Esta tesis doctoral contiene un índice que enlaza a cada uno de los capítulos de la misma.
Existen asimismo botones de retorno al índice al principio y final de cada uno de los capítulos.

Ir directamente al índice

Para una correcta visualización del texto es necesaria la versión de Adobe Acrobat Reader 7.0 o posteriores

Aquesta tesi doctoral conté un índex que enllaça a cadascun dels capítols. Existeixen així mateix botons de retorn a l'index al principi i final de cadascun deis capítols .

Anar directament a l'index

Per a una correcta visualització del text és necessària la versió d' Adobe Acrobat Reader 7.0 o posteriors.

meta Blobs

Avances en Algoritmos Eficientes para Poligonalización

autor: Juan A. Puchol García
director: Dr. Ramón Rizo Aldeguer

Tesis Doctoral

Alicante, 19 Noviembre 2001

Nunca te das cuenta de lo que ya has hecho, sólo puedes ver lo que te queda por hacer

Marie ***Curie***

Agradecimientos

Quisiera agradecer, en general, el apoyo y ánimos prestados por todos mis compañeros. En particular quiero agradecer la inestimable ayuda de mi gran amigo Juan Manuel Sáez, sin cuya colaboración la terminación de esta tesis se habría alargado mucho más en el tiempo, también a Ramón Rizo, que ha demostrado una infinita paciencia y me ha guiado en los momentos más difíciles, en fin gracias a todos.

Indice de Contenidos

Indice de Figuras

Volver al índice/Tornar a l'índex

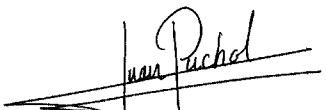
Volver al índice/Tornar a l'índex

Universidad de Alicante

Ciencia de la Computación e Inteligencia Artificial

Prólogo

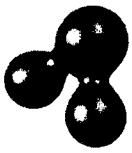
Actualmente la dificultad en el modelado de objetos tridimensionales está aumentando de forma muy rápida, ya que existe la necesidad de representar objetos muy complicados. En esta tesis se aborda el modelado de isosuperficies empleando una formulación de funciones implícitas y analizando las técnicas que mejor se adecuan para realizar una manipulación lo más apropiada a cada uno de los objetos modelados. En particular se aborda el estudio de modelos de superficies curvas empleando el modelo metaBlob. Este modelo tiene gran facilidad de modelado para ciertos tipos de objetos 3D. En esta tesis se desarrollan algoritmos que permiten construir la malla poligonal de forma rápida y eficiente, analizando y comparando éstos con otros algoritmos empleados hasta ahora de forma habitual.



Juan Antonio Puchol García

19 de Noviembre de 2001

1 Motivación y Objetivos



Actualmente el modelado de objetos tridimensionales está sujeto a una amplia variedad de técnicas de codificación, cada una de ellas apropiada a las necesidades de la manipulación final que se desea realizar sobre los objetos 3D modelados. En esta tesis se aborda el modelado de objetos 3D, representados por isosuperficies, empleando una formulación matemática concreta, funciones implícitas, y analizando las técnicas que mejor se adecuan para la obtención de la representación más fidedigna del modelo real. En concreto, en esta tesis se aborda el estudio de modelos de superficies basados en el modelo metaBlob.

1 Motivación y Objetivos

1.1 Motivación

Actualmente el modelado de superficies curvas se ha centrado primordialmente en el empleo de modelos paramétricos de superficies Spline [**CAST93**], [**BEZI93**], en concreto superficies curvas *NURBS* (*Non-Uniform Rational B-Splines*) [**BOOR86**], [**COX72**], [**GORD74**], [**NOWA95**], [**JAIN95**]. Dichos modelos se apoyan en una fuerte formulación matemática, centrada especialmente en elaborar unos puntos de control determinados en un espacio tridimensional, junto a unas tangentes que deseamos que la superficie tenga en tales puntos.

Toda esta información la debe proporcionar el usuario para construir el modelo 3D que desea elaborar. El proceso para decidir qué puntos son los adecuados se convierte, en muchas ocasiones, en un proceso complicado, tedioso y, por no decir, poco intuitivo.

Las aplicaciones dirijidas a la construcción de estos modelos paramétricos de superficies curvas basadas en *NURBS* intentan emplear un interfaz humano-máquina (Human Computer Interface HCI) [**HEWE92**], [**RASK00**] lo más atractivo y sencillo posible, pero es en la propia naturaleza de la superficie a modelar donde surge la verdadera dificultad.

También se une el hecho de que la curva de aprendizaje, que suelen tener este tipo de herramientas que emplean *NURBS*, es bastante elevada, incrementándose de forma notable el tiempo para conseguir manejar estas herramientas con un mínimo de pericia.

Por ello, nuestro objetivo es buscar modelos más sencillos al usuario medio, que pretenda modelar ciertas formas curvas 3D. Dicho modelo es el modelo basado en *metaBlobs*. Este modelo se basa en un modelado más natural al usuario inexperto. En esencia se trata de crear una serie de primitivas 3D, que muchas veces se restringen al uso exclusivo de esferas, en el espacio, asignándole una posición, una fuerza y un tamaño.

Este proceso de modelado se presenta al usuario de una forma más

atractiva, a la vez que más simple. De esta forma, conseguimos una herramienta para este tipo de modelo con una curva de aprendizaje mucho más baja. Además, la fase de elaboración de la forma 3D presenta un tiempo de desarrollo sustancialmente inferior al modelo basado en *NURBS*.

En general un modelo *metaBlob* se puede emplear para construir cualquier superficie curva, y en particular, ciertas formulaciones concretas de este modelo se emplean para la elaboración de formas 3D que representen a seres vivos, personas y animales.

Otro beneficio que obtenemos con el modelo metaBlob es la obtención del interior del objeto 3D que deseamos representar. Proceso que no se puede conseguir con las superficies paramétricas NURBS. En el caso del modelo *metaBlob*, conseguimos tener ambas representaciones (interior y superficie) empleando un único modelo matemático. En realidad, el modelo *metaBlob*, es un modelo de representación de volúmenes curvos. Si tan sólo deseamos representar la superficie, nos quedamos con la capa exterior que dicho modelo representa, es decir su recubrimiento conexo.

Figura 1-1: Modelo metaBlob de una ballena

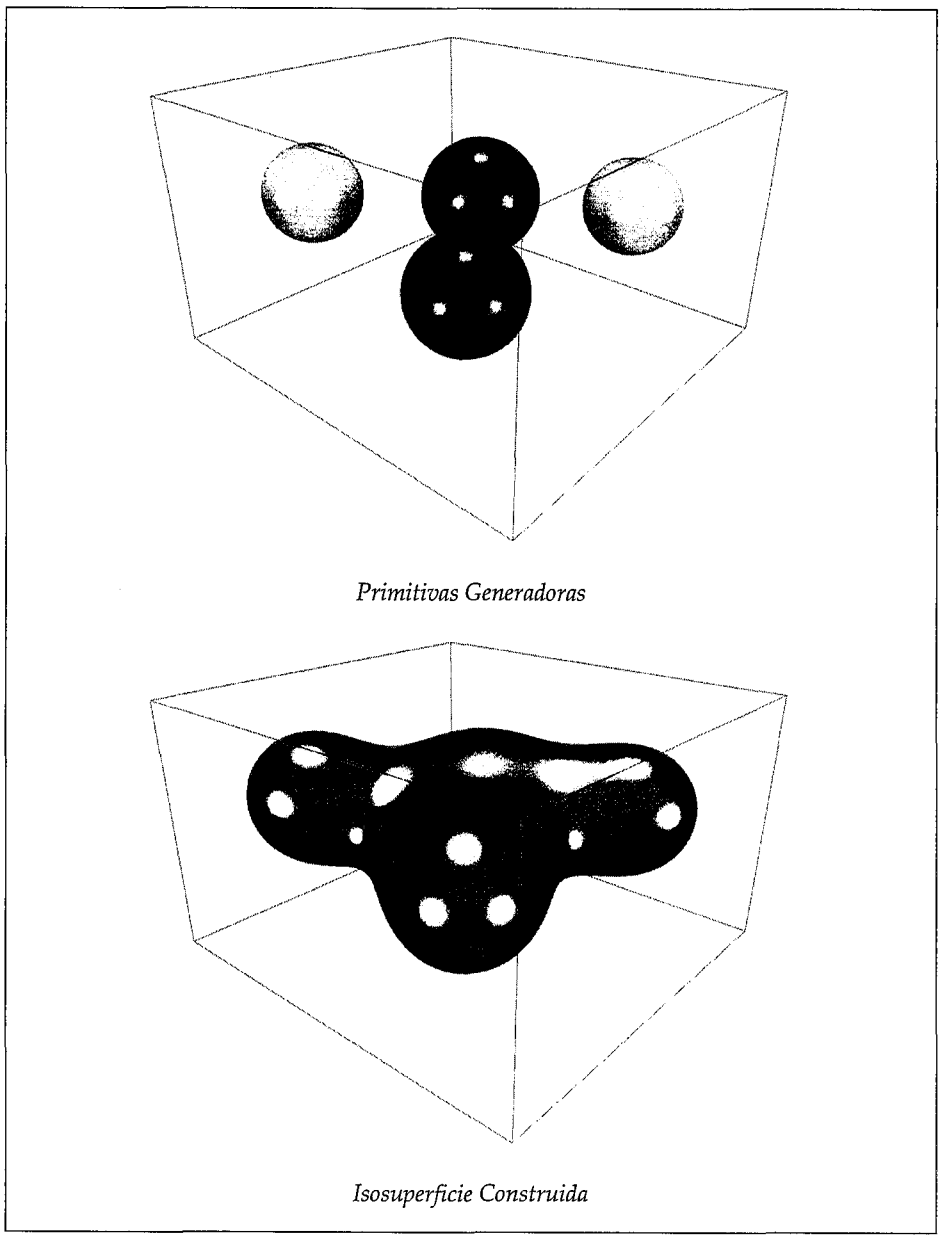


Figura 1-2: Modelado de un metaBlob y sus primitivas generadoras.

1.2 Objetivos

Los algoritmos que se han empleado para construir la superficie generada por un metaBlob han sido, en general, algoritmos de poligonalización, o cálculo del recubrimiento convexo, *clásicos* como la triangulación de Delaunay [**CIGN98**] o diagramas de Voronoi [**BORO95**], y en particular, algoritmos específicos para la poligonalización de funciones implícitas, como *Marching Cubes* [**LORE87**], *Dividing Cubes* [**LORE88**] y demás variantes [**BOUR97**]. Los objetivos que nos vamos a plantear en el presente trabajo son los siguientes:

- Diseño de algoritmos más eficientes que los empleados tradicionalmente hasta ahora. En este aspecto buscaremos las técnicas que nos permitan realizar un cálculo más rápido (dentro del ámbito computacional) y también buscaremos una aproximación más precisa a los puntos reales de la superficie buscada.
- Obtención de las mallas poligonales óptimas. Entendiendo por malla óptima, aquella que cumple una serie de requisitos en cuanto a la forma y tamaño de los triángulos que la componen.
- Adaptabilidad a la curvatura de la superficie. Se trata de emplear mayor densidad de información allí donde la superficie curva lo requiera y menos donde no sea necesario, el parámetro que vamos a medir es el gradiente que presente la superficie entre puntos contiguos, estableciendo así un máximo umbral de diferencia entre los ángulos de los gradientes de dichos puntos. Consiguiéndose una codificación más precisa, y más compacta, puesto que la densidad de información se emplea sólo en las partes necesarias.
- Aceleración genérica susceptible de ser adaptada a cualquiera de estos algoritmos empleando una serie de preprocesos, que reduzcan el volumen de búsqueda lo máximo posible. Para ello estudiaremos posibles métodos para la clasificación en componentes conexas, y de esta forma reducir el volumen de exploración de la superficie de una función ímplicita al conjunto de volúmenes mínimos.

Volver al índice/Tornar a l'índex

1.3 Resumen

Esta tesis se estructura en seis capítulos cuyos contenidos son los que se describen a continuación: un primer capítulo (el presente) de ***Objetivos y Motivación*** donde nos planteamos los motivos que nos sugirieron esta línea de trabajo, así como los objetivos que nos plantemos conseguir en este trabajo; un segundo capítulo de ***Estado del Arte*** que hace un introducción a la teoría base del modelo metaBlob, las curvas Spline (*NURBS*) y los algoritmos clásicos de representación de los metaBlobs; un tercer capítulo de ***Poligonalización de la Superficie*** en el que se proponen nuestros algoritmos; un cuarto capítulo de ***Refinamientos de la Poligonalización*** donde realizamos una serie de mejoras y optimizaciones aplicables a cualquier algoritmo; un quinto capítulo de ***Experimentos y Conclusiones Finales*** donde comparamos nuestros algoritmos con los clásicos de la familia *Marching Cubes*; y un sexto capítulo de ***Líneas Futuras*** donde exponemos las posibles líneas de trabajo.

Para terminar se añade un apéndice: el apéndice A de ***Implementación*** donde describimos los detalles técnicos del prototipo desarrollado para realizar el ensayo de todos los algoritmos que intervienen en esta tesis y, también, una *introducción a OpenGL* donde realizamos una breve descripción a la biblioteca OpenGL y sus estructuras más relevantes.

2 Estado del Arte

Un problema compartido por muchos diseñadores 3D es la necesidad acuciante de modelar formas suaves. Este es un problema difícil de abordar a partir de elementos básicos como cilindros, esferas, elipsoides y cubos para crear formas suaves. Por ello la alternativa es buscar técnicas de modelado más apropiadas para esta tarea. Típicamente, el modelado de formas suaves se ha abordado con la aproximación de curvas y superficies paramétricas, del tipo Bézier y B-Spline, concretamente con las curvas y superficies NURBS. Otros modelos que se están empleando como alternativa son los modelos basados en metaBlobs. Los metaBlobs se construyen a partir de una serie de primitivas, llamadas componentes, que pueden tener cualquier forma (las más comunes son esferas y elipsoides). Cada componente tiene un tamaño y fuerza de atracción (+) o repulsión (-) que las caracteriza. La forma final se calcula basándose en la posición, tamaño y fuerza de cada componente. En este capítulo se trata el estado del arte del modelo metaBlob, superficies paramétricas NURBS, así como el análisis de los modelos mediante técnicas de Elementos Finitos.

2 Estado del Arte

2.1 Introducción

Las superficies implícitas son superficies definidas por funciones implícitas. Una función implícita f es una correspondencia de un espacio n-dimensional R^n al espacio R, descrito por una expresión en la cual todas las variables aparecen a un lado de la ecuación. Hay una manera muy general de describir una correspondencia. Una superficie implícita n dimensional S se define por todos los puntos p en R^n tal que la correspondencia implícita f proyecta p a cero, más formalmente se puede escribir S mediante la siguiente expresión:

$$S_f = \{p | p \in R^n, f(p) = 0\} \qquad \textbf{(2.1)}$$

A menudo las isosuperficies que deseamos visualizar son funciones representadas con la fórmula $f(p)=c$ para una cierta constante c.

Hay distintas clases de superficies implícitas. Las superficies implícitas que son definidas por un polinomio son llamadas superficies algebraicas. Aunque la mayoría de las funciones continuas pueden ser aproximadas por polinomios, las superficies algebraicas representan una clase potente.

Otra clase especial de superficies implícitas están definidas en base a funciones de densidad. Estas funciones de densidad decrecen exponencialmente con respecto a una distancia incremental a un cierto punto central. Estos modelos normalmente se emplean para el uso del estudio de ciertos fenómenos físicos, por ejemplo, modelos de átomos. *Blinn* [**BLIN82**] fue uno de los primeros en intentar la visualización de estas superficies en un computador.

Otro modelo que va ganando importancia es el basado en un conjunto discreto de datos. Tales conjuntos de datos son generados, comúnmente, mediante varios escaners médicos como el NRI y el CT. Estos conjuntos de datos pueden ser vistos también como funciones implícitas donde $f(p)=v$, siendo v el valor escaneado en la posición p. Podemos asumir que si p no es un punto de la rejilla

entonces empleando algún tipo de interpolación entre los puntos vecinos de la rejilla podemos definir v en p.

La visualización de superficies implícitas es una tarea muy complicada y costosa computacionalmente. Usualmente los algoritmos existentes para visualizar superficies implícitas trabajan sólo para casos especiales o conjuntos especiales de funciones. De hecho, hasta la fecha no hay ningún algoritmo que pueda visualizar superficies implícitas arbitrarias de forma precisa.

Los algoritmos para visualización 3D de superficies implícitas se pueden clasificar en tres grupos: transformaciones de representaciones, trazado de rayos y recorrido de superficie. La primera aproximación intenta transformar el problema de una superficie implícita a otra representación que es más fácil de visualizar tales como las mallas poligonales [**ALLG91**], [**BLOO88**], [**HALL90**], [**LORE87**] o representaciones paramétricas [**SEDE85**]. La segunda aproximación para visualizar superficies implícitas emplea algoritmos de trazado de rayos. Han sido desarrollados algoritmos para la intersección eficiente entre rayos y objetos para las superficies algebraicas [**HANR83**], metaballs [**BLIN82**], y una clase de funciones para las cuales las constantes de Lipschitz pueden ser calculadas eficientemente [**HART93**],[**KARL89**]. La tercera aproximación a la visualización de superficies implícitas es discretizar la función. Sederberg y Zundel [**SEDE89**] introdujeron un algoritmo de *scan-line* para una clase de funciones implícitas que pueden ser representadas mediante polinomios de Bernstein.

El modelo de representación que nosotros vamos a emplear se enmarca dentro del primer grupo, es decir transformamos una función del tipo $f(p) = c$ siendo c una constante, a otro modelo, en concreto una malla poligonal.

Por ello pasamos a describir la formulación y conceptos necesarios para la definición de nuestro modelo según una función implícita de estas características.

2 Estado del Arte

2.2 MetaBlobs. Modelado tridimensional de Superficies Implícitas

En 1982, James F. Blinn [**BLIN82**] desarrolló una técnica muy adecuada para aproximar átomos mediante funciones de distribución Gausianas. La idea fue propuesta por Carl Sagan para crear una animación de DNA en su serie de televisión *Cosmos*. La idea es simple: cada átomo tiene algún tipo de función de distribución Gausiana que debe describir su potencial de *van der Waals* como una función de su radio. En cualquier punto donde el valor de potencial del átomo es igual o mayor a un valor *umbral*, la superficie es visible. Si dos o más átomos comparten un espacio común, sus potenciales en dicho punto se suman, y su valor acumulado es el que se compara con el valor umbral de referencia. Los átomos pueden tener potenciales negativos, creando efectos de repulsión entre átomos.

Modelo metaBlob

En definitiva J.F. Blinn creó un método algebraico general de modelado, llamado *"Modelo Blobby o metaBlob"* [**BLIN82**], o modelo metaBlob. En este modelo podemos expresar un objeto 3D en términos de la isosuperficie de un campo escalar que es obtenido por la suma de los campos escalares individuales de un cierto número de primitivas generadoras de campo, las cuales pueden ser no homogéneas. El valor de campo en cualquier punto (x, y, z), creado por cada primitiva P_i en el punto (x_i, y_i, z_i) se expresa de la siguiente manera:

$$V_i(x, y, z) = b_i e^{-a_i f_i(x, y, z)} \qquad \textbf{(2.2)}$$

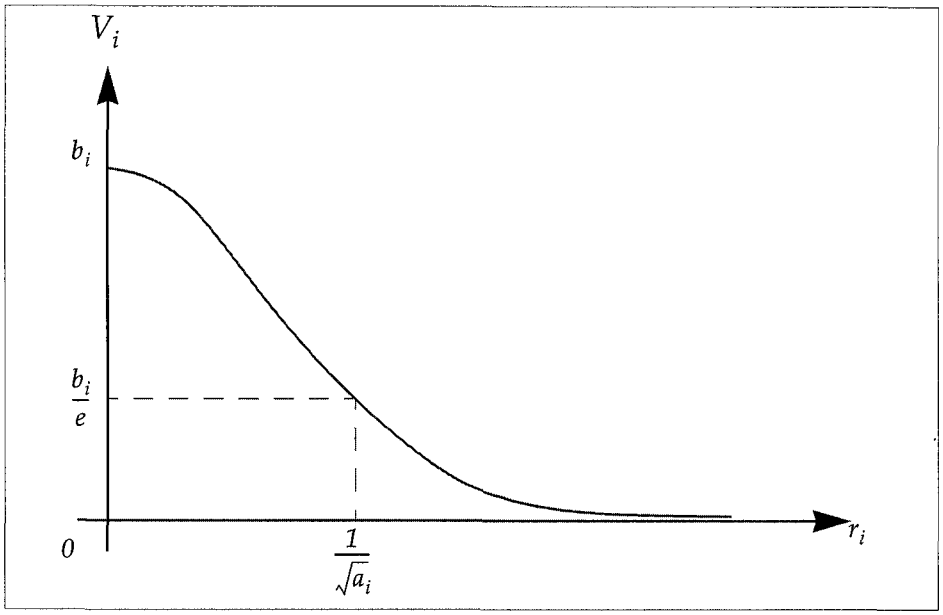
La función $f_i(x, y, z)$ define la forma del campo escalar. Por ejemplo, en el caso de un campo esférico simétrico $f_i(x, y, z)$, es la distancia euclidea al cuadrado entre (x, y, z) y (x_i, y_i, z_i):

$$f_i(x, y, z) = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2 \qquad \textbf{(2.3)}$$

En general a $f_i(x, y, z)$ le vamos a exigir que cumpla ser una función continua y derivable.

La siguiente figura representa el valor de caída del campo $V_i(x, y, z)$ a lo largo de la distancia r_i desde el punto (x_i, y_i, z_i). En la ecuación (2.2) se ha empleado como función $f_i(x, y, z)$ la ecuación (2.3).

Figura 2-1: Valor de caída del campo escalar según la distancia



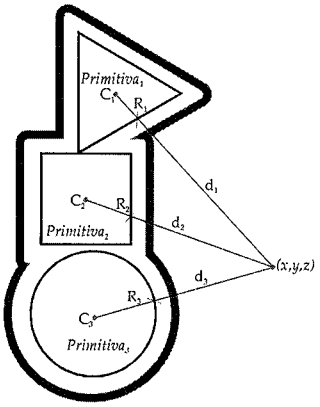
Y en el caso de un campo de forma supercuádrica [**BARR81**], se escribe como:

$$f_i(x, y, z) = \left[(x - x_i)^{\frac{2}{\upsilon_i}} + (y - y_i)^{\frac{2}{\upsilon_i}} \right]^{\frac{\upsilon_i}{\mu_i}} + (z - z_i)^{\frac{2}{\mu_i}} \qquad \textbf{(2.4)}$$

donde μ_i y υ_i son los parámetros relacionados con la forma de las supercuádricas. Si se emplea la ecuación (2.3), el valor del campo V_i decae exponencialmente con la distancia desde (x_i, y_i, z_i) como se muestra en la (figura 2-1). El parámetro a_i (> 0) afecta al grado de

caída y b_i afecta a la fuerza del campo. Si se usan varias primitivas a la vez, entonces el campo escalar de cada primitiva se suma y la isosuperficie resultante puede mostrar una forma muy elaborada. A partir de la ecuación (2.3) el campo que es producido por N primitivas para cualquier punto se expresa de la siguiente forma:

$$V(x,y,z) = \sum_{i=1}^{N} b_i e^{-a_i f_i(x,y,z)} \qquad \textbf{(2.5)}$$



Consecuentemente, la isosuperficie obtenida al restarle el valor de $T(>0)$, valor umbral global que influye en toda la forma, se expresa como una función implícita, donde los valores del campo escalar iguales a cero representan el contorno o superficie del volumen, los valores negativos, pertenecen al exterior del volumen, y los valores positivos pertenecen al interior del volumen

$$V(x,y,z) = 0 \quad \text{si } (x,y,z) \in Superficies \qquad \textbf{(2.6)}$$

La ecuación implícita queda así de la siguiente manera:

$$V(x,y,z) = \sum_{i=1}^{N} b_i \cdot e^{-a_i f_i(x,y,z)} - T \qquad \textbf{(2.7)}$$

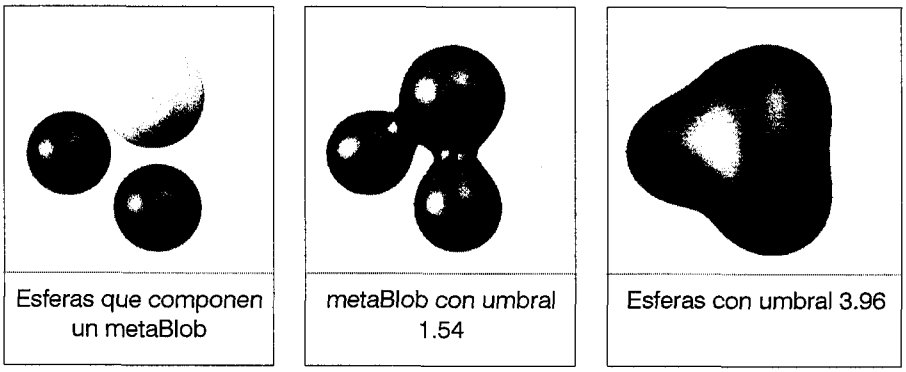
Si se define un valor de atributo, tal como la componente de color, para cada primitiva, podemos calcular el valor para un punto como sigue:

$$C(x,y,z) = \frac{1}{V(x,y,z)} \sum_{i=1}^{N} C_i V_i(x,y,z) \qquad \textbf{(2.8)}$$

Si hay sólo una primitiva, ésta forma una isosuperficie con función $f_i(x,y,z)$. La figura (2-2) muestra la isosuperficie de un campo esférico formado por tres esferas, las más oscuras son de igual

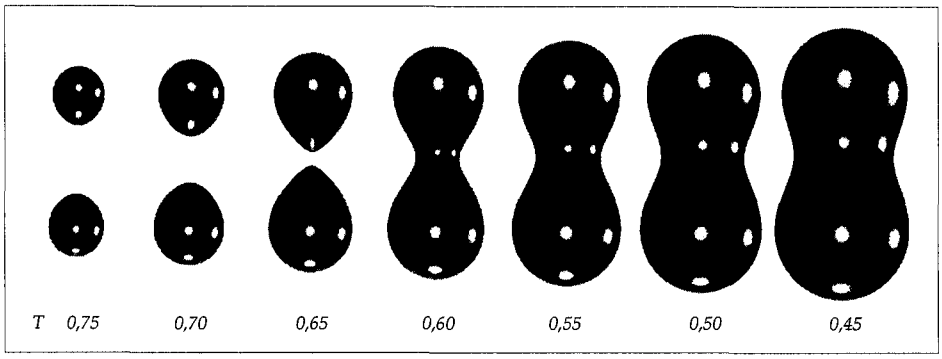
tamaño y la más clara de mayor radio que las otras dos. Modelo de metaBlobs formado a partir de tres esferas, con los valores de umbral de 1.54 y 3.96

Figura 2-2: Modelo metaBlob formado a partir de tres esferas



Además del modelo metaBlob, hay métodos similares tales como los *Metaballs* **[NISH85]** y los *Objetos Suaves* (Soft Objects) **[WYVI86]**. Estos métodos cambian la función de campo de acuerdo a la distancia que hay desde la primitiva, de forma que el efecto del campo de la primitiva es finito. Efecto del valor umbral T aplicado a dos esferas

Figura 2-3: Influencia del valor umbral T aplicado a dos esferas



2 Estado del Arte

Metaball

El modelo *Metaball* [**NISH85**] se define como una combinación de funciones cuadráticas de densidad, se reduce el uso a primitivas de forma esférica:

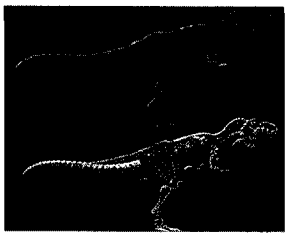
$$f(r) = \begin{cases} b\left(1 - 3\frac{r^2}{d^2}\right) & si\ 0 < r \le \frac{d}{3} \\ \frac{3}{2}b\left(1 - \frac{r}{d}\right)^2 & si\ \frac{d}{3} < r \le d \\ 0 & si\ r > d \end{cases} \quad \textbf{(2.9)}$$

Soft Objects

El modelo de *Objetos Blandos* "Soft Objects" [**WYVI86**] se define con la siguiente función de densidad:

$$f(r) = \begin{cases} 1 - \frac{22r^2}{9d^2} + \frac{17r^4}{9d^4} + \frac{4r^6}{9d^6} & si\ 0 < r \le d \\ 0 & si\ r > d \end{cases} \quad \textbf{(2.10)}$$

Figura 2-4: Dinosaurios modelados con metaballs



2.3 Modelos de Fronteras y Superficies

Las escenas gráficas pueden contener multitud de objetos de distinto tipo: árboles, flores, nubes, rocas, agua, paneles de madera, papel, mármol, cristal, plástico, por sólo mencionar algunos de ellos. Por lo tanto no nos debe sorprender que no sea únicamente un método el que se emplee para describirlos, sino que debamos acudir a varios para describir las distintas características de cada uno de los objetos que compondrán parte de nuestra escena. Y para la producción de escenas realistas necesitamos emplear las representaciones que mejor se ajusten al objeto y así obtener mayor precisión en su descripción.

Las superficies por polígonos y cuádricas proporcionan una descripción precisa para objetos euclídeos, tales como poliedros y elipsoides; las superficies splines y las técnicas de construcción son adecuadas para el diseño de alas de avión, ruedas, y otras estructuras formadas por superficies curvas; los métodos procedurales, tales como los fractales y los sistemas de partículas, nos permiten representar con precisión nubes, hierba y otros objetos naturales; los métodos de modelado basados en física usan sistemas de fuerzas que interactúan que pueden ser empleados para describir el manejo de objetos no rígidos; las codificaciones con árboles octrees se utilizan para codificar el interior de los objetos, tales como aquellos obtenidos con imágenes médicas; y otras técnicas de visualización aplicadas a conjuntos discretos de datos para crear una representación visual de la información.

Los esquemas de representación de objetos sólidos se dividen a menudo en dos categorías, aunque no todas las representaciones caen estrictamente en una de estas dos categorías. Las *representaciones por frontera* describen los objetos como un conjunto de superficies que separan el objeto del entorno, y las *representaciones por enumeración espacial* (por división del espacio) se emplean para describir las propiedades del interior de los objetos. Con este método se parte la región espacial que contiene al objeto en un conjunto de pequeños sólidos contiguos, que no se solapan,

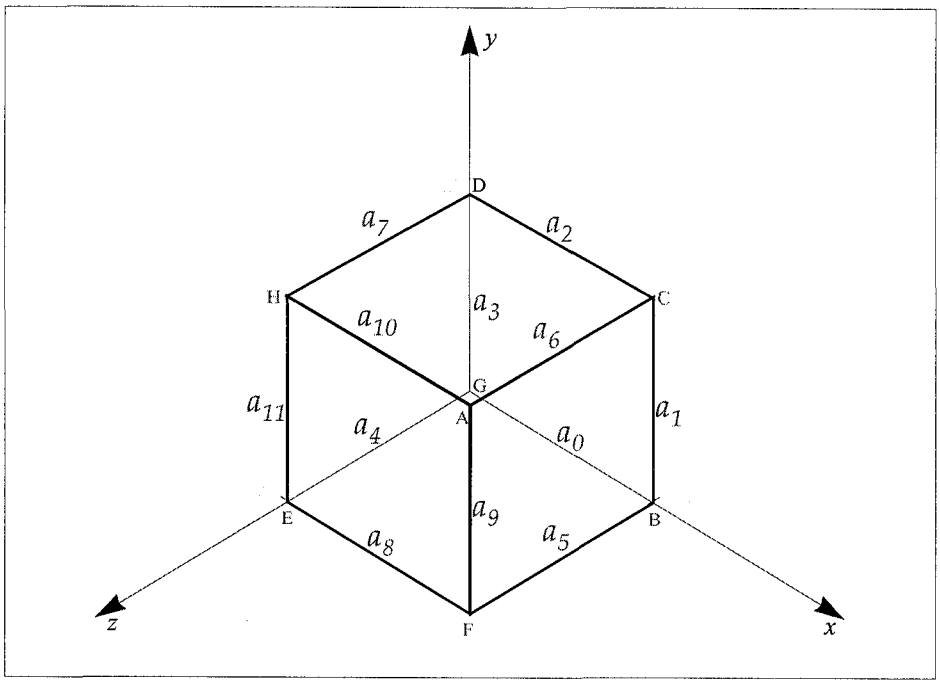
usualmente cubos.

Superficies Poligonales

El método más común para representar las fronteras de un objeto gráfico tridimensional es un conjunto de superficies poligonales que encierran el interior del objeto. Muchas aplicaciones gráficas almacenan la descripción de todos los objetos como superficies poligonales. De esta manera se puede simplificar y acelerar el proceso de visualización (*Rendering)* ya que todas las superficies son descritas con ecuaciones lineales. Por este motivo, esta representación es conocida a menudo como *"objetos gráficos estándares"*

Figura 2-5: Codificación tridimensional de un cubo empleando el modelo de fronteras

.



Codificación Poligonal

La representación del objeto va a estar formada por tres listas diferentes de datos: una **lista de vértices**, donde cada uno de ellos almacenará las coordenadas (x, y, z) de un punto del espacio tridimensional; una **lista de aristas**, formada por los segmentos de líneas o aristas que van a formar los lados del objeto, la información que contendrá será los vértices origen y destino de las aristas; y por último; una **lista de polígonos**, donde se almacenan las aristas que forman dicho polígono, o en su defecto los vértices que lo forman. Veamos la información que contendrían dichas listas si quisiéramos representar el cubo de la figura 2-5.

Figura 2-6: Tablas de codificación asociadas a la figura 2.5

Tabla de Vértices	
A	(0.0, 0.0, 0.0)
B	(1.0, 0.0, 0.0)
C	(1.0, 1.0, 0.0)
D	(0.0, 1.0, 0.0)
E	(0.0, 0.0, 1.0)
F	(1.0, 0.0, 1.0)
G	(1.0, 1.0, 1.0)
H	(0.0, 1.0, 1.0)

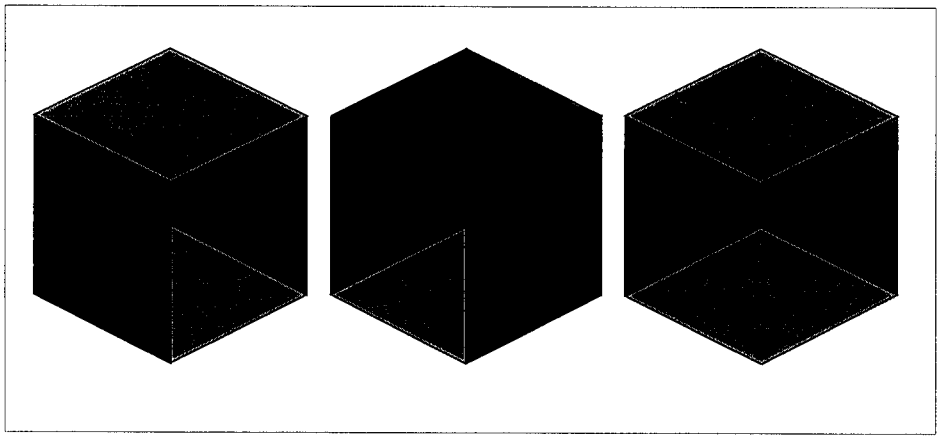
Tabla de Aristas	
a_0	A, B
a_1	B, C
a_2	C, D
a_3	D, A
a_4	A, E
a_5	B, F
a_6	C, G
a_7	D, H
a_8	E, F
a_9	F, G
a_{10}	G, H
a_{11}	H, E

Tabla de Polígonos	
P_0	a_0, a_3, a_2, a_1
P_1	a_1, a_6, a_9, a_5
P_2	a_2, a_7, a_{10}, a_6
P_3	a_3, a_4, a_{11}, a_7
P_4	a_5, a_8, a_4, a_0
P_5	a_8, a_9, a_{10}, a_{11}

En principio se podría pensar que la información de los polígonos

no es necesaria, pero nos daría objetos sujetos a múltiples interpretaciones, como se muestra a partir del cubo anterior en la siguiente figura.

Figura 2-7: Diversas interpretaciones de un cubo

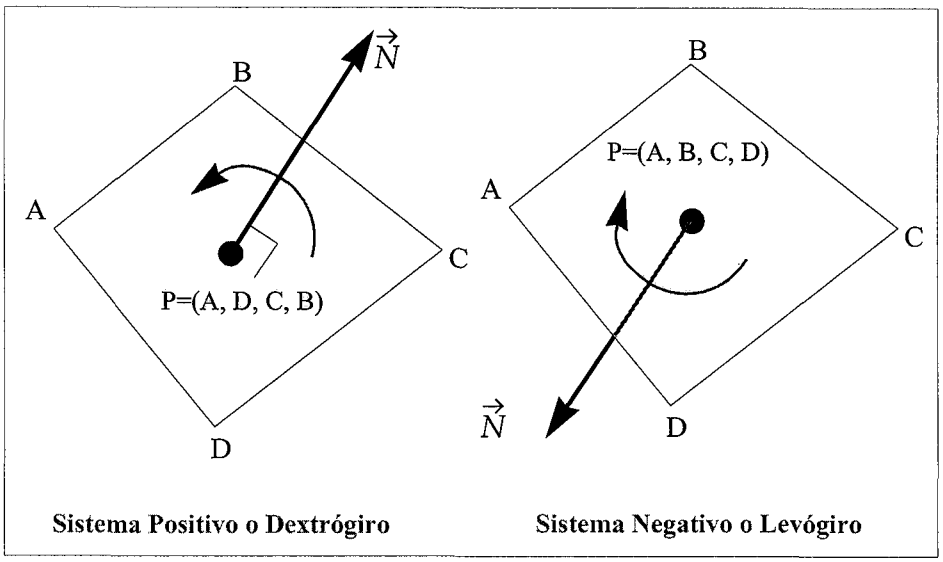


Requicha proporcionó una lista de propiedades deseables que un esquema de representación de sólidos debía cumplir **[REQU80]**. Se ha de buscar por tanto un modelo de representación que no sea *ambiguo*, es decir, que no este sujeto a más de una posible interpretación, cuando un modelo no es ambiguo se dice también que es *completo*. Una representación es *única* si el objeto sólido a representar admite sólo una codificación posible. Una representación es *precisa* si permite que un objeto sea representado sin emplear una aproximación. Idealmente un modelo de representación no debería permitir la representación de objetos imposibles. También es deseable que la representación sea *compacta* para ahorrar espacio en su codificación. Por último una representación debe permitir que se puedan emplear algoritmos eficientes para calcular las propiedades físicas deseadas, realizar las operaciones de modelado apropiadas, y finalmente representar el objeto resultante.

Ecuación del Plano

Otro detalle que hay que tener en cuenta, que es muy importante, es el *orden* en el que damos el conjunto de aristas o vértices a la hora de codificar un polígono. Tenemos dos sentidos posibles a elegir, el sentido positivo o *dextrógiro* o el sentido negativo o *levógiro*, por tanto en esta tesis emplearemos, en toda nuestra notación matemática referente a polígonos, el sistema dextrógiro.

Figura 2-8: Sistemas dextrógiro y levógiro



El orden que elijamos a la hora de dar los vértices de los polígonos va a influir en el *sentido* que tomará su normal (la *dirección* será la misma). O lo que es lo mismo indicará desde qué lado es visible el polígono. Lo cual indica que si un polígono se desea ver por ambos lados deberemos, bien codificarlo como dos polígonos distintos con los mismos vértices pero en sentido inverso uno del otro, bien introduciendo información en la codificación que señale que el polígono es visible desde sus dos lados.

En vistas a aplicar procedimientos de rendering con ocultación de líneas, identificación de superficies visibles, etc., necesitamos calcular las ecuaciones del plano a partir de las tablas con las que

codificamos el modelo tridimensional de un objeto.

La ecuación del plano se expresa de la siguiente forma:

$$Ax + By + Cz + D = 0 \qquad \textbf{(2.11)}$$

Donde las coordenadas (x, y, z) son cualquier punto del plano, y los coeficientes A, B, C y D son constantes que describen las propiedades espaciales del plano. Podemos obtener los coeficientes A, B, C y D resolviendo un conjunto de tres ecuaciones del plano empleando los valores de tres puntos no colineales (es decir, que no están en la misma recta). Para ello seleccionamos tres puntos consecutivos del polígono (x_1, y_1, z_1), (x_2, y_2, z_2) y (x_3, y_3, z_3) y resolvemos el siguiente conjunto simultáneo de ecuaciones de plano lineales para los ratios A/D, B/D y C/D.

$$\frac{A}{D}x_k + \frac{B}{D}y_k + \frac{C}{D}z_k = -1 \qquad k = 1, 2, 3 \qquad \textbf{(2.12)}$$

La solución de este conjunto de ecuaciones se puede obtener en forma de determinantes, usando la *regla de Cramer*, como:

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \quad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$
$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \quad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \qquad \textbf{(2.13)}$$

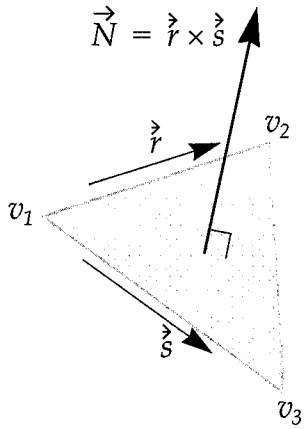
Expandiendo los determinantes podemos escribir cada coeficiente de la siguiente forma:

$$
\begin{aligned}
A &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\
B &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\
C &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\
D &= -x_1(y_2 z_3 - y_3 z_2) - y_2(y_3 z_1 - y_1 z_3) - x_1(y_1 z_2 - y_2 z_1)
\end{aligned}
\qquad \textbf{(2.14)}
$$

Una alternativa para simplificar el cálculo de D, es emplear uno de los vértices del polígono, por ejemplo, el primero (x_1, y_1, z_1), y como ya conocemos A, B y C por la ecuación (2.14), despejar D de la siguiente forma:

$$D = -Ax_1 - By_1 - Cz_1 \qquad \textbf{(2.15)}$$

Otra forma de calcular la normal del plano es emplear el producto vectorial, el producto vectorial se realiza de la siguiente forma:



$$\vec{N} = \vec{r} \times \vec{s} = \begin{vmatrix} r_x & r_y & r_z \\ s_x & s_y & s_z \\ \vec{i} & \vec{j} & \vec{k} \end{vmatrix} = A\vec{i} + B\vec{j} + C\vec{k} \qquad \textbf{(2.16)}$$

Siendo los vectores $\vec{r}$ y $\vec{s}$ los formados por las direcciones de v_1 a v_2 y de v_1 a v_3, respectivamente, y siendo $\vec{i}, \vec{j}$ y $\vec{k}$ los vectores unitarios del origen de coordenadas del espacio euclideo que vamos a emplear.

Área del triángulo $= \dfrac{|\vec{N}|}{2}$

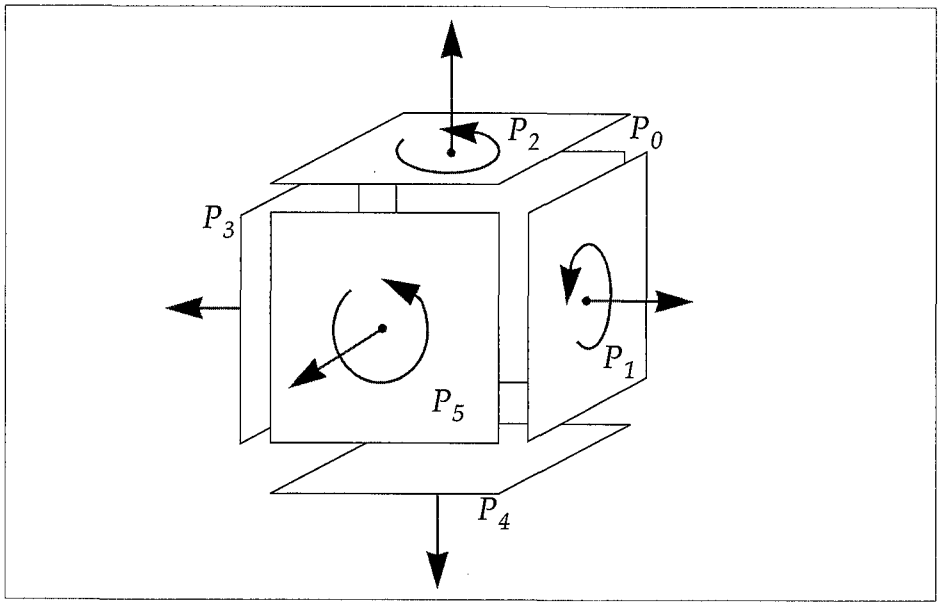
El área del triángulo se obtiene también al calcular la normal al ser ésta la mitad del módulo de dicha normal. El módulo se calcula de la siguiente forma:

$$|\vec{N}| = \sqrt{A^2 + B^2 + C^2} \qquad \textbf{(2.17)}$$

Empleando el sistema dextrógiro, la siguiente figura representa el

sentido correcto para codificar los vértices de cada uno de los polígonos que forman nuestro cubo.

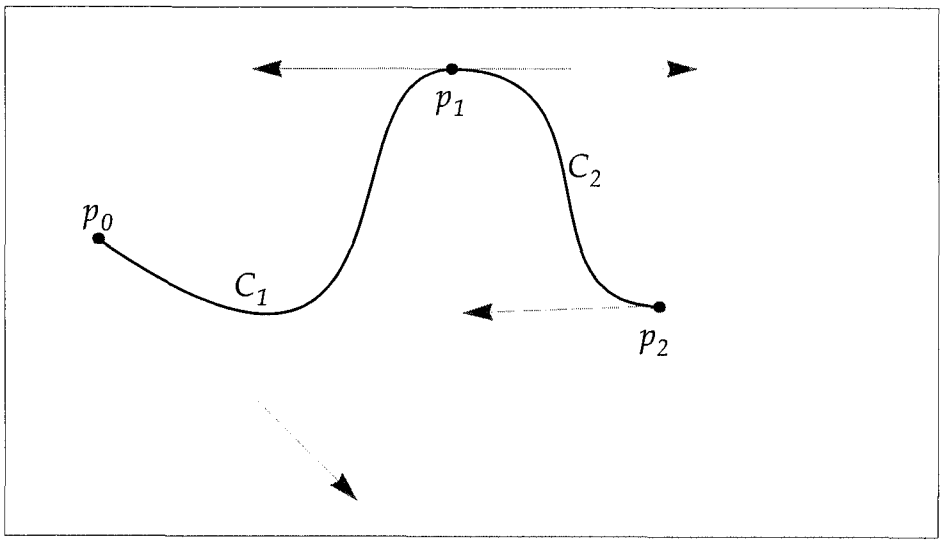
Figura 2-9: Sentido de las Normales para el cubo de la figura 2-5



2.4 Curvas y Superficies Paramétricas. Splines

Las curvas y superficies paramétricas bicúbicas son las que se emplean como herramienta cotidiana en *Informática Gráfica* de forma común para la representación de superficies poligonales de aspecto suave.

Figura 2-10: Representación de una curva paramétrica



Hay tres métodos equivalentes para especificar una representación Spline.

1) Un conjunto de condiciones límite que se imponen a la Spline.
2) Una matriz que caracterice a la Spline.
3) Podemos establecer una serie de funciones *blending* o funciones base que determinan cómo se combinan restricciones geométricas específicas de la curva para calcular posiciones a lo largo de la trayectoria de la curva.

Funciones blending: funciones base que se combinan para formar curvas más genéricas

Supongamos que tenemos la siguiente representación polinomial paramétrica cúbica para la coordenada x:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x, \qquad 0 \le u \le 1 \qquad \textbf{(2.18)}$$

Las condiciones límite podrían ser, por ejemplo, los puntos extremos, $x(0)$ y $x(1)$ y las primeras derivadas en dichos puntos de la función paramétrica $x'(0)$ y $x'(1)$. Estas cuatro condiciones son suficientes para determinar los coeficientes a_x, b_x, c_x y d_x.

A partir de las condiciones límite, podemos obtener una matriz que caracterice la curva spline reescribiendo la ecuación 2.18 en forma de producto matricial:

$$x(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} = U \cdot C \qquad \textbf{(2.19)}$$

donde U es la matriz fila de potencias del parámetro u, y C es la matriz columna de coeficientes. Usando la ecuación 2.19 podemos escribir las condiciones límite en forma matricial y resolver la matriz de coeficientes C como:

$$C = M_{spline} \cdot M_{geom} \qquad \textbf{(2.20)}$$

donde M_{geom} es una matriz columna de cuatro elementos que contiene las restricciones geométricas (condiciones límite) de la curva spline, y M_{spline} es la matriz 4x4 que transforma los valores de las restricciones geométricas a los coeficientes polinomiales y proporciona la caracterización de la curva spline. Así, sustituyendo la matriz C por su valor obtenemos:

$$x(u) = U \cdot M_{spline} \cdot M_{geom} \qquad \textbf{(2.21)}$$

La matriz M_{spline} que caracteriza la curva spline, también recibe el nombre de *matriz base.*

Finalmente, también podemos expresar una curva spline en términos de restricciones geométricas:

$$x(u) = \sum_{k=0}^{3} g_k \cdot BF_k(u) \qquad \textbf{(2.22)}$$

donde g_k son los parámetros de restricciones, tales como las coordenadas de los puntos de control y las pendientes de la curva en dichos puntos de control, y donde $BF_k(u)$ son las funciones *blending* polinomiales.

Uno de los tipos de curvas pertenecientes a la familia de las curvas Splines, son las NURBS, B-Spline racionales no uniformes. Estas curvas se han convertido dentro del ámbito industrial en el estándar más empleado para especificar superficies paramétricas.

2.5 Análisis con Técnicas de Elementos Finitos

En la última década las técnicas de análisis basadas en el *Método de Elementos Finitos (MEF)* han tomado un gran auge en su aplicación al análisis estructural. Los rápidos avances en el hardware de los computadores junto a los sustanciales incrementos de memoria así como la velocidad de procesamiento, en especial en cálculo, todo ello en conjunción al abaratamiento de los equipos, ha propiciado que se puedan abordar análisis basados en estos métodos con relativa facilidad y al alcance del usuario científico medio. Así las simulaciones de análisis estructurales han cobrado una gran importancia en los últimos años.

Qué es el Método de Elementos Finitos

El Método de Elementos Finitos es una técnica de análisis numérico para obtener soluciones aproximadas a una amplia variedad de problemas en ingeniería. Aunque originalmente diseñado como método para el estudio de la tensión (*stresses*) en estructuras aeronáuticas complejas, desde entonces ha sido extendido y aplicado a un amplio abanico de problemas de mecánica continua. A causa de su diversidad y flexibilidad como herramienta de análisis está captando una gran atención por parte de las escuelas de ingeniería así como por la industria en general.

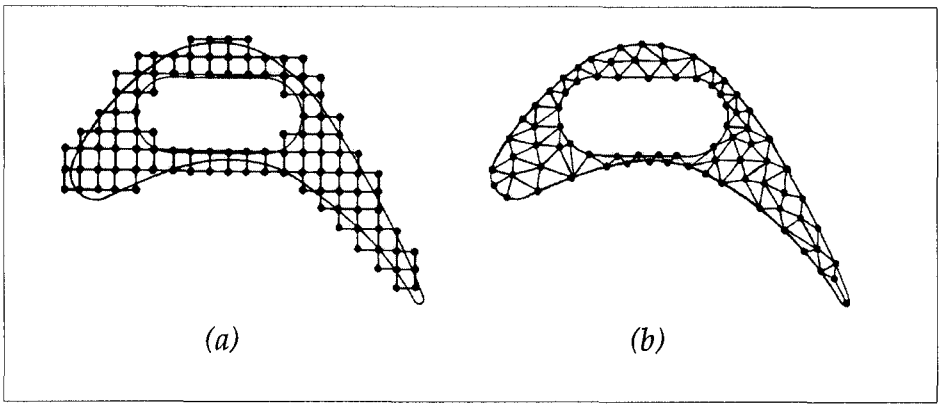
Cómo Funciona el Método

Problema Continuo: cuerpo de materia (sólido, líquido o gas), tomado como una región del espacio donde ocurre un fenómeno particular

En un *problema continuo* de cualquier dimensión, la variable de campo (cualquiera que sea, presión, temperatura, tensión, etc) posee un número infinito de valores debido a que es una función de cada punto genérico en el cuerpo o región solución. Consecuentemente, el problema se convierte en uno con un número infinito de incógnitas. Los procedimientos de discretización de elementos finitos reducen el problema a uno con un número finito de incógnitas dividiendo la región solución en elementos y expresando las incógnitas de la variable de campo en términos de funciones de aproximación asumidas dentro de cada elemento. Las funciones de aproximación (a veces llamadas *funciones de*

interpolación) se definen en términos de los valores de las variables de campo en puntos específicos llamados *nodos* o *puntos nodales*. Los nodos usualmente aparecen en los contornos de los elementos donde se conectan elementos adyacentes. Además de los nodos de contorno, un elemento debe tener algunos pocos nodos interiores. Los valores nodales de la variable de campo y las funciones de interpolación para los elementos definen completamente la apariencia de la variable campo dentro de los elementos.

Figura 2-11: (a) Diferencias finitas y, (b) discretización de una pieza de turbina



Para resumir de forma sencilla cómo funciona este método describimos este proceso en 7 pasos:

Método de Elementos Finitos: (MEF) en 7 pasos

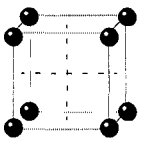
1. *Discretizar la continuidad*. El primer paso es dividir la continuidad o la región solución en elementos discretos.
2. *Seleccionar las funciones de interpolación.*
3. *Encontrar las propiedades de los elementos.*
4. *Ensamblar las propiedades de los elementos para obtener los sistemas de ecuaciones.*
5. *Imponer las condiciones de contorno.*
6. *Resolver el sistema de ecuaciones.*
7. *Realizar cálculos adicionales si se desea.*

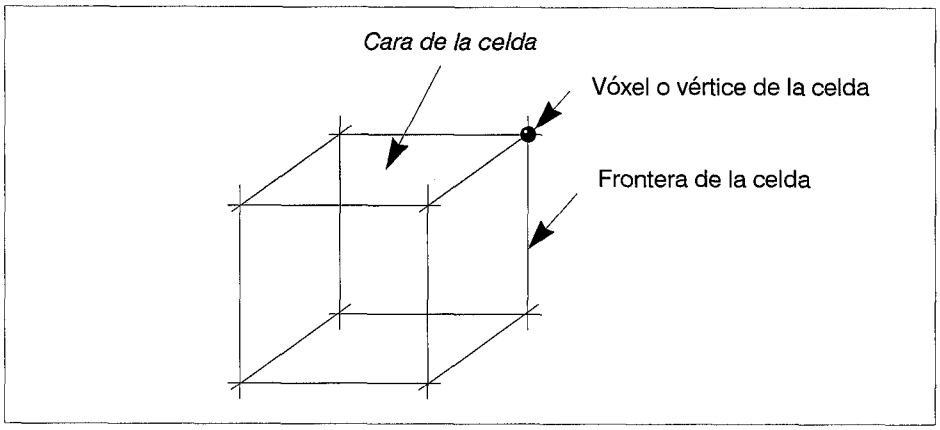
2.6 Algoritmo Marching Cubes

El algoritmo *Marching Cubes* desarrollado por Lorensen y Cline [**LORE87**] fue ideado para crear modelos poligonales basados en triángulos para superficies de densidad constante, isosuperficies, a partir de datos médicos 3D.

Desde entonces ha sido uno de los métodos más empleados para crear el modelo poligonal asociado a un modelo metaBlob (como isosuperficie que es). El algoritmo *Marching Cubes* se realiza principalmente en dos etapas. Se deben crear en primer lugar los triángulos correspondientes a una superficie designada por el usuario. Entonces, se calculan las normales a la superficie en el vértice de cada triángulo para asegurar la calidad de la imagen.

Figura 2-12:
Celda Malla Unidad





Malla Unidad: región cúbica, cuyos ocho vértices son todos vóxels, es decir la posición de sus centros

Este algoritmo produce triángulos conectados resultado de intersectar la isosuperficie con cada celda de una *Malla Unidad.*

El algoritmo se resume en 8 pasos:

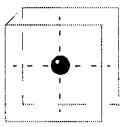
1) El usuario especifica el valor umbral T

2) Se leen cuatro trozos del volumen

3) Se comprueba entre dos trozos y se crea una celda entre cada dos trozos

4) Se clasifican los ocho vértices. Se construye un índice
5) Se busca en la tabla para averiguar las aristas
6) Se calculan las 3 intersecciones entre superficie y arista mediante interpolación lineal
7) Se calculan las normales normalizadas (*Gradiente*) en las 3 intersecciones
8) Se da como salida los vértices de los triángulos y la normal en los vértices

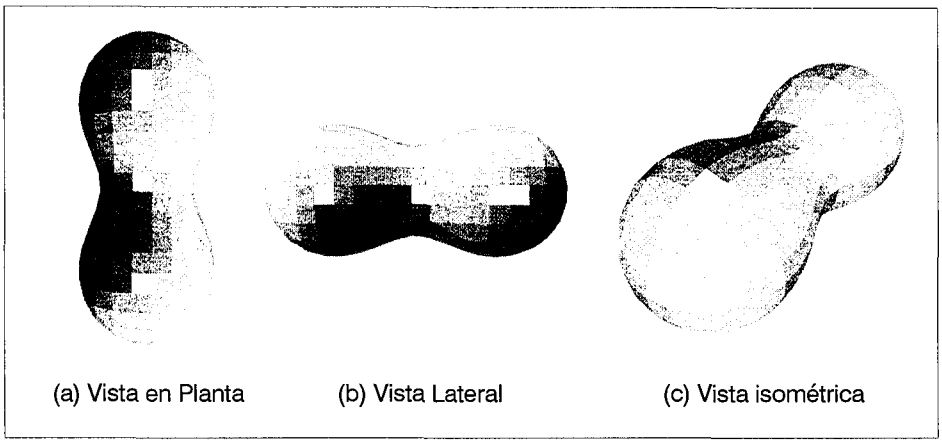
Vóxel: área de valor constante delimitada por una caja que emplea como referencia un punto central



Este algoritmo usa la técnica de "*Divide y Vencerás*" para localizar la superficie en un cubo lógico formado a partir de ocho *vóxels* en los que se descompone dicho cubo. Una vez que el algoritmo determina cómo intersecta la superficie con el cubo, se avanza al siguiente cubo. Puesto que hay *ocho* vértices en cada cubo y dos estados (*interior* o *exterior*), hay 256 formas en las que la superficie puede intersectar con el cubo.

Para resolver estas posibles combinaciones se emplea una tabla con todas las posibles intersecciones entre arista y superficie.

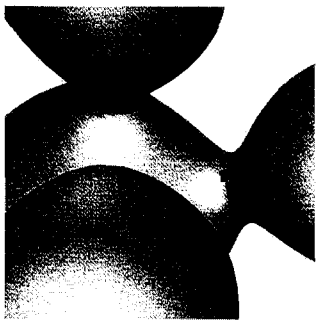
Figura 2-13: metaBlob obtenido por el algoritmo Marching Cubes. *Nótese el efecto vóxel*



Eliminando las simetrías geométricas (rotaciones, reflexiones e intercambio de vértices interiores con vértices exteriores), Lorensen

y Cline demostraron que sólo se tienen que emplear 15 configuraciones básicas para construir los polígonos. Las cuales se muestran en la figura 2-15.

Esta técnica tiene una serie de desventajas:



a) Hay una pérdida de exactitud cuando se visualizan detalles pequeños o difusos.

b) La asunción que se realiza sobre los datos no tiene porqué ser necesariamente correcta.

c) La información del interior de la superficie se pierde, a menos que se almacene junto a la superficie.

d) Además puede ocurrir que queden agujeros en la superficie poligonal resultante.

Aunque ha habido muchos intentos para corregir dichos problemas e incrementar la velocidad del algoritmo, todos los algoritmos resultantes son variantes del algoritmo *Marching Cubes*.

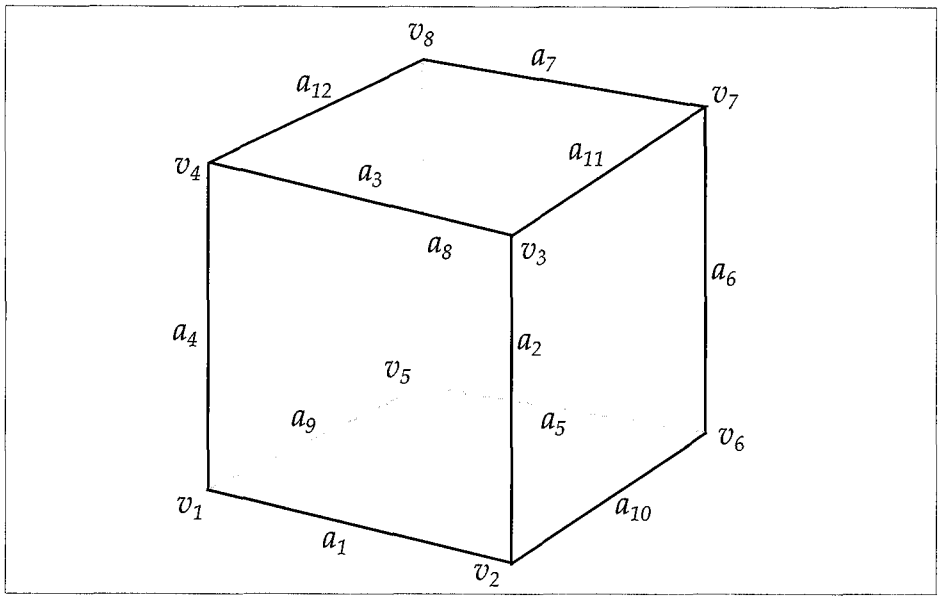


Figura 2-14: Numeración de los vértices y aristas de un vóxel

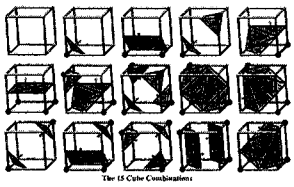
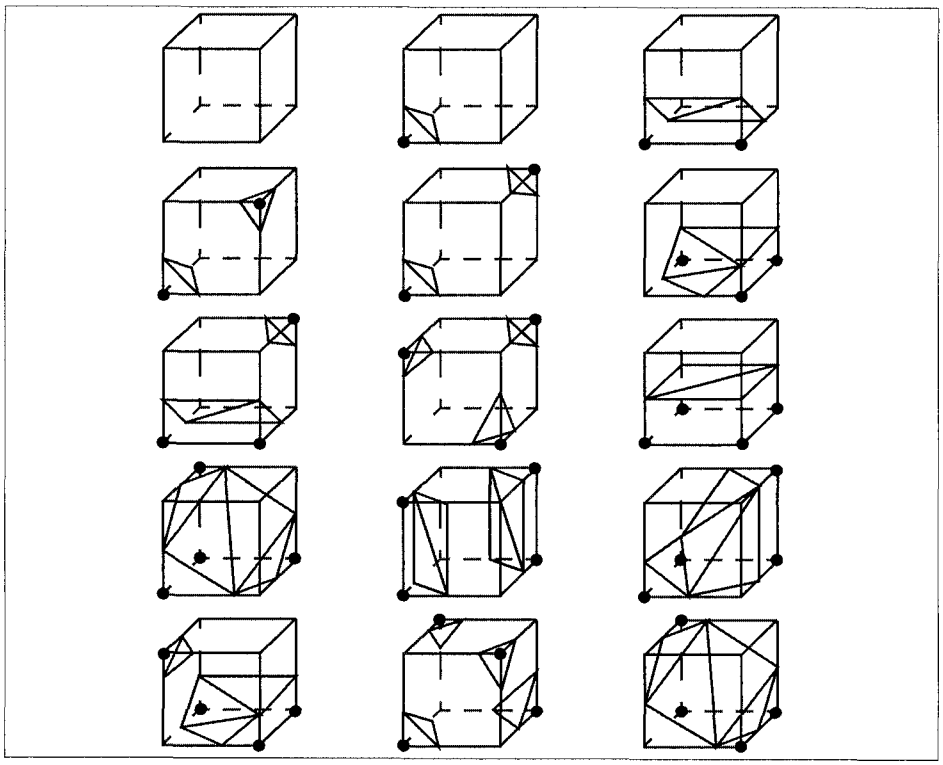
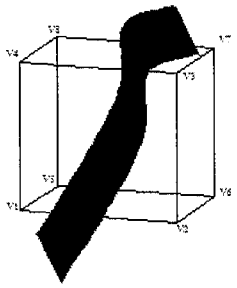


Figura 2-15: Combinaciones resultantes al intersectar un vóxel con la superficie del metaBlob



Para cada vértice se mira si está dentro o no y se construye un índice, o máscara, de la siguiente forma:



v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
0	1	1	0	0	1	1	0

Este índice se emplea junto con una tabla de índices precalculada para saber qué intersecciones hay que emplear y cómo se debe formar los triángulos, siendo el proceso de creación de triángulos en *Marching Cubes* un proceso relativamente rápido si se realiza con una buena implementación.

Ventajas e inconvenientes

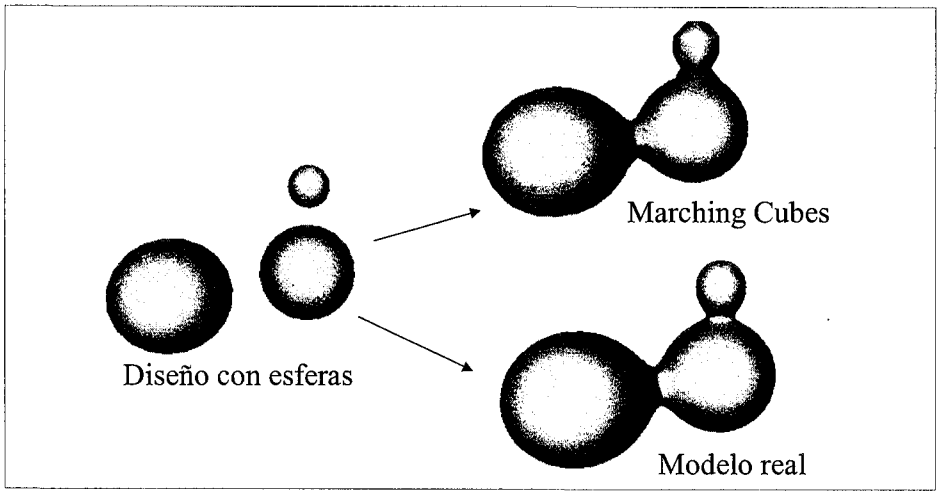
Las ventajas del algoritmo son claras. Podemos enumerar:

1 Sencillez de implementación con respecto a otros algoritmos de poligonalización.
2 Computacionalmente sencillo (coste relativamente bajo), con posibilidad de aprovechar la coherencia espacial.
3 Obtención de una malla poligonal de forma sencilla.

Los inconvenientes del algoritmo son muchos. El principal es que por propia naturaleza del mismo, el rastreo del volumen se realiza a igual nivel de detalle en todas las zonas, por lo que no se asegura la calidad final de la forma. Este problema también se deriva de la escasa información que se toma de la superficie y al ámbito local de la misma. Derivados de éste tenemos:

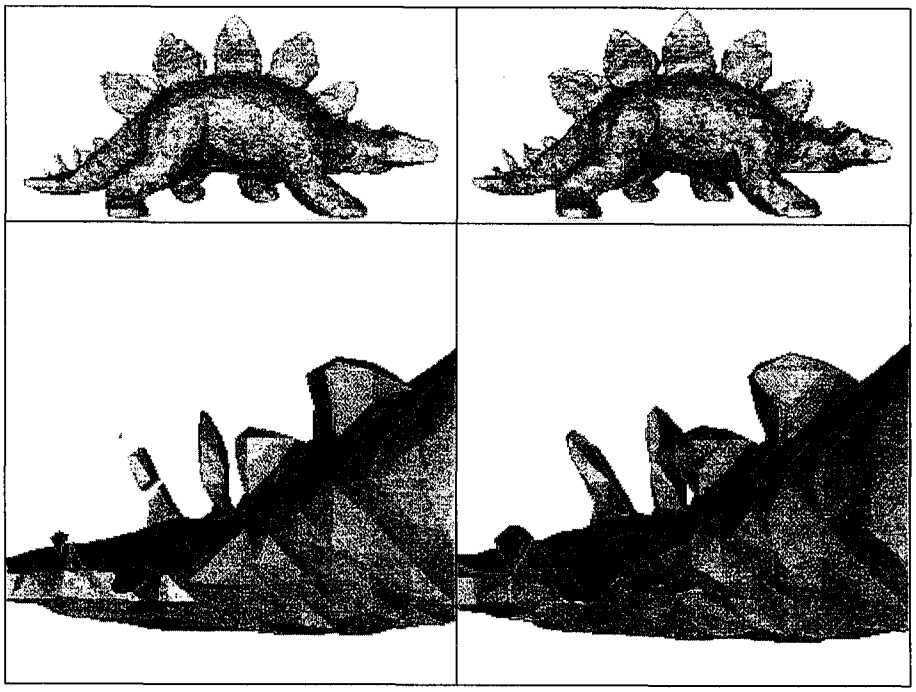
a) No se asegura un ángulo mínimo entre normales de polígonos contiguos, por tanto, la iluminación de la malla poligonal suele contener errores visuales. A mayor tamaño de cubo, mayor es el error en el que se incurre. Esto es lo que se conoce como "efecto vóxel". En la siguiente figura lo podemos observar.

Figura 2-16: Efecto vóxel de un objeto al emplear Marching Cubes



b) No se asegura la coherencia entre el volumen real y el poligonalizado. En algunos casos el modelo no es capaz de captar todo el volumen de la superficie y se detectan superficies separadas donde hay sólo una, y a la inversa. En la figura 2-17 tomada del algoritmo *"Volume Preserving MC"* se puede apreciar dicho problema.

Figura 2-17: Defectos de Marching Cubes



c) La calidad de la forma final depende directamente del tamaño de cubo impuesto para la misma. En la siguiente figura se ha calculado una sola forma con diferentes tamaños de cubo. Puede observarse una degradación de la calidad hasta que llegamos a un resultado erróneo. Las incoherencias espaciales se podrían haber eliminado ajustando el tamaño del cubo a la forma a tratar.

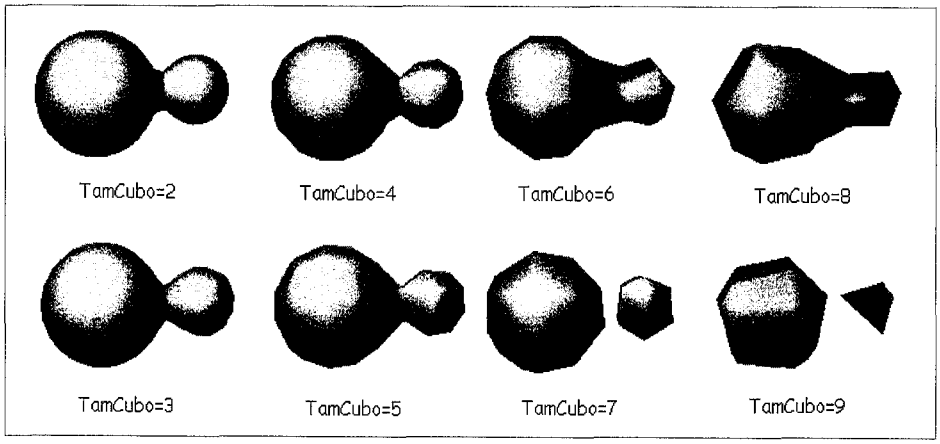


Figura 2-18: Calidad del objeto según el tamaño del cubo

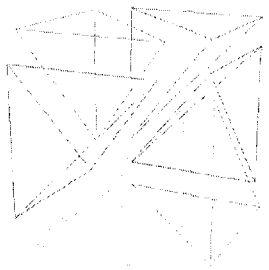
Estos problemas surgieron en la versión inicial del algoritmo, que ha sido modificada por muchos autores, persiguiendo su mejora desde distintas perspectivas. Por ejemplo, en la versión inicial se encontraban huecos en la malla poligonal. Dicho error ha sido solucionado por varios autores añadiendo varios casos de posibles intersecciones del cubo con el blob. Por otro lado, el problema de la coherencia de los volúmenes ha sido solucionado por varios autores. Este es el caso del algoritmo "*Volume preserving MC*". En cuanto al problema del tamaño del cubo, podemos solucionarlo de forma sencilla calculándolo en base a la información geométrica de las primitivas. Este tamaño estaría en función de la primitiva más pequeña que tenga la escena. De esta forma, se asegura que ninguna primitiva se queda totalmente incluida por un cubo, con lo que no obtendríamos intersecciones con la misma, perdiendo la porción de volumen.

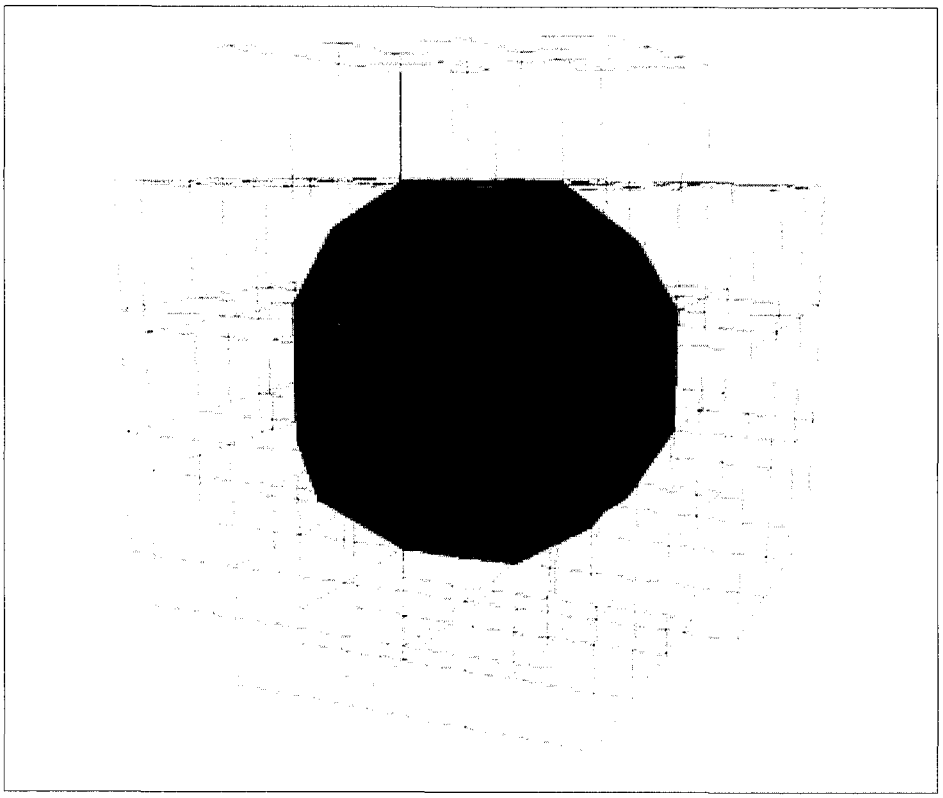
Asegurar la precisión y las normales entre polígonos contiguos es un problema más complicado. Podríamos introducir cualquier técnica de suavizado de aristas para solucionarlo, pero estas técnicas son muy generales y normalmente, al no contar con información del sólido original, hacen que la malla poligonal no sea demasiado parecida a la forma real.

2.6.1 Marching Tetrahedra

Otra modificación interesante del algoritmo *Marching Cubes* es la que aplica la descomposición de cada celda en seis tetraedros, produciendo un polígono triangular o cuadrilateral para cada tetraedro. Este algoritmo recibe el nombre de *Marching Tetrahedra* [**BOUR97**].

Figura 2-19: Descomposición de las celdas de la malla en seis tetraedros

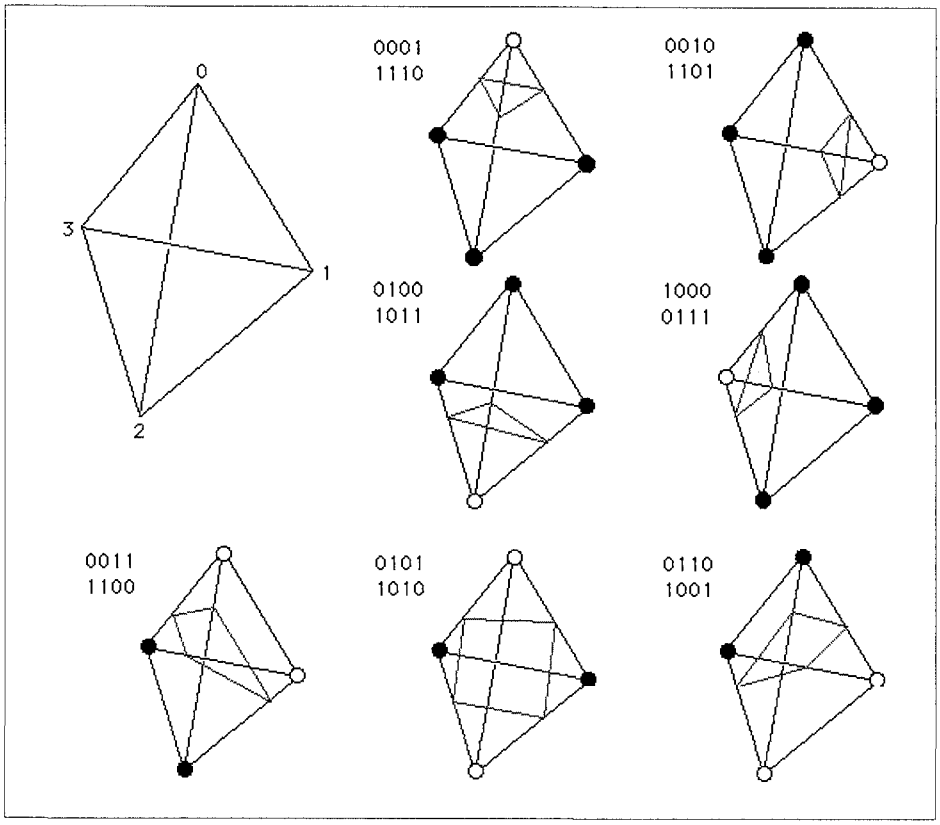




Existen en este algoritmo ocho casos distintos de intersección de cada tetraedro con la superficie que puede generar una isosuperficie.

En la siguiente figura se ilustran estos casos y su índice, o máscara asociada, para el empleo de una tabla de búsqueda al igual que se hace en *Marching Cubes*.

Figura 2-20: Combinaciones de intersectar un tetraedro con la superficie de un metablob



Volver al índice/Tornar a l'índex

2.7 Conclusiones

Los métodos empleados para construir la superficie del modelo metaBlob emplean en última instancia una malla poligonal compuesta, normalmente, por triángulos. Estos triángulos son proyectados, iluminados y visualizados en la pantalla por el procesador gráfico acelerador 3D. Todas las tarjetas gráficas 3D, actualmente, emplean triángulos como base para la representación de cualquier tipo de objeto tridimensional sin importar el modelo empleado para codificar su información. Por tanto, nuestro objetivo se ha centrado en los métodos empleados para generar mallas poligonales.

Hemos realizado la descripción de la formulación matemática que vamos a emplear para construir dichas mallas poligonales, así como uno de los algoritmos más usados para poligonalizar este tipo de modelos, el algoritmo de *Marching Cubes*. Este algoritmo junto con el de triangulación de *Delaunay* son los que empleamos como referentes para realizar comparativas con las técnicas que vamos a desarrollar.

También hemos descrito algunos de los fallos que tienen los algoritmos basados en el empleo de vóxels, siendo nuestro objetivo intentar solucionar algunos de estos problemas.

2 Estado del Arte

Poligonalización de la Superficie

3

Una vez introducida la formulación matemática y los elementos que vamos a emplear como herramientas para la construcción de nuestros modelos, además de los algoritmos que típicamente se han empleado hasta ahora, pasamos a estudiar en profundidad la conversión de isosuperficies generadas por modelos de tipo metaBlob, de forma que se proponen nuevos algoritmos que sustituyen a los empleados en la actualidad, entre ellos el más común es el Marching Cubes y sus variantes. El objetivo es obtener una representación en modelo de fronteras, mediante una codificación en malla poligonal, que sea equivalente a la isosuperficie que representa la función de energía del modelo metaBlob. Además se introduce un control más preciso de la calidad de la superficie con un menor coste temporal.

3 Poligonalización de la Superficie

3.1 Poligonalización de Nubes de Puntos (Delaunay)

En el capítulo anterior se ha visto cómo el algoritmo de *Marching Cubes* puede cometer una serie de errores.

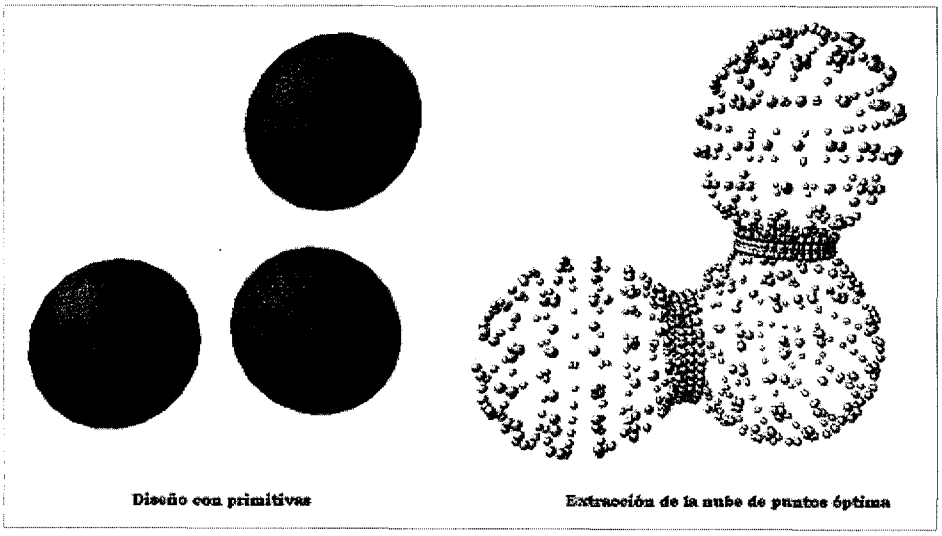
Es por ello que nos hemos decidido a integrar el uso de la derivada de la función de potencial V de un metaBlob en el cálculo de su superficie. Este algoritmo lo llamaremos *Poligonalización de metaBlobs con Rectificación de Precisión, PRP* [**SÁEZ99**]. La ventaja con respecto al anterior es que la generación de la malla poligonal y la rectificación de la misma no son dos procesos independientes. De hecho, la malla poligonal se genera directamente rectificada.

En los siguientes apartados detallaremos el algoritmo que proponemos. Además de la propia explicación teórica de la técnica, aclararemos todas las mejoras y aspectos relevantes a nivel de implementación.

3.1.1 Visión General del Algoritmo

El algoritmo tiene dos partes claramente diferenciadas: la primera parte es la encargada de rastrear el volumen donde se encuentra el metaBlob y obtener del mismo la nube de puntos óptima que deje suficientemente definida su superficie. Dicho conjunto de puntos cumplirá una serie de propiedades, entre las cuales podemos destacar que presentará mayor concentración de puntos en regiones del espacio que precisen de mayor precisión, y menor concentración en regiones que, por el contrario, precisen de menor detalle. Podríamos estudiar todo el volumen a la máxima concentración, pero no obtendríamos mayor precisión. Para realizar este proceso, utilizaremos el gradiente de la fórmula de densidad de forma similar a como se utilizó en el capítulo anterior. Esta primera fase se muestra en la figura 3-1.

Figura 3-1: Modelado y extracción de la nube de puntos óptima



En la figura anterior se puede observar cómo la concentración de puntos depende de la zona de detalle que estudiemos. Esto provocará que la malla que se obtenga de dicha nube de puntos presente las mismas concentraciones de polígonos en las mismas zonas. Como vemos, la rectificación en precisión ya se observa en el primer paso del algoritmo.

La segunda parte del algoritmo consiste en poligonalizar dicha nube de puntos, esto es, a partir de un conjunto de puntos en el espacio pertenecientes a la superficie del metaBlob hemos de encontrar el conjunto de polígonos que encierra al volumen completo y determina la figura.

En la figura 3-2 tenemos un ejemplo de triangulación y representación sólida.

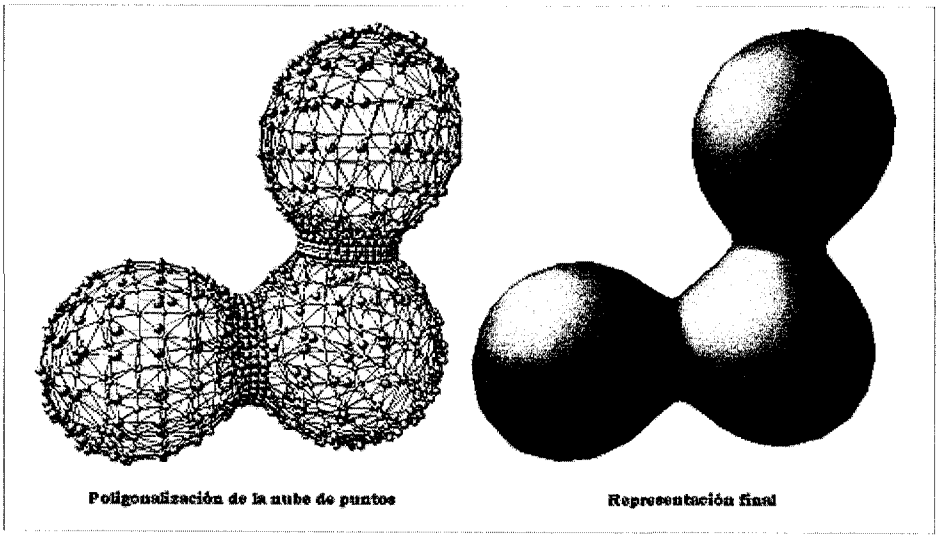


Figura 3-2: Triangulación de la nube de puntos de la figura 3-1 y representación sólida

Las poligonalizaciones de nubes de puntos presentan una causística importante, puesto que no se trata de procesos triviales. Tendremos que aprovechar gran cantidad de propiedades específicas de los metaBlobs para que el coste computacional no se eleve considerablemente.

En los siguientes apartados realizaremos una explicación completa de las distintas partes de las que consta el algoritmo.

3.1.2 Búsqueda de la Nube de Puntos Óptima

En este primer paso del algoritmo tendremos que buscar un conjunto completo de puntos perteneciente a la superficie. Introduciremos el algoritmo mediante la visión geométrica del mismo para después centrarnos en los detalles de su computación.

Visión Lógica de la Búsqueda

Inicialmente, se divide el volumen total del cubo que encierra al metaBlob en cubos más pequeños de tamaño constante. Dicho

tamaño vendrá determinado por la primitiva de menor volumen. Si intersectamos la superficie del metaBlob con elementos de volumen mayor que alguna de sus primitivas, es posible que nuestros cubos engloben alguna de las mismas y no sea posible dicha intersección. Por tanto la división mínima del cubo inicial será tal que los cubos resultantes no tengan un volumen mayor que la mitad del volumen de la primitiva más pequeña.

Después de segmentar el volumen total en pequeños cubos de tamaño constante, intersectamos dichos cubos con el metaBlob, buscando en las aristas de los mismos los puntos pertenecientes a la superficie. Dicho procedimiento coincide con la primera etapa de *Marching Cubes*. A diferencia de éste, en vez de poligonalizar directamente los puntos de intersección de cada cubo, obtenemos el vector gradiente en cada uno de los puntos de intersección que nos proporciona el mismo. Por tanto, tenemos un conjunto de puntos de intersección y los vectores normales de cada punto a la superficie. Si existe un vector tal, que el ángulo con cualquier otro del mismo cubo supera un cierto umbral (unos 30^{o}), dicho cubo necesita mayor grado de estudio. Lo que haremos en este caso será dividir el cubo en vóxels, esto es, en un conjunto de cubos tales que su unión sea igual al cubo inicial. Dicha división podemos hacerla como mínimo en dos partes (caso más sencillo). Si las normales en los puntos de corte del cubo con el metaBlob no forman un ángulo mayor al umbral, dicho conjunto de puntos pertenece a la nube de puntos óptima.

La división del paralelepípedo en vóxels se realiza de forma que los dos paralelepípedos resultantes se asemejen lo mejor posible a un cubo (anchura=altura=profundidad). Dicha división consiste en cortar el paralelepípedo por la dimensión mayor. Por ejemplo, si la dimensión en profundidad es mayor que las dimensiones en altura y anchura, formaremos dos paralelepípedos de igual anchura y altura que el primero, pero con la mitad de profundidad; de forma que de la unión de ambos resultará el paralelepípedo inicial. Los distintos casos de división se muestran en la figura 3-3.

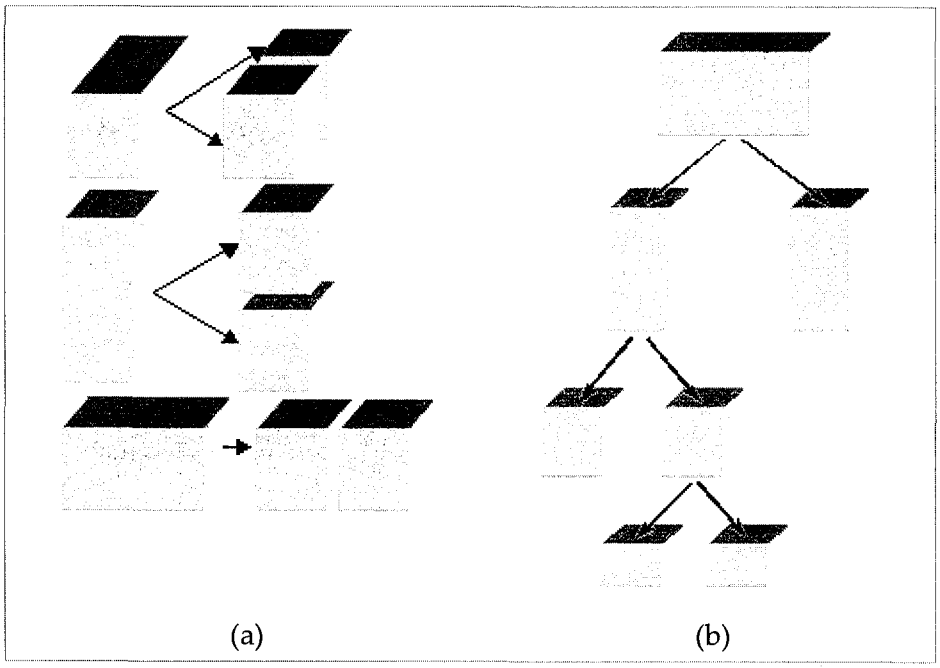


Figura 3-3: (a) División de un paralelepípedo, (b) división recursiva de un cubo

Cada corte provoca un conjunto de cubos derivados, a los que realizaremos el mismo proceso. El algoritmo terminará cuando no existan cubos que cumplan la condición de corte.

Con este procedimiento aseguramos que cada cubo no contenga ningún par de puntos de corte con el metaBlob, cuyos vectores normales difieran del ángulo límite que impongamos. Por tanto, la nube de puntos final cumplirá que los vectores gradiente pertenecientes a cualquier par de puntos cercanos en la misma no diferirán en un ángulo mayor al umbral.

Visión Computacional del Algoritmo.

Una vez introducido el algoritmo, pasamos a relatar cómo sería su implementación vista como una búsqueda (similar a un algoritmo A*). Para ello podemos enfocar el problema de la segmentación en

cubos como un árbol donde la raíz sería el cubo inicial (el cual engloba al volumen total del metaBlob) y los hijos de cada nodo, las divisiones de dicho cubo (figura 3-3). De esta forma, construimos un árbol binario donde las alturas de cada rama nos indican la precisión necesaria en el volumen parcial asociado.

Las hojas pueden contener puntos de intersección, o bien ser cubos no pertenecientes a la frontera (cubos que están completamente fuera del cuerpo o cubos que están completamente dentro). El criterio que hemos tomado para detectar si un cubo está dentro, fuera o intersectando con la superficie, se basa en estudiar el valor del campo en sus vértices; de forma que, si existen vértices interiores (V>0) y exteriores (V<0), estará intersectando y será estudiado. Si todos los vértices están dentro o fuera de la superficie, el cubo no será tratado. La solución del algoritmo vendrá dada por la unión de los conjuntos de puntos de las hojas.

3.1.3 Poligonalización de la Nube de Puntos

A partir del conjunto de puntos obtenido en el primer paso del algoritmo, se ha de construir el conjunto de triángulos que engloba a dichos puntos (metapolígono). Dicho metapolígono ha de ser completo. Podemos definir metapolígono completo como aquel que cumple la siguiente característica: toda arista existente ha de pertenecer a dos, y solamente a dos, polígonos de la malla. Con dicha definición, pretendemos constatar que no han de existir huecos ni cruces poligonales. Realizaremos una técnica que obtenga sólidos con todas las propiedades deseables sobre los mismos, que según Requicha [**REQU92**] serán: objetos completos, no ambiguos y con una interpretación única, precisa y compacta.

La poligonalización de nubes de puntos es un proceso ampliamente estudiado en la bibliografía, tratado desde muchas perspectivas. Por tanto, podríamos poligonalizar la nube de puntos con cualquier método de propósito general. Pero queremos ir más allá y construir un algoritmo de poligonalización de nubes de puntos específico para metaBlobs, aprovechando las características propias de estos

sólidos para llegar a una solución más coherente y con menor coste que una técnica general. Por otro lado, no queremos que todos los puntos de la nube formen parte de la poligonalización. Realizaremos una triangulación inteligente, escogiendo solamente aquellos puntos singulares y utilizando los demás como apoyo para aceptar o rechazar diferentes construcciones posibles.

Para ello hemos adaptado el método de la triangulación de Delaunay para nuestras nubes de puntos. La razón por la que hemos adaptado esta técnica es que nos ofrece un criterio de validez de polígonos local, fácilmente adaptable al caso 3D. Por otro lado, dicho criterio puede unirse al de similitud entre los vectores normales que hemos seguido hasta ahora para que la poligonalización pueda ser coherente con la extracción de la nube de puntos. Con todo ello, tenemos suficiente información local en la superficie para triangularla por partes. Existen otras adaptaciones del algoritmo de Delaunay para triangulaciones 3D **[BORO95]**, pero se han construido de forma general para cualquier nube de puntos, sin contar con información adicional de la superficie ni de la naturaleza del objeto, con lo que la generalización del método provoca un coste excesivo. De igual forma, existen métodos de triangulación de superficies basados en Delaunay, que aseguran una adaptación precisa de la malla resultante **[COUG96]**. El problema de estos métodos es el excesivo coste computacional que requieren, y los criterios de validez de la malla resultante con el sólido original suelen ser demasiado complejos para aplicarlos directamente sobre isosuperficies.

El método que proponemos es una adaptación de los criterios de Delaunay con la información adicional del gradiente del metaBlob y las modificaciones oportunas para el caso 3D, que lo hace más rápido que otras técnicas de poligonalización de mallas, a cambio de una pérdida de generalidad del mismo.

Visión General de la Poligonalización

Inicialmente ordenamos la nube de puntos por orden de x creciente

(esto se puede hacer *in situ* en la primera fase sobre el propio almacenamiento). Este orden nos servirá para reducir drásticamente el espacio de búsqueda. Para cada punto perteneciente a la nube de puntos, construiremos los triángulos posibles que pueda formar dicho punto con sus vecinos cercanos, valiéndonos de la información del gradiente y construyendo lo que denominamos nube cercana a un punto. De todos los posibles, nos quedaremos con aquellos que cumplan el criterio de aceptación de triángulos. El algoritmo termina cuando se recorran todos los puntos.

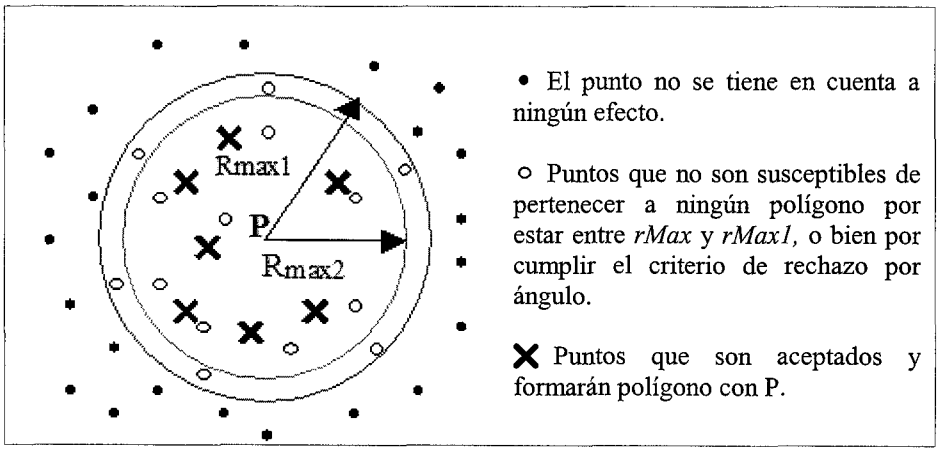
Nube Cercana a un Punto

La nube de puntos cercana a un punto se define como el conjunto de puntos cuya distancia al de estudio es menor a *rMax*. Dicha distancia está en función de la diagonal mayor del cubo más grande aceptado en la primera fase del algoritmo.

Tomaremos una segunda distancia *rMax1*, superior a *rMax*, de forma que los puntos que tengan una distancia al punto de estudio entre *rMax* y *rMax1*, únicamente tendrán significado en el criterio de aceptación de un triángulo, pero nunca en la formación de éste (se consideran excesivamente lejanos). Tampoco serán susceptibles de formar triángulo con el punto de estudio aquellos cuyo ángulo del vector gradiente difiera en un ángulo mayor del umbral con el vector gradiente del punto de estudio. En la figura 3-4 presentamos un gráfico del cálculo de la nube cercana.

La distancia *rMax1* se puede tomar tan grande como se quiera, pero es conveniente tomarla más bien corta para no afectar en el tiempo de cómputo. En los distintos experimentos hemos comprobado que, tomando *rMax1=1.25 rMax*, se obtienen buenos resultados.

Figura 3-4: :
Nube cercana a un punto



En definitiva, todos los puntos cuya distancia al actual sea menor que *rMax1*, pertenecerán a la nube cercana de puntos, pero aquellos que se encuentren entre *rMax* y *rMax1*, serán marcados como pasados; al igual que aquellos cuyo vector gradiente difiera más del ángulo máximo con el gradiente del punto de estudio.

3.1.4 Poligonalización de la Nube Cercana a un Punto

El algoritmo de poligonalización de una nube cercana a un punto consiste básicamente en estudiar todas las combinaciones de tres puntos de la nube cercana y eliminar aquellos que no cumplan las condiciones de la definición de metapolígono completo. El estudio de dichas condiciones se basa en el criterio de aceptación de un triángulo.

Criterio de Aceptación de un Triángulo.

Este criterio nos dirá si tres puntos de una nube forman un polígono correcto (que asegurará un metapolígono completo).

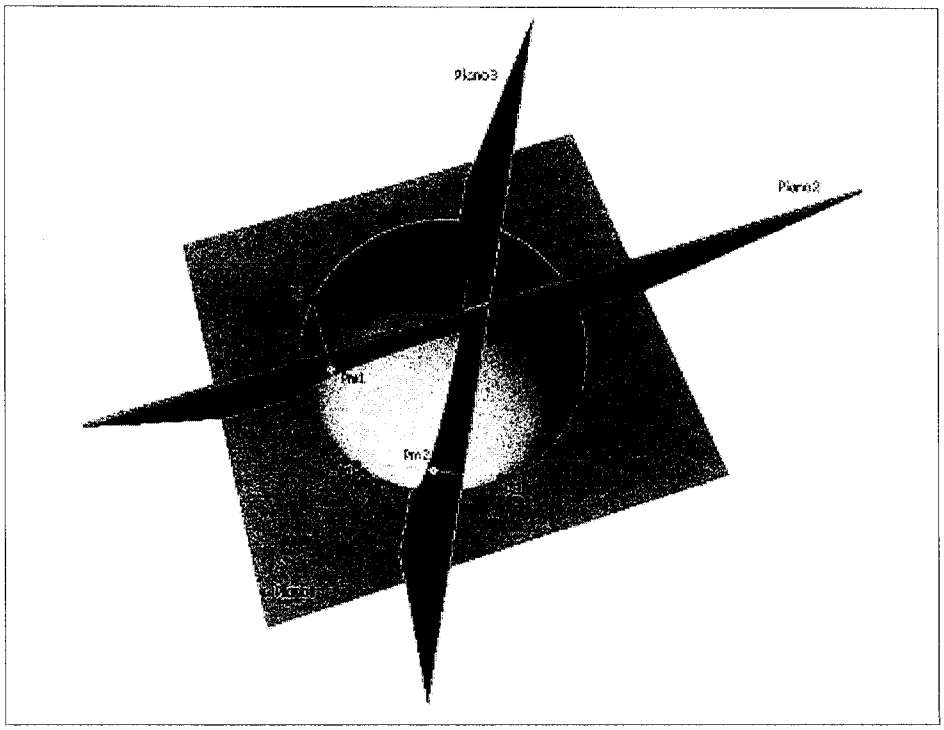
En la triangulación de nubes de puntos en 2D de Delaunay, se expone que, dados tres puntos pertenecientes a una nube en un plano 2D, éstos formarán un triángulo correcto perteneciente a la

triangulación final si se cumple que no existe ningún otro punto dentro del área del círculo determinado por los tres candidatos. Dicha restricción se aplica a todas las posibles combinaciones de puntos pertenecientes a la nube, asegurando una triangulación completa. Para nuestro caso en particular, hemos utilizado dicha restricción adaptándola al problema en 3D de la forma:

Tres puntos (p_1, p_2, p_3) pertenecientes a una nube en un espacio 3D, formarán un triángulo si y sólo si no existe ningún otro punto dentro del volumen determinado por la esfera que contiene en su superficie a los tres puntos iniciales, y cuyo centro se encuentra dentro del plano que contiene a dichos puntos.

El cálculo del centro de la esfera puede realizarse de forma sencilla mediante la intersección de los tres planos que se observan en la figura 3-5.

Figura 3-5:
Criterio de adaptación de Delaunay



La construcción de dichos planos se realiza de la siguiente manera:

- **Plano 1** - Plano que pasa por los tres puntos.
- **Plano 2** - Plano que pasa por el punto medio entre p_1 y p_2 y cuyo vector normal es $\vec{N} = p_2 - p_1$
- **Plano 3** - Plano que pasa por el punto medio de p_1 y p_3 y cuyo vector normal es $\vec{N} = p_3 - p_1$

El centro de la esfera se calcula resolviendo el sistema de tres ecuaciones con tres incógnitas formado por las ecuaciones de los planos calculados. El radio de la esfera se calcula por la distancia del centro a cualquiera de los tres puntos. Como ocurría en la triangulación de Delaunay, (p_1, p_2, p_3) formarán un triángulo válido si dentro de dicha esfera no hay ningún otro punto de la nube cercana.

Poligonalización de la Nube de Puntos Cercana

Con los conceptos de nube cercana a un punto y criterio de aceptación de un triángulo, podemos poligonalizar dicha nube cercana de la siguiente forma:

1) A partir de la nube cercana a un punto, formar todas las combinaciones de tres puntos que cumplan:
   a) Uno de los tres puntos ha de ser el de estudio (P).
   b) Un punto únicamente puede aparecer una vez en la tripleta.
   c) La tripleta no estará formada por ningún punto marcado como *pasado*.
2) Eliminar aquellas combinaciones que no cumplan el criterio de aceptación de triángulo para cualquier punto perteneciente a la nube cercana.
3) Añadir las tripletas resultantes a la lista global de triángulos.

El procedimiento de cálculo de la nube de puntos cercana puede

verse como una aproximación 2D de una porción de superficie, pues se toma una parte de la misma muy pequeña. Igualmente, el criterio de Delaunay adaptado a 3D, puede verse como el original en 2D aplicado a dicha porción de superficie bidimensional.

Por todo ello, el procedimiento es similar a haber realizado una triangulación de Delaunay en 2D. Con lo que podemos asegurar que la triangulación de una nube de puntos cercana a un punto siguiendo este procedimiento es correcta.

Poligonalización Completa.

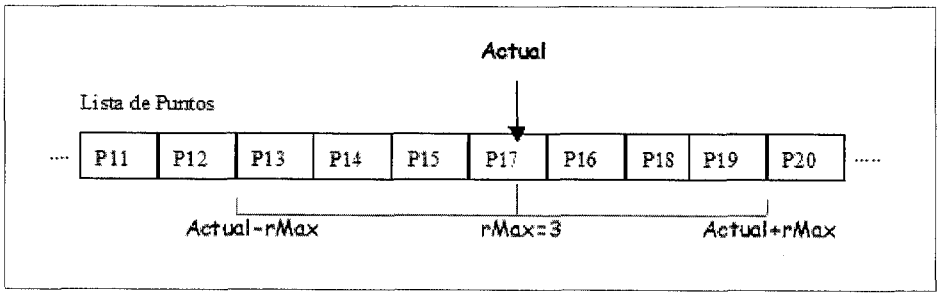
Una vez que podemos poligonalizar la nube de puntos cercana a un punto, obtendremos la malla general de la siguiente manera:

Partiendo de la nube de puntos ordenada en x creciente, recorremos la lista ascendentemente según la coordenada y, para cada punto, calculamos la nube cercana al mismo. Poligonalizamos dicha nube cercana y marcamos el punto como pasado. De esta forma, aseguramos que dicho punto no vuelva a escogerse para poligonalizar, pero sí afecte en el criterio de aceptación de cualquier triángulo sucesivo de la nube cercana a otro punto. El algoritmo termina cuando hemos estudiado toda la lista de puntos.

El coste computacional de la poligonalización de una nube de puntos cercana es muy bajo, pues siempre consta de pocos elementos (distancias muy cortas). Sin embargo, la búsqueda de la nube de puntos cercana tiene coste computacional N (siendo N el número de puntos) y, dado que tenemos que realizarla N veces (1 para cada punto), el coste es cuadrático con respecto al número de puntos. La ordenación en x inicial de la nube de puntos se fundamenta en reducir dicho coste, pues la búsqueda de los elementos que están a una distancia inferior a *rMax1* se realiza dentro de una ventana en la lista de puntos [*actual.x - rMax1*, *actual.x + rMax1*]. Esto es factible al cumplirse la condición de que, si dos puntos están a una distancia euclidea r, cada una de sus coordenadas por separado pueden diferir en valor absoluto r como máximo (distancia cuadrada). Dicho procedimiento puede verse en

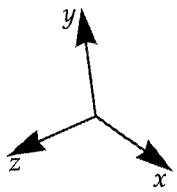
la figura 3-6.

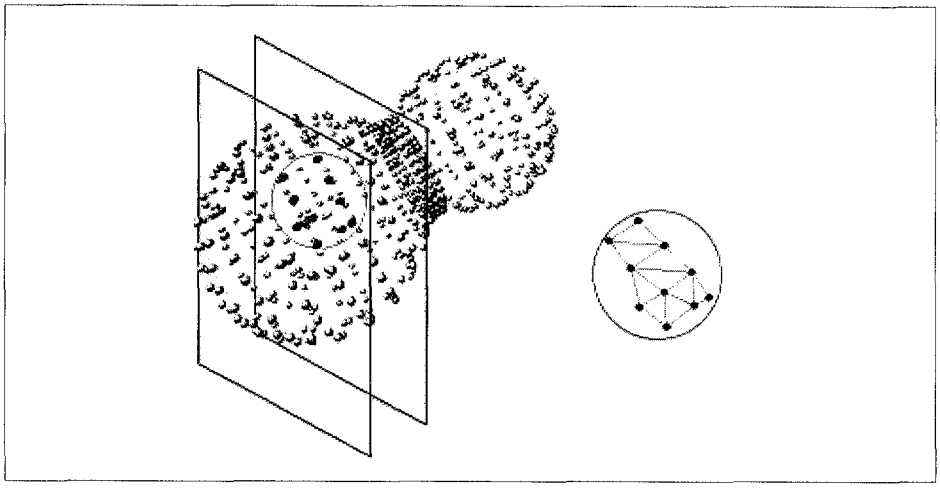
Figura 3-6: Búsqueda en una lista mediante una ventana de puntos cercanos.



El empleo de este método reduce el coste de búsqueda de nubes de puntos a un coste lineal. La utilización de esta ventana de búsqueda puede observarse geométricamente en la figura 3-7.

Figura 3-7: Visión geométrica de la ventana de búsqueda





También podemos reducir el coste de la inserción de dichos puntos de forma ordenada en la primera fase del algoritmo guardándonos la posición de la última inserción en lista y formando la lista doblemente enlazada. Puesto que las inserciones suelen realizarse con puntos cercanos al punto insertado inmediatamente antes, cada inserción se realizaría en un reducido número de iteraciones y no

tenemos que hacer largos recorridos por la lista para encontrar la colocación del punto.

Puesto que hemos utilizando el criterio de triangulación de una nube de puntos cercana para todos los puntos de la nube general, podemos asegurar que la triangulación final es correcta y, por tanto, el algoritmo obtiene una superficie que satisface las propiedades generales que propone Requicha [**REQU92**].

3.1.5 Resultados, Comparativas y Ejemplos.

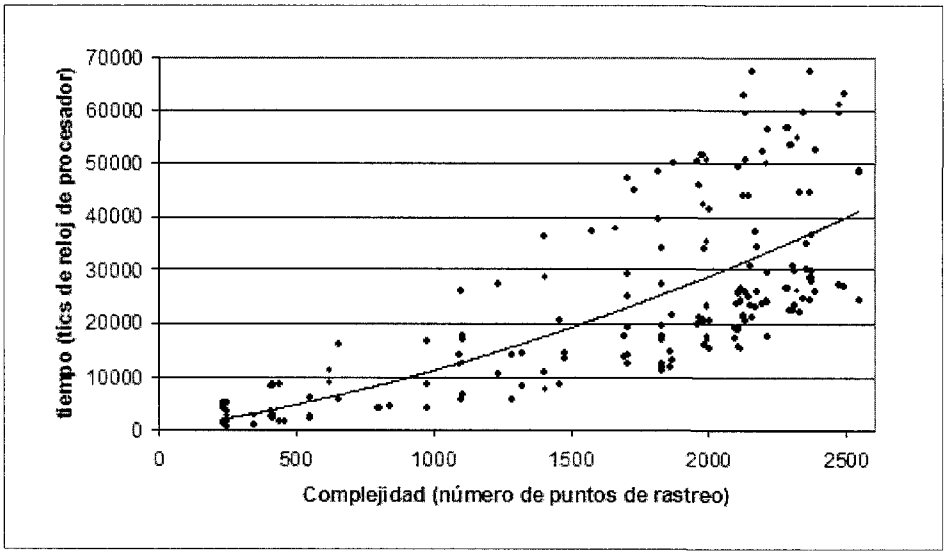
Ventajas, Inconvenientes y Coste General del Algoritmo

Entre las ventajas observadas en la implementación del algoritmo, podemos destacar la elevada precisión que presenta en zonas con pequeños detalles. Esta característica es la que lo diferencia respecto a otros algoritmos similares. Por otro lado, se obtiene un modelo detallado con un nivel de precisión local. La obtención de modelos detallados con ahorro en el número de polígonos para zonas de bajo detalle provoca que los metapolígonos resultantes sean computacionalmente manejables y con un nivel óptimo de calidad. Finalmente, podemos destacar que se obtienen modelos optimizados para su iluminación, pues el modelo asegura cambios de normales entre polígonos adyacentes inferiores al umbral.

El coste del algoritmo en general no puede estimarse de forma sencilla puesto que depende de la complejidad de los cuerpos a tratar. Podemos tomar como medida de complejidad de un metaBlob el número de puntos obtenidos en la primera fase de rastreo del algoritmo. Esta será una medida fiable de complejidad, puesto que en dicha fase se obtiene mayor cantidad de puntos cuanto más complicada es la figura. En la figura 3-8 hemos representado el tiempo de cálculo para un conjunto de muestras frente al número de puntos extraídos para las mismas. También se presenta la línea de tendencia o regresión polinómica de dicho

conjunto de puntos, que nos proporciona una noción del coste del algoritmo.

Figura 3-8:
Coste computacional de Delaunay



Como inconveniente principal del algoritmo, podemos destacar que se produce un aumento del coste computacional en comparación con *Marching Cubes*. Este inconveniente era de esperar dada la simplicidad del primero y la complejidad intrínseca de lo que se pretendía obtener con el segundo. Lo que se perseguía era la obtención de los modelos poligonales correctos para su uso posterior, no el cálculo de los mismos en tiempo real.

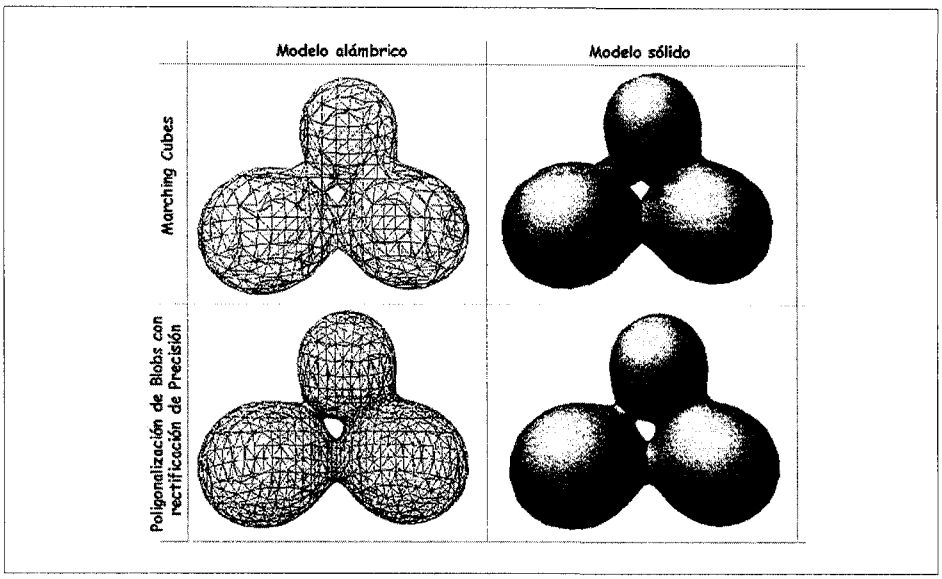
Comparativas y Ejemplos

En este apartado realizaremos una comparativa entre el algoritmo Marching Cubes y nuestro algoritmo, para obtener visualmente las ventajas e inconvenientes de éste. En la figura 3-9 hemos construido un metaBlob con tres primitivas, obteniendo el modelo alámbrico y el sólido de una misma figura con ambos algoritmos.

Como podemos observar en dicha figura, nuestro algoritmo aporta mayor número de polígonos en aquellas zonas donde mayor

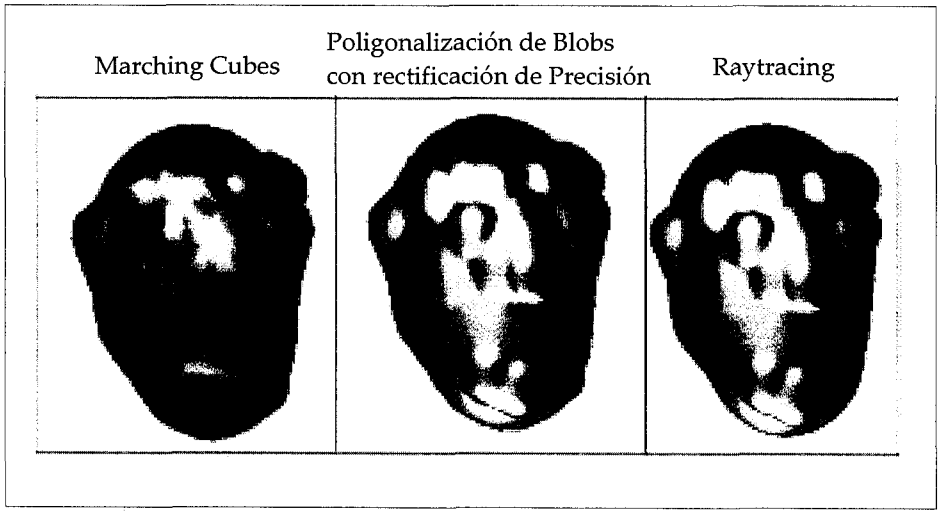
precisión se requiere, obteniendo un modelo mucho más completo. Igualmente puede observarse que la malla poligonal es muy parecida en cuanto a densidad de los polígonos en zonas de precisión más relajada, lo cual era de esperar, puesto que la rectificación de precisión se aplica de forma local. Finalmente, también puede observarse la mejora en la iluminación, al eliminar el efecto vóxel.

Figura 3-9: Comparativa entre Marching Cubes y el algoritmo propuesto



En la figura 3-10 incluimos un ejemplo completo de la construcción de una escena con metaBlobs utilizando el algoritmo propuesto.

Figura 3-10: Comparación de calidad empleando rectificación de precisión



3.1.6 Conclusiones

Como vemos, hemos conseguido construir un algoritmo completo para la poligonalización de metaBlobs que resuelve algunos de los problemas de ambigüedades que presentaban otras técnicas y que proporciona un modelo poligonal con un grado de precisión que mejora la calidad que se puede conseguir en un modelo de fronteras.

Pese a que la técnica es complicada y el coste de la misma indeterminado, presenta una buena alternativa para el modelado con metaBlobs que hasta ahora no había sido resuelta satisfactoriamente.

3.2 Rectificación de Precisión

En este apartado realizaremos una descripción detallada del algoritmo de rectificación de precisión [**PUCH99**] que proponemos. Construiremos un postproceso de *Marching Cubes* lo suficientemente rápido como para no incrementar considerablemente el coste de éste. El objetivo del postproceso será resolver los problemas de precisión de *Marching Cubes*. La idea es obtener una malla poligonal más adaptada a la superficie, y por tanto más precisa, a partir de la que nos proporciona el algoritmo de *Marching Cubes*.

Este procedimiento asegurará la normal entre triángulos contiguos, aplicando criterios basados en información local. Por tanto, también resolveremos problemas relacionados con la pérdida de exactitud en zonas que requieran alto nivel de detalle.

Nos centraremos en la rectificación de la malla poligonal, de forma que en cada volumen parcial donde necesitemos mayor grado de exactitud, se aumente la concentración de polígonos, respetando la malla original en aquellas zonas que presenten calidad suficiente. Para introducir el algoritmo veremos inicialmente cómo puede ayudarnos la expresión del gradiente del metaBlob.

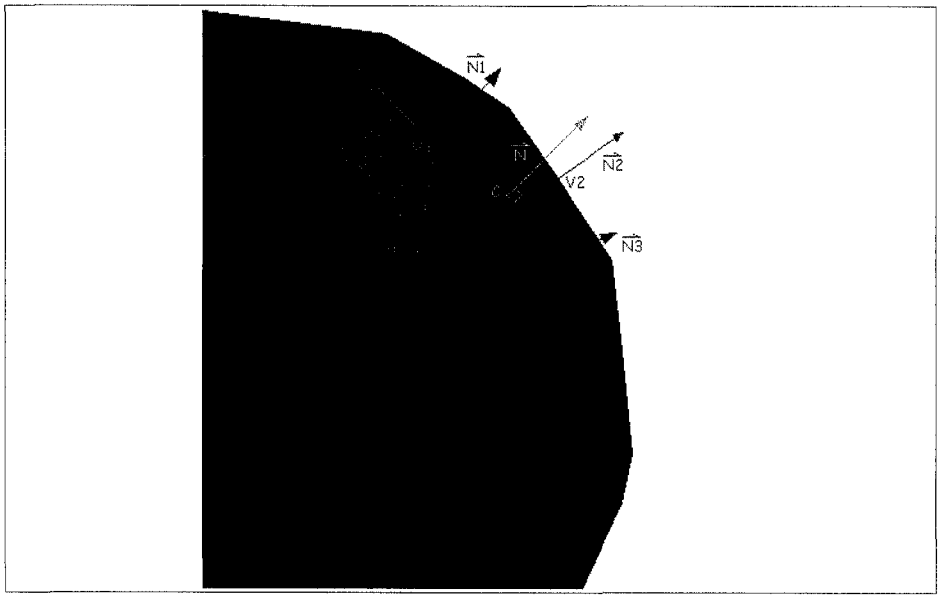
3.2.1 El Gradiente como Medida de Precisión

El gradiente de la fórmula de densidad en un punto se puede definir como el vector normal del plano tangente a la superficie en dicho punto. En todos los procesos de ocultación e iluminación se utiliza la normal de los polígonos. Dicha normal se puede obtener si todos los polígonos de la figura vienen expresados en el mismo sentido (dextrógiro o levógiro). Teniendo la normal en cada vértice mediante el gradiente, podemos hacer la media aritmética de las mismas para obtener la normal del triángulo directamente. Dicho vector nos servirá como medida de precisión de un triángulo, pues nos marca lo bien o mal colocado que está éste.

$$GradV(x,y,z) = \frac{\partial V(x,y,z)}{\partial x}\vec{i} + \frac{\partial V(x,y,z)}{\partial y}\vec{j} + \frac{\partial V(x,y,z)}{\partial z}\vec{k} \quad (3.1)$$

En la figura 3-11 podemos observar la representación geométrica del vector gradiente dentro del problema que nos ocupa. En dicha figura se representan los vectores gradiente calculados en los tres vértices de uno de los triángulos de la malla poligonal perteneciente a un metaBlob.

Figura 3-11: Cálculo de la normal de un triángulo en base al vector gradiente.



V_1, V_2 y V_3 son los vértices que componen el triángulo T y N_1, N_2 y N_3 son las normales de dichos vértices respectivamente. C es el centroide del polígono (media aritmética de las coordenadas de sus vértices) y N es la normal resultante de realizar la media aritmética de las coordenadas de las normales de los vértices.

En los procesos de defacetado, como el de Phong, se usan los vectores gradiente en los vértices, interpolándolos para suavizar el triángulo. El cálculo de la normal en los vértices de una malla común requiere la construcción de una estructura compleja que nos

proporcione los triángulos asociados a cada vértice. Teniendo el gradiente, no es preciso dicho cálculo adicional, puesto que nos proporciona la normal en cualquier punto de manera más rápida y precisa.

Para cada triángulo perteneciente a la malla poligonal que nos calcula Marching Cubes podemos obtener el vector gradiente en cada uno de sus vértices (v_1, v_2, v_3). Seguidamente, obtenemos el ángulo que forman dos a dos los tres vectores gradiente. Con ello resultan tres ángulos: el primero (ángulo entre v_1 y v_2), el segundo (ángulo entre v_2 y v_3), y el tercero (ángulo entre v_3 y $v1$). Definimos también el umbral U, que simboliza el valor máximo que pueden tomar dichos ángulos. Por tanto, la tripleta [1, 2, 3], nos marcará una medida de precisión de cada triángulo. El umbral nos servirá para marcar el grado de detalle que queremos para todos los triángulos que forman la malla poligonal.

Como medida adicional, marcaremos la longitud máxima L que puede tomar una arista perteneciente a un triángulo. De esta forma, se asegura que no habrá polígonos con aristas excesivamente alargadas. Esto nos servirá para que la malla resultante sea más homogénea y no se generen triángulos degenerados. Para ello, tendremos igualmente un vector de longitudes [$l1$, $l2$, $l3$] obtenido mediante el cálculo de las distancias euclídeas entre los pares (v_1, v_2), (v_2, v_3) y (v_3, v_1) respectivamente.

3.2.2 División Poligonal de la Malla

Contando con la medida de precisión de un triángulo, podemos realizar un algoritmo que adapte la malla que nos calcula Marching Cubes, hasta que todos los triángulos de dicha malla cumplan que el ángulo máximo entre cualesquiera dos vectores gradiente, medidos en los vértices del mismo, sea menor que U. Igualmente, la longitud de las distintas aristas que lo componen no excederá L.

El procedimiento puede observarse en el siguiente algoritmo:

Figura 3-12: Algoritmo de subdivisión poligonal

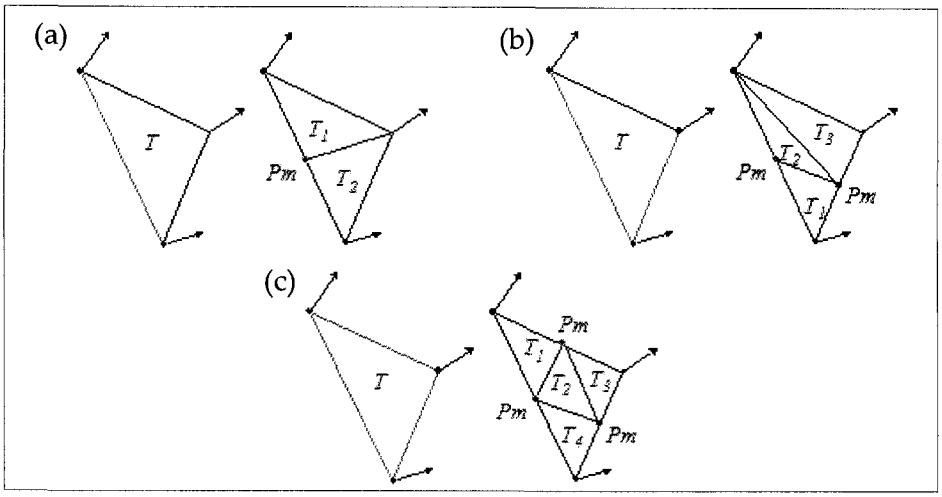
```
Función DivisionPoligonal (MallaTriangular M, Umbral U, Dmáxima L)
Var   MallaTriangular: M_final;
Triángulo: T;
Angulo: a1, a2, a3;
Distancia: l1 l2 l3; Finvar

Para cada triángulo T perteneciente M hacer
        Obtener los tres vectores gradiente de T en sus vértices
        Calcular los tres ángulos [a1, a2, a3] entre los vectores gradiente.
        Calcular las longitudes de las aristas del triángulo [l1, l2, l3]
        Si a1>U o a2>U o a3>U o l1>L o l2>L o l3>L entonces
                Eliminar T de M
                Añadir particiones de T a M según [a1, a2, a3] y [l1, l2, l3]
                y adaptarlas a la superficie
        Sino
                Eliminar T de M
                Añadir T a M_final
        Fsi
Fpara
Devolver M_final
FinFunc
```

Como puede observarse, esta es una función recursiva, puesto que sobre la misma malla insertamos las particiones de los triángulos y éstos vuelven a ser procesados hasta que todos son aceptados por el criterio. Es conveniente determinar una altura suficiente del árbol de recursión, esto es, permitir un número de particiones máximo en un triángulo para no caer en un nivel de detalle excesivo.

El problema ahora consiste en dividir de forma eficiente los triángulos cuando no superan el criterio. Las particiones se realizarán respetando aquellas aristas que superen el criterio y

dejando el resto sin modificaciones. Así, los cambios en aristas compartidas por dos polígonos vecinos se harán de la misma forma en ambos. Por otro lado, los nuevos vértices generados se toman en los puntos medios de las aristas implicadas. Los tres casos posibles de partición aparecen en las figura 3-13.

Figura 3-13: Casos de división de un triángulo.



a) Caso más sencillo de partición de un triángulo. El ángulo entre los vectores normales de una pareja de vértices es mayor al umbral, pero las otras dos parejas forman ángulos menores. Se obtienen dos triángulos que tendrán como vértices comunes el punto medio entre los dos que fallan y el tercero.

b) Segundo caso de partición de un triángulo. El ángulo entre los vectores normales de dos parejas de vértices es mayor al umbral, pero la última pareja forma un ángulo menor. Se obtienen tres triángulos a partir de los puntos medios de las aristas que fallan y los vértices originales. Obsérvese que la arista cuyos vértices forman un ángulo correcto queda igual.

c) Tercer caso de partición de un triángulo. El ángulo entre los vectores normales de las tres parejas de vértices es mayor al umbral. Se obtienen cuatro triángulos a partir de los puntos medios de los vértices que fallan y los vértices originales.

Las posibles subdivisiones de un triángulo se reducen a tres eliminando simetrías. En el primer caso, el triángulo únicamente se divide en dos, pues sólo falla una de las aristas. En el segundo caso, fallan dos aristas, con lo que se generan tres triángulos. El tercer caso es el más desfavorable puesto que fallan todas las aristas y el triángulo ha de dividirse en cuatro partes. La subdivisión siempre se realiza tomando los puntos medios de las aristas que fallan como nuevos vértices de los triángulos que se generan, combinados con los vértices del triángulo original.

Utilizando estos criterios de partición y adaptando los nuevos vértices a la superficie, como veremos a continuación, se consigue que los ángulos entre vectores normales y las longitudes de las aristas decrezcan hasta cumplir el criterio.

Una vez dividido el triángulo, el problema es adaptarlo convenientemente a la superficie del metaBlob. Los vértices originales del triángulo dividido pertenecen a la frontera del metaBlob, pero los nuevos vértices generados no tienen por qué cumplir esa propiedad. Por ello, tenemos que hacer coincidir los nuevos vértices con la frontera, como se aprecia en la figura 3-14.

En la siguiente figura, el triángulo original está formado por los vértices P_1, P_2 y P_3 y los vectores gradiente en los puntos P_1 y P_2 forman un ángulo mayor al umbral (hay un valle bajo éstas). El ángulo entre los vectores gradiente P_1 y P_3 así como el de P_2 y P_3 son correctos. Dividimos el triángulo en dos (primer caso): $[P_1, P_m, P_3]$ y $[P_3, P_2, P_m]$ siendo P_m el punto medio entre P_1 y P_2. Para ajustar el nuevo vértice (P_m), calculamos la normal en él ($\vec{N}$) como la media de los vectores gradiente de P_1 y P_2. Como el valor del campo en Pm es negativo (exterior), nos movemos en dirección opuesta a la normal en la línea definida por el punto P_m y dicha normal, hasta encontrar la nueva posición de P_m sobre la superficie del metaBlob.

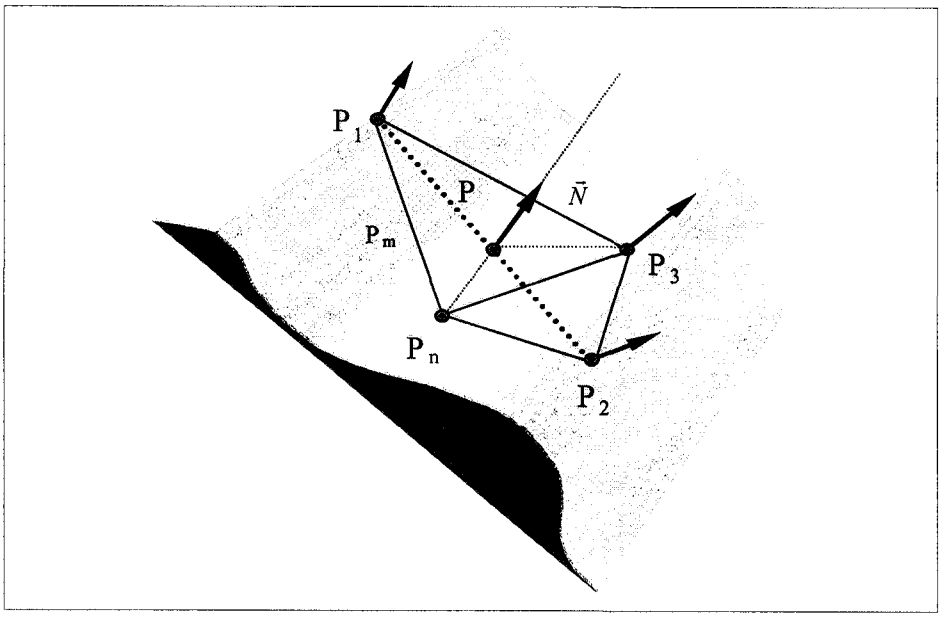


Figura 3-14: Ajuste del nuevo vértice producido por la división de un triángulo.

La razón por la que se toma la media de los vectores normales de la arista en vez del propio gradiente de V en el punto medio de ésta, es que es posible que intentemos calcular el gradiente dentro de una región de campo constante. Esto se debe a que el punto medio no tiene por qué estar contenido en la superficie. Por tanto, el gradiente se anularía y no sabríamos hacia dónde buscar el nuevo punto. Tomando la media de las normales en los vértices de la arista, aseguramos encontrar una dirección de búsqueda puesto que los tres vértices del triángulo original sí pertenecían a la superficie del metaBlob y por tanto su gradiente no podía ser nulo. Una vez encontrado el punto en la superficie, podemos asignar como normal del mismo el propio gradiente del campo en dicho punto.

3.2.3 Costes Computacionales

En el siguiente apartado realizaremos una comparativa con respecto al coste entre el algoritmo Marching Cubes y la rectificación propuesta sobre el mismo. Inicialmente calcularemos

el coste de forma teórica para después contrastarlo con las pruebas prácticas que se han realizado.

Coste Teórico del Método

Siendo V el volumen del paralelepípedo inicial que encierra al conjunto de primitivas y T el volumen constante de los cubos en los que se divide el mismo, se producen V/T iteraciones del algoritmo. Puesto que cada iteración se produce en un tiempo constante (calcular los cortes del cubo con el metaBlob y producir la triangulación de los mismos) y T es constante, el procedimiento Marching Cubes es lineal con respecto al volumen.

En cuanto al postproceso, se aplica a cada uno de los triángulos que genera *Marching Cubes*. Para cada cubo, el número de triángulos que pueden generarse mediante el algoritmo de *Marching Cubes* está entre 0 y 4 (figura 3-13). Por tanto, en el mejor de los casos, cuando ningún cubo genere triángulo alguno, el postproceso no se aplicará y el coste seguirá siendo lineal con respecto al volumen. En el peor de los casos se producirán 4 triángulos por cubo, lo que suponen $4V/T$ triángulos, a los que se les aplicará el postproceso.

El postproceso realiza un número de iteraciones adicionales por triángulo que dependen de la propia colocación del mismo. Puesto que a partir de cierta altura en el árbol de división de los triángulos la segmentación resulta excesiva, introducimos una altura máxima del mismo a la que llamaremos H. Por tanto, en el peor de los casos se producirán $4H$ iteraciones para cada triángulo, puesto que como máximo se producen 4 particiones por triángulo (figura 3-13c), y en el mejor de los casos, ninguna. Cuando no se produzca ninguna división para ningún triángulo, el número de iteraciones del algoritmo con la rectificación será igual que en el algoritmo *Marching Cubes*. En el caso en el que se produzcan todas las particiones posibles de todos los triángulos, el número de iteraciones será $4(V/T)4H$. Dado que H y T son constantes, el coste sigue siendo lineal con respecto al volumen.

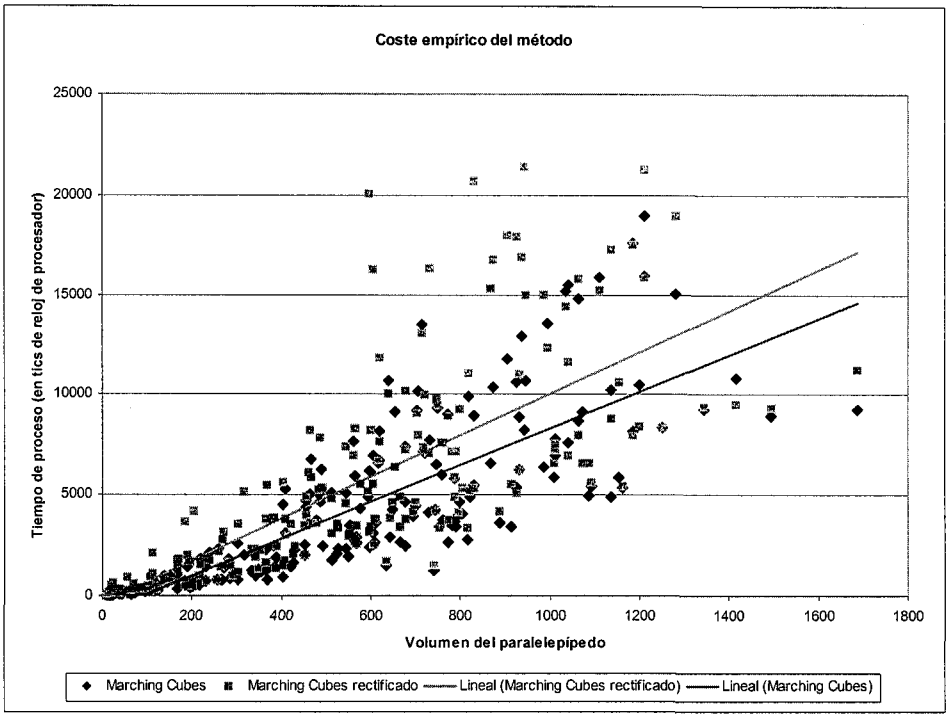
Finalmente podemos afirmar que si tanto para el mejor como para

el peor de los casos el coste es lineal, el algoritmo con el postproceso es lineal respecto al volumen.

Coste Empírico del Método

En la figura 3-15 se ha representado el tiempo de cálculo de ambos algoritmos para el mismo conjunto de muestras, frente al volumen del paralelepípedo que encierra al metaBlob. Sobre la misma gráfica hemos representado la recta de regresión lineal de cada serie para observar cómo afecta el postproceso de forma general.

Figura 3-15: Representación del coste empírico de ambos algoritmos



Cada muestra se ha formado con un número de primitivas aleatorio entre 1 y 10. La posición de cada primitiva ha sido tomada también de forma aleatoria dentro de un volumen de tamaño 5x5x5. Para el tamaño de cada primitiva se ha tomado igualmente un valor aleatorio entre 0 y 4. Dichas medidas están referidas a un espacio

tridimensional estándar de OpenGL. El umbral de ángulo tomado para la rectificación ha sido $25°$. En todos los casos se han tomado primitivas esféricas.

Como vemos en la figura 3-15, las rectas de regresión de ambos algoritmos resultan similares, a excepción del incremento de pendiente del algoritmo rectificado con respecto al original. La magnitud del incremento de pendiente es inversamente proporcional al umbral que marquemos como ángulo máximo entre normales en la rectificación de la malla, puesto que a menor umbral, mayor es la rectificación en la que incurrimos y, por tanto, mayor es la diferencia temporal con *Marching Cubes*.

Ejemplos

En las figuras 3-16 y 3-17 incluimos dos ejemplos representativos de ambos algoritmos. Nótese que se respeta la forma de la malla poligonal obtenida con *Marching Cubes* excepto en aquellas zonas donde tenemos que aumentar el número de polígonos para incrementar la calidad de la figura.

Igualmente puede observarse cómo se elimina el efecto vóxel al aplicar el postproceso, y en consecuencia, se obtiene una forma mucho más definida. Esto es debido a que el postproceso aumenta la densidad de polígonos en las zonas donde los cambios de gradiente se acentúan subdividiendo los polígonos que se encuentran en las mismas.

En la siguiente figura se observa que en las zonas donde el cambio de pendientes es mayor, el modelo introduce mayor densidad de polígonos (triángulo más pequeños, y más ajustados a la superficie). Obsérvese el cambio con respecto a *Marching Cubes*, donde la forma de la malla poligonal es más homogénea.

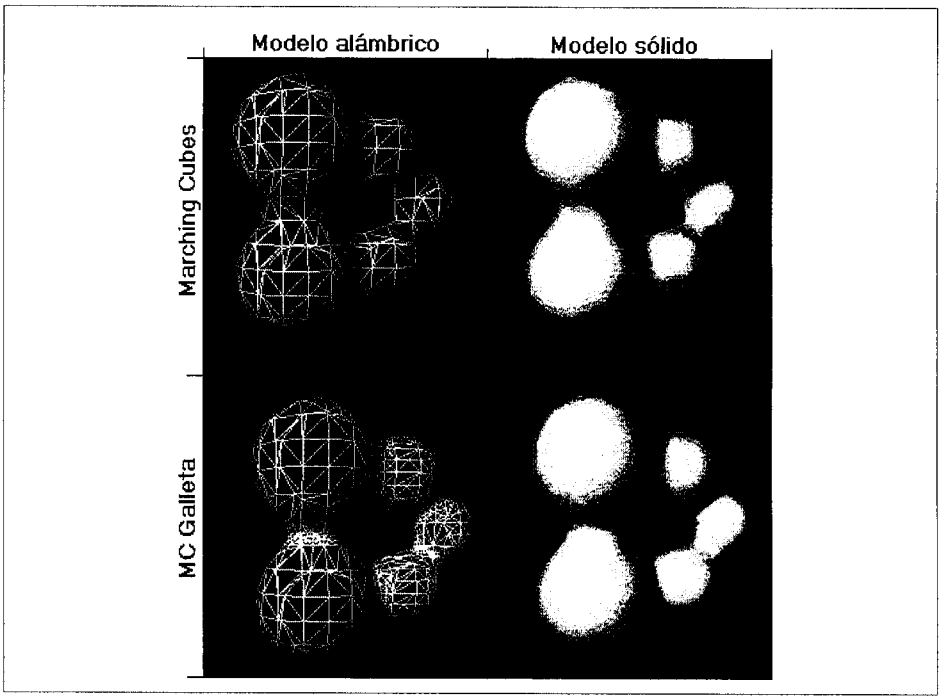


Figura 3-16: Comparación entre Marching Cubes y Rectificación de Precisión

Como en el ejemplo anterior, podemos observar un cambio apreciable, tanto en la distribución de los triángulos como en la densidad de los mismos. Este cambio se refleja en el modelo sólido haciendo una distribución más homogénea.

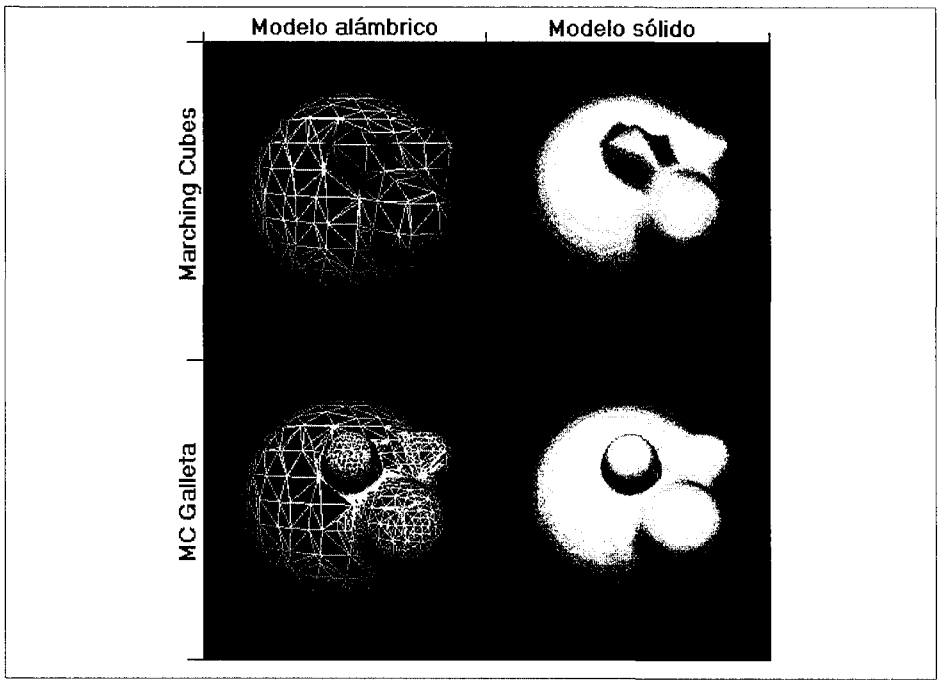


Figura 3-17: Comparación entre Marching Cubes y Rectificación de Precisión

3.2.4 Generalización del Método y Conclusiones

El método propuesto es aplicable a cualquier algoritmo de poligonalización de metaBlobs, puesto que se trata de un postproceso totalmente independiente del algoritmo original. Lo único que necesita el algoritmo es la malla poligonal resultante y la información de la superficie original. Por esta razón, el algoritmo no sería aplicable directamente a un objeto poligonal cualquiera sin la información de la superficie que lo generó.

Igualmente, el algoritmo se puede generalizar como postproceso de algoritmos de poligonalización de otros objetos orgánicos, además de los metaBlobs, que sigan la misma filosofía de construcción, esto es, que estén basados en superficies de campo finito y que dichas superficies sean derivables.

La razón por la que se han elegido los metaBlobs para el desarrollo de la técnica es que son altamente representativos de los objetos orgánicos. Por otro lado, el motivo por el que se ha elegido *Marching Cubes* es que se trata de uno de los algoritmos de menor coste computacional para la generación de mallas en isosuperficies. Esto nos ha permitido establecer comparativas sencillas entre el algoritmo simple y el algoritmo rectificado, sin entrar en otros problemas de coste que plantean otros algoritmos similares y que entorpecerían la labor de la estimación del coste del postproceso.

Como vemos el algoritmo presenta una solución al problema de la rectificación de mallas poligonales para metaBlobs. Ya que se trata de un postproceso, la calidad de la malla final dependerá en gran medida de la calidad de la malla obtenida con el algoritmo original. Como comentábamos, *Marching Cubes* deja huecos en la superficie del sólido debido a incoherencias poligonales. Dichos huecos no se solucionan con el postproceso, puesto que se trata de ausencias de polígonos. Es por ello que este algoritmo únicamente funcionará cuando partamos de un sólido correcto, esto es, una malla que envuelva al sólido que no contenga agujeros ni malformaciones poligonales.

Debido a los problemas que nos surgieron con este algoritmo, hemos continuado la investigación sobre la generación de mallas para metaBlobs siguiendo por la misma idea de la rectificación, pero con un algoritmo de generación de mallas poligonales propio que permita la integración de la rectificación *in-situ*. Es el caso del algoritmo que planteamos en el siguiente apartado.

3.3 Recorrido de la Superficie mediante Gradiente (RSG)

En esta sección vamos a describir un algoritmo que permite pasar de una representación de ***metaBlobs***, con independencia de la función de potencial de energía empleada, a una representación mediante un modelo de fronteras, en particular a un modelo *poligonal*. Este modelo poligonal lo vamos a definir en base a vértices y polígonos, y de forma opcional se pueden obtener las aristas del modelo. Es un modelo muy conveniente al ser un modelo *completo*.

Este algoritmo se basa en recorrer la superficie del metaBlob empleando la búsqueda de puntos cercanos, para realizar dicha búsqueda se emplea el gradiente de la función de campo. Este algoritmo lo llamaremos RSG, recorrido de la superficie mediante gradiente.

El algoritmo permite ajustar el nivel de detalle del modelo poligonal que se va a generar a partir del modelo metaBlob, para ello definiremos una serie de parámetros que iremos introduciendo a medida que describamos el algoritmo, éste se resume en cinco pasos:

Conversión Poligonal Paramétrica (CP^2) en 6 pasos

1) Preproceso.
2) Inicialización.
3) Obtener una sección.
4) Buscar la siguiente sección.
5) Triangulación.
6) Simplificación. Si quedan secciones ir al paso 4.

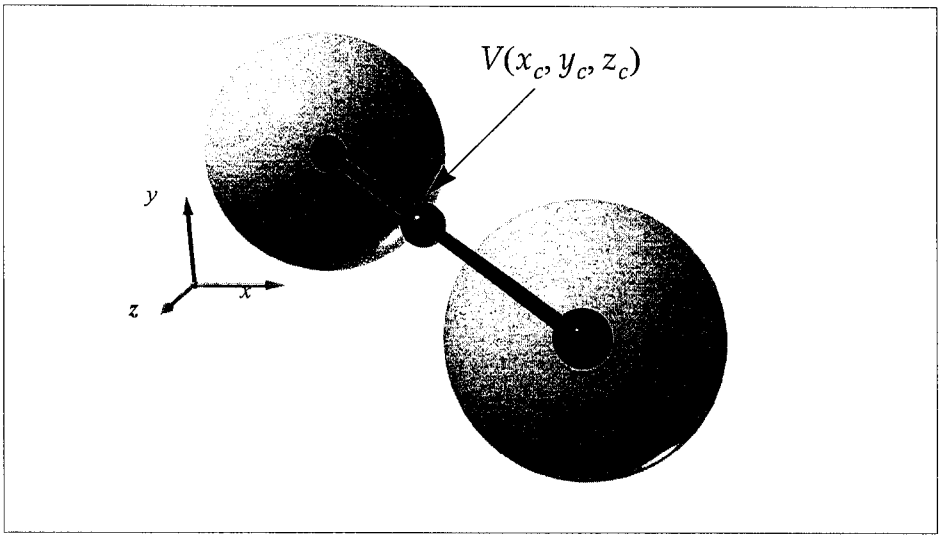
3.3.1 Algoritmo para Conversión a Malla Poligonal

1. Preproceso

La fase de preproceso consiste en realizar una clasificación en clases de equivalencia, de forma que cada uno contenga grupos de

primitivas conexas entre sí. (en el capítulo 4 se detalla este preproceso). Para ello se evalúa la función de densidad, ecuación (2.7), en el centro entre cada dos primitivas. Cogiendo cada primitiva de la lista y analizándola con el resto, llegamos a un orden $O(n) = n \cdot log\, n$, siendo n el número de primitivas.

Figura 3-18: Cálculo de las conectividad entre dos primitivas



2. Inicialización

En primer lugar buscamos el prisma en el cual se encuentre circunscrito el volumen del metaBlob. Dicho prisma será del menor tamaño posible y con la orientación adecuada para minimizar el espacio de búsqueda.

Una vez descrito el espacio de búsqueda, empleamos la dirección principal del volumen para el plano de corte C_i. Buscaremos las distintas secciones a lo largo del eje z.

Al punto inicial lo llamaremos P_0^B, y a partir de él emplearemos como eje de recorrido la dirección principal de orientación del prisma obtenido.

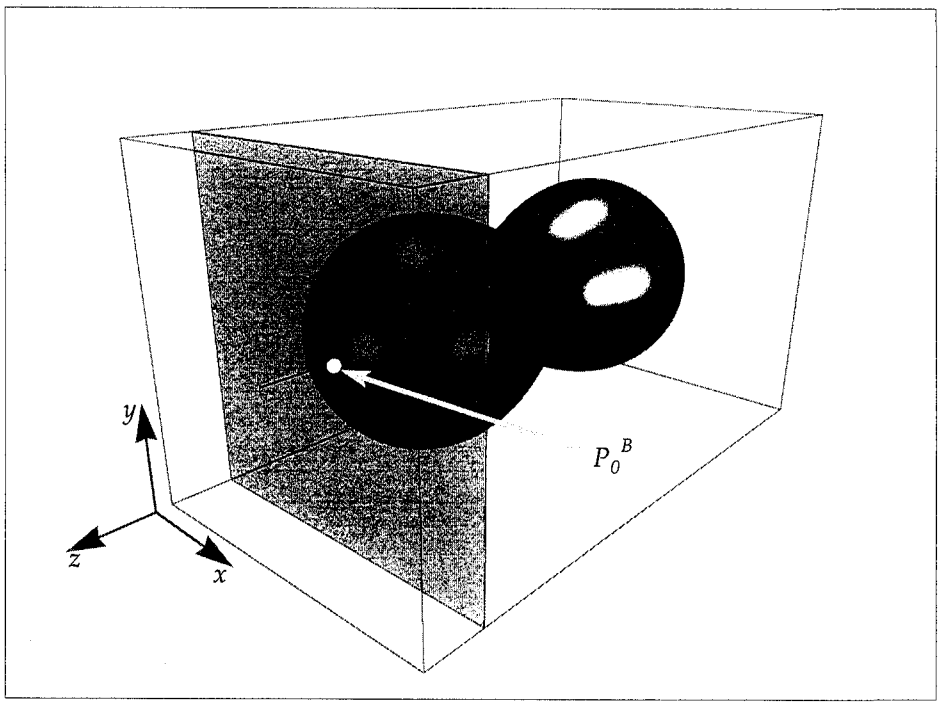


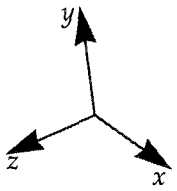
Figura 3-19: Prisma que rodea al metaBlob y eje principal

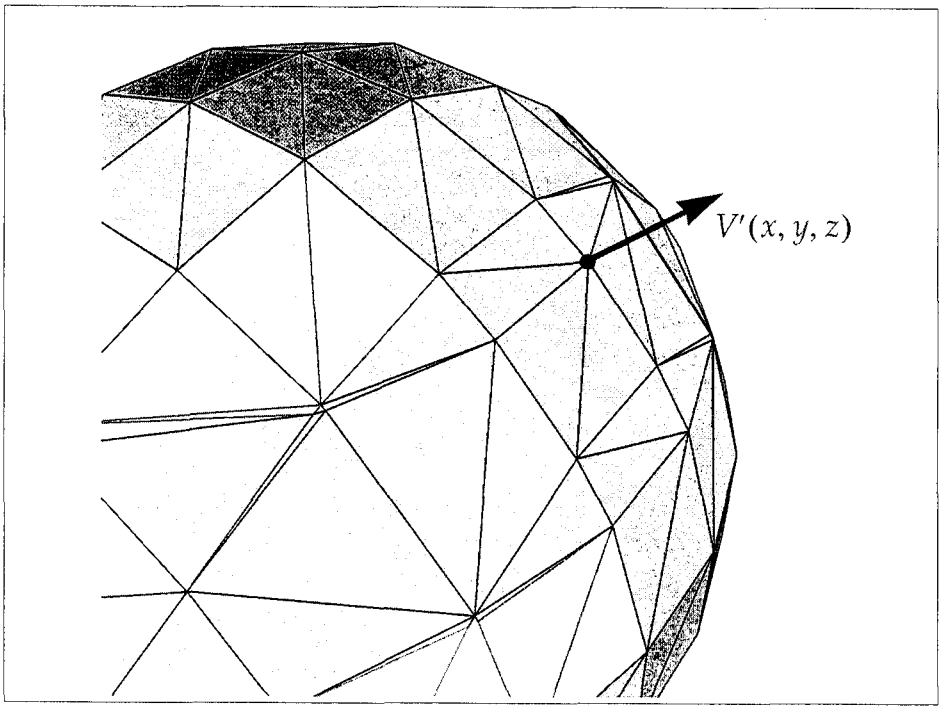
La derivada de la función V, ecuación (2.5) se corresponde con la siguiente expresión:

$$V'(x,y,z) = \sum_{i=1}^{N} b_i \frac{-a_i f' x,y,z}{e^{-a_i f x,y,z}} \qquad \textbf{(3.2)}$$

Mediante el gradiente obtenemos la normal del plano que es tangente a la isosuperficie en el punto (x, y, z), con dicha normal y el propio punto podemos obtener la ecuación del plano.

Figura 3-20:
Gradiente en un punto de la isosuperficie



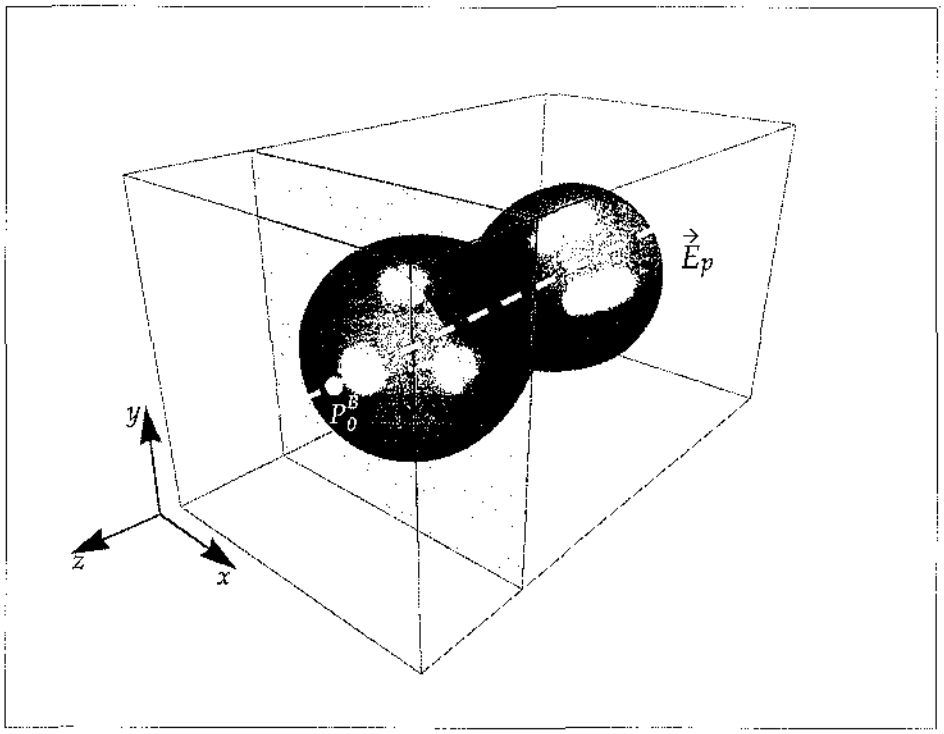


$$GradV(x,y,z) = \frac{\partial V(x,y,z)}{\partial x}\vec{i} + \frac{\partial V(x,y,z)}{\partial y}\vec{j} + \frac{\partial V(x,y,z)}{\partial z}\vec{k} \quad \textbf{(3.3)}$$

La función V y su derivada V' tienen una formulación muy similar, siendo similar el número de operaciones matemáticas para resolverlas, teniendo, por tanto, el mismo tiempo de computación para realizar su resolución.

Situación del eje principal $\vec{E}_p$ desde el punto P_0^B perpendicular al plano ***xy***. Se definirá el eje principal con la obtención de otro punto que llamaremos P_f^B. El eje principal será denotado por $\vec{E}_p$

Figura 3-21: Representación del eje con la dirección principal del metaBlob



3. Obtención de una Sección o Contorno individual

A partir del punto inicial se buscará el comienzo de toda una serie de secciones, cada uno de estos puntos tendrá un vector gradiente $\vec{g}$ cuyo valor aparece expresado por la ecuación (3.4). Los comienzos de cada contorno individual se buscan de izquierda a derecha. Cada sección se encuentra en el plano C_i, cuya normal es el vector del eje principal $\vec{E}_p$.

$$\vec{g} = -\vec{j} \tag{3.4}$$

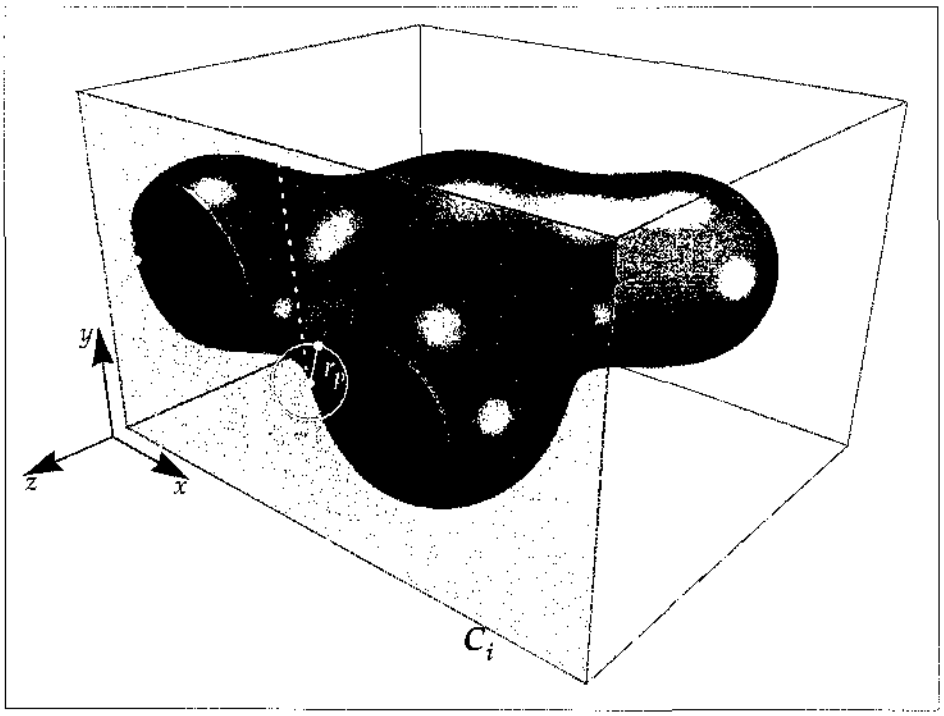
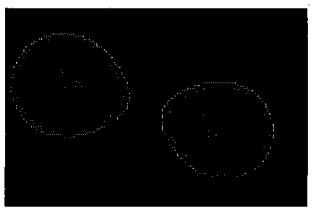


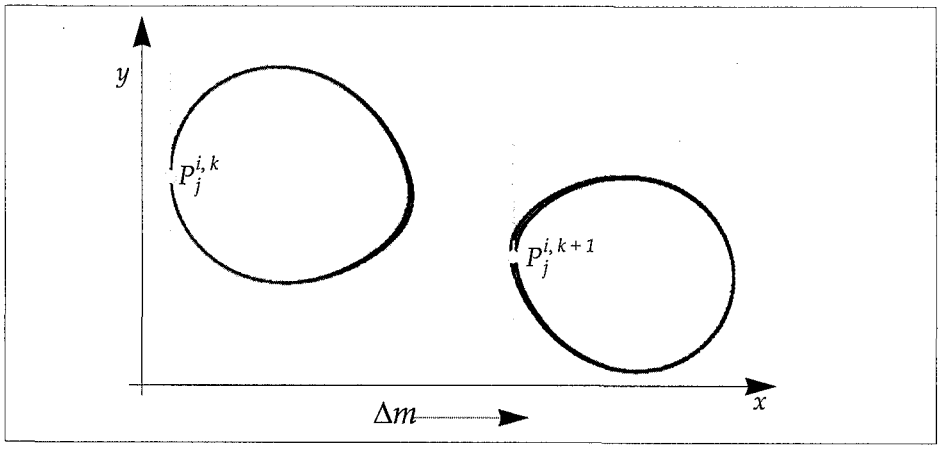
Figura 3-22: Plano de corte para buscar los puntos correspondientes a una sección

Una vez obtenido un punto de referencia $P_j^{i,k}(x_j^{i,k}, y_j^{i,k}, z_j^{i,k})$, siendo i la sección y k el número de contorno, se busca el siguiente punto $P_{j+1}^{i,k}$ dentro de un entorno de proximidad a un radio r_p del punto de referencia $P_j^{i,k}$ (figura 3-24) y cuya diferencia de pendiente respecto el uno del otro difiera en un ángulo menor a α. Para calcular la pendiente empleamos V' dentro del plano xy. Para averiguar la dirección correcta se emplea la normal al gradiente, es decir, el plano tangente al punto de referencia.

3 Poligonalización de la Superficie

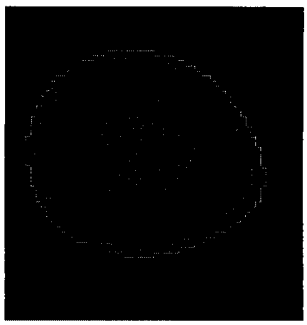
Figura 3-23: Varios puntos de referencia correspondientes a varios contornos

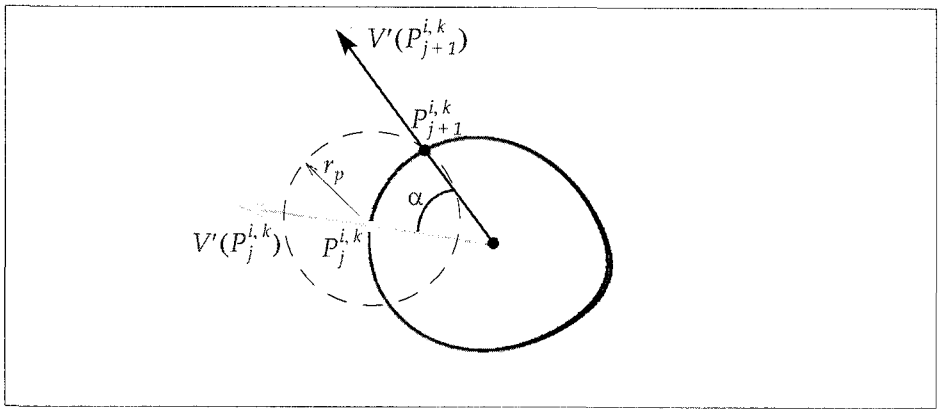




La siguiente figura representa un esquema de la búsqueda del punto $P_{j+1}^{i,k}$ a partir del punto de referencia inicial $P_j^{i,k}$.

Figura 3-24: Esquema de la búsqueda de un punto a partir del anterior



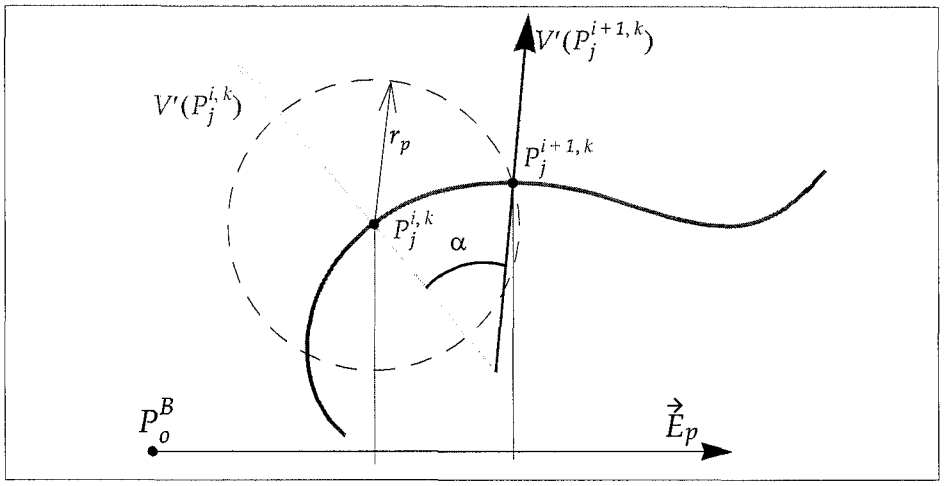


Si la diferencia de pendientes Δm entre ambos puntos supera un valor de referencia o umbral pre-establecido U_α (bien sea de forma absoluta o relativa) entonces se decrementa el valor del ángulo α para volver a calcular de nuevo el punto $P_{j+1}^{i,k}$, realizándose este proceso hasta que la diferencia de pendientes Δm sea menor que el valor de referencia o umbral U_α. El punto se busca dentro de una zona con radio menor a r_p.

4. Buscar la siguiente sección

En esta fase se busca el siguiente punto de referencia $P_j^{i+1,k}$.

Figura 3-25: Esquema de la búsqueda del siguiente punto inicial de contorno



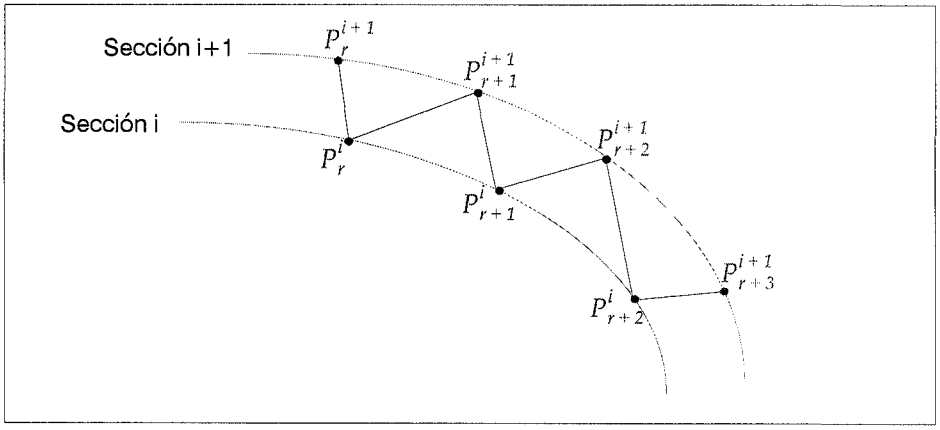
Aquí se emplea el mismo esquema que aplicamos en la fase anterior. Si la diferencia de pendientes Δm entre ambos puntos supera un valor de referencia o umbral pre-establecido (bien sea de forma absoluta o relativa) entonces se decrementa el valor del ángulo U_α para volver a calcular de nuevo el punto $P_j^{i+1,k}$, realizándose este proceso hasta que la diferencia de pendientes sea menor Δm que el valor de referencia o umbral U_α. El punto se busca dentro de la zona con radio menor a r_p.

5. Triangulación

Una vez obtenidas dos secciones consecutivas se realiza el proceso de actualizar la lista de polígonos que define el metaBlob resultante añadiendo los polígonos que cierran ambas secciones.

3 Poligonalización de la Superficie

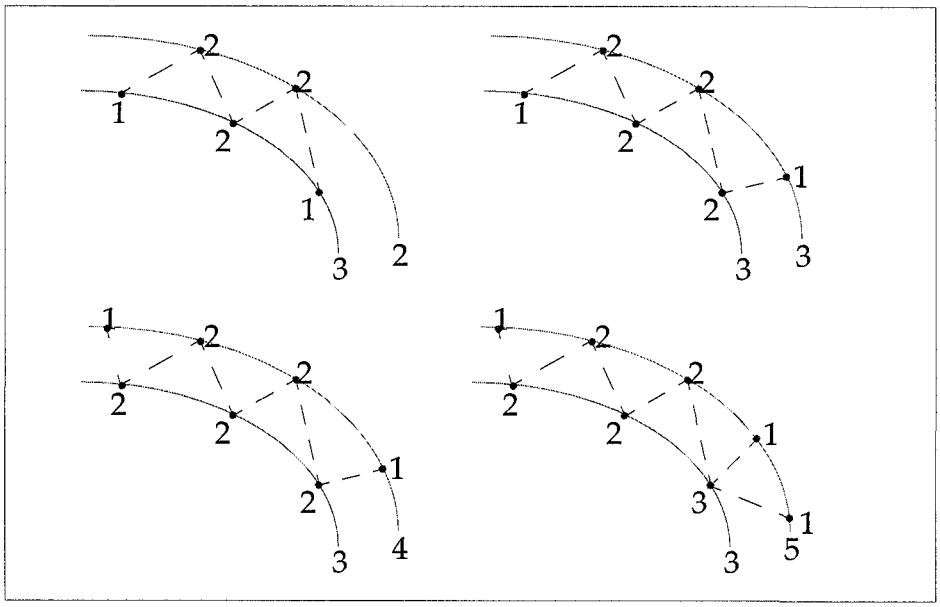
Figura 3-26: Triangulación del modelo metaBlob uniendo la sección i con la i+1



Tenemos tres posibilidades:

a) que la sección *i+1* tenga más puntos que la sección *i*
b) que la sección *i+1* tenga menos puntos que la sección *i*
c) que ambas secciones tengan el mismo número de puntos

Figura 3-27: Triangulación. Casos en la unión de dos contornos



Los números debajo de los arcos indican el número de puntos de cada sección, mientras que los que se encuentran en los puntos indican el número de aristas que surgen de cada uno de los puntos.

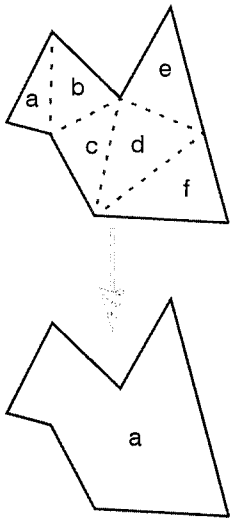


Figura 3-28: Simplificación de la malla poligonal

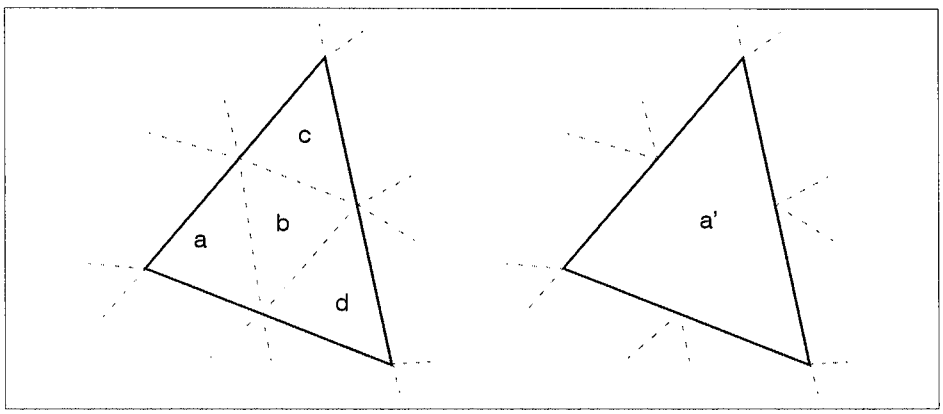
6. Simplificación

Como postproceso se pueden aplicar algoritmos de simplificación de malla poligonal. Básicamente podemos emplear uno de estos dos métodos:

a) Eliminando triángulos, agrupando aquellos que sean coplanares, aproximando la coplanariedad mediante un umbral (*threshold coplanar*) sin mantener la restricción de polígonos triangulares.

.

b) Manteniendo la triangularidad (para aprovechar modelos poligonales optimizados, con aceleración hardware simplificación del modelo, etc...). Restricción conjunta coplanar y colineal.

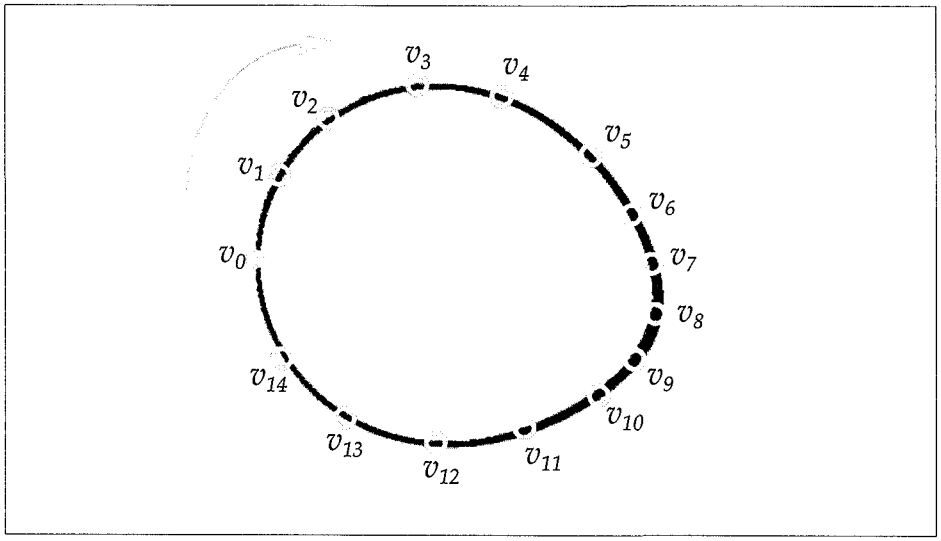


3.3.2 Representación Geométrica.

Una de las ventajas de este algoritmo es la obtención de un contorno ordenado de vértices, dicha ordenación hace posible el

empleo de una representación geométrica de la malla poligonal más compacta que la empleada por *Marching Cubes*.

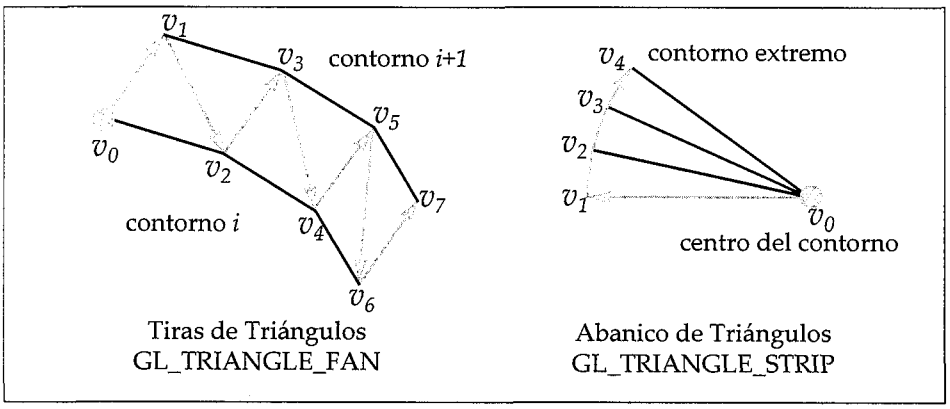
Figura 3-29: Obtención de vértices ordenados de un contorno



Una vez obtenido el conjunto de vértices de un contorno, lo unimos con el siguiente contorno empleando alguna de las estructuras geométricas optimizadas, figura 3-30.

En *Marching Cubes* y otros algoritmos de triangulación, se emplean triángulos independientes para construir la malla poligonal. Esto repercute en un mayor tiempo de computación y en la generación de hasta el triple de información geométrica para construir el mismo modelo que se obtiene con respecto al algoritmo RSG.

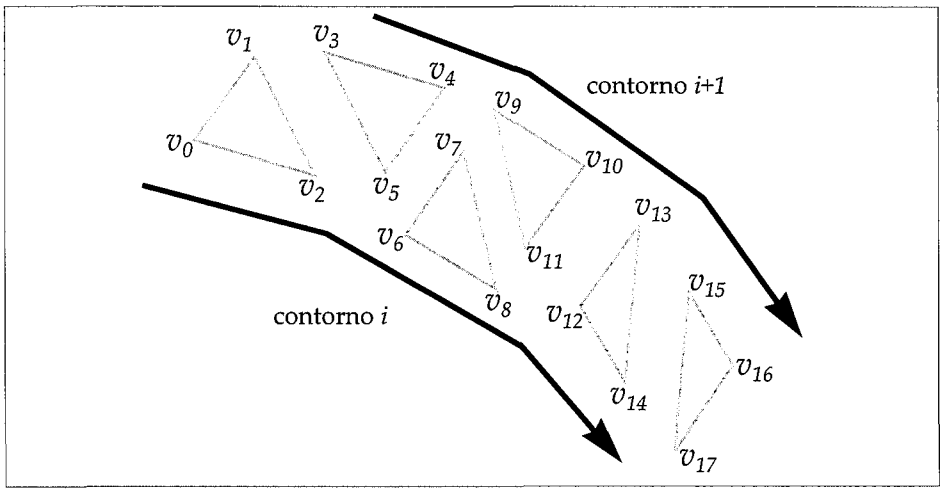
Figura 3-30: Tiras y abanicos de triángulos



[1] Tiempo real entendido en el ámbito de los Gráficos por Computador

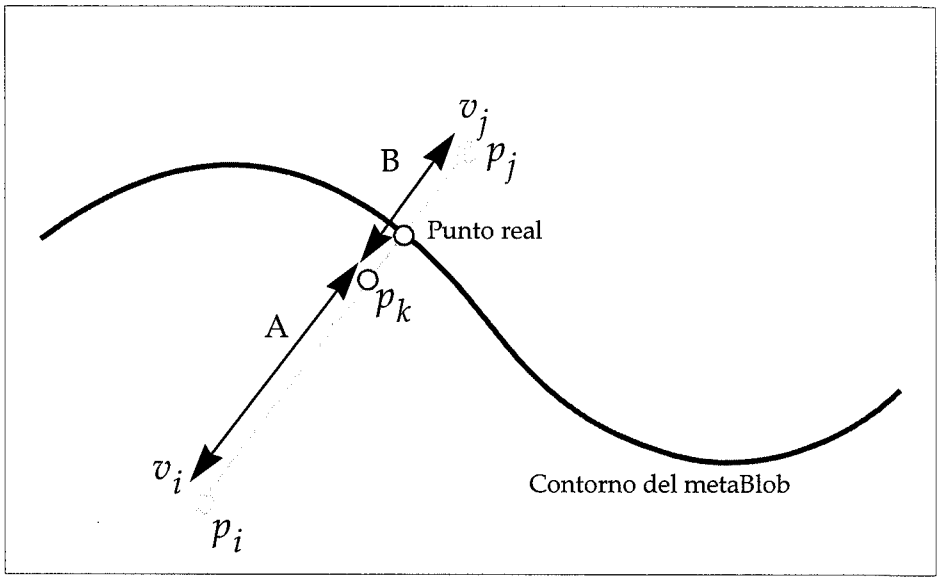
El mayor volumen de información generada se traduce en varios aspectos negativos, mayor tiempo de construcción de la malla poligonal y mayor tiempo de visualización, normalmente realizado por la tarjeta gráfica 3D, que usualmente emplea una aceleración por hardware basada en triángulos, que por muy rápido que vaya, tardará al menos el triple de tiempo en visualizar el objeto por pantalla, teniendo en cuenta que para considerarlo en *tiempo real*[1] se debe realizar, la actualización de la imagen en la pantalla, al menos sobre unas 10-20 veces por segundo.

Figura 3-31: Triángulos independientes



Otro aspecto importante ha sido el cálculo de los puntos de intersección de la superficie con una arista concreta. En la siguiente figura ilustramos el cálculo de dicho punto.

Figura 3-32: Interpolación entre dos puntos del campo



Supongamos que p_i y p_j son los puntos que forman la arista, y p_i se encuentra en el interior del volumen y p_j en el exterior. También conocemos el valor del campo escalar en cada punto, dichos valores son v_i y v_j, para cada uno de los puntos. Tenemos varias opciones, calcular el valor mediante interpolación lineal o mediante aproximaciones sucesivas.

La interpolación lineal se realiza de la siguiente forma:

$$p_k = p_i - \frac{v_i}{(v_j - v_i)} \cdot (p_j - p_i) \qquad \textbf{(3.5)}$$

Sin embargo, mediante aproximaciones sucesivas se calcula el punto medio y nos quedamos con el segmento correcto, siendo el segmento A si p_k está en el exterior del volumen o B si el punto p_k se encuentra en el interior del volumen. Este proceso es recursivo

hasta que el punto p_k tenga un valor de campo lo suficientemente cercano a cero ($<\varepsilon$), o se realiza un cierto número de veces, cota máxima de iteraciones posibles que permitimos realizar en nuestra implementación.

En nuestra implementación hemos aplicado una mezcla de ambas, en vez de calcular el punto medio, vamos interpolando, pero, si el punto obtenido no se aproxima lo suficiente, seleccionamos el segmento adecuado de la arista y volvemos a realizar otra interpolación. La ventaja que obtenemos realizando así el cálculo es que la solución converge más rápidamente a la solución deseada, al incorporar también la información del campo escalar en la realización de la interpolación lineal, que si meramente calculamos el punto medio. Por otro lado esta interpolación lineal se puede realizar si los dos puntos p_i y p_j se encuentran relativamente cercanos.

Toda la experimentación y mejoras que aporta este algortimo se describen en el capítulo 5.

Volver al índice/Tornar a l'índex

3.4 Conclusiones

En este capítulo hemos desarrollado, en primer lugar, una técnica para calcular la adaptación de la malla poligonal lo más cercana posible a la forma real de la superficie, para conseguirlo, hemos hecho uso de la derivada de la función matemática que define la forma de un metaBlob, el gradiente. Como medida de refinamiento hemos medido el ángulo de los gradientes, que es el vector normal a la superficie en un punto, entre dos puntos para saber si el punto lo aceptamos o buscamos uno cuyo ángulo difiera en un valor menor. Dicha rectificación la hemos integrado con el algoritmo de triangulación de *Delaunay*, aunque también hemos realizado una variante adaptándola a *Marching Cubes* con resultados igualmente satisfactorios.

Por otro lado hemos desarrollado un algoritmo de recorrido por la superficie empleando una búsqueda local de puntos próximos y además exigiendo que se cumpla el criterio de adaptabilidad a la superficie, es decir que el ángulo presente una diferencia menor que un cierto ángulo umbral, también empleando el gradiente. El resultado ha sido un algoritmo mucho más rápido que el algoritmo de *Marching Cubes*. Esta mejora de rendimiento se comprueba experimentalmente en el capítulo 5.

Refinamientos de la Poligonalización

4

Una vez explicados los algoritmos que hemos desarrollado, en este capítulo ofrecemos una serie de refinamientos en la conversión poligonal del modelo metaBlob. Los dividiremos en dos categorías: reducción del espacio de búsqueda, mediante el cálculo de las componentes conexas y la optimización de cada volumen de forma que sea mínimo en tamaño y buscando su orientación ideal en el espacio, reduciendo así notablemente el espacio de búsqueda.

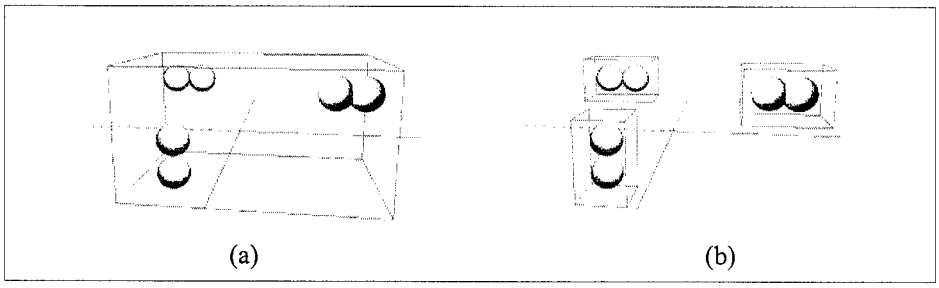
4 Refinamientos de la Poligonalización

4.1 Reducción del Espacio de Cálculo

Como ya se dijo en el capítulo anterior, nuestra idea es reducir el número de cálculos de la función de densidad (2.7) al mínimo. Para ello, en los algoritmos de poligonalización se discretiza el espacio de cálculo, una mayor separación en la discretización de los puntos del espacio 3D reduce el número de cálculos de forma cúbica, figuras **5-2** y **5-3**, pero, al mismo tiempo, produce una peor aproximación a la representación del modelo metaBlob. Nuestro objetivo es conseguir la misma precisión en nuestra representación pero eliminando los cálculos innecesarios. Para ello buscaremos el volumen mínimo que contenga el metaBlob.

Todos los algoritmos que hemos estudiado hasta ahora comienzan con el cálculo del volumen en el cual se encuentra el volumen sólido que queremos representar. Es decir, calculamos el paralelepípedo que vamos a emplear como espacio 3D a discretizar. Nuestro objetivo será, por tanto, minimizar dicho volumen.

Figura 4-1:
Volúmenes óptimos



Abordaremos dos ideas distintas, la primera será agrupar las primitivas en componentes conexas que creen modelos metaBlob independientes. Por otro lado, para cada una de esas componentes conexas, buscaremos el paralelepípedo que se oriente de forma óptima en cada componente conexa de forma independiente, y de manera que el volumen sea el menor posible.

En la figura 4-1 se observa: (a) un paralelepípedo que engloba a la totalidad de las primitivas, pero que no corresponde a un volumen

mínimo, puesto que en realidad la forma global del metaBlob se puede descomponer en tres componentes conexas que representan el mismo volumen que si tratamos las primitivas por separado y, (b) muestra los paralelepípedos óptimos para generar los metaBlobs resultantes.

4.1.1 Componentes Conexas

Nuestro primer paso será calcular las componentes conexas de nuestro modelo metaBlob. En general no se ha empleado esta técnica en ningún algoritmo de conversión de metaBlobs a modelo poligonal debido a que no es una técnica trivial. Si consideramos que queremos reducir el número de cálculos de nuestra función de densidad, y que para comprobar que dos primitivas están unidas hemos de evaluar la función de densidad lo suficiente para asegurar que no incurrimos en ningún error, podemos caer en una contradicción, de forma que el número de cálculos de la función de densidad supere a los que nos queremos ahorrar, siendo inútil, por tanto esta técnica.

En primer lugar veamos una forma consistente y rápida para comprobar que dos primitivas forman parte de una componente conexa, es decir, que pertenecen al mismo modelo metaBlob, y por tanto no pueden separarse.

Podríamos pensar que con evaluar la posición central del metaBlob, podemos rápidamente saber si están unidos o no, pero esto presenta una serie de problemas, debido a que la fuerza de atracción de una primitiva puede ser positiva o negativa, y precisamente en la posición central la función de densidad nos puede dar un valor negativo (que no pertenece al volumen), mientras que dichas primitivas están conectadas.

Sin embargo, lo que sí podemos construir es una función que fácilmente nos diga si dos primitivas no están conectadas. Si observamos la función de campo que una primitiva genera a su alrededor veremos que la frontera se sitúa en un valor entre una

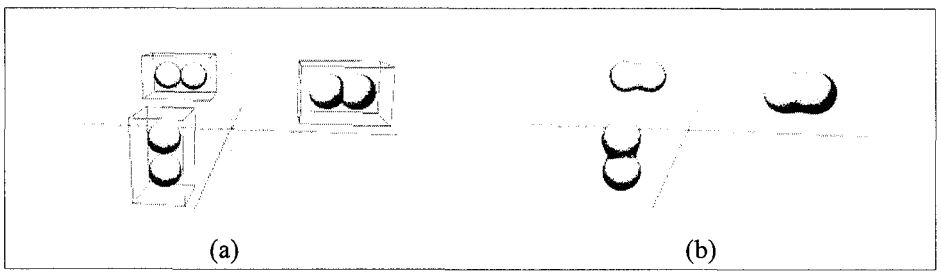
distancia mayor que el radio de la primitiva y menor que tres veces dicho valor.

4.1.2 Cálculo de volúmenes Conexos

Como hemos visto en las propiedades de los metaBlobs, el campo que genera una primitiva se encuentra acotado. Derivado de esta propiedad, las primitivas de un metaBlob no tienen por qué presentar una conectividad total, o sea, un mismo metaBlob se puede presentar mediante varias superficies separadas *(véase el ejemplo de la figura 4-1)*. El proceso de identificación de volúmenes conexos consiste en determinar los subconjuntos de primitivas que se encuentran conectados y que, por tanto, forman una superficie aislada y los que no.

La posibilidad de identificar los grupos de primitivas que forman una superficie aislada nos da la ventaja de que cada superficie puede ser tratada como un metaBlob distinto. Este procedimiento minimiza considerablemente el espacio de búsqueda, puesto que para cada subconjunto se calcula su propio subespacio de búsqueda sin tener en cuenta el resto. En la figura 4-2 podemos observar las primitivas del ejemplo de la figura 4-1 separadas en grupos conectados.

Figura 4-2: (a) Tres grupos conexos, con sus 6 primitivas y (b) sus tres metaBlobs asociados



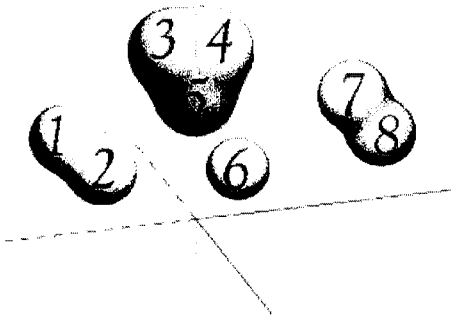
Como vemos en dichas figuras, el campo generado es el mismo. Ahora el problema es que la identificación de volúmenes conexos no ha sido tratada hasta ahora. Esto es debido a que, pese a parecer un proceso trivial, nada más lejos de ello. Asegurar si dos

primitivas están o no conectadas es un proceso costoso que puede tratarse desde varios puntos de vista.

Una forma de hacerlo consistiría en rastrear todo el volumen con el algoritmo de Marching Cubes y determinar un postproceso sobre la malla obtenida que nos proporcionase un conjunto de polígonos vecinos. A partir de dichos grupos se formarían los volúmenes separados de la figura. El problema es que, como hemos visto, este algoritmo presenta irregularidades con la perseverancia del volumen, y esto es justamente lo que queremos evitar. Por otro lado, si lo que queremos es reducir el tiempo de cálculo, realizando este proceso, lo incrementaríamos. La solución que proponemos es una aproximación al problema que, como veremos, es mucho más sencilla y no afecta al tiempo de cálculo del algoritmo.

Definimos la función $Conectadas_{i,j}$ que nos dirá si las primitivas i y j están conectadas o no. A partir de esta función, formamos una matriz cuadrada NxN, siendo N el número de primitivas. Cada casilla de la matriz *(i,j)*, contendrá el valor 1 si la primitiva i está conectada con la j, o el valor 0 en caso contrario. Como es evidente, la diagonal principal de la matriz contendrá en todas sus casillas el valor 1 y la matriz será simétrica con respecto a dicha diagonal. En la figura 4-3 incluimos un ejemplo de la formación de esta matriz.

Figura 4-3: metaBlob formado por ocho primitivas y su matriz de conectividad



	1	2	3	4	5	6	7	8
1	1	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0
3	0	0	1	1	1	0	0	0
4	0	0	1	1	1	0	0	0
5	0	0	1	1	1	0	0	0
6	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	1	1

Con la matriz de conectividad de las primitivas podemos construir

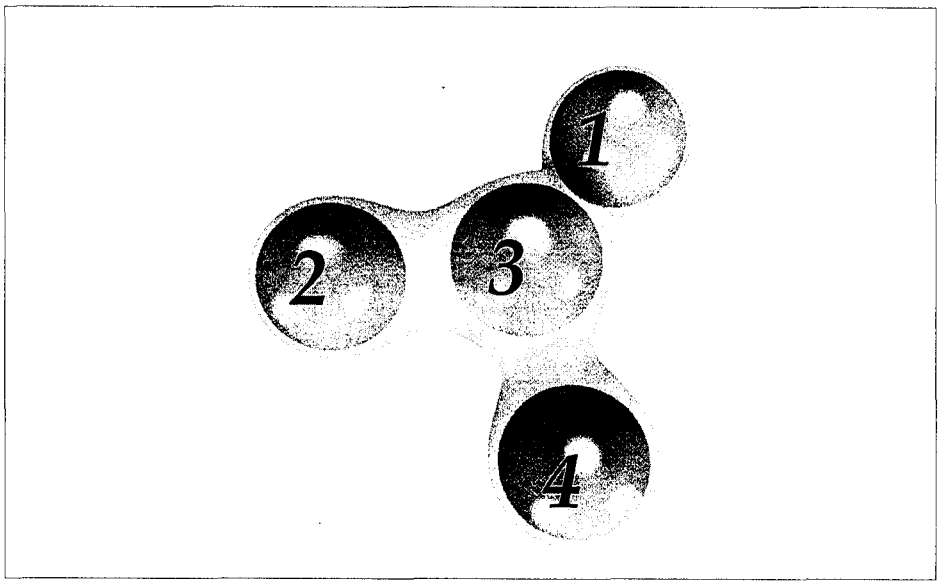
la función ***GrupoConectado*** que nos devolverá el conjunto de primitivas conectadas a una en particular. Esta función se define de la siguiente manera:

$$GrupoConectado_i = \{j \cup GrupoConectado_j | M_{Conectividad}(i,j)\} \quad \textbf{(4.1)}$$

El grupo conectado a *i* se define como las primitivas conectadas directamente a *i* más los grupos conectados a dichas primitivas. Como vemos, se trata de una definición recursiva. Puede parecer que el grupo podría construirse con las primitivas que son directamente alcanzables por *i*, pero puede darse el caso de que dos primitivas, pese a no estar conectadas directamente, lo estén mediante otra. Esto puede observarse gráficamente en el ejemplo de la figura 4-4.

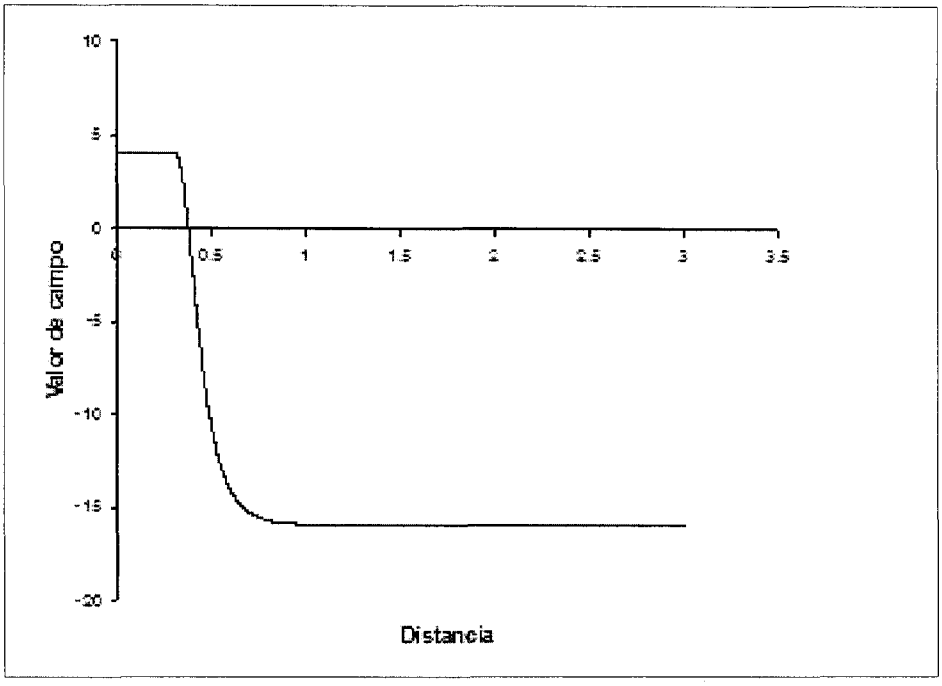
En el ejemplo de dicha figura, la primitiva *2* no está directamente conectada con las primitivas *1* y *4*. Como la primitiva *3* sí lo está con la *1*, *2* y *4*, por la definición recursiva que hemos dado, podemos decir que *GrupoConectado(2)={1,2,3,4}*.

Figura 4-4: Ejemplo de conectividad entre cuatro primitivas



Podemos comprobar que calcular el grupo conectado de una primitiva es sencillo. El problema aún sin resolver es cómo construir la función de conectividad directa entre primitivas dos a dos. Dicho problema no lo podemos determinar de forma sencilla, pero sí podemos decir cuándo dos primitivas *no* están conectadas. Para ello, vamos a detenernos en la gráfica de la figura 4-5.

Figura 4-5: Valor de campo que genera una primitiva a su alrededor



En la figura anterior podemos ver que la frontera de campo de la primitiva (valor de V=0) se encuentra a una distancia entre R (radio de la primitiva) y 3 R. Para determinar el valor exacto de dicha distancia se tiene que igualar la fórmula del campo a 0 y despejar d. El problema es que, como hemos visto, la fórmula de campo no es lineal y por tanto es prácticamente imposible despejarla. Es por ello que tenemos que establecer una cota máxima entre R y 3R que nos asegure que el valor de campo 0 se encuentra por debajo de la misma. Experimentalmente hemos establecido que el valor más aconsejable es 1.5 R. Podríamos haber utilizado directamente 3 R

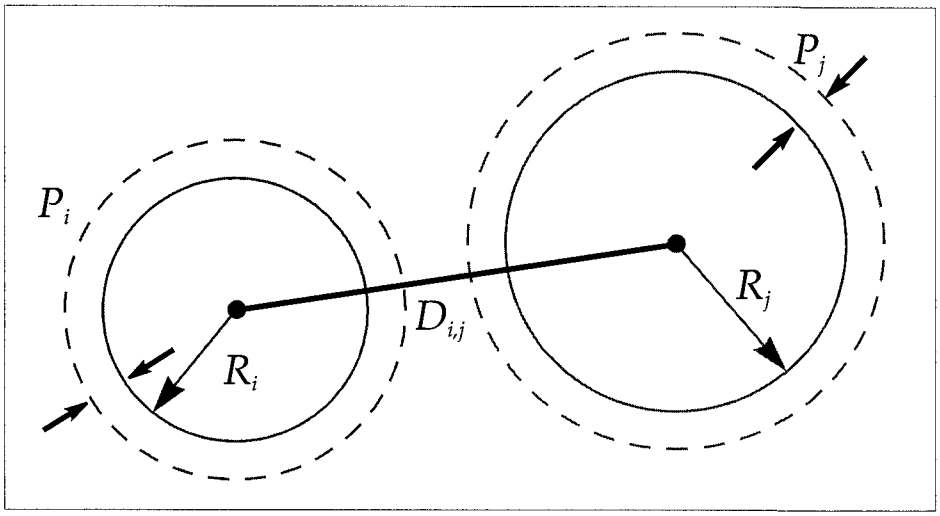
pero es una cota bastante pesimista.

Una vez establecida la cota superior de campo para una primitiva, podemos decir que dos primitivas (i,j) estarán separadas cuando se cumpla:

$$S_{i,j} = \left(D_{i,j} - \frac{R_i + R_j}{2} > 0\right) \quad \textbf{(4.2)}$$

Siendo $D_{i,j}$ la función distancia entre las primitivas i y j.

Figura 4-6: Cálculo de la conectividad entre dos primitivas.



La conectividad puede considerarse como si asumiésemos que el radio de cada primitiva es 3/2 veces el radio original. Si las primitivas con ese nuevo radio no intersectan, podemos asumir que no están conectadas.

El criterio nos marca cuándo no están conectadas. En realidad, no nos hace falta más, puesto que, con esto, mediante la función *GrupoConectado* citada anteriormente, se puede determinar un conjunto de grupos que no están conectados entre sí. Las primitivas que forman dichos grupos no puede asegurarse que estén conectadas, pero podemos asumirlo con un cierto error. Dicho error

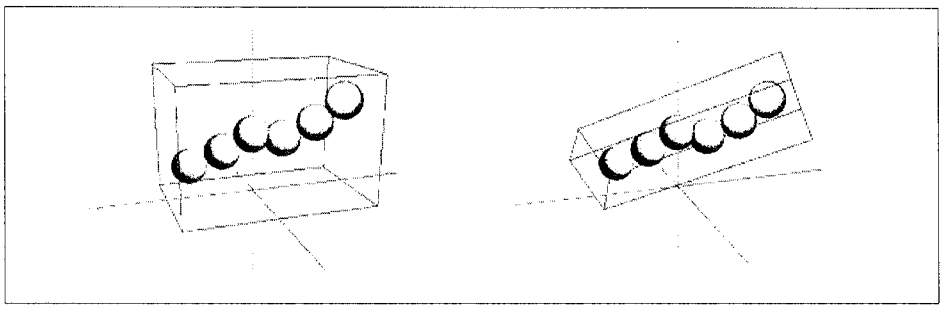
no será considerado, puesto que no nos afectará en la poligonalización.

Esta aproximación al problema de la conectividad nos proporciona un gran adelanto en la aceleración de algoritmos de poligonalización basados en vóxels puesto que, como hemos visto, reduce drásticamente el espacio de búsqueda cuando las primitivas se encuentran dispersas en grupos aislados.

4.1.3 Colocación Óptima del paralelepípedo Inicial

Como hemos dicho, la reducción del espacio de búsqueda (paralelepípedo inicial) que encierra al metaBlob es un factor importante a la hora de poligonalizarlo. Una vez segmentado el metaBlob en volúmenes aislados, es posible reducir aún mas dicho espacio de búsqueda colocando convenientemente el paralelípedo que encierra a cada grupo. De los infinitos paralelepípedos que podemos formar, nos quedaremos con el que mejor capte la "linealidad" de las primitivas. Para ello, orientaremos el paralelepípedo lo más alineado posible con respecto a las primitivas, dejando fuera todo el volumen sobrante que podamos. En la figura 4-7 tenemos un ejemplo del efecto de esta alineación.

Figura 4-7: Orientación del volumen que encierra a un metaBlob



Como es patente en la figura 4-7, cuando las primitivas del metaBlob presentan una cierta alineación espacial, la colocación del paralelepípedo inicial sobre dicha alineación reduce el volumen de rastreo considerablemente.

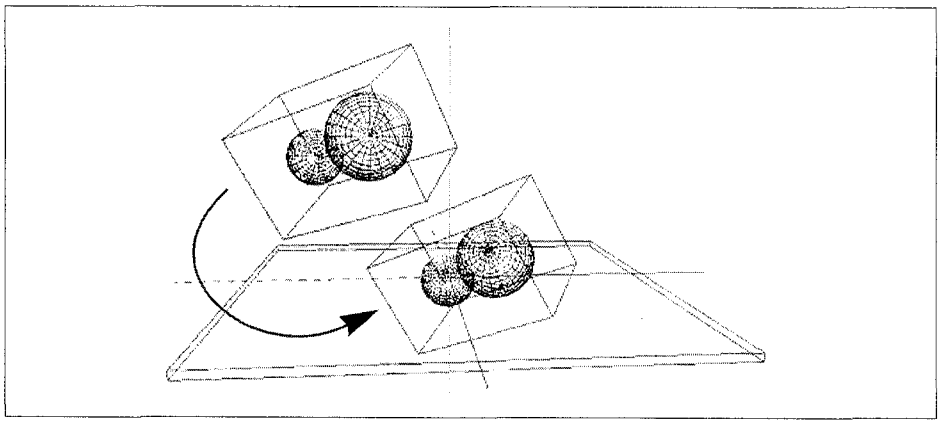
Para captar la linealidad del metaBlob hemos utilizado un cálculo aproximado de la recta de regresión de los centros de las primitivas. La recta de regresión de una nube de puntos se define como aquella recta cuya distancia media a cada uno de los puntos que forma la nube es mínima.

Dado que el cálculo de la recta de regresión de una nube de puntos es un proceso relativamente complicado, hemos realizado una aproximación que simplifica mucho el problema y nos proporciona el mismo efecto. El procedimiento se resume en los siguientes pasos:

a) Obtener la primitiva con la componente x más pequeña. Llamaremos a dicha primitiva P_0. Llamaremos T_0 a la traslación que hay que aplicar a P_0 para centrarla en el origen de coordenadas.

b) Aplicar la traslación T_0 a todas las primitivas. De esta forma, reducimos el cálculo de los ángulos a cuatro cuadrantes en vez de a ocho. Podemos ver un ejemplo de esta traslación en la figura 4-8.

Figura 4-8: Traslación de un metaBlob hasta el origen de coordenadas



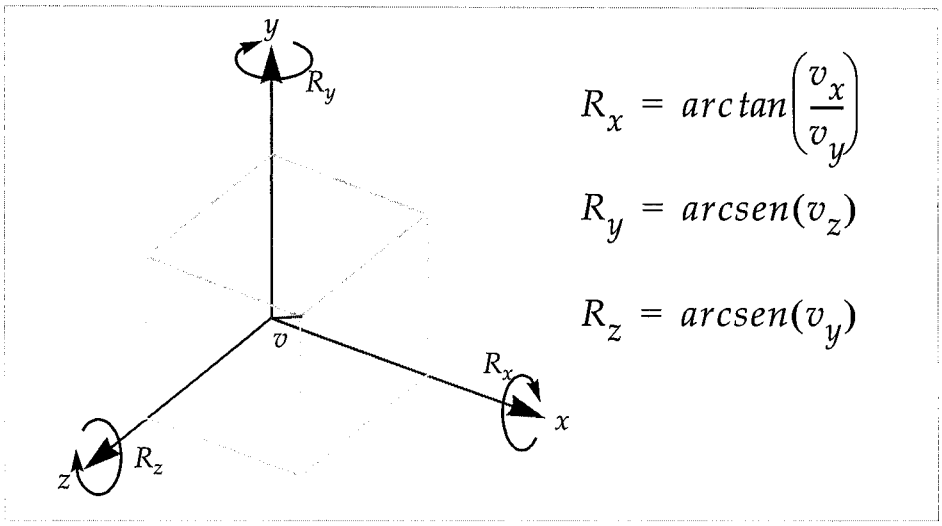
c) Calcular el vector unitario con dirección desde el centro de coordenadas al centro de cada primitiva (excepto para P_0).

d) Calcular los tres ángulos de rotación $[R_x, R_y, R_z]_i$ de cada vector

obtenido en (c) que hay que aplicar para transformarlo en el vector [1,0,0]. La forma de calcular dichos ángulos puede observarse en la figura.

Cálculo de los ángulos de rotación de un vector para transformarlo en el [1,0,0]

Figura 4-9: Cálculo de la media M de los vectores de rotación asociados a cada primitiva



$$M[R_x, R_y, R_z] = \frac{\left(\sum_{i=1}^{N} [R_x, R_y, R_z]\right)}{N-1} \qquad \textbf{(4.3)}$$

El vector media simboliza los ángulos que forman la recta de regresión con los ejes de coordenadas.

Aplicando la traslación $T_o(t_x, t_y, t_z)$ y las rotaciones de $M(r_x, r_y, r_z)$ a todas las primitivas, conseguimos alinear el metaBlob sobre el eje x. En la figura 4-10 presentamos un ejemplo del efecto de este cálculo.

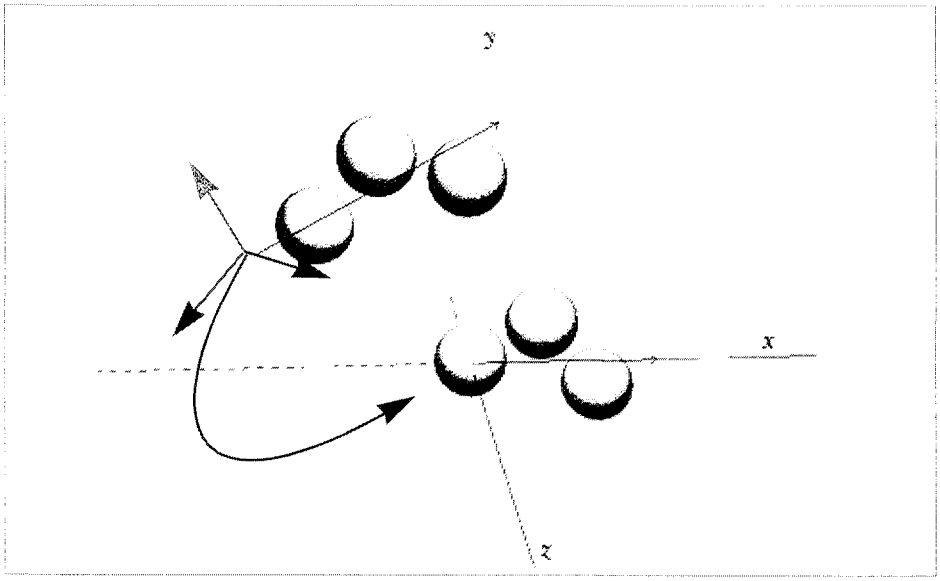


Figura 4-10: Alineado de un grupo de primitivas sobre el eje *x*

El proceso de orientar un grupo sobre el eje x puede considerarse un cambio de sistema de coordenadas. Lo que logramos con este procedimiento es hacer llevar la recta de regresión aproximada de los centros de las primitivas sobre el eje x.

Una vez alineado el grupo, se calcula su paralelepípedo inicial según se explicaba en el algoritmo de *Marching Cubes*. Para llevar el paralelepípedo a su posición original, se han de aplicar las transformaciones de forma inversa, esto es, aplicando primero las rotaciones de M invertidas $M(-r_x, -r_y, -r_z)$ y a continuación, la traslación invertida $T_o(-t_x, -t_y, -t_z)$ a los vértices de dicho paralelepípedo.

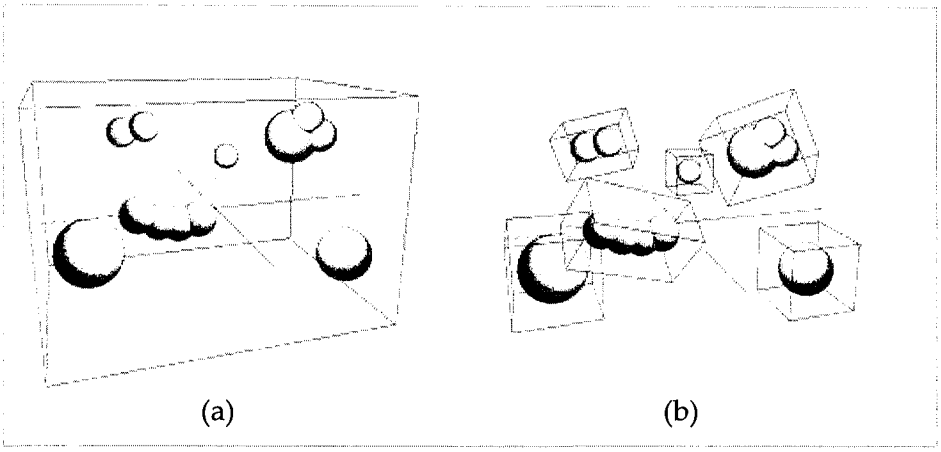
El coste de este algoritmo es lineal con respecto al número de primitivas. Teniendo en cuenta que dicho coste puede considerarse despreciable con respecto a cualquier tipo de poligonalización que se aplique, este preproceso es realmente rápido y puede aplicarse en tiempo real. De hecho, en la aplicación que se ha construido para el proyecto, el cálculo se realiza en tiempo real sobre la fase de

diseño.

4.1.4 Conclusiones

Segmentando el metaBlob en volúmenes conexos y aplicando la colocación óptima a cada uno, reducimos considerablemente el espacio de búsqueda. En la figura 4-11 aportamos un ejemplo de este procedimiento.

Figura 4-11: Cálculo y orientación de los volúmenes óptimos



(a) Cálculo del paralelepípedo inicial sin mejoras. (b) Cálculo de los distintos volúmenes conexos y colocación óptima de los mismos.

En el ejemplo, existe una gran diferencia de volumen entre el espacio de rastreo sin y con mejoras. Esta diferencia se hace notar directamente en el tiempo de cálculo de los distintos algoritmos de poligonalización. Los resultados de la poligonalización son idénticos (figura 4-12).

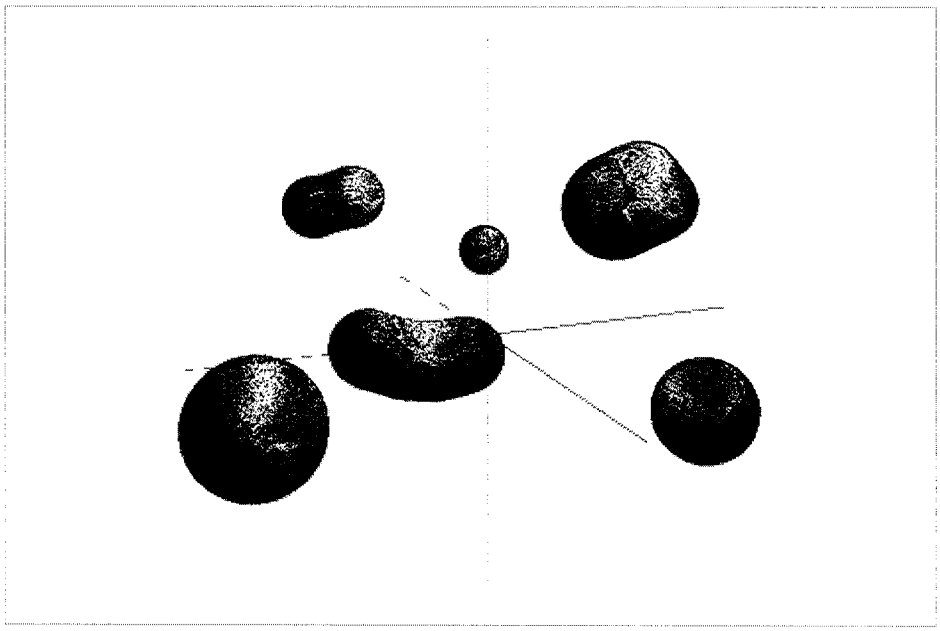


Figura 4-12: Resultado de la poligonalización del metaBlob de la figura 4-11

La única diferencia es que la malla de cada grupo presenta la misma alineación que la recta de regresión calculada para dicho grupo. En el caso del algoritmo con las mejoras, como se realiza la segmentación, cada grupo presenta una alineación distinta. Sin embargo, si no aplicamos las mejoras, como todo es un mismo grupo, toda la malla presenta la misma alineación (recta).

Estas mejoras serán efectivas cuando el metaBlob presente grupos de primitivas separados y, además, cada grupo presente una cierta alineación. Si todas las primitivas del metaBlob se encuentran conectadas y además no presenta ninguna alineación, no obtendremos ninguna mejora. De hecho, como las dos mejoras precisan de un cierto cálculo, podemos empeorar el tiempo de computación. Dicho incremento de tiempo resulta despreciable en comparación al tiempo de computación de cualquiera de los algoritmos de poligonalización mencionados en los capítulos anteriores.

Por estas razones, no hemos estimado conveniente realizar una gráfica comparativa de los algoritmos con y sin mejoras. Estas gráficas aparecerían dispersas, puesto que la mejora temporal depende del caso en particular y no del volumen inicial de rastreo ni de cualquier otro factor de medida del metaBlob. De todas formas, nuestra experiencia en diseño con metaBlobs nos confirma que, en la mayoría de los casos, suelen diseñarse formas separadas y que dichas formas suelen presentar una cierta linealidad. Por ejemplo, a la hora de representar un cuerpo humano, el diseñador suele construir tronco, cabeza y extremidades por separado. Además, tanto el tronco como las extremidades, presentan una clara alineación en la dirección de las primitivas que representan dichas formas.

Resumiendo un poco, hemos elaborado una técnica para subdividir un modelo metaBlob en varios subconjuntos de metaBlobs que están entre sí unidos, consiguiendo de esta forma generar un superficie independiente para cada subconjunto conexo de metaBlobs.

Una vez obtenida esa subdivisión en metaBlobs independientes, pasamos a obtener el volumen más pequeño que recubre al metaBlob, dicho volumen será mínimo si además lo orientamos según la recta de regresión que pasa por los centros de todas las primitivas que componen el metaBlob.

En definitiva, aportamos dos ideas principales, la primera es la clasificación del metaBlob en componentes conexas independientes unas de otras y, la segunda es la obtención y orientación del volumen mínimo para poligonalizar un metaBlob.

4 Refinamientos de la Poligonalización

Experimentos y Conclusiones Finales

5

Una vez vistos los algoritmos que proponemos en el capítulo 3, y las mejoras propuestas en el capítulo 4, pasamos a describir la realización de la experimentación, describiendo su diseño, elaboración y realización. De esta forma, detallamos las propiedades empleadas para realizar una medida adecuada y objetiva de los mismos. Se abordan diferentes aspectos de medición, por un lado el rendimiento de los algoritmos en cuanto a su rapidez (velocidad de ejecución), también se mide la exactitud con el modelo matemático real, además se compara la calidad de la malla poligonal resultante de aplicar cada uno de los diversos métodos, dicha calidad viene dada por la uniformidad que presenten las mallas finales obtenidas. También se han comparado las variantes propuestas para las mejoras en cada uno de los algoritmos. Para finalizar el capítulo analizamos los resultados destacando las mejoras obtenidas en las conclusiones finales, terminando así, los objetivos planteados en esta tesis.

5 Experimentos y Conclusiones Finales

5.1 Diseño de los Experimentos

El diseño de los experimentos es, sin duda, una labor complicada, teniendo en cuenta que debemos cuantificar mediciones de diversos parámetros de los algoritmos estudiados de forma representativa y cuyos resultados sean comparables dentro de unas ciertas condiciones de igualdad.

En nuestro caso, los objetivos planteados han sido obtener ***algoritmos más rápidos*** **[PUCH97]**, que generen una ***malla poligonal más uniforme*** al tiempo que se obtiene la ***representación más exacta*** posible **[PUCH99]**, **[SAEZ99**, **[SAEZ00]**con respecto al modelo real.

Por tanto los tres grandes bloques de parámetros que vamos a medir en nuestros experimentos son:

- Velocidad en la ejecución de los algoritmos.
- Uniformidad de la malla poligonal obtenida.
- Similitud al modelo real.

Por otro lado los algoritmos que vamos a comparar son:

- Marching Cubes.
- Recorrido de las Superficie mediante Gradiente (RSG).

Para terminar, emplearemos también el conjunto de mejoras propuesta a los algoritmos:

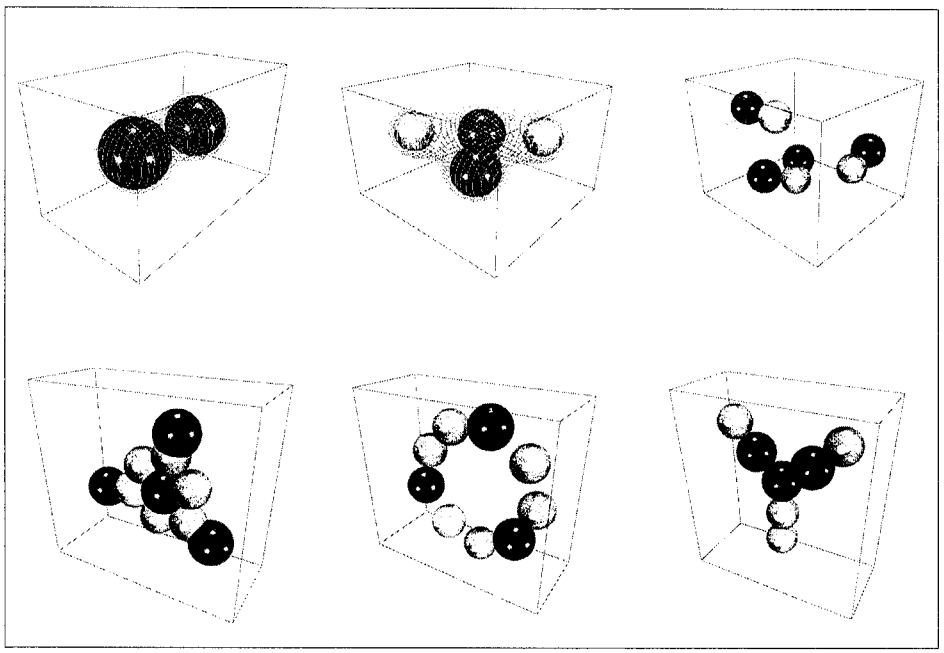
- Segmentación en componentes conexas.
- Orientación de los volúmenes de búsqueda.
- Rectificación de la precisión.

5.1.1 Velocidad de Ejecución

Para medir la velocidad de ejecución de todos estos algoritmos vamos a estudiar de forma comparativa los algoritmos estudiados, para ello hemos dispuesto de unas colecciones de configuraciones de primitivas generadoras lo más representativas posible, algunas

de las mismas se pueden observar en la siguiente figura. Estas configuraciones deben cumplir casos extremos, ya sean favorables o desfavorables, a todos los algoritmos, y casos intermedios.

Figura 5-1: Algunas configuraciones empleadas en los experimentos



La configuración del sistema empleado para la realización de los experimentos se encuentra descrito en el apéndice A, figura A-1.

Para cada una de estas configuraciones variamos la distancia de los puntos a discretizar que constituyen la nube de puntos que ha de formar la superficie metaBlob resultante. Para unificar este criterio se emplean distintos parámetros en cada uno de los algoritmos, pero que se pueden aproximar bastante, para asegurar la validez del experimento. En el caso del algoritmo de *Delaunay*, se emplea la distancia máxima entre puntos. Para el *Marching Cubes* se emplea el tamaño del vóxel y para el *RSG* el ángulo de rechazo y la distancia máxima entre vértices.

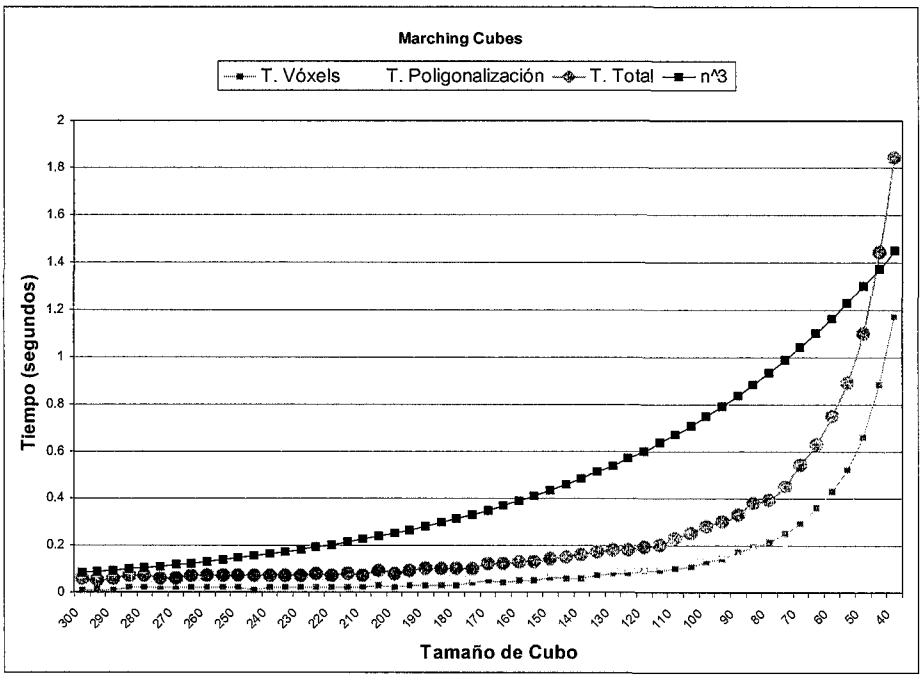


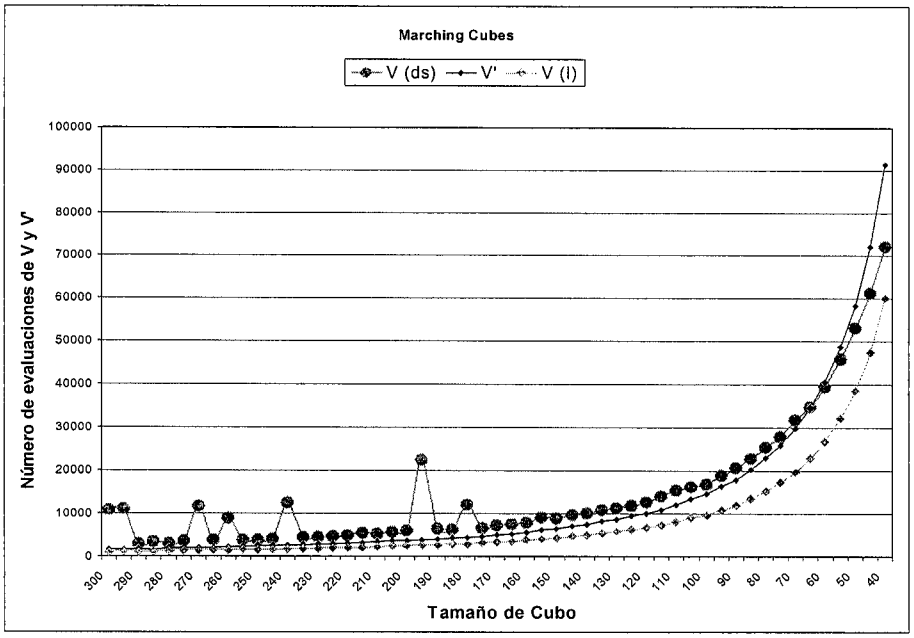
Figura 5-2: Curva del tiempo del cálculo de un *metaBlob* empleando Marching Cubes

De forma clara vemos la tendencia cúbica de la curva de tiempo según disminuimos el tamaño del cubo.

El coste teórico de este algoritmo depende del número de veces que se debe evaluar la función V para cada uno de los puntos. Que como ya dijimos en el capítulo 3 es proporcional al volumen de exploración.

El coste espacial es reducido debido a que se emplean dos matrices bidimensionales que almacenan los valores calculados de todos los valores de campo V para las posiciones z y $z+z+\Delta z$. De forma que se emplean para construir los vóxels en el caso del Marching Cubes o para la búsqueda de los contornos en el RSG, en este algoritmo sólo se emplea una matriz.

Figura 5-3:
Tiempo del cálculo de Marching Cubes, en función de V y V'



En esta gráfica hemos realizado tres tipos de mediciones:

- **Curva V(ds)**, que representa el número de veces que se ha calculado la función de potencial V (ecuación 2.7), pero empleando subdivisiones sucesivas (figura 3-32). Los picos que presenta esta curva son debidos al excesivo tamaño de cubo empleado, con lo que la función V se tiene que evaluar más veces para aproximar el valor del potencial requerido a un valor cercano a 0 muchas más veces, aunque el número de veces está acotado y nunca supera las 1000 divisiones dicotómicas.
- **Curva V'**, que representa el número de veces que se ha tenido que calcular el gradiente, o derivada de V (ecuación 3.2).
- **Curva V(I)**, que representa el número de veces que ha evaluado la función V sin emplear divisiones sucesivas. Curva de crecimiento del cálculo de un *metaBlob* empleando Marching Cubes

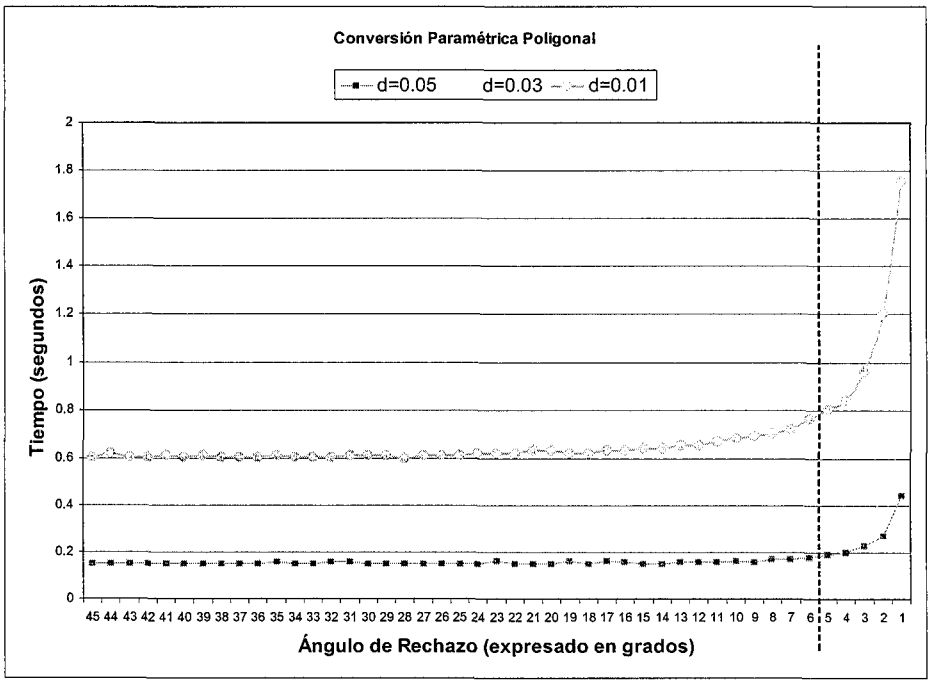


Figura 5-4: Curva de crecimiento del cálculo de un *metaBlob* empleando Marching Cubes

A continuación lugar vamos a comparar el tiempo necesario que necesitan, *Marching Cubes* y el Recorrido de Superficie mediante Gradiente, RSG, para generar un mismo número de triángulos.

El resultado de la siguiente gráfica asegura que el algoritmo RSG es mucho más rápido siendo capaz de generar un volumen de triángulos mucho mayor en menor tiempo.

Existe un punto de intersección, sobre los 16.000 triángulos, donde el algoritmo de Marching Cubes se demuestra más rápido, pero hay que tener en cuenta que la malla poligonal obtenida es de pero calidad.

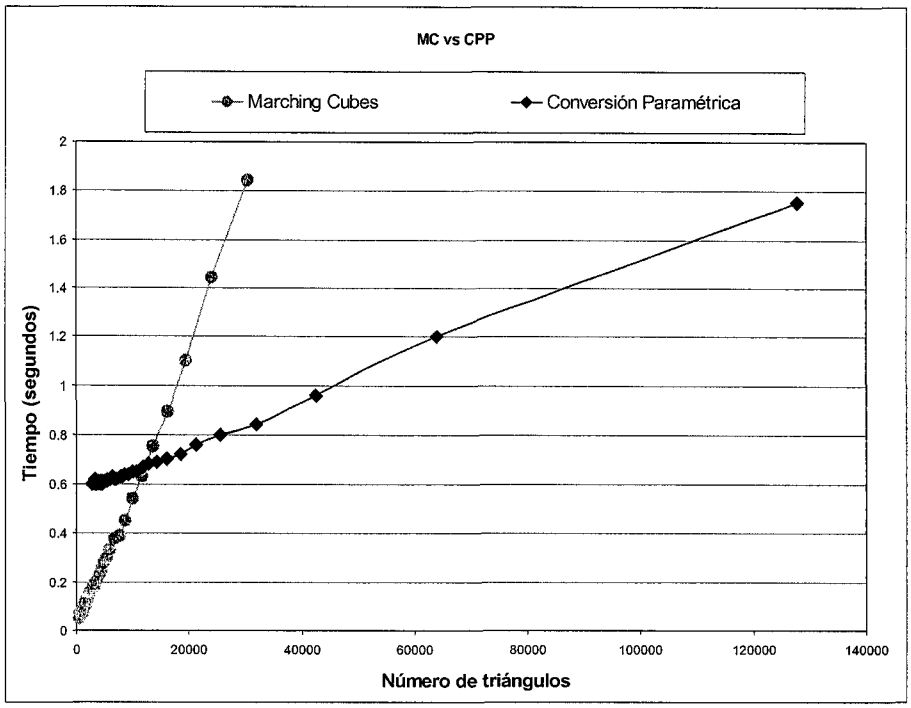


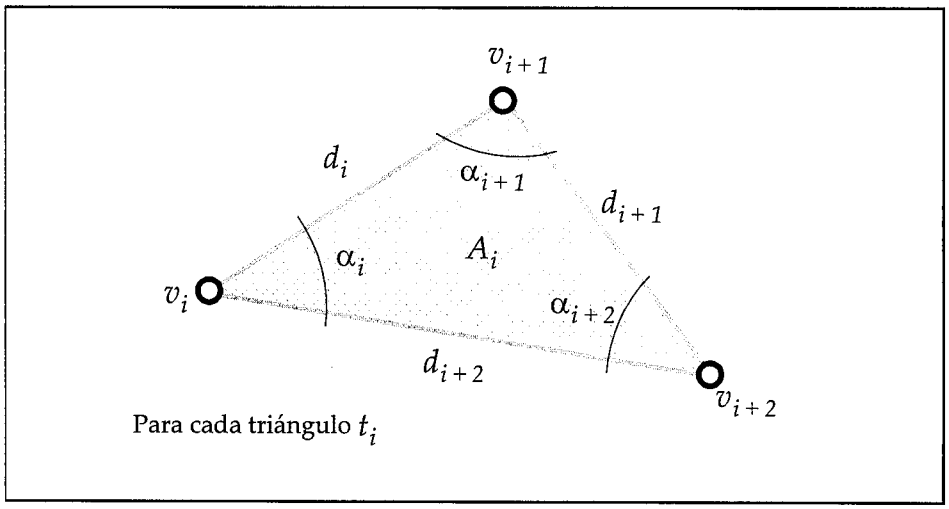
Figura 5-5: Comparativa de MC contra RSG

5.1.2 Uniformidad de la Malla Poligonal

La uniformidad de la malla poligonal es un factor importante para obtener una representación tridimensional con la suficiente calidad del metaBlob que queremos reconstruir. La definición de malla óptima viene asociada a unos criterios de forma y tamaño concretos [REF] necesitamos que los triángulos que generemos cumplan una serie de requisitos. Los cuales los enumeraremos a continuación:

- Sus formas debe ser parecidas (mediremos los tres ángulos de los vértices).
- Sus tamaños debe ser parecidos (mediremos la superficie total y las longitudes de sus aristas).

Figura 5-6: Parámetros de medición de cada triángulo de la malla.



Los ángulos estarán, preferiblemente y según observaciones experimentales, entre estas medidas:

$$34° \leq \alpha_i \leq 90° \qquad \textbf{(5.1)}$$

La manera de puntuar cada modelo será empleando las siguientes propiedades:

- Angulos de los vértices (α_i).
- Distancia de las aristas (d_i).
- Superficie del triángulo (A_i).

Se dan tres valores globales para el metaBlob con el promedio calculado empleando todos los triángulos que componen su malla poligonal. Estos valores serán:

$$\alpha_G = \frac{\sum_{i=0}^{n-1} \alpha_i}{3 \cdot n} \quad d_G = \frac{\sum_{i=0}^{n-1} d_i}{3 \cdot n} \quad A_G = \frac{\sum_{i=0}^{n-1} A_i}{n} \qquad \textbf{(5.2)}$$

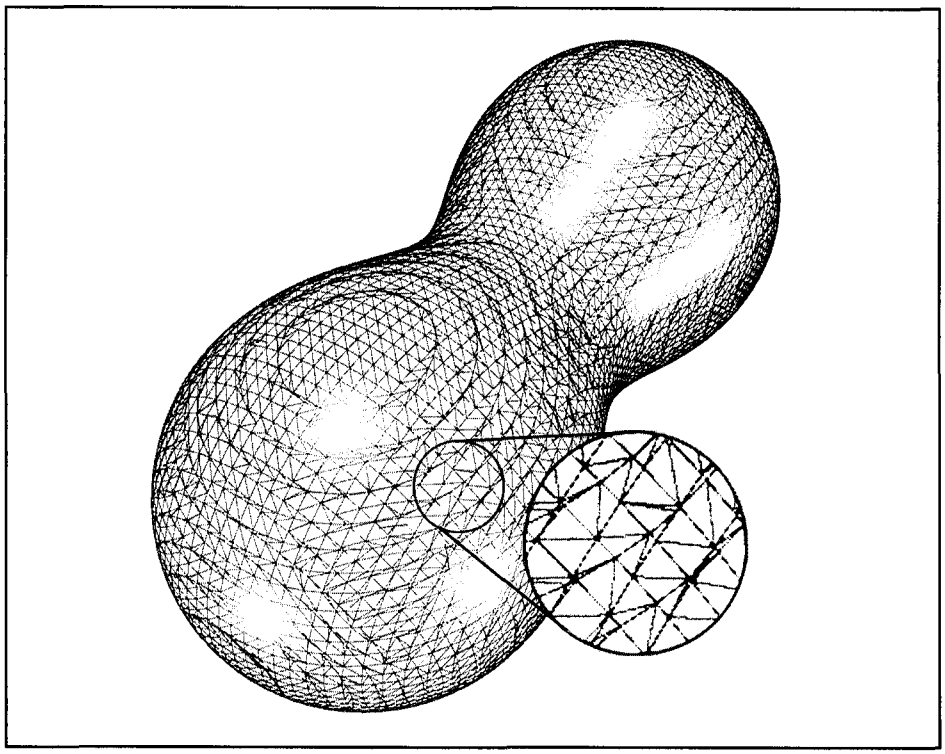


Figura 5-7: Malla obtenida empleando el algoritmo de Marching Cubes

Como se puede observar presenta triángulos de formas irregulares, que pueden producir efectos no deseados al visualizar el modelo sólido asociado a este tipo de mallas poligonales. El principal problema de este tipo de mallas es que presentan triángulos con formas extremas, donde los ángulos de sus vértices son muy pequeños, siendo muy complicado el pegado por ejemplo de texturas. O incluso el cálculo de la Normal no produce efectos satisfactorios, que no es el caso de OpenGL, porque podemos pasarle la normal, el gradiente en dicho vértice, a cada uno de los mismos de forma que la dirección de la normal sea la correcta la malla obtenida empleando el algoritmo RSG.

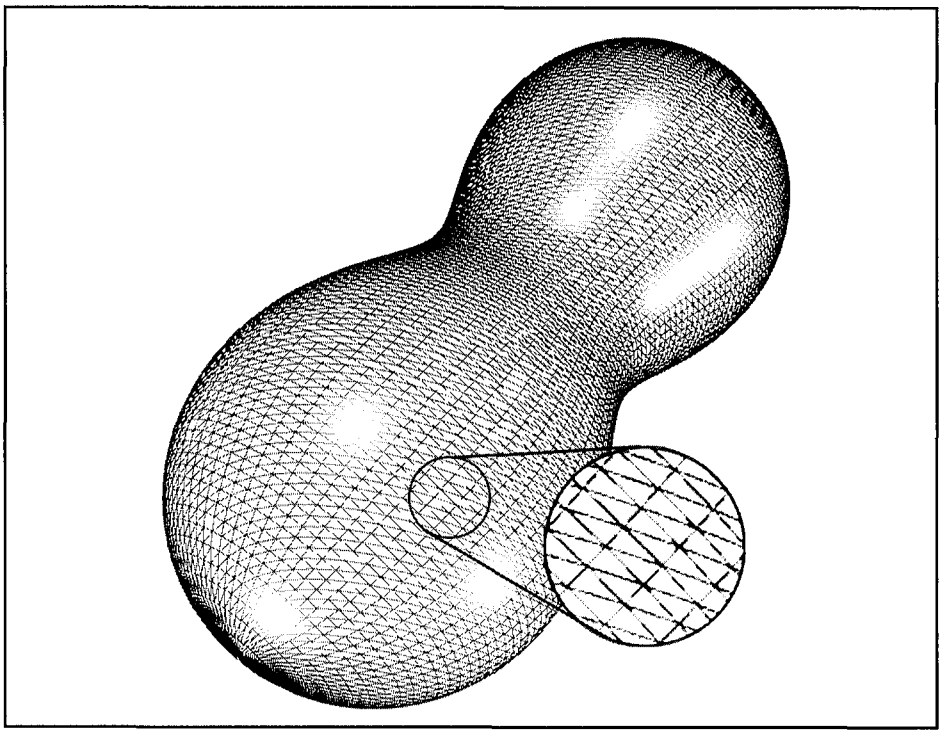


Figura 5-8: Malla obtenida empleando el algoritmo RSG

La uniformidad de la malla poligonal se traduce en una representación visual más correcta a la vez que atractiva. Esta uniformidad influye también en el cálculo de las normales, y junto a la forma de los triángulos repercute en varios aspectos:

- Cálculo de la iluminación difusa.
- Forma y distribución de los brillos especulares.
- Aplicación de texturas.
- Aplicación de mapas de relieve.

En casos extremos, algoritmos como *Marching Cubes* producen efectos incorrectos en ambos aspectos, no siendo la malla adecuada y teniendo por tanto que aplicar postprocesos de mejora de las mallas obtenidas incrementándose así el tiempo de procesamiento de forma notable.

En la siguiente tabla se expresan los resultados esperimentales de los valores de calidad de la malla poligonal.

Marching Cubes	*Valor mínimo*	*Valor máximo*
Angulos	*0.01*	*170*
Longitudes de las aristas	*0.0*	*0.16*
Area	*0.0*	*72.43*

Conversión Paramétrica Poligonal (RSG)	*Valor mínimo*	*Valor máximo*
Angulos	*34*	*90*
Longitudes de las aristas	*0.01*	*0.08*
Area	*0.12*	*0.28*

5.1.3 Medición del Error. (Similitud al Modelo Real).

Por último empleamos un modelo de error para cuantificar el parecido de la malla poligonal obtenida con la superficie del modelo real, para ello medimos la distancia entre el punto medio de cada arista obtenida y, siguiendo la dirección de su gradiente, vector normal del punto, el punto real más cercano con una precisión ε. Una vez obtenido dicho punto definimos el error como la distancia del vértice al punto real. Por tanto la función de error para cada vértice se define como:

$$f_\varepsilon(v_i) = d(v_i, p_i), p_i < \varepsilon \qquad \textbf{(5.3)}$$

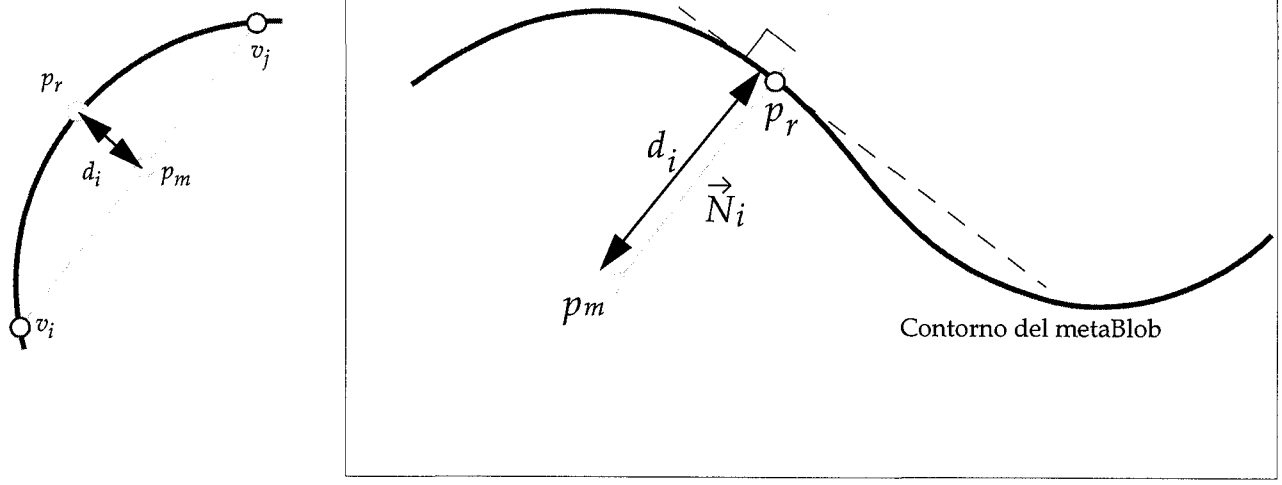
Siendo v_i el i-ésimo vértice, p_i el punto real más cercano al punto

real y d la distancia euclidea. Dicho punto se obtiene buscando a lo largo del vector que define la normal al vértice, gradiente de V en dicho vértice, un punto con valor cercano a 0, es decir cuyo valor sea menor que ε, que expresa la proximidad al valor 0, que es el valor donde se cumple que el vértice pertenece exactamente a la superficie.

$$V(x,y,z) = \sum_{i=1}^{N} b_i \cdot e^{-a_i f_i(x,y,z)} - T = 0 \qquad \textbf{(5.4)}$$

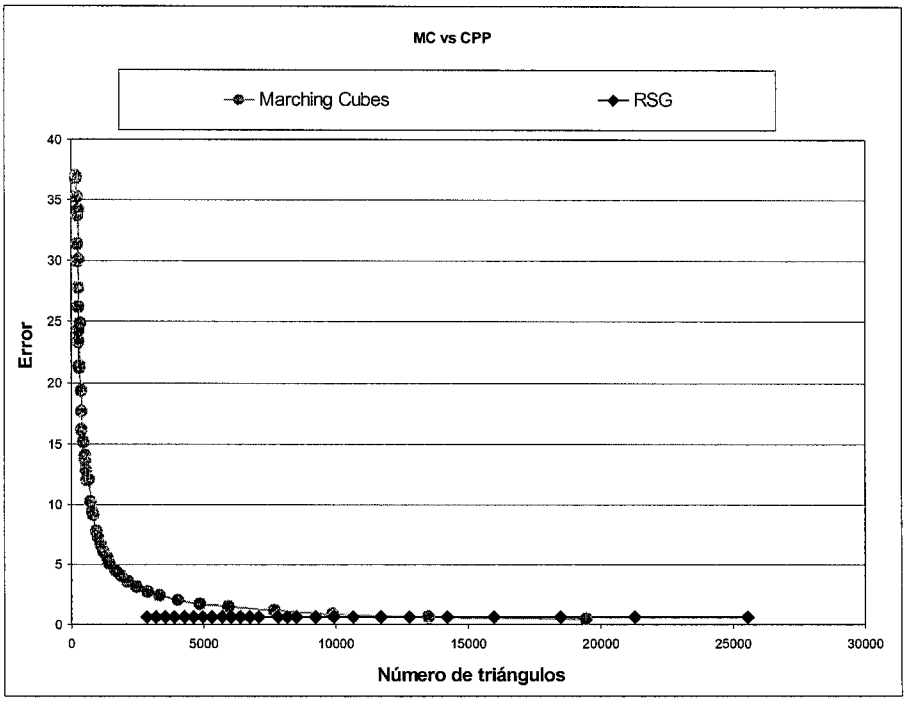
Dicho error mide la distancia entre el punto calculado y el punto real más cercano, cuyo cálculo se realiza empleando el valor de la normal, que es el valor del gradiente, en cada uno de los vértices, y buscando el valor real con un cierto epsilon de exactitud, en la siguiente figura se representa la forma de calcular estos valores.

Figura 5-9: Cálculo del punto real de la superficie del metablob



Como medida de similitud sumamos todos los errores y los normalizamos dividiendo entre el total de puntos o aristas empleados. La siguiente tabla representa el error cometido para cada modelo cada vez que aumentados el tamaño del cubo de búsqueda (es decir cometemos mayor error).

Figura 5-10: Comparación del error de la malla poligonal en MC y RSG



Como se puede ver el error generado por *RSG* es mucho menor que el que genera *Marching Cubes*, prácticamente es constante, mientras que en *MC*, el tamaño del cubo (relacionado con el número de triángulos de forma proporcionalmente), influye en la exactitud de la malla poligonal.

5.2 Conclusiones Finales

Los aspectos más significativos que hemos aportado en esta tesis, para la generación de la malla poligonal de un objeto 3D codificado con el uso del modelo *metaBlob*, son:

- Un algoritmo más rápido que los clásicos,
- y que al tiempo genera una malla poligonal óptima.

Y también:

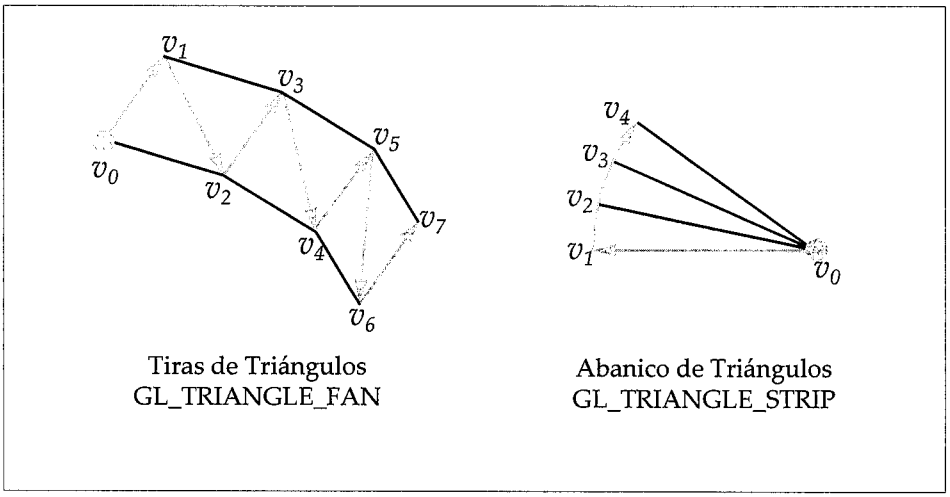
- Un preproceso de localización de volúmenes de exploración óptimos.
- Un postproceso de mejora aplicable a cualquiera de los otros algoritmos.

Con el desarrollo del algoritmo RSG, hemos conseguido un algoritmo que requiere de menor tiempo de procesamiento. Esta mejora en el tiempo de cálculo se ha conseguido gracias al empleo de varias técnicas, por un lado, la simplificación en la construcción, de forma geométrica, de la malla poligonal completa empleando estructuras más adecuadas, y por otro lado, realizamos una exploración de tipo local para averiguar puntos de la superficie.

La primera mejora la obtenemos al calcular un contorno en el que establecemos los puntos ordenados según el recorrido de búsqueda de dicho contorno, esto permite emplear tiras de triángulos (*triangle fan*) para unir un contorno de vértices con el siguiente, y también abanico de triángulos (*triangle strip*), para el comienzo y finalización de contornos. *Marching Cubes*, sin embargo, al no establecer ningún criterio de ordenación en la generación de los triángulos deben construir la malla empleando triángulos independientes.

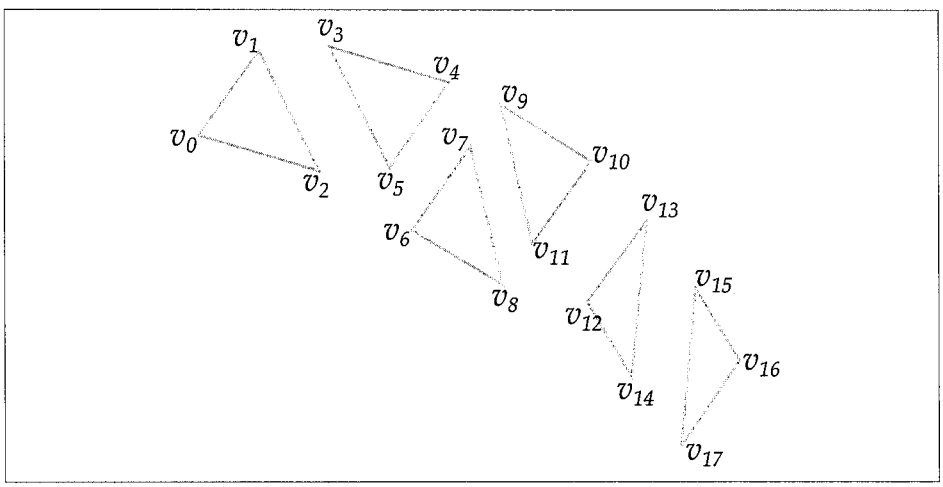
Las estructuras empleadas permiten reducir el número de vértices y vectores Normales a los vértices. Si obtenemos n triángulos, en Marching Cubes tenemos que construir $3n$ vértices y $3n$ normales, sin embargo en el algoritmo RSG, construimos n vértices y n normales, siendo la cantidad de información necesaria para generar una malla poligonal equivalente, tres veces menor.

Figura 5-11: Estructuras geométricas más compactas para especificar una malla poligonal



Por otro lado, la búsqueda de puntos pertenecientes a la superficie del modelo metaBlob, se realiza de forma local, es decir, una vez que conozco un punto de dicha superficie, busco el siguiente punto en una zona próxima siguiendo los criterios explicados en el capítulo 3 (figuras 3-22 a 3-25), lo cual reduce el espacio de búsqueda local al entorno próximo.

Figura 5-12: Representación con triángulos independientes



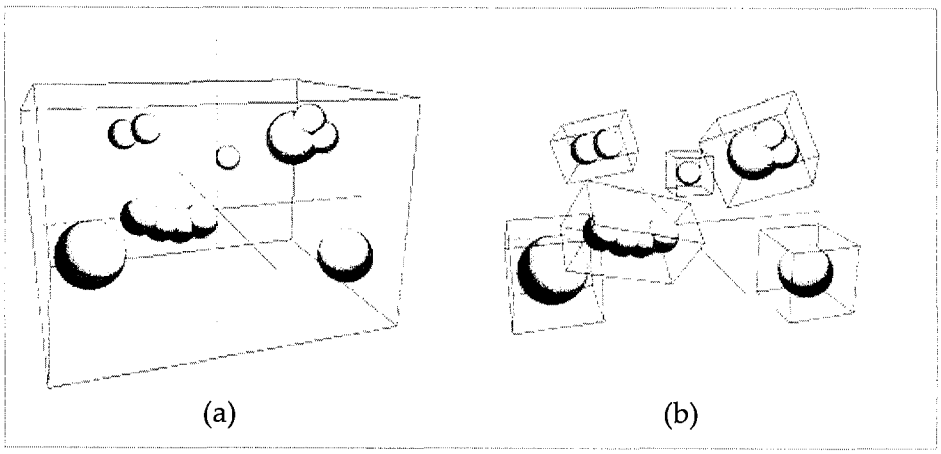
En la siguiente tabla se presenta un ejemplo numérico con cada una de las estructuras geométricas que hemos empleado, y el caso genérico para construir *n* triángulos:

Figura 5-13: Comparativa de los triángulos necesarios por estructura geométrica

	Nº Triángulos	*Triángulos (nº Vértices)*	*Tiras (nº Vértices)*	*Abanico (nº Vértices)*
Figuras 5-11 y 5-12	*6*	*18*	*8*	*8*
Caso General	*n*	*3*n*	*n+2*	*n+2*

Por otro lado el preproceso para reducir el volumen global nos permite reducir drásticamente el tiempo de procesamiento al reducir el espacio de búqueda al máximo. El tiempo de procesamiento es proporcional a la suma de volúmenes obtenidos.

Figura 5-14: Cálculo y orientación de los volúmenes óptimos



Por último el preproceso para la rectificación de polígonos nos permite obtener mayor precisión allí donde se necesita mayor adaptación a la superficie por presentar ésta una mayor curvatura. Este preproceso se ha demostrado también útil para cualquier técnica de poligonalización.

Volver al índice/Tornar a l'índex

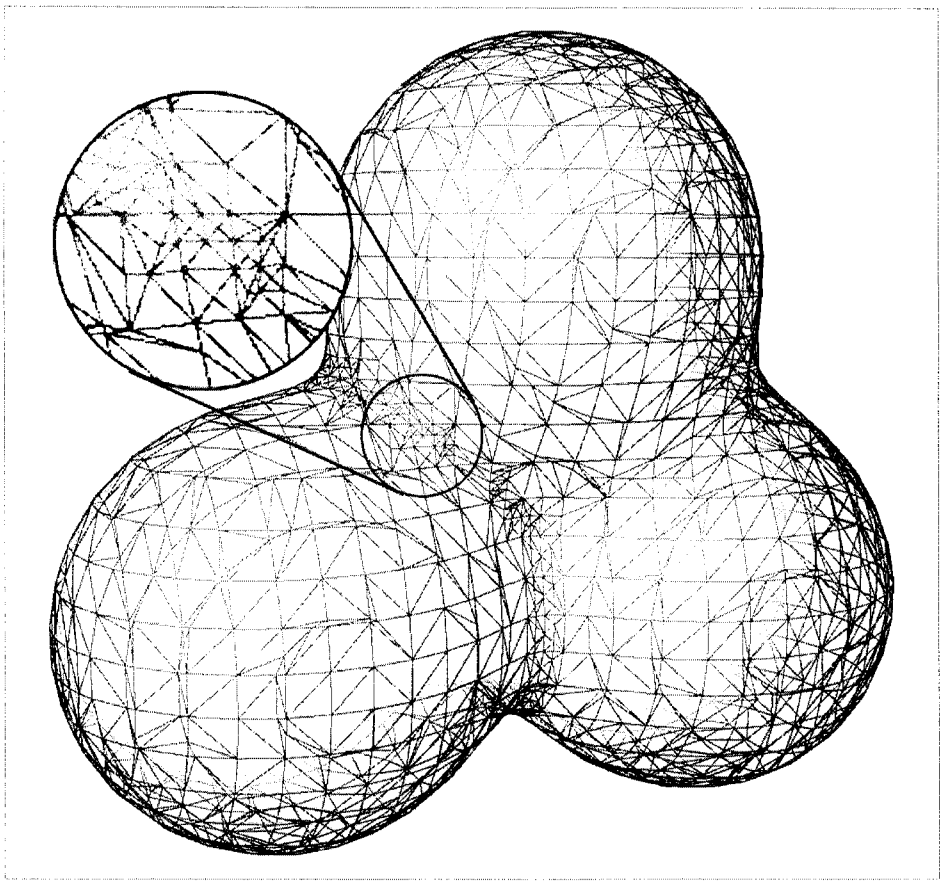


Figura 5-15: Marching Cubes con rectificación de precisión

Como se puede observar en esta figura los triángulos se van subdiviendo hasta que cumplen el umbral de rechazo de ángulo expuesto en el capítulo 3.

Para terminar con las conclusiones hemos comprobado que la precisión que se obtiene por RSG es mucho mayor en un tiempo mucho más reducido (gráfica comparativa 5-5) que la obtenida por *Marching Cubes*.

Por tanto los objetivos que nos planteábamos cubrir con esta tesis se han cubierto en su totalidad y además hemos comprobado la validez de nuestros métodos de forma experimental.

5 Experimentos y Conclusiones Finales

Líneas Futuras

6

En este capítulo realizamos una breve introducción a las líneas futuras, empezaremos con el paralelismo y terminaremos con el uso y empleo de las últimas capacidades aceleradoras por hardware que han aparecido en la última generación de tarjetas gráficas 3D, mediante el uso de los Vertex Shaders. Además del posible empleo de los triángulos curvos PN.

6 Líneas Futuras

6.1 Multiprocesamiento y Paralelización de Algoritmos.

En este capítulo realizaremos una síntesis de aquellas ideas que pretendemos ampliar en un futuro. Pese a que no se trata de técnicas totalmente terminadas, nos ha parecido oportuno incluirlas en este proyecto, puesto que despiertan un gran interés por nuestra parte.

Inicialmente, nos adentraremos en el mundo de la multiprogramación, para explicar cómo podríamos plantear los algoritmos de poligonalización desde el punto de vista de una arquitectura multiprocesador. La posibilidad de que un algoritmo sea paralelizable presenta grandes ventajas, sobre todo cuando estamos tratando con algoritmos de alto coste o pretendemos que un algoritmo responda en tiempo real. Por bajo que sea el coste de cualquier algoritmo, es interesante que pueda ser paralelizado, puesto que la talla de un problema puede hacer que el tiempo de cómputo para procesarlo sea inaceptable. La paralelización de un algoritmo, siempre que esté bien planteada, suele provocar que dicho tiempo de cómputo se reduzca en función del número de procesadores.

Por otro lado, planteamos una serie de ideas para la construcción de algoritmos de poligonalización en tiempo real. Para conseguir esto, no solamente hace falta un buen algoritmo, es preciso tener en cuenta factores de un nivel mucho más bajo como, por ejemplo, guías de cómo evitar cálculos mediante aproximaciones, aprovechar coherencias espaciales, etc. Como veremos, es preciso realizar asunciones no necesariamente ciertas, y saltarnos planteamientos de precisión como los que hemos planteado, que van en contra de esta filosofía.

6.1.1 Posibilidades Multicomputacionales

En este apartado plantearemos cómo podrían paralelizarse los algoritmos de poligonalización vistos en los capítulos anteriores. Inicialmente, introduciremos el paralelismo que hasta ahora se ha

adoptado en otras técnicas dentro de la informática gráfica.

6.1.2 Paralelización en Algoritmos de Raytracing

Las técnicas de raytracing suelen consumir tiempos de computación que, en algunas ocasiones, son inaceptables. Por ejemplo, a la hora de hacer una película en la que aparezcan multitud de escenas sintéticas, se han de generar millones de fotogramas. Dependiendo de la resolución y de la complejidad de las escenas, la representación de un único fotograma con estas técnicas puede tardar horas. Es por ello que se suele recurrir a técnicas de multicomputación. Es el caso de la película *Titanic*, en la que los exteriores del barco se generaron en su totalidad por ordenador. Se creó un multicomputador de 150 nodos bajo el sistema operativo Linux. El software de paralelización que se usó fue PVM (Parallel Virtual Machine).

El tiempo de cómputo para generar una imagen con técnicas de trazado de rayo viene determinado por el tamaño de la matriz a visualizar y por la complejidad de la figura. Para acelerar el algoritmo desde el punto de vista de la complejidad, se ha de modificar la propia técnica. Sin embargo, para mejorar el tiempo de cómputo desde el punto de vista del tamaño de la matriz a generar, tenemos que recurrir a las técnicas de multicomputación.

El procedimiento consiste en dividir el raster (tamaño de la imagen) en varias porciones y tratar cada porción en un procesador distinto. Siguiendo un esquema cliente-servidor, el servidor recibe la escena y asigna una porción de la misma a cada cliente. Cuando el cliente termina con su porción del raster, envía los resultados al servidor. Para ello, cada cliente ha de poseer la información tridimensional de la escena completa. Por este motivo, estas técnicas suelen implementarse en arquitecturas de memoria compartida. En la figura 6-1 incluimos un esquema de este procedimiento.

6 Líneas Futuras

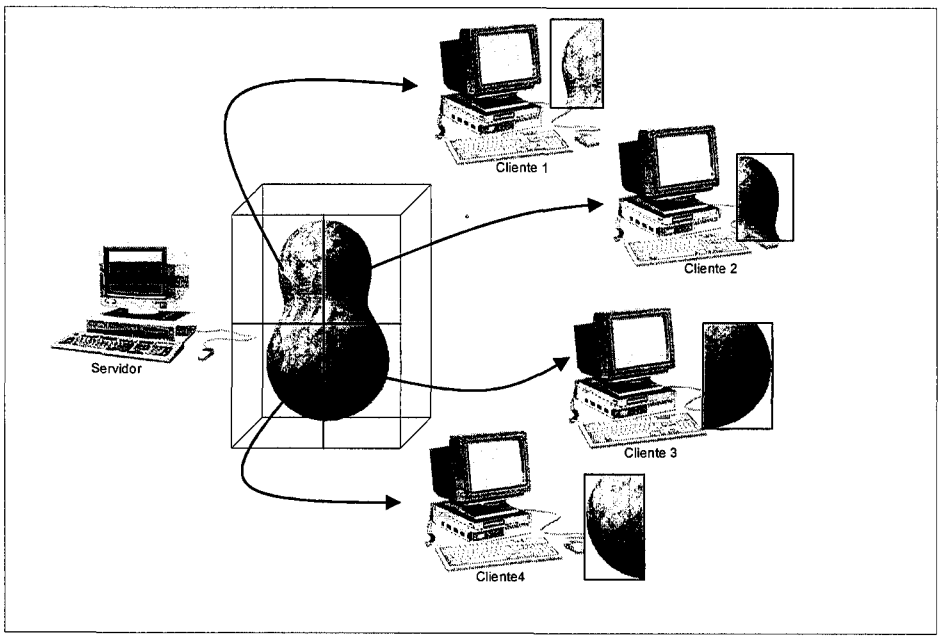


Figura 6-1: Procedimiento de paralelización de algoritmos de raytracing.

Suele realizarse una división del raster en rectángulos del mismo tamaño, aunque, dependiendo de la técnica de trazado que usemos, se puede determinar el tiempo de cálculo de cada porción y realizar dicha división en rectángulos de igual coste. Homogeneizando la carga de los clientes conseguimos que el tiempo de cómputo general disminuya.

Como vemos, esta técnica no es excesivamente complicada y disminuye bastante el tiempo de cómputo de cada render. Por otro lado, el ensamblado de subimágenes que ha de hacer el servidor para montar el raster completo es directo.

A continuación vamos a explicar cómo podríamos aplicar estas técnicas a los algoritmos vistos en los capítulos anteriores.

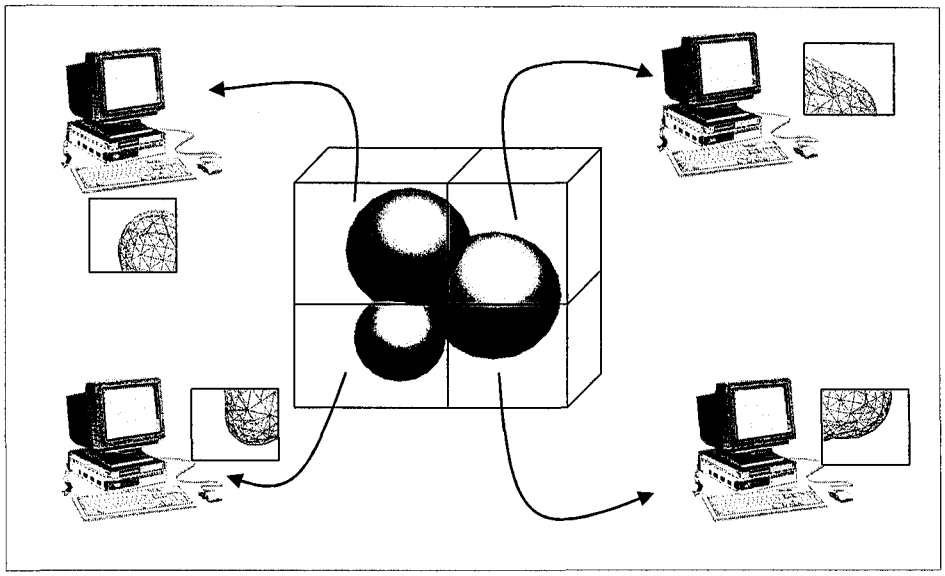
6.1.3 El paralelismo en Algoritmos de Poligonalización

Vamos a plantear una técnica para dividir el problema de la

poligonalización de algoritmos basados en vóxels en subproblemas aislados, que puedan tratarse en procesadores distintos.

Para ello, dividiremos el espacio de búsqueda (paralelepípedo inicial) en N partes. Cada volumen será calculado por un cliente. Puesto que el patrón de corte en los algoritmos basados en vóxels es un paralelepípedo, las divisiones que haremos del mismo también serán paralelepípedos. En principio, el número de divisiones del paralelepípedo inicial será múltiplo de 2. De esta forma podemos cortar el vóxel por mitades de forma que generemos un árbol binario. En la figura 6-2 tenemos un ejemplo de este procedimiento.

Figura 6-2: División del espacio de rastreo en cuatro paralelepípedos.



Tendremos tantas unidades de computación como partes de división en el paralelepípedo (N). Dichas unidades han de contar con la información completa de las primitivas del metaBlob. Cada una realiza el algoritmo de poligonalización sobre el espacio de rastreo (porción del paralelepípedo inicial) que le asigne el servidor. De esta forma, cada cliente devuelve una submalla asociada a su volumen parcial.

Para ensamblar las submallas que calculan los clientes, tenemos que hacer coincidir los vértices frontera de cada una. Para ello, solamente es necesario realizar un cálculo del vecino más cercano a cada vértice. En dicho cálculo contaremos únicamente los vértices frontera de las mallas para que el tiempo de cálculo no aumente.

En el cliente no se tiene que hacer ningún tratamiento especial para calcular la poligonalización. Es suficiente con restringir el espacio de rastreo al paralelepípedo que proporciona el servidor, en vez de calcular el paralelepípedo inicial.

El problema que queda por resolver es cómo tratar la información para que no se produzca un tráfico excesivo. En la fase de envío de la información se ha de enviar un volumen de datos bastante reducido. Sin embargo, en la fase en la que los clientes devuelven la información (malla poligonal), podemos tener problemas de saturación. Esto provocaría un encarecimiento del tiempo de respuesta. Para resolverlo podemos comprimir los datos, o bien establecer una arquitectura basada en conexiones punto a punto entre los clientes y el servidor.

6.1.4 Guías para el Diseño de Algoritmos en Tiempo Real

Diseñar un algoritmo de poligonalización de metaBlobs en tiempo real no es una tarea fácil. En principio, el algoritmo ha de ser lineal. Además, dicha linealidad ha de presentar una pendiente lo más cercana a 0 posible. De hecho, un algoritmo en tiempo real ideal presentaría un coste lineal de pendiente horizontal. Para ello, es necesario eliminar todos los cálculos posibles. La mejor manera de hacerlo sería partir de un algoritmo lineal y reducir todos los cálculos matemáticos a sumas y restas. Desgraciadamente eso no siempre es posible. Vamos a plantear una serie de ideas alternativas para orientar el diseño de estos algoritmos:

Vamos a partir de *Marching Cubes* con el postproceso de rectificación que, como vimos, es lineal. Para reducir el tiempo de cómputo aplicaremos las siguientes ideas:

1) Construir una tabla para las funciones trigonométricas directas que se usen (basta con las funciones seno y coseno). Cada tabla tendrá 360 entradas y contendrá el resultado de la función para cada ángulo. El tiempo de acceso a una tabla de reducido tamaño es considerablemente menor al de calcular estas funciones. Con este planteamiento, incurrimos en un error máximo de 1^{o} en cada cálculo.
2) Aprovechar la coherencia espacial completa almacenando en una matriz las medidas de potencial en los vértices de los cubos. Según Sorensen y Clyne, aprovechando al máximo la coherencia espacial se podía reducir hasta 1/3 el tiempo de computación.
3) La forma más exacta de calcular el punto de corte de una arista con una isosuperficie es realizar una dicotomía en función del potencial. Para evitar este cálculo podemos aproximar el punto de la superficie con una interpolación lineal entre los dos vértices implicados, en función del potencial en los mismos. Este cálculo se realiza contínuamente tanto para cortar cada cubo con el metaBlob como para encontrar los nuevos vértices en el postproceso. Este aspecto reduce drásticamente el tiempo de cómputo, pero los resultados visuales de la aproximación dependen directamente de la distancia a la que se encuentren los vértices.
4) Aplicar las técnicas de segmentación en volúmenes conexos y colocación óptima de volúmenes que se explican en el capítulo 4. Sería aconsejable estudiar la concentración espacial de las primitivas para determinar si se han de aplicar o no estas técnicas en cada caso concreto. Esto se debe a que, en algunos casos, los cálculos pueden encarecer innecesariamente el tiempo de cómputo del algoritmo.

Aún más interesante que construir un algoritmo en tiempo real que, como vemos, es complicado, sería construir una técnica que nos permita diseñar en tiempo real. La idea consiste en modificar la malla poligonal únicamente en el volumen afectado por un cambio significativo. Por ejemplo, si el usuario crea, elimina o transforma una sola primitiva, modificamos únicamente la porción de malla a

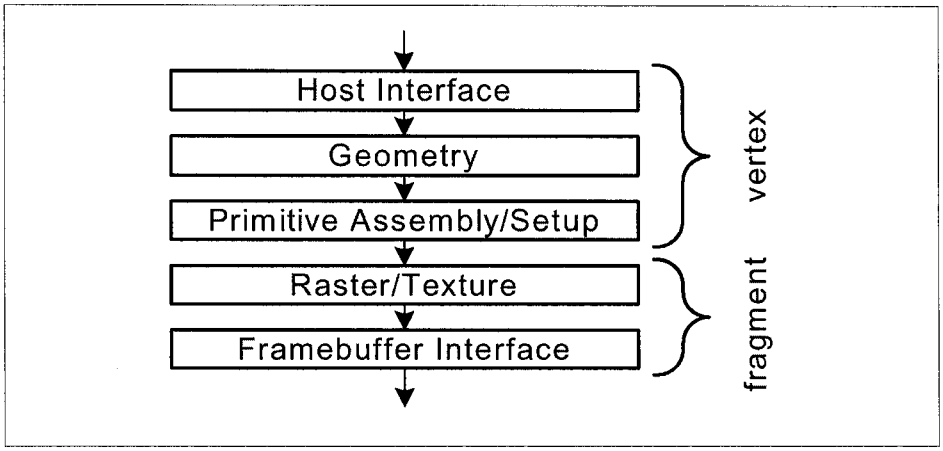
la que puede afectar. Utilizando la cota máxima de afectación de una primitiva (capítulo 4), podemos determinar el volumen de afectación de la misma. Puesto que, en el caso de Marching Cubes, el espacio está dividido en una malla de vóxels, solamente tendríamos que determinar los vóxels que se encuentran incluidos en dicho volumen de afectación. Formamos una estructura que guarde la información de los triángulos que pertenecen a cada vóxel. A continuación calculamos la poligonalización en los vóxels de afectación y la substituimos por la lista de triángulos que guardaba anteriormente.

Como vemos, las técnicas que persiguen resultados en tiempo real para problemas de este calibre, requieren de una gran destreza a la hora de ser implementadas puesto que, a este nivel, cualquier procedimiento es mejorable.

6.2 Aceleración Hardware usando Vertex Shaders

En este apartado vamos a describir el motor de vértices acelerado por hardware que incorporó NVidia en su última generación de tarjetas gráficas 3D, las GeForce3 [**NVIDIA**]. Este tipo de tarjetas se empiezan a considerar como las primeras generaciones que contienen una GPU (Graphics Processor Unit), y no una mera DPU (Display Processor Unit) como tenían las generaciones anteriores. Se puede considerar, desde un punto de vista conceptual, que una tarjeta gráfica 3D es una especie de miniordenador, porque tiene su propia CPU, su propia memoria y no requiere de la atención de la CPU central, por tanto una vez que la tarjeta gráfica dispone de toda la información geométrica 3D es capaz de visualizarla de forma autónoma en la pantalla.

Figura 6-3: Unidad Gráfica de Procesamiento (GPU)



La GPU de la *GeForce3* cuenta con una serie de tuberías (*pipelines*) para procesar y acelerar varias operaciones en paralelo. Una de las informaciones en la que se apoya toda representación de objetos 3D, y por tanto es la unidad básica, son los vértices. Hay dos posibilidades principales para procesar la información de los vértices: como vértices independientes o como parte de una primitiva geométrica, por ejemplo un triángulo. La ventaja de la información a un nivel más primitivo es permitir operaciones tales

como la ocultación de las caras no visibles, reduciendo de esta manera, el tiempo de procesamiento. Sin embargo, se ha determinado que el incremento de complejidad y pérdida de paralelismo en el modelo de procesamiento de primitivas no justifica los beneficios percibidos. Por ello, elegimos un modelo de programación de vértices independientes para explotar la naturaleza paralela de la tarea. El modelo de programación es capaz de expresarlo todo en una función fija de la tubería excepto los planos de recortado fijados por el usuario.

El tipo de datos más empleado con una precisión estándar para transformaciones 3D son los números en coma flotante IEEE de precisión sencilla. Los datos más comunes en gráficos 3D son vectores de tres y cuatro componentes, por ejemplo, la posición, la normal, coordenadas de textura y colores. Por tanto, el tipo básico de datos es una 4-tupla escrita como (x, y, z, w).

Es crítico manejar eficientemente la extracción y empaquetamiento de la información escalar y vectorial en este diseño ya que la tubería de transformación 3D mezcla estas dos operaciones. Dos conceptos muy sencillos pueden resolver este problema:

1. En la entrada los vectores pueden tener sus componentes arbitrariamente reordenadas o duplicadas.
2. Cualquier operación que genere un escalar debe generar un escalar que se duplique en todas sus componentes.

El modelo de programación se ilustra en la figura 6-4. Los atributos actuales del vértice se encuentran disponibles en los registros de entrada, y los atributos de los vértices procesados se escriben en los registros de salida. El banco de registros constantes mantiene los parámetros de transformación e iluminación, existiendo un conjunto de registros que mantiene resultados temporales.

Asegurando que el programa de vértices toma los datos de entrada sólo de lectura y los datos de salida son sólo de escritura, aseguran la naturaleza de flujo de los datos en el diseño y simplifican la implementación.

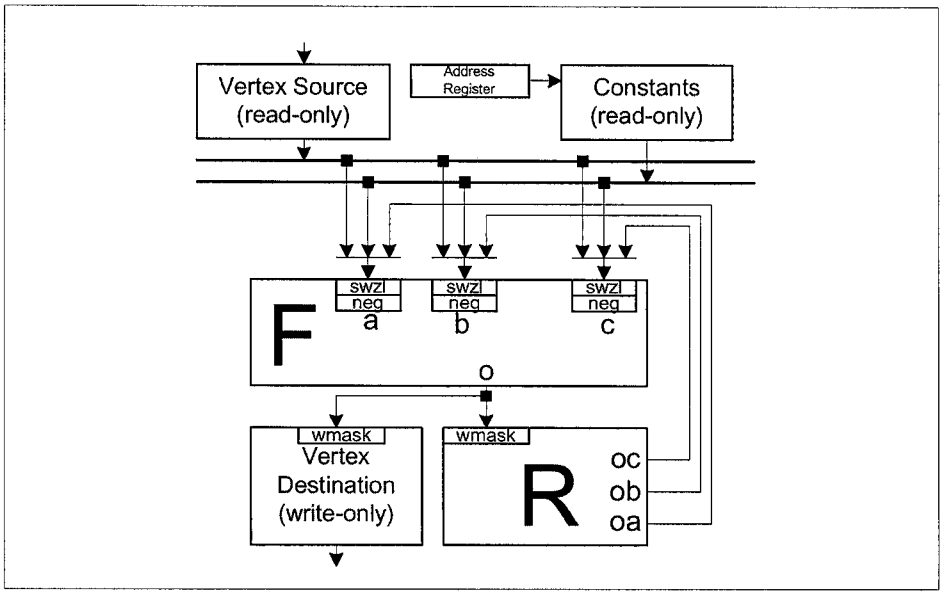


Figura 6-4: Esquema del modelo de programa

Resumiendo, la información que pueden mantener los registros es la siguiente:

1. **Atributos de entrada.** Hay 16 registros de entrada en el vértice siendo cada uno de ellos una 4-tupla de números reales (*floats*) IEEE. Normalmente la información que se almacena es la posición, la normal, dos colores, y hasta 8 conjuntos de coordenadas de textura, niebla y el tamaño del punto.
2. **Atributos de salida.** Ya que la función fija de la tubería trabaja en un espacio de coordenadas homogéneas, todos los resultados son normalizados al rango de valores de 0.0 a 1.0.
3. **Conjunto de instrucciones.** Está compuesto por 17 operaciones, que pueden clasificarse en tres categorías: vectoriales, escalares y operaciones misceláneas. La siguiente tabla ilustra el conjunto completo de operaciones.

6 Líneas Futuras

Figura 6-5: Conjunto de instrucciones de un programa de vértices (vertex program)

Código de operación	*Nombre completo*	*Descripción*
MOV	*Move*	*vector -> vector*
MUL	*Multiply*	*vector -> vector*
ADD	*Add*	*vector -> vector*
MAD	*Multiply and add*	*vector -> vector*
DST	*Distance*	*vector -> vector*
MIN	*Minimum*	*vector -> vector*
MAX	*Maximum*	*vector -> vector*
SLT	*Set on less than*	*vector -> vector*
SGE	*Set on greater or equal*	*vector -> vector*
RCP	*Reciprocal*	*scalar -> replicated scalar*
RSQ	*Reciprocal square root*	*scalar -> replicated scalar*
DP3	*3 term dot*	*scalar -> replicated scalar*
DP4	*4 term dot product*	*vector -> replicated scalar*
LOG	*Log base 2*	*miscellaneous*
EXP	*Exp base 2*	*miscellanous*
LIT	*Phong lighting*	*miscellaneous*
ARL	*Address register load*	*miscellaneous*

En la especificación realizada por Nvidia se proporciona un API de programación tanto para OpenGL como para Direct3D. La programación bajo OpenGL se realiza empleando la extensión que recibe el nombre de *NV_vertex_program*. En la siguiente figura podemos ver un ejemplo de *morphing* empleando un programa de vértices.

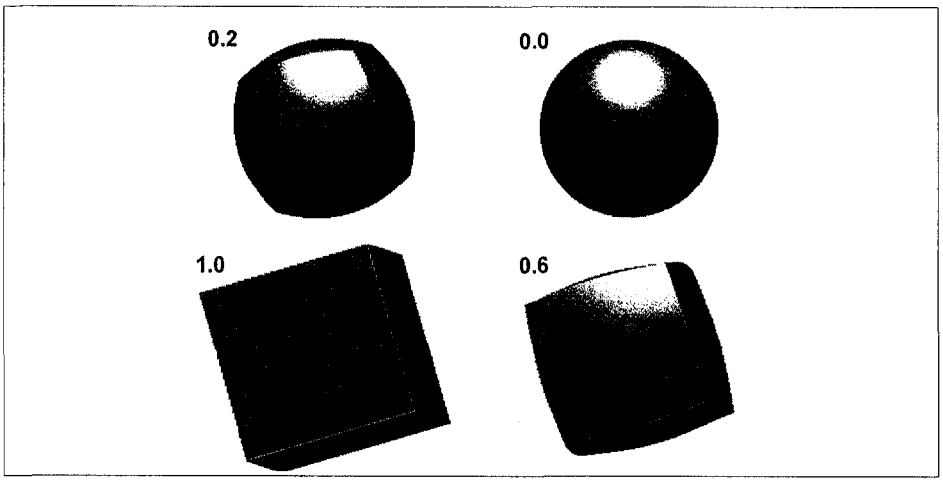


Figura 6-6: Ejemplo de morphing empleando vertex shaders

Visualización de funciones implícitas con programas de vértices

La idea es emplear el acelerador hardware de los programas de vértices (*Vertex Shader*) de la siguiente manera. Nosotros generamos una serie de vértices, una nube más o menos cercana a la superficie real del *metaBlob*, y buscamos la manera para que el *vertex shader* ajuste estos puntos a la superficie de forma precisa, lo cual se realizaría de forma rapidísima.

6 Líneas Futuras

6.3 Triángulos Curvos PN

Para mejorar la calidad visual en los objetos 3D representados mediante conjuntos de triángulos se puede emplear un modelo basado en triángulos llamado Triángulos Curvos PN (*point-normal*) [**VLAC01**].

Los triángulos curvos PN permiten mejorar la calidad visual mediante el suavizado de las aristas y proporcionando más puntos para las operaciones de sombreado basadas en vértices. Cada triángulo PN reemplaza un triángulo plano por una forma curva que es retriangulada en un número programable de pequeños subtriángulos planos. Podemos emplear esta representación en un entorno hardware con recursos limitados de forma que los triángulos PN se pueden generar por la propia GPU sin ayuda del software, en general, es un método con un coste muy bajo para la calidad que es capaz de generar. En la siguiente figura se ilustra un ejemplo en el cual se emplean triángulos PN.

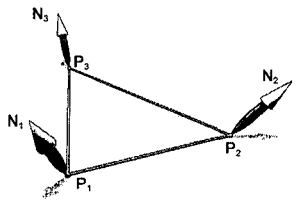
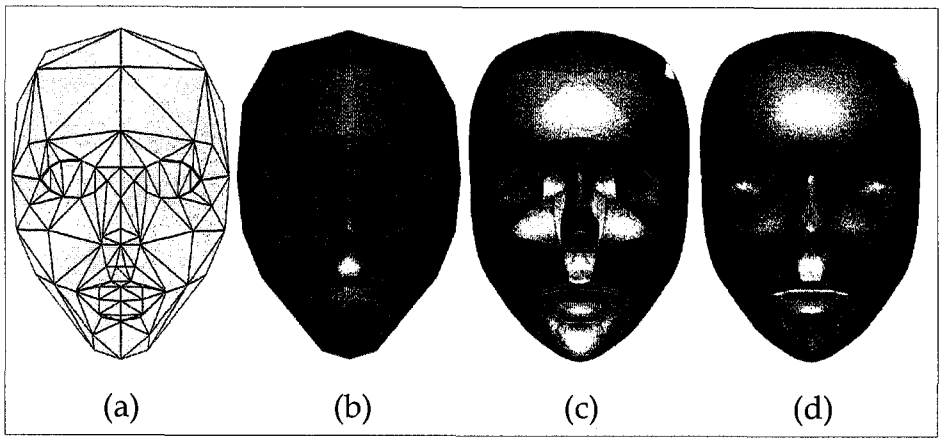


Figura 6-7: Ejemplo de empleo de triángulos curvos PN

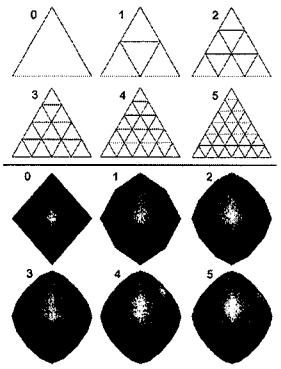


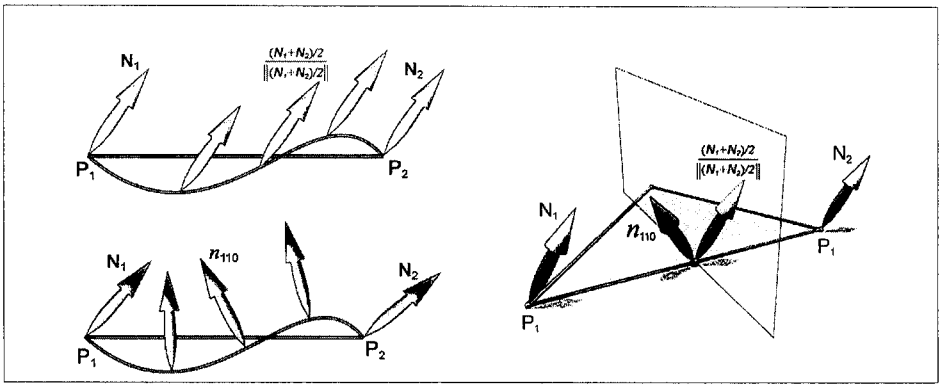
De izquierda a derecha: (a) malla de triángulos del modelo original, (b) modelo iluminado por interpolación de Gouraud, (c) componentes geométricos de los triángulos PN (sombreados acorde a la variación de la normal en la superficie) y (d) triángulos curvos PN (sombreados con normales con variación cuadrática).

Volver al índice/Tornar a l'índex

La geometría de un triángulo PN se define como un parche cúbico de Bézier, donde cada una de las subdivisiones va generando una geometría que varía de forma cúbica y normales con variación cuadrática [**CHIY83**], [**GREG74**], [**NIEL87**], en la siguiente figura aparece un ejemplo de este proceso.

Figura 6-8: Cálculo cuadrático de las normales en triángulos PN





En términos de rendimiento si el procesador es lo suficientemente rápido y el ancho del bus no se satura, los triángulos PN se visualizan a la misma velocidad, prácticamente, que los triángulos planos consiguiéndose una mejora de detalle sin embargo, mucho mayor, como se puede observar en las figuras de los ejemplos.

Adaptación a nuestro modelo

La adaptación a nuestro modelo sería inmediata, tan sólo tenemos que generar una malla poligonal con menos resolución y a partir de ella generar un modelo geométrico empleando triángulos curvos PN, que al realizarse mediante hardware 3D, se aceleraría y tendría una calidad visual mayor. Así conseguiríamos una mejora de rendimiento, al reducir el número de vértices generados y mantendríamos o incluso superaríamos la calidad obtenida con mayor número de vértices empleando una malla poligonal plana.

6 Líneas Futuras

Bibliografía

[BARR81] Barr, A.H.; *"Supercuadrics and Angle-Preserving Transformations"*, IEEE Computer Graphics and Applications. Vol. 1, No. 1, pp. 11-23, 1981.

[BEZI93] Pierre E. Bézier, The First Years of CAD/CAM and the UNISURF CAD System, in Fundamental Developments of Computer-Aided Geometric Modeling, edited by Les Piegl, Academic Press, 1993, pp. 13-26.

[BLIN82] Blinn, J.F.: *"A Generalization of Algebraic Surface Drawing"*, ACM Trans. on Graphics, Vol 1, No. 3, pp. 235-256, 1982.

[BLOO88] J. Bloomenthal. *"Polygonisation of Implicit Surfaces. Computer-Aided Geometric Design"*, 341-355, November 1988

[BOOR96] de Boor, Carl. *"B-spline basics"*. MRC 2952, 1986

[BORO95] H.Borouchaki, F. Hecht, E. Saltel and P.L. George *"Reasonably efficent Delaunay based mesh generator in 3 dimensions"* INRIA Report 1995.

[BOUR97] Bourke, P. *"Polygonising a scalar field using Tetrahedrons"*. May 1997.

[CAST93] Paul de Casteljau, *"Polar Forms for Curve and Surface Modeling as Used at Citroen, in Fundamental Developments of Computer-Aided Geometric Modeling"*, edited by Les Piegl, Academic Press, 1993, pp. 1-12.

[**CHIY83**] Chiyokura, H. and Kimura, F. *"Design of solids with free-form surfaces"*. In *SIGGRAPH 83 Conference Proceedings*, volume 17(3), pages 289–298, July 1983.

[**CIGB98**] P. Cignoni, C. Montani, and R. Scopigno. Dewall: *"A Fast Divide & Conquer Delaunay Triangulation Algoritm in E^d"* .Computer-Aided Design, Elsevier Science, 30(5):-341, April 1998.

[**CLIN88**] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. *"Two Algorithms for the Three-Dimensional Construction of Tomograms. Medical Physics"*, 15:320-327, 1988.
http://astronomy.swin.edu.au/pbourke/modelling/polygonise/

[**COUG96**] H.L. de Cougny and M.S. Shephard, *"Surface meshing using vertex insertion"*. Scientific Computation Research Center 1996.

[**COX72**] Cox, M.G.: *"The numerical evaluation of B-splines"*. Jour. Inst. Math. Applic., Vol. 10, pp. 134-149, 1972.

[**GERG74**] Gregory, J.A. *"Smooth interpolation without twist constraints"*, pages 71–88. Academic Press, 1974.

[**DOI91**] A.Doi, A.Koide. *"An Efficient Method of Triangulating Equivalued Surfaces by using Tetrahedral Cells"*. IEICE Transactions Commun, Elec. Info. Syst, E74(1) 214-224, January 1991.

[**GORD74**] W. J. Gordon and R. Riesenfield, B-Spline Curves and Surfaces, CAGD, Academic Press, 1974.

[**HEWE92**] Thomas T. Hewett; Ronald Baecker; et al. *"Curricula for Human-Computer Interaction. ACM Special Interest Group on Computer-Human Interaction Curriculum Development Group"*. Association for Computing Machinery, 1992.

[**JAIN95**] R. C. Jain, R. Kasturi, and B. G. Schunk, *"Introduction to Machine Vision"*. McGraw-Hill, New York, 1995, pp. 115-126. Analysis and Machine Intelligence, vol. PAMI-8, no. 2, March 1986, pp. 147-161.

[**LORE87**] Lorensen, W.E. and Cline, H.E. *"Marching Cubes: A High Resolution 3D Surface Construction Algorithm"*, ACM Computer Graphics, Vol 21, No. 24, pp. 163-169, Julio 1987.

[**MENO96**] J. Menon, *"An Introduction to Implicit Techniques"*, SIGGRAPH Course Notes on Implicit Surfaces for Geometric Modeling and Computer Graphics, 1996.

[**MURA95**] Muraki, S.; *"Volumetric Shape Description of Range Data Using 'metaBlobby Model' "*, Computer Graphics, SIGGRAPH '91. pp 227-235. 28 Julio-2 Agosto 1991.

[**NIEL87**] Nielson, G. *"A transfinite, visually continuous, triangular interpolant"*, pages 235–246. SIAM, 1987.

[**NISH85**] Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I. and Omura, K., *"Object Modeling by distribution Function and a Method of Image Generation"*, Trans.IEICE Japan, Vol J68-D, No. 4, pp. 718-725.

[**NOWA95**] H. Nowacki, M. Bloor and B. Oleksiewicz, Computational Geometry for Ships, Singapore: World Scientific Publishing Co., 1995, pp. 58-60.

[**NVIDIA**] Kilgard, M.J. *"A User-Programmable Vertex Engine"*. NVIDIA Corporation, 2001.

[**PAYN90**] B.A. Payne and A.W. Toga, *"Surface mapping brain function on 3D models"*, IEEE Computer Graphics and Applications, pp 33-41 1990.

[**PUCH97**] Puchol García, J.A.; Molina Carmona, R.; García Quintana, C. *"Conversión Paramétrica de metaBlobs a Modelo de Fronteras en Tiempo Real"*. IX Congreso Internacional de Ingeniería Gráfica. Bilbao - San Sebastián, Junio 1997.

[**PUCH99**] Puchol García J.A.; Sáez Martínez J.M.; Molina Carmona R. *"Postproceso de Rectificación de Metapolígonos"*. IX Congreso Español de Informática Gráfica - Jaén Junio 1999.

[**RASK00**] Raskin, J. *"La Interfaz Humana (compasiva). Nuevas direcciones en el desarrollo de sistemas interactivos"*. ACM Press - Addison Wesley, 2000.

[**REQU80**] Requicha, A.A.G. and Rossignac, J.R.; *"Solid Modeling and Beyond"*, IEEE Computer Graphics and Applications. pp 31-44. 1992.

[**SAEZ99**] Sáez Martínez J.M.; Puchol García J.A.; Molina Carmona R. *"Poligonalización de Blobs con Rectificación de Precisión"*. IX Congreso Español de Informática Gráfica, CEIG 1999-Jaén, Junio 1999.

[**SAEZ00**] Sáez Martínez J.M.; Puchol García J.A.; Molina Carmona R. *"Aplicación del PRM sobre superficies libres. Aplicación a contornos CAD/CAM"*. X Congreso Español de Informática Gráfica, CEIG 2000-Castellón, Junio 2000.

[**VLAC01**] Vlachos, A., Peters, J., Boyd, Chas., Mitchell, J.L. *"Curved PN Triangles"*ATI, Inc. 2001.

[**WYVI86**] Wyvill, G., McPhetters, C., and Wyvill, B. *"Data Structure for Soft Objects"*, The Visual Computer, Vol. 2, pp. 227-234, 1986.

Apéndice A

Como complemento a esta tesis presentamos un apéndice con detalles técnicos, donde se explican los aspectos más relevantes de la implementación realizada en el desarrollo del prototipo de experimentación. El cual se ha empleado para elaborar, perfeccionar y experimentar los algoritmos descritos en los capítulos anteriores. Así como, una breve introducción a la biblioteca 3D empleada, OpenGL.

A.1 Implementación

Para la elaboración del prototipo empleado en la implementación de todos los algoritmos expuestos en esta tesis hemos trabajado con la siguiente plataforma.

Figura A-1: Características de la plataforma de experimentación

Sistema Operativo	*Windows XP*
CPU	*Athlon 1000MHz*
RAM	*512 MB*
Disco Duro	*40GB*
Tarjeta Gráfica	*GeForce 3*
RAM	*32 MB*
Frecuencia (GPU/RAM)	*Overclock 240/540*
BUS	*AGP*
Software	*Driver NVidia Detonator XP 6.22.80*
	OpenGL ICD 1.3

Como herramienta de desarrollo nos decantamos por el Borland Builder C++ v5.0, hemos seleccionado este entorno de desarrollo por ser un entorno de programación visual, lo cual facilita un prototipado rápido y sencillo. Genera un código bastante óptimo y además emplea como lenguaje de programación el lenguaje C++, por ser OpenGL un API de programación 3D basada en C, nos permite una perfecta integración con el C++. Este entorno se ha divido en dos zonas principales, el panel de visualización y el panel de control.

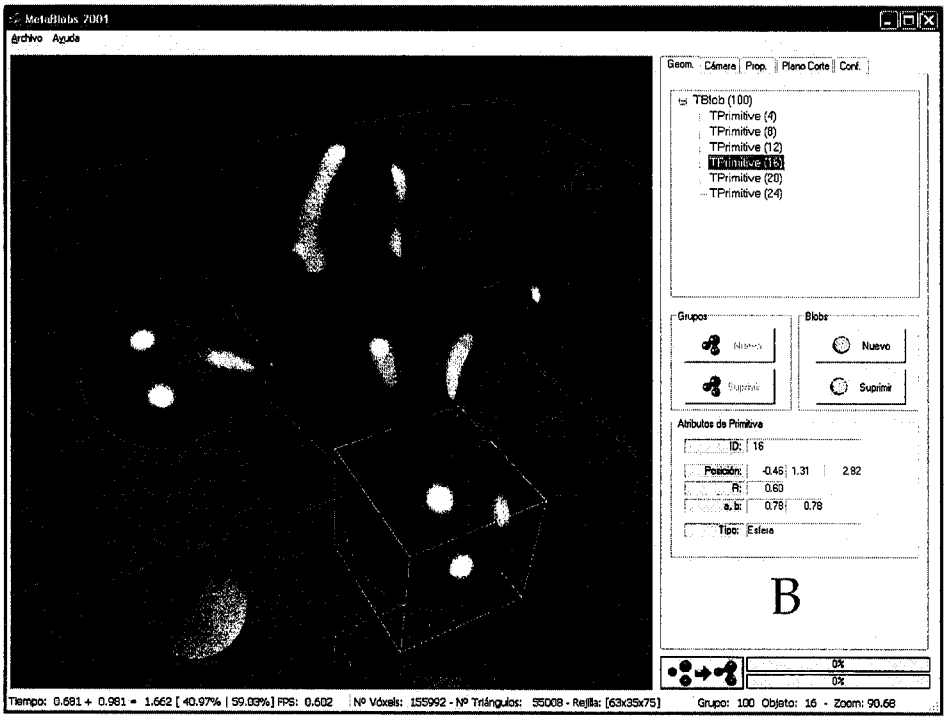
La librería empleada para la programación 3D ha sido OpenGL, cuyas estructuras más importantes se describen en el apéndice B.

Las clases de objetos desarrolladas son las siguientes:

Figura A-2: Clases implementadas en el prototipo

Nombre de la clase	*Descripción y cometidos*
TVertex	*Mantiene la geometría de un vétice 3D (x, y, z)*
TVector	*Mantiene la información de un vector (a, b, c)*
TVoxel	*Mantiene la información necesaria para cada vóxel.* *Ocho vértices.* *Ocho valores de campo (escalares).* *Un índice, que mantiene la información de la intersección de las aristas con la superficie.*
TVoxelList	*Responsable del mantenimiento de las listas de vóxels.*
TPrimitive	*Mantiene la información básica de las primitivas que forman un metablob.* *Posición (x,y,z).* *Fuerza y suavidad (a, b).*
TBlobGroup	*Mantiene la información de una lista de primitivas que forman un único metablob.*
TScene	*Mantiene una lista (de TBlobGroup) para componer la escena completa formada por todos los metablobs.*

Figura A-3:
Aspecto general del entorno desarrollado



A) Zona de visualización de OpenGL.

B) Controles del entorno. Se encuentra subdividido en 5 categorías:

- Geometría.
- Cámara y visualización.
- Algoritmo y parámetros.
- Corte de planos.
- Experimentos.

A.2 OpenGL

En este segundo apartado se explican las estructuras más importantes empleadas de la biblioteca *OpenGL*.

OpenGL es un API de programación para la implementación de aplicaciones gráficas 3D que desarrolló Silicon Graphics, Inc. en 1992.

OpenGL es un estándar multiplataforma para visualización 3D y aceleración mediante hardware 3D. Existen versiones de la biblioteca de desarrollo OpenGL, o compatibles, para sistemas operativos Windows, MacOS, Linux y Unix entre otros. Entre sus principales características destacamos:

1. **Hardware 3D**. OpenGL nos permite de forma sencilla acceder a todo el potencial del hardware 3D consiguiendo aplicaciones rápidas y potentes. Las aplicaciones 3D actuales requieren la manipulación de enormes cantidades de información en tiempo real, lo cual es posible mediante el uso de la geometría acelerada por hardware de OpenGL, así como iluminación, recortado, transformaciones y su visualización.

2. **Efectos 3D en tiempo real**. La aceleración por hardware de OpenGL permite añadir detalles y efectos especiales a las imágenes sin comprometer el rendimiento. Por ejemplo, niebla en tiempo real, antialiasing, sombras, relieve, efectos de movimiento, transparencias, reflexiones, texturas 3D, visualización de volúmenes y algunos efectos más.

3. **Extensibilidad de OpenGL**. El mecanismo de extensión que incorpora OpenGL permite ir incorporándole los últimos avances en hardware y software 3D que no existía cuando OpenGL fue concebida originalmente. De esta forma se garantiza el mejor rendimiento para las aplicaciones 3D a medida que la

tecnología hace que el hardware incorpore nuevos avances.

4. **Multiplataforma**. Otra ventaja muy importante es la existencia de versiones de OpenGL en, prácticamente, todos los sistemas operativos utilizados hoy en día. De esta forma transportar una aplicación 3D de una plataforma a otra se convierte en una tarea relativamente sencilla.

5. **Estabilidad**. Las estaciones de trabajo y supercomputadores más potentes en hardware 3D se han beneficiado del uso de OpenGL desde 1992. Lo cual garantiza que es un sistema muy depurado que permite una potente solución de desarrollo para aplicaciones 3D tanto para entornos profesionales como domésticos

Hardware 3D

En el desarrollo empleado hemos utilizado fundamentalmente el empleo de las listas de visualización, *Display Lists*, que son capaces de almacenar la geometría que deseemos, dentro de las estructuras que nos permite el API de OpenGL, y mediante la asignación de un nombre a la misma, somos capaces de visualizar un objeto sin tener que viajar la información geométrica asociada al objeto que queremos representar cada vez por el bus, que comunica la CPU con la GPU.

La unidad básica para OpenGL es el vértice, toda la información geométrica se construye mediante la especificación de puntos 3D con a 4-tupla (x, y, z, w), siendo el valor w muchas veces empleado como canal *alfa*, o canal de transparencia. Por otro lado, también debemos emplear otra estructura vectorial para construir las normales en cada vértice, esta estructura es una 4-tupla de la forma (a, b, c).

La geometría que nos permite emplear OpenGL se detalla en la siguiente tabla, siendo n el número de vértices creados.

Figura A-4: Estructuras geométricas en OpenGL

Estructura OpenGL	*Descripción*
GL_POINTS	*Trata cada vértice como un punto independiente*
GL_LINES	*Trata cada par de vértices como un segmento de línea independiente*
GL_LINE_STRIP	*Dibuja un grupo conectado de segmentos de línea. Los vértices 2n-1 y 2n definen la línea n. Por tanto se dibujan n/2 líneas.*
GL_LINE_LOOP	*Dibuja un grupo conectado de segmentos de líneas, del primer vértice al último, uniendo el último con el primero. Los vértices n y n+1 definen la línea n, sin embargo la última línea la definen los vértices n y 1. Por tanto se dibujan n líneas.*
GL_TRIANGLES	*Cada tripleta de vértices define un triángulo. Los vértice 3n-2, 3n-1, 3n definen el triángulo n. Por tanto se dibujan n/3 triángulos.*
GL_TRIANGLE_STRIP	*Dibuja un grupo conectado de triángulos. Un triángulo es definido por cada vértice que se añade tras los dos primeros. Si n es par, los vértices n, n+1, y n+2 definen el triángulo n. Si n es impar, los vértices n+1, n, n+2, definen el triángulo n. Se dibujan n-2 triángulos.*
GL_TRIANGLE_FAN	*Dibuja un grupo conectado de triángulos. Un triángulo es definido por cada vértice que se añade tras los dos primeros. Los vértices n, n+1, y n+2 definen el triángulo n. Se dibujan n-2 triángulos.*
GL_QUAD	*Similar a GL_TRIANGLES pero cada 4 vértices definen un cuadirlátero. Se dibujan n/4 cuadriláteros.*
GL_QUAD_STRIP	*Similar a GL_QUAD_STRIP. Se dibujan n cuadriláteros.*
GL_POLYGON	*Dibuja un único polígono. Los vértices 1 a n definen este polígono.*

El esquema que emplea OpenGL para las normales es el siguiente: en primer lugar, las normales van asociadas a los vértices, como unidad geométrica básica de OpenGL, en segundo lugar, es la aplicación la que debe generar dicha información, y en tercer lugar, mientras la información dela normal actual definida no se cambia, se va copiando en cada nuevo vértice que vamos creando. Por tanto si queremos dar una normal diferente en cada vértice debemos llamar a la función *glNormal* tantan veces como a la función *glVertex*.

El esquema completo para la creación de un objeto 3D con una cierta estructura geométrica 3D sería el siguiente:

```
glNewList(identificador, GL_COMPILE);
   glBegin(estructura OpenGL);
      glNormal(n_i);
      glVertex(v_i);
      ...
   glEnd();
glEndList();
```

Características empleadas de OpenGL

Los aspectos que hemos empleado de OpenGL has sido los siguientes:

- Selección 3D de objetos.
- Transformaciones geométricas 3D. Escalado, Rotación y Traslación
- Visualización mediante listas de visualización, *Display Lists*.
- Empleo de backface-culling. Eliminación de las caras opuestas al observador.
- Empleo de transparencias.

Indice de Materias

S

T

V

UNIVERSIDAD DE ALICANTE

Comisión de Doctorado

Reunido el Tribunal que suscribe en el día de la fecha acordó otorgar, por unanimidad Tesis Doctoral de Don/Dña. D Juan Antonio Puchol García la calificación de Sobresaliente Cum laude.

Alicante 17 de DICIEMBRE de 2001

El Secretario,

El Presidente,

FERNANDO MARTÍN

Mª JESÚS CASTEL