de las librerías existentes (como OpenGL o Java3D) para agregar más opciones. Para la rasterización 3D de sólidos hay pocos algoritmos que resuelvan el problema de forma directa; la mayoría utilizan la voxelización como paso intermedio. La Figura 5-1 muestra un sencillo ejemplo de voxelización.

5.2 Revisión de soluciones

El enfoque más simple para realizar la voxelización de un sólido consiste en comprobar la inclusión del centro de cada voxel en el sólido. Para el cálculo de la inclusión suele utilizarse algún algoritmo basado en el Teorema de la Curva de Jordan [ORourque94]. El principal problema de este método, que puede ser calificado de fuerza bruta, es su bajo rendimiento, ya que para cada punto a ser probado hay que ejecutar el algoritmo de intersección rayo-sólido, que implica a su vez la utilización del algoritmo rayo-triángulo (ver Sección 4.2.2).

5.2.1 Algoritmo basado en Scanline

Una forma muy sencilla de voxelizar un sólido consiste en utilizar una variante del algoritmo scanline utilizado en 2D. Con este procedimiento se lanzan rayos siguiendo una dirección alineada con alguno de los ejes coordenados (por ejemplo, en dirección X+). Cada rayo intersecta al sólido en varios puntos contenidos en una fila del espacio de voxels resultante. Ordenando las intersecciones se localizan los intervalos de voxels de la fila que quedan fuera y los que quedan dentro del sólido.

Aunque este método es más eficiente que el de la fuerza bruta, sigue presentando los mismos problemas de precisión y los casos especiales derivados del uso de los algoritmos de intersección rayo-triángulo. También presenta problemas de aliasing, que pueden atenuarse mediante técnicas de filtrado [Sramek99] o de distancia de campos [Frisken98, Jones96]. En cualquier caso, se puede considerar este método como una técnica básica de referencia. La Figura 5-2 muestra varios ejemplos de voxelización utilizando este método.

5.2.2 Algoritmo de Huang

Huang [Huang98] describe un método para voxelizar objetos poligonales que proporciona consistencia topológica mediante medidas geométricas. Este método elimina los defectos típicos de la voxelización producidos en aristas y vértices. Está basado en espacios discretos 3D y en el concepto de separabilidad, esto es, para

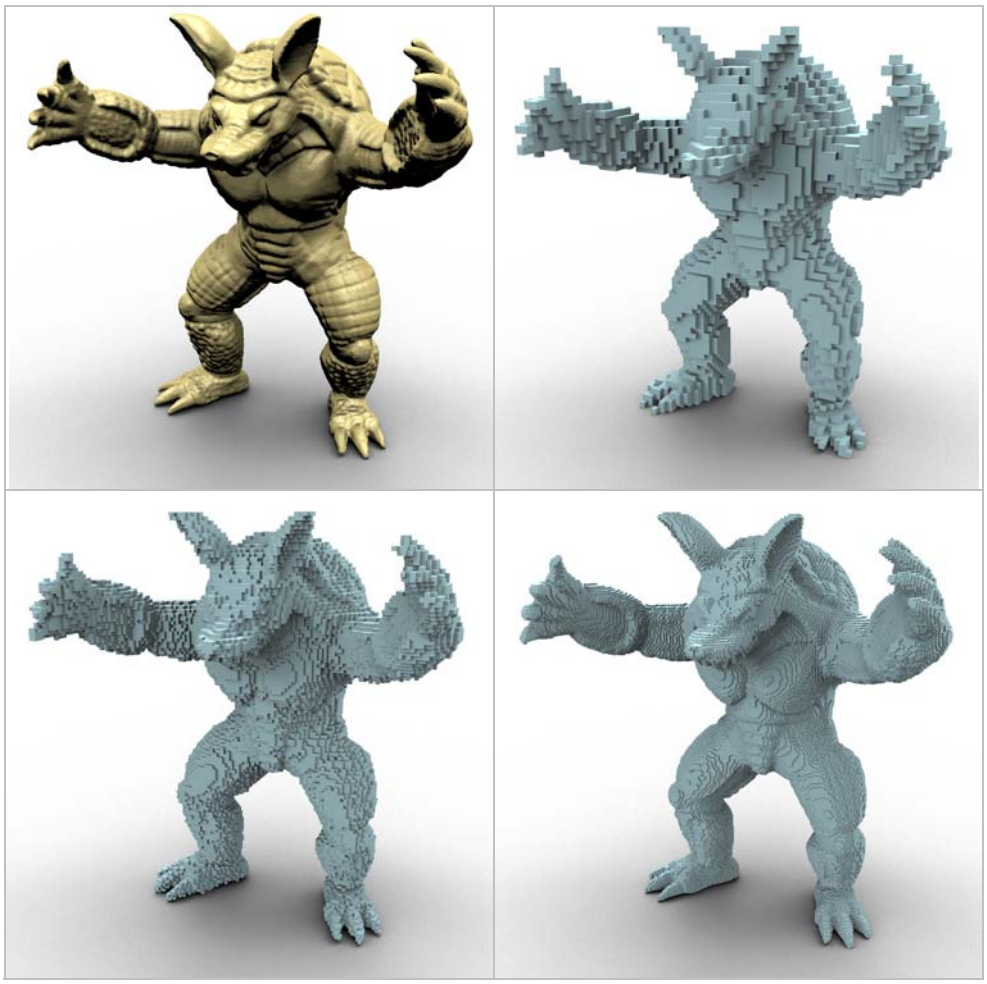


Figura 5-2. Voxelizaciones realizadas con scanline a distintas resoluciones. De izquierda a derecha y de arriba a abajo: sólido original (Armadillo), voxelización 64^3, voxelización 128^3 y voxelización 256^3.

voxelizar un plano (y un polígono) se utilizan otros dos planos auxiliares situados de tal forma que el primero queda entre éstos. Lo que se voxeliza es el espacio que queda entre los planos auxiliares. Este método funciona bien, pero no permite la voxelización del interior del sólido.

5.2.3 Algoritmo de Sramek

Sramek [Sramek99b] presenta el Modelo de Voxelización (V-Model), que es un método de voxelización de objetos con antialiasing. El V-Model de un objeto es su representación en un espacio tridimensional continuo mediante una función de densidad trivaluada. Esta función se muestrea durante la voxelización y los valores resultados se almacenan en un buffer de volumen. Se pueden aplicar métodos de filtrado e interpolación al perfil de densidad de superficie. Este método permite una representación discreta sin mucho aliasing de un objeto, pero no considera el interior del sólido. Es una de las referencias más importantes en algoritmos de voxelización.

5.2.4 Algoritmo de Haumont

El método que presenta Haumont [Haumont02] convierte escenas poligonales completas en una representación voxelizada. Guarda el estado (dentro / fuera) de los subespacios en las celdas de un octree. Primero, el algoritmo busca un punto en el espacio cuyo estado de inclusión pueda ser determinado; segundo, ese estado se propaga a las celdas vecinas visibles. Este doble paso se repite hasta que se calcula el estado de todas las celdas del octree. La ventaja de este algoritmo es su robustez, ya que puede tratar roturas, agujeros, solapamientos e intersecciones de geometrías. El inconveniente es su pobre rendimiento y su enorme ocupación de memoria.

5.2.5 Algoritmo de Jones

Jones [Jones96] ofrece un método que voxeliza un modelo utilizando una función de distancia punto a triángulo. Con este enfoque, cada voxel del espacio resultante es tratado como un punto (realmente su centro), y se calcula su distancia a cada triángulo del sólido. Hay varias optimizaciones que aumentan el rendimiento, pero en general es un método muy lento. Como pasa con la mayoría de métodos descritos, no puede voxelizarse el interior del sólido.

5.2.6 Algoritmo de Fang

Fang [Fang00] propone un método muy rápido para realizar la voxelización de un sólido mediante el uso del hardware gráfico. Realmente se basa en la utilización de métodos clásicos de rasterización Z-buffer de OpenGL. Se divide el espacio que delimita al sólido (la mínima caja envolvente) en lonchas. Cada loncha es realmente un volumen que incluye sólo una parte del sólido, y está delimitado en la práctica por los planos de corte anterior (front) y posterior (back) de OpenGL. Para cada una de estas

lonchas se ajusta el volumen de visión de forma que sólo se dibuja la parte del sólido que queda dentro del volumen ajustado. Cada imagen generada representa la voxelización de la parte del sólido que pertenece al interior de la correspondiente loncha. Estas voxelizaciones parciales deben ser acumuladas mediante operaciones lógicas, de forma que las posiciones escritas del framebuffer indican voxels interiores del sólido. Con dichas operaciones lógicas se permite cambiar de estado entre dentro y fuera a lo largo del eje en el que se sitúan las lonchas de corte.

El algoritmo es extremadamente eficiente ya que hace uso del hardware gráfico de forma muy natural. Además, permite la voxelización del interior del sólido sin coste añadido. Sin embargo, aparecen una serie de problemas que hacen que el método no sea robusto, como por ejemplo, cuando se presentan caras perpendiculares a los planos de corte anterior y posterior, además de otras dificultades reconocidas por los propios autores [Fang00]. Junto con el algoritmo de Sramek, es una de las referencias más importantes.

5.2.7 Algoritmo de Karabassi

Al igual que el método de Fang, el algoritmo de Karabassi [Karabassi02] utiliza el hardware gráfico de una forma eficiente. Utiliza seis Z-buffers tomados cada uno del rendering en perspectiva ortográfica del sólido desde cada lado de cada eje coordenado, estando el objeto centrado en el origen de coordenadas. Con las seis vistas del sólido y sus correspondientes datos de profundidad se construye en memoria principal la voxelización final del sólido. Aunque este método fue mejorado en [Passalis04], todavía tiene el problema de funcionar correctamente sólo con un número determinado de sólidos, lo que hace que no pueda ser considerado como un algoritmo de voxelización general.

5.3 Voxelización basada en Recubrimientos Simpliciales

De la misma forma que ocurre con el algoritmo de inclusión de puntos en sólidos descrito en el Capítulo 4, el algoritmo de voxelización propuesto aquí se basa en la descomposición del sólido en símplices. Los conceptos teóricos sobre los recubrimientos simpliciales aplicados en este caso se han presentado formalmente en los Apartados 3.4 y 3.5.

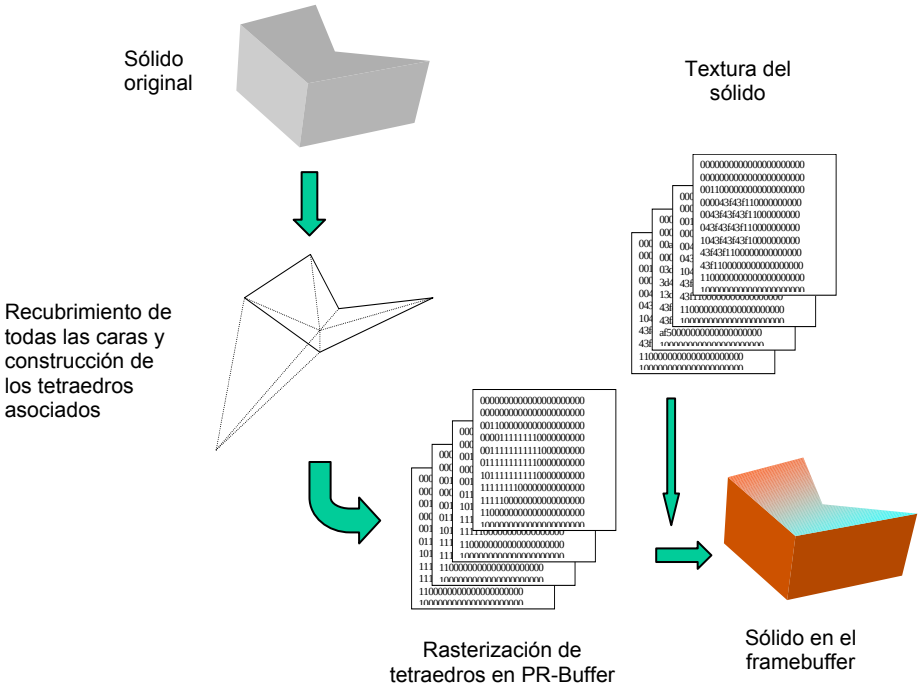


Figura 5-3. Resumen del proceso de voxelización.

5.3.1 Descripción del algoritmo

Dado un sólido definido mediante una malla de polígonos, la voxelización del mismo se consigue mediante su descomposición en símplices (tetraedros), que serán rasterizados en el buffer resultante, proporcionando una representación del sólido basada en una segmentación uniforme. La rasterización de los tetraedros deberá realizarse utilizando algunas operaciones adicionales para asegurar la rasterización correcta del sólido.

La Figura 5-3 muestra un esquema del proceso de voxelización. Este algoritmo sirve para todo tipo de sólidos definidos mediante B-Rep de caras planas, incluyendo agujeros y concavidades, sólidos variedad y no-variedad [Ogayar06b]. La Figura 5-4 muestra un ejemplo de voxelización de un sólido variedad que incluye agujeros.

Los fundamentos del algoritmo son los mismos que los del método de inclusión presentado en el Capítulo 4 en lo que se refiere a la descomposición del sólido en símplices. En primer lugar, en caso de tener una estructura con polígonos de más de

1	Para cada triángulo ABC del recubrimiento de cada cara del sólido, se construye un tetraedro *OABC* mediante la unión de dicho triángulo con el centroide del sólido (O).
2	Se rasteriza cada tetraedro obtenido en el paso 1 en el PR-Buffer, cambiando el valor de las posiciones ocupadas por dicho tetraedro.
3	Se transfieren las posiciones con un valor de presencia igual a 1 desde el PR-Buffer hasta el buffer final aplicando la función de presencia, que devuelve un valor por cada punto en el sólido o en el buffer. La definición de esta función es totalmente libre.

Algoritmo 5-1. Descripción básica del algoritmo de voxelización de sólidos.

tres lados, cada uno de éstos se reduce a un conjunto de triángulos, que componen su recubrimiento simplicial 2D [Feito95]. Realizado este paso el resultado es una malla de triángulos con signo. Esta malla será la base para la construcción del recubrimiento simplicial del sólido mediante tetraedros.

Cada tetraedro del recubrimiento del sólido está formado por los vértices de cada triángulo del recubrimiento de los polígonos de la malla que representa al objeto y un punto adicional, común a todos los tetraedros del conjunto. Este punto adicional puede ser cualquiera (en el algoritmo de inclusión se tomaba el origen de coordenadas; ver Apartado 4.3.1), aunque en esta ocasión el centroide del sólido suele ser el más adecuado (ver Apartado 3.6.1), ya que el volumen total de los tetraedros construidos tiende a ser el mínimo para la mayoría la casos. Esto es importante ya que los tetraedros deberán ser rasterizados, por lo que deben reducirse en tamaño todo lo posible para aumentar el rendimiento.

La Figura 5-3 muestra una descripción básica del proceso. Para el procedimiento de rasterización se utiliza el denominado *Buffer de Presencia* ó ***PR-Buffer***, y que sirve para almacenar el valor de presencia de cada voxel en el sólido, lo que significa la presencia en cada uno de los tetraedros que componen su recubrimiento simplicial. El valor de presencia en cada posición del buffer puede representarse con un sólo bit, que se invierte cuando uno de los tetraedros rasterizados lo cubre. Esto es equivalente a una acumulación de los signos de los tetraedros en cada voxel.

Cuando se ha completado la rasterización de todos los tetraedros, la información almacenada en el PR-Buffer se transfiere al framebuffer, aplicando una ***función de aspecto*** [Torres93]. Esta función tiene como objetivo determinar las propiedades de cada voxel en el buffer resultado, y puede ser desde un color hasta una

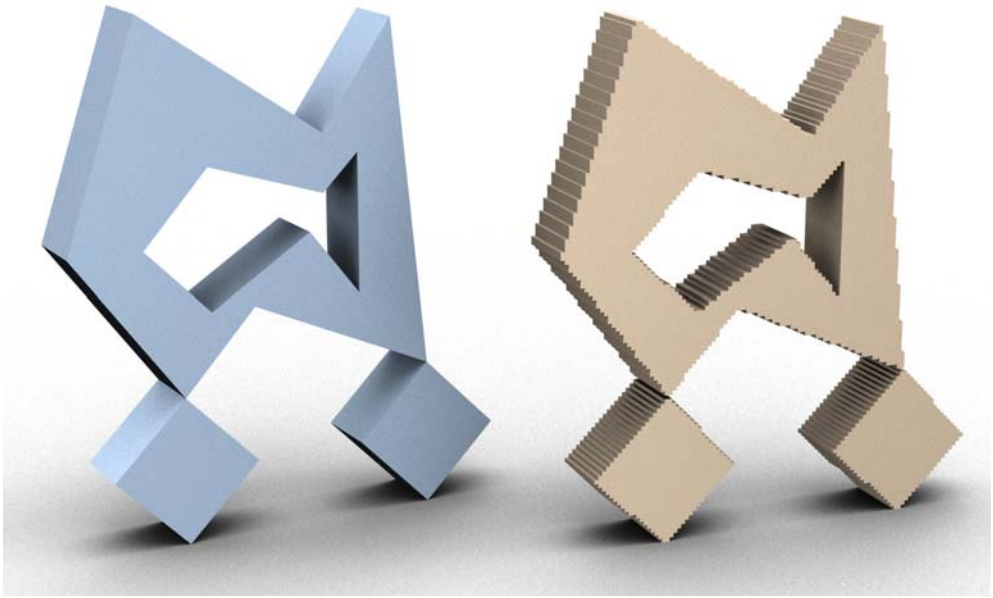


Figura 5-4. Sólido representado mediante B-Rep y de forma discreta tras una voxelización. Todas las caras del objeto tienen más de tres lados, y algunas incluyen agujeros. La voxelización se realiza sin problemas gracias a la descomposición de cada cara en triángulos signados formando un recubrimiento simplicial. Como puede verse, la teselación de las caras no es necesaria.

textura, pasando por cualquier efecto programado. El aspecto de cada voxel suele depender de las propiedades del sólido. Si se consideran sólo sólidos homogéneos, el color de todos los voxels será el mismo. La aplicación del aspecto por voxel no es un proceso fácil, ya que requiere un tratamiento individual de la textura en cada fragmento. El Algoritmo 5-1 presenta un breve resumen del proceso.

Una de las ventajas de este algoritmo es que permite la voxelización de sólidos B-Rep de caras planas con cualquier número de lados, incluyendo agujeros. Ya que la composición del recubrimiento simplicial de cada cara es trivial, el coste extra de preparación del sólido para ejecutar el algoritmo es ínfimo. La mayoría de los otros algoritmos de voxelización necesitan realizar una teselación de cada polígono de más de tres lados para obtener triángulos, ya que sólo trabajan con estas primitivas.

5.3.2 Rasterización de tetraedros

Como puede verse en el Algoritmo 5-1, el núcleo principal es la rasterización de cada tetraedro *OABC* del recubrimiento simplicial del sólido en el buffer de presencia. La rasterización de tetraedros se basa en la descomposición de los mismos en *lonchas* 2D que forman polígonos de 3 o 4 lados (siempre convexos). Estos polígonos 2D, perfectamente alineados con uno de los ejes del espacio de voxels, son rasterizados mediante un algoritmo scanline. Para mayor sencillez, si el polígono es de 4 lados y teniendo en cuenta que siempre es convexo, se convierte de forma trivial a dos triángulos. Este proceso puede realizarse tanto por software como por hardware.

La rasterización de cada tetraedro del recubrimiento se realiza como sigue. Dado un tetraedro *ABCD*, se siguen los siguientes pasos:

1. Se selecciona una dirección alineada con algún eje coordenado para formar las lonchas de rasterización (secciones 2D). A partir de ahora y sin perder generalidad, seleccionamos la dirección que sigue el eje *y*. Se ordenan los vértices del tetraedro por su coordenada *y* (ya que coincide con la dirección). Supongamos que *A* es el vértice con mayor coordenada *y*, *B* el siguiente, y así con *C* y *D* (ver Figura 5-5 y Figura 5-6). El barrido empieza por $y_s = A.y$ y termina en $y_s = D.y-1$.

2. Se calculan las intersecciones de los lados del tetraedro con el plano actual de barrido. Estos puntos de intersección se denominan a_i, b_i, c_i, d_i, tal y como se muestran en la Figura 5-5. Estas intersecciones pueden calcularse

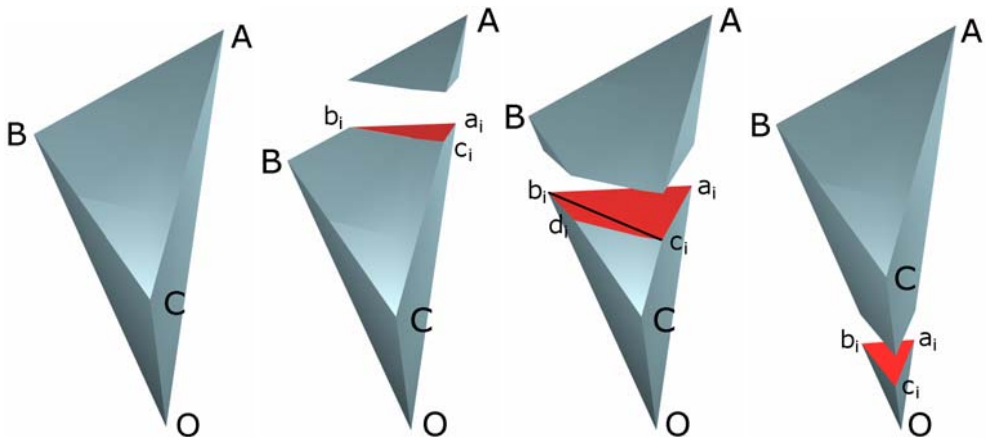


Figura 5-5. Obtención de varias secciones de rasterización para un tetraedro.

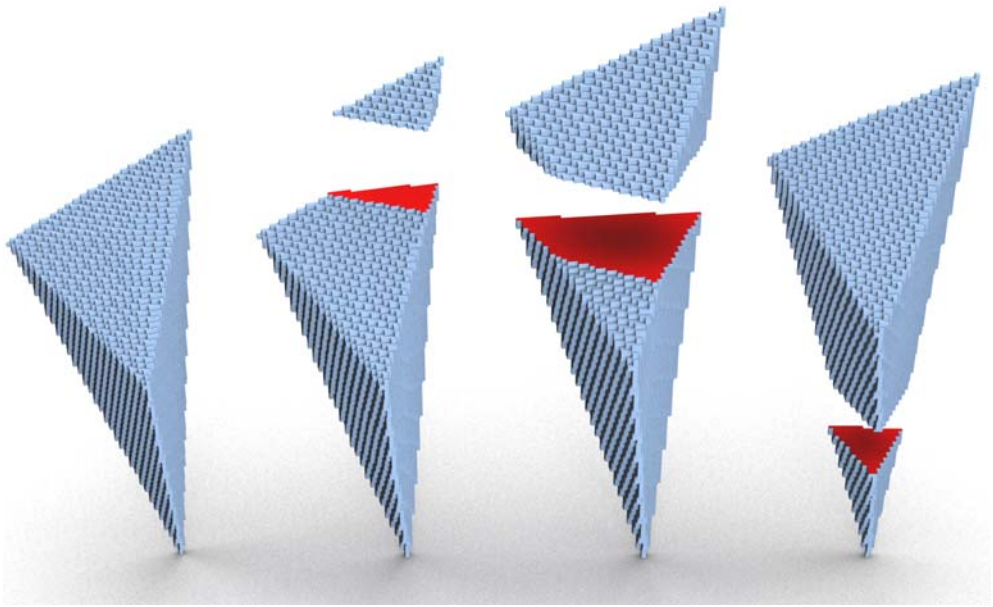


Figura 5-6. Resultado de la voxelización de un tetraedro. También se muestran varias secciones 2D del resultado final.

con una simple interpolación lineal o aplicando un enfoque incremental, que resulta más eficiente aunque puede introducir problemas de precisión. La Tabla 5-1 muestra los lados del tetraedro que deben utilizarse para calcular las intersecciones dependiendo del valor de barrido y_s. Nótese que el punto d_i sólo aparece en el intervalo $B_y>y_s>C_y$, que es cuando la sección de tetraedro se compone de 4 lados y por tanto se rasterizan dos triángulos (Figura 5-5).

3. Se rasteriza la loncha y_s del tetraedro. Esto puede hacerse utilizando simplemente un algoritmo scanline clásico de dibujado 2D de triángulos. Siempre debe rasterizarse el triángulo $\Delta a_i b_i c_i$ como muestra la Figura 5-5. Si se cumple la condición $B_y>y_s>C_y$ deberá rasterizarse un segundo triángulo $\Delta d_i b_i a_i$. Durante esta operación, los valores binarios de los voxels (x, y_s, z) cubiertos por los triángulos deben ser invertidos.

4. Se decrementa y_s y se repiten los pasos 2 y 3 hasta cumplir $y_s = D_y$.

Una vez elegido un eje principal y utilizando las aristas del tetraedro se van calculando los triángulos que corresponden a cada loncha. Puede darse el caso de que

Punto	$A_y>y_s\geq B_y$	$B_y>y_s\geq C_y$	$C_y>y_s\geq D_y$
a_i	*AD*	*AD*	*AD*
b_i	*AB*	*BD*	*BD*
c_i	*AC*	*CD*	*CD*
d_i	-	*BC*	-

Tabla 5-1. Lados requeridos para la interpolación de los puntos de intersección en el plano de barrido.

en una loncha la sección de tetraedro se corresponda con un polígono de 4 lados, en cuyo caso se dividirá en dos triángulos de forma trivial (ya que este polígono siempre será convexo). La Figura 5-6 muestra la obtención de varios tipos de lonchas o secciones 2D. Para aumentar la eficiencia se pueden utilizar las pendientes de las aristas para calcular las intersecciones en cada loncha y no tener que evaluar la ecuación punto-pendiente.

La rasterización de cada loncha (formado por hasta dos triángulos) puede realizarse por software mediante un algoritmo scanline convencional, o bien utilizando el hardware gráfico y volcando el resultado al PR-Buffer de forma conveniente. Esta última opción debe realizarse con cuidado, ya que el peso relativo de las transferencias entre la memoria principal y el hardware puede ser muy alto en algunos sistemas.

5.3.3 Implementación en GPU de pipeline fijo

Durante la ejecución de la versión estándar del algoritmo, la mayoría del tiempo se invierte en la rasterización de triángulos en 2D para formar los tetraedros en 3D. Esta tarea puede ser realizada por el hardware gráfico de forma eficiente. En este Apartado se presenta una implementación del algoritmo utilizando GPUs de pipeline fijo. Consultar el Apartado A.4 para más detalles sobre este tipo de hardware gráfico.

El enfoque original del algoritmo consiste en recorrer la lista de tetraedros asociados al sólido para rasterizarlos en orden en el buffer de presencia. Esto supone, como es lógico, el acceso aleatorio a cualquier posición del buffer de presencia para escribir el resultado. Cuando se utiliza la rasterización por hardware esto no es eficiente, ya que habría que rasterizar triángulo por triángulo las lonchas o secciones de los tetraedros en un framebuffer y transferir el resultado al buffer de presencia. Las trasferencias de información entre GPU y memoria principal se realizarían para cada triángulo, lo que resulta extremadamente ineficiente.

Es necesario replantear el algoritmo para evitar en lo posible las transferencias de datos entre GPU y CPU, y hacer rentable la rasterización de triángulos por hardware. Para conseguirlo, en lugar de iterar sobre los tetraedros del recubrimiento simplicial y rasterizar las lonchas, lo ideal es iterar sobre las secciones 2D del buffer de presencia y rasterizar los tetraedros de forma parcial según proceda. El objetivo es crear un framebuffer con el tamaño de una sección 2D del buffer de presencia resultado. Esta sección 2D intersecta un conjunto de los tetraedros a ser rasterizados. Seguidamente se obtienen las lonchas de dichos tetraedros que corresponden con la sección 2D del buffer final, y se rasterizan. De esta forma, todos los tetraedros se rasterizan de forma paralela, y tan sólo hay una transferencia de datos por cada sección del buffer 3D. Los pasos del algoritmo quedan como sigue:

1. Se crea un buffer en la GPU con las dimensiones de una sección 2D del buffer final. De esta forma, si el resultado se almacena en un espacio de 512^3 voxels, el buffer de la GPU será de 512x512.

2. Se inicializa y_s a la dimensión del espacio de voxels menos 1 (siguiendo el ejemplo anterior, 511). Se supone por tanto, que se toma el eje y como dirección de voxelización, procesando en cada paso el equivalente a una sección 2D del buffer final 3D.

3. Se inicializa el buffer GPU de rasterización (a cero en todos los valores) y se establece la operación lógica de rasterización *xor*. Se inicializa el color de dibujo a blanco, esto es, $rgb=(1,1,1)$.

4. Se calcula la lista de tetraedros que intersecta a la loncha y_s, tal que verifica $A_y > y_s > D_y$.

5. Se calculan los puntos de intersección a_i, b_i, c_i, d_i de cada tetraedro de la lista con la loncha actual. Dibujar el triángulo $\Delta a_i b_i c_i$, y si se cumple $B_y > y_s > C_y$, también el triángulo $\Delta d_i c_i b_i$ (ver Tabla 5-1).

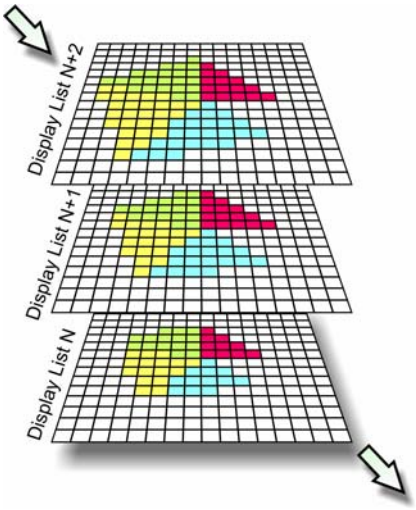
6. Transferir el contenido del buffer a una sección 2D de una textura 3D o a una estructura de datos en memoria principal. Este paso no es necesario si se realiza un rendering directo a textura.

7. Decrementar y_s y volver al paso 3 hasta que $y_s=0$.

La Figura 5-7 muestra cómo la voxelización debe realizarse por lonchas del buffer resultado, donde cada una de ellas tiene una display list asociada con los datos de los triángulos a rasterizar en cada paso. Los triángulos son distintos para cada sección 2D de rasterización.

Sólido original

Para cada sección 2D del buffer de presencia se utiliza una display list con los datos de los triángulos que forman las secciones 2D de los tetraedros a rasterizar.



El resultado se copia a memoria principal, textura 3D, o se realiza un rendering directo a sección de textura 3D.

Sólido voxelizado

Figura 5-7. Voxelización en GPU de pipeline fijo.

La transferencia de cada loncha rasterizada a textura 3D es importante ya que es más eficiente que la transferencia a memoria principal y deja la opción de utilizar los datos mediante técnicas de visualización volumétrica [Cabral94]. Sin embargo, la limitación en la velocidad de las transferencias GPU a CPU se va mejorando con la aparición de nuevos buses de comunicación.

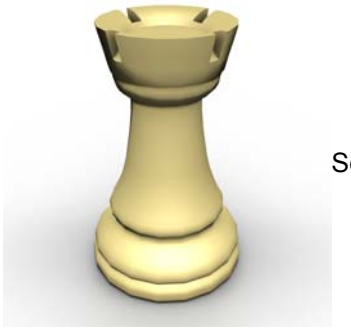
Con algunas GPUs modernas es posible renderizar directamente en una loncha o sección 2D de una textura 3D, eliminando la necesidad de realizar transferencias de datos. Sin embargo, la incorporación de esta funcionalidad suele significar que la GPU es de nueva generación y por tanto de pipeline programable, con lo que se recomienda el método de voxelización presentado en la Sección 5.3.4.

El principal inconveniente de este método es que hay que procesar una gran cantidad de triángulos en memoria principal para posteriormente transferirlos a la GPU por cada loncha rasterizada, lo que implica un retardo considerable. Esto se debe a que los triángulos cambian con cada loncha, lo que hace imposible reutilizarlos en forma de display list. Además, al estar utilizando una GPU de pipeline fijo, la función de aspecto que se ejecuta por cada voxel resultado no es fácil de implementar.

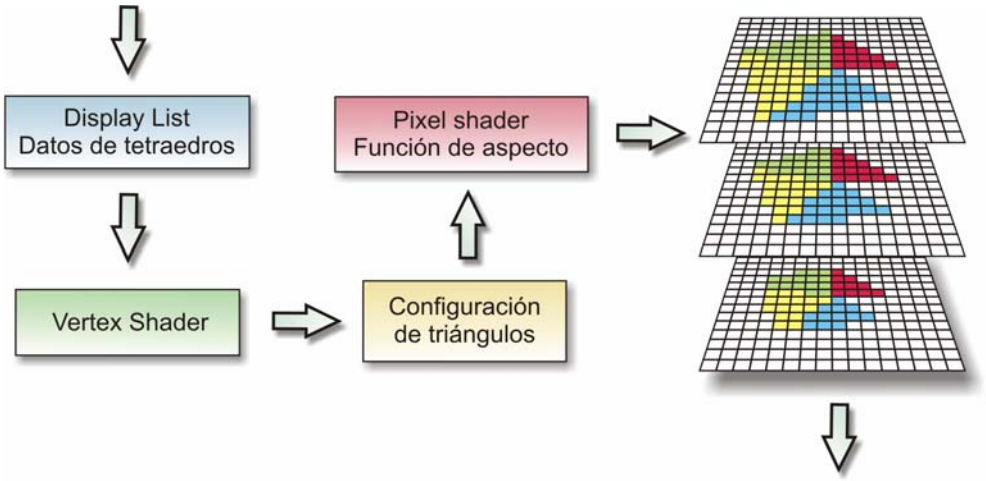
5.3.4 Implementación en GPU programable

La nueva generación de GPUs programables permite tomar el control de una buena parte del pipeline de rendering. Para el algoritmo de voxelización basado en lonchas por hardware presentado en la Sección 5.3.3 este hecho supone un beneficio significativo. La Sección A.5 contiene más detalles sobre este tipo de hardware gráfico. Con el enfoque anterior, cada loncha que se renderiza al buffer de la GPU contiene los triángulos en 2D que componen la rasterización 3D de los tetraedros. El problema principal surge debido a que los datos de estos triángulos están vinculados a cada loncha del espacio de voxels. Esto es así ya que obviamente cada uno de los triángulos de las rasterizaciones tiene una posición fija en el espacio de voxels. La solución a este problema consiste en utilizar una lista fija de triángulos que es transferida una sola vez a la GPU. Para cada tetraedro a rasterizar había que transferir hasta dos triángulos *por cada loncha* que tuviera ese tetraedro en el espacio de voxels resultante, esto es, las secciones de intersección con el espacio de voxels. El objetivo ahora es transferir a la GPU sólo dos triángulos por tetraedro, e ir ajustando las coordenadas a cada loncha del espacio de voxels donde hay intersección.

Gracias a este nuevo enfoque, con sólo dos triángulos se hace el barrido de un tetraedro completo. La modificación de las coordenadas de los vértices implicados en los triángulos rasterizados puede realizarse dentro de la GPU gracias a un vertex shader, que se ejecuta en la primera fase del pipeline gráfico.



Sólido original



Se utiliza una display list para todo el proceso o un grupo de display lists para varios intervalos del buffer 3D. Sólo se transfieren los datos de dos triángulos por tetraedro que serán recolocados según convenga desde el vertex shader. El pixel shader aplica la función de aspecto a cada voxel final.

Sólido voxelizado

Figura 5-8. Voxelización en GPU programable.

Esta ligera variación en el enfoque hace que el algoritmo sea mucho más eficiente gracias a la programabilidad de las nuevas GPUs. La Figura 5-8 muestra un diagrama del proceso.

El vertex shader es un programa en GPU que se encarga de transformar los vértices de los triángulos de forma adecuada. Calcula la posición de los puntos a_i, b_i, c_i y d_i para un tetraedro determinado teniendo en cuenta la loncha actual de rasterización y aplicando una interpolación lineal de los lados del tetraedro según se indica en la Tabla 5-1. Las coordenadas de los vértices de los tetraedros A, B, C y D son pasados al programa como parámetros variables, así como el índice del punto (0-3). Estos parámetros son en realidad los vértices que componen los triángulos a ser rasterizados. Los dos triángulos que componen el barrido de cada tetraedro utilizan esta información de vértices. Cada uno de los triángulos lleva un identificador, 0 ó 1, para determinar los lados del tetraedro sobre los que hay que realizar las interpolaciones.

Como las matrices de proyección y modelado, así como el índice de la sección de rasterización actual, no cambian durante la rasterización de la loncha del espacio de voxels actual, son todos ellos especificados como parámetros uniformes del vertex shader. El código utilizado por el vertex shader es el mostrado en el Algoritmo 5-2.

Ya que se envía el conjunto completo de tetraedros al pipeline gráfico por cada loncha del espacio de voxels, el vertex shader debe verificar si la loncha actual de rasterización intersecta a cada tetraedro en algún punto. Si esta comprobación es negativa, todos los vértices que contienen los datos asociados al tetraedro deben ser enviados a una posición fuera del volumen de visión para evitar ser procesados por el rasterizador de primitivas de la GPU y por tanto tener salida al pixel shader. Este paso puede realizarse como preprocesamiento en CPU. Para llevarlo a cabo se disponen varias display lists con subconjuntos de datos que corresponden, por la posición de los tetraedros asociados, a una zona concreta del espacio de voxels.

Para procesar varios conjuntos de tetraedros en lugar de uno sólo, se divide el espacio de voxels en otros subespacios disjuntos que llevarán asociados grupos de tetraedros, que no tienen por qué ser disjuntos, esto es, es posible y muy probable que un tetraedro esté presente en varios grupos. Aunque este enfoque aumenta la cantidad de memoria necesaria, suele suponer una ganancia más que considerable en el rendimiento.

Hay que destacar que aunque el espacio de voxels se divida en varias zonas, éstas son independientes de la resolución utilizada. Esto implica además que pueden rasterizarse porciones del sólido reutilizando el mismo preprocesamiento.

```
#define IS_MAIN_TRIANGLE (vertexData.x == 0)
#define VERTEX_INDEX (vertexData.y)
#define IN_INTERVAL (v, a, b) (a > v && v >= b)
#define INTERP (A, B, slice) (lerp (A, B, (slice-A[1])/(B[1]-A[1])) )

tetraVoxelization (
  // x -> tipo triángulo (0-1) y -> índice vértice (0-3)
  in int2 vertexData: POSITION,
  in float3 tVertexA,                          // vértice A
  in float3 tVertexB,                          // vértice B
  in float3 tVertexC,                          // vértice C
  in float3 tVertexD,                          // vértice D
  out float4 resultVertex: POSITION,           //
  out float4 resultColor: COLOR,               // resultado
  uniform float slice,                         // loncha actual
  uniform float4x4 modelViewProjectionMatrix )
{
   // se inicializa la posición de vértice fuera del volumen de visión
   float4 pos = float4 ( 10000, 10000, 0, 1 );

   // sólo se procesa el slice que intersecta al tetraedro
   if ( IN_INTERVAL ( slice, tVertexA[1], tVertexD[1] ) )
   {
      // Se procesa sólo si:
      //  El triángulo es P0P1P2 ó
      //  El triángulo es P2P1P3 y el slice intersecta el intervalo
      //   activo (B[1], C[1])
      if ( IS_MAIN_TRIANGLE ||
           IN_INTERVAL ( slice, tVertexB[1], tVertexC[1] ) )
      {
         if ( VERTEX_INDEX == 0 )    // p0
            pos.xy = INTERP (tVertexA, tVertexD, slice).xz;
         else
         if ( VERTEX_INDEX == 1 )    // p1
            pos.xy = ( slice >= tVertexB[1] ?
                     INTERP ( tVertexA, tVertexB, slice ).xz :
                     INTERP ( tVertexB, tVertexD, slice ).xz;
         else
         if ( VERTEX_INDEX == 2 )    // p2
            pos.xy = ( slice >= tVertexC[1] ?
                     INTERP ( tVertexA, tVertexC, slice ).xz :
                     INTERP ( tVertexC, tVertexD, slice ).xz;
         else
         if ( VERTEX_INDEX == 3 )    // p3
            pos.xy = INTERP (tVertexB, tVertexC, slice).xz;
      }
   }

   resultVertex = mul ( modelviewProjectionMatrix, pos );
   resultColor = float4 ( 1, 1, 1, 1 );
}
```

Algoritmo 5-2. Vextex shader para el ajuste de los vértices de los triángulos que componen la loncha variable de rasterización de cada tetraedro.

También hay que tener en cuenta que debido a la gran cantidad de información que hay asociada a los vértices enviados a la GPU, es muy posible que se produzcan demasiados fallos de caché durante el proceso. Por tanto, es conveniente sustituir cada display list por un grupo disjunto de display lists mediante una técnica de batching. Esto evita en gran medida pérdidas importantes en el rendimiento. La Sección A.7.8 detalla más este aspecto.

La rasterización en GPU programable se realiza de la siguiente forma:

1. Se crea un buffer en la GPU con las dimensiones de una sección 2D del buffer final. Si el resultado se almacenará en un espacio de 512^3 voxels, por ejemplo, el buffer de la GPU será de 512x512.

2. Se inicializa y_s a la dimensión del espacio de voxels menos 1 (siguiendo el ejemplo anterior, 511). Se supone que se toma el eje y como dirección de voxelización, procesando en cada paso el equivalente a una sección 2D del buffer final 3D.

3. Se inician las matrices de modelado y proyección.

4. Se inicializa una display list incluyendo dos triángulos para cada tetraedro a rasterizar. Deben especificarse los datos de los tetraedros como parámetros variables para el vertex shader.

5. Se inicializa el buffer GPU de rasterización (a cero en todos los valores) y se establece la operación lógica de rasterización *xor*.

6. Se establece como parámetro uniforme del vertex shader la loncha de rasterización actual y_s y se ejecuta la display list. Ahora el pipeline está lleno con los vértices y los triángulos, con lo que se ejecuta el vertex shader por cada vértice, realizando las tareas de transformación mencionadas con anterioridad.

7. Se transfiere el contenido del buffer a una sección 2D de textura 3D o a una estructura de datos en memoria principal. Este paso no es necesario si se realiza un rendering directo a textura.

8. Decrementar y_s y volver al paso 5 hasta que $y_s=0$.

En esta implementación la voxelización se resuelve casi por completo en la GPU, ahorrando considerablemente los recursos de la CPU. Una de las ventajas principales que presenta este método es que las display lists compiladas en GPU sirven

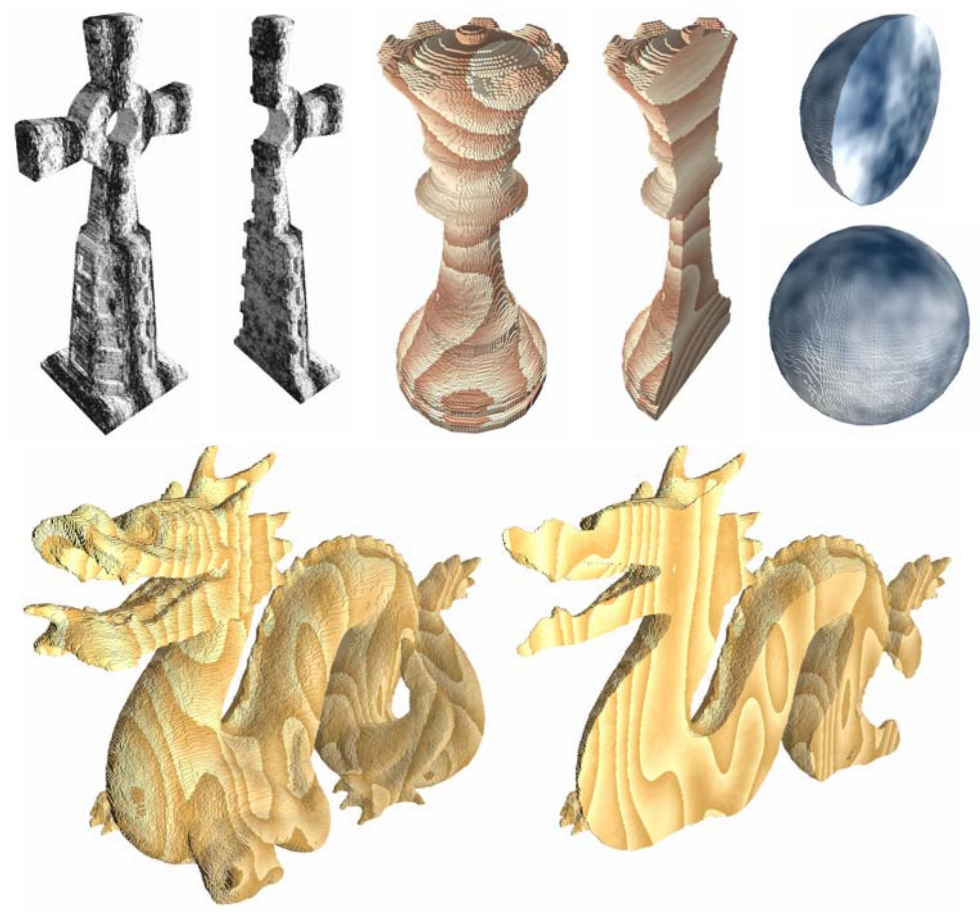


Figura 5-9. Voxelización de sólidos utilizando una función de aspecto. Algunas imágenes muestran el efecto de un corte vertical. La función de aspecto viene determinada por un generador Perlin 3D que simula distintos materiales en base a unos parámetros.

para realizar múltiples voxelizaciones a distintas resoluciones, lo que permite, entre otras cosas, facilitar la construcción de un octree o una voxelización adaptativa, en la que algunas partes del volumen rasterizado se representan con mayor resolución.

Tanto las versiones software como hardware del algoritmo se benefician del empleo de una función de aspecto que trabaja a nivel de voxel. En la versión hardware esto se lleva a cabo mediante el pixel shader, que puede controlar el color final de los pixels. En la Figura 5-9 pueden verse varias funciones de aspecto basadas en un generador de ruido Perlin en 3D aplicado al proceso de voxelización. En [Green05] se presenta un método muy eficiente para conseguir ruido Perlin en 3D directamente en GPU.

5.3.5 Implementación distribuida y paralela

El algoritmo de voxelización de sólidos en cualquiera de sus versiones, tanto software como hardware, se basa en la descomposición del sólido en tetraedros y la rasterización de los mismos. La forma de conseguir un procesamiento distribuido y/o paralelo consiste en dividir el total de tetraedros en varios conjuntos y rasterizarlos por separado en distintas unidades de procesamiento.

Al igual que ocurre con el algoritmo de inclusión de puntos basado en recubrimientos simpliciales, cualquier combinación de múltiples CPUs y GPUs sirve para repartir el trabajo de forma más o menos eficiente (Apartado 4.3.4). De igual modo, una estrategia de clustering permite aprovechar las capacidades de varios ordenadores en red.

La idea principal del reparto del trabajo en la voxelización de un sólido radica en la distribución de conjuntos de tetraedros entre varias unidades de computación. Cada una de ellas puede realizar la voxelización de cualquier forma, ya sea en CPU o en GPU. Al final, cada proceso obtendrá una voxelización parcial del sólido en un buffer 3D. Todos los buffers resultado distribuidos deben compactarse en uno, que será el resultado final.

La distribución del trabajo puede ser muy eficiente, ya que cada proceso se encarga de un conjunto de tetraedros y por tanto no necesita el sólido completo. El problema principal aparece en la fase de compactación de los buffers resultados, que dependiendo del coste de las transferencias (en memoria o en red), puede hacer que no se produzca ninguna ganancia en el rendimiento. En general, y como es lógico, si el sólido contiene un recubrimiento simplicial de pocos tetraedros, no es conveniente su procesamiento en paralelo. No ocurre lo mismo con sólidos de mucha complejidad (de varios millones de triángulos), donde el elevado número de tetraedros hace factible y rentable la distribución de la carga de trabajo, aunque ésta sea en red.

Una segunda opción para el reparto de la carga de trabajo en el proceso de rasterización consiste en repartir lonchas 2D del buffer 3D resultado entre las distintas unidades de procesamiento, por lo que cada una rasteriza determinadas lonchas. Esto obliga a que todas ellas tengan disponible el conjunto completo de tetraedros del recubrimiento. Una vez terminado el proceso en cada unidad, se transmite el resultado al buffer final. Este enfoque es el mismo que utilizan muchos programas para hacer un rendering distribuido en red, sobre todo los basados en ray tracing, que reparten partes de lo que será la imagen final entre varios nodos.

5.4 Comparación de soluciones

Es esta sección se presentan los resultados obtenidos. En el apéndice B se describen los detalles técnicos sobre el hardware, el lenguaje de programación utilizado, etc. A continuación se presentan algunos aspectos sobre la implementación de los algoritmos probados, así como los resultados de las pruebas realizadas.

5.4.1 Implementación

Para implementar el algoritmo de voxelización hay que utilizar una estructura de datos auxiliar para representar el PR-Buffer, que es una matriz 3D. El tipo base puede ser desde un simple bit hasta cualquier tipo elaborado. Si se opta por un bit por voxel, hay que tener en cuenta que hay que compactarlos para ocupar menos memoria en un tipo de datos básico del lenguaje utilizado (por ejemplo, un entero). Además hay que tener en cuenta que las operaciones de acceso y modificación serán algo más lentas. En la implementación realizada para este trabajo se ha optado por el byte como tipo base para aumentar el rendimiento.

La función de aspecto implementada para las pruebas es un generador de ruido Perlin en 3D parametrizable [Perlin85, Perlin02], que proporciona un valor para cada voxel. Este valor sirve para calcular el color correspondiente que tendrá el voxel resultante en el framebuffer.

Se han implementado dos versiones en GPU programable del algoritmo y una en GPU de pipeline fijo. La diferencia entre las versiones de GPU programable consiste en el uso o no de una técnica de multipasada. Este método permite evitar las excesivas faltas de caché que hacen que el rendimiento caiga de forma importante cuando la cantidad de tetraedros a rasterizar es considerable. Aunque el empleo del batching (ver Sección A.7.8) ayuda a este propósito, para la voxelización es necesario utilizar una descomposición mayor del trabajo. En lugar de mandar a la GPU todos los tetraedros de una zona a rasterizar agrupadas en una o varias display lists para un procesamiento en una sola fase de rendering, se realiza un particionamiento del conjunto de tetraedros para ser procesados en distintas fases. De esta forma, por cada pasada o fase de rendering, se voxeliza sólo una parte del sólido en el buffer final. Con este enfoque se hace necesario copiar en el framebuffer el contenido del buffer de presencia antes de rasterizar, con el objeto de acumular los resultados del rendering de cada loncha. Aunque con el método multipasada se realicen más transferencias de datos, los resultados que se presentan en las secciones siguientes demuestran que se compensa la pérdida de rendimiento producida con el método básico de una sola pasada.

Además del algoritmo de voxelización propuesto, se han implementado dos métodos adicionales para establecer comparaciones. Uno de ellos es el método de Fang [Fang00], descrito en la Sección 5.2.6. El otro método de voxelización restante es el Sramek [Sramek99b], descrito en la Sección 0, que ha sido implementado utilizando código de la librería *vxt* [Sramek99].

5.4.2 Resultados

Para probar el rendimiento del algoritmo basado en recubrimientos simpliciales se han probado varias versiones: una implementación puramente software (CPU), una implementación basada en GPU de pipeline fijo (FGPU) y dos implementaciones basadas en GPU de pipeline programable, de las cuales una es de simple pasada (PGPU) y la otra de múltiples pasadas (PGPU-MP). Para la versión de múltiples pasadas, éstas se han ajustado a un tamaño de display list óptimo para el hardware utilizado y para el modelo a voxelizar. Los valores para las display lists oscilan entre 10000 y 150000 tetraedros según el modelo.

La Tabla 5-2 muestra los resultados de las pruebas. Se han utilizado varios modelos con diferentes características topológicas y distinta complejidad poligonal. Las láminas 1 a 9 (páginas centrales del libro) muestran algunos datos sobre los sólidos tratados. La Figura 5-10 y la Figura 5-11 muestran gráficas comparativas de los métodos de voxelización para diferentes resoluciones. Como puede verse en los resultados, el rendimiento de todos los métodos está en función de la complejidad poligonal del sólido así como de la resolución del espacio de voxels. Hay que destacar que en las versiones que utilizan la GPU no se ha tenido en cuenta el tiempo de compilación de las display lists, considerado como preprocesamiento tanto en el algoritmo de Fang como los basados en el método aquí presentado.

El método de Sramek depende mucho más de la resolución del espacio de voxels que de la complejidad del sólido. Manteniendo la misma resolución para pruebas con distintos sólidos, se ha comprobado que los tiempos están más en función del volumen que ocupa el sólido que de su número de triángulos. Esto prueba que el cuello de botella del método de Sramek está en la rasterización.

El algoritmo de Fang es muy eficiente y presenta tiempos cercanos a los del algoritmo de Segura en GPU programable de simple pasada. Aparte de los defectos de robustez inherentes al método, se observa en los tiempos obtenidos que presenta problemas de rendimiento a partir de un número determinado de polígonos. La eficiencia en estos casos sigue siendo buena, sin embargo, se rompe la tendencia lineal del algoritmo. Esto se debe a la cantidad de información que se codifica en la display

list. La gran ventaja de este algoritmo es que tan sólo necesita de una GPU de pipeline fijo, con lo que es muy portable y fácil de implementar.

La complejidad del algoritmo propuesto está siempre en función del número de tetraedros a rasterizar y por tanto del número de triángulos del recubrimiento simplicial de la superficie del sólido. El volumen del sólido también es importante, ya que incide en la cantidad de voxels que ocupa. La versión software es comparable a la de Sramek, ya que también es una implementación completa en CPU. Los tiempos son mejores en el caso de Sramek para resoluciones medias y bajas, así como con sólidos muy complejos. Sin embargo con resoluciones altas (a partir de 512^3) el algoritmo de Segura en versión software presenta un rendimiento mucho mejor. Además hay que recordar que el método de Sramek sólo rasteriza la frontera del sólido.

Las versiones GPU del algoritmo de voxelización propuesto presentan un rendimiento lineal en función del número de caras del sólido. Sin embargo, todas tienen problemas a partir de cierto número de polígonos. Esto es debido a la información pasada a la GPU mediante display lists. Si el volumen de dichas display lists es considerable se producen fallos de caché en la GPU. Todos los algoritmos que utilizan la GPU adolecen de este problema, sin embargo éste no se presenta bajo las mismas condiciones. Como puede verse en las gráficas de la Figura 5-10 y Figura 5-11, el algoritmo que permite mayor cantidad de polígonos sin producir faltas de caché es el de Fang y el de propuesto en GPU de pipeline fijo. Esto se debe a que ambos algoritmos incluyen poca información por cada triángulo, es decir, sólo las coordenadas de los vértices. En cambio, la versión GPU de pipeline programable debe incluir todos los datos de un tetraedro por cada vértice de cada triángulo a rasterizar, lo que multiplica el volumen de las display lists utilizadas. Por esta razón es imprescindible utilizar la técnica de batching junto con la rasterización en varias pasadas para obtener buenos resultados.

La utilización de la rasterización en varias pasadas en el algoritmo presentado basado en GPU permite mantener un buen rendimiento incluso con resoluciones altas o con un gran número de polígonos. Sin embargo, como puede comprobarse en la Tabla 5-2, la Figura 5-10 y la Figura 5-11, sigue produciéndose un salto en la progresión de los tiempos. La técnica de múltiples pasadas evita fallos de caché durante el rendering pero introduce una penalización en transferencias de memoria. Cada pasada extra supone multiplicar el número de transferencias realizadas incluso si se utiliza un rendering directo a loncha de textura 3D, ya que para conseguir un proceso acumulativo hay que copiar al framebuffer los resultados acumulados en la pasada anterior. En cualquier caso el rendimiento obtenido siempre es igual o mejor que utilizando una sola pasada.

		Segura				Fang-FGPU	Sramek
	Caras	**CPU**	**FGPU**	**PGPU**	**PGPU-MP**		
64^3							
Torusknot	1200	0,015149	0,007471	0,000405	0,000421	0,001581	0,043791
Celtic Cross	2366	0,029465	0,019143	0,000418	0,000431	0,004875	0,029908
Vertebra	10444	0,119677	0,062529	0,000426	0,000429	0,001402	0,059924
Lion Head	25946	0,231812	0,124458	0,000451	0,000452	0,002671	0,074216
Golf Ball	46205	0,540364	0,301136	0,000511	0,000511	0,005085	0,141123
Armadillo	150000	1,837243	1,016027	0,000724	0,000731	0,026919	0,336862
Venus Sculpture	277512	3,372123	1,978259	0,000882	0,000892	0,048618	0,548238
Happy Buddha	500000	5,807109	3,171073	0,104842	0,003111	0,463575	1,060379
Chinese Dragon	871414	9,026048	4,289311	0,293115	0,004759	2,928737	3,229641
128^3							
Torusknot	1200	0,034065	0,014315	0,000884	0,000802	0,002796	0,382456
Celtic Cross	2366	0,046817	0,031899	0,000871	0,000859	0,005272	0,266386
Vertebra	10444	0,188831	0,126547	0,000861	0,000868	0,002903	0,343938
Lion Head	25946	0,344746	0,247748	0,000901	0,000894	0,003967	0,330267
Golf Ball	46205	0,840593	0,584013	0,001034	0,001011	0,006971	0,440203
Armadillo	150000	2,719677	2,002369	0,240484	0,003853	0,032366	0,603721
Venus Sculpture	277512	4,823303	3,816574	0,327957	0,004393	0,191359	0,808351
Happy Buddha	500000	8,382913	6,140368	3,455901	0,005916	3,305221	1,341927
Chinese Dragon	871414	12,217672	8,284901	4,862735	0,010811	5,900796	3,172377
256^3							
Torusknot	1200	0,182728	0,029841	0,001688	0,001608	0,006487	2,603154
Celtic Cross	2366	0,112931	0,064966	0,001665	0,001675	0,006663	2,167684
Vertebra	10444	0,450808	0,252491	0,001841	0,001798	0,007461	2,422523
Lion Head	25946	0,642524	0,493342	0,001831	0,001791	0,007431	2,267399
Golf Ball	46205	1,780951	1,201157	0,002022	0,002021	0,014924	2,569438
Armadillo	150000	4,531374	3,910761	0,785951	0,008743	0,039751	2,574931
Venus Sculpture	277512	7,937363	7,297263	1,258522	0,027561	0,474771	2,720338
Happy Buddha	500000	13,793924	12,110505	6,780037	0,039485	6,575625	5,270133
Chinese Dragon	871414	18,813853	16,304152	13,524312	0,077126	12,685605	4,972352
512^3							
Torusknot	1200	1,034998	0,080832	0,003868	0,003305	0,024265	22,205487
Celtic Cross	2366	0,483088	0,151475	0,003413	0,003377	0,020439	19,275034
Vertebra	10444	1,739618	0,543567	0,004891	0,003459	0,028192	20,577171
Lion Head	25946	1,805827	1,052591	0,003656	0,003635	0,073741	37,233876
Golf Ball	46205	5,780758	2,393368	0,381098	0,007404	0,061976	26,326105
Armadillo	150000	9,249501	7,859491	1,928604	0,029304	0,077904	40,720821
Venus Sculpture	277512	14,761977	14,585575	5,874548	0,096305	0,116249	31,076511
Happy Buddha	500000	24,818549	24,074927	14,264692	0,160961	13,339729	35,349826
Chinese Dragon	871414	33,077078	32,411275	23,953486	0,215207	23,222722	36,772731

Tabla 5-2. Tiempos de voxelización (en seg.) para resoluciones de 64^3, 128^3, 256^3 y 512^3.

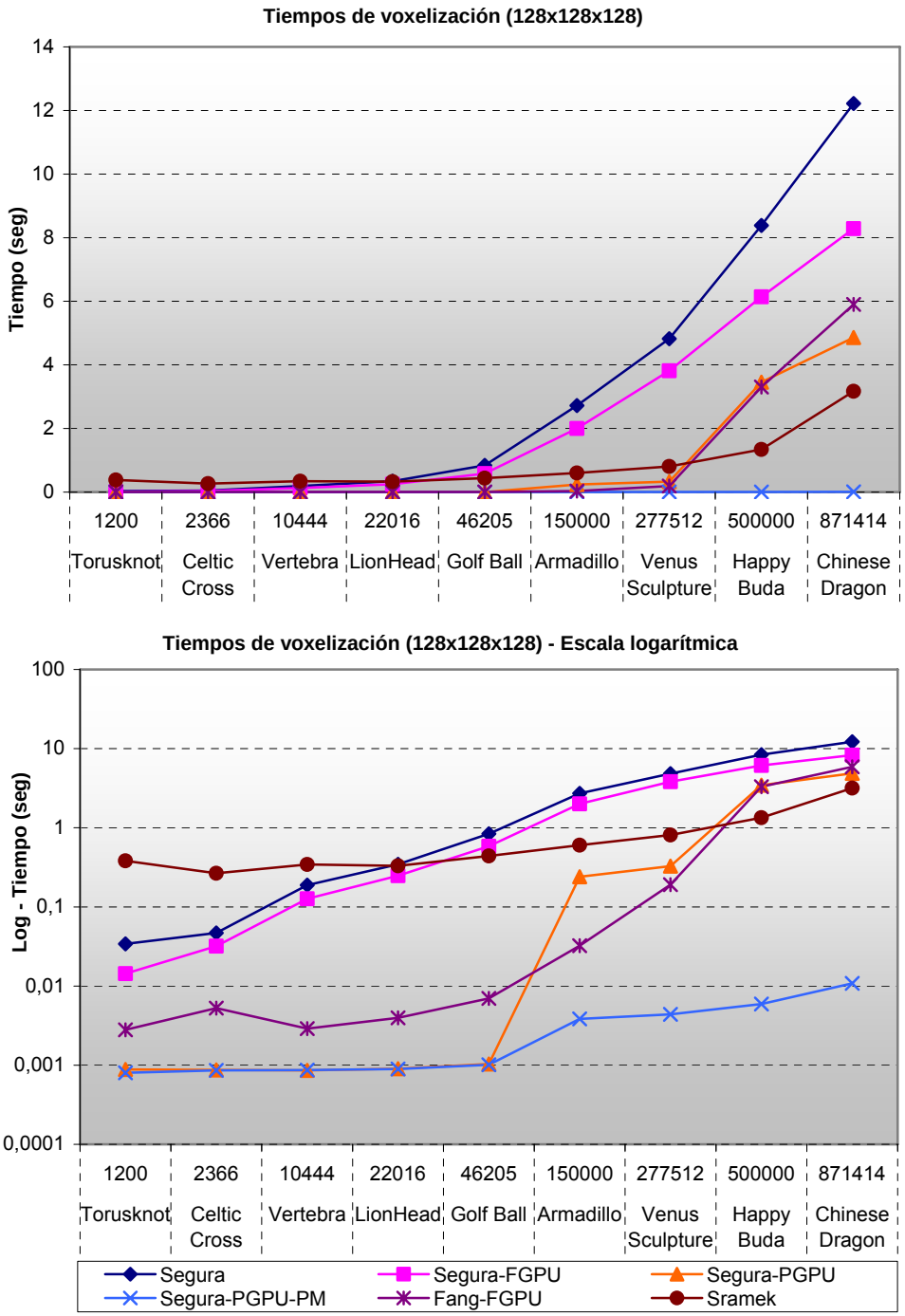


Figura 5-10. Tiempos de voxelización con resolución de 128^3.

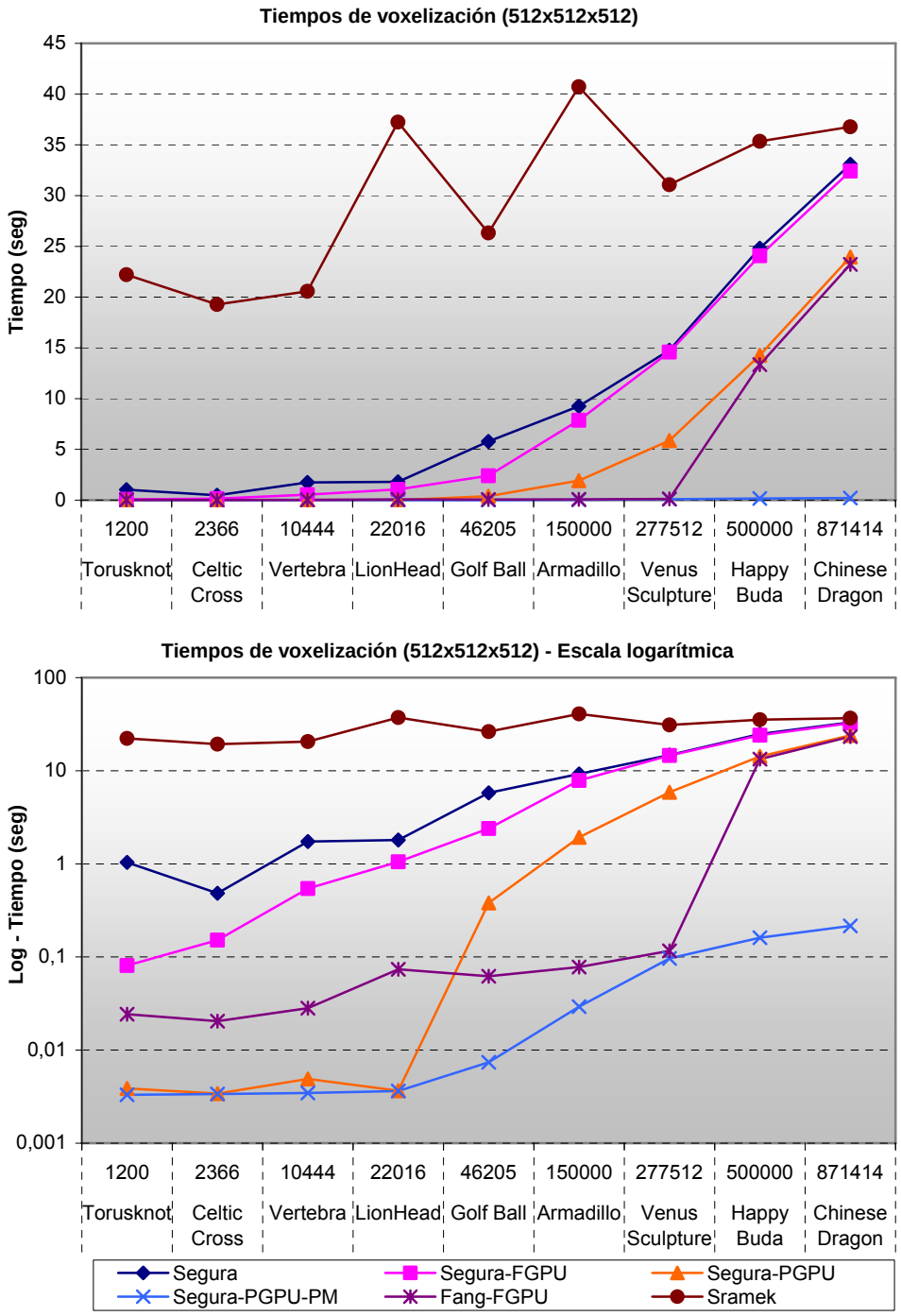


Figura 5-11. Tiempos de voxelización con resolución de 512^3.

La razón de que el algoritmo de Fang no sea tan eficiente como el de Segura en GPU programable de múltiples pasadas radica en que para todas las lonchas del buffer de voxels resultado deben procesarse todos los triángulos del modelo. Es el proceso de clipping del volumen de visión el que se encarga de descartar los que no intersectan a cada loncha de rasterización. En cambio, el algoritmo de Segura permite agrupar los tetraedros por zonas que abarcan una o varias lonchas del espacio de voxels, lo que permite que para loncha sólo se procesen los tetraedros que la intersectan.

5.4.3 Conclusiones

La Tabla 5-3 muestra una comparativa de las características de cada método de voxelización implementado utilizando valoraciones subjetivas como con tablas anteriores. Las calificaciones presentadas se basan en los resultados obtenidos y en otros factores, y muchas de las calificaciones para cada algoritmo son relativas a las valoraciones del resto de métodos. Con esta comparación se pretende resaltar los puntos fuertes y débiles de cada algoritmo en base a unos criterios comunes. Las opciones que presentan mayor rendimiento son el método de Segura basado en GPU programable de múltiple pasada y el algoritmo de Fang.

El método de Fang no necesita de ninguna optimización y se puede ejecutar en hardware de pipeline fijo. Además presenta una buena escalabilidad y apenas tiene preprocesamiento, tan sólo la compilación de una display list sin atributos por vértice. Es un método muy eficiente y fácil de implementar. El único inconveniente que acarrea es que en ocasiones presenta algunos problemas de robustez.

Las implementaciones del método presentado son muy robustas y presentan una buena escalabilidad, en especial la versión software. Esta versión es sencilla de implementar y no necesita ningún tipo de preprocesamiento. Además puede servir de base para otras técnicas más complejas, como la presentada en la Sección 5.5 que sirve para construir un grid indexador de tetraedros. Las versiones hardware tienen el inconveniente de necesitar cierto preprocesamiento y presentar algunos problemas con las faltas de caché en la GPU, lo que hace necesario el uso de técnicas adicionales para paliar los problemas. En cualquier caso el rendimiento aumenta de forma considerable mediante el uso de la GPU, en especial la versión de GPU programable de múltiples pasadas.

Todas las versiones que utilizan el hardware gráfico de alguna forma se benefician de la posibilidad de implementar la función de aspecto en GPU, lo que incluye al método de Fang. Sin embargo, es necesario contar con una GPU programable. La forma más eficiente de implementar la función de aspecto es mediante un pixel shader, que devuelve un valor por pixel de la imagen final.

	Segura				
	Software	**GPU Pipeline fijo**	**GPU Programable**	**Sramek**	**Fang**
Rendimiento	Medio	Medio	Muy alto	Medio	Alto
Robustez	Muy alta	Muy alta	Muy alta	Media	Media
Uso de GPU	No	Pipeline fijo	Programable	No	Pipeline fijo
Preprocesamiento	Ninguno	Medio	Alto	Ninguno	Bajo
Escalabilidad (voxels)	Excelente	Excelente	Excelente	Buena	Excelente
Escalabilidad (polígonos)	Excelente	Muy buena	Buena	Buena	Muy buena

Tabla 5-3. Características de los algoritmos de voxelización presentados.

El método de Sramek es más o menos asequible en cuanto a dificultad de implementación. Es muy portable, ya que se trata de un algoritmo destinado a ser ejecutado en CPU, y no necesita de ningún preprocesamiento. Además cuenta con algunos mecanismos para atenuar el aliasing, por supuesto a costa de algún tiempo de procesamiento. Presenta una escalabilidad adecuada, ya que depende casi por completo de la resolución utilizada, sin embargo no tanto de la complejidad poligonal del sólido. La robustez es sólo aceptable, ya que puede presentar algunos problemas con su método de muestreo. En cuanto al rendimiento, éste es muy bueno para las resoluciones bajas y no tanto para las altas. El principal problema en comparación al resto de métodos es que tan sólo rasteriza la frontera del sólido.

Como conclusión final hay que destacar la eficiencia y robustez del método presentado en este trabajo. Las implementaciones realizadas, totalmente por software, en hardware de pipeline fijo y en hardware de pipeline programable, ofrecen un abanico de alternativas que permiten adaptar el algoritmo a múltiples situaciones ofreciendo un rendimiento excelente y unos resultados precisos en la medida de lo posible.

5.5 Obtención de un clasificador espacial mediante voxelización

Muchos algoritmos en Informática Gráfica son susceptibles de ser optimizados mediante un clasificador espacial, como los métodos de detección de colisiones e intersección entre entidades. El caso que más interesa aquí es el algoritmo de clasificación punto-en-sólido presentado en el Apartado 4.3. A continuación se explica cómo modificar el método de voxelización presentado en este Capítulo para construir un clasificador espacial de tetraedros.

Tal y como se menciona en el Apartado 4.3.2.4, utilizando una estructura de segmentación uniforme se puede acelerar la consulta de elementos hasta orden constante. En concreto, los elementos que interesa indexar son los tetraedros que componen el recubrimiento simplicial del sólido sobre el que se lanzará el test de inclusión punto-en-sólido. Para este algoritmo de clasificación el objetivo es que dado un punto pueda conocerse la lista de tetraedros del recubrimiento simplicial del sólido que pasan por la misma zona. Es decir, se clasifica el punto a probar en el grid y se selecciona la lista de tetraedros del nodo donde queda incluido dicho punto. De esta forma el test de inclusión utiliza sólo un subconjunto de tetraedros del recubrimiento del sólido.

Aunque hay otros algoritmos que utilizan estructuras de datos espaciales en GPU como octrees [Lefebvre05], desgraciadamente no es posible hacerlo de forma eficiente con el método de inclusión punto en sólido presentado y el grid optimizador. Sin embargo, la reducción del conjunto de tetraedros que hay que procesar en el algoritmo es compatible con la implementación del mismo en GPU, lo que hace compatibles las dos optimizaciones.

Para la construcción del grid indexador de tetraedros es necesario modificar ligeramente el algoritmo de voxelización de sólidos. Se supone que cada tetraedro del recubrimiento simplicial del sólido está identificado numéricamente. La idea es crear una matriz 3D de listas de identificadores. Por cada nodo se almacena la lista de identificadores de tetraedros cuya rasterización incluye al voxel representado por dicho nodo. De esta forma, el algoritmo de rasterización debe ser modificado para escribir en la matriz 3D de listas el identificador de cada tetraedro en las posiciones convenientes. Como puede verse, en lugar de utilizar una matriz 3D de voxels se utiliza una matriz 3D de listas. En cualquier caso hay una correspondencia directa entre voxels, nodos y listas de tetraedros.

Resumiendo, el algoritmo de creación del grid es una variante del algoritmo de rasterización 3D, cuyo funcionamiento es el siguiente. Tomando como base el conjunto de tetraedros del recubrimiento simplicial del sólido, éstos son rasterizados de una forma especial. La rasterización de cada tetraedro se realiza de forma habitual en lo que

se refiere a la obtención de secciones 2D y al direccionamiento de los voxels. La diferencia radica en la operación de escritura. En lugar de escribir o invertir los valores de un buffer de bits para cada voxel ocupado por el tetraedro, se realiza una inserción del identificador en la lista del nodo que representa a dicho voxel. Este hecho hace que la implementación en GPU de la voxelización orientada a la creación de un grid indexador no sea eficiente. La razón principal es la imposibilidad de crear una matriz 3D de listas en la GPU.

Cuanto mayor sea la resolución del grid, menores serán los conjuntos de tetraedros asociados a cada nodo. Hay que tener en cuenta que el mismo tetraedro puede aparecer en más de un nodo del grid, con lo que al aumentar la resolución también aumenta la redundancia de la información almacenada en la estructura, es decir, los identificadores de los tetraedros se repiten con mayor frecuencia.

5.5.1 Rasterización conservativa

El algoritmo de rasterización 3D presentado delega en la rasterización 2D de segmentos formados por triángulos para la composición de los tetraedros en la matriz 3D resultado. Tanto la rasterización en 3D como en 2D presentan el problema del aliasing, que surge debido a la naturaleza discreta del resultado. El método empleado para el trazado de las lonchas de los tetraedros así como el ajuste de las coordenadas de los mismos al grid 3D resultado está orientado a evitar el solapamiento de las primitivas dibujadas. De esta forma, al dibujar varios triángulos pertenecientes a varios tetraedros, éstos se rasterizan de tal forma que no se solapan en ningún voxel, tal y como aparece en la parte izquierda de la Figura 5-12. De igual forma ocurre globalmente con los tetraedros. La razón es que si un voxel no es ocupado por un tetraedro, será ocupado por otro tetraedro adyacente o quedará vacío por pertenecer al exterior de un sólido. Si los tetraedros se solaparan, habría voxels que tendrían una suma incorrecta de signos y quedarían vacíos cuando realmente deberían estar ocupados. En resumen, todo el proceso de voxelización está basado en una *rasterización estándar*, que es la que utiliza cualquier GPU para el dibujado de primitivas.

Sin embargo, para la construcción del indexador de tetraedros, la forma habitual de rasterización no es válida, ya que no asegura la presencia de cada tetraedro en todos los voxels que realmente cubre. Para el grid indexador es de vital importancia que todos los voxels que intersectan a cada tetraedro lo contengan en sus listas de identificadores. Para solucionar este problema se emplea la denominada *rasterización conservativa sobreestimada* [Hasselgren05]. La parte derecha de la Figura 5-12 muestra un ejemplo de dibujado de dos triángulos en 2D mediante este tipo de rasterización. Como puede verse, todo pixel que intersecta a cada triángulo es activado, produciendo zonas de solapamiento.

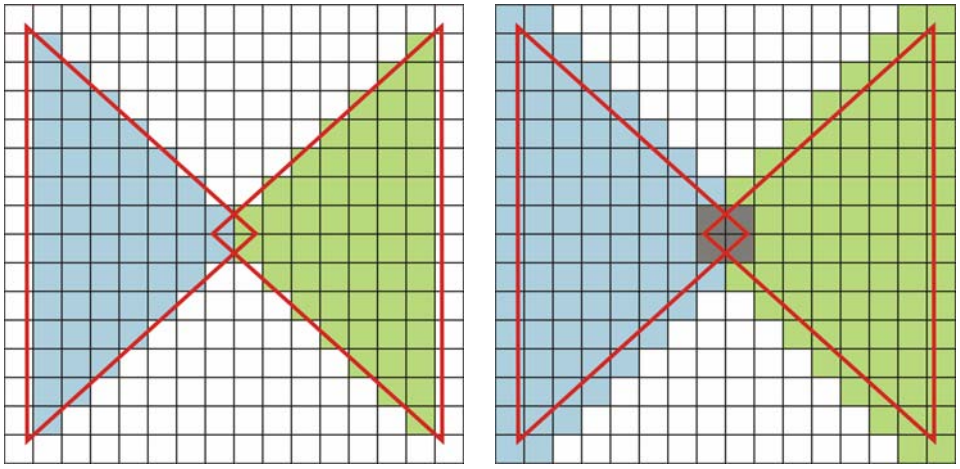


Figura 5-12. Comparación entre rasterización no conservativa (izquierda) y conservativa (derecha). La rasterización conservativa asegura la presencia de los triángulos en todos los pixels intersectados.

Existen dos formas de conseguir una rasterización conservativa. El primer método consiste en sustituir la operación de marcado de un pixel (o voxel en el caso 3D) por el marcado de un entorno de pixels. El problema es que para conseguir resultados aceptables hay que utilizar un entorno considerable, lo que hace que aumenten excesivamente las operaciones de escritura. En el caso 3D este aumento es todavía mayor, ya que el equivalente de un entorno de 9 pixels en 2D es un entorno de 27 voxels en 3D.

La Figura 5-14 muestra un ejemplo de rasterización conservativa basada en entornos de 9 pixels centrados en cada uno de los pixels que marca la rasterización estándar. Puede verse que hay un pixel que debería estar marcado y no lo está. La única solución sería aumentar el entorno para cubrir una zona más extensa. En cualquier caso, son demasiados los pixels que son marcados sin necesidad, y además se repiten con demasiada frecuencia las operaciones de escritura para cada posición del buffer resultado. En el caso de la creación del grid clasificador existe un problema añadido, ya que el uso de entornos hace que se escriba varias veces sobre cada voxel que cubre un tetraedro. Esto implica añadir varias veces el identificador del mismo tetraedro a una misma celda del grid, lo que produce duplicados en la lista de tetraedros. Estos duplicados deben ser eliminados al final de la rasterización, lo que implica un proceso bastante costoso.

El modo más eficiente de conseguir una rasterización conservativa consiste en modificar las coordenadas del triángulo de forma que utilizando la rasterización estándar todos los pixels que intersectan el triángulo original queden marcados. Es

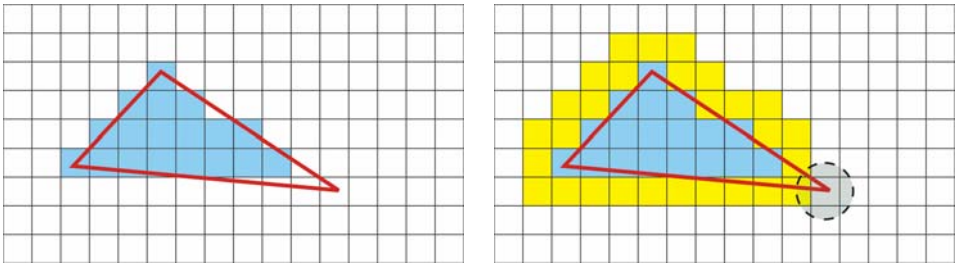


Figura 5-14. Ejemplo de rasterización 2D estándar (izquierda) comparada con una rasterización conservativa (derecha). Se ha utilizado un entorno de 9 pixels alrededor de cada pixel del triángulo rasterizado normalmente. Los nuevos pixels aparecen en amarillo. El círculo punteado indica un pixel que debería haberse marcado pero que no es cubierto por ningún entorno dibujado.

decir, se rasteriza un triángulo que no es el original pero que se deriva de éste. De la misma manera sucede con los tetraedros en 3D. Es necesario modificar la posición de los vértices de forma que al rasterizar del modo habitual el tetraedro modificado, todo voxel que intersecta al tetraedro original queda cubierto.

La Figura 5-13 muestra el ejemplo para el caso 2D. El desplazamiento de los vértices se realiza de forma indirecta, ya que el objetivo es desplazar las aristas del triángulo hacia el exterior en una distancia determinada. Para obtener resultados satisfactorios en el caso 2D, una distancia igual a la diagonal de un pixel es suficiente. Para el caso 3D, es necesario desplazar los planos de cada tetraedro siguiendo la dirección de las normales de sus caras. Los vértices serán también desplazados, y con las nuevas coordenadas puede rasterizarse el tetraedro asegurando su presencia en todos los voxels que intersecta. La distancia mínima recomendada para el desplazamiento de los planos de cada tetraedro es igual al doble de la diagonal máxima de un voxel, aunque esto depende del método de rasterización empleado y de su ajuste a la rejilla del espacio de voxels. Con el método utilizado en el algoritmo presentado

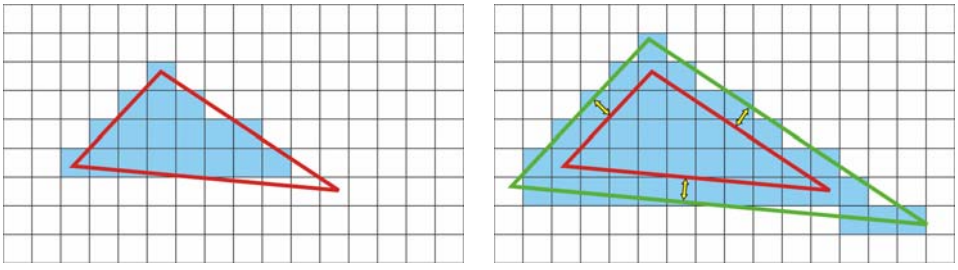


Figura 5-13. Ejemplo de rasterización 2D estándar (izquierda) comparada con una rasterización conservativa (derecha). Se ha utilizado una ampliación del triángulo y una rasterización estándar de dicha ampliación. Los pixels del nuevo triángulo cubren por completo el triángulo original.

esta distancia proporciona resultados fiables.

Los dos métodos de rasterización conservativa suponen un coste extra en cuanto al tiempo de procesamiento, así como un incremento en el almacenamiento del grid indexador. Sin embargo, el método del entorno por cada voxel es mucho más costoso, ya que requiere muchas más operaciones de escritura así como la eliminación final de duplicados. En cuanto al resultado de la rasterización, a pesar de que en ocasiones los dos enfoques ocupan más voxels de lo debido, el método de la adaptación de los vértices produce una voxelización mucho más ajustada al volumen original del tetraedro.

5.6 Conclusiones

En este Capítulo se han presentado varias soluciones basadas en recubrimientos simpliciales para la voxelización de sólidos B-Rep de caras planas. Las implementaciones en GPU presentadas demuestran la facilidad de adaptación del algoritmo de Segura al hardware gráfico, así como la ganancia en rendimiento de cada uno. También se han presentado unas bases para un procesamiento distribuido y paralelo del método.

El estudio comparativo de diversas técnicas de voxelización presentado demuestra la eficiencia del algoritmo, que es superior en rendimiento a otros métodos que sólo rasterizan la frontera del sólido. El método propuesto es muy eficiente y robusto en comparación con el resto, y aunque tiene algunos inconvenientes, es la mejor opción en la mayoría de las situaciones.

Como ocurre con el algoritmo de inclusión de puntos, el uso del hardware gráfico programable es un aspecto a tener en cuenta. Aunque el algoritmo de voxelización no se adapta de forma natural a la GPU, la solución planteada aprovecha en gran medida las capacidades del hardware gráfico para mejorar el rendimiento del algoritmo básico. Se espera que con la evolución de la GPU programable pueda hacerse una implementación que se adapte de forma más natural a la forma de procesamiento del hardware gráfico.

Capítulo 6

Optimización de Operaciones Booleanas con B-Reps

En este Capítulo se presenta un algoritmo robusto y eficiente para el cálculo de operaciones booleanas entre sólidos cuyas fronteras están delimitadas por mallas de triángulos. Con este nuevo algoritmo se pueden obtener los resultados de varios tipos de operaciones booleanas con casi el mismo coste computacional de una sola. Una de las bases del método es el algoritmo de inclusión punto en sólido presentado en el Capítulo 4, por lo que puede ser implementado utilizando el hardware gráfico programable actual.

6

6.1 Introducción

Este Capítulo presenta un algoritmo para la evaluación de operaciones booleanas con mallas de triángulos. Realmente es una versión del enfoque clásico de división de superficies y clasificación de elementos [Mäntylä83, Pilz89, Shapiro01]. El método presentado es una continuación de trabajos anteriores descritos en [Rivero02, Rivero04, Rivero06, Ogayar06c]. La base del método de evaluación consiste en adaptar la estructura de las mallas de forma que todo triángulo esté totalmente contenido, excluido o en la frontera del otro sólido, para posteriormente clasificar todos los triángulos obtenidos. Mediante una conveniente simplificación, el problema de la clasificación de los triángulos puede convertirse en clasificación de puntos. Este enfoque es muy eficiente, ya que permite utilizar el algoritmo de inclusión de puntos presentado en el Capítulo 4, que además puede ser implementado utilizando el hardware gráfico programable actual.

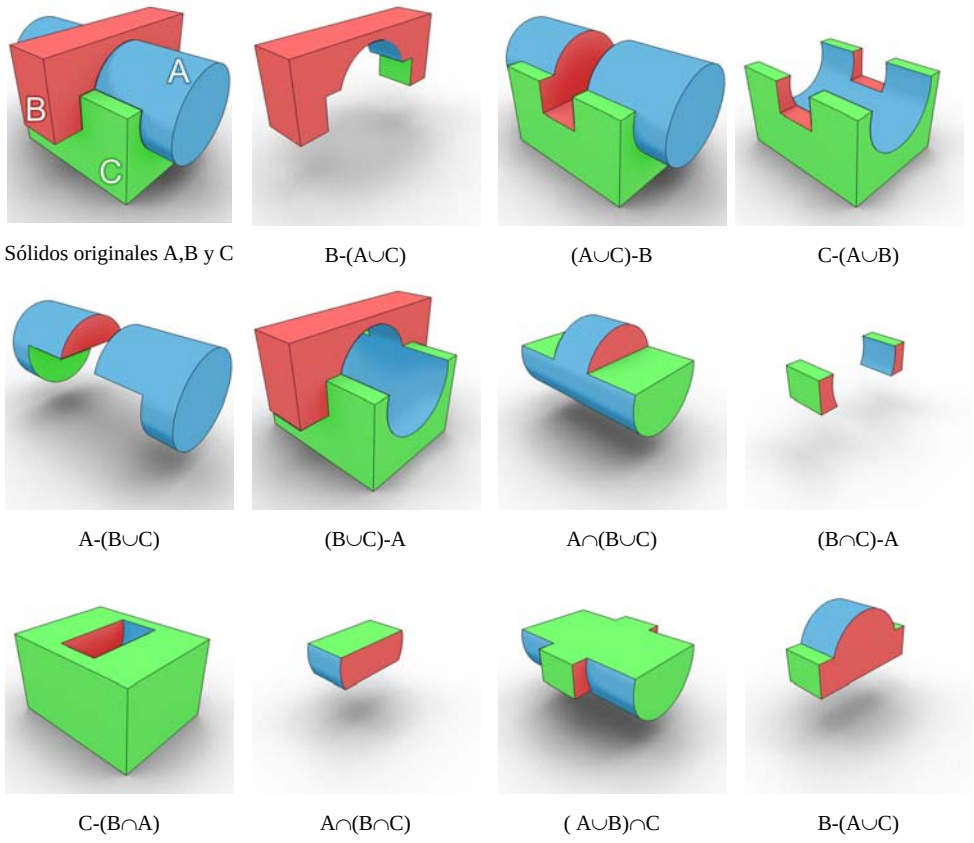


Figura 6-1. Ejemplos de operaciones booleanas entre 2 prismas y un cilindro. A partir de varias formas simples puede obtenerse toda clase de objetos. Este tipo de operaciones constituyen uno de los pilares fundamentales del modelado de sólidos.

6.2 Fundamentos

La clasificación de dos sólidos *A* y *B* consiste en establecer tres conjuntos disjuntos: *A-B*, *B-A*, *A∩B*, llamados regiones de la subdivisión. Esta operación es imprescindible para el cálculo de las operaciones booleanas entre sólidos representados mediante su frontera [Shapiro01]. En este caso, una vez realizada la subdivisión, se selecciona una de las regiones resultado en función de la operación booleana a calcular. De esta forma, para la operación de substracción únicamente habría que seleccionar la región *A-B* de la subdivisión. La Figura 6-1 muestra varios ejemplos de operaciones booleanas entre sólidos.

Son diversas las técnicas utilizadas para la subdivisión de mallas triangulares. Estas se centran en el establecimiento de líneas de sección [Mäntylä83], o bien en la realización de un recorte de triángulos [Pilz89]. El algoritmo presentado se basa en éste último tipo, estableciendo para cada par de triángulos los puntos de intersección para posteriormente dividir ambos.

Los otros métodos alternativos basados en líneas de sección suelen utilizar complejas estructuras de datos para almacenar la conectividad entre los elementos. Mediante la clasificación y recorte de las aristas del objeto se obtienen los resultados de las operaciones booleanas. El principal inconveniente de este tipo de algoritmos es que dependen por completo de la correcta construcción de la representación de los sólidos. Esto quiere decir que cualquier problema en la conectividad entre elementos produce resultados incorrectos. Además, con cada división de aristas es necesario introducir nuevos elementos y actualizar las relaciones de conectividad. El método propuesto sólo necesita un conjunto de triángulos inconexos a nivel de estructura de datos, con lo que se eliminan los problemas citados anteriormente.

6.2.1 Fundamentos de operaciones booleanas

La función característica de un conjunto *S* suele definirse como aquella función que devuelve el valor 0 si un elemento no pertenece a *S* y 1 si pertenece. En el caso del modelado geométrico necesitamos introducir una pequeña variante que nos proporciona más información. Se denomina clasificación de pertenencia de puntos (PMC : *Point Membership Classification*) [Shapiro01].

$$M(P)=\begin{cases} in & si \quad P \in in(S) \\ on & si \quad P \in \partial(S) \\ out & si \quad P \in in(S^C) \end{cases}$$

Las cadenas devueltas por *M* {*in*, *on*, *out*} determinarán si el punto es interior, de la frontera o del exterior del conjunto. Es claro que si esta clasificación es adecuada, la representación del conjunto *S* será no ambigua ya que nos indica los puntos que determinan el conjunto *S*.

Si en vez de hablar de un punto *P* hablamos de un conjunto candidato *X*, puede ocurrir que al contener infinitos puntos, haya partes de *X* en el interior de *S*, partes en el exterior, y otras sobre la frontera. En este caso la definición de *PMC* debe ser extendida. En concreto dado un conjunto *S* y un conjunto *X* se define la clasificación de *X* respecto a *M* como:

$M(X, S) = (X\ in\ S, X\ on\ S, X\ out\ S)$

Es decir, se trata de determinar la parte de X que está en S, sobre la frontera o en el exterior de S. Es evidente la relación entre la clasificación del conjunto X y las operaciones booleanas entre X y S. El algoritmo de evaluación presentado se basa en la reducción de la clasificación de un conjunto X respecto a otro S, a la clasificación de determinados puntos de X respecto de S, que evidentemente es más sencilla. La siguiente sección muestra los fundamentos del algoritmo de clasificación propuesto para mallas de triángulos.

6.2.2 Operaciones booleanas con mallas de triángulos

El algoritmo propuesto funciona básicamente de la siguiente forma: se realiza un refinamiento de las mallas implicadas en la operación booleana de forma que todos los triángulos de cada malla están totalmente en el exterior, totalmente en el interior o totalmente sobre la frontera de la otra malla. De esta forma cualquier triángulo de una malla puede ser completamente clasificado utilizando cualquiera los puntos de su interior. El baricentro es una buena opción, ya que los puntos pertenecientes a las aristas del triángulo podrían causar una clasificación de éste incorrecta. La prueba de inclusión triángulo-en-sólido queda por tanto simplificada a punto-en-sólido.

Cuando los triángulos de las dos mallas están clasificados, pueden construirse conjuntos de triángulos para obtener el resultado de cualquier operación booleana. Las condiciones a cumplir para los conjuntos son las siguientes. Sean T_A el conjunto de triángulos de la malla A, T_B el conjunto de triángulos de la malla B, y por notación sean T^{-n} un conjunto de triángulos con las normales invertidas y $n(t)$ el vector normal del triángulo t. Suponiendo que todos los triángulos de cada malla están totalmente contenidos, totalmente excluidos o totalmente en la frontera de la otra malla, entonces se definen los siguientes conjuntos que incluyen los triángulos que cumplen las condiciones de cada operación* como:

- $\{A \cap B\}$: $\{T_A\ in\ B\} \cup \{T_A\ on\ B\} \cup \{T_B\ in\ A\}$. Operación de intersección: los triángulos de A incluidos en B o en su frontera y los triángulos de B incluidos en A.

** Algunas de las condiciones presentadas para la obtención de conjuntos de triángulos incluyen la expresión $\{T_A\ on\ B\}$ para designar los triángulos de A que están sobre la frontera de B. Esta expresión puede sustituirse por $\{T_B\ on\ A\}$, que designa los triángulos de B que están sobre la frontera de A. Aunque el conjunto de triángulos obtenido puede ser distinto, la superficie representada es la misma, y por tanto puede concluirse que la frontera resultante es equivalente.*

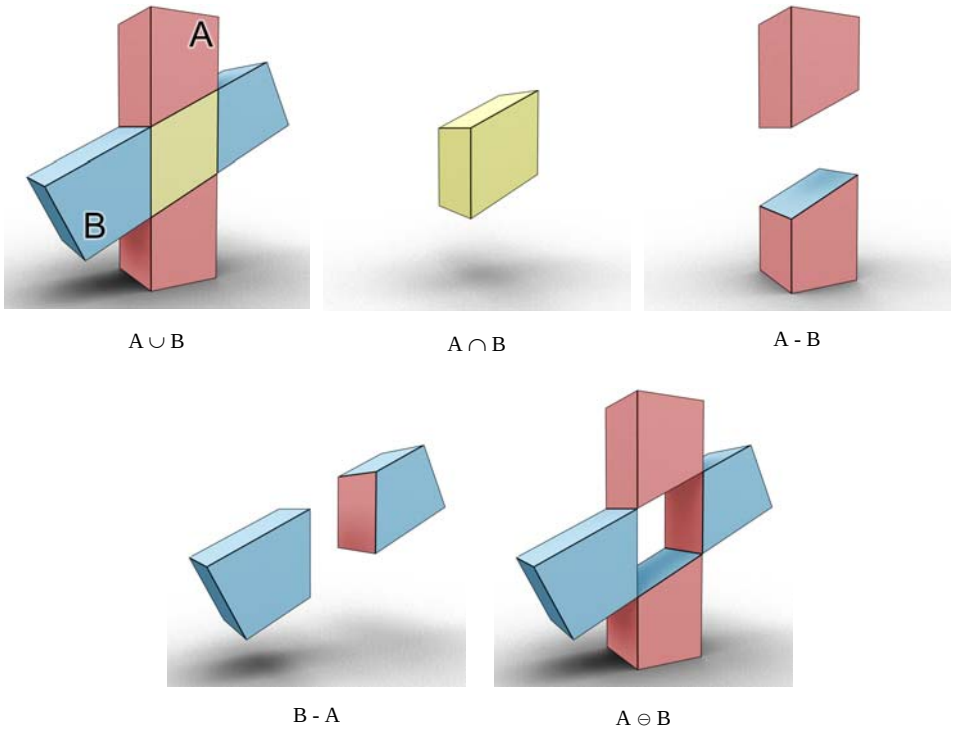


Figura 6-2. Operaciones booleanas básicas entre dos instancias del mismo sólido.

- $\{A \cap^* B\}$: $\{T_A \text{ in } B\} \cup \{T_B \text{ in } A\} \cup \{T_A \text{ on } B \,/\, n(t_i) = n(t_j);\ t_i \in T_A,\ t_j \in T_B\}$. Operación de intersección regularizada: los triángulos de *A* incluidos en *B*, los triángulos de *B* incluidos en *A* y los triángulos de *A* en la frontera de *B* con la misma normal que los de *B* con los que se superponen.

- $\{A \cup B\}$: $\{T_A \text{ out } B\} \cup \{T_B \text{ out } A\}$. Operación de unión: los triángulos de *A* no incluidos en *B* y los triángulos de *B* no incluidos en *A*.

- $\{A \cup^* B\}$: Operación de unión regularizada, en este caso igual que la operación no regularizada $\{A \cup B\}$.

- $\{A - B\}$: $\{T_A \text{ out } B\} \cup \{T_{B^{-1}} \text{ in } A\}$. Operación de diferencia: los triángulos de *A* no incluidos en *B* y los triángulos de *B* incluidos en *A* invirtiendo su normal.

- $\{A -^* B\}$: $\{T_A \text{ out } B\} \cup \{T_{B^{-1}} \text{ in } A\} \cup \{T_A \text{ on } B \,/\, n(t_i) \neq n(t_j);\ t_i \in T_A,\ t_j \in T_B\}$. Operación de diferencia regularizada: los triángulos de *A* no incluidos en *B*, los triángulos

de *B* incluidos en *A* invirtiendo su normal y los triángulos de *A* en la frontera de *B* con distinta normal que los de *B* con los que se superponen.

- $\{A \ominus B\}$: $\{T_A \; out \; B\} \cup \{(T_A \; in \; B)^{-1}\} \cup \{T_B \; out \; A\} \cup \{(T_B \; in \; A)^{-1}\}$. Operación de diferencia simétrica, equivalente a $(A\text{-}B) \cup (B\text{-}A)$: los triángulos de *A* no incluidos en *B*, los triángulos de *A* incluidos en *B* invirtiendo su normal, los triángulos de *B* no incluidos en *A* y los triángulos de *B* incluidos en *A* invirtiendo su normal.

- $\{A \ominus^* B\}$: Operación de diferencia simétrica regularizada, en este caso igual que la operación no regularizada $\{A \ominus B\}$.

El resultado final se obtiene copiando los triángulos que cumplen las condiciones de la operación booleana en la nueva malla. La Figura 6-2 muestra el resultado de los distintos tipos de operaciones booleanas expuestas entre dos sólidos sencillos. La diferencia simétrica puede obtenerse en un paso con el algoritmo propuesto; como puede verse, es equivalente a la unión de las diferencias, esto es, $(A\text{-}B) \cup (B\text{-}A)$.

6.3 Evaluación de operaciones booleanas

Básicamente el algoritmo se compone de dos etapas: la intersección entre los dos sólidos y la clasificación de los triángulos resultantes. Cada una de las fases puede ser implementada siguiendo diversos enfoques, con lo que pueden obtenerse varias soluciones distintas basadas en el mismo método básico de evaluación. En este trabajo se ha llevado a cabo una implementación razonablemente sencilla que proporciona un buen rendimiento.

En primer lugar se realiza la transformación en el espacio de los sólidos utilizando el mismo sistema de coordenadas. Se realiza la descomposición de cada uno de los triángulos de ambos sólidos de modo que en el conjunto resultante los triángulos, o bien sean coincidentes o bien tengan por intersección de su interior el conjunto vacío. En otras palabras, se comprueba la intersección de cada triángulo con todos los triángulos de la otra malla, y se realiza la teselación o triangulación adecuada de forma que todos ellos estén totalmente dentro, fuera, o en la frontera del otro sólido. Para reducir el coste de esta etapa, se utilizará una estructura espacial que permite reducir el número de intersecciones a realizar (ver Sección 6.4).

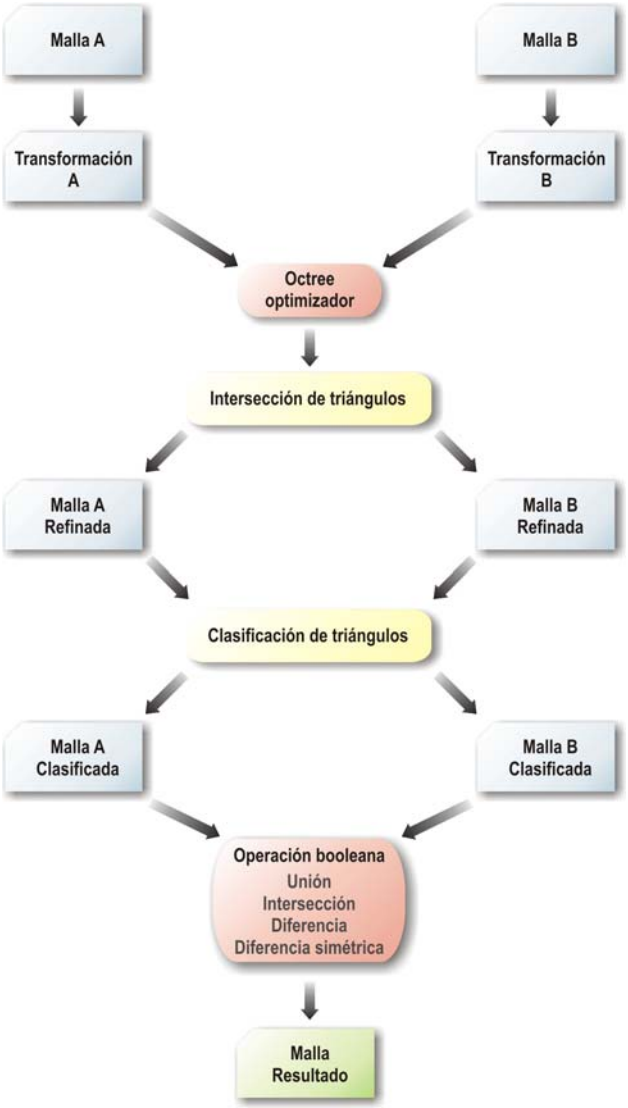


Figura 6-3. Esquema del algoritmo planteado para realizar operaciones booleanas con mallas de triángulos.

Una vez adaptadas las triangulaciones de cada malla al perfil de la intersección, se pasa a calcular el estado de inclusión de cada triángulo en cada sólido. Lógicamente, todo triángulo de la malla *A* está sobre la frontera de *A*, y lo mismo ocurre con *B*. Para probar si un triángulo de *A* está fuera, dentro, o en la frontera de *B*, se utiliza el baricentro del triángulo y se ejecuta el algoritmo de inclusión *punto en sólido* descrito en el Capítulo 4. Ya que las teselaciones de las mallas se ajustan perfectamente

a las zonas de intersección entre los sólidos, el baricentro de cada triángulo es representativo de toda la superficie del mismo a efectos de inclusión en cualquiera de los dos sólidos.

Como resultado del test de inclusión del punto (baricentro en este caso) en el sólido, se etiqueta cada uno de los triángulos correspondientes como {*in*, *out*, *on*}, esto es, dentro del sólido, fuera del sólido o en la frontera. Posteriormente, y en función de la operación a realizar, se eligen los triángulos cuyos baricentros hayan sido etiquetados de acuerdo con dicha operación. Estos nuevos triángulos formarán la malla del sólido resultante de la operación booleana.

Dada la sencillez de las operaciones a realizar y la robustez del método de inclusión utilizado, puede afirmarse que el algoritmo es más sencillo y robusto que otras opciones existentes. Además, tal y como se describe en el Apartado 4.3.3, el algoritmo de inclusión utilizado puede implementarse en la GPU. La Figura 6-3 muestra las distintas etapas del proceso completo. En los Apartados siguientes se presentarán los detalles de todas las fases.

6.4 Refinamiento de mallas

Antes de la aplicación del algoritmo planteado es necesario realizar un refinamiento de cada una de las mallas que forman los objetos *A* y *B*, de manera que los triángulos resultantes en cada malla sean coincidentes o la intersección de su interior sea vacía. Esto nos asegurará que los triángulos obtenidos de cada uno de los sólidos de *A* y *B*, serán susceptibles de clasificarse enteramente como {*in*, *out*, *on*} respecto de *B* y *A*, respectivamente. En otras palabras, mediante el refinamiento de la malla, cualquier punto de cada triángulo resultante sirve para realizar el test de inclusión triángulo en sólido, reducido entonces a un test de punto en sólido.

En el refinamiento se intersecta cada triángulo de una malla con todos los triángulos de la otra. Mediante este proceso se obtienen dos mallas nuevas cuyas triangulaciones estarán adaptadas de forma que cada triángulo de una estará completamente dentro, fuera o en la frontera de la otra malla, como puede verse en la Figura 6-4. Con cada intersección triángulo-triángulo aparecen nuevos vértices en las dos mallas. Estos nuevos vértices, que lógicamente pertenecen a las fronteras de las dos mallas, provocarán una reconfiguración de dichas fronteras mediante cambios en las triangulaciones, esto es, se sustituyen los triángulos afectados por las intersecciones por otros que quedan asociados a los nuevos vértices introducidos. La Figura 6-4 muestra cómo se dividen los triángulos situados en la zona de intersección.

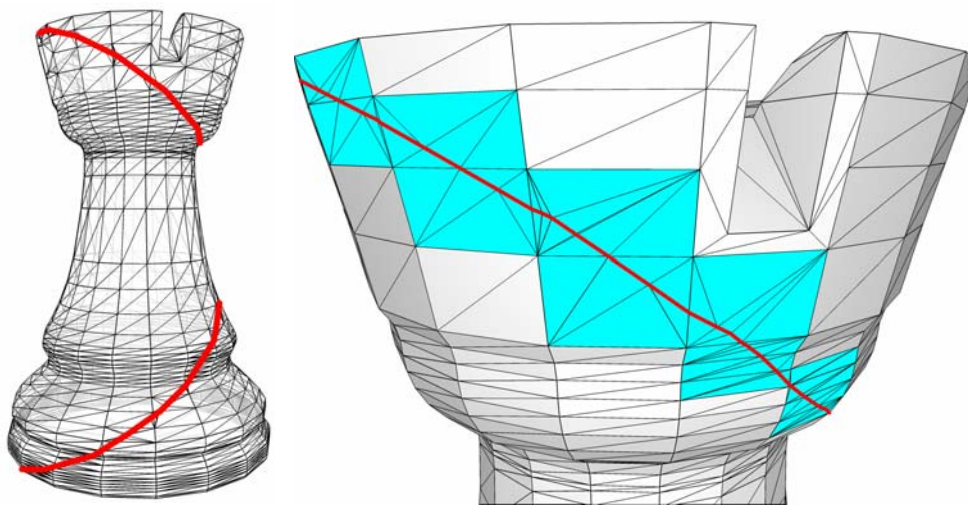


Figura 6-4. Resultado del refinamiento de una malla para ajustarse a la intersección con una esfera. En la figura de la derecha aparece una línea de intersección y los triángulos implicados en el refinamiento en color azul.

En la Figura 6-5 puede verse el efecto de la intersección entre dos triángulos. Para cada triángulo intersectado de la malla se calculan nuevos triángulos que sustituyen al original. La nueva teselación se adapta perfectamente a la zona de intersección. En la figura sólo aparece la intersección entre dos triángulos. En las operaciones booleanas, cada triángulo puede intersectar a un número bastante elevado de caras de la otra malla. Con el método propuesto se realizan todas las intersecciones en un sólo paso para cada triángulo, lo que genera un conjunto de puntos contenidos en el mismo. Con este conjunto de puntos que incluye los vértices originales del triángulo se calcula la teselación final que sustituye a dicho triángulo.

El cuello de botella de esta etapa del algoritmo de evaluación de operaciones booleanas es el test de intersección triángulo-triángulo, ya que debe ser efectuado para cada combinación de dos triángulos de cada malla. Para resolver este problema de rendimiento se utiliza un clasificador espacial que acelera las consultas sobre los triángulos de cada malla. En este trabajo se ha utilizado el octree. Con esta estructura se mantiene una jerarquía de nodos que contienen los identificadores de cada triángulo que los intersectan. Hay que resaltar que se utiliza el mismo octree para las dos mallas a la vez, esto es, se crea la estructura teniendo en cuenta el mínimo cubo englobante que incluye los dos objetos transformados en el espacio. Cada nodo mantiene los identificadores de los triángulos de las dos mallas que intersectan dicho nodo. En el momento de realizar la intersección de un triángulo de un sólido con la frontera del

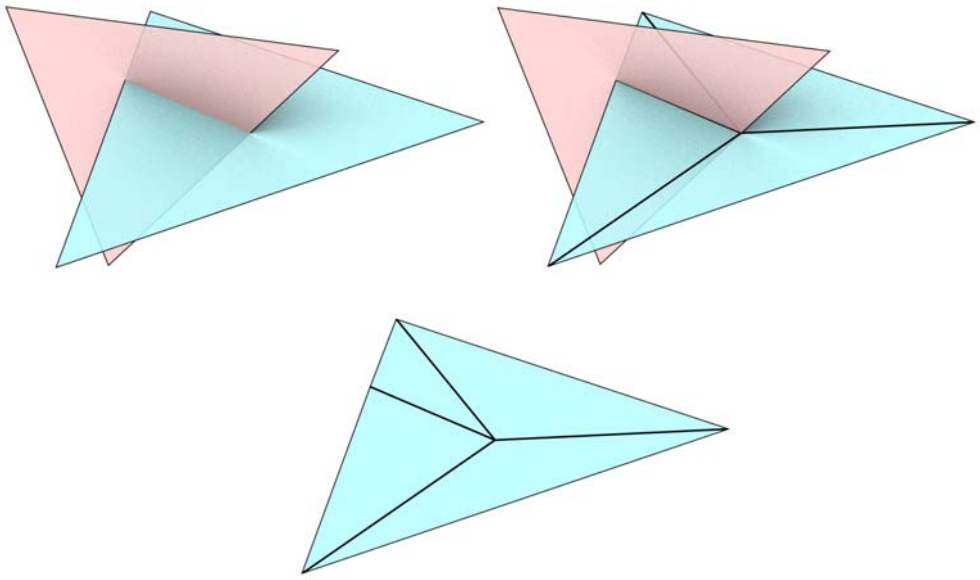


Figura 6-5. Teselación producida por la intersección entre dos triángulos. La parte inferior muestra la división final de uno de los polígonos.

otro sólido, sólo hay que probar los triángulos de éste con los que comparte nodos en el octree. Esto supone una sencilla consulta espacial. Con esta optimización se pasa de orden $O(n^2)$ a $O(logn)$ en el test de intersección.

La Figura 6-6 muestra un octree utilizado para indexar los polígonos de dos sólidos durante una operación booleana. En el Apartado 4.2.2 se presenta el uso del octree como optimizador espacial orientado a la aceleración de consultas de polígonos de un sólido. Ese tipo de octree suele ser el habitual, ya que sus nodos hoja se concentran en las zonas con mayor densidad de polígonos para facilitar la búsqueda espacial de elementos. En esta ocasión, sin embargo, el octree concentra sus nodos hoja en las zonas de intersección entre los sólidos que participan en la operación booleana. Esto se debe a que la consulta que interesa realizar durante el proceso es la obtención de la lista de polígonos de un sólido que están próximos a un determinado polígono del otro sólido. Por muy elevada que sea la concentración de caras de un sólido en una zona, si no hay también polígonos del otro objeto en el mismo subespacio no es necesario profundizar más en el octree. Este hecho puede comprobarse en la imagen de la parte superior derecha de la Figura 6-6.

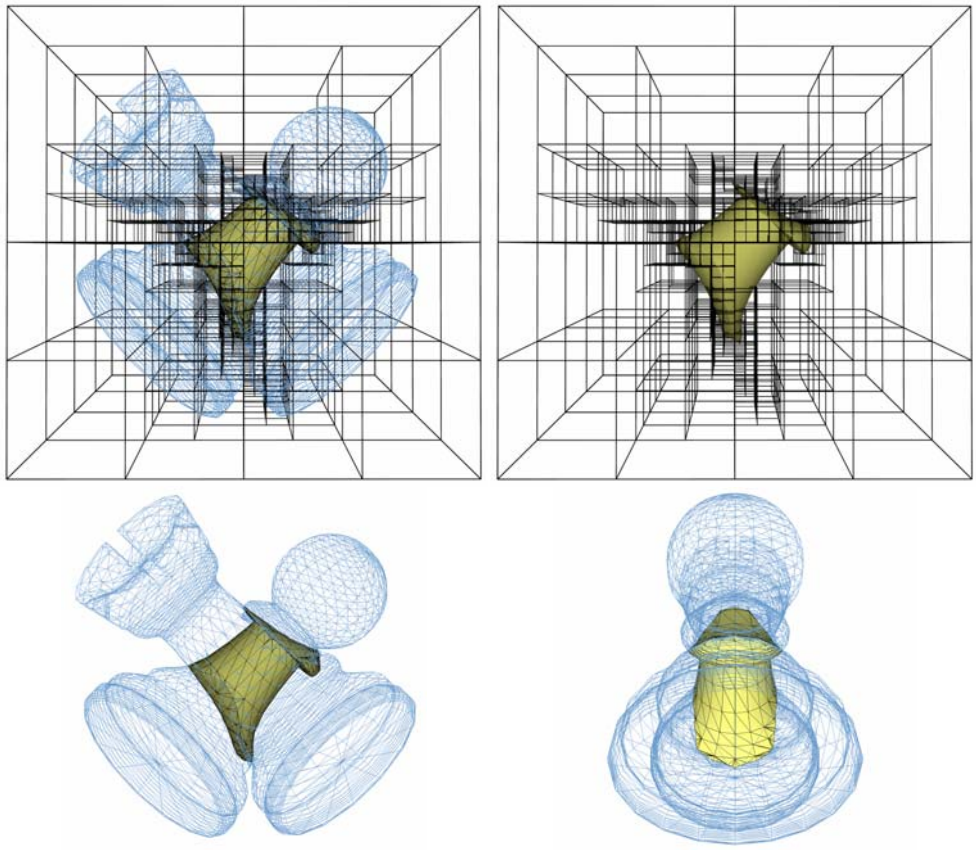


Figura 6-6. Octree optimizador empleado para la intersección de dos objetos. Los nodos hoja se concentran en las zonas de intersección entre los sólidos.

El refinamiento de las mallas, que provoca un aumento del número de vértices y triángulos, puede hacerse de dos formas. La primera consiste en calcular las intersecciones de un triángulo de la primera malla con todos los de la segunda en varias pasadas [Rivero02, Rivero04, Rivero06], de forma que cuando se detecta una intersección entre dos triángulos, éstos se subdividen para insertar y vincular los nuevos vértices. El proceso es repetido hasta que no se detectan más intersecciones. Con este enfoque se obtienen teselaciones intermedias en cada paso. En la segunda alternativa (implementada en este trabajo), para cada triángulo de una malla se itera sobre todos los triángulos de la otra malla calculando las intersecciones en una sola pasada, que van añadiéndose a una lista. Con los nuevos vértices de la lista y los vértices iniciales del triángulo, se realiza una nueva triangulación utilizando cualquier método válido (se ha empleado *Delaunay* [Lischinski94] en este trabajo). Estos nuevos triángulos sustituyen en la malla al inicial y tendrán su misma normal.

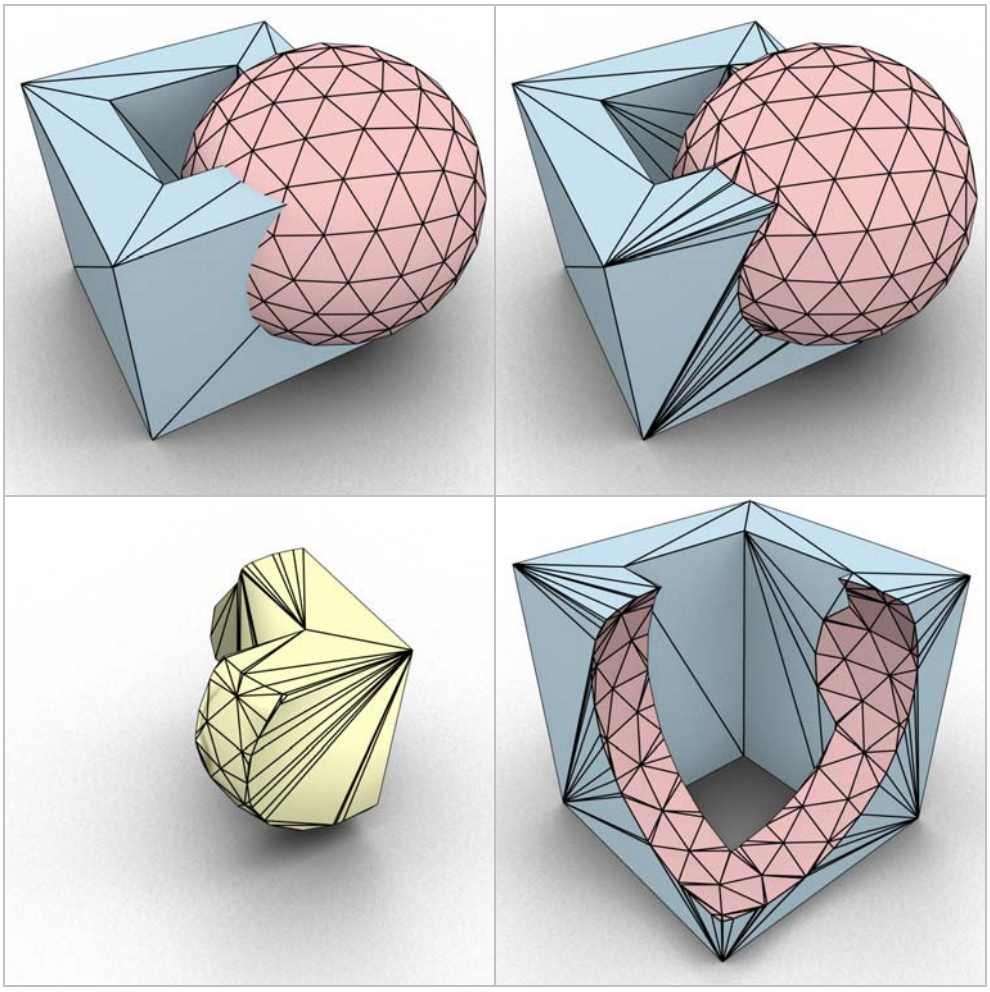


Figura 6-7. De izquierda a derecha y de arriba a abajo: sólidos antes de la operación booleana; refinamiento de las triangulaciones de los sólidos; intersección de los sólidos; diferencia entre el primer y el segundo sólido. Puede verse el efecto de la división de los triángulos para la adaptación de las superficies a las zonas de intersección.

Los puntos de intersección en las dos mallas serán siempre los mismos. Sin embargo, la triangulación no siempre resultará igual, ya que se realiza la teselación por separado para las dos mallas (para sencillez del algoritmo). Si bien esto podría representar un problema, ciertamente no lo es, ya que aunque las triangulaciones para un poliedro resultante (y común en las dos mallas) sean distintas, la superficie representada es equivalente. Además, al ser el poliedro el resultado de una intersección, su estado de inclusión es el mismo en uno y en otro sólido, y por tanto, el

mismo para cada uno de los triángulos de cualquiera de las teselaciones posibles para dicho poliedro. La Figura 6-7 muestra el efecto final de la división y clasificación de dos mallas en una operación booleana.

Debe tenerse especial cuidado con las intersecciones entre triángulos coplanares, ya que pueden dar lugar a casos especiales. Según el método de intersección que se utilice dichos casos pueden requerir un tratamiento adicional. En este trabajo se ha utilizado el algoritmo de Möller [Möller97], que resuelve las intersecciones de forma robusta y eficiente, pero se necesita de Delaunay para completar el refinamiento. Otros métodos como el descrito en [Rivero04] permiten realizar los dos procesos en un sólo paso, pero con el inconveniente de presentar casos especiales.

6.5 Clasificación de triángulos

La clasificación de triángulos es el paso final del proceso de evaluación de operaciones booleanas. Como resultado de la etapa anterior hay dos mallas refinadas, es decir, teseladas y adaptadas al perfil de intersección. Estas mallas cumplen la condición de que todos sus triángulos están completamente dentro, fuera o en la frontera de la otra malla. Lo anterior permite simplificar el test de inclusión de triángulo-en-sólido a punto-en-sólido, lo que proporciona una gran mejora en el rendimiento.

El algoritmo de inclusión punto-en-sólido descrito en el Capítulo 4 es la opción más conveniente para resolver esta parte de la evaluación de operaciones booleanas. Este algoritmo de inclusión, que además puede implementarse utilizando el hardware gráfico, se ha mostrado más eficaz que la mayoría de algoritmos similares, excepto en el caso de usar BSP (Binary Space Partition). Sin embargo, para el caso de mallas de tamaño elevado, como es habitual en los modelos actuales, no es posible usar esta técnica porque el tamaño del BSP suele superar la memoria disponible.

Hay que destacar que utilizando el algoritmo de inclusión del Capítulo 4 basado en GPU, también pueden utilizarse los optimizadores presentados en dicho Capítulo. De esta forma, aunque hay que emplear tiempo de procesamiento en construir las estructuras necesarias para los optimizadores, siempre se obtiene una ganancia en el rendimiento proporcional al coste de los mismos. Para el algoritmo de evaluación de operaciones booleanas con mallas de complejidad media o alta siempre es rentable el uso de los optimizadores en el algoritmo de inclusión de puntos.

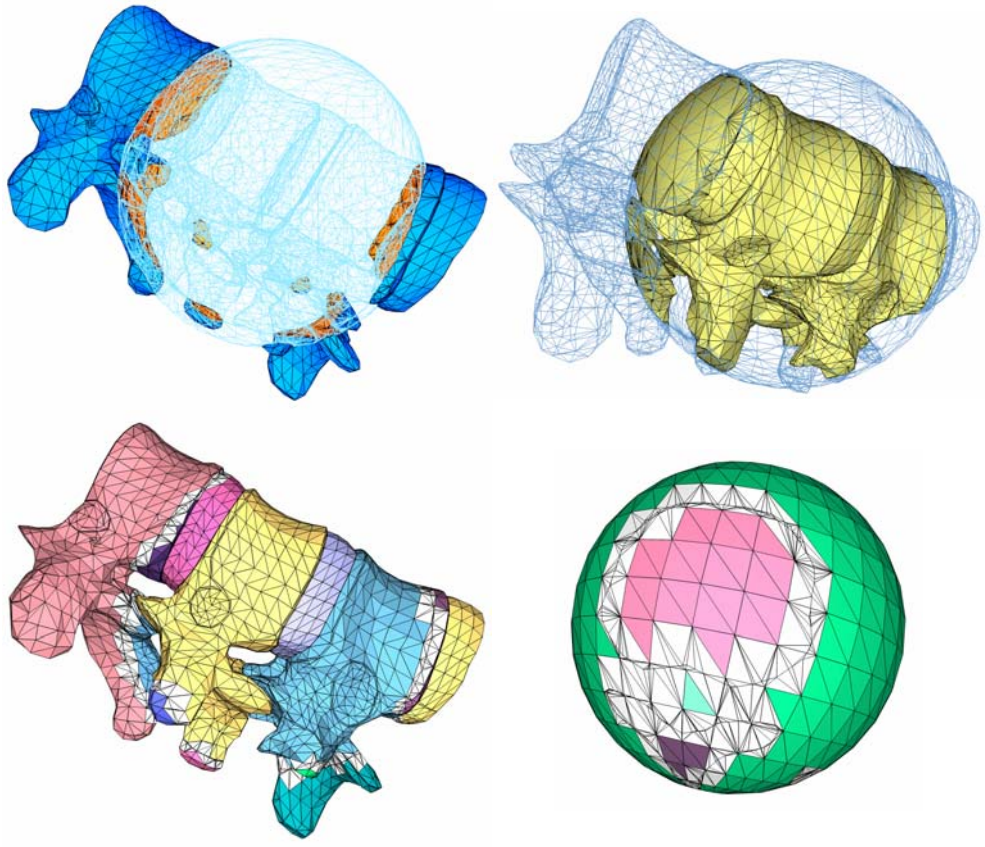


Figura 6-8. De izquierda a derecha y de arriba a abajo: diferencia entre el modelo Vertebra y una esfera; intersección entre los dos modelos; teselación del modelo Vertebra mostrando los grupos de polígonos que conforman las zonas de clasificación en distintos colores; teselación del modelo esfera mostrando los grupos de polígonos que conforman las zonas de clasificación en distintos colores.

6.5.1 Clasificación optimizada

Cuando se calcula la intersección entre los dos sólidos involucrados en una evaluación booleana, hay triángulos de cada malla que no se dividen porque están totalmente dentro, fuera o en la frontera de la otra malla desde el principio. Estos triángulos están localizados en grupos que representan una parte de la frontera del sólido al que pertenecen. Dichas zonas de la superficie normalmente están delimitadas por los triángulos que deben subdividirse como consecuencia de estar implicados en la teselación, es decir, los triángulos situados en la zona de intersección entre los sólidos.

Si se calcula la información de conectividad entre los triángulos de la malla, pueden construirse grupos de triángulos separados por las zonas de intersección. El cálculo de la conectividad entre caras es un proceso relativamente sencillo que puede realizarse al vuelo y sin necesidad de contar con datos precalculados. Los grupos resultantes cumplen la propiedad de estar totalmente dentro o fuera del otro sólido, ya que están compuestos de triángulos conectados que cumplen esta misma propiedad. De esta forma, cada grupo de triángulos conectados puede clasificarse respecto del otro sólido utilizando un sólo punto perteneciente a la superficie a la que representan, esto es, tomando cualquier punto de cualquiera de los triángulos del grupo.

La Figura 6-8 muestra el efecto de la teselación sobre los dos sólidos que intersectan. La imagen de la derecha presenta cada grupo de triángulos de un color diferente. Estos grupos forman partes de la superficie que pueden ser clasificadas como un sólo punto. Los triángulos en blanco pertenecen a la zona de intersección, y deben ser clasificados individualmente. Con esta optimización el número de test punto-en-sólido totales que hay que realizar para la evaluación de las operaciones booleanas decrece drásticamente. Para el caso concreto de la figura el número de tests realizados es del 28.1% sobre el total, con lo que la reducción es del 71.9%. Esta reducción se aplica sobre un conjunto de triángulos que incluye todos los triángulos de las dos mallas implicadas tras ser refinadas.

La proporción de tests que pueden ser descartados depende de la complejidad de las mallas y de la extensión y complejidad de la zona de intersección. Con mallas muy sencillas, lo que supone que los polígonos ocupan mucha superficie, casi todas las caras pueden estar implicadas en la intersección y posterior teselación. En estos casos, la reducción es muy baja. Cuando se realizan operaciones con mallas muy complejas donde los triángulos ocupan muy poca superficie, pocos de éstos intervienen en las intersecciones. En estos casos la reducción puede llegar a ser muy alta, de hasta 85% a 95% sobre el total, dependiendo de la extensión de las zonas de intersección.

Ya que el test punto-en-sólido es el proceso que más tiempo de cálculo demanda de todo el algoritmo, esta optimización es fundamental para obtener un buen rendimiento, en especial con mallas de triángulos muy complejas.

6.5.2 Evaluación booleana

Cuando todos los triángulos de cada malla están clasificados respecto a la otra (con estados *in*, *out*, *on*), ya pueden obtenerse nuevos sólidos mediante operaciones booleanas. Para ello, basta con seleccionar los triángulos que cumplan las condiciones necesarias según la operación. En la Sección 6.2.2 se presentan dichas condiciones para obtener los conjuntos de triángulos. Hay que destacar que el algoritmo permite tanto operaciones regularizadas como no regularizadas [Rossignac99]. También hay que

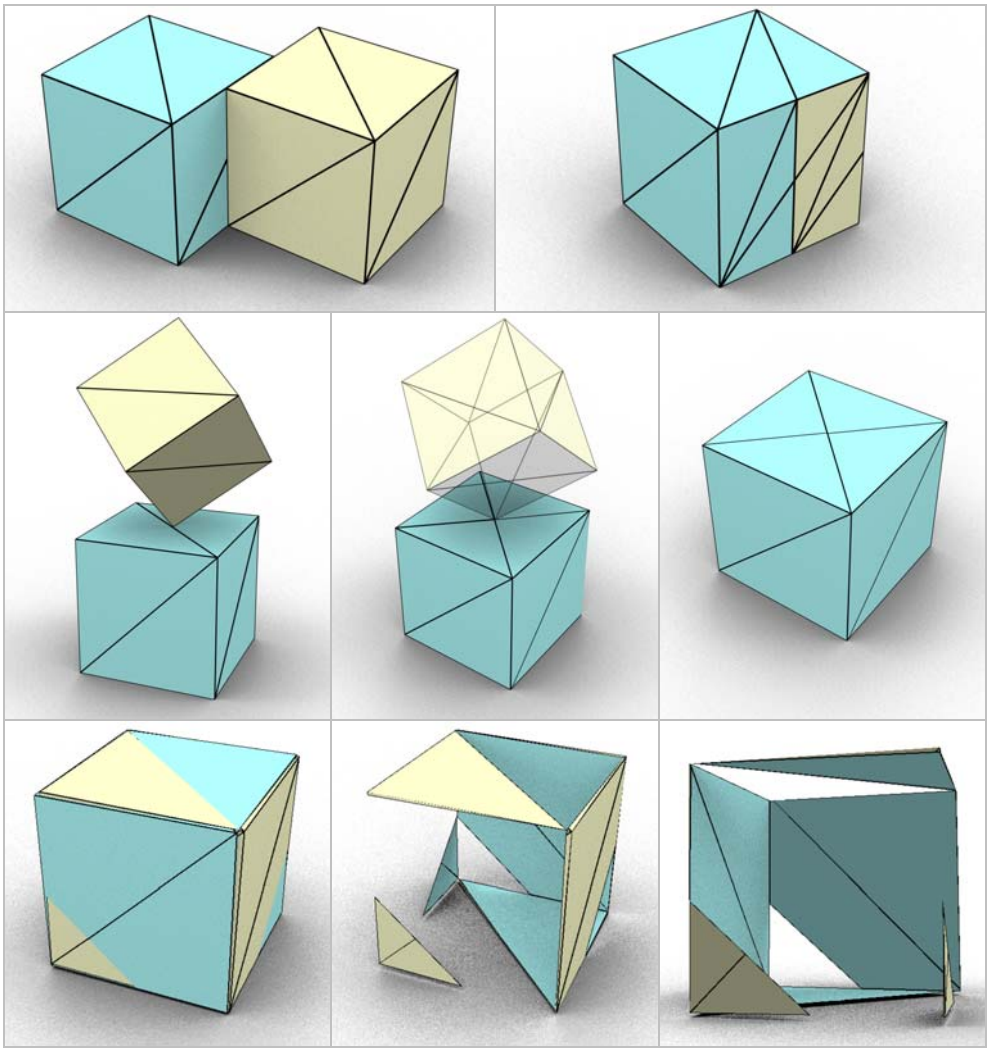


Figura 6-9. Diversas operaciones de diferencia booleana entre cubos. La posición de los objetos hace que se produzcan casos especiales en las intersecciones, incluyendo cara con cara (fila superior), vértice con arista (fila intermedia) y sólidos muy cercanos (fila inferior).

mencionar que cualquiera de las operaciones booleanas puede ser realizada de forma directa a partir de las mismas mallas refinadas. Por esta razón, el coste de realizar todas las operaciones booleanas simples que pueden aplicarse entre dos sólidos es casi el mismo que el de una sola de ellas. Esto puede ser de utilidad en algunas ocasiones, como en la evaluación optimizada de un árbol CSG.

La Figura 6-9 muestra varios casos especiales que pueden presentarse en las operaciones booleanas. Hay que destacar el tercer ejemplo (fila inferior), que muestra el resultado de la diferencia entre dos instancias del mismo objeto ligeramente rotadas. El resultado se compone de varias piezas muy finas aunque definidas correctamente. Se ha comprobado que otros métodos de evaluación basados en recorte de aristas presentan problemas en estos casos.

6.6 Implementación y resultados

En este Apartado se describe la implementación realizada para probar la validez del algoritmo. El método de evaluación de operaciones booleanas presentado puede implementarse de varias formas. Cada una de las fases en que se divide puede ser llevada a la práctica con distintos enfoques. Por ejemplo, el refinamiento de mallas puede realizarse utilizando otro algoritmo de triangulación distinto al presentado aquí, y el test de inclusión punto-en-sólido puede estar basado en GPU o no. En cualquier caso, en este trabajo se ha llevado a cabo una implementación robusta y razonablemente eficiente que hace uso intensivo de la GPU, además de otras optimizaciones interesantes ya indicadas. Para que las operaciones booleanas sean más versátiles, las funciones implementadas permiten la aplicación de una matriz de transformación completa a cada una de las mallas. Al permitirse el uso de cualquier tipo de escalado, las normales de cada triángulo deben recalcularse después de la transformación. Las figuras 6-10, 6-11 y 6-12 muestran ejemplos de operaciones booleanas.

El primer paso de la evaluación consiste en realizar la intersección entre los dos objetos y calcular la zona de intersección. Como resultado intermedio se obtienen dos mallas teseladas convenientemente que contienen la línea de intersección como parte de la malla. El cuello de botella de esta fase es el algoritmo de intersección triángulo-triángulo, ya que debe realizarse para cada combinación de dos triángulos de las dos mallas. Esto lleva a un coste del orden de $O(n^2)$ en la fase de intersección, lo que es muy ineficiente. Gracias a la utilización de un octree optimizador común para las dos mallas transformadas, el orden se reduce a $O(log(n))$. Cada nodo del octree contiene las listas de identificadores de triángulos de las dos mallas que intersectan el volumen representado por dicho nodo. Con una profundidad de octree de 7 u 8 el rendimiento aumenta drásticamente, manteniendo un tiempo de construcción de la estructura relativamente bajo para mallas complejas. Esta es la optimización más importante de la primera fase de la evaluación de operaciones booleanas.

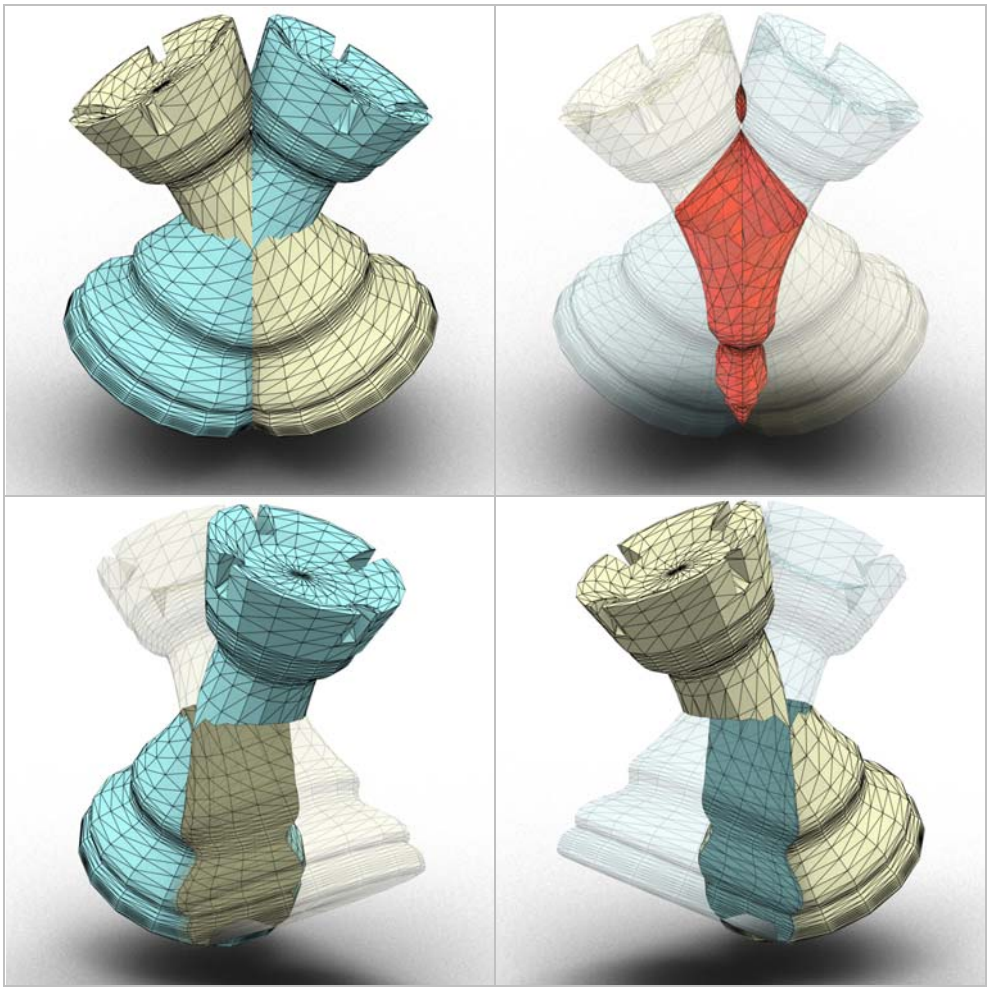


Figura 6-10. De izquierda a derecha y de arriba a abajo: instancias del objeto Torre; intersección; diferencia torre azul-torre amarilla; diferencia torre amarilla-torre azul.

Para la intersección triángulo-triángulo se ha utilizado el algoritmo de Möller [Möller97]. Cuando se producen intersecciones deben introducirse nuevos triángulos en cada malla. Aquí se ha seguido la estrategia de realizar todas las intersecciones de cada triángulo de la malla *A* con todos los de la malla *B* en una sola pasada. El resultado es una nube de puntos coplanarios y siempre contenidos dentro del triángulo inicial. Esto permite aplicar el método de Delaunay [Lischinski94] obteniendo buenos resultados y tratando correctamente cualquier tipo de caso especial en las intersecciones, como triángulos que intersectan en un sólo vértice o en parte de una arista. Además, este enfoque permite evitar triangulaciones intermedias.

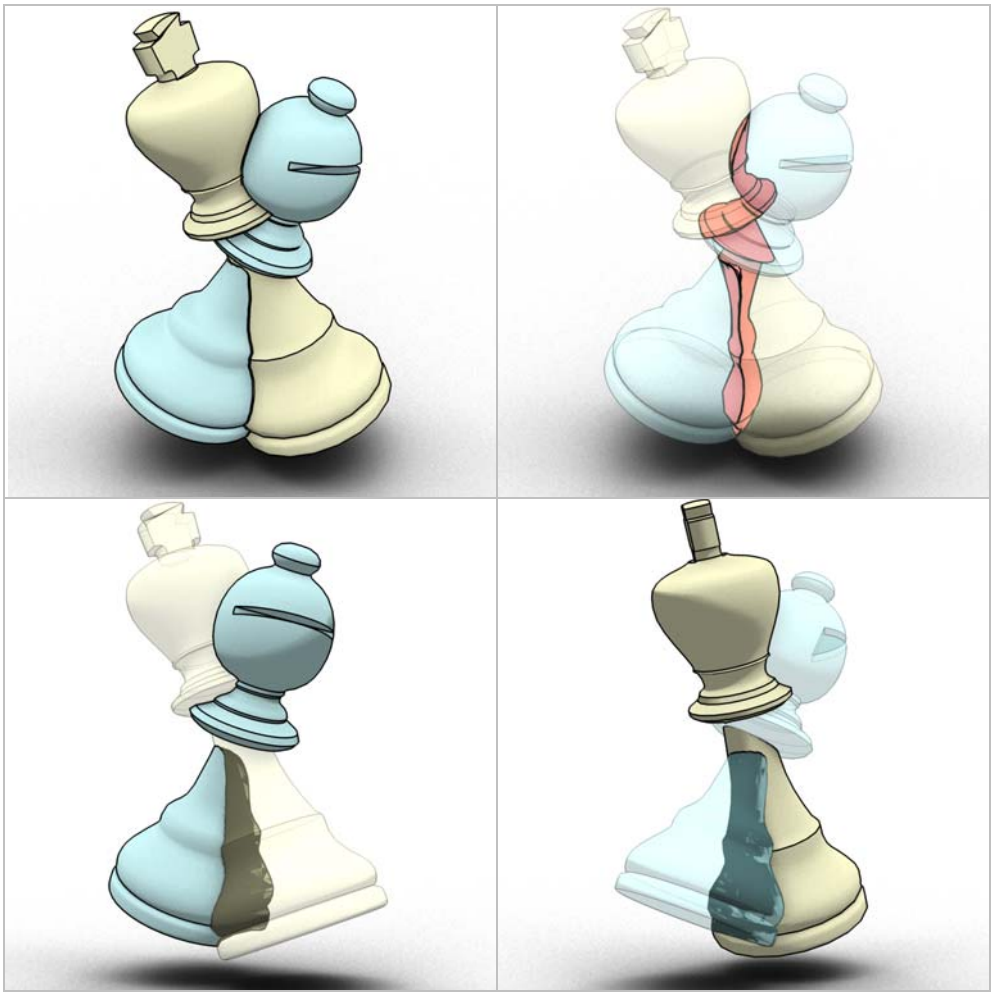


Figura 6-11. De izquierda a derecha y de arriba a abajo: objetos Alfil y Rey situados para realizar las operaciones booleanas; intersección; diferencia Alfil-Rey; diferencia Rey-Alfil.

Para el segundo paso del método de evaluación se utiliza el algoritmo de clasificación de puntos basado en GPU presentado en el Apartado 4.3.3, ya que el test de inclusión triángulo-en-sólido puede reducirse a punto-en-sólido tal y como se explica en las Secciones 6.4 y 6.5. Para comprobar la inclusión de los triángulos se utiliza su baricentro. La precisión utilizada para los cálculos matemáticos es muy importante, ya que hay algunas operaciones críticas para la validez del resultado. La más susceptible de producir imprecisiones es el test de inclusión de puntos, que con resultados muy cercanos a 0 determina si un punto se encuentra exactamente sobre la superficie del sólido. Este caso puede ocasionar problemas cuando hay varias caras de los dos objetos en el mismo plano.

El algoritmo de inclusión de puntos admite una serie de optimizaciones para aumentar su rendimiento, tal y como se presentó en las Secciones 4.3.2 y 4.3.3. Para las pruebas se ha utilizado el preprocesamiento básico, la indexación de tetraedros mediante octantes y la implementación en GPU. La utilización de la indexación mediante segmentación uniforme a través de la voxelización de los sólidos también ofrece buenos resultados, aunque presenta un coste alto en la utilización de memoria.

Para las pruebas se ha utilizado la clasificación de triángulos optimizada, esto es, se calcula la información de conectividad entre triángulos y se agrupan en conjuntos delimitados por las zonas de intersección (ver Apartado 6.5.1). Esta optimización reduce en gran medida el número de tests de inclusión punto-en-sólido que hay que realizar. Con mallas de complejidad baja (hasta ~15000 triángulos) la reducción está entre el 55% y el 65%. Con mallas de complejidad más alta (a partir de 50000 triángulos) la reducción está entre el 85% y el 90%. A partir de 150000 triángulos se ha observado que la reducción llega hasta el 95%. El porcentaje de reducción depende en gran medida de la extensión de las zonas de intersección. En las pruebas se ha procurado colocar los objetos de forma que la cantidad de triángulos a ser teselados fuera la mayor posible.

Cuando los triángulos de las mallas quedan clasificados, la obtención de la estructura resultante de una operación booleana es trivial, ya que basta con copiar desde las dos mallas implicadas los triángulos (y vértices asociados) según cumplan las condiciones expuestas en el Apartado 6.5. Si los triángulos llevan atributos asociados como coordenadas de textura, color, shader, etc., éstos deben ser tenidos en cuenta durante todo el proceso. En la implementación realizada sólo se ha utilizado un color y una normal por triángulo.

Para obtener resultados óptimos se realizan las siguientes operaciones sobre la malla resultante de la operación booleana:

- Puede simplificarse la malla manteniendo la topología, ya que es probable que aparezcan triángulos coplanarios.

- Es necesario realizar una compactación y reindexación de los vértices y los triángulos asociados para obtener un sólido válido. Esto es debido a la copia de triángulos desde dos mallas distintas y a la realización de teselaciones en éstas, por lo que aparecerán vértices repetidos en la estructura de datos. Si se realizan varias operaciones booleanas consecutivas, es más eficiente realizar la compactación sólo después de la última operación. Los vértices duplicados no afectan el desempeño del algoritmo en las operaciones booleanas, tan sólo a la validez del sólido final.

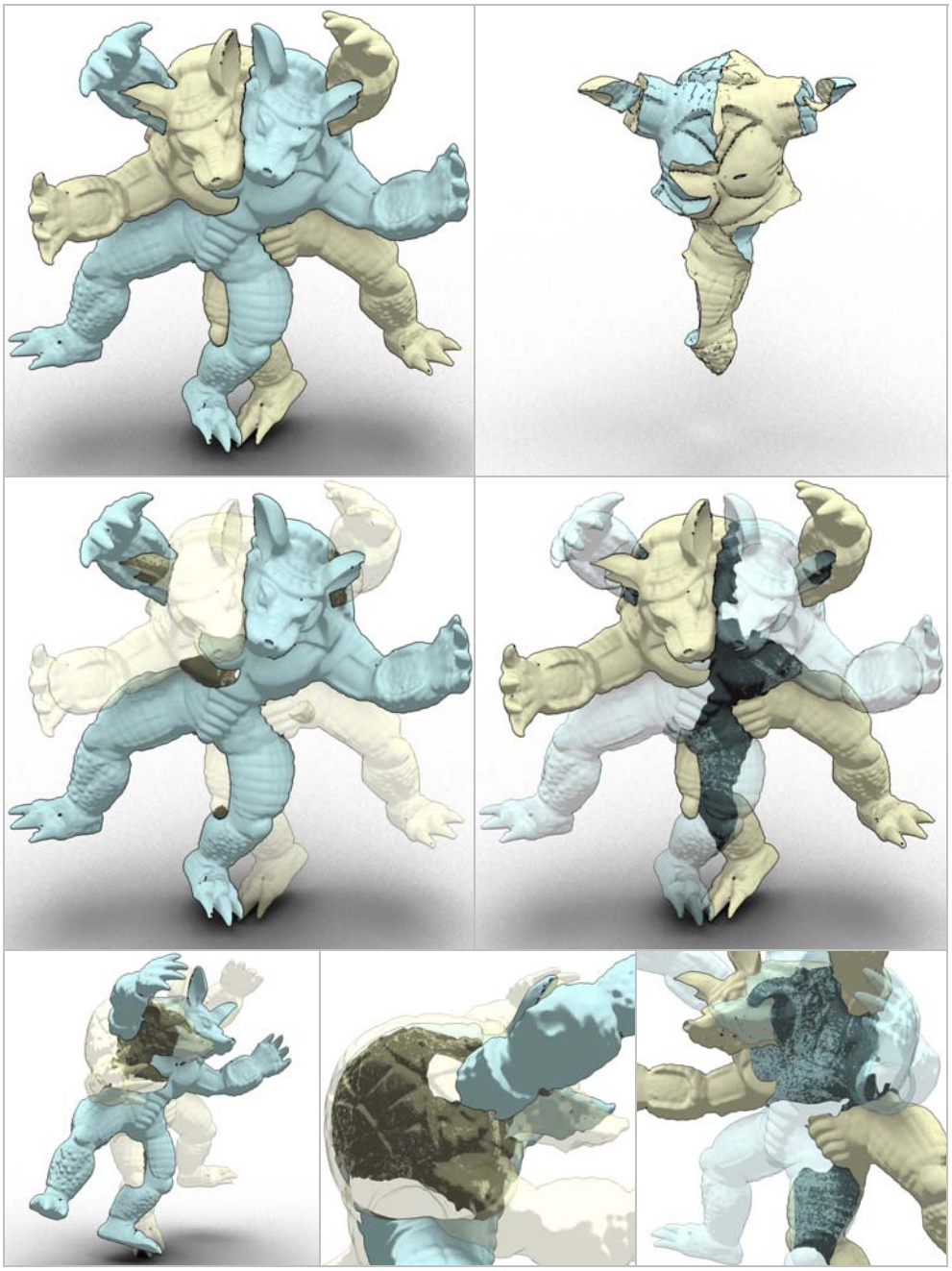


Figura 6-12. De arriba a abajo por filas: instancias del objeto Armadillo y su intersección; diferencias entre instancias; detalles de algunas operaciones.

- Si se utilizan normales por vértice, deberán ser recalculadas al final. Es usual obtener las normales mediante un procesamiento automático que considera un ángulo umbral para suavizar las distintas partes de la superficie del sólido. Si no es el caso, debe conservarse la normal de cada vértice desde el principio y tenerla en cuenta en el momento de compactar vértices.

Para probar la evaluación de operaciones boolanas se han utilizado varias mallas de triángulos de diferente topología y complejidad poligonal. Las láminas 1 a 10 muestran algunas imágenes de los objetos. En el apéndice B se describen los detalles sobre el hardware, el lenguaje de programación utilizado y otros aspectos de implementación. Las figuras 6-9, 6-10, 6-11 y 6-12 muestran algunos ejemplos de operaciones booleanas entre mallas de triángulos utilizando la implementación propuesta.

La Tabla 6-1 muestra los resultados obtenidos en las pruebas. Los tiempos para el algoritmo propuesto abarcan todos los pasos descritos en las secciones anteriores. Además de indicar el total, también se muestran los resultados desglosados para las dos fases principales del algoritmo. Esto se ha hecho así para poder mostrar la incidencia de utilizar el método de inclusión de puntos basado en GPU en la versión optimizada. La primera fase incluye la creación del octree optimizador y el refinamiento de las mallas. La segunda comprende la clasificación de los triángulos y la creación de la estructura de la malla final.

El algoritmo presenta una complejidad lineal que depende del número de polígonos. Hay que tener en cuenta que sin utilizar el octree optimizador en la fase de refinamiento el orden de complejidad sería de $O(n^2)$. El rendimiento del algoritmo de inclusión de puntos basado en GPU es bastante mejor que la versión software, con lo que mejora todavía más los resultados.

Hay que recordar que la versión software es algo mejor que la basada en hardware para mallas poligonales poco complejas, ya que no necesita inicializar buffers ni estructuras. Sin embargo, la influencia de estos factores desaparece cuando aumenta el número de polígonos. La tabla muestra cómo a mayor número de polígonos, mayor es la ventaja de utilizar la GPU en la versión optimizada.

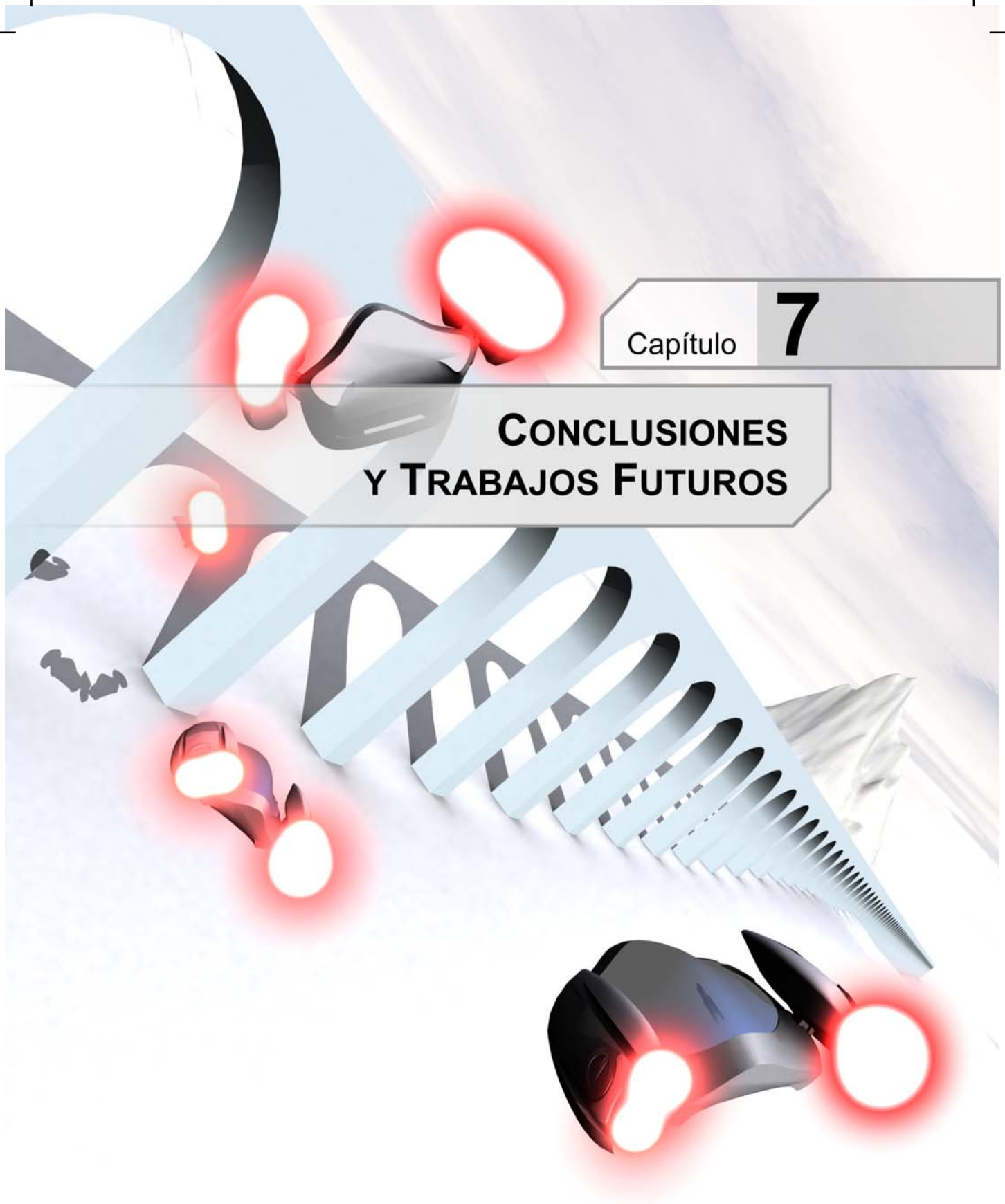
En la Tabla 6-1 se incluye además una comparativa con el operador booleano estándar de 3D Studio Max 8. Como se puede observar, 3DSMax presenta problemas de rendimiento cuando las mallas involucradas en las operaciones booleanas son poligonalmente muy complejas.

	Total de triángulos	Nuevo			Nuevo optimizado			3DS MAX 8
		1ª fase	2ª fase	Total	1ª fase	2ª fase	Total	
Torre ∪ Torre	4512	0,195112	0,668883	0,863995	0,195113	0,039968	0,235081	0,5
Torre ∩ Torre		0,180659	0,685969	0,866628	0,180659	0,039736	0,220395	0,5
Torre – Torre		0,195521	0,676856	0,872377	0,195521	0,040093	0,235614	0,5
Peón ∪ Peón	4800	0,236572	0,864581	1,101153	0,236572	0,179641	0,416213	0,5
Peón ∩ Peón		0,224926	0,863123	1,088049	0,224926	0,176956	0,401882	0,5
Peón – Peón		0,238768	0,859105	1,097873	0,238768	0,178441	0,417209	0,5
Rey ∪ Rey	6428	0,269904	1,131417	1,401321	0,269904	0,173001	0,442905	0,7
Rey ∩ Rey		0,231623	1,105656	1,337279	0,231623	0,165821	0,397444	0,7
Rey – Rey		0,260222	1,085959	1,346181	0,260222	0,171742	0,431964	0,7
Reina ∪ Reina	7184	0,274021	1,038952	1,312972	0,274021	0,136902	0,410922	0,7
Reina ∩ Reina		0,223331	1,046386	1,269716	0,223331	0,135464	0,358794	0,7
Reina – Reina		0,243337	1,024091	1,267428	0,243337	0,136051	0,379388	0,7
Alfil ∪ Alfil	11628	0,533772	3,277029	3,810801	0,533773	0,306708	0,840481	2,3
Alfil ∩ Alfil		0,464169	3,201479	3,665648	0,464169	0,306688	0,770857	2,3
Alfil – Alfil		0,480701	3,256663	3,737364	0,480701	0,300272	0,780973	2,3
Caballo∪Caballo	30700	0,874293	8,815271	9,689564	0,874293	0,869506	1,743799	15,5
Caballo∩Caballo		0,844598	8,721725	9,566323	0,844598	0,873453	1,718051	15,5
Caballo–Caballo		0,849038	8,795241	9,644279	0,849038	0,874447	1,723485	15,5
LionHd∪LionHd	44032	0,628043	20,726558	21,354601	0,628043	0,922698	1,550741	25
LionHd∩LionHd		1,666165	19,060393	20,726558	1,666165	0,924898	2,591063	25
LionHd–LionHd		1,717659	19,304697	21,022356	1,717659	0,925416	2,643075	25
Rey ∪ Alfil	9028	0,374347	1,708604	2,082951	0,374347	0,236069	0,610416	1
Rey ∩ Alfil		0,306443	1,722149	2,028592	0,306443	0,234299	0,540742	1
Rey – Alfil		0,329011	1,741061	2,070071	0,329011	0,234321	0,563331	1
Torre ∪ Reina	5848	0,209749	0,724059	0,933808	0,209751	0,121701	0,331451	0,8
Torre ∩ Reina		0,185785	0,721739	0,907524	0,185785	0,120327	0,306112	0,8
Torre – Reina		0,206651	0,737438	0,944089	0,206651	0,119357	0,326008	0,8
Alfil ∪ Caballo	21164	0,741871	3,593821	4,335691	0,741871	0,482082	1,223952	3,8
Alfil ∩ Caballo		0,585457	3,528766	4,114223	0,585457	0,480052	1,065509	3,8
Alfil – Caballo		0,581894	3,606522	4,188416	0,581894	0,479868	1,061762	3,8
Golfball – Golfball	92410	5,075748	58,612443	63,688191	5,075748	1,674591	6,750339	86
Armadillo – Armadillo	300000	29,394585	665,436772	694,831357	29,394585	73,105584	102,500169	1125

Tabla 6-1. Tiempos en segundos para distintas operaciones booleanas. Las posiciones relativas de los objetos son las mismas para todas las pruebas. La versión optimizada del nuevo algoritmo incluye una implementación en GPU del método de inclusión de puntos. Los tiempos para 3D Studio Max 8 tienen un error $|e| \leq 0.25s$.

6.7 Conclusiones

En este Capítulo se ha presentado un algoritmo eficiente y robusto para la evaluación de operaciones booleanas con mallas de triángulos. El método es bastante flexible y puede implementarse de múltiples formas, ofreciendo un abanico de soluciones adaptables. No obstante, la implementación realizada y presentada ofrece un rendimiento excelente con diversas optimizaciones, además de hacer uso de la GPU para la parte más costosa del proceso. La estructura de datos utilizada para la representación de los modelos es de los esquemas más simples para el modelado de B-Reps, por lo que casi cualquier formato 3D de los existentes puede ser utilizado sin conversiones de datos complejas. Con la futura evolución de la GPU se irán integrando más partes del proceso descrito en el hardware gráfico.



Capítulo 7

Conclusiones y Trabajos Futuros

A lo largo de este trabajo se ha presentado una serie de técnicas, optimizaciones e implementaciones que solucionan varios problemas geométricos relacionados con el modelado de sólidos. Los enfoques empleados utilizan la representación de sólidos mediante recubrimientos simpliciales. Las implementaciones tanto en CPU como en GPU son muy eficientes, ofreciendo ganancias sustanciales en rendimiento a la vez que se lleva la última tecnología hardware al límite.

7

7.1 Conclusiones

A lo largo de este trabajo se han presentado diversos algoritmos basados en la teoría de recubrimientos simpliciales presentada en los trabajos de Feito, Torres y Segura [Feito95b, Feito97, Segura01] que permiten representar objetos gráficos basándose en la descomposición de los mismos en símplices. Esta forma de representación ofrece un nuevo enfoque para la resolución de múltiples problemas. Aunque en trabajos anteriores se habían presentado soluciones a dichos problemas, realmente no se habían desarrollado implementaciones eficientes ni versiones optimizadas. En esta tesis se han propuesto soluciones avanzadas e implementaciones optimizadas para conseguir un buen rendimiento. Además, gracias al auge actual del hardware gráfico programable se pueden implementar los métodos propuestos haciendo uso de las posibilidades de la GPU, ya que estos algoritmos se adaptan muy bien al procesamiento vectorial.

Se ha presentado una solución completa para la inclusión de puntos en sólidos basada en recubrimientos simpliciales. Se ha realizado una serie de implementaciones optimizadas que ofrecen un rendimiento excelente. La implementación en GPU programable es muy eficiente y demuestra la facilidad para adaptar el algoritmo de inclusión de puntos al procesamiento vectorial. Las comparativas realizadas demuestran que el algoritmo propuesto destaca por su rendimiento, siendo la propuesta más eficiente sin preprocesamiento.

Se han desarrollado varias soluciones al problema de la voxelización de sólidos B-Rep mediante el uso de recubrimientos simpliciales. El algoritmo de voxelización presentado permite una implementación eficiente en CPU, en GPU de pipeline fijo y en GPU de pipeline programable. Esta última se muestra muy eficiente y aprovecha al máximo las posibilidades de la GPU actual. También hay que destacar la calidad de los resultados, ya que los métodos alternativos con los que ha sido comparado el algoritmo propuesto presentan problemas más acentuados de robustez y de aliasing.

Por último, se ha presentado un algoritmo eficiente y robusto para la evaluación de operaciones booleanas con mallas de triángulos. La implementación realizada y presentada ofrece un rendimiento excelente con diversas optimizaciones, además de hacer uso de la GPU para la parte más costosa del proceso. La estructura de datos utilizada para la representación de los modelos es de los esquemas más simples para el modelado de B-Reps, por lo que casi cualquier formato 3D de los existentes puede ser utilizado sin conversiones de datos complejas. El método propuesto se muestra eficiente y robusto, y en la mayoría de los casos es mejor que algunas implementaciones comerciales.

Para todos los algoritmos propuestos se han realizado estudios comparativos detallados que incluyen soluciones muy referenciadas en la literatura de Informática Gráfica. En el caso de la evaluación de operaciones booleanas las comparaciones se han llevado a cabo utilizando un software profesional como es 3D Studio Max 8, la última versión en el momento de escribir estas líneas. Hay que destacar que todos los algoritmos implementados, tanto los propios como los ajenos, se han desarrollado dentro del mismo software, en las mismas condiciones y de la forma más eficiente posible.

Se han presentado implementaciones para la GPU programable actual. Se ha explotado la última tecnología gráfica hasta el momento, y aunque todavía no está lo suficientemente evolucionada para algoritmos de propósito general, ofrece sustanciales beneficios en términos de rendimiento. Se espera que el hardware que aparezca próximamente sea mucho más versátil y que permita unas implementaciones de los algoritmos propuestos mucho más naturales y eficientes.

Parte de los temas presentados ya han sido avalados en varias publicaciones de carácter nacional e internacional, tanto congresos como revistas.

7.2 Trabajos futuros

Todavía quedan muchas líneas de investigación relacionadas con este trabajo en las que se puede profundizar más. También surgen nuevas aplicaciones que utilizan las técnicas propuestas, así como ideas sobre temas relacionados con la utilización de la GPU para algoritmos de propósito general. Entre las tareas futuras en las que se pretende trabajar destacan las siguientes:

- Adaptación e implementación del algoritmo de inclusión de puntos en sólidos para mallas de millones de triángulos mediante técnicas de streaming. Este aspecto es importante al tratar sólidos creados mediante escaneado tridimensional y representaciones de superficies muy extensas y detalladas como terrenos.

- Desarrollar una heurística para ajustar el tamaño del optimizador TetraCell que dependa del número de polígonos y la distribución de los mismos. Intentar implementar alguno de los optimizadores usados en el algoritmo de inclusión en GPU.

- Realizar implementaciones y pruebas del algoritmo de inclusión de puntos en sólidos utilizando un cluster de CPUs y GPUs (*Rendering Farm*).

- Probar nuevas estructuras de datos para la GPU programable que incrementen el rendimiento del algoritmo de inclusión de puntos. Probar nuevas extensiones del hardware como *Render-to-Vertex-Buffer*, etc.

- Aplicar técnicas de antialiasing al algoritmo de voxelización mediante implementación de funciones de aspecto. Trasladar el algoritmo de voxelización del espacio discreto al continuo.

- Intentar utilizar la instanciación de geometrías (*geometry instancing*) en el proceso de voxelización en GPU programable. Con esta extensión se despejaría por completo el flujo de entrada de vértices al pipeline gráfico.

- Desarrollar técnicas más específicas para el tratamiento de sólidos heterogéneos con la ayuda de funciones de aspecto en el proceso de voxelización. Estas funciones pueden ser implementadas en la GPU.

- Aplicar el algoritmo de voxelización a otros problemas como la detección de siluetas en sólidos.

- Utilizar técnicas de clustering para evaluar operaciones booleanas con mallas de millones de triángulos. Utilizar la adaptación del algoritmo de inclusión al clustering mencionada anteriormente y mejorar el resto del proceso de evaluación para integrarlo en cluster.

- Intentar trasladar más etapas de la evaluación booleana a la GPU para completar la implementación en hardware.

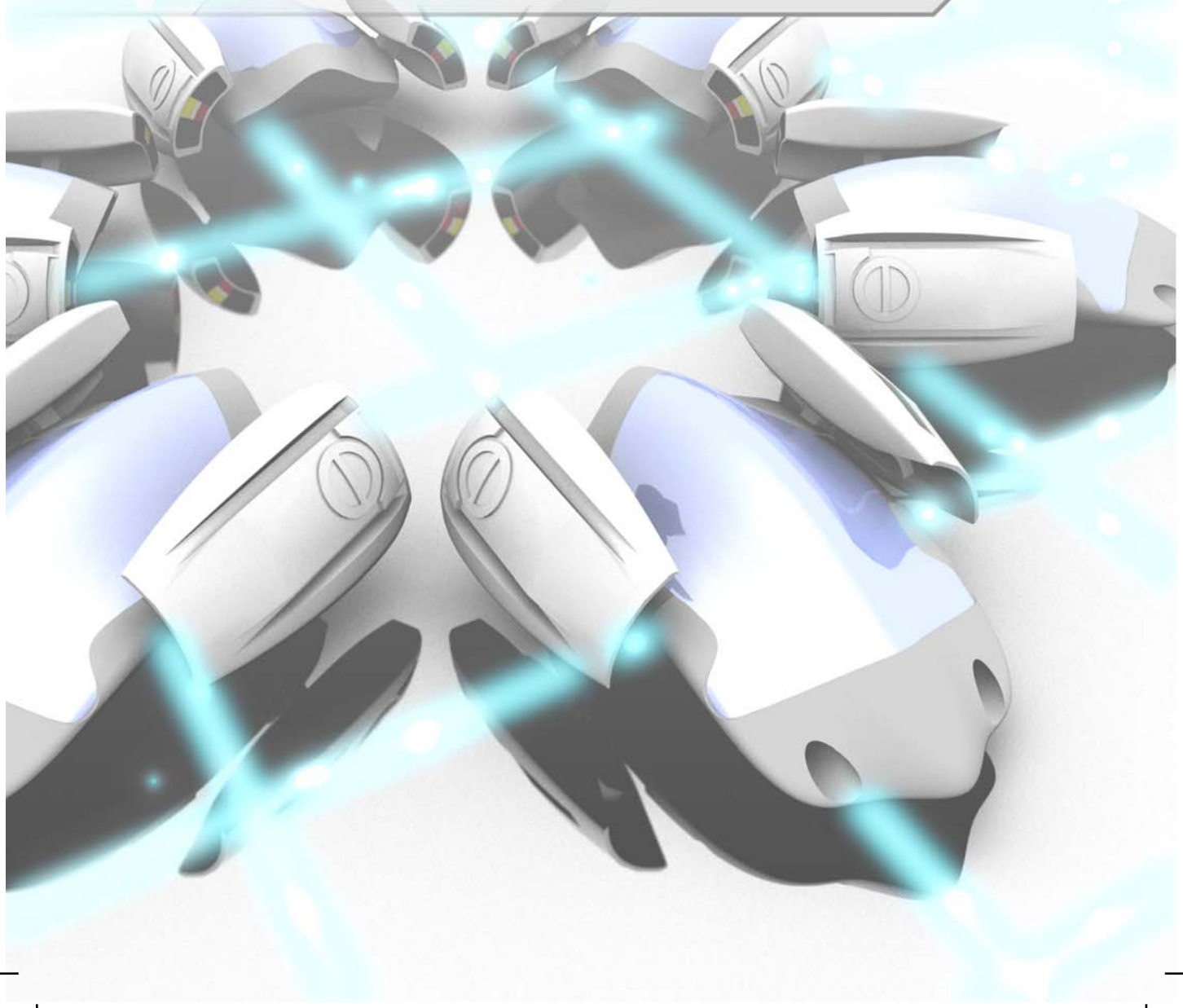
- Adaptación de todos los algoritmos propuestos a las GPUs futuras. Con la evolución del hardware gráfico surgirán nuevas formas de procesamiento que probablemente beneficiarán a los algoritmos que no están directamente relacionados con el pipeline clásico de rendering.

- Adaptación de otros algoritmos de propósito general a la GPU. Con la realización de este trabajo se muestra el beneficio de la implementación en GPU de diversos algoritmos que pueden adaptarse al procesamiento vectorial.

- Realizar una formalización para la adaptación de algoritmos secuenciales al uso del hardware gráfico programable. Parte de esta tarea consiste en paralelizar los algoritmos secuenciales, mientras que la otra parte estaría dedicada a la conversión de estructuras y flujos de datos para ser llevadas a la GPU.

Apéndice **A**

PROGRAMACIÓN DE LA GPU



En este apéndice se presentan algunos conceptos sobre la programación del hardware gráfico actual, ya que muchas de las optimizaciones presentadas se basan en la nueva generación de GPUs programables. Una descripción detallada de este amplio tema de actualidad queda fuera del ámbito de este trabajo, por lo que la información que se expone a continuación es sólo un pequeño resumen acerca de la evolución y del estado actual de la GPU programable.

A

A.1 Introducción

La rápida evolución que se está produciendo en el hardware gráfico está permitiendo en los últimos años un salto cuantitativo y cualitativo en el rendimiento de las aplicaciones gráficas. De una parte, la inclusión de procesadores gráficos cada vez más potentes y capaces de realizar operaciones específicas utilizadas en Informática Gráfica permite la optimización de muchos de los métodos empleados. De otra parte, la posibilidad del desarrollador de añadir programas en el proceso de rendering (*graphics pipeline*) añade la posibilidad de realizar de una manera mucho más rápida algoritmos que han de procesarse para los modelos gráficos [Kilgariff05].

Muchos autores han percibido la presencia de las GPUs (Graphics Processing Unit) en los equipos actuales como una posibilidad para resolver problemas de tipo genérico, o simplemente para paralelizar algunos procesos utilizando de manera conjunta la CPU y la GPU. En este sentido es importante advertir que los procesadores gráficos están diseñados específicamente para resolver problemas de tipo gráfico, por lo que se hace precisa como paso previo una conversión del problema original a términos que puedan ser procesados y aprovechados por este tipo de hardware. Este Capítulo presenta una breve descripción de la evolución de la GPU así como de su estado actual, de los recursos que ofrece y de las posibilidades para la computación general o GPGPU (General Purpose computation on Graphics Processing Units).

Hay que mencionar que suele hablarse de algoritmo hardware o algoritmo implementado en hardware haciendo referencia a un software que se compila y ejecuta en la GPU. En contraposición se denomina algoritmo software (o por software) a un algoritmo que se ejecuta en la CPU. Realmente es una simplificación o abuso del lenguaje muy extendido en Informática Gráfica para distinguir los programas basados en CPU y en GPU. Como es sabido, un algoritmo sólo está implementado en hardware cuando está incluido en un chip a nivel físico. Un algoritmo basado en GPU es simplemente un software que se compila y ejecuta sobre un procesador gráfico.

A.2 Evolución del hardware gráfico

Los procesadores modernos están construidos con millones de transistores. Según avanza la tecnología, estos transistores y la conexión que se establece entre ellos se fabrican en cada vez menos espacio. En 1965 Gordon Moore predijo que el número de transistores que podría fabricarse de forma económica en un sólo procesador se doblaría cada año [Moore65], y que ese crecimiento se mantendría con el tiempo. A esta predicción se la conoce como ***Ley de Moore***. Aunque las referencias a esta *ley* se refieren generalmente al incremento en el rendimiento, la predicción real de Moore se refería solamente al número de dispositivos que podrían incluirse por unidad de superficie en un chip.

Las nuevas generaciones de chips no sólo cuentan con mayor número de transistores, sino que además éstos son de menor tamaño. Debido a esto, pueden operar y comunicarse a mayor velocidad, con lo que el rendimiento global del chip es mayor. Cada año las capacidades de computación de un procesador se incrementan en un 71% respecto de la versión anterior [Owens05].

Aunque el crecimiento en capacidad de computación está asegurado con el paso del tiempo, la clave real para obtener el máximo rendimiento es dedicar el máximo de unidades de computación a la vez a la misma tarea. Esto se consigue con el

paralelismo de varias unidades de procesamiento trabajando al 100% de sus posibilidades. Como es lógico, los recursos son siempre limitados.

Los procesadores de alto rendimiento actuales están pensados para ejecutar aplicaciones de propósito general. Por lo general, este tipo de aplicaciones no tienen tanto paralelismo ni tantas necesidades de rendimiento, aunque sí utilizan más operaciones de control del pipeline de ejecución. Todo esto hace que los procesadores de propósito general se adapten mal al funcionamiento del pipeline gráfico y a otras aplicaciones con características similares. Los modelos de programación de CPU son generalmente en serie y no se adaptan bien al procesamiento paralelo de las aplicaciones. Aunque se han introducido extensiones en las CPUs modernas para aprovechar cierto paralelismo a nivel de instrucción, como SSE de Intel o AltiVec de PowerPC, lo cierto es que el grado de paralelismo alcanzado dista mucho del ofrecido por las GPUs actuales. En cualquier caso, las CPUs no contienen hardware especializado para determinadas funciones concretas, ya que su objetivo es la aplicación de propósito general. La GPU, sin embargo, puede incluir en hardware soporte para múltiples funciones específicas, entre las que se encuentran todo tipo de operaciones matemáticas sobre vectores de datos.

A.2.1 Historia reciente de la GPU

A mediados de los 90 el hardware más rápido del mundo consistía en multitud de CPUs que trabajaban en conjunto para dibujar escenas en una pantalla. El ordenador más complejo dedicado a gráficos estaba compuesto de una docena de chips repartidos entre varias placas. Según pasaba el tiempo y evolucionaba la tecnología de semiconductores, los ingenieros de hardware integraban cada vez más elementos y prestaciones en un sólo chip gráfico.

Actualmente la GPU sobrepasa a la CPU en número de transistores. Por ejemplo, en un Intel Pentium 4 a 2.4 GHz se incluyen 55 millones de transistores, mientras que NVidia utilizó mas de 125 millones en la primera GeForce FX [Fernando03]. Precisamente fue NVidia la que introdujo el término GPU (Graphics Processing Unit) a finales de los 90, en una etapa en la que las siglas VGA (Video Graphics Array) ya no describían la realidad de forma exacta. IBM creó el hardware VGA en 1987. Su principal aportación era la posibilidad de mostrar imágenes con una resolución de 640x480 pixels con 16 colores y de 320x240 pixels con 256 colores. Después aparecerían los estándares XGA y SVGA con mejores prestaciones en resolución y color. En cualquier caso, este tipo de dispositivos actuaban como lo que hoy se podría denominar un framebuffer pasivo. Esto significa que la CPU es la responsable de realizar casi todas las operaciones de dibujado. Hoy en día, la CPU

prácticamente no realiza ninguna operación de bajo nivel, ya que el hardware se encarga de la mayoría de las operaciones tanto en 2D como en 3D.

Los expertos en el sector del hardware identifican hasta cuatro generaciones de GPUs [Fernando03]. Cada una introduce una serie de mejoras que afectan a la flexibilidad de programación, a las prestaciones y al rendimiento en general. La evolución del hardware gráfico moderno se ha producido a la par que las dos interfaces de programación más importantes, OpenGL y Direct3D. OpenGL [Woo97] es un estándar de programación 3D disponible para las plataformas más extendidas. DirectX es una interfaz de programación multimedia que incluye Direct3D para la programación 3D [Kovach00].

Primeros sistemas gráficos en hardware

Antes de la llegada de la GPU como tal, compañías como Silicon Graphics (SGI) y Evans & Sutherland (E&S) diseñaron hardware específico muy costoso orientado a gráficos. En esta etapa se introdujeron una serie de conceptos fundamentales como la transformación de vértices y el mapeado de texturas. Estos sistemas fueron muy importantes en el desarrollo de la Informática Gráfica, pero debido a su elevado coste sólo estaban presentes en un mercado altamente especializado. Hoy en día, las GPUs corrientes son mucho más potentes que cualquiera de aquellos sistemas.

Primera generación de GPUs

La primera generación abarca hasta 1998, incluyendo tarjetas como NVidia TNT2, la gama ATI Rage y la 3dfx Vodoo3. Estas GPUs eran capaces de rasterizar triángulos pre-transformados y aplicar una o dos texturas de forma simultanea. En este punto, el hardware gráfico se encargaba de las tareas de rasterización casi por completo (las más costosas), liberando a la CPU de parte del pipeline. De hecho, las primeras placas gráficas orientadas a 3D tan sólo implementaban por hardware el dibujado de triángulos, como la Graphics Blaster de Creative, la PowerVR o la 3dfx Vodoo1 de 1996. Aunque en esta etapa la rasterización por hardware empezaba a ser eficiente, tan sólo se podían aplicar una serie de funciones fijas para combinar texturas y controlar algunos aspectos de la rasterización. El aspecto que más sorprendía a los usuarios de esta época era el filtrado lineal de texturas, que permitía imágenes con una calidad muy superior a las generadas por software.

Segunda generación de GPUs

Esta etapa incluye las GPUs lanzadas entre 1999 y 2000, como NVidia GeForce 256, GeForce 2, ATI Radeon 7500 y S3 Savage3D. Estos chips implementaban la transformación e iluminación de vértices por hardware, lo que se conocía como

Figura A-1. Detalle de una GPU GeForce 7800GTX.

hardware transform & lighting (T&L). La transformación de vértices en hardware era hasta el momento la gran diferencia entre las estaciones de trabajo profesionales y el hardware de consumo. Aunque esta generación es más flexible al introducir texturas cúbicas y operaciones raster adicionales, todavía es muy limitada. Esta es la generación que más se adapta al modelo de rendering de pipeline fijo.

Tercera generación de GPUs

En esta etapa, que comprende los años 2001 y 2002, se empieza a adaptar el hardware gráfico al modelo de rendering de pipeline programable. GPUs importantes de esta época son NVidia GeForce 3 y GeForce 4 Ti, ATI Radeon 8500 y la consola XBox de Microsoft. Además de ofrecer T&L por hardware para mantener la compatibilidad, las GPUs ofrecían la posibilidad de programar a nivel de vértices. Sin embargo, aunque se implementaron más funciones de combinación y procesamiento de pixels, no era posible programar a nivel de fragmentos y el control del flujo de ejecución era fijo, por lo que esta generación es sólo de transición hacia el pipeline completamente programable.

Cuarta generación de GPUs

La cuarta y actual generación de GPUs (desde 2002) incluye productos de la familia NVidia GeForce FX (con arquitectura CineFX) y los basados en ATI Radeon 9700, llegando a modelos más evolucionados como ATI Radeon X850 y NVidia GeForce 7 (Figura A-1 y Figura A-2). En esta ocasión el hardware gráfico ofrece la posibilidad de programar a nivel de vértices y de fragmentos (posteriormente se define este concepto), con lo que puede tomarse el control de las partes más importantes del pipeline. Además el control del flujo de ejecución en esta generación es dinámico, lo que permite más control sobre los algoritmos. A partir de este momento es cuando es interesante el

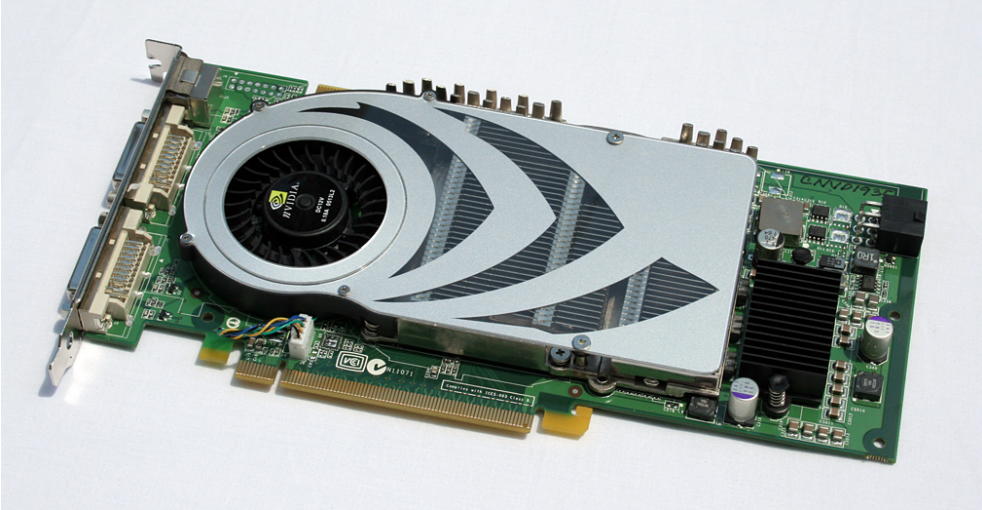


Figura A-2. Detalle de una tarjeta aceleradora 3D GeForce 7800GTX. Lanzada al mercado en el verano de 2005.

uso de la GPU para programación a alto nivel, ya sea de métodos de visualización o de algoritmos de propósito general. Las nuevas GPUs introducen más prestaciones y posibilidades de programación cada año, sin embargo, los fundamentos siguen siendo los mismos que el resto de GPUs de esta generación.

A.2.2 Procesamiento vectorial

Una de las razones por las que la CPU no se adapta bien a las aplicaciones gráficas de alto rendimiento es su modelo de programación en serie, que no soporta el paralelismo y los patrones de comunicación necesarios para este tipo de software. El procesamiento vectorial o basado en flujo (*stream programing model*) en cambio sí se adapta a las estructuras de esta clase de programas de forma que ofrece una gran eficiencia en la computación y la comunicación [Kapasi03]. Este modelo de computación es la base para la programación de la GPU moderna.

En el modelo de programación vectorial todos los datos son representados como un flujo de datos (*stream*), el cual se define como un conjunto ordenado de datos del mismo tipo. Este tipo puede ser desde un entero hasta una matriz de transformación compuesta por números en coma flotante. Las operaciones sobre flujos

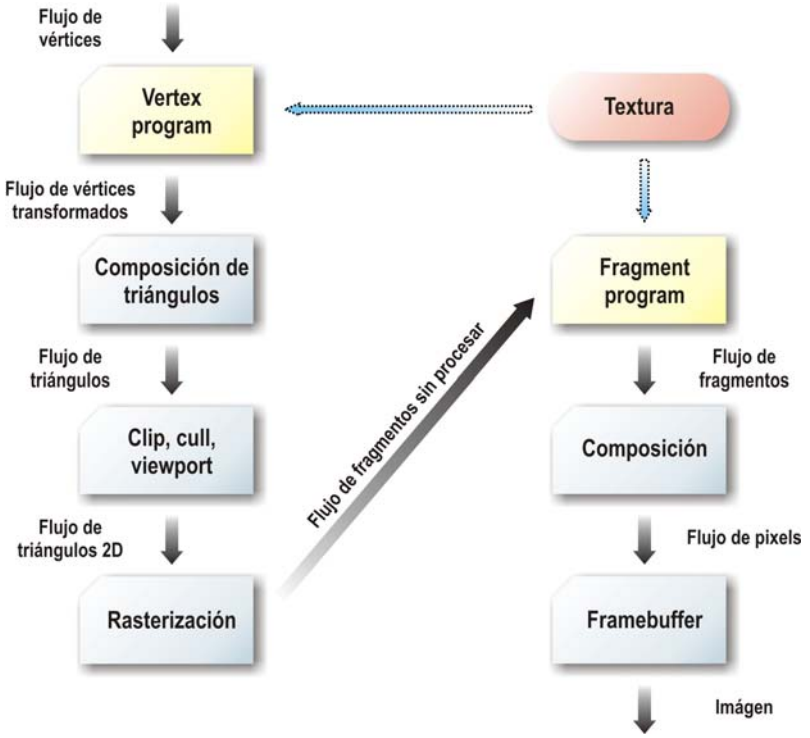


Figura A-3. Adaptación del pipeline gráfico al modelo de procesamiento vectorial (Streaming). La formulación vectorial trata todos los datos como flujos y los núcleos de computación como kernels (indicados como cajas).

de datos muy grandes serán más eficientes que en el modelo en serie, ya que el coste por operación se reduce.

Por cada dato del flujo de entrada opera un *kernel*. Puede considerarse el kernel como un operador que actúa sobre cada dato en forma de función, aunque realmente puede contener un algoritmo genérico. Mediante la utilización del kernel sobre un flujo de datos se está ejecutando su código con todos los datos de dicho flujo. La característica principal de esta forma de procesamiento es que los datos son procesados de forma individual entre ellos, con lo que puede alcanzarse un alto grado de paralelismo mediante la dedicación de múltiples unidades de computación a la ejecución de un kernel sobre un flujo de datos.

El pipeline gráfico se adapta muy bien al procesamiento vectorial. Esto se debe a que se divide en varias etapas de procesamiento conectadas por flujos de datos. Esta

estructura es análoga a un flujo de entrada sobre el que opera un kernel. Los flujos de datos entre etapas del pipeline gráfico están muy localizados; siempre como salida de una etapa y como entrada de la siguiente. Las operaciones que se realizan en cada etapa suelen ser uniformes para un conjunto de primitivas, por lo que pueden codificarse como kernels. La Figura A-3 muestra la especificación del pipeline gráfico como un modelo vectorial.

A.2.3 Naturaleza vectorial de la GPU

El modelo de programación vectorial especifica los algoritmos de forma que aprovecha el paralelismo. La conversión de los programas a la filosofía basada en flujo de datos es sólo parte de la solución. El hardware gráfico de alto rendimiento debe aprovechar la eficiencia aritmética y de computación que permite el enfoque vectorial.

Para construir una GPU de alto rendimiento debe trasladarse cada kernel del pipeline gráfico a una unidad funcional independiente en un chip [Owens02]. Cada kernel es considerado, por tanto, como paralelo a nivel de tarea., ya que todos ellos pueden funcionar de forma simultánea sobre datos distintos. Además, pueden tener funciones distintas y, ya que están situados en el mismo chip, los costes de comunicación son mínimos al pasar de una etapa a otra.

Dentro de cada fase del pipeline gráfico, los kernels correspondientes pueden procesar múltiples flujos simultáneamente. La combinación del paralelismo a nivel de tarea junto con el paralelismo a nivel de datos permite a la GPU sacar provecho de docenas de unidades funcionales de forma simultánea. La entrada al pipeline gráfico debe procesarse secuencialmente por cada kernel. Esto hace que pasen miles de ciclos hasta que se complete el proceso del pipeline para cada elemento. Si hacen falta accesos de alta latencia a memoria (accesos lentos) cuando se procesa un elemento, la unidad de procesamiento puede dedicarse a trabajar con otros elementos hasta que los datos requeridos estén listos. Los cauces tan profundos de la GPU permiten operaciones de alta latencia.

Durante muchos años los kernels del pipeline gráfico se implementaban como funciones fijas que ofrecían poca o ninguna libertad al programador. Por primera vez, en el 2000, las GPUs permitían al usuario programar kernels personalizados en el pipeline gráfico. La GPU moderna implementa dos kernels programables de alto rendimiento que trabajan en paralelo: un programa de vértices (*vertex program*) y un programa de fragmentos (*fragment program*). El vertex program permite ejecutar un algoritmo por cada vértice que pasa a través del pipeline gráfico. El fragment program (también llamado *pixel program*) permite operar sobre cada fragmento que terminará convirtiéndose en un pixel del buffer resultado. A los programas de vértices y

Figura A-4. Pipeline de renderizado en la GPU moderna. Este esquema básico describe tanto a la GPU de pipeline fijo como a la de pipeline programable.

fragmentos también se les conoce como *shaders*, debido a la herencia de RenderMan [Upstill89] y OpenGL en el diseño del pipeline gráfico moderno. Estas novedades estaban originalmente pensadas para ofrecer más posibilidades de sombreado e iluminación en el rendering. Sin embargo, la capacidad de ofrecer altas tasas de procesamiento con programas definidos por el usuario y con suficiente precisión, convierte a la GPU en un procesador vectorial programable. Esto hace que el hardware gráfico sea de gran utilidad para ejecutar una extensa variedad de aplicaciones que van más allá del proceso clásico de rendering.

A.3 Pipeline de rendering

El proceso completo de visualización de una escena 3D se conoce como *pipeline de rendering*. Hay muchos tipos de renderizado según la tecnología que se utilice: ray-tracing, Z-buffer, etc., e incluso soluciones mixtas. El tipo que se presenta aquí es el implementado en el hardware gráfico actual, y que está basado en el algoritmo Z-buffer y en una parte de la especificación de la librería OpenGL [Woo97], cuyo sistema de rendering en tiempo real ha sido el más popular desde su aparición. Las distintas etapas en las que puede dividirse el pipeline de rendering pueden agruparse en tres grandes grupos funcionales: el procesamiento geométrico, la rasterización y las operaciones finales sobre pixels. La forma en que se implementa el pipeline de rendering depende del hardware y en cierta medida del software que lo utiliza.

La Figura A-4 muestra el pipeline del hardware gráfico actual. La aplicación 3D envía a la GPU una secuencia de vértices y una secuencia de primitivas geométricas, normalmente puntos, líneas y polígonos. Cada vértice tiene una posición en el espacio y una serie de atributos como color, coordenadas de textura, vector normal, etc. Hay que recordar que el vector normal indica la dirección perpendicular a la superficie en el vértice asociado, y suele utilizarse para cálculos de iluminación.

A.3.1 Procesamiento geométrico

La etapa de procesamiento geométrico se encarga de calcular las transformaciones lineales de los vértices que componen los elementos geométricos de la escena. Esto incluye las rotaciones, traslaciones, escalados y proyecciones de los vértices. En esta etapa se incluyen cálculos adicionales como la iluminación a nivel de vértice. A esta etapa se la conoce como de transformación e iluminación o T&L (*Transform and Lighting*). Los pasos ordenados son: creación de primitivas, transformación de entidades, transformación de perspectiva, transformación de la vista, recorte de primitivas con el volumen de visión e iluminación.

A.3.2 Rasterización

En esta etapa se obtiene una lista de fragmentos por cada primitiva enviada al pipeline gráfico y que utiliza vértices ya procesados en la fase anterior. Cada fragmento ocupa una posición en el buffer donde se almacena la imagen resultado, y puede considerarse como un futuro pixel. Cada primitiva (puntos, líneas y triángulos principalmente) ocupa una serie de posiciones en el buffer y genera los correspondientes fragmentos. Para cada uno de ellos hay asociadas propiedades como colores, profundidad, coordenadas de texturas, etc., que son obtenidas a partir de los datos de los vértices de la primitiva mediante interpolación.

A.3.3 Operaciones raster

La etapa final del proceso se encarga de operar sobre los fragmentos para calcular las propiedades de los pixels que componen el resultado del rendering. Estas operaciones son una parte estándar de OpenGL y Direct3D. Durante esta fase se eliminan las superficies ocultas mediante la comparación de profundidad (*depth test*). Entre las operaciones adicionales destacan *scissors test*, *alpha test*, *stencil test*, *blending*, *dithering* y diversas operaciones lógicas [Woo97].

Las operaciones realizadas durante esta etapa modifican las propiedades de cada fragmento, como el color, la profundidad, la posición del pixel final y otros valores asociados. Si algunos de los tests falla, como el de profundidad o el de recorte, el fragmento es descartado y no se genera un pixel en el framebuffer. Funciones como el blending, el dithering o las operaciones lógicas implican una modificación de la imagen en función de los valores acumulados en este. Tras esta etapa, cada fragmento que ha pasado los tests pertinentes genera un pixels que es almacenado en la imagen final (framebuffer).

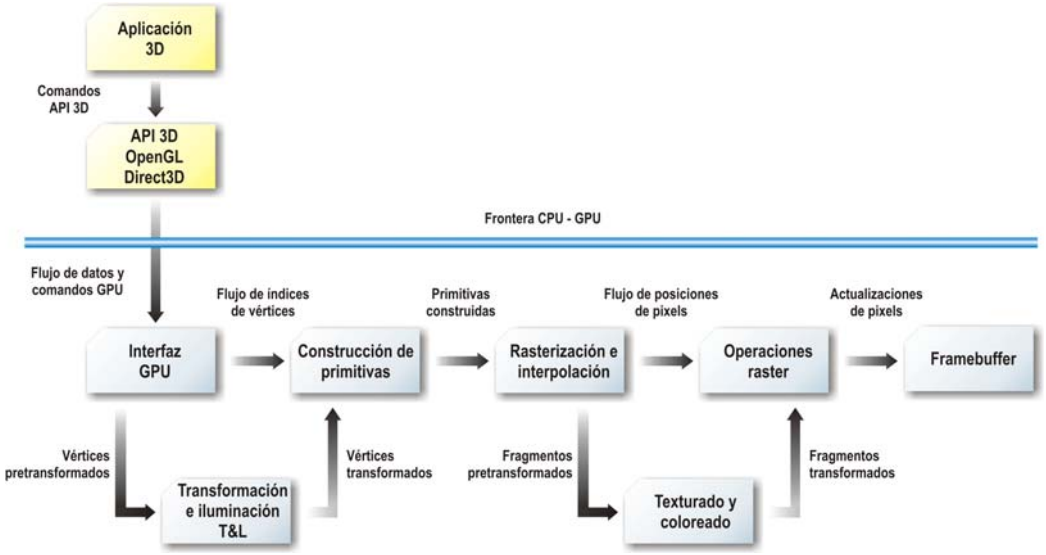


Figura A-5. Organización de la GPU de pipeline fijo.

A.4 GPUs de pipeline fijo

La GPU de pipeline fijo se corresponde con las primeras generaciones de hardware gráfico. En un principio se buscaba la implementación del pipeline clásico de OpenGL (y Direct3D) en la GPU. Este tipo de pipeline incluye los primeros métodos de transformación e iluminación que se utilizaban en dichas librerías gráficas. El modelo de iluminación del pipeline fijo es el de Phong [Foley96]. En cuanto al procesamiento de fragmentos, se permite parametrizar las funciones de texturado y combinación de colores, pero éstas son un conjunto predeterminado en el hardware.

A

En la Figura A-5 se muestra la organización del pipeline fijo en GPU. Las unidades de transformación e iluminación, así como la de texturado y coloreado son parametrizables pero no programables. Las primeras generaciones de GPUs buscaban la implementación de cada vez más etapas de este pipeline en el hardware. Una vez que se contaba con una implementación completa en las GPUs de segunda generación, los fabricantes de hardware comenzaron a buscar un modo de aumentar la flexibilidad del pipeline llegando al enfoque de pipeline programable.

A.5 GPUs de pipeline programable

La corriente dominante hoy en día es dotar de más capacidades de programación a la GPU. Aunque se mantiene la compatibilidad con el pipeline fijo, a veces mediante emulación, el futuro está reservado en exclusiva a la GPU programable. Hay que recordar que utilizando el pipeline programable pueden conseguirse los mismos resultados que con el pipeline fijo, esto es, la funcionalidad implementada en el hardware de pipeline fijo puede trasladarse al pipeline programable usando los correspondientes shaders. La Figura A-6 muestra la estructura del pipeline programable en la GPU actual. Es interesante destacar la diferencia con la Figura A-5. Las unidades de T&L y de texturado son sustituidas por los procesadores programables de vértices y fragmentos.

A.5.1 El procesador de vértices programable

El procesador de vértices se encarga de ejecutar los vertex shaders o programas de vértices sobre el flujo de datos que recibe. Es la primera fase programable del pipeline y se corresponde con las primeras etapas del pipeline fijo clásico, entre las que destaca la transformación e iluminación de vértices. No obstante, en la GPU programable pueden realizarse más acciones, incluyendo las relacionadas con la programación de algoritmos genéricos.

El flujo de vértices que llega al procesador de vértices contiene datos como posición, colores, coordenadas de textura, etc. El procesador de vértices ejecuta el kernel activo, esto es, el vertex shader sobre cada vértice del flujo de forma paralela. Cada vértice circula por un cauce al ser procesado, por lo que se opera sobre tantos vértices a la vez como cauces hay disponibles. El resultado del programa de vértices es de sólo lectura y pasa a la siguiente fase del pipeline. Este resultado incluye la nueva posición del vértice y la modificación de los atributos del mismo. Estos valores son utilizados en la etapa de rasterización (no programable) para interpolarlos en cada primitiva y obtener los atributos de los fragmentos correspondientes que serán enviados al procesador de fragmentos.

A.5.2 El procesador de fragmentos programable

El procesador de fragmentos ejecuta los pixel shaders o programas de fragmentos sobre el flujo de datos de entrada. Este procesador cuenta más o menos con las mismas capacidades de cálculo matemático que el procesador de vértices. Sin embargo, al estar orientado a la generación de pixels, ofrece mucha más flexibilidad en

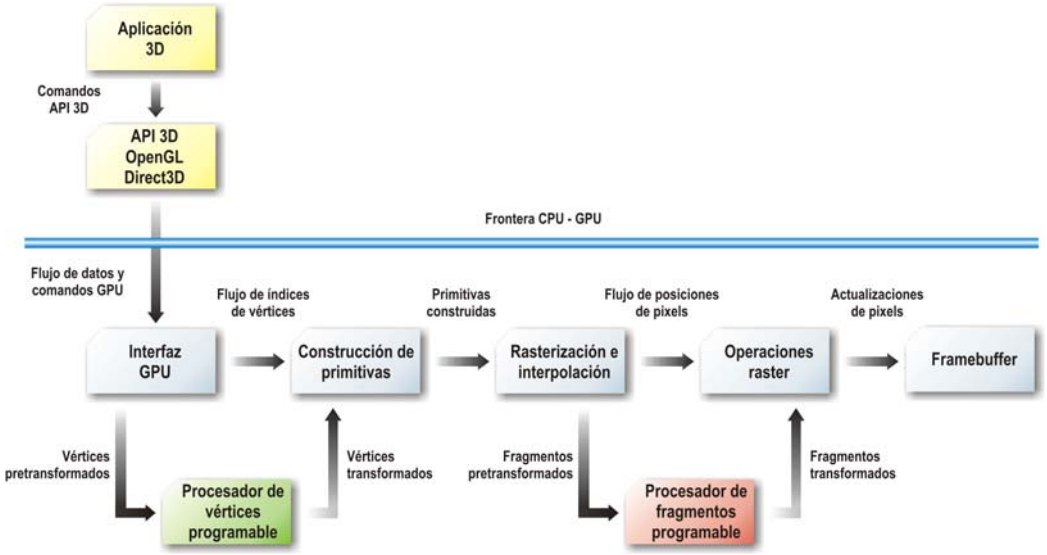


Figura A-6. Organización de la GPU de pipeline programable. Los procesadores programables de vértices y fragmentos permiten la integración de shaders en el proceso de rendering.

el tratamiento de texturas. En esta etapa se permite un gran cantidad de operaciones de lectura de texturas, incluyendo métodos de interpolación y filtrado.

Las GPUs modernas ofrecen soporte para operaciones en coma flotante en todo el pipeline, y en la fase de procesamiento de fragmentos se pueden utilizar texturas y framebuffers en coma flotante. Esto permite la implementación de muchos algoritmos de propósito general, ya que no es necesario hacer conversiones en el formato de los datos. Sin embargo, este aspecto es perjudicado por la imposibilidad de la GPU actual para realizar bifurcaciones arbitrarias en la ejecución de los shaders. Se espera que esta limitación desaparezca en el futuro.

El procesador de fragmentos suele contar con un número bastante mayor de cauces que el procesador de vértices, ya que por norma general se procesan muchos más fragmentos que vértices. Hay que recordar que no todos los fragmentos terminan generando un pixel visible en la imagen final, ya que una buena parte es descartada en el proceso.

A.6 Sistemas de rendering

Hay dos componentes fundamentales que se utilizan para acceder al hardware gráfico desde una aplicación: la interfaz de programación 3D y los lenguajes de sombreado. Algunas interfaces fueron diseñadas al comienzo del desarrollo del hardware gráfico mientras que otras lo hicieron cuando éste ofrecía las prestaciones suficientes para basar productos de calidad en el mismo. Los lenguajes de sombreado fueron los impulsores de la integración de procesadores programables en la GPU, ya que sirvieron como base a un nuevo enfoque que cambiaría el pipeline fijo. Estos lenguajes han evolucionado al mismo tiempo que el hardware, siendo la influencia entre ambos determinante para sus respectivos desarrollos. La Figura A-7 muestra los flujos de datos entre aplicación, API 3D y shaders.

A.6.1 Lenguajes de sombreado

Para la programación a alto nivel de los procesadores de la GPU se han desarrollado algunos lenguajes que facilitan la implementación de los métodos de iluminación y sombreado utilizados en el proceso de visualización. Estos lenguajes están basados en su mayoría en conceptos utilizados anteriormente en diversos sistemas de rendering. La mayoría parte de lenguajes como el utilizado en RenderMan y otros desarrollados en el entorno académico. Las soluciones de alto nivel actuales se adaptan además a las APIs más importantes como OpenGL y Direct3D para sacar el máximo partido al hardware gráfico programable.

El RenderMan Interface Standard describe el mejor lenguaje de sombreado conocido para el sombreado no interactivo. Pixar desarrolló este lenguaje a finales de los 80 para generar animaciones por ordenador de alta calidad para anuncios y películas. Esta empresa creó todo un sistema de rendering compuesto por múltiples componentes, entre los que se encuentra su famoso lenguaje de sombreado [Upstill89].

Shade Trees

El lenguaje de sombreado de RenderMan parte de una idea anterior llamada *shade trees* [Cook84]. Un árbol de sombreado organiza varias operaciones de sombreado en los nodos de una estructura en forma de árbol. Los nodos hoja son las entradas y los nodos intermedios las operaciones. Los investigadores de Pixar se dieron cuenta que un árbol de sombreado es una especie de programa. Esta fue la base de su sistema de sombreado.

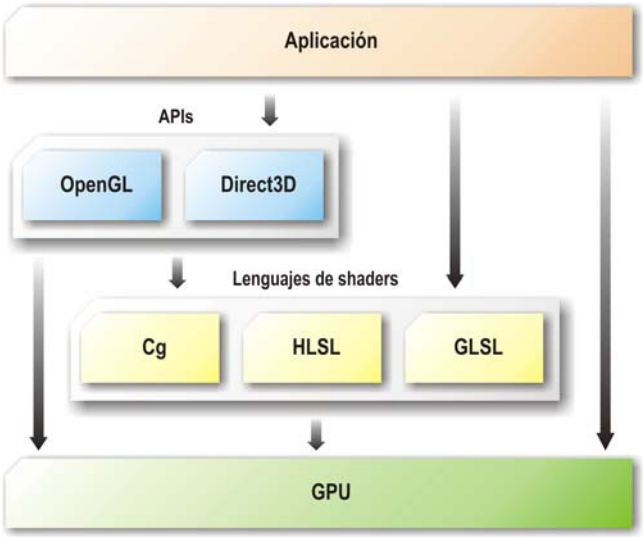


Figura A-7. Flujos de información principales entre los sistemas de una aplicación 3D.

El lenguaje de sombreado de RenderMan

Basándose en los árboles de sombreado, este lenguaje ofrece todo el control sobre el aspecto de las superficies dibujadas. En la búsqueda del fotorrealismo es imprescindible contar con métodos eficientes para controlar cada paso del rendering. Los sistemas de rendering no interactivos más extendidos se basan en algún tipo de lenguaje para el sombreado. El que ofrece RenderMan es el más popular y fue muy extendido y mejorado a finales de los 90 [Upstill89].

Lenguajes adaptados al hardware gráfico

La implementación hardware de un algoritmo es más eficiente cuando se divide en varias etapas formando un pipeline. El enfoque basado en vértices y fragmentos es muy fácil de implementar en hardware. Sin embargo, el algoritmo utilizado en RenderMan no es propenso a ser trasladado al hardware debido a su tratamiento geométrico de alto nivel.

En la Universidad de North Carolina (UNC) se investigó a mediados de los 90 una arquitectura hardware de gráficos llamada PixelFlow, que por ser muy costosa no tuvo éxito. Al mismo tiempo, en Silicon Graphics se seguía un camino distinto para conseguir lo mismo. Esta vez se utilizaba un sistema de rendering basado en OpenGL con múltiples pasadas para la implementación de shaders.

Figura A-8. Integración de los shaders en el pipeline de rendering.

En la Universidad de Stanford un grupo de investigadores, entre los que se encontraba Bill Mark, empezaron a crear un lenguaje de sombreado diseñado específicamente para ser utilizado con GPUs de segunda y tercera generación [Proudfoot01]. El lenguaje de sombreado en tiempo real de Stanford podía compilarse en varias pasadas de rendering basadas en OpenGL.

La investigación realizada en Stanford llevó a NVidia a desarrollar conjuntamente con Microsoft un lenguaje de sombreado pensado para el hardware gráfico comercial. Bill Mark se unió a NVidia en 2001 para llevar este proceso hasta la creación de Cg [NVidia04]. Microsoft hizo lo propio con HLSL [Peeper03] al lanzar DirectX 9 en 2002, y en el verano de 2004 apareció junto con OpenGL 2.0 el lenguaje de sombreado GLSL [Tatarchuk05]. La Figura A-8 muestra la integración de los shaders en el pipeline [Zeller04].

A.6.2 Interfaces de programación

En los primeros días de los gráficos 3D en PC, antes de la llegada de las GPUs, la CPU se encargaba de todas las transformaciones de vértices y las operaciones con pixels para renderizar una escena 3D. Los programadores tenían que implementar todos los algoritmos de dibujado en software. Aunque todo era completamente programable, la CPU era incapaz de procesar incluso los efectos más sencillos en tiempo real. Hoy en día, los algoritmos de rendering están integrados en GPU, y se utilizan interfaces para su programación. Las dos APIs (Application Programming Interface) más extendidas son OpenGL y Direct3D.

OpenGL

A primeros de los 90, Silicon Graphics desarrolló OpenGL en coordinación con la organización OpenGL ARB (*Architecture Review Board*), que incluía a los fabricantes más importantes de hardware gráfico. Originalmente OpenGL sólo funcionaba en estaciones de trabajo Unix. Microsoft, uno de los miembros del ARB, hizo una implementación, primero para Windows NT y luego para el resto de sus sistemas operativos. OpenGL también está disponible para Apple Macintosh, y los usuarios de Linux pueden hacer uso de la implementación de código abierto Mesa. Esta flexibilidad hace que OpenGL sea la mejor interfaz multiplataforma para gráficos 3D.

En la última década, OpenGL ha evolucionado junto con el hardware gráfico. La interfaz es extensible, lo que significa que puede añadirse funcionalidad de forma incremental. Hoy en día, ofrece extensiones para trabajar con programas o shaders para vértices y fragmentos. Con la aparición de la versión 2.0 de la interfaz, se introdujo el lenguaje de sombreado de alto nivel GLSL (*GL Shading Language*), que permite la programación de shaders a alto nivel [Tatarchuk05].

Direct3D

Microsoft comenzó a desarrollar Direct3D sobre el año 1995 como parte de la librería multimedia DirectX. Estas interfaces fueron creadas para ocupar parte del mercado de consumo 3D, en concreto el de Windows PC. La consola de Microsoft Xbox también ofrece soporte para Direct3D. Esta interfaz es la más popular en el entorno Windows debido a su historial de acercamiento a las prestaciones del hardware gráfico.

Cada año más o menos Microsoft actualiza DirectX, incluyendo Direct3D, para aprovechar al máximo las nuevas capacidades de la GPU. En este momento está disponible DirectX 9, que incluye HLSL [Peeper03], el lenguaje de alto nivel para programación de shaders que comparte sintaxis con Cg.

A.7 Recursos de la GPU programable para GPGPU

En esta sección se presentan algunos de los recursos que son de utilidad para la implementación de algoritmos de propósito general. La programación en GPU puede llegar a resultar ardua sin algunos conocimientos sobre gráficos. A continuación se presentan algunas ideas y equivalencias con la CPU que facilitan la tarea. Se supone a partir de ahora que el software que se implementa sobre la GPU es de propósito general, y no el relacionado con el rendering tradicional.

A.7.1 Procesadores programables

La GPU actual tiene dos tipos de procesadores programables: el de vértices y el de fragmentos (ó pixels). El procesador de vértices se encarga de procesar flujos de vértices compuestos de posición, colores, vectores normales y otros atributos. Estos vértices componen las primitivas geométricas que van a dibujarse, como líneas, triángulos, etc. Lo más habitual en rendering en tiempo real es tratar con mallas de triángulos para representar la mayoría de los objetos de una escena típica. El procesador de vértices aplica un programa de vértices (*vertex program* ó *vertex shader*) para transformar cada vértice basándose en su posición respecto de la cámara y en otros aspectos. Cuando los vértices de una primitiva han sido transformados, se generan los fragmentos correspondientes a dicha primitiva. Por ejemplo, un triángulo transformado a coordenadas de ventana se convierte en un triángulo 2D que ocupa una serie de puntos en la imagen resultado. Cada uno de estos puntos genera un fragmento, que puede ser considerado como un *futuro pixel*. Contiene toda la información necesaria para generar un pixel en la imagen final, incluyendo color, profundidad y posición en el buffer final. El procesador de fragmentos ejecuta un programa de fragmentos (*fragment program* ó *pixel shader*) sobre cada elemento del flujo de datos para obtener el color final del pixel correspondiente.

Los procesadores de vértices se encargan principalmente de cambiar la posición de los vértices que reciben como entrada, aunque también pueden alterar otros atributos asociados. El procesador de vértices opera en paralelo, según el número de cauces disponibles, sobre un número determinado de vértices. La posición de salida de cada vértice determina también las zonas del buffer final que serán cubiertas por pixels. A nivel de vertex shader se puede leer desde textura mediante VTF (*Vertex Texture Fetch*) en GPUs de última generación. De esta forma pueden implementarse operaciones basadas en recopilación de múltiples datos. El problema principal es que cada vértice está aislado del resto en el flujo de datos, lo que impide operaciones en las que intervengan varios elementos.

Los procesadores de fragmentos funcionan de forma parecida a los de vértices, esto es, ejecutan instrucciones SIMD sobre el flujo de datos. Tienen más capacidades de acceso a textura, sin embargo, la posición final de cada fragmento no puede modificarse. La GPU suele tener más procesadores de fragmentos que de vértices. La razón es que siempre se procesan más pixels que vértices. Además, los cálculos que se realizan a nivel de fragmento suelen ser mucho más complicados que los realizados a nivel de vértice.

A.7.2 Gestión de memoria

Las posibilidades de gestión de memoria de la GPU son todavía limitadas [Lefohn05]. La principal razón es la necesidad de preservar el paralelismo del procesamiento de los datos para mantener la ventaja de rendimiento. A pesar de esto, en el futuro se solucionarán en parte estas restricciones. No hay que olvidar que la tecnología del hardware gráfico sigue en continua evolución. La siguiente lista presenta una seria de reglas para las unidades de procesamiento de vértices y fragmentos que cumplen las especificaciones Vertex Shader 3.0 y Pixel Shader 3.0 [Microsoft04]:

- No hay acceso a memoria principal ni a disco. No existe pila (stack) ni espacio utilizable (heap).
- Se permiten lecturas desde textura.
- Se permiten la lectura desde registros constantes. Los vertex shaders pueden utilizar indexación relativa de registros constantes.
- Se permite la lectura/escritura desde/en registros temporales. Estos registros son locales a cada flujo de datos. No se permite indexación relativa de registros.
- Se realiza una lectura de flujos (streaming). El procesador de vértices lee flujos de vértices. El procesador de fragmentos lee flujos de fragmentos.
- Se realiza una escritura de flujos (de salida). La posición de escritura en el flujo queda fijada en cada elemento. El procesador de vértices puede escribir hasta 12 vectores de 4 números en coma flotante. El procesador de fragmentos puede escribir hasta 4 vectores de 4 números en coma flotante.

A.7.3 Rasterizador

Después de que el procesador de vértices transforma los vértices del flujo de datos, se sintetiza cada primitiva geométrica utilizando información de conectividad entre vértices y se genera un flujo de fragmentos que cubren dicha primitiva. Esta labor la realiza el rasterizador, que puede ser considerado como un interpolador. Para cada fragmento generado se calculan los atributos en función de la posición respecto de los vértices que intervienen en la primitiva. Las funciones del rasterizador son muy especializadas y no son programables por el usuario. Además están muy optimizadas para renderizar triángulos.

A.7.4 Unidades de textura

El procesador de fragmentos (y el de vértices en las GPUs modernas) pueden acceder a memoria en forma de texturas. Cada textura se comporta como una interfaz de sólo lectura. Las texturas se utilizan en los algoritmos de propósito general como datos de entrada que están disponibles en la ejecución de todos los vertex shaders y pixel shaders.

A.7.5 Rendering en textura

Cuando la GPU genera una imagen, el resultado queda almacenado en el framebuffer para ser visualizado o bien puede ser escrito en una textura. Esta técnica es fundamental para la implementación de algoritmos de propósito general en GPU, ya que es la única forma de mantener los resultados de un programa en memoria de video sin tener que transferir los datos de nuevo desde memoria principal. Esta operación es como una interfaz de memoria de sólo escritura. También es posible realizar una copia de los datos desde el framebuffer a textura, aunque esto implica un coste adicional.

La razón de realizar las escrituras de datos en GPU en un buffer es que el procesador de fragmentos puede leer de cualquier forma desde textura, pero puede escribir tan sólo una vez por cada fragmento en el buffer resultado al final del shader. La salida del procesador de fragmentos es un flujo de pixels, por lo que la lectura y escritura de datos con la GPU debe hacerse de la forma descrita.

En un principio se empezó a denominar rendering en textura a todo proceso de rendering cuyo resultado era un buffer que no sería visualizado sino utilizado posteriormente como datos se entrada. Esto es debido a que los primeros algoritmos que necesitaban de esta técnica eran los de generación dinámica de sombras, proyección del entorno en texturas cúbicas, etc. Además todos estos buffers se reutilizaban como texturas en las primeras GPUs programables. Últimamente ha cambiado un poco el concepto, ya que los lenguajes de sombreado de alto nivel como Cg [Fernando03] consideran la entrada desde memoria como *samplers* (orígenes de datos para muestreo) en lugar de simples texturas. Esto es así ya que dichas *texturas* pueden disponer de formatos en coma flotante con datos no visualizables, lo que rompe el concepto clásico de textura y las presenta como buffers o matrices convencionales. Es por ello que sería más apropiado en la mayoría de las situaciones hablar de *rendering en buffer*. Las tecnologías relacionadas con este aspecto son los *p-buffers* y los *frame buffer objects*.

A.7.6 Analogías CPU-GPU

La programación de algoritmos de propósito general en GPU puede resultar bastante complicada, ya que presenta una serie de peculiaridades que provocan un replanteamiento de la mayoría de los algoritmos antes de implementarlos para ser ejecutados en el hardware gráfico. A continuación se presentan algunas equivalencias conceptuales entre CPU y GPU [Harris05].

Las texturas en GPU pueden considerarse como matrices clásicas en CPU. La textura es la estructura de datos fundamental en el hardware gráfico, junto con los vectores de vértices que sirven como entrada al pipeline. Casi siempre que puede utilizarse una matriz para un algoritmo en CPU, puede hacerse lo propio con una textura en GPU, que será accesible desde un programa de fragmentos o desde uno de vértices en GPUs de última generación.

Los programas de fragmentos y de vértices pueden servir como el cuerpo de un bucle. Con la CPU se utiliza un bucle para iterar sobre un conjunto de elementos, que puede estar organizado como un vector o un flujo de datos, para procesarlos de forma secuencial. En el caso de la GPU los elementos sobre los que se itera pertenecen al flujo de entrada, ya sea a nivel de vértices o de fragmentos. El kernel que se ejecuta en el procesador de vértices o de fragmentos constituye el cuerpo del bucle. El grado de paralelismo que se alcanza con la GPU depende del número de cauces de que disponga, pero también del aprovechamiento del paralelismo a nivel de instrucción mediante las operaciones aritméticas del hardware gráfico.

El proceso de rendering en textura puede ser considerado como la operación de escritura del resultado de un algoritmo en CPU. La mayoría de los programas en GPU están divididos en varias etapas, de forma que cada una depende de la anterior. En la GPU cada flujo debe procesarse por completo antes de tener el resultado disponible. La obtención de resultados, aunque sean parciales, es trivial en la CPU gracias a su modelo de memoria unificada. Con la GPU no es tan fácil, debido a la necesidad de completar el proceso sobre el flujo de datos hasta vaciar el pipeline. El rendering en textura (o en cualquier buffer en GPU) es el único modo de disponer de los datos de salida en la siguiente ejecución de un programa en hardware.

El proceso de rasterización de primitivas geométricas puede verse como el punto de entrada de un algoritmo implementado en vertex o en pixel shader. La forma de invocar un programa en la GPU puede ser extraña ya que debe hacerse de forma indirecta para conseguir la ejecución de un kernel sobre el flujo de datos deseado. Para lanzar la ejecución de vertex shader hay que introducir un vértice en el pipeline para cada iteración del programa requerida. Los argumentos necesarios deben ir codificados como atributos de ese vértice. En el caso del pixel shader hay que dibujar primitivas

geométricas cuya rasterización genere fragmentos para lanzar dicho pixel shader tantas veces como sea necesario. Lo habitual con algoritmos de propósito general es dibujar un cuadrilátero en 2D sobre el viewport que representa un grid o matriz 2D. De esta forma se provoca la ejecución del pixel shader sobre cada elemento del grid.

A.7.7 Reducciones

Hasta ahora se ha supuesto que los algoritmos ejecutados en GPU tienen como objetivo el procesamiento en paralelo de múltiples elementos de un flujo que da como resultado otro flujo con los resultados. De esta forma, cada elemento se procesa de forma independiente de los demás, y por tanto genera un resultado también independiente. No obstante, en muchos programas el resultado final depende de todos los elementos del flujo, y suele ser un único número o un conjunto más reducido que el original. Ejemplos típicos son la suma de resultados parciales o el cálculo del máximo de un conjunto. A este tipo de cálculo en la GPU se le denomina *reducción en paralelo*.

Las reducciones en GPU se realizan alternando el rendering y la lectura de datos con un par de buffers. En cada pasada, el tamaño de la salida se reduce en una fracción. Para producir cada dato de salida un pixel shader lee desde un buffer (tratado como textura) dos o más valores para producir el resultado utilizando un operador de reducción, como una suma o el máximo valor. En el siguiente paso el buffer de salida se convierte en origen de datos (textura) y se repite el proceso hasta que la salida contiene la cantidad de datos deseada, que puede ser una matriz o un único valor. En la Sección 4.3.3.2 se muestra un proceso de reducción utilizado en este trabajo.

A.7.8 Batching

Hay varias formas de enviar datos al pipeline de la GPU. La información de vértices constituye el flujo de datos principal, mientras que la información de conectividad de primitivas es algo menos importante en cuanto a la forma de envío. Cuando se envía un conjunto de primitivas a la GPU se produce normalmente un cambio de contexto, que implica el vaciado de todos los cauces del pipeline. El contexto o estado de rendering se refiere a los parámetros actuales de rendering que se aplican a todas las primitivas, como el texturado, luces, etc. Es mucho más eficiente enviar pocas secuencias de millones de triángulos que enviar millones de secuencias de pocos triángulos. A cada secuencia de instrucciones que se *empaqueta* como una sola operación de rendering se la conoce como *batch* [Wloka03]. Sin embargo, hay algunos factores que hay que tomar en consideración para alcanzar el máximo rendimiento.

Para enviar los datos de los vértices a la GPU pueden utilizarse diversos métodos que dependen de la interfaz de programación utilizada. Todas las APIs ofrecen mecanismos para enviar primitivas de dibujo de una en una, lo que es extremadamente ineficiente debido a los cambios de contexto que suelen utilizarse. Siempre es más eficiente utilizar batching para enviar datos y renderizar las primitivas. Para ello pueden emplearse display lists y vertex arrays en OpenGL, y vertex buffers tanto en OpenGL como en Direct3D. Estas formas de envío de información a la GPU son muy convenientes, ya que minimizan la cantidad de transferencias CPU-GPU y los cambios de contexto.

Uno de los problemas que aparecen con el rendimiento al utilizar batching es la proximidad de los datos de los vértices en memoria de GPU. En efecto, puede producirse un impacto en el rendimiento relacionado con el tamaño de las estructuras utilizadas para enviar datos desde la CPU (Ej.: display lists). Este hecho está relacionado con el tamaño de la caché pre-T&L del hardware gráfico. Cuando una secuencia de triángulos hace referencia a una gran cantidad de vértices que están muy separados se producen faltas de caché dentro de la GPU, lo que impacta notablemente en el rendimiento.

El segundo problema con el rendimiento implica a la CPU. Cuando se utilizan buffers (batchs) muy pequeños, la CPU está ocupada enviando datos todo el tiempo. La idea del batching respecto de la CPU es liberarla tras enviar la suficiente cantidad de datos a la GPU para mantenerla ocupada. Como puede verse, existe un tamaño óptimo de buffer (display lists, vertex array, etc.) que depende del hardware gráfico utilizado, de la velocidad del sistema y de la API utilizada [Wloka03].

A.8 Adaptación de la GPU a la programación de propósito general

Tradicionalmente la GPU ha sido un coprocesador gráfico ligado al resto del ordenador. Sin embargo, la evolución de este hardware ha permitido en los últimos años que la comunidad científica desarrolle implementaciones alternativas de algoritmos de propósito general. El motivo principal que induce al uso de la GPU en lugar de la CPU es la mayor capacidad de la primera para realizar cálculos matemáticos en coma flotante [Buck04]. Por ejemplo, un Pentium 4 a 3.6GHz puede alcanzar un máximo de 7.2Gflops con un ratio de transferencia de 6GB/s, mientras que una GeForce 7800GTX a sólo 430Mhz puede llegar hasta los 165 gigaflops con un ratio de transferencia de 38.4GBs. Además de las ventajas actuales de las GPUs, todo parece indicar que la evolución del hardware gráfico será más rápida que la de las CPUs. En cualquier caso, siempre será posible utilizar los dos tipos de procesadores al mismo tiempo para maximizar el rendimiento.

Las GPUs actuales trabajan con vertex shaders y fragment shaders sobre datos geométricos para el proceso de rendering. Sin embargo, puede codificarse información de todo tipo en forma de vectores de vértices con atributos o como texturas, utilizando los shaders como pequeños programas de cálculo de propósito general que serán utilizados en paralelo.

Para que un algoritmo pueda adaptarse a esta forma de procesamiento, debe ser paralelizable. Ya que los shaders no pueden compartir memoria para almacenar datos con operaciones de escritura (sí de lectura), los cálculos que se realizan sobre cada elemento deben ser realizados de forma independiente. Por tanto, la función a aplicar a los datos debe ser exactamente la misma.

El hardware gráfico actual no dispone de las suficientes salidas de datos para la mayoría de los algoritmos de propósito general. Debido a que la única salida del pipeline gráfico es el valor devuelto por el pixel shader en forma de color, los datos que pueden procesarse en una sola pasada quedan limitados. Es previsible que esta limitación desaparezca en el futuro.

Un problema común en las GPUs actuales es la lentitud de las transferencias de memoria GPU-CPU. Los valores de salida de los algoritmos implementados en el hardware gráfico siempre quedan almacenados en un buffer que hay que recuperar desde la tarjeta gráfica. Cuando se utiliza un bus AGP, esto presenta un problema, ya que este tipo de conexión es asimétrica, estando mucho más optimizada para subir datos a la GPU que para recuperarlos. Todo esto puede hacer que el rendimiento del algoritmo en términos globales resulte menor mediante el uso de la GPU. Gracias a las nuevas placas con PCI-Express, los efectos de esta limitación disminuyen.

Con todo lo visto, no todos los algoritmos que se implementan en la CPU pueden llevarse al modelo de programación de la GPU. Es obligatorio paralelizar el proceso, así como adaptarlo a las limitaciones mencionadas anteriormente, lo que obliga en la mayoría de las ocasiones a replantear por completo el problema. Aún cuando es posible adaptar los algoritmos a la GPU, el cambio del planteamiento y modo de operación puede hacer que la versión por CPU sea más eficiente, aunque ésta sea menos potente. Suele ocurrir cuando se utilizan estructuras de datos más o menos complejas o cuando los datos intermedios dependen unos de otros, lo que obliga a dividir el proceso en varias pasadas.

Se espera que en el futuro las GPUs evolucionen de tal forma que permitan más posibilidades de programación que, aunque sin llegar a la generalidad de una CPU convencional, permitan la implementación de algoritmos más complejos de forma más eficiente.

A.9 Conclusiones

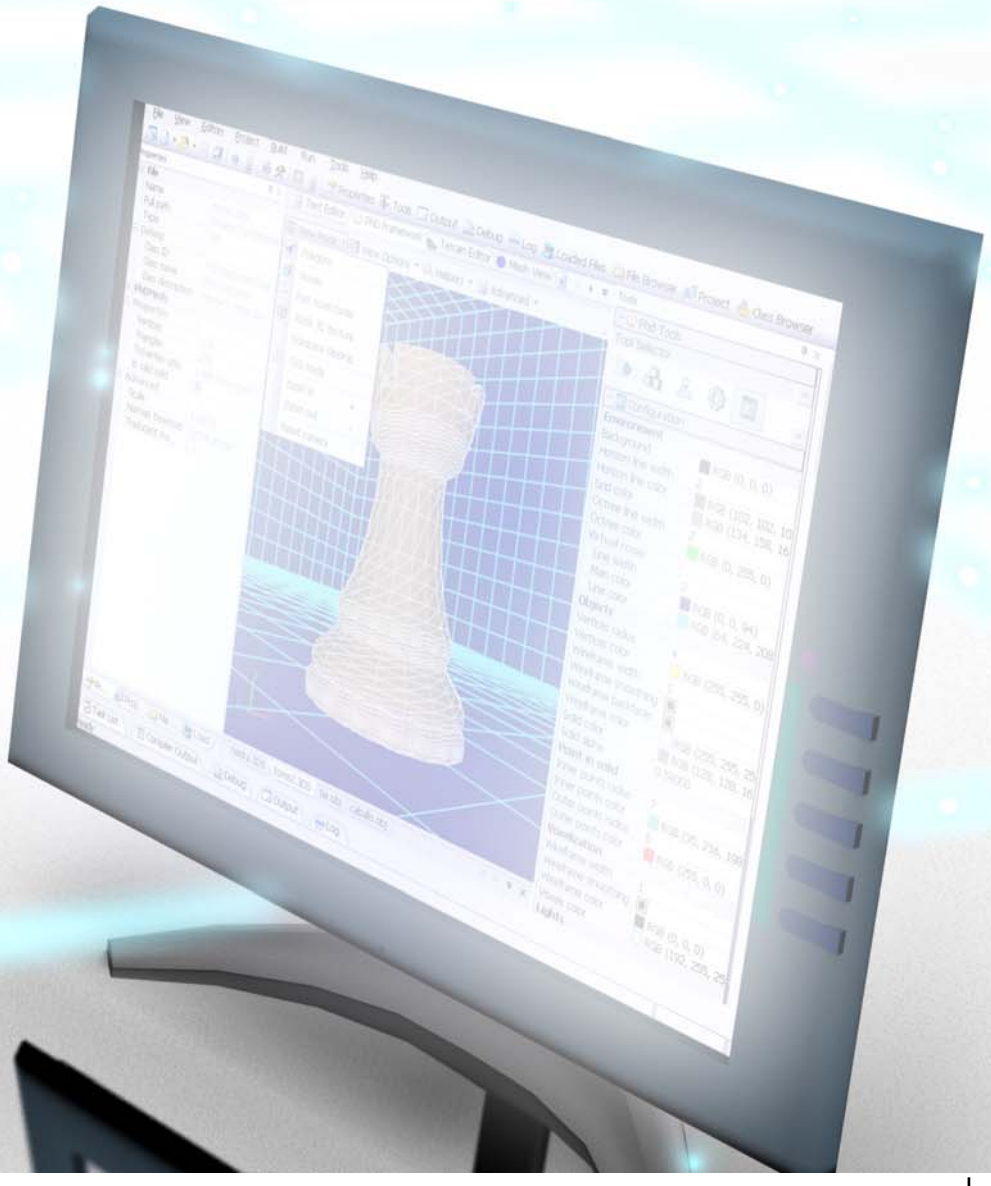
En este Capítulo se han presentado algunos conceptos relacionados con la GPU programable actual. La historia de la GPU programable es relativamente breve en comparación con otros aspectos de la informática. Sin embargo, la evolución ha sido, y sigue siendo, espectacular. Cada año se implementan nuevas mejoras y se aumenta el rendimiento, hasta un punto en el que mantener actualizado un software 3D puede llegar a ser una ardua tarea. Realmente el ciclo de vida de una gama de GPUs puede llegar a ser menor que el tiempo de desarrollo de algunos programas.

La GPU programable ofrece la posibilidad de adaptar algunos algoritmos secuenciales al procesamiento vectorial. Aunque la GPU no es realmente un procesador vectorial puro orientado a flujos de datos y kernels genéricos, sí ofrece las suficientes posibilidades para ejecutar programas con alta utilización de operaciones matemáticas de forma más eficiente que la CPU.

En la actualidad existen varios lenguajes de sombreado de alto nivel que permiten la implementación de shaders para visualización y de algoritmos de propósito general. Aunque hay otras alternativas de mayor nivel, los lenguajes Cg, HLSL y GLSL se presentan como las opciones que ofrecen un mayor aprovechamiento del hardware a la vez que permiten un desarrollo rápido. Las APIs más utilizadas son OpenGL y Direct3D, y ambas permiten el desarrollo de aplicaciones 3D eficientes que hacen uso del hardware gráfico de forma directa o a través de los lenguajes de sombreado.

Apéndice **B**

SOFTWARE IMPLEMENTADO



En este Capítulo se presentan las características generales del software implementado para realizar las pruebas de todos los algoritmos descritos en esta tesis. El objetivo es proporcionar unas nociones de cómo y en qué circunstancias se han llegado a obtener los resultados prácticos.

B.1 Antecedentes

Durante el desarrollo de esta tesis ha sido necesario implementar una cantidad considerable de algoritmos. Las soluciones presentadas a lo largo de este trabajo tienen una parte práctica muy importante, y necesitaban de un entorno que permitiera la experimentación y evolución de las implementaciones. Uno de los problemas iniciales era obtener un software que permitiera el desarrollo de todos los algoritmos necesarios bajo las mismas condiciones y de la forma más eficiente posible. El resultado es un programa llamado *Solid*.

Como suele ser habitual en este tipo de trabajos, y en particular los relacionados con la Informática Gráfica, lo normal es concentrar todos los esfuerzos de desarrollo en elaborar los algoritmos necesarios dejando de lado cualquier aspecto secundario, como la interfaz de usuario, la conectividad, la entrada de datos, etc. El resultado suele ser un software, normalmente considerado como *de uso propio*, en el que la calidad, la usabilidad y en ocasiones hasta el rendimiento ocupan un lugar secundario (si es que ocupan alguno). Cuando comenzó el diseño de Solid, el objetivo fundamental era evitar todos estos factores.

Entre los propósitos de este trabajo figuran muchas y variadas implementaciones que deben ser tan eficientes como sea posible. Esto incluye algoritmos de distintos autores además de los propios, así como implementaciones basadas en GPU. Para que los estudios comparativos sean concluyentes todos los métodos para resolver el mismo problema deben utilizar las mismas clases, estructuras de datos y funciones auxiliares siempre que sea posible.

Cuando empezó a diseñarse el software se tuvieron en cuenta una serie de factores que determinarían el resultado final. Lo que se buscaba era un sistema que fuera fácil de extender y actualizar, ya que su vida útil sería de algunos años. Los aspectos que definen la estructura del programa desarrollado son los siguientes:

- El desarrollo del software debe ser muy rápido para poder concentrar los esfuerzos en los experimentos con los algoritmos a desarrollar.

- El software debe ser lo más eficiente posible. El acceso al hardware gráfico debe estar garantizado, tanto a nivel de API como de shaders.

- La interfaz debe ser lo bastante elaborada para permitir una utilización rápida y cómoda del software, y lo bastante sencilla como para consumir poco tiempo de desarrollo.

- El software debe estar preparado para agregar funcionalidad, nuevas estructuras y algoritmos dentro de unos límites aceptables.

- Gráficamente debe permitir la visualización interactiva de los resultados y obtener imágenes de alta resolución en tiempo real.

- Se debe aceptar una cantidad suficiente de formatos gráficos como entrada y ofrecer una estructura de datos que soporte mallas de polígonos compatibles con recubrimientos simpliciales.

En función de los criterios anteriores se decidió implementar todo el código en C++, ya que permite un diseño orientado a objetos estructurado y fácilmente modificable [Pender03], a la vez que permite implementaciones razonablemente eficientes y acceso a recursos de bajo nivel.

Durante los primeros meses de desarrollo se optó por basar el software en Linux, utilizando la librería Mesa (OpenGL) como API 3D. En un principio la portabilidad y la utilización de software libre fueron consideradas razones fundamentales, y la elección más lógica era el uso de un sistema Linux. Para despejar dudas sobre la elección del sistema, la primera versión del software se codificó en C++ estándar, utilizando OpenGL y la librería GLUT para la interfaz. Este programa compilaba tanto en Windows como en Linux, lo que permitía establecer comparaciones entre las condiciones de desarrollo en ambos sistemas.

Durante los primeros meses se observó que los controladores de OpenGL de Linux causaban problemas de estabilidad, tenían funciones sin implementar y había problemas de compatibilidad entre placas gráficas de distintos fabricantes. Posteriormente se descubrieron fallos en el optimizador de código del compilador de C++, y se comprobó que los primeros algoritmos funcionaban más rápido en Windows, ya que Visual Studio generaba código Pentium 4 con SSE2 más optimizado que el compilador de Linux (gcc/g++). Además, mientras que en Linux no había entonces un entorno de desarrollo realmente cómodo y efectivo, Visual Studio ofrecía todas las posibilidades necesarias. Aunque las soluciones Linux mejoraron desde entonces, se optó por el continuar el desarrollo exclusivamente en Windows.

B.2 Estructura del software

El entorno de desarrollo utilizado durante todo el ciclo de vida del software es Visual Studio de Microsoft. Las versiones que se han utilizado son Visual Studio 6.0, Visual Studio .NET y Visual Studio 2005 bajo Windows XP. Esta elección viene motivada por la posibilidad de creación de la interfaz de usuario, la ejecución y depuración completa dentro del entorno, así como las optimizaciones de código disponibles (SSE2, multitarea, hebras, etc.), además de la gran base de conocimiento disponible MSDN (Microsoft Developer Network). Para el acceso a la GPU programable se ha utilizado NVidia Cg para la programación de shaders y OpenGL 1.5 y 2.0 para el API 3D.

El software es en parte una evolución de trabajos anteriores, algunos relacionados con las primeras etapas de la investigación y otros aplicados a tareas distintas [Heras05], aunque todos relacionados con la Informática Gráfica. Para cumplir con los requerimientos iniciales se diseñó una estructura de clases minimalista y

sencilla abierta a futuras extensiones. Desde el principió se separaron claramente las clases relacionadas con la interfaz de las consideradas parte del núcleo del software.

Como es sabido, C++ introduce una serie de penalizaciones en el rendimiento, debido sobre todo a factores como la ligadura dinámica, los constructores, la sobrecarga de operadores, la RTTI (Run-Time Type Information), etc. El lenguaje se ha utilizado como medio y no como fin, por lo que la aplicación, las estructuras y los algoritmos están optimizados para el rendimiento; se ha evitado el uso de la ligadura dinámica (funciones virtuales), el diagrama de clases es mínimo, se han usado funciones inline y macros cuando ha sido posible, y además se ha evitado en todo momento el paso de argumentos complejos por valor, entre otras medidas.

B.2.1 Organización del software

La Figura B-1 muestra las unidades funcionales más importantes del software y sus dependencias más significativas. Cada unidad se compone de una o varias clases interrelacionadas de forma conveniente. El programa está diseñado de forma que se define un núcleo y una interfaz totalmente separados. Con el fin de ofrecer una breve descripción del software se describirán las unidades funcionales en lugar de toda la jerarquía de clases.

Las clases del núcleo se encargan de realizar todas las tareas relacionadas con los algoritmos geométricos implementados, los optimizadores espaciales, la carga de datos, la gestión de shaders y las operaciones básicas sobre los sólidos. Estas clases están programadas en ANSI C++ y no dependen de ninguna librería propia del sistema operativo ni de ningún software propietario. El objetivo es mantener el núcleo portable a otros sistemas operativos y compiladores.

Las clases de la interfaz están muy relacionadas con el sistema operativo. En concreto, están basadas en MFC (Microsoft Foundation Classes), que es una librería que proporciona una abstracción en C++ del API de Windows, lo que permite el control de todos los aspectos de la interfaz de usuario. Debido a que la adaptación de las clases de la interfaz a una librería propia de Windows impide su portabilidad a otros sistemas, este grupo de clases se mantiene muy separado del núcleo.

Las figuras B-2, B-3 y B-4 muestran algunas imágenes de Solid durante su ejecución. A continuación se realiza una breve descripción de cada unidad funcional que se corresponde con un conjunto de clases:

Interfaz. Este conjunto de clases se encarga de todos los aspectos de la interfaz de usuario. El software está basado en el modelo Documento-Vista en el que se define

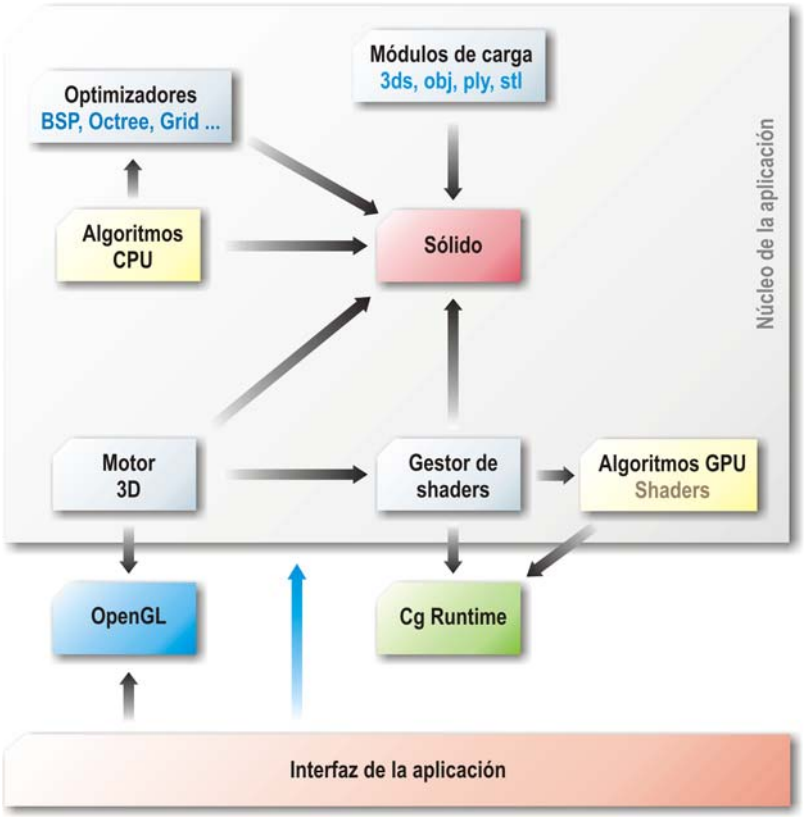


Figura B-1. Diagrama básico del núcleo de Solid que indica las unidades funcionales más importantes y sus dependencias. Cada unidad del núcleo y la interfaz se compone de una o más clases que están interrelacionadas convenientemente.

una clase para manipular documentos (datos de los sólidos) y otra para gestionar la forma de representación, que en esta aplicación es una ventana de OpenGL. Todo lo relacionado con ventanas de diálogo, menús, interacción, etc., es gestionado por esta gran unidad funcional.

B

Módulos de carga. Para la carga de sólidos se han implementado varios módulos que permiten importar datos desde archivos *3ds*, *obj*, *ply* y *stl*. También se pueden grabar datos en *stl*, pero es una función menor. Todos los datos cargados son traducidos a una malla de polígonos. La Sección B.2.2 contiene más detalles sobre los módulos de carga.

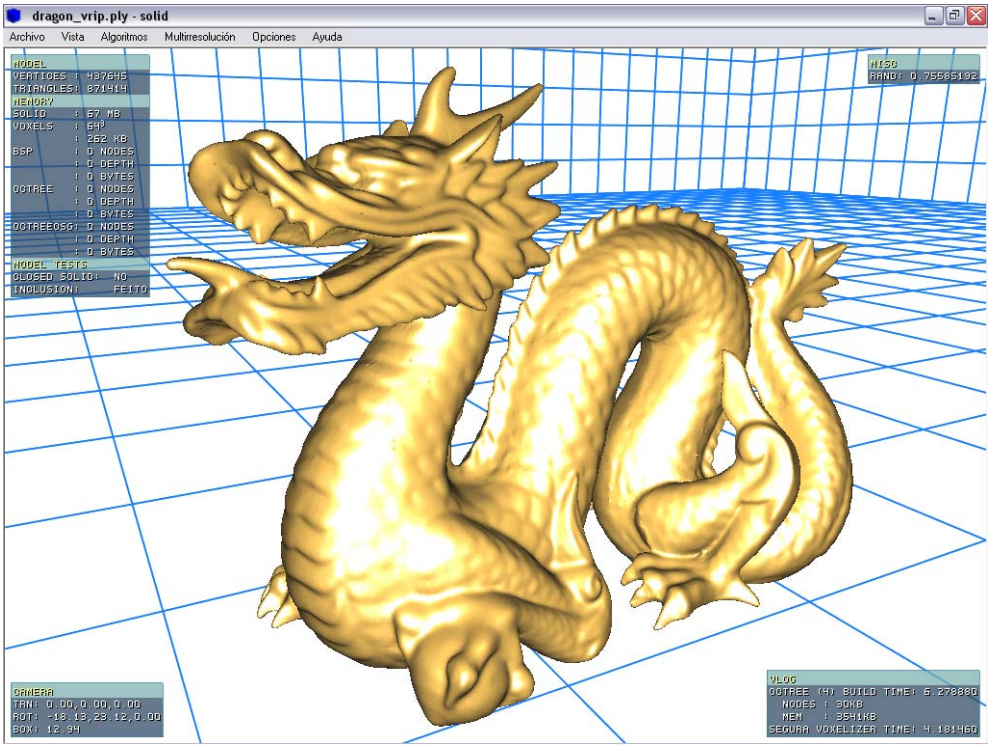


Figura B-2. Visualización de un modelo en tiempo real con Solid.

Sólido. Esta unidad se compone de una sola clase que organiza y gestiona lo relacionado con la malla de polígonos que representa al sólido, incluyendo su recubrimiento simplicial y algunos atributos adicionales. Aunque contiene funciones dedicadas al tratamiento de las mallas de polígonos, los algoritmos de mayor nivel son implementados en otras unidades.

Optimizadores. Muchos algoritmos implementados hacen uso de una estructura de datos espacial como el octree o el grid. En esta unidad se implementan todas esas estructuras que están vinculadas a un sólido.

Algoritmos CPU. En esta unidad se incluyen las clases que contienen los algoritmos geométricos presentados en este trabajo. Estos algoritmos no se incluyen como parte de la clase que gestiona los sólidos ya que restan flexibilidad al diseño. Se les denomina algoritmos CPU porque están implementados en C++ para ser compilados y ejecutados en la CPU.

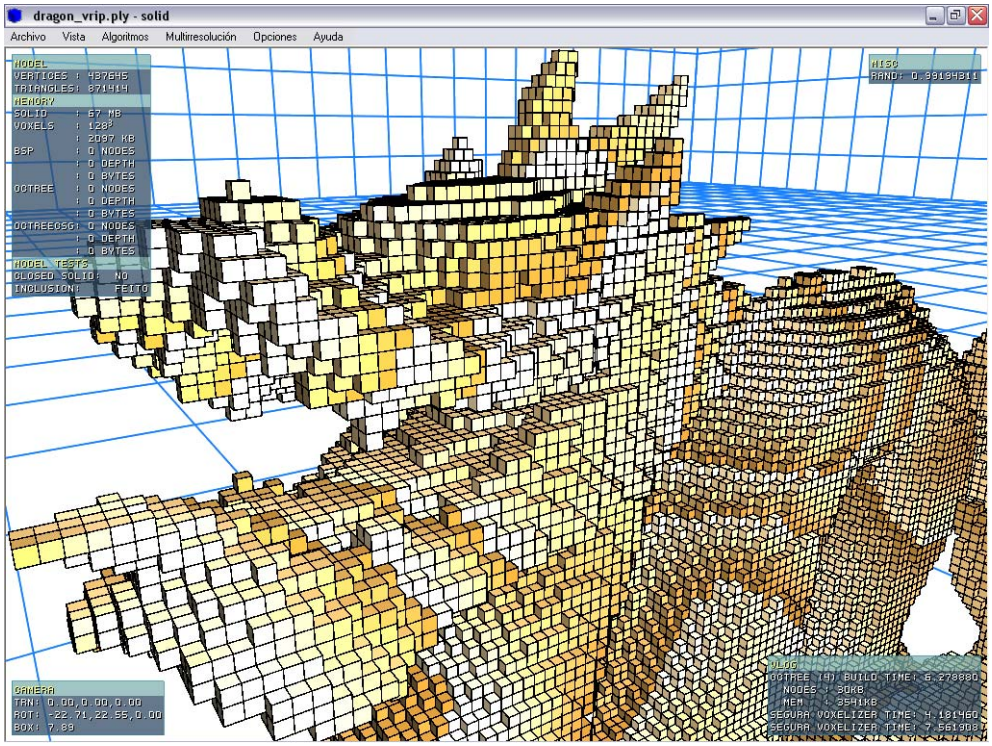


Figura B-3. Voxelización de un sólido con Solid utilizando una función Perlin 3D.

Motor 3D. Este grupo de clases se encarga de todo el proceso de rendering de la escena, así como del control de los shaders o programas para la GPU. Es la unidad funcional más compleja.

Gestor de Shaders. Se encarga de gestionar la carga, compilación y ejecución de los programas para la GPU. Está siempre al servicio del motor 3D y constituye el enlace con Cg.

Algoritmos GPU. Esta unidad se compone realmente de un conjunto de ficheros Cg que contienen la versión GPU de algunos algoritmos implementados en este trabajo. Parte de estos algoritmos necesitan soporte a nivel de CPU para preparar estructuras, convertir resultados, etc. Esta parte está incluida en el gestor de shaders.

Cg Runtime. Es la librería de NVidia que da soporte al lenguaje Cg. Se encarga de compilar, configurar y ejecutar los shaders a partir de ficheros con código fuente en Cg.

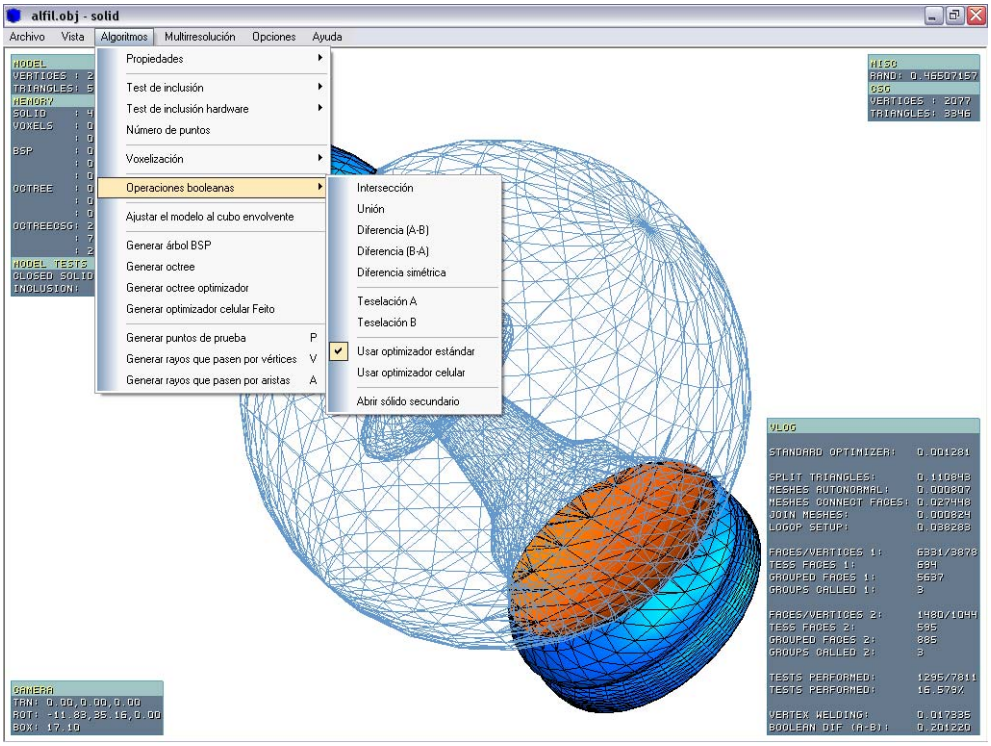


Figura B-4. Diferencia booleana entre dos sólidos realizada con Solid.

B.2.2 Formatos gráficos

Para probar los algoritmos sobre una gran cantidad de sólidos de forma cómoda, se han desarrollado módulos de carga de datos para trabajar con distintos formatos 3D. Para poder tratar mallas provenientes del diseño industrial y del escaneado tridimensional se ha implementado el soporte de *ply* y *Stereolitho (stl)* para cargar datos y de *stl* para grabarlos. Para procesar mallas creadas con programas de diseño se ha implementado la carga de datos en formato *WaveFront (obj)* y *3D Studio (3ds)*.

Cada formato gráfico tiene sus características. Por ejemplo, *3ds* sólo admite triángulos, mientras que *obj* permite polígonos de más de tres lados. En cualquier caso el conjunto de formatos cubre las necesidades del software para la entrada de datos. El formato *3ds* ha sido utilizado además para importar objetos desde los programas comerciales *3D Studio Max*, *Maya* y *Cinema4D*.

B.3 Entorno hardware

Durante el desarrollo el software se ha probado sobre múltiples plataformas, incluyendo ordenadores portátiles, de sobremesa y estaciones de trabajo. Se ha probado sobre CPUs de AMD y de Intel, incluyendo versiones de 64 bits. También se ha probado el programa sobre un buen número de GPUs de varios fabricantes, en especial ATI y NVidia, y que van desde la segunda hasta la cuarta generación. Todas estas pruebas fueron determinantes para localizar fallos e incompatibilidades en el software, en especial en la parte relacionada con la GPU y los shaders.

El hardware con el que se han realizado las pruebas finales de todos los algoritmos de este trabajo es el siguiente:

- CPU Intel Pentium 4E, 3417MHz (17x201), caché 1024Kb.
- Chipset Intel Grantsdale i915.
- 1024Mb de memoria RAM. 800Mhz FSB.
- Tarjeta gráfica NVidia GeForce 7800GTX, 256Mb, PCI-Express x16.

B.4 Evolución del software

Todo software evoluciona de alguna forma. La clave para mejorar un sistema es localizar los problemas y carencias que presenta, estudiar su futura utilidad y determinar lo mejor posible los cambios que deben realizarse. Durante el proceso de investigación ha sido necesario realizar tareas importantes de implementación, depuración y pruebas. El software que nos ocupa fue pensado en principio para experimentar sólo con algunos algoritmos geométricos. Sin embargo, con el transcurso del tiempo siempre se espera algo más del software, por lo que su evolución es inevitable.

En cualquier contexto de investigación en Informática, en especial en los de ámbito personal y de grupos reducidos, lo más habitual es ir aprovechando una y otra vez el mismo software o componentes del mismo, incluso para tareas muy distintas. Lo usual es invertir el mínimo tiempo posible en la programación de aplicaciones, y por esta razón casi todo el mundo tiene un software *muy personal* que sufre evoluciones a lo largo del tiempo para adaptarse a casi todo. Esto incluye clases en Java o C++, módulos de cualquier tipo, scripts, etc.

Es normal que en el campo de la investigación se reste importancia a la calidad de un software en favor del tiempo dedicado a la experimentación. Sin embargo, llega

un momento en el que la envergadura de los proyectos en los que se trabaja obliga a replantear el software, de forma que el resultado sea más o menos profesional y que se adapte adecuadamente a las necesidades en cada momento, incluyendo las de otros usuarios.

El programa Solid ha llegado a un punto en el que no es muy ampliable debido a su diseño minimalista y a su sencilla interfaz de usuario. Teniendo en cuenta que se realizarán más desarrollos relacionados con los temas tratados en esta tesis, los planteamientos deben cambiar para el software, ya que es muy evidente que la reutilización de código es inevitable en futuros trabajos. El objetivo principal es no tener que empezar ningún software desde cero ni tener que modificar el mismo programa durante años, aunque éste no se adapte a las necesidades concretas en cada situación.

B.4.1 Hacia un software multidisciplinar

En Informática Gráfica suele utilizarse una serie de herramientas relacionadas con geometría, topología, cálculo, álgebra, física y un largo etcétera. Estas utilidades pueden agruparse como librerías de clases que permiten un desarrollo muy eficiente en todos los aspectos. Una organización conveniente de estos componentes ofrece una solución inicial para la reutilización de código en proyectos de todo tipo. Más aún, se pueden utilizar componentes como librerías de sonido, utilidades de compresión de archivos, módulos de inteligencia artificial, manipulación de video, etc. En proyectos de realidad virtual suele darse la necesidad de utilizar una gran cantidad de componentes de muy diverso tipo.

La utilización de librerías para la programación de aplicaciones es muy conveniente. Sin embargo, la parte de la interfaz de usuario suele seguir siendo un problema en cualquier software, ya que la mayoría de las librerías dedicadas a este tema sólo proporcionan componentes de bajo nivel.

En el diseño de Solid se contemplaba la necesidad de elaborar grupos de clases dedicadas a tareas concretas, muchas de las cuales podrían formar librerías completas, como es el caso de las clases geométricas. La interfaz de usuario necesitó de un número considerable de clases para obtener resultados muy discretos. Para el futuro del software se consideró la opción de separar el núcleo en librerías organizadas por su funcionalidad, y aumentar la interfaz con clases más potentes y versátiles que permitieran solucionar la mayoría de los problemas mediante abstracciones.

Paralelamente a las últimas fases del proceso de investigación que da lugar a esta tesis se comenzó a desarrollar un software con una filosofía completamente

distinta a la empleada con Solid. La motivación principal se basa en la dificultad de adaptar un software de propósito muy específico a otros usos para los que no fue diseñado. El objetivo principal es conseguir un sistema con el que dar salida a todas las actividades relacionadas con un proceso de investigación y/o desarrollo en Informática, y en especial en temas relacionados con la Informática Gráfica, ya que las actividades futuras se realizarán en este ámbito. Esto incluye tareas como la implementación e integración de componentes software, la edición de código fuente, el tratamiento de documentos de cualquier tipo (gráficos vectoriales, escenas 3D, etc.), entre otras.

Tomando como idea de partida la potencia y flexibilidad de las tecnologías COM (Component Object Model) y .NET, pero a un nivel mucho más sencillo, se comenzó a diseñar este nuevo software que actualmente está en fase de implementación. La Figura B-7 muestra un esquema básico de los componentes del nuevo sistema. Lo que une a Solid con este nuevo software, además de ser un punto de partida en su diseño, es la posibilidad de integrar el primero como un componente del segundo. Los objetivos principales de este nuevo sistema son los siguientes:

- Proporcionar un entorno o marco de trabajo (*framework*) para desarrollar software de calidad relacionado con cualquier proceso de investigación y/o desarrollo, en especial en Informática Gráfica. La idea es crear un entorno orientado al tratamiento de documentos a través de módulos o componentes (*plugins*) que los manipulen. Algunos ejemplos de programas que integran la edición de distintos tipos de documentos son los editores de niveles de muchos videojuegos. Con estos programas pueden editarse los escenarios 3D, los scripts de comportamiento de los personajes, los eventos de la historia, los sonidos, etc. Como puede verse, son múltiples tipos de documentos dentro de un mismo proyecto. La Figura B-5 muestra un ejemplo de edición de varios documentos.

- Proporcionar las funciones típicas de cualquier software a los distintos módulos que gestionan documentos. Casi todas las aplicaciones tienen que cargar archivos, imprimir, editar propiedades de objetos, etc. La elaboración que se hace de estas funciones suele ser la misma a nivel de interfaz, con lo que esta parte puede implementarse de forma común.

- Ofrecer medios para crear y gestionar proyectos y grupos de proyectos. Un proyecto se define como un conjunto de documentos relacionados de forma jerárquica. Los documentos están organizados en disco como cualquier archivo, pero mantienen en el proyecto una organización igual o distinta a la del sistema de ficheros. Mediante un espacio de trabajo (*workspace*) se permite integrar un grupo de proyectos, lo que es de utilidad cuando éstos deben estar sujetos a los mismos parámetros de

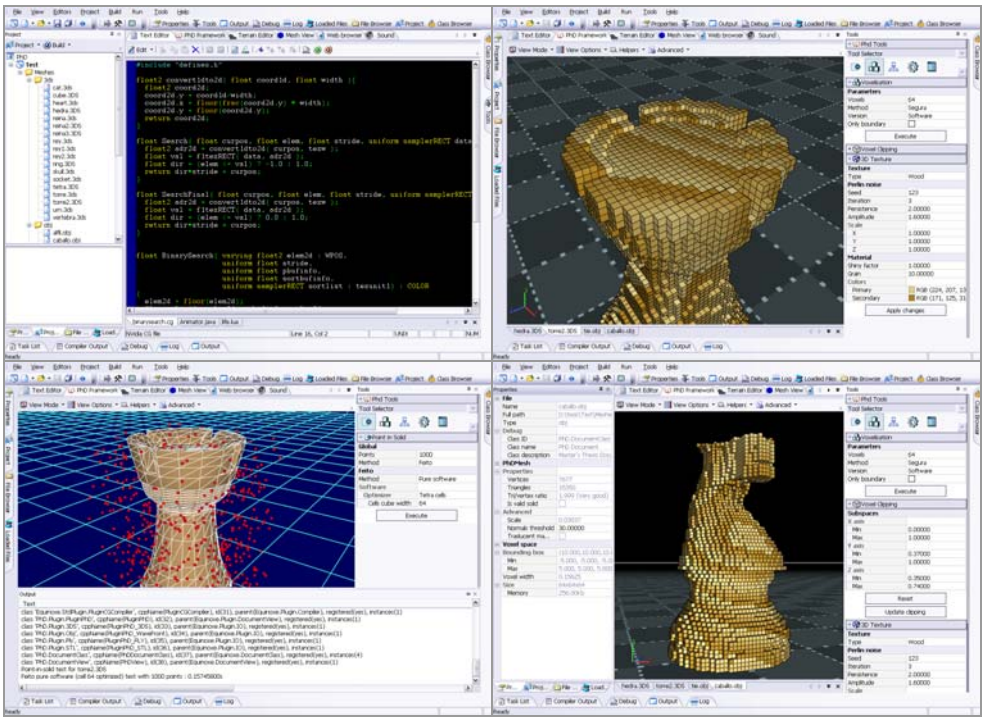


Figura B-5. Además de ofrecer soporte para múltiples gestores de documentos a través de módulos (plugins), se permite trabajar con varios archivos dentro de la misma vista. La interfaz es totalmente personalizable, pudiendo trabajar con distintos tipos de escritorio y configuraciones de ventanas. Cualquier aplicación que se añade al entorno se beneficia de estos servicios.

desarrollo. Esta forma de organización del trabajo se utiliza en entornos como Visual Studio. La idea es extenderla a todo tipo de documentos (Figura B-6).

- La interfaz de usuario debe ofrecer mecanismos para añadir funcionalidad adaptada a cada módulo dentro del entorno. Esto se lleva a cabo a través de paneles adosados que pueden adoptar cualquier configuración en la ventana principal. Cada módulo puede definir sus propios paneles para su parte específica de interfaz.

Con este sistema se simplifica el desarrollo de casi cualquier software, ya que la parte básica de la interfaz de usuario queda resuelta desde el principio. Cuando se implementa un software sencillo, ya se dispone desde el principio de la opción de trabajar con proyectos, opciones de ficheros, menús a varios niveles y otras opciones,

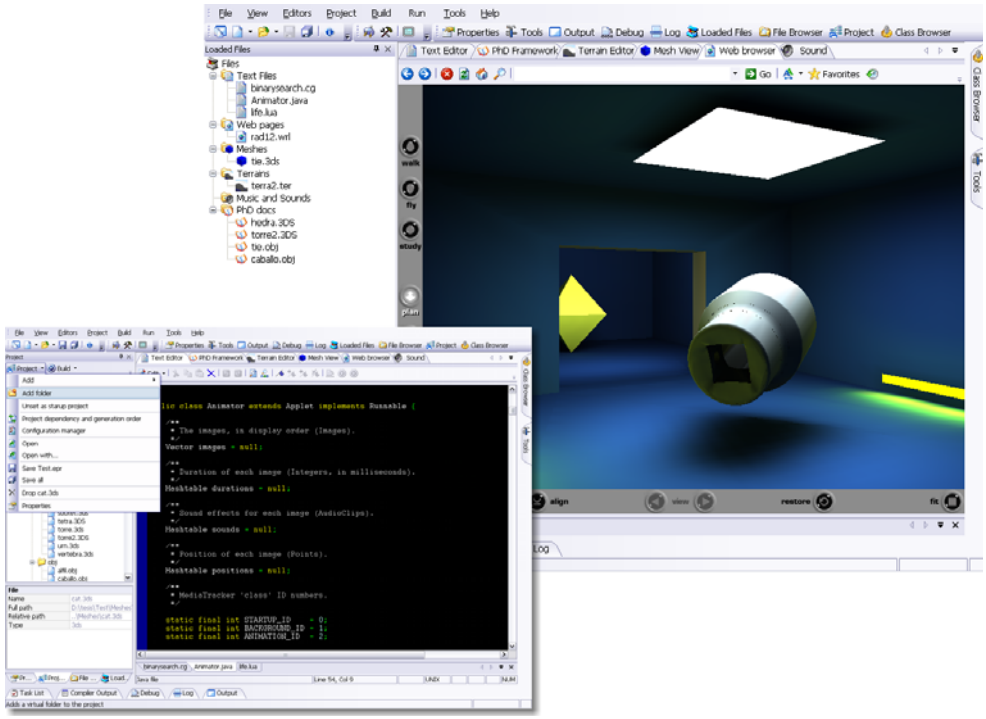


Figura B-6. La gestión de proyectos (abajo) permite organizar todo tipo de documentos y editarlos con distintos módulos programables. La incrustación de aplicaciones externas (arriba; cliente VRML Cortona) aporta flexibilidad al entorno y ofrece nuevas posibilidades para el tratamiento de datos de cualquier tipo.

todo ello dentro de un entorno que ofrece una interfaz de alta calidad. Estos aspectos suelen representar en un proyecto pequeño un coste relativo demasiado grande, por lo que no suelen implementarse. Por ejemplo, los servicios de interfaz se usuario que ofrece el nuevo entorno trasladados a Solid supondrían una cantidad de líneas de código fuente mayor que la del propio Solid.

Ya que en el ámbito de la investigación se desarrollan multitud de pequeños programas, ninguno de éstos tiene volumen suficiente para justificar la inversión en el desarrollo de una interfaz de usuario completa y efectiva. Sin embargo, el desarrollo de un entorno que proporcione servicios básicos a múltiples aplicaciones permite que éstas tengan acceso a una funcionalidad e interfaz profesional sin coste añadido.

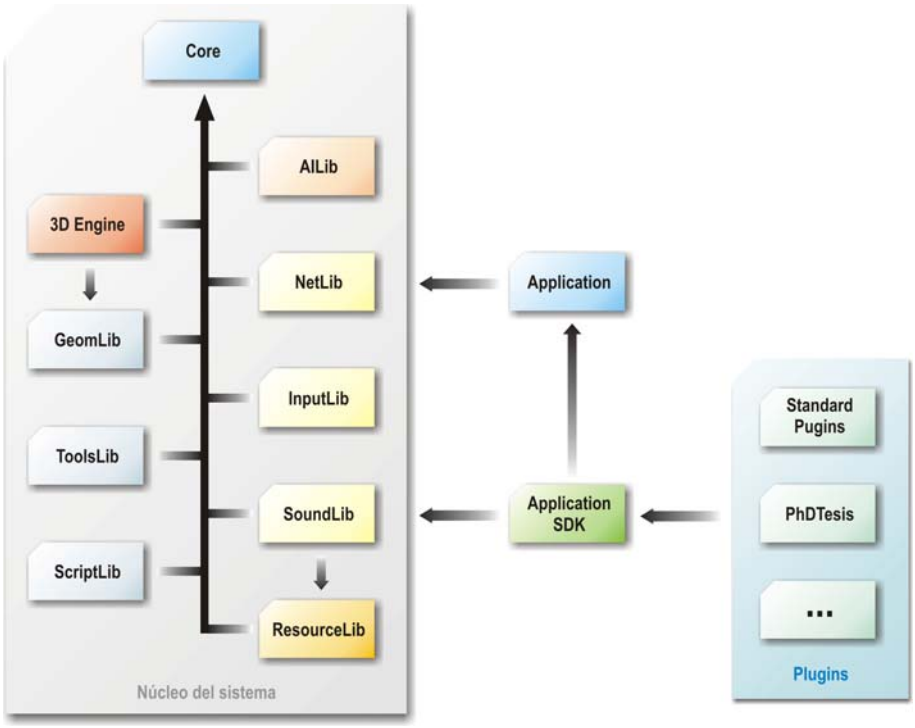


Figura B-7. Arquitectura del nuevo software. El núcleo del sistema se compone de un conjunto de librerías especializadas e interrelacionadas tal y como indican las flechas. La aplicación principal y su SDK dependen directamente del núcleo. Los módulos externos (plugins) se comunican con el resto del sistema a través de la librería de desarrollo (SDK).

B.4.2 Estructura del sistema

El nuevo software, al igual que Solid, está escrito en C++ y se ha desarrollado con Visual Studio. La estructura general se basa en una jerarquía de clases que lo divide en unidades funcionales. La Figura B-7 muestra un esquema básico de las unidades funcionales principales del sistema.

El núcleo principal se compone de una serie de librerías especializadas que proporcionan gran parte de los servicios necesarios para una aplicación de realidad virtual o Informática Gráfica. Cada una de las librerías se compila por separado y genera un archivo DLL (Dynamic Link Library). El núcleo siempre está abierto a ampliaciones, y actualmente ofrece las siguientes opciones:

- **Core**. Este módulo es la base del resto del sistema. Proporciona clases, macros, estructuras y utilidades básicas de monitorización del rendimiento (profiler), depuración (debug), bitácora (log), gestión controlada de memoria, recolector de basura (garbage collector), definición de tipos de datos base para compilación portable, ficheros abstractos, y un largo etcétera.

- **3D Engine**. Este complejo módulo, junto con la librería geométrica, proporciona múltiples servicios relacionados con el rendering de escenas 3D. Además de servir como enlace con OpenGL y Direct3D, aporta utilidades relacionadas con mallas de triángulos, materiales, gestión dinámica de texturas, estados de rendering, etc.

- **GeomLib**. La librería geométrica aporta toda la base matemática necesaria para la manipulación de información 3D, incluyendo clases para la gestión de puntos, vectores, matrices y quaternions. También se incluyen clases para manipular entidades geométricas como planos, tetraedros, envolventes convexas, etc.

- **ToolsLib**. Esta librería proporciona clases de utilidad múltiple pensadas para complementar el motor 3D. Se incluyen todo tipo de estructuras espaciales como BSP, octrees, grids, mapas de elevación para terrenos, así como métodos numéricos para la resolución de ecuaciones, interpolación, cálculo, teselación por Delaunay, etc.

- **ScriptLib**. La librería de scripts permite la compilación y ejecución de ficheros script orientados a la manipulación de varias funciones del sistema de forma programada. Actualmente sólo soporta *Lua*, aunque está diseñado para admitir otros lenguajes como *Python*. Las clases que heredan de *PublicClass* (ver Figura B-8) están automáticamente disponibles para ser accedidas desde scripts.

- **AILib**. La librería de inteligencia artificial proporciona clases y métodos para la resolución de algunos problemas de este ámbito, lo que incluye lógica difusa, redes neuronales, algoritmos genéticos, algoritmos de búsqueda, etc.

- **NetLib**. Este módulo está encargado de ofrecer servicios de red a nivel de aplicación. Está pensado para la comunicación entre cualquier software desarrollado dentro del entorno de forma distribuida o en arquitectura cliente-servidor.

- **InputLib**. Proporciona servicios de gestión de dispositivos de entrada, la mayoría a través de DirectInput (parte de DirectX). Ya que DirectInput ofrece acceso a un nivel bastante bajo, este módulo proporciona una abstracción que facilita las tareas de detección, configuración y control de dispositivos de entrada.

- **SoundLib**. Proporciona servicios de audio a alto nivel. Internamente utiliza DirectSound (parte de DirectX), pero ofrece una interfaz mucho más simplificada. Contiene capacidad completa de reproducción de sonido envolvente sobre cualquier hardware compatible con DirectX, así como posibilidades de manejar formatos *wave*, *mp2*, *mp3*, *ac3* y *ogg vorbis*.

- **ResourceLib**. Proporciona servicios de empaquetado y compresión de archivos mediante sistemas de ficheros virtuales. El acceso a un archivo se realiza especificando su ruta dentro de un fichero maestro comprimido. De esta forma puede operarse sobre datos empaquetados, comprimidos y encriptados como si se tratase de un fichero normal en operaciones de lectura, y de forma algo más restringida en operaciones de escritura. También ofrece soporte para archivos *zip*.

El resto del sistema queda dividido en tres grandes unidades funcionales que dependen directa o indirectamente del núcleo. Estas unidades forman la aplicación principal y los módulos asociados.

- **Application**. Además de constituir la aplicación principal que sirve como entorno donde se acoplan el resto de módulos, también proporciona los servicios de interfaz de usuario.

- **Application SDK**. La librería de desarrollo o SDK (Software Development Kit) proporciona el acceso a todas las partes del sistema para la construcción de módulos. Esta librería es el componente necesario para desarrollar aplicaciones compatibles con el entorno.

- **Plugins**. Los módulos son en realidad ficheros DLL que se vinculan al entorno en tiempo de ejecución. Cada módulo se valida a través de un punto de entrada que proporciona información al entorno. Cuando éste se inicia, vincula todos los módulos disponibles, prepara la interfaz principal para alojar las vistas y configura los menús y opciones pertinentes para comenzar a trabajar.

La Figura B-8 muestra las clases más importantes relacionadas con la gestión de documentos y módulos programables. Estas clases son una implementación del

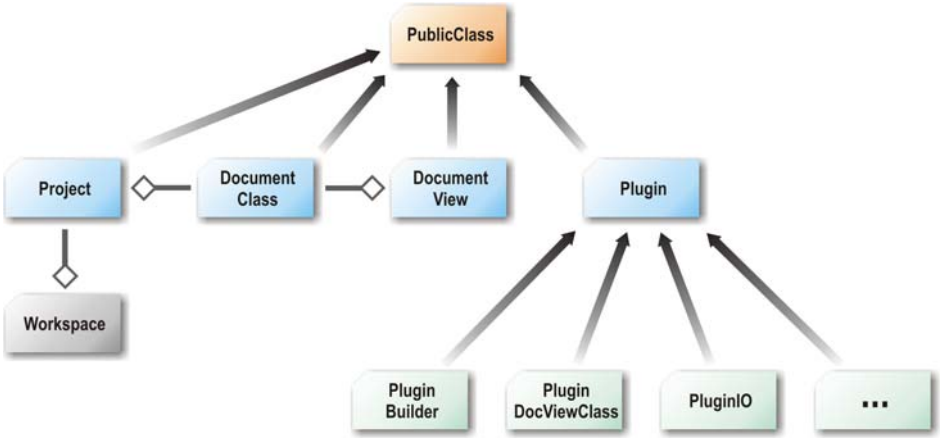


Figura B-8. Diagrama de las clases más importantes del sistema desde el punto de vista de la arquitectura Documento-Vista. Se muestran también algunas clases utilizadas para implementar módulos externos. Todas las clases que heredan de PublicClass están disponibles a través de lenguajes de script para la programación avanzada a muy alto nivel.

enfoque Documento-Vista, que proporciona soporte para múltiples documentos y múltiples vistas sobre cada uno. La clase *PublicClass* proporciona opciones básicas para las clases más importantes y ofrece de forma casi automática acceso a las mismas a través de lenguaje de script. Las clases *DocumentClass* y *DocumentView* sirven para definir tipos de documentos (y sus datos asociados) y vistas de los mismos (incluyendo la interfaz), respectivamente. La clase *Project* sirve para representar proyectos mientras que *Workspace* representa espacios de trabajo que se componen de grupos de proyectos. Por último, la clase *Plugin* y derivados definen tipos de módulos que sirven como base para el desarrollo de aplicaciones compatibles con el entorno.

El entorno ideal es aquel que permite la integración de módulos desarrollados en múltiples lenguajes. En principio se diseñó para dar servicio sólo a aplicaciones escritas en C y C++, sin embargo sería posible implementar conexiones que permitieran al entorno comunicarse con módulos desarrollados en otros lenguajes. Esto se ha llevado a cabo con éxito en Windows para comunicar DLLs escritas en distintos lenguajes, como Visual Basic o Java. Hay que recordar que cualquier aplicación desarrollada para ser integrada en el entorno debe estar construida utilizando su librería de desarrollo.

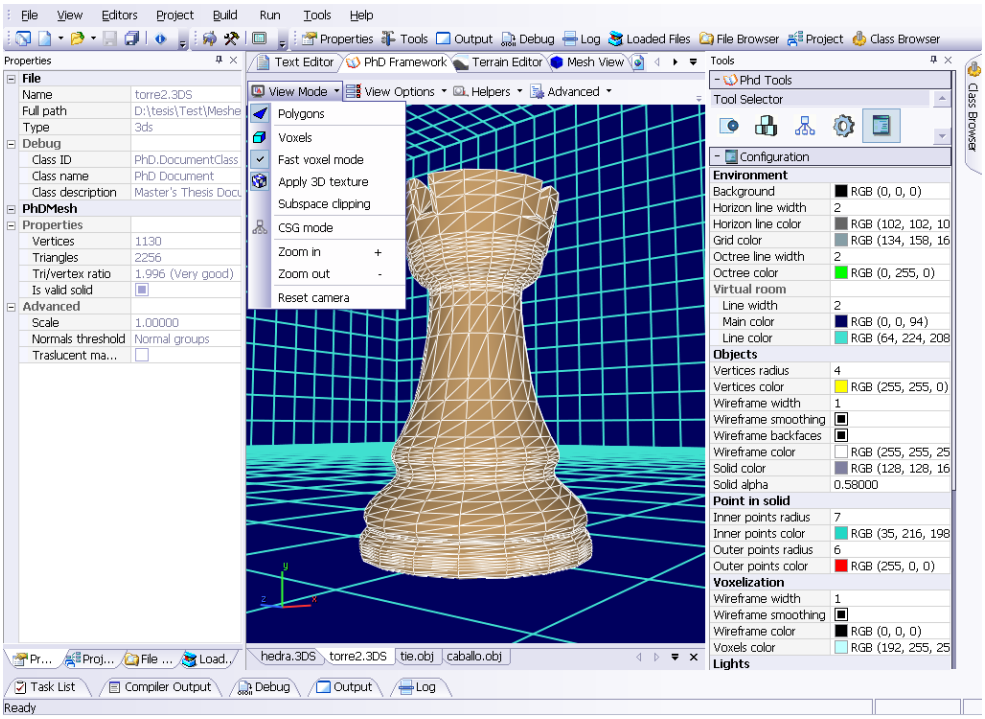


Figura B-9. La integración de Solid en el nuevo entorno permite ofrecer nuevas opciones de interfaz sin coste de desarrollo añadido gracias a los servicios ofrecidos.

B.4.3 Integración de Solid

Solid es un ejemplo claro de aplicación que debe desarrollarse manteniendo al mínimo todo esfuerzo dedicado a tareas no esenciales, esto es, todo lo que no sean los algoritmos geométricos que sirven para demostrar las premisas de esta tesis. Aunque el desarrollo de Solid está completo, su adaptación al entorno permite continuar la implementación de nuevas versiones de los algoritmos de una forma más eficiente.

Para la conversión de Solid al entorno de desarrollo se ha reimplementado la unidad funcional correspondiente a la interfaz, que ahora se adapta a los servicios ofrecidos por el entorno. En lugar de mostrar pequeñas ventanas para la introducción de parámetros para cada algoritmo, se ofrece un panel organizado en secciones que permite establecer los valores necesarios en cada momento. Esta nueva interfaz es mucho más cómoda de utilizar y extremadamente fácil de mantener (en cuanto a cambios en el software). El núcleo de Solid ha sufrido algunas modificaciones en las

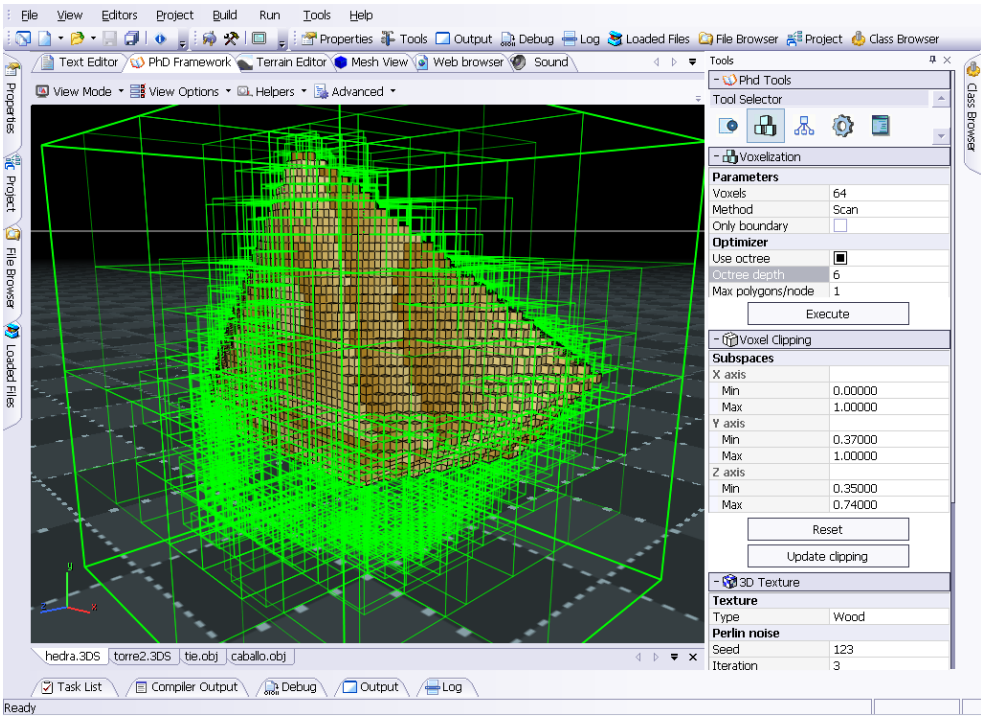


Figura B-10. Además de ofrecer soporte para múltiples gestores de documentos a través de módulos, se permite trabajar con varios archivos dentro de la misma vista. La interfaz es totalmente personalizable, pudiendo trabajar con distintos tipos de escritorio y configuraciones de ventanas.

unidades funcionales dedicadas a la comunicación con OpenGL, ya que el nuevo sistema proporciona servicios dedicados a esta API. La carga de datos se realiza a través del entorno mediante asociaciones con extensiones de archivos.

Las clases más importantes del núcleo de Solid se mantienen intactas, con lo que la conversión del software se ha realizado de forma rápida. No obstante, se ha modificado ligeramente el código para aprovechar los servicios de depuración (debugger), bitácora (logs) y análisis del rendimiento (profiler) que ofrece el entorno. Con estas opciones se obtienen datos muy precisos sobre el desempeño del código fuente y del rendimiento de los algoritmos.

B.5 Conclusiones

En este capítulo se han presentado unas nociones sobre el software desarrollado y el hardware utilizado para la obtención de los resultados presentados a lo largo de esta tesis. Tanto los diseños como las implementaciones de Solid y el nuevo sistema han sido llevados a cabo íntegramente por el autor de este trabajo, por lo que seguramente son susceptibles de ser mejorados en muchos aspectos. Sin embargo, Solid ha desempeñado satisfactoriamente todas las funciones para las que fue diseñado, y aunque se trate de un proyecto reducido contiene una considerable cantidad de implementaciones muy optimizadas de diversos algoritmos. Además ha servido como punto de partida para el diseño de un nuevo software que será de gran utilidad en investigaciones futuras. Este nuevo sistema permite un desarrollo cómodo, rápido y eficiente de la mayoría de las tareas relacionadas con la investigación en Informática Gráfica, a la vez que proporciona al programador una interfaz de usuario de calidad sin coste de desarrollo añadido.


Google
Google
Google
Google
Bibliografía
home
feedback
join
go shopping
The First Society in Computing
Association for Computing Machinery
ACM delivers resources that advance computing as a science and a profession. ACM provides the computing field's premier Digital Library and serves its members and the computing profession with leading-edge publications, conferences, and career resources.
Membership
Professionals
Students
Advanced Member Grades
Need to Renew?
Portal
The Digital Library
The Guide to Computing Literature
Publications
Queue: Latest computing trends
Ubiquity: IT opinion magazine and forum
TechNews: News Gathering Service for IT Professionals
eLearn: Distance learning magazine
MemberNet: Your Key to the World of ACM...and Beyond
Computers in Entertainment: New ACM online magazine
Special Interest Groups (SIGS)
What's New
Participate in ACM's 2006/2007 Member-Get-A-Member Drive!
2006 ACM Council Election Underway
ACM Announces Advanced Member Grades
Call for Nominations -TOSEM Editor-in-Chief
Software Pioneer Peter Naur Wins ACM's 2005 Turing Award
Announcing the 2005 ACM Award Winners
Join ACM
Join SIGs
Subscribe to Publications
Purchase
Books
Proceedings
Manage/Create your acm.org web account
Access your Message Center
Explore
Online Books
Online Courses
Check System Availability
Mail Alerts
View
Conference Calendar

Bibliografía

[Abra97] Abrash, M., ***Michael Abrash's Graphics Programming Black Book***, Special Edition. The Coriolis Group, inc., Scottsdale, Arizona, 1997.

[Ayala85] Ayala, D., Brunet, P., Joan, R., Navazo, I., ***Object Representation by Means of Nonminimal Division of Quadtrees and Octrees***, ACM Transactions on Graphics, 4(1), pp. 41-59, 1985.

[Badouel90] Badouel, D. ***An Efficient Ray-polygon Intersection***, in Graphic Gems. Andrew S. Glassner, Academic Press, pp. 390-393, 1990.

[Baer79] Baer, A., Eastman, C., Henrion, M. ***Geometric Modeling: a Survey***, Computer Aided Design, 11(5), pp 253-272, 1979.

[Baumgart75] Baumgart, B., ***A Polyhedron Representation for Computer Vision***, National Computer Conference, pp. 589-596, AFIPS Conf. Proc., 1975.

[Berg97] Berg, M., ***Trends and Development in Computational Geometry***, computer Graphics Forum 16(1), 1997.

[Bern92] Bern, M., Eppstein, D. ***Mesh Generation and Optimal Triangulation Computing in Euclidean Geometry***, World Scientific, 1992.

[Blundell00] Blundell, B., Schwarz, A. ***Volumetric Three-Dimensional Display Systems***. John Wiley, 2000.

[Brunet85] Brunet, P., Navazo, I., ***Geometric Modeling Using Exact Octree Representation of Polyhedral Objects***. Eurographics'85, pp. 159-169, 1985.

[Brunet90] Brunet, P., Navazo, I., ***Solid Representation and Operation Using Extended Octrees***. ACM Transaction on Graphics, Vol. 9, No. 2, 1990.

[Brunet90b] Brunet, P., ***Face Octrees. Envolved Algorithms and Applications***. Documento de trabajo LSI-90-14, Dpto. Lenguajes y Sistemas Informáticos, Universidad Politécnica de Cataluña, Barcelona, 1990.

[Brunet92] Brunet, P., Juan, R. Navazo, I., ***Octree Representations in Solid Modeling*** in Progress in Computer Graphics, pp 164-215, 1992.

[Buck04] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P., ***Brook for GPUs: Stream Computing on Graphics Hardware***. SIGGRAPH 2004 Paper Talk August 12, 2004.

[Buck04b] Buck, I., Purcell, T., ***A Toolkit for Computation on GPUs***, in GPU Gems, NVidia, 2004.

[Cabral94] Cabral, B., Cam, N., Foran, J., ***Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware***. Proceedings of IEEE Symposium on Volume Visualization, pp 91-98, 1994.

[Cano02] Cano, P., Torres, J.C., ***Representation of Polyhedral Objects Using SP-Octrees***. Journal of WSCG, 10(1):95-101. 2002.

[Cano04] Cano, P. ***SP-Octree, Representación Jerárquica de Sólidos Poliédricos***. PhD Thesis, Universidad de Granada, 2004.

[Chazelle91] Chazelle, B., ***Triangulating a Simple Polygon in Linear Time***. Dicrete & Computacional Geometry (6). 1991.

[Cook84] Cook, R., ***Shade Trees***. ACM Siggraph, 1984.

[Duce88] Duce, D., ***Theoretical Foundation of Computer Graphics and CAD***, in Format Specification of Computer Graphics. Springer-Verlag. 1988.

[Durst89] Durst, M.J., Kunii, T.L. ***Integrated Polytrees: A Generalized Model for Integrating Spatial Decomposition and Boundary Representation***, in Theory and Practice of Solid Modelling. Springer Verlag, 1989

[Eberly01] Eberly, David H. ***3D Game Engine Design***, Morgan Kaufmann, pp 417-426.

[Ebert99] Ebert, D., Bedwell, E., Maher, S., Smoliar, L. Downing, E. ***Realizing 3D Visualition Using Crossed-Beam Volumetric Displays***. Communications of the ACM, 42(8):101-107, 1999.

[Elber99] Elber, G., Kim, M., ***Special Isuue on Sweeps and Minkowski Sums***. Computer Aided Design, 31, 1999.

[Fang00] Fang, S., Cheng, H., ***Hardware Accelerated Voxelization***. Computer & Graphics, 24, 433-442, 200.

[Feito95] Feito, F., Torres, J., Ureña, A., ***Orientation, Simplicity and Inclusion Test for Planar Polyhedron***. Computer & Graphics, 1995, 19, 595-600.

[Feito95b] Feito, F., ***Modelado de Sólidos y Álgebra de Objetos Gráficos***. PhD thesis, Depto. Lenguajes y Sistemas Informáticos, Universidad de Granada, 1995.

[Feito97] Feito, F.R. and Torres, J.C. ***Inclusion Test for General Polihedra***. Computer & Graphics, 21(1):23-30, 1997.

[Feito97b] Feito, F., Torres, J.C. ***Boundary Representation of Polyhedral Heterogeneous Solids in the context of a Graphic Object Algebra***. The Visual Computer (13), 64-77, 1997.

[Fernando03] Fernando, R., Kilgard, M.J., ***The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics***. NVidia, Addison-Wesley, 2003.

[Fiume89] Fiume, E., ***The Mathematical Structure of Raster Graphics***. Academic Press. 1989.

[Foley96] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., ***Computer Graphics. Principles and Practices***. 2nd Edition. Addison-Wesley, 1996.

[Freytag04] Freytag, M., Shapiro, V., ***B-rep SE: Simplicially Enhanced Boundary Representation***. ACM Symposium on solid Modeling and Applications, 2004.

[Frisken98] Frisken, F.S., ***Using Distance Maps for Accurate Surface Representation in Sampled Volumes***. IEEE Symposium on Volume Visualization, pp. 23-30, 1998.

[Fuchs80] Fuchs, H., Kedem, Z.M., Naylor, B.F., ***On Visible Surface Generation by A Priori Tree Structures***, Computer Graphics (SIGGRAPH '80 proceedings), pp 124-133, July 1980.

[Gargantini93] Gargantini, I. and Atkinson, H.H. ***Ray Tracing an Octree: Numerical Evaluation of the First Intersection***. Computer Graphics Forum, 12(4), 1993.

[Glassner84] Glassner, A.S. ***Space Subdivision for Fast Ray Tracing***. IEEE Comput. Graph. Appl., 4(10):15-22, October 1984.

[Glassner89] Glassner, A., ***An Introduction to Ray Tracing***, Academic Press, 1989, pp. 217-220.

[Gordon91] Gordon, D., Shulong, C., ***Front-to-Back Display or BSP Trees***, IEEE Computer Graphics and Applications, vol. 11, no. 5, pp. 79-86, September 1991.

[Green05] Green, S., ***Implementing Improved Perlin Noise***, in GPU Gems 2, NVidia, 2005.

[Haumont02] Haumont, D., Warze, N. ***Complete Polygonal Scene Voxelization***. Journal of Graphics Tools, 7(3) pp 27-41, 2002.

[Harris05] Harris, M., ***Mapping Computacional Concepts to GPUs*** in GPU Gems 2, NVidia, 2005.

[Hasselgren05] Hasselgren, J., Akenine-Möller, T., Ohlsson, L., ***Conservative Rasterization*** in GPU Gems 2, NVidia, 2005.

[Heras05] Heras, S.M., Valenzuela, A., Ogayar, C., Valverde, A.J., Torres, J.C., ***Computer-Based Production of Comparison Overlays from 3D-Scanned Dental Casts for Bite Mark Analysis***. Journal of Forensic Sciences. Vol. 50 (1). January 2005.

[Hoffmann89] Hoffmann, C. M., ***Geometric and Solid Modeling: an Introduction***. Morgan Kaufmann, 1989.

[Horn89] Horn, W., Taylor, D. L., ***A Theorem to Determine the Spatial Containment of a Point in a Planar Polyhedron***. Computer Vision, Graphics and Image Processing, 1989, 45, 106-116.

[Huang98] Huang, J., Yagel, R., Filippov, V., Kurzion, Y. ***An Accurate Method for Voxelizing Polygon Meshes***. Proceedings of the IEEE symposium on Volume Visualization, pp 119-126, 1998.

[Hui94] Hui, K.C., ***Solid Modeling with Sweep-CSG Representation***. Proceedings of the CSG 94 Set Theoretic Solid Modeling; Techniques and Applications, pp. 119-131, 1994.

[James99] James, A. ***Binary Space Partitioning for Accelerated Hidden Surface Removal And Rendering of Static Environments***, Ph.D. Thesis, University of East Anglia, August 1999.

[Jimenez05] Jiménez, J.J., Ogayar, C.J., Segura, R.J., Feito, F.R., ***Detección de Colisión entre un Sólido y una Nube de Partículas mediante el uso de Hardware Gráfico Programable***. XV Congreso Español de Informática Gráfica (CEIG 2005). Granada (Spain), 2005.

[Jimenez06] Jiménez, J.J., Feito, F.R., Segura, R.J., Ogayar, C.J., ***Particle Oriented Collision Detection using Simplicial Coverings and Tetra-Trees***. Computer Graphics Forum, 25(1), 2006.

[Joan96] Joan, R., Vinacua, A., Brunet, P., ***Classification of a Point with Respect to a Polyhedron Vertex***. International Journal of Computational Geometry and Applications, 6(2), pp. 157-167, 1996.

[Jones96] Jones, M.W. ***The Production of Volume Data from Triangular Meshes Using Voxelization***. Computer Graphics Forum, 15, No. 5, pp. 311-318. 1996.

[Kalay82] Kalay, Y. E., ***Determining the Spatial Containment of a Point in General Polyhedra***. Computer Graphics and Image Processing, 1982, 19(1). 203-334.

[Kapasi03] Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Ahn, J.H., Mattson, P., Owens, J.D., ***Programmable Stream Processors***. IEEE Computer, pp. 54-62, 2003.

[Karasick88] Karasick, M., ***On the Representation and Manipulating of Rigid Solid***. PhD Thesis, Cornell University, 1988.

[Karabassi02] Karabassi, E., Papaioannou, G., Theoharis, T. ***A Fast Depth-Buffer-Based Voxelization Algorithm***. ACM Journal of Graphics Tools, 4, pp. 114-124, 2002.

[Kaufman93] Kaufman, A. Cohen, D. Yagel, R. ***Volume Graphics***. IEEE Computer 26(7), pp. 51-64. 1993.

[Kaufman94] Kaufman, A. ***Trends in Volume Visualization and Volume Graphics***. Scientific Visualization, pp 3-19, Academic Press Ltd., 1994.

[Kilgariff05] Kilgariff, E. Fernando, R, ***The Geforce 6 Series GPU Architecture***, in GPU Gems 2, NVidia, Addison Wesley, 2005.

[Kobbelt04] Kobbelt, L., Botsch, M., ***A Survey of Point-Based Techniques in Computer Graphics***, Computers & Graphics, 28(6), pp. 801-814, 2004.

[Kovach00] Kovach, P.J., editor, ***Inside Direct3D***, Microsoft Programming Series, 2000.

[Lane84] Lane, J., Magedson, B., Rarick, M. N., ***An Efficient Point in Polyhedron Algorithm***. Computer Vision, Graphics and Image Processing, 1984, 26, 118-125.

[Lefebvre05] Lefebvre, S., Hornus, S., Neyret, F., ***Octree Textures on the GPU*** in GPU Gems 2. NVidia, 2005.

[Lefohn04] Lefohn, A., ***GPU Data Formatting and Addressing***. GPGPU, SIGGRAPH 2004.

[Lefohn05] Lefohn, A., Kniss, J., Owens, J., ***Implementing Efficient Parallel Data Structures on GPUs*** in GPU Gems 2. NVidia, 2005.

[Lischinski94] Lischinski, D., ***Incremental Delaunay triangulation***. in Graphics Gems IV, pp 47-59. Academic Press, 1994.

[Mäntylä83] Mäntylä, M., Tammine, M. ***Localized Set Operations for Solid Modelling***. Computer Graphics, 17(3), 1983.

[Mäntylä88] Mäntylä, M. ***An Introduction to Solid Modeling***. Computer Science Press, 1988.

[Microsoft04] Microsoft Corporation, ***Vertex Shader 3.0 Spec & Pixel Shader 3.0 Spec***. Disponible en http://msdn.microsoft.com, 2004.

[Möller97] Möller, T. and Trumbore, B. ***Fast, Minimum Storage Ray-Triangle Intersection***. Journal on Graphics Tools, 2(1):21-28, 1997.

[Möller97b] Möller, T. ***A Fast Triangle - Triangle Intersection Test***. Journal of Graphics Tools, vol. 2, pp 25-30. A.K. Peters, 1997.

[Möller02] Möller, T., Haines, E. ***Realtime Rendering***. Second edition. A. K. Peters, 2002, pp 349-353, 583-586, 596-597.

[Moore65] Moore, G., ***Cramming More components onto Integrated Circuits***. Electronics, 38(8), 1965.

[Moreland03] Moreland, K., Angel, E. ***The FFT on a GPU***. Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003, pp. 112-120.

[Mortenson85] Mortenson, M.E. ***Geometric Modelling***. John Wiley & Sons, 1985.

[Mortenson97] Mortenson, M.E. ***Geometric Modelling, 2nd Edition***. Wiley Computer Publishing, 1997.

[Munkres01] Munkres, J.R., ***Topología***. 2ª Edición (en castellano), Prentice Hall, 2001.

[Navazo87] Navazo, I., Fontdecaba, J., Brunet, P. ***Extended Octrees, Between CSG and Boundary Representations***. Euragraphics'87, pp. 239-247, 1987.

[Navazo89] Navazo, I., ***Extended Octree Representation of General Solids with Plane Faces: Model Structure and Algorithms***. Computer & Graphics, 13, pp. 5-16, 1989.

[Naylor90] Naylor, B. ***Binary Space Partitioning Trees as an Alternative Representation of Polytopes***. Computer Aided Design, 22(4), pp. 250-252, 1990.

[Naylor90b] Naylor, B., Amanatides, J., Thibault, W. ***Merging BSP Trees Yields Polyhedral Set Operations***. ACM Computer Graphics, 24(4), pp. 115-124, 1990.

[NVidia04] NVidia Corp. ***NVidia Cg Toolkit: User´s Manual***. 2004.

[Ogayar03] Ogayar, C.J., Segura, R.J., Feito, F.R., ***Técnicas para el Cálculo de Inclusión de Puntos en Mallas de Triángulos. Estudio Comparativo***. XIII Congreso Español de Informática Gráfica (CEIG 2003). A Coruña (Spain), 2003.

[Ogayar03b] Ogayar, C.J., Segura, R.J., Feito, F.R., ***Point In Solid Tests for Triangle Meshes. Comparative Study***. Annual Conference of the European

Bib

Association for Computer Graphics, EuroGraphics 2003, Granada (Spain). Sep. 2003.

[Ogayar04] Ogayar, C.J. Segura, R.J., ***Optimización de Algoritmos Geométricos Básicos mediante el uso de Recubrimientos Simpliciales***, en Plataforma Avanzada de Modelado Paramétrico en CAD (ISBN 84-609-2575-7), 2004.

[Ogayar05] Ogayar, C.J., Segura, F.R., Feito, F.R., ***Point in Solid Strategies***. Computers & Graphics, vol 29(4), 2005.

[Ogayar05b] Ogayar, C.J., Jiménez, J.J., Segura, R.J., Feito, F.R., ***Inclusión de Puntos en Sólidos mediante el uso de Hardware Gráfico Programable***. XV Congreso Español de Informática Gráfica (CEIG 2005). Granada (Spain), 2005.

[Ogayar06] Ogayar, C.J., Jiménez, J.J., Segura, R.J., Feito, F.R., ***Point in Solid Test on the GPU***. Technical Report, Departamento de Informática, Universidad de Jaén, 2006.

[Ogayar06b] Ogayar, C.J., Rueda, A.J., Segura, F.R., Feito, F.R., ***Fast and simple hardware accelerated voxelizations using simplicial coverings***. Technical Report, Departamento de Informática, Universidad de Jaén, 2006.

[Ogayar06c] Ogayar, C.J., Feito, F.R., Segura, F.R., Rivero, M.L. ***GPU-based Evaluation of Boolean Operations on Triangulated Solids***. Symposium Ibero Americano de Computación Gráfica, SIACG 2006.

[ORourque94] O'Rourque, J., ***Computational Geometry in C***. Cambridge University, 1994.

[Owens05] Owens, J., ***Streaming Architectures and Technology Trends*** in GPU Gems 2, NVidia, 2005.

[Owens02] Owens, J.D., ***Computer Graphics on a Stream Architecture***. PhD Thesis, Stanford University, 2002.

[Passalis04] Passalis, G., Kakadiaris, I. A., Theoharis, T. ***Efficient Hardware Voxelization***, Proceedings of the Computer Graphics International, 2004.

[Pastoor97] Pastoor, S., Kiesewetter, R. ***3-D Displays: A Review of Current Technologies DISPLAYS***, 17, pp. 100-110, 1997.

[Paterson90] Paterson, M.S., Yao, F.F., ***Optimal Binary Space Partitions for Orthogonal Objects***, Proceedings of 1st Symposium on Discrete Algorithms, pp. 100-106, 1990.

[Peeper03] Peeper, C., Mitchell, J.L., ***Introduction to the DirectX 9 High Level Shading Language***, in W. Engel, editor, ShaderX² – Introduction and Tutorials with DirectX 9. Wordware, 2003.

[Pender03] Pender, T., ***UML Bible***, Wiley Publishing, 2003.

[Perlin85] Perlin, K., ***An Image Synthesizer***, Computer Graphics. Proceedings of ACM SIGGRAPH 85, 24(3). 1988.

[Perlin02] Perlin, K., ***Improving Noise***, Proceedings of ACM SIGGRAPH 2002.

[Pilz89] Pilz, M., Kamel, H.A., ***Creation and Boundary Evaluation of CSG Models***, Engineer Computer, 5, pp. 105-118, 1989.

[Proudfoot01] Proudfoot, K., Mark, W.R., Tzvetkov, S., Hanrahan, P., ***A Real-Time Procedural Shading System for Programmable Graphics Hardware***, ACM SIGGRAPH 2001.

[Purcell02] Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P., ***Ray Tracing on Programmable Graphics Hardware***, ACM Transactions on Graphics 21(3), pp. 703-712.

[Purcell03] Purcell, T.J., Donner, C., Cammamaro, M., Jensen, H.W., Hanrahan, P., ***Photon Mapping on Programable Graphics Hardware***, Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003, pp. 41-50.

[Requicha78] Requicha, A.A.G., Tilove, R.B., ***Mathematical foundations of Constructive Solid Geometry: General Topology of Regular Closed Sets***. Tech. Memo, 27a. Production Automation Project, University of Rochester, NY, 1978.

[Requicha80] Requicha, A., ***Representation of Solid Objects: Theory, Methods and Systems***. Computing Surveys of the ACM, 12(4), pp: 437-464, 1980.

[Requicha82] Requicha, A., Voelcker, H., ***Solid Modelling: A Historical Summary and Contemporary Assesment***. IEEE Computer Graphics and Applications, 2(3), pp: 9-24, 1982.

[Requicha83] Requicha, A., Voelcker, H., ***Solid Modelling: Current Status and Research Directions***. IEEE Computer Graphics and Applications, 3(7), pp: 25-37, 1983.

[Requicha85] Requicha, A., Voelcker, H., ***Boolean Operations in Solid Modelling: Boundary Evaluations and Merging Algorithms***. Proceedings of the IEEE, vol. 73(1), 1985.

[Requicha88] Requicha, A., Tilove, R., ***Mathematical Foundations of Constructive Solid Geometry; General Topology of Closed Regular Sets***. Cornell Programmable Automation, 1988.

[Revelles00] Revelles, J., Ureña, C., Lastra, M., ***An Efficient Parametric Algorithm for Octree Traversal***. Journal of WSCG, vol 8(1), 2000.

[Ribelles02] Ribelles, J., Belmonte, O., Remolar, I., Chover, M., ***Multiresolution Modeling of Arbitrary Polygonal Surfaces: a Characterization***. Computers & Graphics, 26(3), 2002.

[Rivero00] Rivero, M., Feito, F.R., ***Boolean Operations on General Planar Polygons***. Computer & Graphics 24, 2000.

[Rivero02] Rivero, M.L., ***Algoritmos para las Operaciones Booleanas en 2D y 3D, bajo un Sistema de Representación Formal***, PhD Thesis, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada.

[Rivero04] Rivero, M., Feito, F.R., ***Refinamiento de Mallas Triangulares. Aplicación para el Cálculo de Operaciones Booleanas en 3D***. CEIG 2004, Sevilla 2004.

[Rivero06] Rivero, M., Feito, F.R., Segura, R.J., Ogayar, C.J., ***Algorithms for Boolean Operations in 3D***. Technical Report, Departamento de Informática, Universidad de Jaén, 2006.

[Rossignac89] Rossignac, J., Voelcker, H.B., ***Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection, and Shading Algorithms***, ACM Transactions on Graphics, 8(1), pp. 51-87, 1989.

[Rossignac99] Rossignac, J., Requicha, A.R., ***Solid Modeling***, In J. Webster, editor, Encyclopedia of Electrical and Electronics Engineering. Webster, John Wiley & Sons, 1999.

[Rossignac04] Rossignac, J., ***Surface Simplification and 3D Geometry Compression***. Chapter 54 in Handbook of Discrete and Computational Geometry (second edition), CRC Press, Editors: J. E. Goodman and J. O'Rourke. 2004.

[Rueda04] Rueda, A.J., Segura, R.J., Feito, F.R., Ruiz de Miras, J., ***Rasterizing complex polygons without tessellations***. Graphical Models 66(3): 127-132 (2004).

[Ruiz97] Ruiz, J., Feito, F., ***Modelado Geométrico con Parches Algebráicos: Test de Inclusión***. CEIG'97, 1997.

[Ruiz01] Ruiz, J., ***Modelado de Sólidos de Forma Libre***. Phd Thesis, Universidad de Granada, 2001.

[Samet84] Samet, H. ***The Quadtree and Related Hierarchical data Structures***. ACM Computing Surveys, 16(2), pp. 187-260, 1984.

[Samet88] Samet, H., Webber, R.E., ***Hierarchical Data Structures and Algorithms for Computer Graphics. Part I: Fundamentals***. IEEE Computer Graphics and Applications, 8, pp: 48-58, 1988.

[Samet88b] Samet, H., Webber, R.E., ***Hierarchical Data Structures and Algorithms for Computer Graphics. Part II: Applications***. IEEE Computer Graphics and Applications, 8, pp: 59-75, 1988.

[Samet89] Samet, H. ***The Design and Analysis of Spatial Data Structures***. Addison-Wesley, Reading, Massachusetts, 1989.

[Segura01] Segura, R. ***Modelado de Sólidos Mediante Recubrimientos Simpliciales***. PhD Thesis, 2001.

[Segura04] Segura, R.J., Feito, F.R., Ogayar, C.J., ***Voxelization of Solids Using Simplicial Coverings***. 12-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2004 (WSCG'2004), Plzen (Czech Republic). Vol. 2, 2004.

[Segura05] Segura, R, Feito, F.R., Ruiz de Miras, J., Torres, J.C., Ogayar, C. ***An Efficient Point Classification Algorithm for Triangle Meshes***. Journal of Graphics Tools, 10(3), A.K. Peters, 2005.

[Shapiro91] Shapiro, V., ***Representations of Semi-algebraic Sets in Finite Algebras Generated by Space Decompositions***. PhD Thesis, Cornell University, New York, 1991.

[Shapiro93] Shapiro, V., Vossler, D.L., ***Separations for Boundary to CSG Conversion***. ACM Transactions on Graphics, 12(1), pp. 35-55, 1993.

[Shapiro01] Shapiro, V., ***Solid Modeling***. en Handbook of Computer Aided Geometric Design, G. Farin, J. Hoschek, M.S. Kim, eds. Elsevier Science, 2001.

[Sramek99] M. Sramek, A. Kaufman, ***vxt: a C++ Class Library for Object Voxelization***, Proc. International Workshop Volume Graphics, 1999.

[Sramek99b] Sramek, M., Kaufman A. ***Alias-Free Voxelization of Geometric Objects***. IEEE Transactions on Visualization and Computer Graphics, 5(3), 1999.

[Tatarchuk05] Tatarchuk, N., Licea-Kane, B., ***GLSL Real-Time Shader Development***, in W. Engel, editor, Shader X^3 – Advanced Rendering with DirectX and OpenGL. Charles River Media, 2005.

[Thibault87] Thivault, W.C., Naylor, B., ***Set Operations on Polyhedra Using Binary Space Partitioning Trees***. ACM Computer Graphics, 21(4), pp. 153-162, 1987.

[Tilove80] Tilove, R.B., ***Set Membership Classification: a Unified Approach to Geometric Intersection Problems***. IEEE Transactions on Computers, C-29 (1), pp. 847-883, 1980.

[Torres92] Torres, J., ***Representación Abstracta de Objetos Gráficos. Teoría de Objetos Gráficos***. PhD thesis, depto. Lenguajes y Sistemas Informáticos, Universidad de Granada, 1992.

[Torres93] Torres, J., Clares, B., ***Graphic Object: A Mathematical Abstract Model for Computer Graphics***. Computer Graphics Forum 12(5), 1993.

[Upstill89] Upstill, S., ***The RenderMan Companion: A Programmer´s Guide to Realistic Computer Graphics***, Addison-Wesley, 1989.

[Vanecek89] Vanecek, G., ***Set Operations on Polyhedra Using Decomposition Methods***, PhD Thesis, University of Maryland, 1989.

[Watt00] Watt, A., ***3D Computer Graphics***, Third Edition, Addison-Wesley, pp. 51-54.

[Wloka03] Wloka, M., ***Batch, Batch, Batch: What Does It Really Mean?***, Game Developers Conference, 2003

[Woo97] Woo, M., Neider, J., Davis, T., ***OpenGL Programming Guide***, Addison-Wesley, 1997.

[Woodwark87] Woodwark87, J., ***Blends in Geometric Modeling***, in The Mathematics of Surfaces II, pp. 255-297, 1987.

[Zeller04] Zeller, C., Fernando, R., Wloka, M., Harris, M., ***Programming Graphics Hardware***, Eurographics 2004 tutorial series, 2004.

Índice

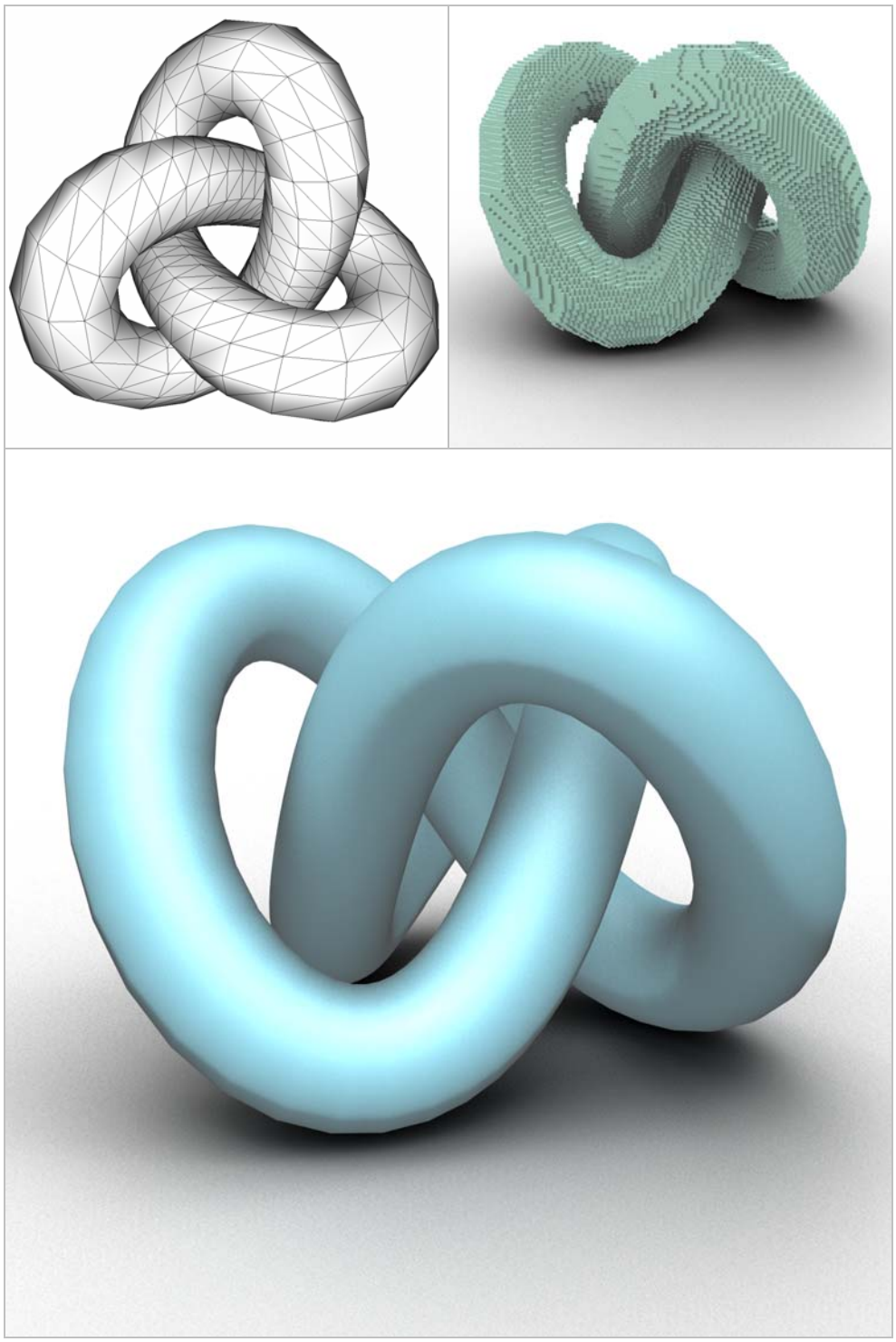


Lámina 1. Modelo Torus Knot. 808 vértices. 1200 triángulos.

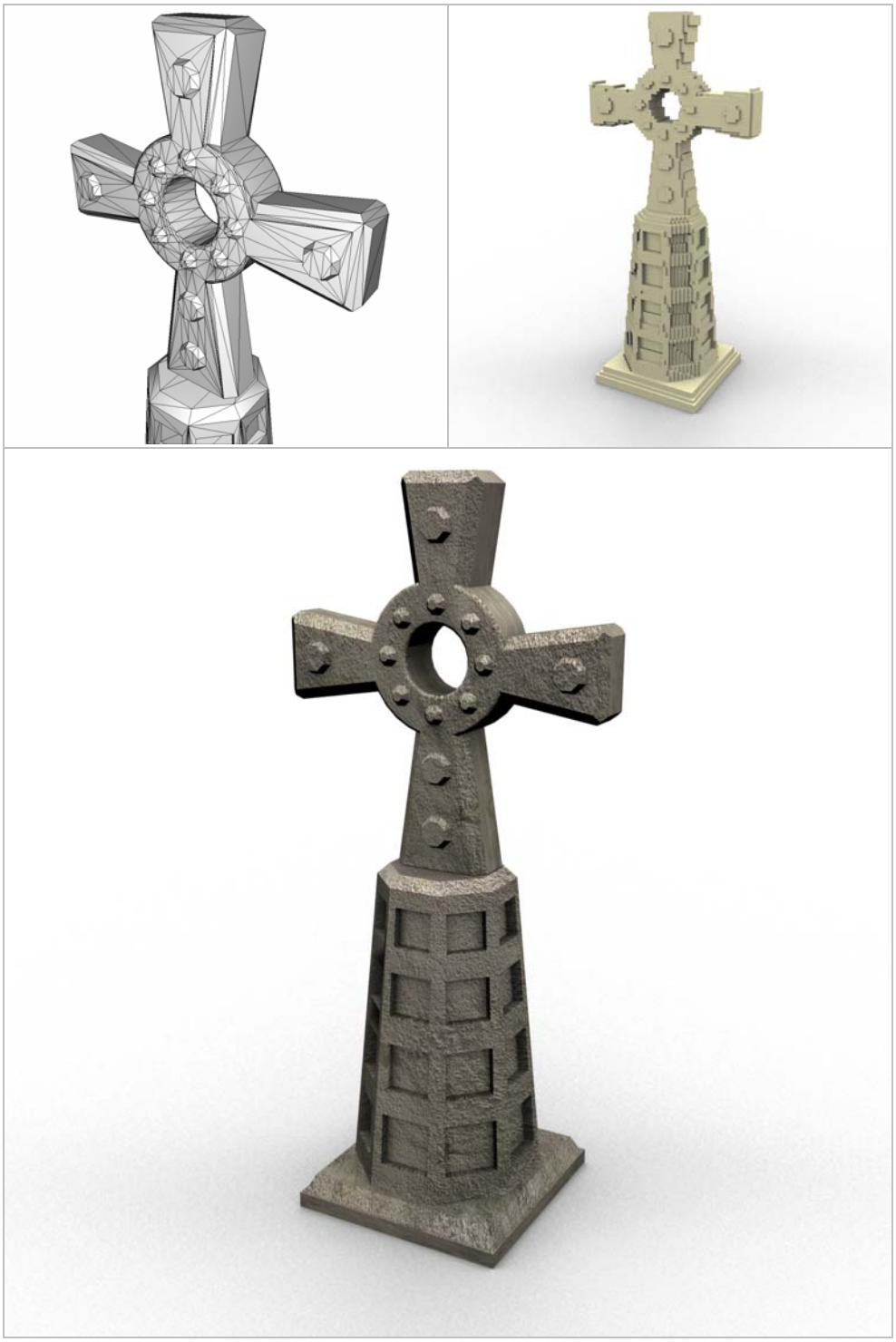


Lámina 2. Modelo Celtic Cross. 1849 vértices. 2366 triángulos.

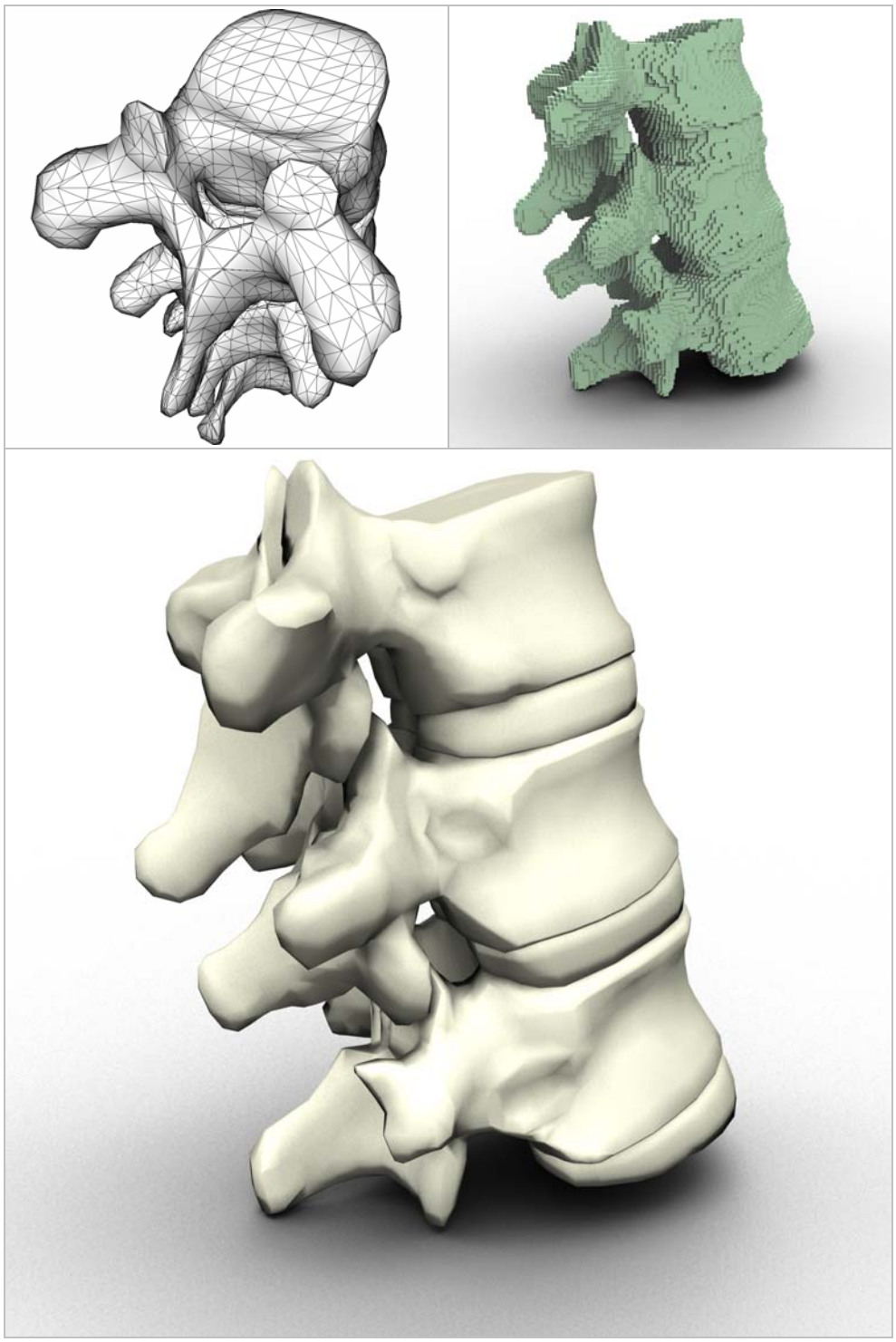


Lámina 3. Modelo Vertebra. 5228 vértices. 10444 triángulos.

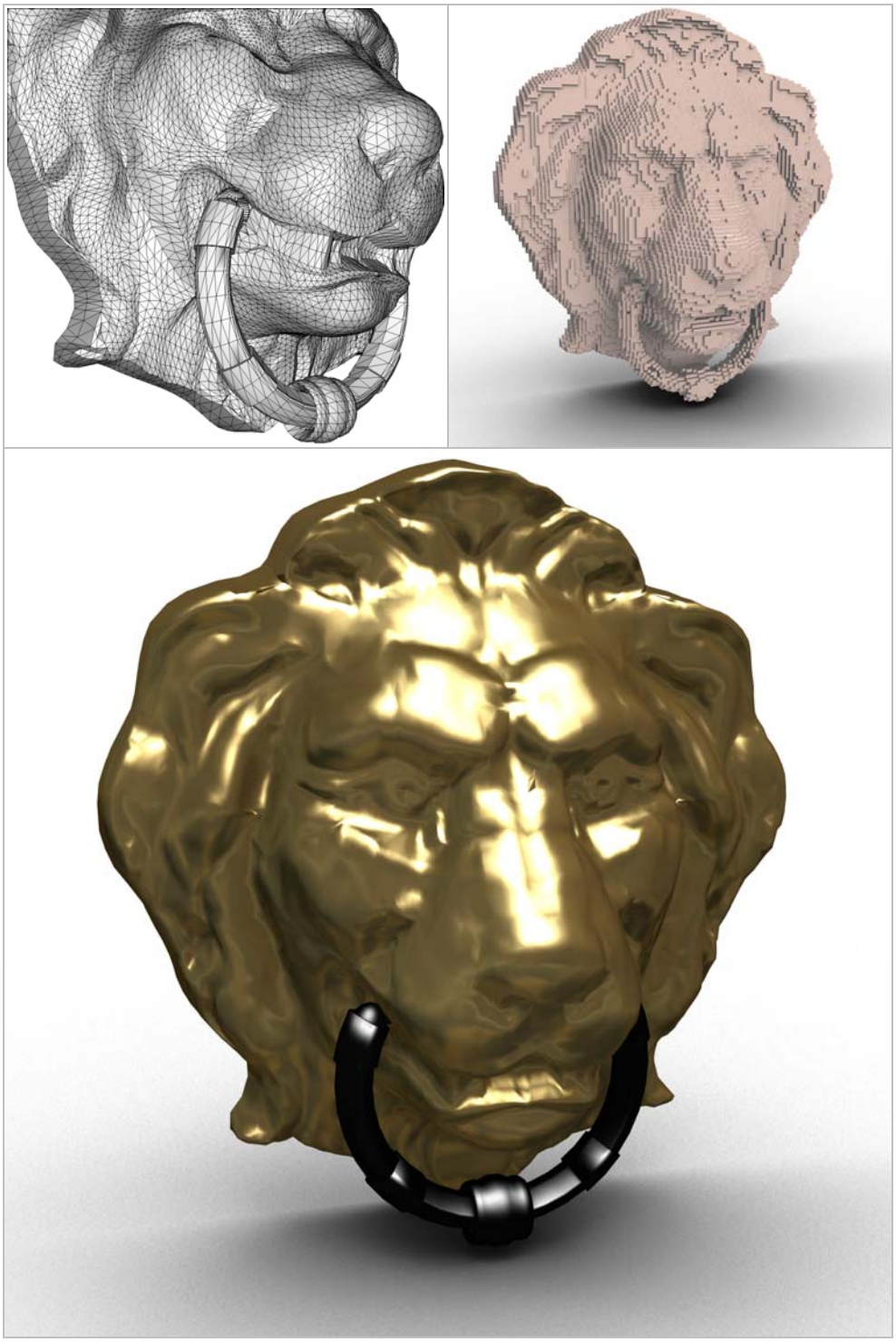


Lámina 4. Modelo Lion Head. 11033 vértices. 22016 triángulos.

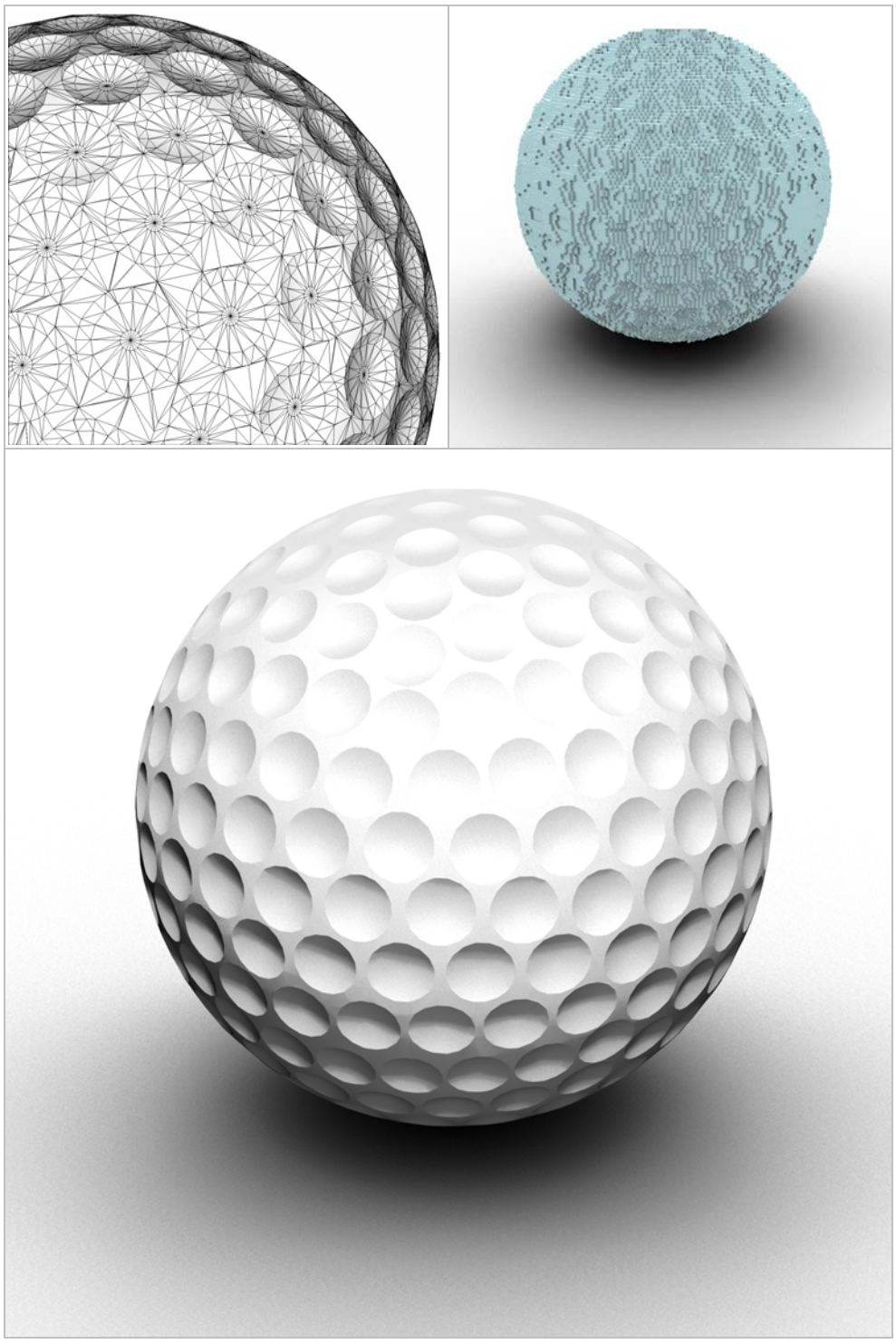


Lámina 5. Modelo Golf Ball. 23370 vértices. 46205 triángulos.

Lámina 6. Modelo Armadillo. 75002 vértices. 150000 triángulos.

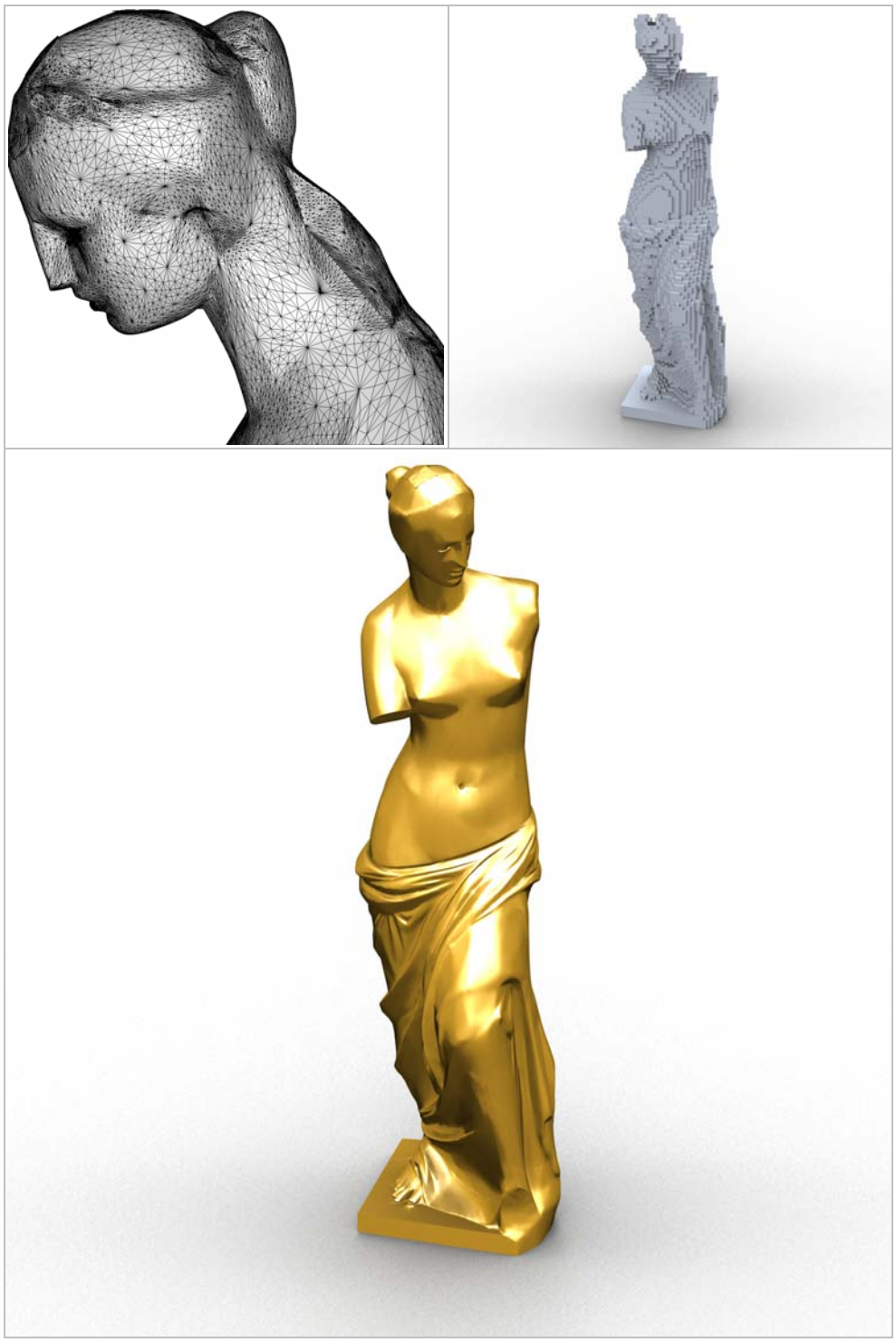


Lámina 7. Modelo Venus Sculpture. 139217 vértices. 277512 triángulos.

Lámina 8. Modelo Happy Buddha. 250007 vértices. 500000 triángulos.

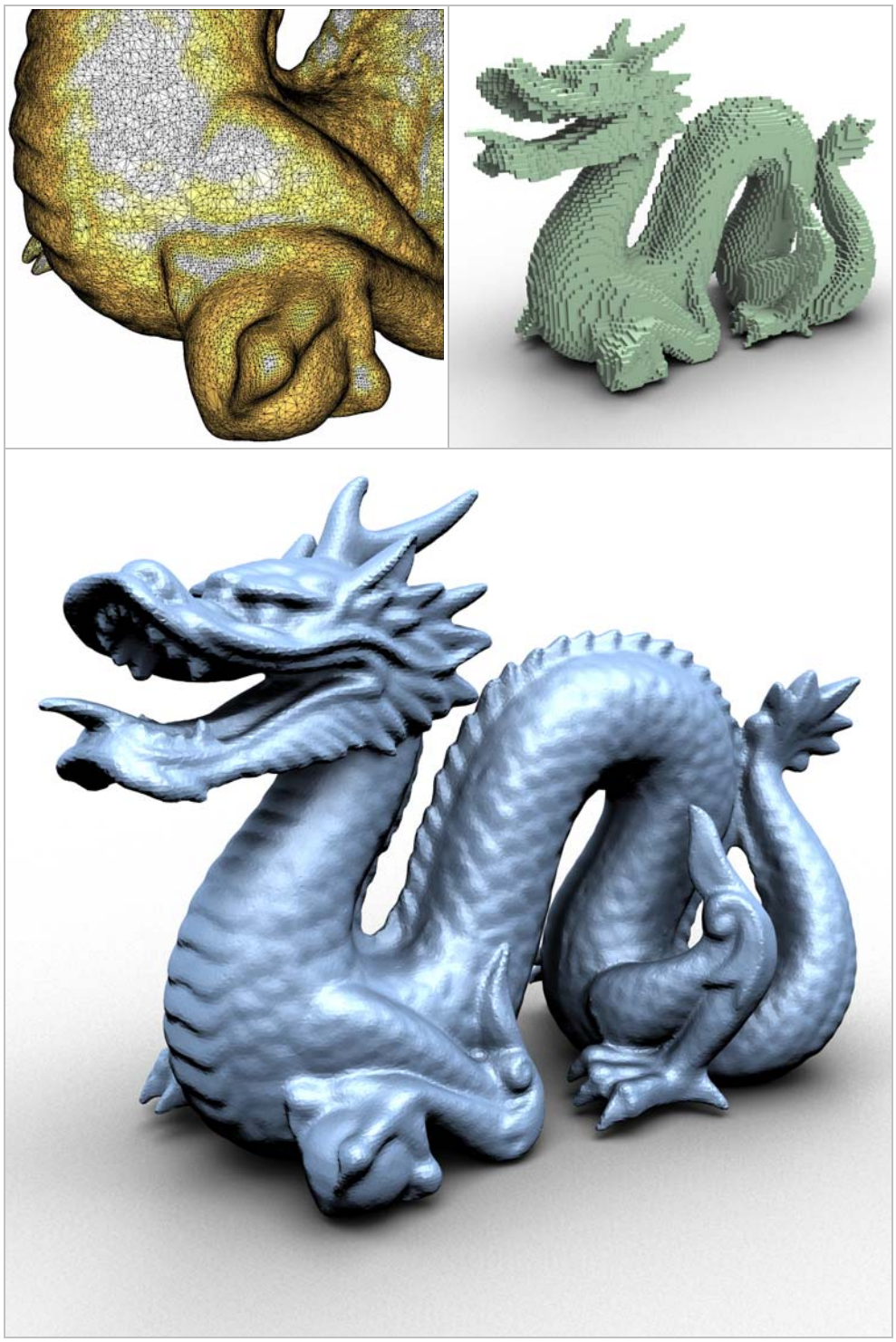


Lámina 9. Modelo Chinese Dragon. 437645 vértices. 871414 triángulos.

	Vértices	*Triángulos*
Torre	1130	2256
Peón	1202	2400
Rey	1610	3214
Reina	1798	3592
Alfil	2909	5814
Caballo	7677	15350

Lámina 10. Modelos de ajedrez.