Detección de Colisiones mediante Recubrimientos Simpliciales

Tesis Doctoral

Departamento de Lenguajes y Sistemas Informáticos

Universidad de Granada

Granada, Septiembre de 2006

Autor

D. Juan José Jiménez Delgado

Directores

D. Francisco R. Feito Higueruela

D. Rafael J. Segura Sánchez

Departamento de Informática

Universidad de Jaén

Editor: Editorial de la Universidad de Granada
Autor: Juan José Jiménez Delgado
D.L.: Gr. 2161 - 2006
ISBN: 84-338-4137-8

La memoria titulada ***Detección de Colisiones mediante Recubrimientos Simpliciales***, que presenta D. Juan José Jiménez Delgado para optar al grado de Doctor en Informática, ha sido realizada en el Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada bajo la dirección de los Doctores D. Francisco R. Feito Higueruela y D. Rafael J. Segura Sánchez.

Granada, Septiembre de 2006

El doctorando

Juan José Jiménez Delgado

Los directores

Francisco R. Feito Higueruela

Rafael J. Segura Sánchez

A mi esposa y familia por su apoyo incondicional,

a Francisco Feito y Rafael Segura por sus consejos y ayuda,

a Robert Joan, Juan Carlos Torres y Lidia Ortega por sus sugerencias y su revisión exhaustiva,

y a mis compañeros del Departamento de Informática, que me han ayudado en los momentos difíciles.

Índice

Contenidos

Definiciones

Teoremas, lemas y corolarios

Propiedades

Heurísticas

Figuras

Tablas

Algoritmos

CAPÍTULO

1

Introducción

En este capítulo realizamos una breve descripción del contenido de esta memoria. Se dará una reseña de lo distintos capítulos que la componen, así como un resumen de la notación utilizada en la misma.

1. Introducción

Conseguir que unos modelos geométricos con una representación gráfica conozcan las posiciones relativas de unos respecto de otros es bastante costoso en términos computacionales. En aplicaciones de tiempo real es necesario que se detecten *colisiones* entre objetos en una pequeña fracción de tiempo, siendo esta tarea normalmente compartida con otros aspectos, como la visualización, o con determinados cálculos asociados a la aplicación, cálculos que deben ser realizados entre *frames*[*].

La disciplina llamada *Detección de Colisiones (DC)* se encarga de obtener en algunos casos una medida de la cercanía de esos modelos, y en otros de determinar simplemente si dos o más de estos se encuentran en colisión. Esta disciplina puede contemplar también la llamada *Respuesta a Colisión* que se encarga de modificar las trayectorias de los objetos o la deformación de estos. También suele darse respuesta a una serie de cuestiones sobre la colisión, como cuáles son las *características*[†] de los modelos implicadas en la colisión, o la distancia mínima y la dirección para separar dos objetos que intersecan en el espacio.

Es habitual que la representación de estos objetos implique la utilización de un método de detección de colisión u otro. Es más, son pocos los métodos apropiados para objetos no convexos y en la mayor parte de los casos es necesario, o bien descomponer el objeto en piezas convexas, o bien utilizar una descomposición espacial que clasifique las características que componen el objeto, posiblemente utilizando determinadas jerarquías de volúmenes envolventes para llevar a cabo esta descomposición.

---

[*] Utilizaremos éste término preferiblemente al de fotograma por ser de amplio uso en Informática Gráfica.

[†] Suele utilizarse el término *feature* en inglés para referirse a un vértice, una arista o una cara.

En este trabajo utilizamos un esquema de representación de objetos desarrollado por Feito [Fei95] y Segura [Seg01] denominado *Recubrimiento Simplicial*. Este esquema de representación es apropiado para objetos formados por caras planas que llamaremos complejos, pues pueden ser objetos no convexos, con agujeros en su interior, o incluso no-variedad. Usando esta representación hemos desarrollado algoritmos para determinar si se produce o no colisión entre objetos tanto en dos como en tres dimensiones, siendo posible además la determinación de los elementos implicados en la colisión.

Los algoritmos desarrollados utilizan la coherencia temporal y espacial para acelerar los cálculos implicados, y así diferenciar estos algoritmos de simples algoritmos de inclusión o de intersección:

- Por una parte hemos desarrollado un nuevo tipo de descomposición espacial que hemos llamado árbol de *tri-conos* en 2D o árbol de *tetra-conos* en 3D (*tri-tree* y *tetra-tree* respectivamente). Mediante este tipo de descomposición espacial clasificamos las características de los objetos para disminuir así el número de las mismas a utilizar en la detección de colisión, aprovechando en cierta medida la coherencia espacial.

- Por otra parte, aprovechamos ciertos cálculos realizados en frames anteriores para tratar, en primer lugar, las partes de los objetos con mayor probabilidad de colisión, es decir, aprovechamos el grado de coherencia temporal presente.

En algunas situaciones realizamos un cálculo incremental, y en otras utilizamos nuevos algoritmos especialmente diseñados para la detección de colisión, como un nuevo algoritmo de intersección segmento/triángulo especialmente diseñado para la detección de colisión. Todos los algoritmos diseñados son bastante robustos y eficientes debido a la naturaleza de los cálculos realizados y a la ausencia de casos especiales complejos.

El trabajo desarrollado en esta memoria queda abierto a numerosas posibilidades de extensión; podríamos decir que es el punto de inicio de nuevos métodos de detección de colisión entre objetos complejos. Entre estas extensiones podemos destacar la optimización de los algoritmos desarrollados, o la aplicación de nuevas técnicas que aprovechen en mayor medida la coherencia temporal y geométrica. La descomposición espacial desarrollada puede utilizarse en distintas disciplinas, como el propio modelado geométrico, la representación de terrenos, etc.; además pensamos que puede utilizarse eficientemente para modelos deformables. Entre otras posibles extensiones, podríamos considerar la adaptación de los algoritmos desarrollados para realizar una implementación basada en la utilización de la GPU

mediante una programación de propósito general (GPGPU*), siendo posible aprovechar el paralelismo de éste tipo de implementación debido a la naturaleza de los cálculos que se realizan.

Hemos estructurado esta memoria en 7 capítulos, incluyendo este capítulo de introducción. A continuación pasamos a describir brevemente el contenido de los mismos:

En el Capítulo 2 definimos el término *Detección de Colisión*, y la parte de la misma que vamos a tratar en ésta memoria. Definimos conceptos importantes como la *Eficiencia*, *Coherencia* y *Robustez*, necesarios todos ellos para el desarrollo adecuado de los distintos métodos de detección de colisión incluidos en la memoria. Repasamos las técnicas más utilizadas, como el uso de volúmenes y jerarquías de volúmenes envolventes, así como algunos esquemas de descomposición espacial. También se resumen los métodos más importantes relacionados con la detección de colisión, seguidos de muchas de las posibles aplicaciones y de las librerías más importantes que se están utilizando en esta disciplina.

El Capítulo 3 está dedicado a la representación de objetos. Una vez definido el tipo de objetos con los que vamos a trabajar, presentamos el modelo que los describe mediante el *Álgebra de Objetos Gráficos* de Torres [Tor92] así como el esquema de representación de sólidos mediante *Recubrimientos Simpliciales* de Feito [Fei95] y Segura [Seg01], tanto para objetos 2D como 3D. Veremos algunas de las propiedades de este esquema de representación y cómo se ha utilizado en esta memoria.

En el Capítulo 4 describimos las técnicas y métodos utilizados en la detección de colisión entre objetos bidimensionales, es decir, entre polígonos. Los conceptos desarrollados en este capítulo serán la base para el desarrollo de algoritmos y de una teoría sobre detección de colisión en tres dimensiones. Se describe un nuevo tipo de descomposición espacial basado en *tri-conos*, que hemos llamado *tri-tree*, que se ha utilizado para simplificar el número de primitivas a tratar. El capítulo se fundamenta en el desarrollo de algoritmos de detección de colisión entre diversos tipos de entidades geométricas como son entre *punto/polígono*, *circunferencia/polígono* y *polígono/polígono*. Para todos estos casos se establecen los principios generales que permiten, por una parte, aprovechar la coherencia temporal y espacial para el desarrollo de estos algoritmos, y por otra, aprovechar las características de algoritmos de inclusión, intersección y detección de colisión más sencillos para el desarrollo de nuevos algoritmos de detección de colisión entre objetos más complejos.

* General Purpose programming with the GPU – Programación de propósito general con la GPU.

El Capítulo 5 comprende el desarrollo de principios, técnicas y algoritmos de detección de colisión en tres dimensiones, basándose en los conceptos explicados en el capítulo anterior. Se utiliza una descomposición espacial denominada *Tetra-Tree*, basada en *Tetra-Conos,* así como la coherencia espacial y temporal para el desarrollo de algoritmos de detección de colisión entre *puntos*, *esferas* y *poliedros* en relación a *poliedros*. Al igual que en el caso 2D se utilizan algoritmos de inclusión y de interferencia basados en algoritmos completamente nuevos para obtener la detección de colisión, como es el caso del nuevo algoritmo de intersección *segmento/triángulo* desarrollado para este fin.

En el Capítulo 6 se describe la utilización del *hardware gráfico* para la implementación de algoritmos de interferencia entre objetos y en particular para el caso de detección de colisión entre una nube de partículas y un poliedro. Se establece un método de representación de objetos utilizando texturas y cómo cambiar el planteamiento de algoritmos de detección de colisión para adaptarlos a las características particulares de la GPU, que está especializada en la visualización. Muchos de los conceptos necesarios para comprender este capítulo se han resumido en un Apéndice. En este apéndice se realiza una descripción tanto de la arquitectura de las tarjetas gráficas de última generación, como de la posibilidad de utilización del Hardware Gráfico para la programación de problemas de propósito general. Se describe también brevemente un lenguaje de programación especializado denominado Cg, que es el utilizado en la implementación de los algoritmos expuestos en el Capítulo 6.

En los Capítulos 4, 5 y 6 se ha realizado un estudio temporal que muestra la eficacia de la implementación realizada de los distintos algoritmos. En este estudio se han variado muchos de los posibles parámetros implicados en la detección de colisión, y se ha comparado los métodos desarrollados con métodos clásicos basados en quadtrees y octrees.

Finalmente en el Capítulo 7 se hace un resumen de las principales aportaciones realizadas en cuanto a detección de colisiones, así como las futuras ampliaciones y mejoras que pueden llevarse a cabo en relación al trabajo presentado en esta memoria.

Notación

Símbolo	Elemento
P, V_i	Punto, vértice
PQ, V_iV_j	Segmento, arista
T, S, PQR, $V_iV_jV_k$	Triángulo ó 2-símplice
T, S, $PQRS$, $V_iV_jV_kV_l$	Tetraedro ó 3-símplice
O, V_o	Centroide, origen del recubrimiento
H	Cara
F	Polígono, poliedro, figura, malla, objeto
C, C_H, C_F, $C_{\angle Ti}$	Recubrimiento, de una cara, de un objeto, clasificado en un tetra,tri-cono
$(\alpha,\beta,\gamma[,\delta])$	Coordenadas baricéntricas respecto a un 2,3-símplice
$(\alpha_i(P),\beta_i(P),\gamma_i(P)[,\delta_i(P)])$	Coordenadas baricéntrica de P respecto al 2,3-símplice i
ε	Acotación de error
$\lvert ABC \rvert$, $\lvert ABCD \rvert$	Determinante de un 2,3-símplice
$\angle$T	Tri-cono o tetra-cono asociado a un 2,3-símplice
$\angle\backslash$T	Tri-cono o tetra-cono truncado
TT, TT_F	Tri-Tree o Tetra-Tree, de un objeto
$[\ldots]_i$	Instante de tiempo i, frame i
[... *in* ...]	Inclusión
[*not* (... *in* ...)]	No inclusión
[T *clasif* ...]	Triángulo o tetraedro clasificado en ...
T_{env}	Triángulo o tetraedro envolvente
$T_{env\text{-}reg}$	Triángulo o tetraedro envolvente que forma parte de una envolvente regular
TT_{env}	Conjunto de triángulos o tetraedros envolventes de un nivel de un tri-tree o tetra-tree
$TT_{env\text{-}reg}$	Envolvente regular
E, E(C,r)	Circunferencia o esfera, de centro C y radio r
E_{env}	Circunferencia o esfera envolvente
Δ	Incremento
$offset^{\pm}(T_i,r)$, $offset^{\pm}_i(r)$	Offset positivo-negativo de tamaño r de un 2,3-símplice
$AI(T_i,r)$, $AI_i(r)$	Área de influencia de tamaño r de un 2,3-símplice
$AIE(T_i,r)$, $AIE_i(r)$	Área de inf. extendida de tamaño r de un 2,3-símplice
$AIEL(T_i,r)$, $AIEL_i(r)$	Área de inf. ext. limitada de tamaño r de un 2,3-símplice
$Ext(T_i,r)$, $Ext_i(r)$	Extensión en r unidades de un 2,3-símplice
$Ext(\angle T_i,r)$, $Ext_{\angle Ti}(r)$	Extensión en r unidades de un tri,tetra-cono
sign(...), signo(...)	Función signo

CAPÍTULO

2

Antecedentes de la Detección de Colisiones

En este capítulo se define el concepto de Detección de Colisiones así como los métodos y técnicas utilizados habitualmente en este campo. También se hace un estudio de las posibles aplicaciones, así como una breve descripción de las librerías que implementan algunos de los métodos de Detección de Colisión más importantes.

CONTENIDOS:

1. **Definición, dominio del problema**
2. **Conceptos asociados a la detección de colisión**
3. **Técnicas básicas usadas en detección de colisión**
4. **Métodos para objetos convexos**
5. **Aplicaciones de la detección de colisión**
6. **Librerías específicas para la detección de colisión**

2. Antecedentes de la Detección de Colisiones

En este capítulo vamos a delimitar el problema tratado en esta memoria. Definimos por una parte el concepto de detección de colisión y por otra qué aspectos de la misma vamos a considerar. Asimismo tendremos en cuenta ciertos aspectos de eficiencia y robustez, debido sobre todo a las peculiaridades en cuanto a complejidad y tiempo de respuesta esperados en aplicaciones de detección de colisión. Por este motivo repasaremos ciertas consideraciones generales sobre la eficiencia y robustez de los algoritmos.

Llevaremos a cabo una revisión de las técnicas más utilizadas en detección de colisiones. Estas técnicas comprenden desde el uso de volúmenes y jerarquías de volúmenes envolventes, a distintos tipos de descomposiciones espaciales; todas ellas técnicas que ayudan a la hora de acelerar los cálculos y tratar con un menor número de primitivas. Tras esta revisión, describiremos algunos de los métodos más conocidos y eficientes para la detección de colisión, generalmente para piezas convexas. Finalmente enumeraremos las aplicaciones más importantes relacionadas con la detección de colisiones, así como algunas de las librerías que hay disponibles para el desarrollo de este tipo de aplicaciones.

2.1. Definición, dominio del problema

Una *colisión* es una configuración en la que dos objetos ocupan parte de una misma porción del espacio al mismo tiempo. Una colisión se produce como resultado del movimiento de los objetos; cuando esto ocurre con dos objetos estáticos lo denominaremos interferencia entre los dos objetos.

Hablaremos de *movimiento de los objetos* como un flujo continuo de configuraciones entre objetos. Una animación simula este flujo continuo de configuraciones entre objetos actualizando el estado entre éstos en momentos discretos o frames. A pesar de que no se detecte colisión en dichos frames, podemos averiguar si se ha producido o no colisión en el tiempo comprendido entre dichos momentos basándonos en las trayectorias de los objetos.

En aplicaciones interactivas es bastante costoso encontrar estas colisiones entre frames, sobre todo por las restricciones de tiempo a las que se ven sometidas. En este tipo de aplicaciones deben enviarse al procesador gráfico entre 30 y 60 frames por segundo, y parte de este tiempo debe estar dedicado a la detección de colisiones.

Si consideramos objetos formados por caras planas, el desarrollo de un modelo matemático que encuentre estas colisiones entre objetos no es difícil. El principal problema es encontrar algoritmos de detección de colisión para entornos complejos en los que se encuentran gran cantidad de objetos en movimiento, y que se realice dicha detección de colisiones en tiempo real.

Para poder cumplir estas especificaciones relacionadas con aplicaciones interactivas podemos utilizar algunas propiedades del movimiento de los objetos, como son la coherencia espacial y temporal. Además es posible reducir la complejidad de los objetos descomponiéndolos o utilizando jerarquías de volúmenes envolventes.

Sin embargo, el tiempo de procesamiento necesario para determinar la colisión entre objetos no es el único problema a resolver. En la mayor parte de los ordenadores, tanto la capacidad de almacenamiento como la precisión numérica están limitadas. En este sentido, pueden surgir problemas numéricos provenientes de la aritmética finita de los ordenadores que influirán significativamente en la corrección y robustez de los algoritmos.

2.1.1. Definición de Detección de Colisión

El término *Detección de Colisión* es parte del término general *Manipulación de la Colisión*, que se divide en tres partes [AH02]: *Detección de Colisión*, *Determinación de Colisión* y *Respuesta a la Colisión* (Figura 2.1). El resultado de la *Detección de Colisión (DC)* es un valor lógico que indica si dos o más objetos colisionan, mientras que la *Determinación de Colisión* hace referencia al cálculo de las intersecciones entre objetos; la *Respuesta a Colisión* señala las acciones a realizar en respuesta a la colisión entre objetos.

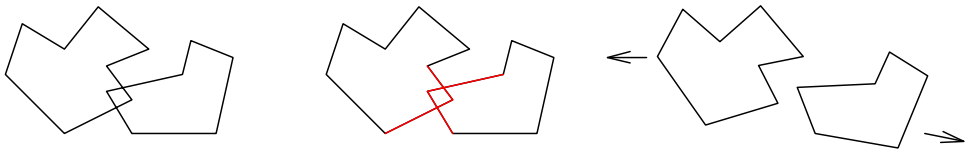


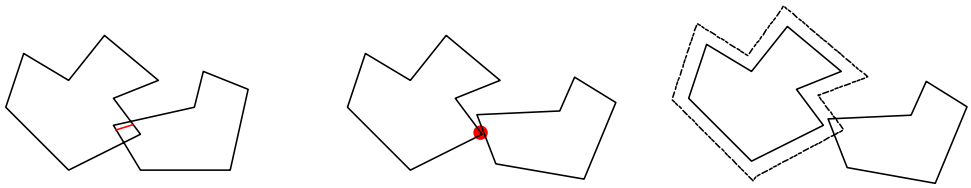
Figura 2.1: Detección, determinación y respuesta a colisión.

Estas tres etapas presentes en la *Manipulación de la Colisión* suelen realizarse en el orden dado. Entre estas etapas destacamos la *Respuesta a Colisión*, que puede ser bastante compleja. Una vez establecida la colisión entre objetos y obtenidas las intersecciones entre objetos, la respuesta a colisión puede actuar de diversas formas. Una de ellas consiste en modificar la trayectoria de los objetos implicados. Para ello es necesario conocer las trayectorias previas de los objetos y tener en cuenta las fuerzas aplicables entre éstos, como la gravedad, etc. [Ebe04]. Otro tipo de respuesta implica la deformación de los objetos, incluso la rotura de los mismos en partes [Ber04] [TKH*04]. Normalmente la respuesta es múltiple implicando diversas técnicas.

En este trabajo desarrollaremos algoritmos de Detección de Colisión para aplicaciones interactivas, aunque explicaremos cómo pueden extenderse para la Determinación de Colisión, no siendo objeto de esta memoria la llamada Respuesta a Colisión.

2.1.2. Otros tipos de cuestiones a resolver

Además de determinar simplemente si se produce o no colisión entre objetos podemos obtener otro tipo de información acerca de esa colisión, como por ejemplo las intersecciones entre objetos (que hemos llamado Determinación de Colisión). Dependiendo de la aplicación concreta, estas intersecciones pueden calcularse de forma aproximada, utilizando una cierta tolerancia o error; o bien de manera exacta, lo que implica normalmente una falta de robustez. Además puede ser interesante proporcionar otro tipo de información (Figura 2.2), como es:



a) Mín. dist. para separar objetos. b) Punto de contacto c) Colisión con tolerancia

Figura 2.2: Otros tipos de cuestiones a resolver.

- Una medida de la penetración entre objetos, entendiéndose como tal la mínima distancia para separar los objetos que se encuentran en colisión.
- Una medida de la distancia entre objetos, de manera que cuando la distancia es nula, se puede concluir que los objetos colisionan.
- El tiempo de impacto o colisión, que es el momento en el que dos objetos van a colisionar. Este tiempo puede utilizarse en el establecimiento de un incremento de tiempo en la simulación.
- Puede también ser necesario obtener el punto exacto de colisión y la normal en dicho punto de contacto.

2.1.3. Algoritmo básico

Veamos a continuación un algoritmo genérico de detección de colisión [JSF02] (Algoritmo 2.1). A través de este algoritmo deduciremos algunos de los problemas que presenta la detección de colisión. En la sección 2.3 presentamos diversas técnicas que permiten minimizar estos problemas.

El algoritmo que presentamos considera una simulación en la que coexisten *M* objetos estáticos y *N* objetos en movimiento e intenta obtener las colisiones entre objetos en el intervalo de tiempo $[t_i, t_f]$.

```
PARA t ← t_i HASTA t_f EN PASOS DE Δt
    PARA CADA Objeto_i ∈ {ObjetoDinam_1, ..., ObjetoDinam_N}
        MOVER Objeto_i A SU POSICIÓN EN EL TIEMPO t
        PARA CADA Objeto_j ∈ {ObjetoDinam_1, ..., ObjetoDinam_N}, j≠i
            MOVER Objeto_j A SU POSICIÓN EN EL TIEMPO t
            SI (la repres. geométrica de Objeto_i y Objeto_j interpenetran)
                ENTONCES hay colisión en el tiempo t
            FIN SI
        FIN PARA
        PARA CADA Objeto_k ∈ {ObjetoEstat_1, ..., ObjetoEstat_M}
            SI (la repres. geométrica de Objeto_i y Objeto_k interpenetran)
                ENTONCES hay colisión en el tiempo t
            FIN SI
        FIN PARA
    FIN PARA
FIN PARA
```

Algoritmo 2.1: Algoritmo básico de detección de colisión.

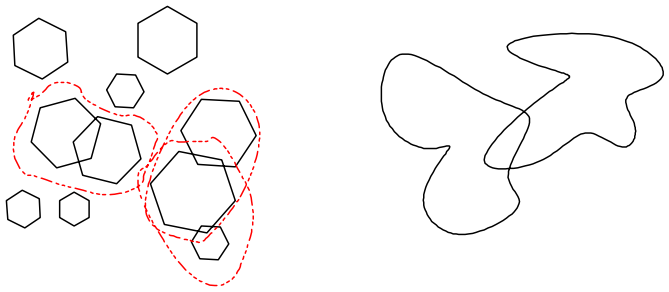
2.1.3.1. Problemas implícitos en la Detección de Colisión

Dependiendo de las características de la aplicación concreta a desarrollar surgen diversos problemas a la hora de aplicar un algoritmo de detección de colisión. Una vez mostrado el algoritmo básico podemos establecer algunos problemas inherentes a éste.

Problema relacionado con la comprobación de todos los pares de objetos

Por una parte, nos encontramos con el problema del número de objetos presentes en la aplicación. En el peor de los casos, un algoritmo de fuerza bruta debe comprobar cada uno de los objetos con todos los demás, necesitando un tiempo $O(N^2)$. En el algoritmo, este tiempo es $O(N \cdot M+N^2)$, es decir, se deben comprobar todos los pares de objetos en movimiento y todos los pares formados por un objeto estático y otro en movimiento. Para reducir este tiempo cuadrático debemos disminuir el número de pares de objetos sobre los que realizar la detección de colisión. Para ello, esta tarea se divide en dos fases: *fase ancha* y *fase estrecha* (Figura 2.3). En la fase ancha* se descartan rápidamente pares de objetos sobre los que no se realizará la comprobación de colisión, sobre todo porque se sabe a priori que no van a colisionar (normalmente mediante la realización de un test sencillo y rápido). De esta forma sólo ciertos pares de objetos pasan a la fase estrecha†, que trata de determinar si se produce o no colisión entre dichos pares de objetos.

Diversos autores proporcionan soluciones a este problema. Podemos destacar las técnicas que utilizan una subdivisión del espacio para obtener los pares de objetos que se encuentran en una misma zona espacial. Podemos ver algunas de estas técnicas en los trabajos de [Her86] [Duf92] [MW88] [Lub91] [Hub93] y [OF05].



a) b)

Figura 2.3: Fases en la detección de colisión. a) Fase ancha. b) Fase estrecha.

* También llamada fase de poda o colisión dinámica.

† También llamada fase detallada, colisión detallada, colisión de bajo nivel o colisión estática.

Problema relacionado con la detección de colisión a bajo nivel

Por otra parte, debemos determinar eficiente y exactamente si se produce interpenetración o no entre objetos. Es lo que hemos llamado *fase estrecha* o test detallado. Normalmente depende de la representación de los objetos, y en el caso de una representación poliédrica (que suele ser de las más habituales), puede haber una gran cantidad de casos especiales entre los distintos elementos que lo componen (caras, aristas y vértices). Los algoritmos para este tipo de representación no suelen ser muy eficientes ni robustos.

No solo puede ser necesario obtener si se produce o no interpenetración entre objetos, sino que, como vimos anteriormente, puede ser necesario obtener otra serie de características como las primitivas involucradas o el área de interpenetración, para poder así enviar esta información a un módulo de respuesta y actuar en consecuencia. En estos casos, los algoritmos también dependen de la representación de los objetos y tampoco suelen ser muy eficientes ni robustos. En muchos de los trabajos relacionados con la fase ancha [Her86] [Duf92] [MW88] [Lub91] [Hub93] y [OF05] se ha resuelto éste problema, para ello suelen utilizar técnicas de descomposición o jerarquías de volúmenes envolventes.

Problema relacionado con el movimiento simultáneo de objetos

Debemos considerar que en el mundo real los objetos se mueven de manera simultánea. Sin embargo, realizar esto en un ordenador es bastante costoso, pues habría que calcular el momento en el que se produciría la siguiente colisión y avanzar la simulación a ese momento.

Para ciertas configuraciones, por ejemplo una bola moviéndose por un plano, el tiempo entre colisiones sería muy pequeño, en este caso serían continuas, aumentando el número de comprobaciones de colisión por unidad de tiempo pues habría que hacerlas para todos los pares de objetos además del plano y la bola. Por este motivo los objetos suelen moverse secuencialmente en la simulación, uno detrás de otro. Esto puede dar lugar a situaciones en las que se detecten colisiones entre objetos que no se producirían bajo un movimiento simultáneo, y a que no se detecten colisiones entre objetos que sí ocurrirían en el mundo real (Figura 2.4). Para resolver este problema, la solución generalmente adoptada consiste en establecer un incremento de tiempo entre comprobaciones lo suficientemente pequeño para que unos objetos no pasen a través de otros sin que se detecte la colisión y, en caso de producirse, permitir volver al paso anterior para determinar detalladamente el punto de contacto exclusivamente entre ese par de objetos.

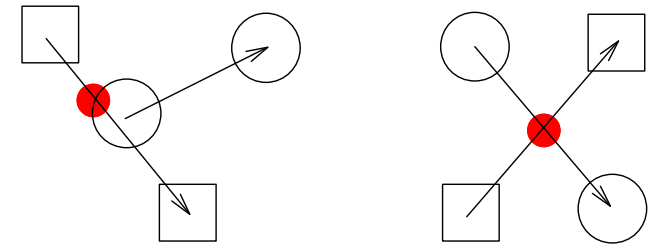


Figura 2.4: Movimiento simultáneo vs. secuencial. En la primera figura no se detectaría la colisión entre objetos si el movimiento es simultáneo. Si se mueve primero el rectángulo se produce colisión en el caso de movimiento secuencial. En la segunda figura no se produciría colisión si el movimiento es secuencial, pero sí se produce cuando el movimiento es continuo.

Problema relacionado con el movimiento continuo de los objetos

Asociado al problema anterior nos encontramos con un problema derivado que consiste en que en el mundo real los objetos se mueven de manera continua. Sin embargo, en el ordenador, los objetos deben moverse en intervalos de tiempo finitos o discretos (Δt. en el algoritmo). A veces establecer un incremento de tiempo de manera que se detecten todas las colisiones no es posible, pues puede haber objetos de distinto tamaño en la simulación y con velocidades de movimiento distintas (Figura 2.5.a).

La solución adoptada suele consistir en establecer un volumen de barrido (Figura 2.5.b) en el intervalo de tiempo considerado para un par de objetos, de manera que en primer lugar se compruebe la colisión entre los volúmenes de barrido de los dos objetos. Si se produce colisión entre dichos volúmenes, puede que se dé colisión o no entre los objetos, pero si no se produce interferencia sabremos que no ocurrirá entre objetos. Afortunadamente el movimiento de los objetos suele considerarse lineal, seguido de una rotación en posiciones estacionarias, con lo que los volúmenes a estudiar suelen ser más sencillos. En estas circunstancias es posible tratar un objeto como estacionario y otro móvil con un movimiento relativo al primero. Así el test de colisión entre volúmenes de barrido se convierte en un test exacto entre un objeto y un volumen de barrido.

Otra solución a este problema suele consistir en escoger un intervalo de tiempo adaptativo, de manera que siempre que sea posible el valor de Δt sea lo más alto posible. Diversos autores han resuelto este problema en sus algoritmos. Para ello deben conocer el movimiento de los objetos para deducir este valor de Δt adaptativo. Generalmente obtienen ecuaciones para el movimiento en función del tiempo y

resuelven esas ecuaciones para obtener el momento de la siguiente colisión [Can86] [Sny92] [CK86]. También suelen utilizar subdivisiones del espacio (rejillas regulares, octrees, etc.), de forma que controlan el paso de unas a otras regiones para obtener el intervalo de tiempo adaptativo [ST85] [Duf92].

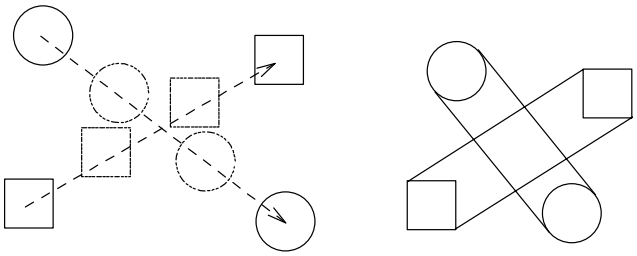


Figura 2.5: a) Cuando el movimiento se produce en intervalos de tiempo puede ser que no se detecten colisiones. b) Para solucionarlo pueden considerarse volúmenes de barrido.

Problema relacionado con el aprovechamiento de la coherencia

Llegados a este punto podemos considerar un último problema menos intuitivo, que consiste en el desaprovechamiento de información relativa a situaciones anteriores, es decir, puede haber información previa que sea común al momento actual en la simulación y que se está calculando varias veces.

Sería posible almacenar cierta información obtenida en un frame para que sirviera en frames posteriores para así excluir posibles objetos en colisión. Dicho de otra forma, se podría aprovechar la coherencia entre frames. Esto quiere decir que en determinados sistemas en los que los objetos se mueven poco entre frames, se podrían excluir en el frame actual pares de objetos que se encontraban distantes en el frame previo; o incluso aprovechar ciertos cálculos realizados en frames anteriores o realizar dichos cálculos de una manera incremental. Los métodos de [Bar92] y [LC91] son métodos representativos que aprovechan información entre frames.

2.2. Conceptos asociados a la Detección de Colisión

En aplicaciones interactivas, el tiempo disponible para la detección de colisión está limitado en primera instancia por el tiempo entre frames. Dependiendo de la aplicación, sólo una fracción de este tiempo tendrá que dedicarse a la detección de colisión, pues existen otras tareas a realizar, como la visualización, y posiblemente

otra serie de cálculos adicionales que limitan el tiempo que se puede dedicar a la tarea que tenemos entre manos. Además es necesario conocer el número máximo de objetos sobre el que realizar la Detección de Colisión, pues esta situación nos reduce el tiempo a dedicar a cada uno de los pares de objetos sobre los que realizar los cálculos.

Podemos definir fácilmente un modelo matemático que establezca las colisiones entre objetos de una manera totalmente exacta. Sin embargo, en ordenadores reales debemos cumplir ciertas especificaciones en cuanto a tiempo de procesamiento, espacio en memoria, robustez, etc. En este apartado vamos a tratar diversos aspectos que debemos considerar para el desarrollo de algoritmos eficientes y robustos en ordenadores reales.

2.2.1. Eficiencia

Habitualmente se puede ganar eficiencia aumentando la cantidad de memoria utilizada. Por ejemplo, los algoritmos más rápidos requieren una representación especial o la reutilización de datos almacenados provenientes de cálculos anteriores. Hoy día en la mayoría de ordenadores se ha aumentado en gran medida la memoria disponible, por lo que es posible utilizar métodos que empleen gran cantidad de memoria. Por otra parte, el acceso a memoria es lento en comparación con el tiempo de procesamiento, de manera que la consulta de grandes cantidades de información puede reducir la eficiencia temporal de los algoritmos.

Además podemos obtener mejores tiempos de Detección de Colisión si reducimos la complejidad de los objetos; así podemos sustituir formas complejas por formas más simples para determinar si se produce o no colisión.

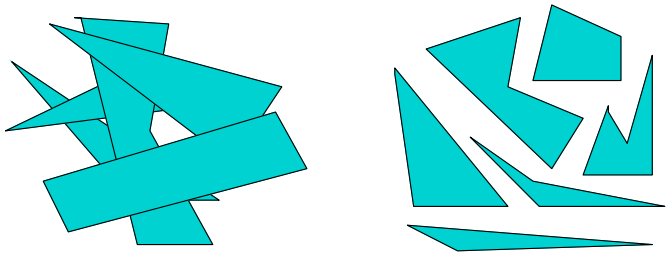
Para aumentar de alguna manera la eficiencia de los algoritmos de Detección de Colisión podemos utilizar la *coherencia*. La *coherencia espacial* nos dice que un objeto ocupa normalmente una zona pequeña en el espacio, de manera que las colisiones entre objetos suelen ser raras e infrecuentes. Por otra parte, cuando hay *coherencia temporal*, la configuración entre objetos cambia poco entre instantes de tiempo consecutivos. El movimiento en esta situación suele ser suave.

Lo habitual en la mayor parte de aplicaciones es que haya coherencia espacial, con lo que el número de colisiones entre objetos será pequeño; además si hay coherencia temporal, podremos aprovechar ciertos cálculos relacionados con la colisión entre objetos, realizados en un instante de tiempo para el siguiente, con el consiguiente ahorro computacional.

2.2.1.1. Coherencia Espacial o Geométrica

La *coherencia espacial* de un modelo complejo, también denominada *coherencia geométrica*, puede describirse como el grado de separabilidad del conjunto de objetos en el modelo [Ber04]. Este modelo puede ser un objeto que se subdivide en piezas más pequeñas, o puede ser una escena en la que hay diversos objetos.

Decimos que dos objetos pueden separarse si las regiones definidas por las envolventes convexas de los objetos son disjuntas. El grado de separabilidad disminuye si el grado de solapamiento entre las envolventes convexas aumenta (Figura 2.6).



a) baja coherencia geométrica b) alta coherencia geométrica

Figura 2.6: Grado de coherencia geométrica en un conjunto de objetos.

La utilización de volúmenes envolventes es un método que hace uso de la coherencia espacial. Un *volumen envolvente* es una primitiva simple que encierra a una forma más compleja y al que se le puede hacer un test de intersección más rápido en términos computaciones. Si la probabilidad de que los volúmenes envolventes de los objetos intersequen es baja (es decir, hay un bajo grado de coherencia espacial entre los objetos del modelo), podemos reducir el tiempo de Detección de Colisión realizando en primer lugar un test de intersección entre volúmenes envolventes. Sólo será necesario realizar un test de colisión exacto entre objetos para aquellos cuyos volúmenes envolventes intersequen. El uso de volúmenes envolventes requiere un espacio adicional en memoria así como cálculos adicionales, pero se obtiene una ganancia en tiempo de procesamiento en el caso de que la probabilidad de que los volúmenes envolventes sean disjuntos sea alta.

2.2.1.2. Coherencia Temporal o entre Frames

La *coherencia temporal* es también llamada *coherencia entre frames*. En el supuesto de que se produzcan pequeños cambios entre frames, se suelen repetir bastantes cálculos para los mismos valores de entrada. Si estos cálculos se almacenan y

reutilizan, puede reducirse el tiempo de cálculo en cada frame. Llamamos coherencia temporal a la medida de la reutilización de cálculos provenientes de frames previos [Ber04].

Podemos denominar "*observación*" a un conjunto de información importante acerca de una configuración formada por un par de objetos, de manera que ésta información pueda ser utilizada para determinar rápidamente la detección de colisión entre ese par de objetos, y siempre bajo el supuesto de que esa configuración cambiará poco. Una observación puede ser positiva o negativa. Un ejemplo de observación positiva podría ser el punto en común entre los dos objetos. Un ejemplo de observación negativa podría consistir en el uso de un plano de separación entre objetos convexos (Figura 2.7).

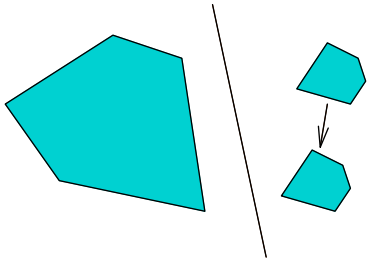


Figura 2.7: Observación negativa: Plano de separación. Entre frames el objeto está en el mismo lado del plano, esto nos asegura que no se produce colisión.

La idea de almacenar y utilizar observaciones se basa en el hecho de que la comprobación de una observación en un frame puede servir para el siguiente, con lo que no es necesario recalcular dicha observación completamente desde el principio, lo que resulta siempre más costoso. Por ejemplo, para determinar si se produce intersección entre un par de objetos, se puede realizar la inclusión de puntos en ambos objetos, que siempre es un test más simple. Por lo tanto, si una observación es válida en un frame (la inclusión de un punto en ambos objetos) y hay un alto grado de coherencia temporal, puede ser válida en el siguiente frame, por lo que es aconsejable comprobar en primer lugar la validez de la observación en dicho frame. Sólo si no es válida esa observación se debe realizar un costoso test de intersección y calcular una nueva observación.

Además del coste adicional de comprobar la validez de la observación, es necesario añadir el coste de almacenar y obtener dicha información entre frames.

2.2.2. Robustez

La geometría es la base para el desarrollo de algoritmos de Detección de Colisión. En un algoritmo teórico, todos los números son reales, y las operaciones entre reales son exactas. Sin embargo, en la implementación de estos algoritmos en ordenadores reales no hay disponible una representación exacta de estos números, o si es posible esta representación sería muy costosa. Estos números se representan mediante reales en coma flotante, cuya aritmética no es exacta, pues se producen redondeos para aproximar los resultados a una precisión dada. Este error introducido en los algoritmos puede causar bastantes problemas en los algoritmos de Detección de Colisión.

Un programa *robusto* proporcionará resultados correctos ante valores de entrada "problemáticos", mientras que un programa *no robusto* puede fallar o no obtener el resultado esperado ante los mismos valores de entrada. Podemos clasificar los problemas de robustez en *robustez numérica* y *robustez geométrica*.

La robustez numérica es consecuencia del uso de variables con precisión finita en los cálculos. La robustez geométrica consiste en asegurar la corrección topológica y una consistencia geométrica global [Eri05].

2.2.2.1. Robustez numérica

La *robustez numérica* hace referencia a la capacidad de un programa para tratar con cálculos numéricos y configuraciones geométricas que son difíciles de manejar de algún modo. Un algoritmo robusto obtiene resultados consistentes y trata las situaciones degeneradas de la forma esperada.

Los programas que tratan la Detección de Colisión son especialmente sensibles a problemas de robustez, debido sobre todo a la naturaleza numérica y geométrica de los algoritmos utilizados.

Principalmente nos encontramos con dos categorías de problemas de robustez: debidos a una precisión inadecuada y a los casos degenerados. Los problemas de precisión vienen causados por el hecho de que los cálculos en el ordenador no son realizados con aritmética real exacta, sino que utilizan una aritmética con una precisión limitada. Los casos degenerados son casos especiales o condiciones especiales para los que, en principio, un algoritmo no está diseñado para funcionar correctamente.

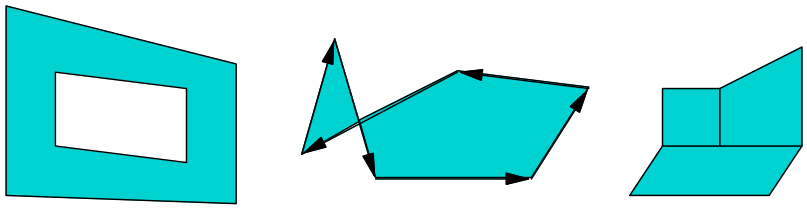
Normalmente la *imprecisión numérica* viene dada por alguna de las siguientes situaciones: errores de conversión y representación; errores debidos a overflow y underflow; errores de redondeo; errores por cancelación de dígitos (por ejemplo cuando restamos un valor grande de uno pequeño, ambos representables).

Los *casos degenerados* hacen referencia a casos especiales que necesitan un tratamiento especial y diferente del caso general. Por ejemplo, calcular el signo de un determinante para resolver en qué lado de un plano se encuentra un punto. Si el punto está sobre el plano o muy próximo a él un pequeño error en la posición del punto puede cambiar drásticamente el signo obtenido en los cálculos.

Para resolver estos problemas de robustez numérica pueden utilizarse librerías específicas para la representación de reales, con el consiguiente aumento del tiempo de cálculo y espacio de almacenamiento, que en aplicaciones de tiempo real no son siempre adecuados [FW93]. El uso de operaciones en punto flotante puede ser resuelto de forma más robusta para la Detección de Colisión utilizando tolerancias, es decir, una vez realizada una operación podemos comparar el valor obtenido con un error dado. Por ejemplo, en lugar de utilizar la comparación *if (x=0.0f)* utilizar *if (abs(x) <= epsilon)*. Otra forma de resolver los errores de redondeo producidos en los cálculos en punto flotante es utilizar la llamada aritmética de intervalos [Moo66] o utilizar aritmética entera con mayor precisión [Knu97].

2.2.2.2. Robustez geométrica

Es importante que la geometría de los objetos sobre los que se realice la detección de colisiones esté libre de errores y represente de forma adecuada al propio objeto. Las relaciones entre vértices, aristas y caras es fundamental para los cálculos implícitos en la detección de colisión, de manera que una información incorrecta sobre estas relaciones puede dar lugar a una determinación incorrecta de la colisión entre objetos. Algunas características geométricas pueden causar problemas, como: vértices, aristas o caras repetidas; caras degeneradas, incluyendo caras no convexas, con autointersecciones o con un área igual a cero o próxima a cero; caras con una orientación incorrecta; agujeros, uniones en T; caras no planas, etc. (Figura 2.8).



a) Caras con agujeros b) Polígono con autointersecciones c) Unión en T

Figura 2.8: Algunos problemas de la representación geométrica.

Aunque dependiendo del método de detección de colisión, ciertos problemas pueden no ser tales, como en nuestro caso, la existencia de caras no convexas o con agujeros. En general, este tipo de problemas en la geometría debe detectarse y corregirse para asegurarnos un correcto funcionamiento de los algoritmos de detección de colisión. Normalmente se dan los siguientes pasos para corregir estas anomalías: unión de vértices; unión de agujeros entre caras adyacentes; unión de caras coplanarias en una única cara; y descomposición en piezas convexas. Esta corrección de la geometría puede ser bastante dificultosa debido a que estos pasos no están plenamente desarrollados y todavía se está investigando en su mejora. Podemos ver distintas técnicas relacionadas con la obtención de una geometría correcta en [Eri95].

2.2.3. Representación de Objetos

Generalmente los sistemas de detección de colisión dependen de la representación de los objetos, siendo algunas técnicas más apropiadas que otras para ciertos tipos de representaciones. En [LG98] se presenta un estudio acerca de estas técnicas en función de la representación de los objetos.

El hardware gráfico actual utiliza triángulos como primitiva básica de visualización. Como consecuencia, la *representación poligonal* es la elección natural para la representación de objetos, así como para el desarrollo de algoritmos de detección de colisión. Una representación bastante utilizada también es la *sopa de polígonos*: una colección desordenada de polígonos sin información de las posiciones relativas de unos polígonos respecto de otros. Podemos considerar que los algoritmos que utilizan esta representación suelen ser menos eficientes y robustos que aquellos que poseen información topológica adicional.

Los polígonos también pueden conectarse entre sí para formar superficies poligonales mayores, son las llamadas *mallas de polígonos*. Este es uno de los métodos más utilizados para la representación de objetos.

La mayor parte de los algoritmos de detección de colisión desarrollados hasta el momento tratan *objetos convexos* formados por *polígonos convexos*. El desarrollo de algoritmos eficientes para objetos no convexos o formados por polígonos no convexos es importante. Esto es así debido a que algunas aplicaciones soportan polígonos no convexos, como es el caso de VRML97. Además, para n primitivas son potencialmente necesarios $O(n^2)$ tests de intersección, si reducimos el número de primitivas a tratar ganaremos eficiencia. Así mismo no será necesario descomponer en partes convexas (línea de investigación todavía abierta en cuanto a la obtención de algoritmos eficientes y robustos), con el consiguiente ahorro de tiempo de pre-procesamiento y ahorro de espacio de almacenamiento.

2.3. Técnicas básicas usadas en Detección de Colisión

En aplicaciones interactivas en las que es necesario conocer la colisión entre objetos necesitamos reducir el número de tests de colisión entre pares de objetos. Para ello son necesarias estructuras de datos especiales y heurísticas que reduzcan estos $O(n^2)$ pares de tests en el peor de los casos. Además se tratará de reducir el número de pares de características a tratar en el test de colisión detallado entre un par de objetos.

En esta sección trataremos con algunas de las estrategias más utilizadas para reducir esa complejidad [JSF02] [JSF02b]. Puede ampliarse información acerca de distintas técnicas en [JTT01] [GSF94] y [Eri05].

2.3.1. Volúmenes Envolventes

Comprobar la colisión entre dos objetos es muy costoso, sobre todo cuando están formados por miles de polígonos. Para minimizar este coste, suele comprobarse previamente la colisión entre sus volúmenes envolventes, de manera que sólo si colisionan realicemos un test de colisión detallado[*] entre objetos.

Un *volumen envolvente* está formado por un volumen simple que contiene uno o más objetos de naturaleza compleja. La idea consiste en utilizar un test de colisión entre volúmenes envolventes, menos costoso que el propio test de colisión entre objetos.

Suri et al. [SHH98] enunciaron un teorema que da soporte teórico al uso de volúmenes envolventes para la detección de colisión. Este teorema establece que el número de intersecciones entre volúmenes envolventes (con una relación de aspecto y factor de escala constantes) es casi el mismo que el número de intersecciones entre objetos.

Cuando los objetos se encuentran en colisión, es cierto que se añade un pequeño coste a la detección de colisión, pues previamente se ha realizado un test de colisión entre volúmenes envolventes. En realidad se produce una ganancia, pues la mayoría de las veces no se produce colisión entre los volúmenes envolventes, con el consiguiente ahorro de tests detallados entre objetos.

[*] Suele llamarse también test de colisión estático o test de la segunda fase de la detección de colisión, es decir, fase ancha (narrow phase).

Para que sea útil, un volumen envolvente debe cumplir las siguientes propiedades:

1. Debe ajustarse al objeto tanto como sea posible.
2. El cálculo de colisión entre volúmenes envolventes debe ser poco costoso en comparación con el test de colisión entre objetos.
3. El cálculo del volumen envolvente debe ser poco costoso, sobre todo si éste debe recalcularse con cierta regularidad.
4. El volumen envolvente debe poder rotarse y transformarse fácilmente.
5. Debe representarse utilizando poca cantidad de memoria.

Los volúmenes envolventes más representativos son las *esferas envolventes*, las *cajas envolventes alineadas con los ejes,* las *cajas envolventes orientadas* y los *k-DOPs*, utilizándose normalmente las siglas en inglés de estos términos. En la Figura 2.9 podemos ver estos volúmenes envolventes y cómo se ajustan unos mejor que otros a un objeto determinado. A continuación pasamos a describir cada tipo de volumen envolvente.

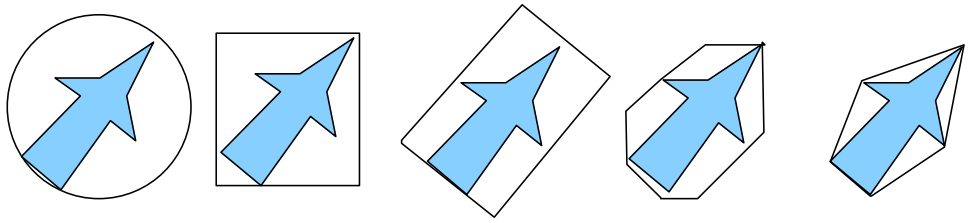


Figura 2.9: Tipos de volúmenes envolventes: a) Esfera. b) Caja envolvente alineada con los ejes. c) Caja envolvente orientada. d) *k*-dop. e) Envolvente convexa.

Esfera envolvente:

Se le suele llamar BS (Bounding sphere). Una esfera envolvente se beneficia de un test de colisión poco costoso, además tiene la ventaja de ser un volumen invariante ante rotaciones y de necesitar poco espacio de almacenamiento. El inconveniente de este tipo de volumen envolvente consiste en que no se ajusta a determinados objetos, siendo difícil obtener la esfera que mejor ajuste a un objeto dado. Sin embargo existe un algoritmo [Wel91] que obtiene la mínima esfera envolvente en tiempo lineal. Este algoritmo es recursivo y puede producir un desbordamiento de la pila. En [Cap01] puede verse una implementación de este algoritmo que evita este problema.

Caja envolvente alineada con los ejes:

Normalmente conocida como AABB (Axis aligned bounding box), es una caja rectangular cuyas caras están orientadas de manera que sus normales son paralelas a los ejes coordenados. La ventaja de este tipo de volumen es que el test de colisión es muy rápido, consistiendo en la comparación directa de los valores de coordenadas. Sin embargo, no se produce un ajuste apropiado para muchos objetos. Además, actualizar este volumen envolvente suele ser más costoso que otros cuando los objetos rotan.

Caja envolvente orientada:

También conocida como OBB (Oriented Bounding Box), este tipo de volumen envolvente es representado por una caja rectangular con una orientación arbitraria, aunque suelen utilizarse determinadas direcciones prefijadas. Al contrario que las AABB, este tipo de volumen envolvente tiene un test de colisión bastante costoso, basado en el Teorema del eje separador* [Got96] cuya utilización para detección de colisiones fue sugerida por Larcombe [Lar95]. Sin embargo, su uso viene justificado por su mejor ajuste a los objetos y su rápida actualización en las rotaciones. Además es necesario guardar mucha más información para representar este volumen, normalmente bajo la forma del centro de la caja, tres longitudes y una matriz de rotación.

Politopo de orientación discreta:

Es un volumen envolvente basado en la intersección de *"slabs"*. Un slab es la región infinita del espacio delimitada por dos planos paralelos. Este volumen conocido como *k*-DOP (Discrete Orientation Polytope) es un politopo convexo definido por slabs cuyas normales pertenecen a un conjunto de direcciones predefinidas y comunes para todos los *k*-DOPs. Las componentes de los vectores normales están restringidas al conjunto de valores {-1,0,1}, y éstos no están normalizados (Figura 2.10).

Un AABB es un 6-DOP, pues está formado por 3 slabs. Evidentemente, mientras mayor sea el número de slabs de un *k*-DOP mejor se ajusta éste a los objetos, siendo necesario comprobar $k/2$ intervalos para determinar la colisión de volúmenes envolventes. La actualización de un *k*-DOP es algo más costosa que un AABB debido a su mayor número de slabs, pero proporcional a *k*. El almacenamiento de un *k*-DOP depende también del número de slabs pero es poco costoso debido sobre todo a que estos slabs están restringidos a ciertas configuraciones. Este volumen tiene la ventaja

* Comúnmente suele utilizarse su nombre en inglés: *Separating axis theorem*.

adicional de que siempre se puede modificar el tipo de ajuste a un objeto cambiando el número de slabs a utilizar.

Existen muchos otros tipos de volúmenes envolventes que los aquí expuestos, como *conos* [Hel97] [Ebe02], *cilindros* [Hel97] [Ebe02] [SSTY00], *elipsoides* [WCCKW02] [CL03], etc.

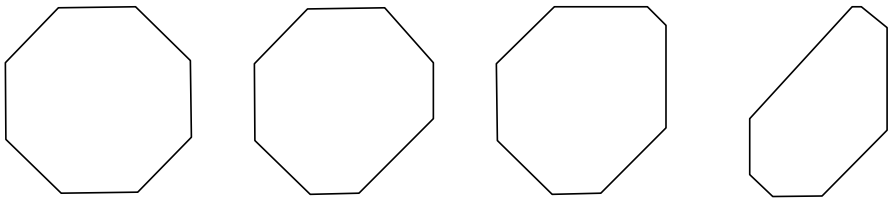


Figura 2.10: *k*-dops. Los distintos slabs son ajustados para adaptarse a un objeto, con lo que cambia la forma del volumen envolvente. Las planos que forman un slab siempre deben tener el mismo ángulo con la vertical que la configuración original.

2.3.1.1. Jerarquías de Volúmenes Envolventes

Utilizar Volúmenes Envolventes, como hemos visto, puede disminuir el tiempo de cálculo de colisión entre objetos. Sin embargo el número de parejas de objetos sobre el que realizar este test no varía, por lo que, el tiempo de ejecución es el mismo aunque reducido por una constante. Uniendo los volúmenes envolventes a una jerarquía podemos reducir esta complejidad a logarítmica, construyendo para ello una *jerarquía de volúmenes envolventes*.

Para *n* objetos podemos construir una jerarquía de volúmenes envolventes en forma de árbol (Figura 2.11) de la siguiente manera: En primer lugar obtenemos los volúmenes envolventes de los objetos individuales y los colocamos como nodos hoja del árbol que representa dicha jerarquía. Estos nodos son agrupados utilizando un nuevo volumen envolvente de manera recursiva hasta obtener un volumen envolvente que agrupe todos los objetos (representado por la raíz del árbol). Con esta jerarquía comprobaríamos la colisión entre hijos de un nodo sólo si se produce colisión entre los nodos padre.

En una jerarquía de volúmenes envolventes no es necesario que un volumen padre englobe los volúmenes hijos, sino que simplemente debe englobar los dos objetos que representan los volúmenes envolventes. Además dos volúmenes envolventes del mismo nivel pueden tener partes comunes (intersecciones).

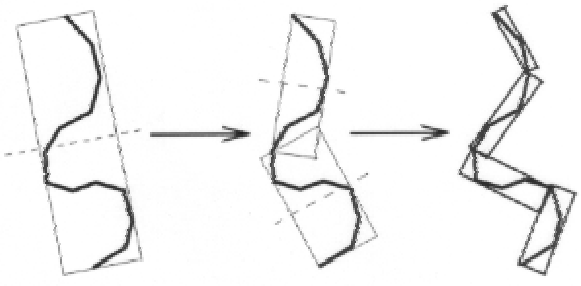


Figura 2.11: Jerarquía de Volúmenes Envolventes.

Una jerarquía de volúmenes envolventes puede utilizarse para organizar los objetos de una escena o bien pueden utilizarse jerarquías de volúmenes envolventes individuales para cada uno de los objetos. Suelen usarse para objetos poliédricos no convexos, de manera que se dividen en componentes convexas. Debemos tener en cuenta que esta partición en componentes convexas no suele ser trivial.

Sería deseable que una jerarquía de volúmenes envolventes tuviera unas serie de propiedades [KK86][Hub95]:

1. La geometría contenida en los nodos de cualquier sub-árbol debería estar próxima entre sí.
2. Cada nodo de la jerarquía debería tener el mínimo volumen posible.
3. Debemos considerar en primer lugar los nodos cercanos a la raíz de la jerarquía, pues eliminar uno de estos nodos es siempre mejor que hacerlo a mayor profundidad del árbol.
4. El solapamiento entre nodos del mismo nivel debería ser mínimo.
5. La jerarquía debería ser equilibrada en cuanto a su estructura y su contenido.
6. En aplicaciones de tiempo real, la determinación de colisión utilizando la jerarquía no debe ser mucho peor que el caso medio. Además la jerarquía debe generarse de forma automática, sin la intervención de un usuario.

Algunas de estas descomposiciones mediante jerarquías de volúmenes envolventes incluyen entre otros: *AABB-Trees* [Ber97], *OBB-Trees* [GLM96], *Box-Trees* [BCGMT96] [Zac98], *Sphere-Trees* [Hub96] [BO04], *k-DOP trees* [KHMSZ98], *Shell-Trees* [Zac98] [KGLMP98], *R-Trees* [BKSS90] y *QuOSPO-Trees* [He99].

2.3.1.2. Volúmenes Envolventes a distinto nivel de detalle

Podemos utilizar distintos volúmenes envolventes, cada uno con mayor nivel de ajuste o de detalle que el anterior y formar una sucesión de volúmenes envolventes

(Figura 2.12). La proximidad entre objetos se obtiene de la misma forma para todos los niveles de este conjunto. Cuando no se puede determinar de forma exacta si dos objetos colisionan, se pasa al siguiente nivel de detalle.

Esta técnica implica un mayor coste de almacenamiento debido a la multiplicidad de volúmenes envolventes y un mayor número de tests cuando se produce colisión entre objetos. Además la actualización de un mayor número de volúmenes es mucho más costosa. Sin embargo, esta variabilidad en el ajuste de los volúmenes envolventes permite realizar tests de colisión entre volúmenes envolventes en menor número cuanto más ajustado al objeto sea el volumen (y más costoso), y mayor número de tests entre volúmenes envolventes con menor ajuste (y menor coste).

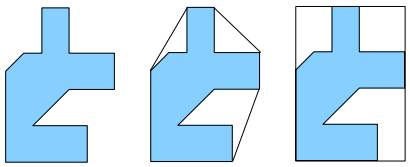


Figura 2.12: Volúmenes envolventes a distinto nivel de detalle.

2.3.2. Descomposición del Espacio

La *descomposición espacial* es una técnica utilizada normalmente para descartar pares de objetos en la fase ancha. En general puede utilizarse para determinar en qué zona del espacio se encuentra un objeto o parte del mismo (incluidas las características de un poliedro), y realizar un test de colisión entre objetos (o partes) sólo si ocupan la misma zona del espacio. A continuación veremos algunos métodos de descomposición espacial utilizados en detección de colisiones.

2.3.2.1. Rejilla de Vóxeles

Una *rejilla* [GSF94] es una partición del espacio en celdas rectangulares y uniformes, es decir, todas las celdas tienen el mismo tamaño. Cada objeto se asocia con las celdas en las que se encuentra. Para que dos objetos colisionen deben ocupar una misma celda. Sólo debe comprobarse la colisión entre objetos que ocupen la misma celda (Figura 2.13).

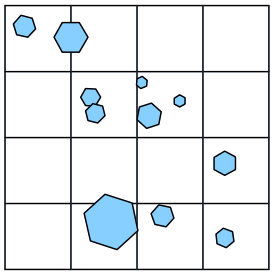


Figura 2.13: Rejilla utilizada en detección de colisión.

Averiguar la celda a la que pertenece una coordenada consiste en dividir entre el tamaño de la celda, esto es simple y eficiente. Además las celdas adyacentes a una dada se pueden localizar fácilmente. Esta simplicidad ha hecho que las rejillas se utilicen comúnmente como método de descomposición espacial.

En términos de eficiencia, uno de los aspectos más importantes de los métodos basados en rejillas consiste en elegir el tamaño apropiado de las celdas. Hay cuatro situaciones en las que el tamaño de las celdas es un problema:

1. La rejilla es muy fina. Si las celdas son muy pequeñas, el número de celdas a actualizar será alto y por tanto costoso en términos de espacio y tiempo empleados.
2. La rejilla es demasiado dispersa en relación al tamaño de los objetos. Si los objetos son pequeños y las celdas grandes, habrá muchos objetos por celda. En esta situación puede ser necesario realizar muchos pares de test de colisión entre los numerosos objetos clasificados en una celda.
3. La rejilla es demasiado dispersa en relación a la complejidad de los objetos. En este caso, el tamaño de las celdas es adecuado al tamaño de los objetos. Sin embargo, el objeto es demasiado complejo. Las celdas deben reducirse de tamaño, y los objetos deben descomponerse en piezas más pequeñas.
4. La rejilla es demasiado fina y dispersa. Si los objetos son de tamaños muy dispares, las celdas pueden ser muy grandes para los objetos pequeños y viceversa.

Normalmente el tamaño de celda de una rejilla se ajusta para que acomode el objeto de mayor tamaño rotado un ángulo arbitrario. Así se asegura que el número máximo de celdas que ocupe un objeto sea de cuatro en 2D y de ocho en 3D. De esta forma se reduce el esfuerzo necesario para insertar o actualizar los objetos en la rejilla, así como el número de test de colisión a realizar.

2.3.2.2. Octrees y k*-d trees*

Un *octree* [ABJN85][Swa93] es una partición jerárquica del espacio mediante *vóxeles* alineados con los ejes. Cada nodo padre se divide en ocho hijos. El nodo raíz suele representar el volumen del espacio completo a representar (la escena). Este volumen se divide en ocho sub-volúmenes tomando la mitad del volumen en los ejes x, y, z. Estos ocho volúmenes se dividen recursivamente de la misma forma. El criterio de parada puede consistir en alcanzar una profundidad máxima en el árbol generado o un volumen mínimo para un nodo.

En un *k*-d tree [HKM95] cada región es dividida en dos partes por un plano alineado con los ejes. Un *k*-d tree puede realizar divisiones uniformes del espacio o variables (existen también octrees con divisiones variables). Las divisiones variables, aunque necesitan mayor almacenamiento para la estructura, se adaptan mejor a los objetos.

Utilizar una subdivisión jerárquica del espacio permite la adaptación a las densidades locales del modelo. En regiones en las que hay un gran número de objetos, se puede dividir más que en regiones en las que hay menos objetos. En aplicaciones de detección de colisión entre objetos en movimiento, una celda puede subdividirse cuando entran nuevos objetos y agruparse celdas en una cuando el número de objetos disminuye.

2.3.2.3. BSP trees

Un *árbol de partición binaria del espacio* [NAT90] es una estructura jerárquica que subdivide el espacio en celdas convexas. Cada nodo interno del árbol divide la región convexa asociada con el nodo en dos regiones, utilizando para ello un plano con orientación y posición arbitrarios. Es análogo a un *k*-d tree variable sin la restricción de que los planos estén orientados ortogonalmente con los ejes.

Un *árbol BSP* puede utilizarse como esquema de representación de objetos o como partición del espacio. Normalmente como técnica para la fase ancha de la detección de colisiones, un árbol BSP se utiliza como partición del espacio. Sin embargo esta técnica presenta el problema de cómo elegir los planos sobre los que dividir el espacio para obtener una descomposición óptima. Este es un problema complejo como puede apreciarse en [Nay93].

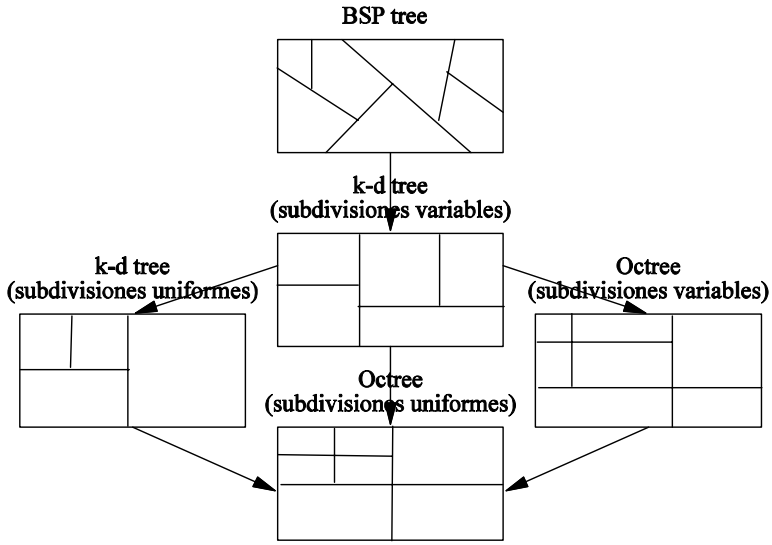


Figura 2.14: Taxonomía de estructuras jerárquicas de descomposición espacial. BSP tree, k-d tree y octree.

2.4. Métodos para objetos convexos

Como ya hemos visto en secciones anteriores, los métodos de detección de colisión más desarrollados y de uso común son específicos para objetos convexos, pues en el caso de objetos cóncavos estos se suelen descomponer en piezas convexas. Los objetos convexos poseen una serie de propiedades que los hacen adecuados para el desarrollo de métodos eficientes de detección de colisión. Una de estas propiedades radica en la existencia de un plano de separación entre objetos convexos cuando estos no se encuentran en colisión. Otra propiedad consiste en que la distancia entre dos puntos (uno de cada objeto) es un mínimo local y global, de manera que la distancia entre dos objetos convexos puede encontrarse fácilmente hallando ese mínimo global. Veremos a continuación algunos de los métodos que explotan estas propiedades.

2.4.1. Algoritmo de las características más cercanas

En simulaciones en tiempo real los objetos suelen moverse o rotar poco entre frames. Para objetos convexos, esta coherencia entre frames nos dice que los puntos más cercanos entre dos objetos que no colisionan estarán en la vecindad de los puntos más cercanos del frame anterior. Sin embargo, para objetos poliédricos, un pequeño cambio de orientación puede dar lugar a que el punto más cercano pase de un

extremo a otro de una cara. Por tanto, para poliedros, en lugar de tratar con los puntos más cercanos podemos tratar con las características* más cercanas. Decimos que un par de características, una de cada poliedro, son las características más cercanas si contienen al par de puntos más cercanos para los dos poliedros.

Uno de los algoritmos más conocidos que utiliza este método es el algoritmo de Lin-Canny [LC91]. Sin embargo, existe otro algoritmo que resuelve algunos problemas de éste, desarrollado por Mirtich, llamado V-Clip [Mir98].

El algoritmo V-Clip se basa en el siguiente teorema (Figura 2.15):

"Sean F_A y F_B un par de características de dos poliedros convexos y disjuntos, A y B, y sean $VR(F_A)$ y $VR(F_B)$ sus regiones de Voronoi. Sean $P_A \in F_A$ y $P_B \in F_B$ el par de puntos más cercanos entre las características. Entonces, si $P_A \in VR(F_B)$ y $P_B \in VR(F_A)$, F_A y F_B son el par de características más cercanas globalmente entre A y B (no necesariamente únicas)."

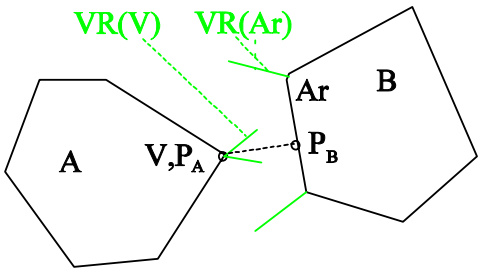


Figura 2.15: Determinación de las características más cercanas, mediante las regiones de Voronoi del vértice *V* y de la arista *Ar*.

El algoritmo comienza con dos características, una de cada poliedro. En cada iteración del algoritmo, se comprueba si las características cumplen las condiciones del teorema para ser el par de características más cercano. Si es así, el algoritmo determina que no hay intersección y devuelve el par de características más cercanas. Si no se cumplen las condiciones, V-Clip actualiza una de las características a una característica vecina y comienza de nuevo.

Las características vecinas de una dada son:

- Los vecinos de un vértice son las aristas incidentes en el vértice.
- Los vecinos de una cara son las aristas que forman dicha cara.
- Los vecinos de una arista son los dos vértices y las dos caras que inciden en la arista.

* Recordamos que el término se refiere a un vértice, una arista o una cara de un poliedro.

2.4.2. Algoritmo de niveles de detalle de Ehmann

Ehmann et al. [EL00] desarrollaron un algoritmo que utiliza una técnica parecida a la descrita en la Sección 2.3.1.2. sobre volúmenes envolventes a distinto nivel de detalle. En primer lugar precalculan una serie de volúmenes con distinto nivel de detalle (LOD) con un error acotado utilizando la *jerarquía de Dobkin-Kirkpatrick (DK)* [DK90] (Figura 2.16). La jerarquía DK de un poliedro es la secuencia de politopos convexos P_0, P_1, ..., P_k donde P_0 es el poliedro de entrada. Cada politopo está formado por un menor número de características que el anterior.
El subconjunto de vértices que se utiliza entre dos politopos de la secuencia, es seleccionado de manera que formen un conjunto de vértices independientes máximo, es decir, que no haya dos vértices adyacentes en el conjunto.

Mediante esta jerarquía se pueden acelerar ciertos cálculos, como encontrar el punto más cercano a otro. En cuanto a la intersección entre dos polígonos convexos puede obtenerse en tiempo *O(log n)* y entre poliedros convexos en tiempo *O(log² n)*.

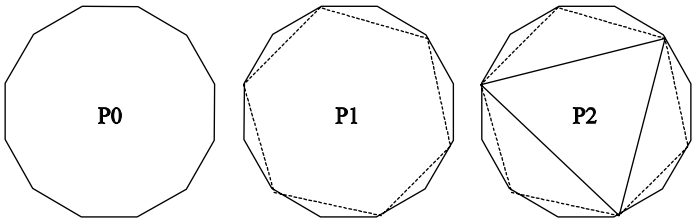


Figura 2.16: Jerarquía de Dobkin-Kirkpatrick.

2.4.3. Programación lineal

Aunque es natural considerar el problema de detección de colisión como un problema geométrico, puede tratarse también como un problema matemático de optimización. Si consideramos cada poliedro como el volumen de intersección de un número de semiespacios, el test de intersección booleano entre poliedros es equivalente al problema de factibilidad de la programación lineal [Eri05]. Éste consiste en determinar si hay solución a un conjunto de desigualdades lineales. En términos del problema de detección de colisión, consiste en determinar si existe un punto interior a todos los semiespacios de los dos poliedros (cada subespacio se representa mediante una desigualdad lineal). Además se puede calcular la distancia de separación y los puntos más cercanos como un problema de optimización, específicamente como un problema de programación cuadrática.

2.4.4. Algoritmo de Gilbert-Johnson-Keerthi (GJK)

Uno de los algoritmos mas eficientes para determinar la intersección entre dos poliedros es un algoritmo iterativo denominado *GJK* [GJK88]. Éste es un algoritmo que, dados dos conjuntos de vértices como entrada, encuentra la distancia Euclídea (y los puntos más cercanos) entre las envolventes convexas de estos conjuntos. El algoritmo opera sobre la *diferencia de Minkowski* [LGLM00] de los dos objetos, con lo que el problema de encontrar la distancia entre dos conjuntos convexos se convierte en encontrar la distancia entre el origen y un único conjunto convexo (Figura 2.17).

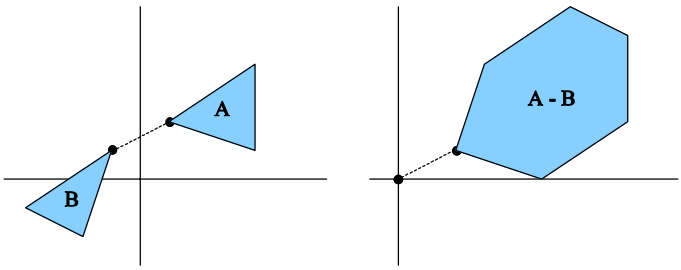


Figura 2.17: Diferencia de Minkowski de dos polígonos.

El algoritmo no calcula realmente la diferencia de Minkowski de los dos objetos, bastante costosa, sino que utiliza una función de soporte. Utilizando estas funciones puede calcular puntos de la diferencia de Minkowski.

Para encontrar la diferencia de Minkowski para el punto más cercano al origen, el algoritmo utiliza el teorema de Caratheodory*. Éste dice que "para un cuerpo convexo H de R^d cada punto de H puede expresarse como una combinación convexa de no más de $d+1$ puntos de H". Esto permite al algoritmo GJK encontrar el volumen de la diferencia de Minkowski manteniendo un conjunto Q de hasta $d+1$ puntos de dicha suma en cada iteración. La envolvente convexa de Q forma un símplice dentro de la diferencia de Minkowski. Si el origen está incluido en el símplice actual, los objetos originales intersecan y el algoritmo para. En otro caso, el conjunto Q se actualiza para formar un nuevo símplice. Este proceso debe terminar con Q conteniendo el punto más cercano al origen. En caso de que no se produzca intersección, la distancia más pequeña entre los objetos es calculada por el punto de mínima normal en la envolvente convexa de Q. A continuación podemos ver dicho algoritmo (Algoritmo 2.2 y Figura 2.18):

* Una buena definición, junto con su demostración puede verse en la página:
http://en.wikipedia.org/wiki/Carath%C3%A9odory's_theorem_(convex_hull)

1. Inicializar el conjunto de símplices Q a uno o más puntos (hasta *d+1* puntos, siendo *d* la dimensión) de la diferencia de Minkowski de los objetos A y B.
2. Calcular el punto *P* de mínima normal en la envolvente convexa de Q (*CH(Q)*)
3. Si *P* es el propio origen, el origen está incluido claramente en la diferencia de Minkowski de A y B. Parar y devolver que A y B intersecan.
4. Reducir Q al conjunto más pequeño Q' de Q tal que *P* pertenezca a *CH(Q')*. Es decir, quitar cualquier punto de Q que no determine el subsímplice de Q en el que está *P*.
5. Sea $V = s_{A-B}(-P) = s_A(-P)-s_B(-P)$ un punto de soporte en la dirección *–P*.
6. Si *V* no es más extremo en la dirección *–P* que el propio *P*, parar y devolver que A y B no intersecan. La longitud del vector desde el origen a *P* es la distancia de separación entre A y B.
7. Añadir *V* a Q y volver al paso 2.

Algoritmo 2.2: Algoritmo GJK.

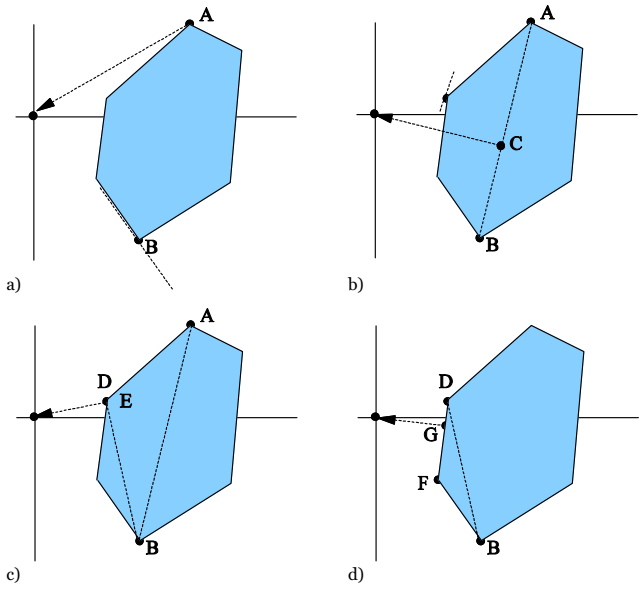


Figura 2.18: Algoritmo GJK. Determinación del punto más cercano al origen de la diferencia de Minkowski de dos objetos.

2.4.5. Algoritmo de Chung-Wang

También llamado *algoritmo del vector de separación* [CW96] [Chu96] consiste en realidad en dos algoritmos. El primero comienza con un candidato a vector de separación y, en el caso de que no haya intersección entre los poliedros P y Q, actualiza iterativamente el vector a uno cada vez más cercano al vector de separación. Otro algoritmo detecta el caso en que P y Q colisionen, pues en esta situación el primer algoritmo iteraría indefinidamente.

Este algoritmo opera sobre los vértices de los poliedros. En cada iteración i, el algoritmo principal calcula los dos vértices más extremos $p_i \in P$ y $q_i \in Q$ con respecto al candidato actual, el vector s_i. Si este par de vértices indica separación de los objetos a lo largo de s_i, el algoritmo determina que no hay intersección. En otro caso, el algoritmo calcula un nuevo vector de separación candidato proyectando s_i sobre la perpendicular a la línea que pasa por p_i y q_i. A continuación mostramos este algoritmo (Algoritmo 2.3 y Figura 2.19):

1. Comenzar con un vector de separación candidato s_0. Sea $i=0$.
2. Encontrar los vértices extremos p_i de P y q_i de Q tales que $p_i \cdot s_i$ y $q_i \cdot -s_i$ son minimizados.
3. Si $p_i \cdot s_i < q_i \cdot -s_i$, entonces s_i es un eje de separación. Salir devolviendo no intersección.
4. En otro caso, calcular un nuevo vector de separación como $s_{i+1}=s_i-2(r_i \cdot s_i)r_i$, donde $r_i=(q_i-p_i)/||q_i-p_i||$.
   $i = i+1$. Ir al paso 2.

Algoritmo 2.3: Algoritmo de Chung-Wang.

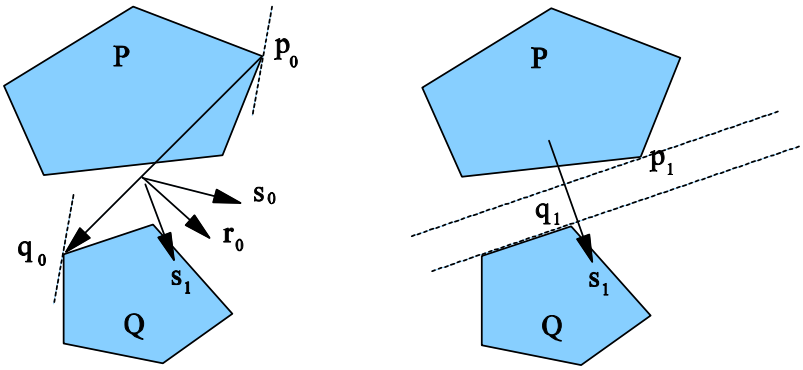


Figura 2.19: Algoritmo de Chung-Wang para encontrar un vector de separación entre los polígonos P y Q. En este caso el vector de separación se encuentra en la segunda iteración del algoritmo.

2.4.6. PIVOT

PIVOT [HZLM01] es un método que utiliza el hardware gráfico para acelerar la detección de colisión entre polígonos 2D. Usa técnicas de visualización en varias pasadas para resolver diversos tipos de cuestiones sobre detección de colisión. Entre este tipo de cuestiones se incluyen el cálculo de la intersección, la distancia de separación, la distancia de interpenetración y los puntos de contacto junto con sus normales.

Es un método adecuado para polígonos cerrados simples, aunque si el polígono no es convexo es necesario realizar una triangulación mediante triángulos disjuntos. Emplea una técnica híbrida que consiste en utilizar la geometría y técnicas en el espacio de la imagen para compensar la carga de trabajo entre la CPU y la GPU. En una primera etapa, relacionada con la geometría, localiza las regiones de intersección potenciales entre dos objetos. En una segunda etapa utiliza técnicas en el espacio de la imagen para calcular información de proximidad en dichas regiones.

Veamos a continuación estas dos etapas:

- **Etapa 1:** Utiliza una rejilla uniforme con puntos de muestreo en las regiones del espacio en las que puede haber intersecciones potenciales. La rejilla utilizada se corresponde con el "frame-buffer" por lo que las consultas implican operaciones a nivel de píxel, y por tanto la eficiencia depende de la resolución. Para evitar una excesiva carga de trabajo, se realiza en primer lugar una fase de localización, en la que se marcan las regiones de intersección en potencia y las de no intersección. Para obtener dichas regiones se utiliza un esquema de jerarquías de cajas envolventes.
- **Etapa 2:** Hay tres tipos de intersecciones posibles entre dos polígonos: frontera-frontera, frontera-volumen, volumen-volumen. En esta etapa se dibujan en primer lugar los dos objetos (A y B) dentro de una de las regiones de interés utilizando el hardware gráfico, considerando como puntos de intersección los píxeles de los puntos de muestreo que son comunes a ambos objetos (aquellos sobre-escritos al dibujarlos). El tipo de intersección determina si se dibuja la frontera o el interior del objeto. Una posible estrategia para llevar a cabo esta etapa (entre otras) consiste en: borrar el "stencil-buffer", dibujar el objeto B incrementando en 1 el "stencil-buffer" para aquellos píxeles que cubre el objeto B, dibujar a continuación el objeto A de manera que para aquellos píxeles con valor 1 en el "stencil-buffer" se incrementen en 1 si cubren al objeto A. Obtenemos así intersección en los puntos del "stencil-buffer" con valor 2. Se pueden utilizar operaciones de histograma para no tener que leer el buffer desde la CPU y optimizar la retroalimentación de esta etapa en caso de que no haya intersecciones.

2.5. Aplicaciones de la Detección de Colisión

Son muchas las aplicaciones en las que es necesario determinar o controlar la detección de colisiones entre objetos. A continuación pasamos a enumerar algunas de ellas según su campo de aplicación:

- **Robótica**: Se trata de encontrar la interferencia entre objetos en el tiempo, considerando éste como continuo. Por este motivo suelen usarse estructuras 4-dimensionales que incluyen el tiempo en sus cálculos.

- **Planificación de Caminos**: Normalmente se determina un camino libre de colisiones en entornos cerrados y controlados en los que el movimiento de los objetos es conocido y modificado por el sistema.

- **Haptics**: Es la ciencia que investiga la percepción por parte del ordenador, de forma similar a como se realiza con el sentido del tacto. Básicamente consiste en obtener determinados puntos de colisión de forma anticipada y con una precisión adecuada. Suele estar relacionada con la presencia de sensores para aplicaciones de robótica. En este tipo de aplicaciones es importante también la realimentación que se produce hacia el usuario en la forma de sensaciones perceptivas.

- **Entornos inmersivos y Realidad Virtual**: Este tipo de aplicaciones están formadas normalmente por un escenario estático, en el que un avatar es introducido en la escena y movido por agentes externos al sistema. Se persigue que el avatar sea capaz de detectar la colisión con los objetos del entorno y, cuando sucede esta colisión, detener su movimiento. Para este tipo de aplicaciones, se suele definir la *detección de colisión de vista* (Figura 2.20) como aquella en la que se comprueba la colisión exclusivamente con objetos que están en un determinado radio de acción y en una determinada dirección respecto del avatar.

- **Diseño asistido por ordenador**: Son muchos los ejemplos en los que cobra importancia la detección de colisión. En sistemas CAD/CAM, se puede utilizar para verificar los espacios libres en un ensamblaje de piezas; o para la revisión de modelos CAD compuestos de millones de primitivas, en los que es necesario manipular diversos componentes o comprobar su accesibilidad, de manera que se puedan alcanzar, manipular y/o extraer cierta parte del modelo para su inspección y/o reparación.

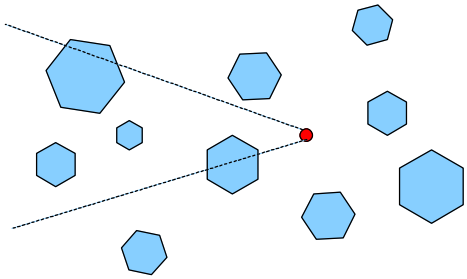


Figura 2.20: Detección de colisión de vista.

- **Fabricación**: Cuando se desea fabricar un producto a partir de un modelo, es necesario prever colisiones en el mundo real de manera anticipada en el modelo, de manera que el producto sea fabricado sin error y sin peligro. En el proceso de fabricación es necesario obtener la máxima calidad y seguridad para los productos. Sería deseable realizar una simulación para poder encontrar fallos en el diseño antes de comenzar la producción. Esta simulación podría predecir ciertas propiedades físicas del producto, así como sus posibles reacciones, y en el caso del movimiento de ciertas piezas mecánicas o de robots, son necesarios métodos eficientes para detectar las posibles colisiones.

- **Simulación de sistemas físicos**: Existen aplicaciones que tratan de simular sistemas físicos, como sistemas de partículas, en los que es necesario una visualización realista e interactiva. Para la simulación de un sistema que imite un proceso físico real, no sólo es necesario visualizar imágenes que parezcan reales con relación al material físico y a las propiedades geométricas de los objetos, sino que también debe ser posible modelar de forma precisa la interacción entre esos objetos. La interacción entre objetos puede causar que choquen entre sí, que reboten, se deformen, etc. Para tratar estas interacciones es necesario que en primer lugar se detecten las colisiones entre objetos y en el caso de que éstas se produzcan determinar los puntos de contacto entre los mismos.

- **Sistemas de Ingeniería**: Puede ser necesario crear una representación para piezas mecánicas, herramientas, máquinas, etc. en las que hay que comprobar aspectos como interconectividad, funcionalidad y seguridad. La meta de estos es la de reducir el tiempo de procesamiento y disminuir los costes de fabricación de los prototipos físicos actuales. La utilización de un simulador gráfico con un sistema de detección de colisiones puede ser también de gran ayuda para analizar sistemas de ingeniería, en los que se pueden simular experimentos que son muy costosos de construir o impracticables.

- **Simuladores**: Los simuladores de vuelo o de coches son aplicaciones destinadas al entrenamiento de usuarios. En estas aplicaciones tanto la detección de colisión como la respuesta a colisión son tareas fundamentales para tratar de simular la realidad. Las restricciones de tiempo son altas y son necesarios algoritmos que funcionen en tiempo real. Por ejemplo, un simulador de choque entre vehículos (crash test) puede conseguir establecer decisiones de diseño basadas en parámetros establecidos en la simulación de colisión y respuesta a colisión.

- **Cine de Animación**: En la animación por ordenador con fines cinematográficos se trata de representar una escena en movimiento y que simplemente sea lo más realista posible. Para eso debe implementar un módulo de detección de colisión, de manera que ante colisiones entre objetos, se actúe de la manera adecuada.

- **Videojuegos**: Los videojuegos han influido también de manera decisiva al desarrollo de los aspectos de detección de colisión, de manera que se consigan interacciones entre objetos más realistas, como sucede en el caso del cine de animación.

- **Simulación de telas**: Es otro área de aplicación, en la que un modelo, en este caso un vestido, es colocado sobre la simulación gráfica de una persona. Este modelo debe asemejar a un vestido real en cuanto a sus características y reacciones, plegándose al igual que lo haría una tela real. Para esto debe implementarse un método de detección de colisión entre el vestido y la persona, uno de autocolisión con el propio vestido, y un método de respuesta a colisión.

Muchas de estas aplicaciones pueden resolverse en 2D ó 2½D. Por ejemplo la planificación de caminos puede realizarse en 2½D, así como algunas aplicaciones de Realidad Virtual o juegos, en las que el avatar se mueve por una planta de un edificio, realizándose una primera aproximación a la detección de colisión en 2½D. Muchas simulaciones pueden realizarse también en 2D, así como el diseño asistido por ordenador de objetos bidimensionales.

2.6. *Librerías específicas para la Detección de Colisión*

En esta sección veremos algunas librerías que implementan métodos de detección de colisión. Sólo hemos considerado las más significativas.

- **I-COLLIDE** [CLMP95] es una librería de detección de colisión para grandes entornos. Utiliza un método de dos fases con cajas envolventes y determina la colisión exacta entre pares de objetos poliédricos convexos. Utiliza la coherencia temporal para acelerar sus cálculos, siendo bastante rápido cuando el movimiento entre frames es pequeño. Su código está disponible en:

  http://www.cs.unc.edu/~geom/I_COLLIDE/Index.html

- **RAPID** [GLM96] es una librería para la detección de colisión entre polígonos no estructurados. No realiza la fase de poda, de la que se encarga la librería V-COLLIDE que veremos a continuación. Es especialmente apropiada por su sencillez de uso y por su rapidez cuando hay pocos objetos en el entorno. El código se encuentra en:

  http://www.cs.unc.edu/~geom/OBB/OBBT.html

- **PQP** [GLM96] es una librería similar a RAPID en cuanto a su interfaz. Es capaz de responder a diversas cuestiones sobre colisión con una cierta tolerancia. Puede usarse con polígonos sin necesidad de información topológica. Utiliza volúmenes basados en esferas de barrido para la determinación de la distancia entre objetos, así como diversos tipos de volúmenes envolventes que pueden ser seleccionados por el usuario. Podemos encontrar su código en:

  http://www.cs.unc.edu/~geom/SSV/index.html

- **V-COLLIDE** [HLCGM97] es una librería construida para la poda previa en el sistema RAPID. Determina los pares de objetos que se encuentran potencialmente en contacto. Estos objetos son utilizados por la librería RAPID para la determinación de colisión. Permite entornos con un gran número de objetos, estáticos y dinámicos, así como la inserción o borrado interactivo de objetos. El código está disponible en:

  http://www.cs.unc.edu/~geom/V_COLLIDE/index.html

- **SWIFT** [EL00] es también una librería para la detección de colisión que hace uso de la coherencia espacial y temporal mediante el uso de regiones de Voronoi y jerarquías de niveles de detalle, utilizando el algoritmo de Lin-Canny de las características más cercanas el cual ha sido mejorado. Se utiliza para poliedros convexos con movimiento rígido. Permite resolver cuestiones como la detección de intersección, la distancia exacta o aproximada entre objetos, y la determinación del contacto entre objetos. Su código podemos encontrarlo en:

  http://www.cs.unc.edu/~geom/SWIFT/

- **SWIFT++** [EL01] es una librería basada en SWIFT. Se le ha incorporado la posibilidad de descomposición jerárquica de los objetos. Genera pares de jerarquías de volúmenes envolventes sobre las que comprobar la colisión con el algoritmo de la librería SWIFT. Su código podemos encontrarlo en:

  http://www.cs.unc.edu/~geom/SWIFT++/

- **H-COLLIDE** [GLGT99] es un entorno de trabajo para la detección de colisión para aplicaciones *"Haptics"*. Está especializada en determinar los contactos entre un dispositivo y objetos en un entorno virtual. Utiliza una adaptación específica de algoritmos clásicos de detección de colisión para este tipo de aplicación. Además usa una descomposición espacial en rejillas uniformes así como una jerarquía de volúmenes envolventes basada en OBB-Trees. También hace uso de la coherencia entre frames para realizar cálculos incrementales. Podemos encontrar más información el la dirección:

  http://www.cs.unc.edu/~geom/H_COLLIDE/index.html

- **IMMPACT** [WLML99] es una librería adecuada para modelos masivos compuestos de millones de primitivas geométricas. Utiliza optimizaciones para el almacenamiento y recuperación del modelo o partes del mismo desde memoria secundaria. Implementa grafos de solapamiento para obtener las zonas del modelo a recuperar de memoria secundaria. Utiliza también jerarquías de volúmenes envolventes y la coherencia espacial y temporal para mejorar su eficiencia. Podemos consultar más información acerca de ésta librería en:

  http://www.cs.unc.edu/~geom/MMC/index.html

- **DEEP** [KLM02] es un algoritmo incremental para estimar el grado de penetración entre politopos convexos, así como la dirección de penetración entre objetos. Hace uso de la coherencia entre frames. Podemos obtener su código de:

  http://www.cs.unc.edu/~geom/DEEP/

- **PIVOT** [HZLM01] es una librería para polígonos no convexos y cerrados en 2D. Utiliza el hardware gráfico y técnicas basadas en regiones de Voronoi para determinar la colisión, la intersección, la distancia de separación y los puntos de contacto. Hace uso de una técnica híbrida en el espacio de los objetos y de la imagen que permite ponderar el uso de la CPU y GPU según las necesidades del usuario. Su código podemos encontrarlo en:

  http://www.cs.unc.edu/~geom/PIVOT/index.html

- **V-CLIP** [Mir98] está basado en el algoritmo de Lin-Canny [LC91] de características más cercanas y trata casos especiales que no maneja el algoritmo de Lin-Canny, además de aportar una solución más simple y robusta. Es un algoritmo de bajo nivel que trata objetos que pueden ser poliedros no convexos, para esto los descompone utilizando una jerarquía de piezas convexas. Podemos obtener más información en:

  http://www.merl.com/projects/vclip

- **SOLID** [Ber04] es también una librería para la detección de colisión entre objetos poliédricos convexos que pueden deformarse. Realiza una implementación eficiente del algoritmo GJK y cajas envolventes orientadas con los ejes como volumen envolvente. Su código podemos encontrarlo en:

  http://www.win.tue.nl/~gino/solid/

- **CULLIDE** [GRLM03] **y Q-CULLIDE** [GLM05] son sistemas de poda para determinar la intersección y la auto-intersección entre objetos complejos y deformables mediante el uso del Hardware Gráfico. Realiza cálculos de visibilidad en la GPU para eliminar conjuntos de primitivas que no están próximas entre si. Podemos obtener más información en las páginas:

  http://www.cs.unc.edu/~geom/CULLIDE y
  http://www.cs.unc.edu/~geom/QCULLIDE

- **GJK** [Cam97]. Podemos encontrar el código del algoritmo GJK para obtener la distancia entre un par de poliedros convexos (en tiempo acotado por una constante) en la siguiente dirección:

  http://web.comlab.ox.ac.uk/oucl/work/stephen.cameron/distances/index.html

- **QuickCD** [KHMSZ98]. Es una librería para la detección de colisión de propósito general, capaz de tratar objetos extremadamente complejos que pueden estar formados por sopas de polígonos. Utiliza una jerarquía de volúmenes envolventes basados en *k*-dops. Podemos obtener esta librería en:

  http://www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html

CAPÍTULO

3

Representación de Objetos

En este capítulo hablaremos del modelado de sólidos y del esquema de representación utilizado para caracterizar los objetos entre los que realizar la detección de colisión. Este esquema de representación está basado en recubrimientos simpliciales, y es adecuado tanto para objetos en dos como en tres dimensiones.

CONTENIDOS:

3. Representación de Objetos

El esquema de representación utilizado para los distintos objetos que componen una escena determina en gran medida el conjunto de técnicas que se pueden utilizar en Detección de Colisiones, así como el diseño de los algoritmos implicados en la Detección de Colisión. Debido a esto, es necesario definir los fundamentos de la representación utilizada.

En este capítulo realizaremos una descripción matemática del concepto de objeto gráfico utilizado en los siguientes capítulos. Para ello nos basaremos en las definiciones dadas por el Dr. Torres [Tor92] y el Dr. Feito [Fei95] en cuanto a álgebra de objetos gráficos y sistema generador de objetos gráficos respectivamente, para obtener un esquema de representación basado en recubrimientos simpliciales [Seg01]. Finalmente hablaremos de cómo hemos utilizado este esquema de representación para el desarrollo de los algoritmos.

3.1. Tipos de Objetos

A continuación vamos a definir los distintos tipos de entidades geométricas utilizadas para la representación de objetos en aplicaciones de detección de colisión.

Consideraremos que un *objeto* es un conjunto no vacío de puntos en el espacio Euclídeo tridimensional. Este conjunto debe ser cerrado y acotado. Cerrado quiere decir que la frontera es considerada como parte del objeto, y acotado que existe una esfera de radio finito que encierra el objeto.

Un objeto es *convexo* si y solamente si todos los segmentos que conectan cualquier pareja de puntos pertenecientes el objeto están contenidos en el propio objeto. Los objetos no convexos son denominados *cóncavos*.

Normalmente los algoritmos de detección de colisión operan con objetos convexos, pues estos algoritmos son más rápidos o simples. Los objetos no convexos suelen ser descompuestos en partes convexas para poder aplicar los algoritmos de detección de colisión (Figura 3.1).

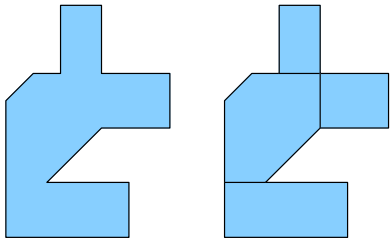


Figura 3.1: Descomposición en piezas convexas.

Podemos decir que uno de los esquemas de representación más ampliamente utilizado para la detección de colisión es el esquema *B-Rep*. Mediante este esquema, los objetos, denominados *poliedros* en el caso 3D, están formados por vértices, aristas y caras. Dependiendo de la morfología del objeto, podemos distinguir entre poliedros convexos y no convexos o cóncavos. Un objeto puede estar formado por caras convexas o no convexas, y ser cóncavo o convexo (Figura 3.2).

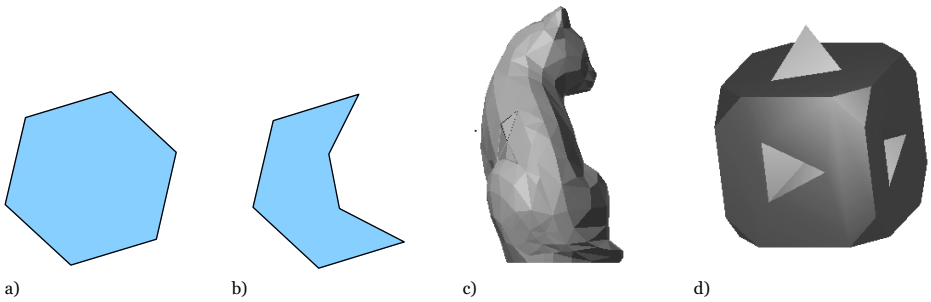


Figura 3.2: Tipos de Objetos. a) Cara convexa. b) Cara no convexa. c) Objeto no convexo formado por caras convexas. d) Objeto no convexo con algunas caras no convexas.

Los algoritmos que vamos a desarrollar en esta memoria utilizan un esquema de representación *B-rep* basado en *recubrimientos simpliciales* que facilita el trabajo con objetos no convexos sin la necesidad de descomponerlos en partes convexas.

3.2. Modelado de Sólidos

Dentro del campo del modelado de sólidos son numerosos los trabajos que se han realizado. Sin embargo, los estudios encaminados hacia la formalización en Informática Gráfica son escasos, teniendo especial importancia el trabajo de Fiume [Fiu89] y la generalización propuesta por Torres [Tor92], que se dirigen hacia aspectos de formalización de la visualización y modelado respectivamente. Torres [Tor92] propone en su trabajo una teoría matemática que puede ser utilizada como metalenguaje para el desarrollo de representaciones abstractas de sistemas gráficos, permitiendo expresar aspectos de la visualización y del modelado. Esta teoría está basada en el concepto de *objeto gráfico*, definido como un *aspecto* y una *presencia*, como veremos después. A partir de esta definición, Torres define un conjunto de operaciones (suma, producto por escalar, unión, intersección, diferencia, producto de objetos y producto circular de objetos) estableciendo una serie de propiedades sobre dicho conjunto y las operaciones definidas.

A partir de los trabajos de Torres [Tor92], Feito [Fei95] establece un sistema formal de modelado basado en el recubrimiento de los sólidos mediante símplices. Para ello, en primer lugar, establece un sistema de generadores basado en objetos gráficos muy sencillos [FT97]. Define un conjunto de operaciones regularizadas sobre dicho sistema de modelado, y propone finalmente una representación basada en el recubrimiento de los sólidos mediante símplices, creando una estructura en árbol similar a la de la representación basada en CSG.

En este trabajo utilizaremos el esquema de representación de objetos basado en recubrimientos simpliciales de Feito [Fei95]. Mediante este esquema de representación, descompondremos un sólido 2D y 3D mediante símplices que pueden solaparse en el espacio. Usando esta representación desarrollaremos algoritmos específicos para la Detección de Colisión, de manera que aprovechen las particularidades de este tipo de representación de objetos.

A continuación veremos la definición de *álgebra de objetos gráficos* en la que se basa el esquema de representación mediante recubrimientos simpliciales. Una vez desarrollada esta teoría veremos el esquema de representación de objetos en primer lugar para el caso 2D y después para el caso 3D.

3.3. Álgebra de Objetos Gráficos

Veamos a continuación la definición de *objeto gráfico* según Torres [Tor92]. Para ello es necesario definir el concepto de *ζ-estructura*.

Definición 3.1: Una *ζ-estructura* sobre un campo K es un conjunto Z en el cual se definen las siguientes operaciones.

- Una operación interna llamada suma, y que notaremos por +, con la cual Z es un grupo conmutativo.
- Una operación externa sobre el campo K, y que notaremos por *, llamada producto por escalar, de manera que $(Z,+,*)$ es un espacio vectorial.
- Tres operaciones internas llamadas unión, intersección y complementario (unaria), y que notaremos por $\cup$, $\cap$ y $\sim$ respectivamente, de manera que Z con esas operaciones es un álgebra de Boole.
- Una operación interna que llamaremos producto, y que notaremos por $\times$, que satisface las propiedades asociativa, conmutativa y elemento neutro.

La definición de objeto gráfico contempla dos factores: la *presencia* y el *aspecto*. El aspecto se refiere a propiedades del objeto gráfico desde el punto de vista visual (color, transparencia,...) así como otros atributos no geométricos (densidad, tipo de material, etc.), mientras que la presencia describe la ocupación de espacio por parte del objeto gráfico, es decir, la multiplicidad con la que un objeto ocupa el espacio. Estos dos factores se definen como dos *ζ-estructuras* llamadas *espacio de aspecto* (o *dominio de aspecto*) θ y *espacio de presencia* (o *dominio de presencia*) π.

Veamos a continuación cómo definir un objeto gráfico en base a los conceptos de aspecto y presencia.

Definición 3.2: Un *volumen* V se define como un subconjunto de $\Re^n$. El volumen de un objeto gráfico es la porción de espacio ocupada por el objeto. Definimos *universo de objetos gráficos*, U, como una tripleta U=(π,θ,n), siendo n la dimensión del espacio euclídeo [TC93].

Definición 3.3: Definimos *objeto gráfico* en el universo U como un par (μ,σ), donde μ es la *función de presencia* definida como $\mu:\Re^n \rightarrow \pi$, y σ es la *función de aspecto* definida como $\sigma:\Re^n \rightarrow \theta$.

En base a esta definición, podemos especificar, por ejemplo, un triángulo de la siguiente manera:

$Triangle(P_1,P_2,P_3) = (\mu,\sigma)$ con
$\mu(P) = 1$ *if* $\exists a,b,c \in [0,1]$, $a+b+c = 1$ / $P = a \cdot P_1 + b \cdot P_2 + c \cdot P_3$; 0 *en otro caso*
$\sigma(P) = k \cdot \mu(P)$, *donde k es el valor de aspecto del triángulo.*

Torres [TC93] demostró que el conjunto de objetos gráficos es una *ζ-estructura*, por lo que se definían todas las operaciones de las *ζ-estructuras*.

A partir de las definiciones de objeto gráfico que acabamos de señalar, Torres [TC93] desarrolló un *Álgebra de Objetos Gráficos*, definiendo para ello operaciones regularizadas que permitían mantener la regularidad de los objetos. Con dichos resultados, y aprovechando el resultado de que una *ζ-estructura* con las operaciones + y * es un espacio vectorial, Feito [FT97] desarrolló un sistema generador de objetos gráficos y demostró su validez. Para ello, utilizó símplices originales. En lo que sigue, supondremos que todas las operaciones entre sólidos (o símplices) son operaciones regularizadas, en el sentido de que devuelven un conjunto de puntos cerrado y dimensionalmente homogéneo.

3.4. Recubrimientos Simpliciales

La idea básica de Feito [FT97] consiste en descomponer un polígono en pequeñas piezas similares y utilizar el espacio vectorial de los objetos gráficos para representar un polígono como una combinación de componentes básicos. Así obtiene un sistema generador para polígonos basado en información de la frontera. Otros autores han utilizado símplices para representar sólidos, como [Dey91] y [PRS93]. Sin embargo estos autores utilizan triangulaciones en lugar de recubrimientos.

Utilizando el recubrimiento de un objeto, de manera que podamos generarlo a partir de símplices, lo aplicaremos a objetos gráficos cuyo volumen determine un sólido. Supondremos que los objetos son uniformes, es decir, que tienen presencia y aspecto constante, y que el espacio de presencia es normal, o lo que es lo mismo, que los posibles valores que devuelve la función de presencia son cero y uno.

A continuación definiremos el concepto de *símplice*. Para ello es necesario definir previamente la clausura convexa de un conjunto de puntos.

Definición 3.4: Se define *clausura convexa* de *d+1* puntos $(P_0, P_1, P_2, ..., P_d \in \Re^n)$ como el conjunto de las combinaciones lineales de todos los puntos, esto es,

$$S=\{ P / P= \lambda_0 P_0 + \lambda_1 P_1 + \lambda_2 P_2 + ... + \lambda_d P_d \, , \lambda_d \geq 0 \wedge \sum_{i=0}^{d} \lambda_i = 1 \}$$

Definición 3.5: Definimos un *d-símplice* como la clausura convexa de *d+1* puntos que son linealmente independientes.

Podemos ver que un *0-símplice* es un punto, un *1-símplice* es una línea, un *2-símplice* es un triángulo, y un *3-símplice* es un tetraedro.

A partir de ahora utilizaremos indistintamente el término *d*-símplice y el término específico punto, línea, triángulo o tetraedro según corresponda.

El orden en el que aparecen los puntos en la definición del símplice no cambia el mismo, pero puede hacer que cambie el resultado de cálculos realizados según la secuencia de los puntos. Por tanto, es importante fijar un orden para la secuencia de puntos en el *d*-símplice.

Definición 3.6: Un *d-símplice orientado* es un *d*-símplice en el que los puntos están dados en una sucesión específica.

Definición 3.7: Se dice que dos *d*-símplices orientados tienen la misma orientación cuando la secuencia de puntos de uno de ellos puede ser obtenida de la del otro mediante una permutación de orden par.

Por ejemplo para un 2-símplice las dos posibles orientaciones son *ABC* y *ACB*.

Para la definición de recubrimiento simplicial es necesario definir la función *signo*. Esta función será ampliamente utilizada en el resto de capítulos.

Definición 3.8: Sea w un número real. Definimos la función signo, *sign (w)*, como sigue:

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

Esta función signo nos permitirá realizar operaciones más robustas [She96] [She97] cuando la comparación con el valor *0* se realice para un intervalo $[-\varepsilon,+\varepsilon]$ como veremos en los algoritmos desarrollados en esta memoria.

Un recubrimiento puede aplicarse a sólidos homogéneos y no-variedad, recordemos dichos conceptos.

Definición 3.9: Un conjunto S se dice que es *dimensionalmente homogéneo* si todos los puntos de S pertenecen al interior de S, o bien a la frontera de S.

Intuitivamente, S es dimensionalmente homogéneo si no tiene puntos o aristas aisladas.

Definición 3.10: Un sólido F se dice que es *variedad* si para cada punto de la superficie del sólido existe un entorno de dicho punto de radio $\delta>0$ tal que dicho entorno es homeomorfo a un disco en $\Re^2$ [Man88].

Veamos a continuación el concepto de recubrimiento simplicial en 2 y 3 dimensiones.

3.4.1. Recubrimientos Simpliciales en 2D

Para establecer el teorema que define el recubrimiento simplicial en 2D mediante triángulos es necesario definir el concepto de área signada.

Definición 3.11: Sean tres puntos $A,B,C \in \Re^2$, con coordenadas $A(x_A,y_A)$, $B(x_B,y_B)$, $C(x_C,y_C)$; el *área signada* del triángulo ABC se define como [Hof89]:

$$|ABC| = \frac{1}{2} * \begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix}$$

El área signada de este triángulo puede ser negativa, dependiendo del orden de los puntos. Para un orden contrario a las agujas del reloj puede demostrarse fácilmente que el área signada es positiva y cero si los puntos están alineados.

A continuación mostramos el teorema que da soporte a la representación de objetos gráficos mediante recubrimientos simpliciales en 2D.

Teorema 3.1: Cualquier objeto gráfico de aspecto y presencia constante, variedad o no-variedad, con o sin agujeros en 2D, puede representarse mediante una expresión de objetos gráficos de la siguiente forma [FT97]:

Un polígono 2D con k vértices, F, con presencia constante μ_o y aspecto constante σ_o, delimitado por la secuencia de aristas $\{E_i,\ i=1,\ldots,k\}$ dadas en una orientación conforme, con $E_i=\{Q_i,Q_{i\oplus 1}\}$, puede ser expresada como:

$$F = \sum_{i=1..k} [sign(S_i) * S_i(\mu_o, \sigma_o)]$$

donde S_i es el triángulo $OQ_iQ_{i\oplus 1}$ y O es cualquier punto de referencia prefijado (O puede ser el centroide del polígono). La demostración puede verse en [FT97].

Este teorema establece una representación para cualquier polígono basada en el álgebra de objetos gráficos, utilizando elementos muy simples. Además, el número de términos de la expresión es el número de aristas para el caso 2D.

Definición 3.12: Sea F un polígono, el *recubrimiento del polígono* (Figura 3.3), denominado C_F, es el conjunto de 2-símplices obtenidos uniendo un punto arbitrario *O* en el plano de F con cada una de las aristas del polígono. A este punto arbitrario *O* lo llamaremos origen del recubrimiento y de los 2-símplices.

En esta memoria utilizaremos el centroide como origen del recubrimiento de un sólido 2D. Por tanto al referirnos al recubrimiento del polígono lo haremos en base al centroide.

Definición 3.13: Llamaremos *aristas originales* o *interiores* a las aristas de los triángulos del recubrimiento que incluyen al origen del recubrimiento, y llamaremos *aristas no originales* o *exteriores* a las aristas de los triángulos del recubrimiento que no incluyen dicho origen.

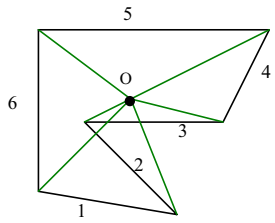


Figura 3.3: Recubrimiento de un polígono. Las aristas con vértice en *O* son las arista interiores. Las aristas que no contienen al punto *O* (numeradas) son las aristas exteriores. El origen de todas las aristas interiores es el punto *O*.

Podemos definir un triángulo como positivo o negativo dependiendo del signo de su área signada (Figura 3.4).

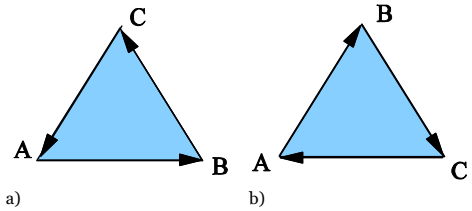


Figura 3.4: Orientación de un triángulo. a) Positivo. b) Negativo.

Definición 3.14: Un triángulo T es *positivo* si *sign (*|T|*)*=1, y *negativo* si *sign (*|T|*)*= -1. En caso de que *sign (*|T|*)*=0 diremos que el triángulo es *degenerado*.

3.4.2. Recubrimientos Simpliciales en 3D

Veamos a continuación la ampliación del Teorema 3.1 para poder definir el recubrimiento simplicial en 3D mediante tetraedros. Para ello es necesario definir previamente el concepto de volumen signado.

Definición 3.15: Sean cuatro puntos $A,B,C,D \in \Re^3$, con coordenadas $A(x_A,y_A,z_A)$, $B(x_B,y_B,z_B)$, $C(x_C,y_C,z_C)$, $D(x_D,y_D,z_D)$; el volumen signado del tetraedro $ABCD$ se define como [Hof89]:

$$|ABCD| = 1/6 * \begin{vmatrix} x_A & y_A & z_A & 1 \\ x_B & y_B & z_B & 1 \\ x_C & y_C & z_C & 1 \\ x_D & y_D & z_D & 1 \end{vmatrix}$$

El volumen signado de este tetraedro puede ser negativo, dependiendo del orden de los puntos. Si para cada tres puntos, están dados en orden contrario a las agujas del reloj, puede demostrarse fácilmente que el volumen signado es positivo, y cero si existen tres puntos que se encuentren alineados.

El siguiente teorema da soporte a la representación de objetos gráficos mediante recubrimientos simpliciales en 3D.

Teorema 3.2: Cualquier objeto gráfico de aspecto y presencia constante, variedad o no-variedad, con o sin agujeros y homogéneo en 3D puede representarse mediante una expresión de objetos gráficos de la siguiente forma [FT97]:

Un poliedro 3D con n caras, F, con presencia constante μ_o y aspecto constante σ_o, delimitado por la secuencia de caras $\{H_i, i=1,...,n\}$, con $H_i=\{E_{i,j}, j=1,...,l_i\}$ dadas en una orientación conforme (el vector normal de cada cara apunta hacia el exterior del sólido), con $E_{i,j}=(Q_{i,j},Q_{i,j\oplus 1})$, siendo $Q_{i,j}$ el vértice j de la cara i, puede ser expresada como:

$$F = \sum_{i=1}^{n} \left(\sum_{j=1}^{l_i} sign(S_{ij}) * S_{ij}(\mu_0, \sigma_0) \right)$$

donde S_{ij} es el 3-símplice $OW_iQ_{i,j}Q_{i,j\oplus 1}$, siendo W_i cualquier punto de referencia prefijado y situado en el mismo plano que la cara H_i (por ejemplo el centroide de la cara), y siendo O cualquier punto de referencia (por ejemplo el centroide del poliedro). La demostración puede verse en [FT97].

Este teorema establece una representación para cualquier poliedro basada en el álgebra de objetos gráficos, utilizando elementos muy simples, en nuestro caso tetraedros orientados. Además, el número de términos de la expresión es la suma de las aristas de cada cara del poliedro.

Definición 3.16: Sea F un poliedro, el *recubrimiento del poliedro* (Figura 3.5), denominado C_F, es el conjunto de 3-símplices obtenidos uniendo un punto arbitrario *O*, y los triángulos obtenidos en el recubrimiento de cada cara del poliedro. A este punto arbitrario *O* lo llamaremos *origen del recubrimiento* y de los 3-símplices. Al punto arbitrario sobre cada cara W_i, lo llamaremos *origen del recubrimiento de cada cara*, o punto interno de la cara.

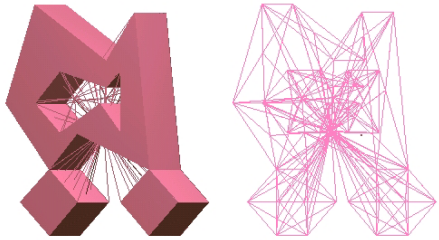


Figura 3.5: Recubrimiento de un poliedro.

En esta memoria utilizaremos el centroide como origen del recubrimiento de un sólido. Por tanto al referirnos al recubrimiento del poliedro lo haremos en base al centroide del mismo. A su vez, cada cara del poliedro estará recubierta mediante triángulos con origen en el centroide de la cara.

Al igual que hicimos en el caso 2D, daremos distintos nombres a las aristas y caras de los tetraedros del recubrimiento de un objeto gráfico, como podemos ver en la Tabla 3.1 y en la Figura 3.6.

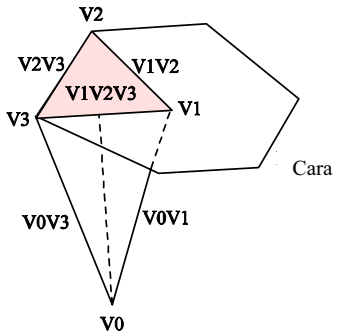


Figura 3.6: Denominación de las partes de un tetraedro del recubrimiento de un poliedro.

Características	Nombre
Vértice V_0	Vértice original
Vértice V_1	Vértice no original, origen de la cara, vértice interno de la cara
Vértice V_2	Vértice no original
Vértice V_3	Vértice no original
Arista V_0V_1	Arista original o interior
Arista V_0V_2	Arista original o interior
Arista V_0V_3	Arista original o interior
Arista V_1V_2	Arista no original o exterior
Arista V_2V_3	Arista no original o exterior; Arista de la cara
Arista V_3V_1	Arista no original o exterior
Cara $V_1V_2V_3$	Cara (o triángulo) no original o exterior
Cara $V_3V_2V_0$	Cara (o triángulo) original o interior
Cara $V_3V_0V_1$	Cara (o triángulo) original o interior
Cara $V_0V_1V_2$	Cara (o triángulo) original o interior
Tetraedro $V_0V_1V_2V_3$	3-Símplice o Tetraedro

Tabla 3.1: Denominación de los elementos de un tetraedro del recubrimiento de un poliedro según puede verse en la Figura 3.6.

Definición 3.17: Las aristas de los tetraedros del recubrimiento que incluyen el origen del recubrimiento las llamaremos *aristas originales* o *interiores*. Las aristas de los tetraedros del recubrimiento que no incluyen dicho origen las llamaremos *aristas no originales* o *exteriores*. La arista no original de los tetraedros del recubrimiento que no incluye el origen del recubrimiento de su correspondiente cara la llamaremos *arista de la cara* del poliedro.

Definición 3.18: Las caras de los tetraedros del recubrimiento que incluyen el origen del recubrimiento las llamaremos *triángulos originales* o *interiores*. La cara de los tetraedros del recubrimiento que no incluye dicho origen la llamaremos *triángulo no original* o *exterior*.

El motivo de utilizar estos nombres no es otro que el de facilitar la comprensión de los algoritmos que se han desarrollado en esta memoria, pues la forma de actuar, por ejemplo, cuando un punto se encuentra en la frontera, es distinta dependiendo del vértice, arista o cara concreta sobre la que se encuentre.

También podemos definir un tetraedro como positivo o negativo dependiendo del signo de su volumen signado (Figura 3.7).

Definición 3.19: Un tetraedro T es *positivo* si *sign (*|T|*)*=1, y *negativo* si *sign (*|T|*)* = -1. Si *sign (*|T|*)*=0 diremos que el tetraedro es *degenerado*.

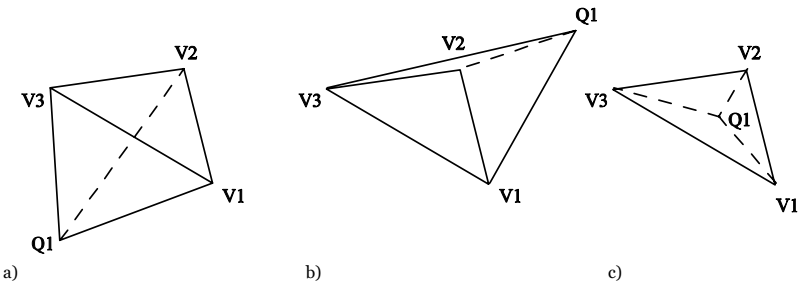


Figura 3.7: Orientación de un tetraedro. a) Positivo. b) Negativo. c) Degenerado.

3.4.3. Propiedades

Podemos ver que el hecho de utilizar recubrimientos y no triangulaciones (Figura 3.8) presenta numerosas ventajas. En primer lugar, el recubrimiento de un polígono mediante triángulos puede obtenerse directamente de la representación del polígono, pues sólo es necesario decidir cual será el punto origen del recubrimiento. Los triángulos del recubrimiento están formados por dicho origen y los dos vértices de cada una de las aristas del polígono. En cambio, realizar una triangulación del mismo polígono conlleva un coste de pre-procesamiento que suele ser de orden *O(n log n),* aunque en [Cha91] se presenta un algoritmo de orden lineal, que no ha llegado a implementarse de manera efectiva.

Por otra parte, se unifica el tratamiento que se hace de los objetos gráficos en espacios *n*-dimensionales, lo que permite que los algoritmos presentados en una dimensión sean fácilmente extensibles a dimensiones superiores. De hecho, y como veremos más adelante, los algoritmos de inclusión y de detección de colisión de puntos en poliedros no son más que una generalización de la inclusión y detección de colisión de puntos en polígonos, cambiando únicamente el espacio del símplice de 2-símplice a 3-símplice.

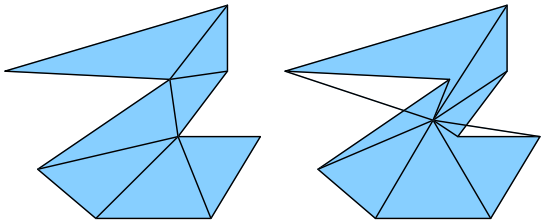


Figura 3.8: Diferencia entre triangulación y recubrimiento.

Además, el sistema generador desarrollado por Feito es válido para cualquier tipo de sólido poliédrico, ya sea variedad o no-variedad, con o sin agujeros, cóncavo o convexo, etc. (Figura 3.9). Ello proporciona un mecanismo de modelado en el que los casos especiales no deben ser tratados de forma independiente, sino que los algoritmos desarrollados se muestran sencillos de implementar al no haber distinción entre los distintos tipos de sólidos. Además, el sistema generador es válido tanto para sólidos homogéneos como para sólidos heterogéneos. Para el tratamiento de los sólidos compuestos por en un número finito de componentes regulares, basta con descomponerlo en un número finito de objetos homogéneos, los cuales pueden expresarse como una combinación de los objetos gráficos del sistema generador.

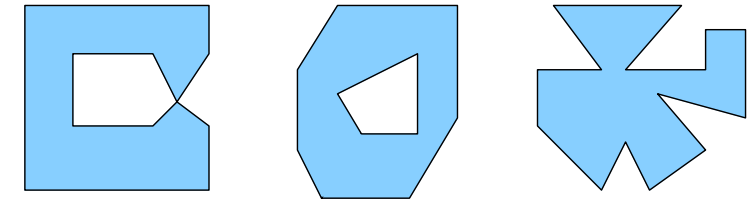


Figura 3.9: Ejemplos de sólidos permitidos. a) No variedad. b) Con agujeros. c) Cóncavo.

3.5. Representación de objetos mediante recubrimientos simpliciales

En esta sección describiremos la estructura de datos utilizada en la representación de los objetos* tanto en dos dimensiones como en tres dimensiones.

3.5.1. Representación de objetos en 2D

Un sólido† 2D es un polígono plano simple. Para representar un polígono plano existen diversas soluciones. Hemos optado por una representación que en primer lugar facilite las operaciones mediante símplices. Además buscamos una representación que ocupe poco espacio en memoria, rápida de calcular y que sea compatible con otro tipo de representaciones.

---

* Nos referimos a un objeto gráfico. Emplearemos indistintamente este término para referirnos a objetos gráficos bidimensionales o tridimensionales, realizando esta distinción por el contexto.

† Hablaremos de sólidos en 2D y en 3D para referirnos a polígonos planos simples y a poliedros formados por caras planas simples respectivamente.

Definición 3.20: Un *polígono* plano H es una secuencia de aristas $\{E_0, E_1, ..., E_n\}$, cerrada, es decir, el vértice final de la arista E_n coincide con el vértice inicial de la arista E_0.

Definición 3.21: Un *polígono* plano H' diremos que es *simple* si no se produce intersección entre sus aristas, salvo en los extremos de las mismas, es decir, en los vértices.

Utilizaremos para la representación del polígono plano simple la secuencia de vértices $V_0,V_1,...V_n$ que forman las aristas del polígono, entendiendo que la arista E_i está formada por los vértices $V_iV_{i\oplus 1}$.

Esta definición junto al Teorema 3.1 nos proporciona una estructura de datos para la representación de sólidos en 2D. Esta estructura está formada por:

- Una *lista de vértices* para representar la *geometría* del objeto. Estará formada por los vértices de las aristas, evitando la duplicidad de los mismos para un almacenamiento óptimo, pero no siendo un requisito indispensable para la representación.
- Una *lista de triángulos* para representar la *topología* del objeto, no necesariamente disjuntos, que se formarán a partir de un punto arbitrario y tomando los vértices de una arista del polígono. Utilizaremos el centroide del polígono como punto de referencia, debido sobre todo a que el reparto de área entre las aristas se realiza de forma equitativa.

Por tanto, necesitamos una lista de vértices y una lista de triángulos. La lista de vértices se utilizará para formar la lista de triángulos. Esta lista de triángulos se formará mediante una referencia a un punto arbitrario que será común a todos los triángulos, en nuestro caso, el centroide del polígono, y referencias a un par de vértices. La lista de triángulos, por comodidad, se dará ordenada y encadenada, es decir, se dará la lista de referencias a vértices ordenada conforme al contorno del polígono. Éste no es un requisito indispensable, pues sólo es necesario conocer la lista de triángulos en cualquier orden, pero mediante esta representación se ahorra espacio de almacenamiento. Veamos algunos ejemplos:

- Dada la lista de vértices $V_1,V_2, ..., V_n$, y el punto de referencia O, formaremos la lista de triángulos con los índices $1,2,...,n$, de manera que representa los triángulos OV_1V_2, OV_2V_3, ..., OV_nV_1. En este caso el orden de los vértices coincide con el orden de los triángulos.
- Para otro polígono con la misma lista de vértices podríamos haber formado la lista de triángulos mediante los índices $2,3,1,4,5$, por ejemplo, con lo que representaríamos a los triángulos OV_2V_3, OV_3V_1, OV_1V_4, OV_4V_5 y OV_5V_2.

El sentido en el que se dan los vértices en la lista de triángulos nos da la orientación del polígono en el plano en el que se encuentra. Si se dan en sentido anti-horario la normal va hacia fuera del polígono.

3.5.2. Representación de objetos en 3D

Un sólido en 3D podemos representarlo mediante la frontera. Para eso podemos dar un conjunto de polígonos planos simples de manera que el volumen acotado sea cerrado.

Definición 3.22: Un *poliedro* F en 3D es un conjunto de puntos cerrado, limitado por un conjunto de caras planas, $H=\{H_0, H_1, ..., H_n\}$. Cada una de estas caras será un polígono simple de acuerdo con la Definición 3.21.

Esta definición junto al Teorema 3.2 nos proporciona una estructura de datos para la representación de sólidos en 3D. Esta estructura está formada por:

- Una *lista de vértices* para representar la *geometría* del objeto. Estará formada por los vértices de las aristas de todas las caras, evitando la duplicidad de los mismos para un almacenamiento óptimo, no siendo, al igual que en el caso 2D, un requisito indispensable para la representación.

- Una *lista de caras* para representar la *topología* del objeto. Cada cara estará formada por un conjunto de tetraedros no necesariamente disjuntos, que se formarán a partir de un punto arbitrario y tomando los triángulos del recubrimiento de cada cara del poliedro. Utilizaremos el centroide del poliedro como punto de referencia común para formar los tetraedros.

Por tanto, necesitamos una lista de vértices y una lista de caras. La lista de vértices se utilizará para formar la lista de caras, cada una compuesta por un conjunto de tetraedros. Estos tetraedros se formarán mediante una referencia a un punto arbitrario que será común a todos los tetraedros, en nuestro caso, el centroide del poliedro y referencias a tres vértices (dos vértices de la arista de una cara y el punto de referencia usado en el recubrimiento de la cara). Utilizaremos al igual que en el caso 2D una lista de referencias a vértices ordenada para formar los tetraedros con los vértices de cada cara. Podríamos utilizar distintas listas para representar los tetraedros que pertenecen a una cara, pero utilizaremos una única lista, con un elemento nulo de separación entre caras (-1 para el caso de una implementación mediante índices). Veamos algún ejemplo de representación:

- Dada la lista de vértices $V_1,V_2, ..., V_n$, el centroide del poliedro O y el centroide de las caras $O_1, O_2, O_3, ...$, formaremos la lista de caras con los índices *1,2,3,4,-1,3,4,6,8,-1,4,6,3,-1,...* de manera que represente a las caras F_1, F_2, F_3, ..., mediante los tetraedros $F_1 = \{OO_1V_1V_2, OO_1V_2V_3, OO_1V_3V_4, OO_1V_4V_1\}$, $F_2 = \{OO_2V_3V_4, OO_2V_4V_6, OO_2V_6V_8, OO_2V_8V_3\}$, $F_3 = \{OV_4V_6V_3\}$, ...

Debemos considerar el caso en el que un polígono está formado por sólo 3 vértices. En este caso no es necesario realizar un recubrimiento del polígono, sino que utilizamos el triángulo formado por esos tres vértices como único triángulo del recubrimiento del polígono, sin aumentar el número de triángulos. En el ejemplo anterior se ha utilizado este triángulo para la cara F_3. Esto es muy útil cuando se utilizan mallas de triángulos como representación ya que no se aumenta inútilmente la cantidad de información a almacenar, reduciendo los cálculos necesarios para la realización de diversas operaciones.

Existen situaciones especiales que pueden ser tratadas con esta representación. Por ejemplo, cuando una cara está formada por más de un bucle, es decir, contiene agujeros. En esta situación es necesario incluir aristas *"falsas"*, es decir, aristas que no pertenecen a la frontera de la cara, pero que se utilizan de manera auxiliar para conectar varios bucles en una cara. En la Figura 3.10 podemos ver cómo introducir una arista *"falsa"* para conectar dos bucles. Esta arista falsa puede ser cualquiera que una dos vértices, uno de cada bucle. Dicha arista se introduce dos veces en la lista, una para cada sentido de su recorrido. En la figura anterior podemos definir la lista del siguiente modo: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,7,6,-1.

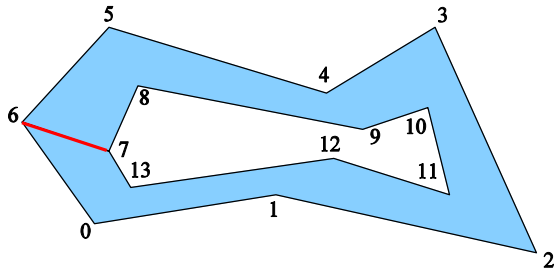


Figura 3.10: Figuras con agujeros, introducción de aristas falsas (en rojo).

La introducción de aristas falsas en la representación permite definir agujeros en el interior de los polígonos sin la necesidad de introducir otros conceptos como el de bucle, así es posible representar gráficamente este tipo de objetos. Sin embargo, a la hora de utilizar la representación en algoritmos de detección de colisión, estas aristas falsas son ignoradas y no tratadas por ninguno de los algoritmos, pues en la mayor parte de estos, para acelerar los cálculos, se considera inclusión o intersección entre un elemento y un polígono cuando se da en su frontera, y estas aristas falsas no pertenecen a la misma.

CAPÍTULO

4

Detección de Colisiones 2D

Antes de afrontar la Detección de Colisiones entre Sólidos tridimensionales, se estudiarán algoritmos para casos 2D. Se utilizarán recubrimientos simpliciales y posteriormente un nuevo tipo de descomposición espacial basada en tri-conos denominada tri-tree. Este estudio en 2D facilitará no sólo la comprensión de las técnicas desarrolladas en el próximo capítulo dedicado a la detección de colisión 3D, sino que será la base para multitud de aplicaciones, donde ciertos problemas pueden acometerse mediante algún tipo de simplificación bidimensional.

CONTENIDOS:

4. Detección de Colisiones 2D

Existen problemas de naturaleza bidimensional que necesitan de una detección de colisión (DC) en dos dimensiones. Además, muchas aplicaciones que operan en el espacio tridimensional pueden simplificarse, de manera que es posible encontrar una solución que opere en dos dimensiones. En otros tipos de aplicaciones pueden utilizarse algoritmos 2D como paso previo a una detección de colisión exacta entre objetos de dimensión mayor. Además de lo anterior, consideramos que es conveniente desarrollar una teoría que actúe sobre modelos 2D, mucho más simples, para así extender y obtener una teoría adecuada y correcta sobre detección de colisión en 3D.

En este capítulo desarrollaremos los conceptos básicos y algoritmos necesarios para la detección de colisión en 2D. Describiremos algunas de las relaciones existentes entre puntos, segmentos, triángulos y polígonos complejos[*], así como las propiedades que nos permitan el uso de la coherencia en los algoritmos. Veremos diversos algoritmos para la inclusión, intersección y detección de colisión entre varios tipos de objetos representados mediante recubrimientos simpliciales, en primer lugar sin realizar ningún tipo de descomposición, y finalmente utilizando una nueva descomposición espacial basada en conos triangulares o *tri-conos,* denominada árbol de tri-conos o *tri-tree*. Mediante tri-trees descompondremos los objetos en partes más simples sobre las que realizar la detección de colisión.

Nos centraremos en la detección de colisión a bajo nivel (fase estrecha). La fase de poda o fase ancha se ha implementado utilizando un esquema de circunferencias

[*] Nos referimos al Sólido 2D definido en el Capítulo 3 (polígono simple). Hemos utilizado la denominación de *polígono complejo* para enfatizar el hecho de que el polígono puede ser cóncavo, con agujeros o ser no-variedad.

envolventes de manera que se descartan los pares de objetos más distantes entre sí, aunque pueden utilizarse otras alternativas como puede verse en [OJF04].

Vamos a emplear un enfoque constructivo a lo largo de este capítulo, mediante el cual iremos exponiendo cada uno de los algoritmos y técnicas necesarias para el desarrollo de algoritmos auxiliares, y finalmente de los algoritmos de detección de colisión.

Comenzaremos este capítulo con la caracterización de la inclusión y la posición relativa de un punto en relación a un triángulo, operación básica que será utilizada en última instancia por la mayor parte de los algoritmos aquí expuestos. A continuación mostraremos el desarrollo de los algoritmos de detección de colisión entre un punto y un polígono, una circunferencia y un polígono y entre polígonos. Para el caso de colisión entre un punto y un polígono se han obtenido algoritmos específicos para distintos tipos de polígonos: convexos, con recubrimiento positivo y no convexos.

Una vez expuestos estos algoritmos, se describe el nuevo tipo de descomposición espacial utilizada basada en tri-conos, mediante la cual simplificaremos el número de primitivas a tratar. Tras esto mostraremos los algoritmos de detección de colisión utilizando este tipo de descomposición. Por último realizaremos un estudio temporal de los algoritmos desarrollados. En este estudio mostraremos también la eficiencia del método de descomposición espacial utilizado.

4.1. Caracterización de la inclusión de puntos en triángulos

Según vimos en el Capítulo 3, utilizaremos una representación basada en recubrimientos simpliciales. En el caso 2D, el recubrimiento del sólido se realiza mediante triángulos, si bien utilizaremos como origen del recubrimiento el centroide[*] del objeto. Mediante esta representación limitamos las operaciones necesarias para la detección de colisión a operaciones entre triángulos, entre triángulos y segmentos, o entre triángulos y puntos. Por tanto, comenzaremos caracterizando la posición de un punto respecto a un triángulo que será una operación básica para todos los algoritmos desarrollados.

Hay distintos métodos para obtener la inclusión de puntos en triángulos. La utilización de las coordenadas baricéntricas de un punto respecto a un triángulo es

[*] Salvo que digamos lo contrario, todos los recubrimientos se han realizado utilizando el centroide del polígono como origen. Todas las definiciones y algoritmos se basan en un recubrimiento de este tipo.

un método sencillo y eficiente para determinar la relación* del punto respecto a ese triángulo. Otros autores [FT97b] [SFMOT05] han utilizado áreas signadas de triángulos para establecer estas relaciones. A continuación vamos a desarrollar una teoría basada en coordenadas baricéntricas que generaliza el método de áreas signadas anterior.

Para clasificar un punto respecto de un triángulo utilizaremos tres valores únicos que representan las coordenadas baricéntricas de un punto respecto a un triángulo en el espacio 2D. A lo largo de este capítulo supondremos que puntos, aristas y triángulos están definidos en $\Re^2$.

Partiremos de la definición clásica de coordenadas baricéntricas signadas de un punto respecto de un triángulo [Eri05].

Definición 4.1: Sean V_0,V_1,V_2 los vértices ordenados de un triángulo de área no nula. Sea P otro punto en $\Re^2$. Entonces existen unos únicos valores $\alpha,\beta,\gamma \in \Re$, de manera que cumplen que $\alpha + \beta + \gamma = 1$ y que $P = \alpha V_0 + \beta V_1 + \gamma V_2$. Estos valores son:

$$\alpha = |PV_1V_2| \,/\, |V_0V_1V_2|$$
$$\beta = |PV_2V_0| \,/\, |V_0V_1V_2|$$
$$\gamma = |PV_0V_1| \,/\, |V_0V_1V_2|$$

Los valores α,β y γ son las *coordenadas baricéntricas signadas* del punto P respecto a los puntos V_0, V_1 y V_2 (Figura 4.1). En adelante las llamaremos simplemente *coordenadas baricéntricas* de un punto en relación a un triángulo.

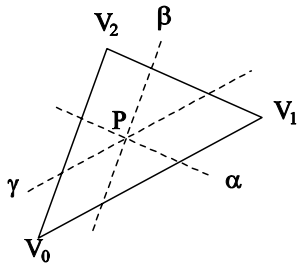


Figura 4.1: Coordenadas baricéntricas de un punto P respecto a un triángulo $V_0V_1V_2$.

Podemos determinar si un punto se encuentra o no dentro de un triángulo utilizando sus coordenadas baricéntricas, para ello podemos utilizar la siguiente propiedad:

* La relación entre un punto y un triángulo se refiere no sólo a su inclusión, sino a la parte del triángulo donde se produce dicha inclusión, como puede ser en un vértice o arista concretos.

Propiedad 4.1: Sea P un punto con coordenadas baricéntricas (α,β,γ) respecto al triángulo formado por los puntos V_0,V_1,V_2. El punto P está dentro del triángulo definido por los puntos V_0,V_1,V_2 si y solo si $0 \leq \alpha, \beta, \gamma \leq 1$.

Se puede realizar una interpretación geométrica de la Propiedad 4.1 para comprobar en qué posición se encuentra un punto respecto a un triángulo. Para llevar a cabo esta interpretación examinamos el signo de los determinantes que definen las coordenadas baricéntricas del punto. Dependiendo de la orientación del triángulo tendremos dos situaciones distintas:

- Si el triángulo $V_0V_1V_2$ viene dado en sentido anti-horario, entonces el triángulo tiene área signada positiva*, es decir: $|V_0V_1V_2|>0$. En este caso la coordenada baricéntrica α tiene valor 0 cuando se cumple que $|PV_1V_2|=0$. Esto ocurre exclusivamente cuando el punto P se encuentra sobre la recta soporte† de la arista V_1V_2. Además α es negativa cuando $|PV_1V_2|<0$, es decir, cuando P se encuentra en el lado opuesto a V_0 respecto a la recta soporte de V_1V_2; y positiva cuando $|PV_1V_2|>0$, es decir, cuando P se encuentra en el mismo lado que V_0 respecto a la recta soporte de V_1V_2.
- Si el triángulo viene dado en orden horario, entonces el triángulo tiene área signada negativa‡, es decir: $|V_0V_1V_2|<0$. En esta situación α es negativa cuando $|PV_1V_2|>0$, lo que indica que P se encuentra en el lado opuesto a V_0 respecto a la recta soporte de V_1V_2. Del mismo modo α es positiva cuando $|PV_1V_2|<0$, lo cual indica que P se encuentra en el mismo lado que V_0 respecto a la recta soporte de V_1V_2. Cuando $|PV_1V_2|=0$ entonces la coordenada baricéntrica α es cero, por lo que P está sobre la recta soporte de V_1V_2.

Podemos llegar a conclusiones similares respecto a las coordenadas baricéntricas β y γ del punto respecto al triángulo. En estos casos el signo de cada coordenada baricéntrica β y γ en relación a los segmentos V_2V_0 y V_0V_1 respectivamente representa el lado en el que se encuentra P respecto a la recta soporte de dichos segmentos.

La utilización del signo de las coordenadas baricéntricas nos proporcionará un mecanismo para el desarrollo de algoritmos para la detección de colisión más eficientes y robustos, que utilizan aritmética entera basada en sumas signadas en lugar de utilizar aritmética real.

---

* Recordemos que un triángulo con área signada positiva lo hemos llamado *triángulo positivo*.

† Dada una arista V_1V_2, llamamos *recta soporte* de la arista a la recta que pasa por los vértices V_1 y V_2.

‡ Es lo que hemos llamado un *triángulo negativo*.

Una consecuencia evidente de la Propiedad 4.1 es la fórmula para determinar si un punto se encuentra fuera de un triángulo.

Propiedad 4.2: Sea un punto P con coordenadas baricéntricas (α,β,γ) en relación al triángulo formado por los puntos V_0,V_1,V_2. El punto P está fuera del triángulo definido por los puntos V_0,V_1,V_2 si y solo si $\alpha < 0 \vee \beta < 0 \vee \gamma < 0$.

A modo de conclusión, podemos caracterizar la posición de un punto respecto a un triángulo $T=V_0V_1V_2$ utilizando las coordenadas baricéntricas de ese punto (Figura 4.2). En nuestro caso nos interesa conocer además de si un punto está dentro o fuera de un triángulo, si está sobre un vértice o arista concreto, o en el interior del triángulo. Todas las expresiones que mostramos a continuación pueden ser substituidas por expresiones equivalentes que consideren únicamente el signo de la correspondiente coordenada baricéntrica. Veamos las posibles situaciones de un punto respecto a un triángulo*:

- El punto está sobre el vértice V_0 sii $\alpha=1$
- El punto está sobre el vértice V_1 sii $\beta=1$
- El punto está sobre el vértice V_2 sii $\gamma=1$
- El punto está en el interior de la arista V_1V_2 sii $\alpha = 0$ y $\beta,\gamma > 0$
- El punto está en el interior de la arista V_2V_0 sii $\beta = 0$ y $\alpha,\gamma > 0$
- El punto está en el interior de la arista V_0V_1 sii $\gamma = 0$ y $\alpha,\beta > 0$
- El punto está dentro de $V_0V_1V_2$ sii $\alpha,\beta,\gamma \geq 0$
- El punto está fuera de $V_0V_1V_2$ sii $\alpha,\beta,\gamma < 0$
- El punto está en el interior de $V_0V_1V_2$ sii $\alpha,\beta,\gamma > 0$

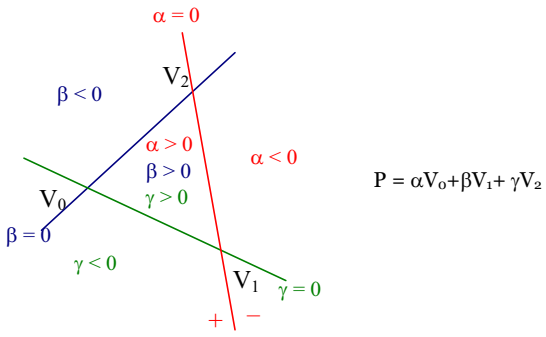


Figura 4.2: Relaciones entre un punto y un triángulo. Distintas zonas en las que se divide el espacio en función de las coordenadas baricéntricas.

* Recordemos que los vértices del triángulo están ordenados, y la definición de las coordenadas baricéntricas de un punto respecto a un triángulo se ha realizado en base al orden de dichos vértices.

Podríamos diseñar un algoritmo que obtuviese la posición de un punto respecto a un triángulo utilizando la caracterización anterior. Sin embargo, puede no ser necesaria toda la información acerca de la posición de un punto, por ejemplo, para una aplicación en la que se necesitara conocer si un punto está en el interior, en la frontera, o en el exterior del triángulo, y que no necesita información adicional acerca de si está en un vértice o arista concreto. Este algoritmo puede optimizarse de manera que sólo distinga entre interior, exterior y frontera del triángulo.

Dependiendo de la aplicación concreta de que se trate utilizaremos distintos algoritmos, de manera que se obtenga de una manera óptima la información que se necesita basándonos para ello en la descripción que hemos realizado acerca de la posición de un punto respecto a un triángulo mediante sus coordenadas baricéntricas.

Notación: Sea T_i el triángulo $V_0V_1V_2$, llamaremos $\alpha_i(P),\beta_i(P),\gamma_i(P)$ a las coordenadas baricéntricas del punto P respecto a T_i, siendo $\alpha_i(P) = |PV_1V_2| \,/\, |V_0V_1V_2|$, $\beta_i(P) = |PV_2V_0| \,/\, |V_0V_1V_2|$ y $\gamma_i(P) = |PV_0V_1| \,/\, |V_0V_1V_2|$.

4.2. Algoritmo de Detección de Colisión Punto/Polígono

En esta sección desarrollaremos un algoritmo para la detección de colisión entre un punto y un polígono [JSF02c]. En primer lugar veremos un algoritmo para la determinación de la inclusión de puntos en polígonos utilizando recubrimientos simpliciales basado en el algoritmo de [FTU95]. A continuación mostraremos cómo utilizar la coherencia temporal y geométrica para aprovechar cálculos entre frames, de manera que podamos utilizar este algoritmo de inclusión en un algoritmo genérico de detección de colisión punto/polígono para polígonos no convexos. Finalmente particularizaremos este algoritmo para dos casos, en primer lugar para polígonos convexos, y posteriormente para polígonos con recubrimiento positivo respecto al centroide del objeto. En estos dos casos los algoritmos son más eficientes que el algoritmo genérico, pues aprovechan en mayor medida la coherencia debido a la geometría especial de los polígonos [JSF02c].

A lo largo de este capítulo daremos los vértices de los triángulos en el orden adecuado para que α siempre se corresponda con la coordenada baricéntrica del punto respecto a la arista exterior del triángulo del recubrimiento, es decir, respecto a una arista del polígono*. De esta forma, dicha coordenada baricéntrica nos

* Para que esto ocurra, basta con comenzar la definición del triángulo por el vértice origen del recubrimiento V_0 (el centroide), de manera que el triángulo se dará en la forma $V_0V_1V_2$.

proporcionará un mecanismo para decidir si un punto se encuentra a un lado u otro de la recta soporte de la arista no original del triángulo del recubrimiento con respecto al centroide, que en nuestro caso es el punto de referencia utilizado para realizar el recubrimiento del polígono.

4.2.1. Inclusión Punto/Polígono

Un problema fundamental en Informática Gráfica y Geometría Computacional consiste en la determinación de la inclusión de puntos en polígonos. Existen multitud de soluciones a este problema [Hai94] [Huang97], siendo el algoritmo de *crossings-count* uno de los más rápidos (sin tener en cuenta la posibilidad de realizar una descomposición espacial). Veamos a continuación un algoritmo de inclusión para polígonos representados mediante recubrimientos simpliciales basado en el algoritmo de Feito et. al [FTU95].

Hemos modificado el algoritmo publicado en [FTU95] de manera que utilizamos las coordenadas baricéntricas del punto considerado respecto a los 2-símplices o triángulos del recubrimiento del polígono. Así, para comprobar la inclusión en el polígono, se comprueba la inclusión en los triángulos del recubrimiento, asignando el valor 0, +1 ó +2 dependiendo de la situación del punto respecto del triángulo correspondiente como veremos a continuación, y multiplicando por el signo del triángulo. Finalmente obtenemos la inclusión del punto en el polígono si la suma de estos valores es igual a +2.

Estos valores de inclusión (0, +1 ó +2) se corresponden a las distintas situaciones del punto respecto de cada triángulo: Cuando el punto no está incluido en el triángulo se asigna un valor de 0. El valor +1 indica que el punto pertenece a una arista original del triángulo, incluyendo los vértices, es decir, una de las aristas utilizadas para unir un vértice del polígono con el punto de referencia utilizado para el recubrimiento (este caso representa a una arista compartida por dos triángulos del recubrimiento). Y el valor de +2 indica que el punto pertenece al triángulo y no está sobre ninguna arista original[*].

El algoritmo se ha optimizado de manera que detecta cuando un punto se encuentra sobre alguna de las aristas del polígono, es decir, cuando se encuentra sobre alguna de las aristas o vértices no originales de los triángulos del recubrimiento. En esa situación el punto se encuentra dentro del polígono y no es necesario contabilizar la inclusión del mismo en el resto de los triángulos del recubrimiento.

[*] Nótese que en este caso se ha incluido la pertenencia del punto a la arista no original del triángulo, pero no se incluyen los vértices del mismo.

También hemos considerado la situación especial en la que el punto se encuentre sobre el centroide del polígono. En este caso, como estamos utilizando dicho punto como origen del recubrimiento, no es posible determinar si el punto está o no dentro del polígono. Hemos optado por clasificar dicho punto en tiempo de pre-procesamiento, utilizando cualquier otro método de clasificación de puntos en polígonos*. Así, en el caso de que el punto coincida con el centroide, devolvemos el valor de inclusión almacenado previamente.

El algoritmo modificado que incluye las consideraciones anteriores es el siguiente (Algoritmo 4.1):

```
bool poligono::inclusion2D(punto P) {
   inclusion = 0
   for (i=0;i<this.numeroTriangulos();i++) {
      pos = this.triangulo(i).posiciónInclusion2D(P)
      switch (pos) {
         V0V1V2:        inclusion += 2*sign(this.triangulo(i))
         V1, V2, V1V2:  return true
         V0V1, V0V2:    inclusion += sign(this.triangulo(i))
         V0:            return V0.inclusion2D()
      }
   }
   return (inclusion==2)
}

posición triangulo::posiciónInclusion2D(punto P) {
   if (this.degenerado()==true)
      if (this.V1V2.inclusion(P)==true) return V1V2
      else return OUT
   else {
      sα = sign(this.α(P))
      if (sα < 0) return OUT
      sβ = sign(this.β(P))
      if (sβ < 0) return OUT
      sγ = sign(this.γ(P))
      if (sγ < 0) return OUT
      if (sα == sβ == sγ == 1) return V0V1V2
      if (sα == 0 && sβ == sγ == 1) return V1V2
      if (sβ == 1 && sα == sγ == 0) return V1
      if (sγ == 1 && sα == sβ == 0) return V2
      if (sβ == 0 && sα == sγ == 1) return V0V2
      if (sγ == 0 && sα == sβ == 1) return V0V1
      if (sα == 1 && sβ == sγ == 0) return V0
   }
}
```

Algoritmo 4.1: Inclusión Punto/Polígono.

* Puede ser este mismo método, cambiando el origen del recubrimiento para este caso a otro punto distinto del centroide.

En el Algoritmo 4.1. hemos incluido un algoritmo para la determinación de la inclusión de un punto en un triángulo del recubrimiento. Este algoritmo es especial para este caso concreto pues considera sólo las posiciones que interesan para determinar la inclusión del punto en el polígono.

Como las coordenadas baricéntricas de un punto sólo pueden calcularse cuando el triángulo no es degenerado, y como consecuencia del recubrimiento del polígono pueden darse estos casos especiales (Figura 4.3), se ha adoptado una solución que consiste en marcar estos triángulos degenerados al realizar el recubrimiento, de manera que para comprobar la inclusión del punto en dicho triángulo degenerado, se realiza la inclusión del punto en la correspondiente arista del polígono, utilizando para ello un algoritmo paramétrico.

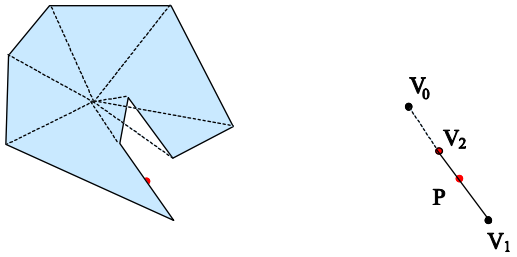


Figura 4.3: Triángulo del recubrimiento degenerado ($V_0V_1V_2$). En el algoritmo se comprueba la inclusión del punto en el segmento V_1V_2 utilizando un algoritmo clásico.

Evidentemente el algoritmo de inclusión mostrado puede optimizarse, de manera que se realice un menor número de comparaciones y en un orden determinado para que sea más eficiente. En esta ocasión podría reducirse el número de operaciones agrupando los casos que representan la frontera del polígono.

4.2.2. Coherencia temporal y geométrica

En esta sección estudiaremos la utilización de la coherencia en los algoritmos de detección de colisión entre un punto y un polígono. En el caso que nos ocupa, podemos hacer uso de la coherencia para aprovechar ciertos cálculos entre frames sucesivos. Una solución simple al problema de detección de colisión consistiría en repetir el test de inclusión visto en la sección anterior, pero evidentemente no aprovecharíamos ciertas propiedades presentes en el movimiento o en la geometría de los objetos.

Haciendo uso de la coherencia y sabiendo que no es necesario calcular todas las coordenadas baricéntricas de un punto respecto a un triángulo para determinar si se

encuentra fuera del mismo (Propiedad 4.2), vamos a desarrollar un algoritmo de detección de colisión punto/polígono. Según esta propiedad, si cualquiera de las 3 coordenadas baricéntricas del punto respecto a un triángulo es negativa, el punto está fuera. Esto nos lleva a calcular β sólo cuando α es positiva o cero, y γ sólo cuando α y β son también positivas o cero.

A continuación vamos a enunciar un lema que nos permite obtener la inclusión de un punto en un polígono en ciertos casos especiales, en base al estado de inclusión anterior y al valor de la coordenada baricéntrica α del punto respecto de los triángulos del recubrimiento, sin la necesidad de obtener el resto de coordenadas baricéntricas. Pero antes definamos la notación utilizada.

Notación: Dados dos objetos A y B utilizaremos la notación A *in* B para indicar que se produce la inclusión del objeto A en el objeto B. De igual modo, *not (*A *in* B*)* indicará que no se produce la inclusión del objeto A en el objeto B. Debemos mencionar que utilizamos el término inclusión tanto para referirnos a la inclusión en el interior del objeto, como a la intersección con la frontera del objeto.

Notación: Utilizaremos los corchetes para indicar que una situación o propiedad se produce en un momento puntual en el tiempo (en un frame), por ejemplo [sign($\alpha(P)$)=0] indica que el signo de α es cero en un momento determinado. Cuando el frame es importante o se quiere relacionar con otros frames, utilizaremos un subíndice tras los corchetes para indicar a qué frame se corresponde, suponiendo una sucesión ordenada de frames, por ejemplo $[P \text{ in } F]_i$.*

Lema 4.1: Sea P un punto cualquiera y F un polígono con recubrimiento $C_F=\{T_i\}$ / $i=1..n$. Si se cumple que $[not\ (P\ in\ F)]_j$ y que $[sign(\alpha_i(P))]_j=[sign(\alpha_i(P))]_{j+1}$ para todo $i=1..n$, entonces $[not\ (P\ in\ F)]_{j+1}$. De igual modo, si se cumple que $[P\ in\ F]_j$ y que $[sign(\alpha_i(P))]_j= [sign(\alpha_i(P))]_{j+1}$ para todo $i=1..n$, entonces $[P\ in\ F]_{j+1}$.

Demostración: Para que un punto P que se encuentra fuera de un polígono pase de un instante de tiempo al siguiente a estar dentro del polígono, debe atravesar una de las aristas que forman el polígono. Esto implica un cambio de signo (de negativo a positivo o cero) de la coordenada baricéntrica α respecto al triángulo del recubrimiento del que forma parte esa arista. Debido a esto, si no se produce ningún cambio de signo en la coordenada α respecto a ningún triángulo del recubrimiento, el punto no habrá entrado en el mismo y permanecerá fuera.

* Debido a que utilizamos una detección de colisión en momentos discretos de tiempo o frames, nos encontramos con un problema relacionado con el *movimiento continuo de los objetos* como vimos en la sección 2.1.3.1. En este caso no se detectarán situaciones de colisión intermedias entre frames.

Del mismo modo, para que un punto que está dentro de un polígono pase a estar fuera, debe producirse un cambio de signo en α respecto a alguno de los triángulos del recubrimiento (un cambio de positivo a cero o negativo). Si no se produce ningún cambio de signo el punto permanecerá dentro.

Podemos considerar el caso especial en el que el punto se mueva por la recta soporte de una arista. En este caso, si el punto se encuentra fuera (o dentro en el segundo caso), para pasar a dentro (o fuera en el segundo caso) del polígono debe atravesar otra de las aristas del polígono, la que incide en uno de los vértices de la arista en cuestión, por lo que el planteamiento anterior sigue siendo válido para esta situación. □

Según el lema anterior, es necesario que se produzca un cambio en el signo de α para alguno de los triángulos del recubrimiento cuando el punto cambia su estado de inclusión respecto al polígono. Por tanto, podemos guardar el signo de α respecto a todos los triángulos del recubrimiento del frame anterior y compararlo con su signo en el frame actual. Si no se produce ningún cambio de signo, el estado de inclusión actual del punto será el mismo que el estado anterior. De este modo, dado un triángulo T_i del recubrimiento, no será necesario calcular las coordenadas baricéntricas β_i y γ_i cuando no cambie el signo de α_i.

A partir del Lema 4.1 y de la Propiedad 4.1 podemos deducir las siguientes propiedades que nos permiten aprovechar un poco más la coherencia:

Propiedad 4.3: Sea F un polígono con recubrimiento $C_F=\{T_i\}$ / $i=1..n$. Para todo punto P tal que $[not\ (P\ in\ F)]_j$ y $[P\ in\ F]_{j+1}$, existe algún $k\in[1,n]$ tal que $[\alpha_k(P) < 0]_j$ y $[\alpha_k(P) \geq 0]_{j+1}$.

Propiedad 4.4: Sea F un polígono con recubrimiento $C_F=\{T_i\}$ / $i=1..n$.. Para todo punto P tal que $[P\ in\ F]_j$ y $[not\ (P\ in\ F)]_{j+1}$, existe algún $k\in[1,n]$ tal que $[\alpha_k(P) \geq 0]_j$ y $[\alpha_k(P) < 0]_{j+1}$.

Según estas propiedades, cuando un punto pasa de estar fuera a estar dentro del polígono, se produce un cambio de negativo a cero o positivo en el signo de α respecto a algún triángulo del recubrimiento. Podemos aprovechar esta propiedad en el algoritmo de detección de colisión desarrollado en esta sección. Para esto guardamos en una máscara de bits si se cumple la relación $\alpha_i(P) \geq 0$ para todos los triángulos $T_i\in C_F$, $i=1..n$. Cada bit b_i valdrá 1 si $\alpha_i(P) \geq 0$ y 0 en caso contrario. Esta codificación divide el espacio que ocupa el polígono en distintas zonas, cada una de las cuales tiene asociada una máscara de bits y un estado de inclusión (Figura 4.4).

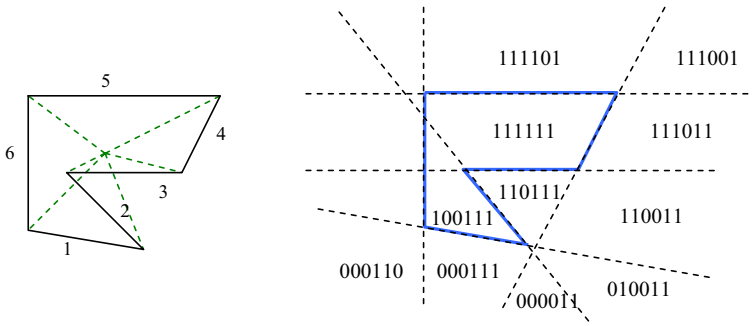


Figura 4.4: Interpretación geométrica del cambio de signo de α: división del espacio en zonas según $sign(\alpha_i)$. Los bits representados son $b_i = (\alpha_i(P) \geq 0)$.

4.2.3. Detección de Colisión Punto/Polígono no convexo

Haciendo uso de las propiedades anteriores hemos desarrollado un algoritmo genérico para la detección de colisión entre un punto y un polígono no convexo (Algoritmo 4.2 y Figura 4.5). Este algoritmo también es válido para polígonos convexos o con un recubrimiento positivo en base al centroide, aunque para dichos casos se han elaborado otros algoritmos más eficientes como veremos en las Secciones 4.2.4 y 4.2.5.

Hemos establecido un orden en los triángulos del recubrimiento, de manera que éste se corresponde con el orden de las n aristas del polígono dadas en secuencia anti-horaria. Así podemos utilizar una máscara de bits para cada punto P sobre el que deseamos obtener la colisión con el polígono. En dicha máscara cada bit b_i, $i=1..n$, se corresponde con un triángulo del recubrimiento y representa si $\alpha_i(P) \geq 0$.

Este algoritmo calcula dicha máscara de bits en cada frame, comparando esta máscara con la máscara del frame previo. Si las máscaras son iguales, el punto se encuentra en la misma zona espacial que en el frame anterior, por lo que su estado de inclusión es el mismo, y no es necesario realizar cálculos adicionales.

Cuando las máscaras son distintas, es necesario calcular el signo del resto de coordenadas baricéntricas para obtener la inclusión del punto en cada triángulo y determinar así la inclusión en el polígono. Los valores de $\beta_k(P)$ sólo se calculan si el correspondiente bit b_k de la máscara que representa $\alpha_i(P)$ es igual a 1*. Del mismo modo, los valores de $\gamma_k(P)$ sólo se calculan si $\beta_k(P) \geq 0$.

---

* Cuando el valor de este bit es 0 indica que el signo de la coordenada baricéntrica α es negativo, y debido a la Propiedad 4.3 el punto se encuentra fuera del triángulo.

```
static previob_α = 0

bool poligono::testDeteccionColision2D(punto P) {
   //Inclusion en la circunferencia envolvente
   if (this.circunferenciaEnvolvente().inclusion(P)==false) return false
   s_α = s_β = s_γ = b_α = b_β = b_γ = 0
   //Cálculo de la máscara de bits alfa
   for (t=0;t<this.numeroTriangulos();t++) {
      s_α[t] = sign(this.triangulo(t).α(P))
      b_α[t] = (s_α[t]>=0)
   }
   //Si la máscara de bits es la misma que en la iteración anterior
   if (b_α == previob_α) return EQUAL_STATE
   previob_α = b_α
   //Cálculo de la máscara de bits beta
   for (t=0;t<this.numeroTriangulos();t++)
      if (b_α[t]==1) {
         s_β[t] = sign(this.triangulo(t).β(P))
         b_β[t] = (s_β[t]>=0)
      }
   //Cálculo de la mascara de bits gamma
   for (t=0;t<this.numeroTriangulos();t++)
      if (b_β[t]==1) {
         s_γ[t] = sign(this.triangulo(t).γ(P))
         b_γ[t] = (s_γ[t]>=0)
      }
   //Inclusión solo donde la mascara de bits gamma = 1
   inclusion = 0
   for (t=0;t<this.numeroTriangulos();t++) {
      if (b_γ[t]==1) {
         if (this.triangulo(t).degenerado()==true)
            posicion = this.triangulo(t).posicionInclusion2D(P) // Contempla caso degenerado
         else
            posicion = posicionDeteccionColision2D(s_α[t], s_β[t], s_γ[t])
         switch (posicion) {
            V_0V_1V_2:             inclusion += 2*sign(this.triangulo(t))
            V_1, V_2, V_1V_2:      return true
            V_0V_1, V_0V_2:        inclusion += sign(this.triangulo(t))
            V_0:                   return V_0.inclusion()
         }
      }
   }
   return (inclusion == 2)
}

posicion posicionDeteccionColision2D(int s_α, int s_β, int s_γ) {
   if (s_α == s_β == s_γ == 1) return V_0V_1V_2
   if (s_α == 0 && s_β == s_γ == 1) return V_1V_2
   if (s_β == 1 && s_α == s_γ == 0) return V_1
   if (s_γ == 1 && s_α == s_β == 0) return V_2
   if (s_β == 0 && s_α == s_γ == 1) return V_0V_2
   if (s_γ == 0 && s_α == s_β == 1) return V_0V_1
   if (s_α == 1 && s_β == s_γ == 0) return V_0
}
```

Algoritmo 4.2: Detección de colisión punto/polígono no convexo.

Posición	$[P]_0$	$[P]_1$	$[P]_2$	$[P]_3$
i	123456	123456	123456	123456
Signo T_i	+-++++	+-++++	+-++++	+-++++
$b_\alpha[i]$	110111	110011	110011	111111
$b_\beta[i]$	11X000	10XX01	XXXXXX	100100
$b_\gamma[i]$	11XXXX	1XXXX0		0XX1XX
estado	fuera	fuera	igual	dentro

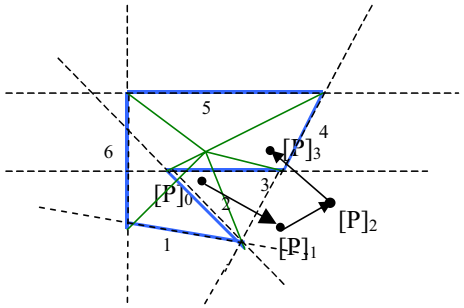


Figura 4.5: Modo de operación del algoritmo con máscaras de bits. El símbolo 'X' indica que no ha sido necesario calcular el valor correspondiente.

Debido a la coherencia temporal, los puntos sobre los que obtenemos la colisión cambian poco de posición en el espacio, estando la mayor parte del tiempo en la misma zona, por lo que la máscara de bits que representa el signo de $\alpha_i(P)$ cambiará un menor número de veces cuanto mayor sea la coherencia temporal, con el consiguiente aumento de eficiencia. Este aspecto está íntimamente relacionado con la coherencia geométrica*, pues mientras ésta se de en mayor grado también se presentará mayor coherencia temporal.

Cuando un triángulo del recubrimiento es degenerado, se comprueba la inclusión del punto en la correspondiente arista del polígono. Estos casos han sido detectados y marcados al realizar el recubrimiento del polígono en tiempo de pre-procesamiento.

4.2.4. Detección de Colisión Punto/Polígono convexo

El algoritmo anterior, válido para cualquier tipo de polígono, puede optimizarse para el caso de polígonos convexos. En ese caso, el recubrimiento con origen en el centroide del polígono se convierte en una triangulación del mismo, pues todos los triángulos tienen área signada positiva. Este nuevo algoritmo de DC punto/polígono convexo se basa en encontrar una arista del polígono cuya recta soporte deje al punto y al polígono cada uno a un lado respecto de dicha recta [Ber04]. Debido a la convexidad del polígono, cualquier recta que pase por una de sus aristas deja al polígono completamente a un lado de la recta (Figura 4.6). En el siguiente lema relacionamos dicho concepto con el uso de las coordenadas baricéntricas del punto.

* Evidentemente el tamaño de las zonas que se generan dependen de la geometría del objeto. Además cuanto mayor sea el número de aristas del polígono, mayor será el número de zonas espaciales obtenidas, con la consiguiente disminución de coherencia geométrica.

Lema 4.2: Sea P un punto y F_c un polígono convexo con recubrimiento $C_{Fc}=\{T_i\}$ / $i=1..n$, entonces [*not (P in* F_c*)*] sii existe al menos un triángulo $T_j \in C_{Fc}$ tal que $sign(\alpha_j(P))<0$.

Demostración: Supongamos que F_c es un polígono convexo, que O es su centroide, y que existe un triángulo $T_j=V_oV_1V_2$, $T_j \in C_{Fc}$ tal que P cumple que $sign(\alpha_j(P))<0$, entonces el punto P estará en el lado opuesto a O respecto a la recta soporte de V_1V_2. Al ser F_c un polígono convexo, todo $T_i \in C_{Fc}$, $i=1..n$, se encontrará en el mismo lado que O respecto a la recta soporte de V_1V_2, y en el lado opuesto al punto P. Por consiguiente, no existe ningún $T_i \in C_{Fc}$, $i=1..n$, tal que [P *in* T_i], por lo que se cumple que [*not (P in* F_c*)*].

Del mismo modo, si [*not (P in* F_c*)*] debe existir al menos un triángulo $T_j \in C_{Fc}$ en el que el punto P cumpla que $sign(\alpha_j(P))<0$, pues si esto no fuese así se cumpliría que $sign(\alpha_i(P))\geq 0$ para todo $i=1..n$, lo que nos indicaría que [P *in* F_c], puesto que [O *in* F_c] al ser F_c convexo y P se encontraría en el mismo lado que O respecto a todas las rectas soporte de las aristas que forman el polígono; lo cual es falso, pues partimos de la premisa de que [*not (P in* F_c*)*]. □

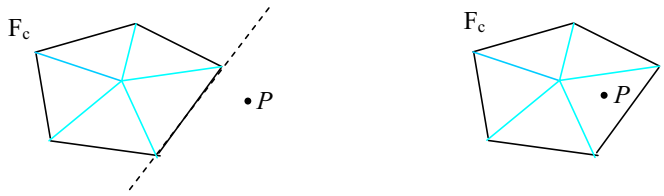


Figura 4.6: Un punto está fuera de un polígono convexo si existe algún T_i tal que $sign(\alpha_i(P))<0$. Un punto está dentro si para todo triángulo T_i se cumple $sign(\alpha_i(P))\geq 0$ o si existe algún triángulo T_j tal que [P *in* T_j].

Mediante este lema hemos establecido la relación de inclusión entre un punto y un polígono convexo a través del signo de la coordenada baricéntrica $\alpha_j(P)$. Si encontramos algún triángulo del recubrimiento T_j en el que se cumpla que $sign(\alpha_j(P))<0$, sabremos que el punto está fuera del Polígono.

Como consecuencia del lema anterior podemos enunciar el siguiente corolario:

Corolario 4.1: Dado un punto P y un polígono convexo F_c con recubrimiento $C_{Fc}=\{T_i\}$ / $i=1..n$, entonces [P in F_c] sii para todo triángulo $T_i \in C_{Fc}$, $i=1..n$, se cumple que $sign(\alpha_i(P))\geq 0$.

Demostración: Es la negación del Lema 4.2. □

Es evidente que si un punto se encuentra en algún triángulo del recubrimiento del polígono, al ser todos los triángulos positivos, podemos determinar que el punto se encuentra en el polígono, sin necesidad de comprobar la inclusión en todos y cada uno de los triángulos del recubrimiento (Figura 4.6).

Mediante la siguiente heurística combinamos la coherencia del movimiento del punto con el Lema 4.2 para utilizarlo en el algoritmo de DC punto/polígono convexo.

Heurística 4.1: Dado un punto *P* y un polígono convexo F_c con recubrimiento $C_{Fc}=\{T_i\}$ / *i=1..n* tal que $[not\ (P\ in\ F_c)]_j$ y un triángulo $T_k \in C_{Fc}$ tal que $[sign(\alpha_k(P))<0]_j$, debido a la coherencia temporal y geométrica, lo más probable es que $[sign(\alpha_k(P))<0]_{j+1}$.

Si lo anterior no se cumple pueden ocurrir dos cosas:

1. Si $[P\ in\ T_k]_{j+1}$ entonces $[P\ in\ F_c]_{j+1}$
2. Si $[not\ (P\ in\ T_k)]_{j+1}$ entonces
   - 2.1. Si $[sign(\beta_k(P))<0]_{j+1}$ entonces comenzar la búsqueda de un triángulo que cumpla el Lema 4.2 por el triángulo $T_{k\oplus 1}$ y en ese sentido.
   - 2.2. Si $[sign(\gamma_k(P))<0]_{j+1}$ entonces comenzar la búsqueda de un triángulo que cumpla el Lema 4.2 por el triángulo $T_{k\oplus -1}$ y en ese sentido.

El primer paso del algoritmo de DC consiste en encontrar un triángulo del recubrimiento que cumpla la condición establecida en el Lema 4.2. A continuación, una vez encontrado dicho triángulo, en cada movimiento del punto se comprueba en primer lugar la posición del punto respecto a ese triángulo. Debido a la coherencia, la mayor parte del tiempo el punto estará en la misma posición respecto al mismo triángulo, con lo que no será necesario realizar cálculos adicionales para determinar su inclusión.

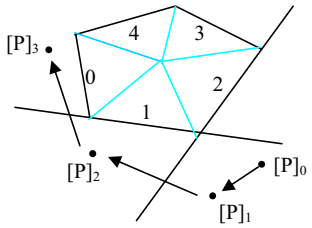
Cuando no es posible determinar la inclusión del punto, dado un triángulo del recubrimiento, esta heurística define cómo actuar para encontrar un triángulo que cumpla el Lema 4.2, si lo hay, basándonos en el signo de las coordenadas $\beta_k(P)$ ó $\gamma_k(P)$. Así, si $sign(\beta_k(P))<0$ el siguiente triángulo que debemos utilizar para comprobar que cumpla dicho lema será el $T_{k\oplus 1}$ y si no se cumple el $T_{k\oplus -1}$. Una vez establecida la dirección de la búsqueda, ésta se realiza en el mismo sentido, sin la necesidad de realizar comprobaciones adicionales. Si no existe ningún triángulo que cumpla el Lema 4.2, determinamos que no se produce la inclusión del punto en el polígono. Haciendo uso de esta heurística podemos obtener un algoritmo de DC de la siguiente forma (Algoritmo 4.3 y Figura 4.7):

```
bool poligonoConvexo::testDeteccionColision2D(punto P, int & numtriangulo) {
   //Inclusion en la circunferencia envolvente
   if (this.circunferenciaEnvolvente().inclusion(P)==false) return false

   s = sign(this.triangulo(numtriangulo).α(P))
   if (s>=0)
      return this.inclusion2D(P,numtriangulo)
   return false
}

bool poligonoConvexo::inclusion2D(punto P, int & numtriangulo) {
   indice = numtriangulo ⊕ 1
   s = sign(this.triangulo(indice).α(P))
   if (s<0) {
      numtriangulo = indice
      return false
   }
   indice = numtriangulo ⊕ -1
   s = sign(this.triangulo(indice).α(P))
   if (s<0) {
      numtriangulo = indice
      return false
   }
   for (t=0;t<this.numeroTriangulos();t++) {
      indice = numtriangulo ⊕ (t+2)
      s = sign(this.triangulo(indice).α(P))
      if (s<0) {
         numtriangulo = indice
         return false
      }
   }
   return true
}
```

Algoritmo 4.3: Detección de colisión punto/polígono convexo.



$[sign(\alpha_2(P))]_0 = -1$, el punto está fuera

$[sign(\alpha_2(P))]_0 = [sign(\alpha_2(P))]_1 = -1$ y el punto sigue fuera

$[sign(\alpha_2(P))]_1 \neq [sign(\alpha_2(P))]_2$, comprobamos si $[sign(\beta_2(P))]_2 < 0$, como es así la siguiente arista es $2 \oplus -1 = 1$

$[sign(\alpha_1(P))]_2 = -1$, el punto está fuera

$[sign(\alpha_1(P))]_2 \neq [sign(\alpha_1(P))]_3$, comprobamos si $[sign(\beta_1(P))]_3 < 0$, como es así la siguiente arista es $1 \oplus -1 = 0$

$[sign(\alpha_0(P))]_3 = -1$, el punto está fuera

Figura 4.7: Funcionamiento del algoritmo para polígonos convexos.

Veamos a grandes rasgos como funciona el algoritmo:

- En primer lugar se calcula si $P \in F_c$. Para realizar esto, se comprueba si $[sign(\alpha_i(P)) \geq 0]_0$ para todo $i=1..n$. Si no se cumple para algún valor de i, dicho triángulo del recubrimiento, digamos el triángulo T_k, es utilizado como referencia para el siguiente frame.

- En cada movimiento del punto P, se comprueba si $[sign(\alpha_k(P)) < 0]_j$, si se cumple esta condición, el punto está fuera, si no, se comprueba si está dentro del polígono F_c (en concreto del triángulo T_k) y en caso de que no esté, se comprueban las aristas contiguas a la izquierda o a la derecha (dependiendo del signo de la coordenada baricéntrica β) hasta que se encuentre alguna arista que permita concluir que el punto está en el polígono o que no está en él.

4.2.5. Detección de Colisión Punto/Polígono con recubrimiento positivo

Hemos desarrollado un algoritmo específico para polígonos con un recubrimiento de triángulos positivos respecto al centroide*, de manera que aprovecha en mayor medida las características particulares de la geometría del polígono. El algoritmo está basado en el siguiente lema.

Lema 4.3: Dado un punto P y un polígono F_+ con recubrimiento positivo respecto al centroide $C_{F+}=\{T_i\}$ / $i=1..n$, entonces [*not (P in* F_+*)*] sii existe al menos un triángulo del recubrimiento T_j tal que $sign(\alpha_j(P)) < 0 \wedge sign(\beta_j(P)) \geq 0 \wedge sign(\gamma_j(P)) \geq 0$.

Demostración: Para que un punto esté dentro del polígono debe estar en un único triángulo del recubrimiento o en la frontera entre dos triángulos del recubrimiento, pues todos los triángulos son positivos. Debido a que todos los triángulos del recubrimiento son positivos, no existe ningún triángulo que tenga parte del mismo dentro de otro triángulo del recubrimiento, salvo la frontera entre dos triángulos. Por tanto, si un punto se encuentra en la región del espacio que cumple que $\beta \geq 0 \wedge \gamma \geq 0$ respecto de un triángulo, que representa una región angular determinada, pero no se encuentra en dicho triángulo, es decir $\alpha < 0$, podemos asegurar que el punto no está en ningún otro triángulo que interseque con dicha región angular, pues no lo hay al ser todos positivos, y por tanto el punto no estará dentro del polígono. □

* Por comodidad utilizaremos en lugar de *recubrimiento mediante triángulos positivos* el término *recubrimiento positivo*. Debemos notar que los polígonos convexos se incluyen en esta categoría, por lo que el algoritmo aquí expuesto es válido para este tipo de polígonos.

Mediante el Lema 4.3 podemos determinar que el punto no está incluido en el polígono encontrando un triángulo del recubrimiento que cumpla la condición establecida en dicho lema, no siendo necesario realizar cálculos adicionales que involucren al resto de triángulos del recubrimiento (Figura 4.8.a).

A partir del Lema 4.3 podemos obtener la condición necesaria y suficiente para que se produzca la inclusión de un punto en un polígono con recubrimiento positivo.

Corolario 4.2: Dado un punto *P* y un polígono F_+ con recubrimiento positivo respecto al centroide $C_{F+}=\{T_i\}$ / *i=1..n*, entonces [*P in* F_+] sii existe al menos un triángulo del recubrimiento $T_j \in C_{F+}$ tal que [*P in* T_j].

Demostración: Es la negación del Lema 4.3. □

Al igual que para el caso de figuras convexas, este corolario nos permite determinar la inclusión de un punto en un polígono con recubrimiento positivo respecto al centroide sin la necesidad de comprobar la inclusión en todos los triángulos del recubrimiento, pues con sólo encontrar un triángulo en el que esté incluido, se determina que está incluido también en el polígono (Figura 4.8.b).

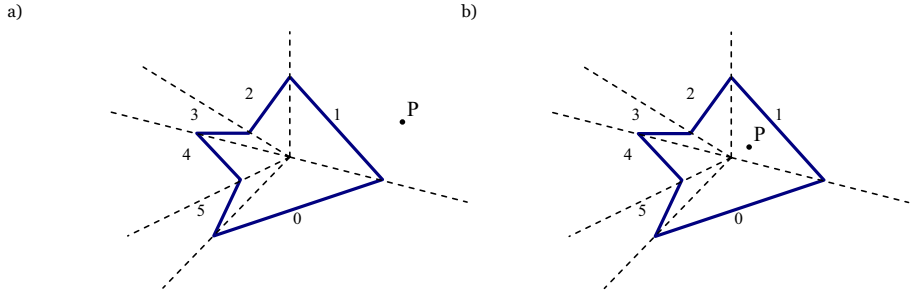


Figura 4.8: Inclusión de un punto en un polígono con recubrimiento positivo respecto del centroide: a) $sign(\alpha_1(P))<0 \wedge sign(\beta_1(P))\geq 0 \wedge sign(\gamma_1(P))\geq 0$, por lo tanto el punto está fuera del polígono. b) [*P in* T_1], por lo tanto el punto está dentro del polígono.

Mediante la siguiente heurística combinamos la coherencia del movimiento del punto con el Lema 4.3 para obtener un algoritmo de DC punto/polígono con recubrimiento positivo.

Heurística 4.2: Dado un punto *P* y un polígono F_+ con recubrimiento positivo respecto al centroide $C_{F+}=\{T_i\}$ / *i=1..n*, tal que [*not (P in* F_+*)*]$_j$ y un triángulo $T_k \in C_{F+}$ tal que $[sign(\alpha_k(P))<0 \wedge sign(\beta_k(P))\geq 0 \wedge sign(\gamma_k(P))\geq 0]_j$, debido a la coherencia temporal y geométrica, lo más probable es que $[sign(\alpha_k(P))<0 \wedge sign(\beta_k(P))\geq 0 \wedge sign(\gamma_k(P))\geq 0]_{j+1}$

Si lo anterior no se cumple pueden ocurrir dos cosas:

1. Si $[P \; in \; T_k]_{j+1}$ entonces $[P \; in \; F_+]_{j+1}$
2. Si $[not \; (P \; in \; T_k)]_{j+1}$ entonces
   2.1. Si $[sign(\beta_k(P))<0]_{j+1}$ entonces comenzar la búsqueda de un triángulo que cumpla el Lema 4.3 por el triángulo $T_{k\oplus 1}$ y en ese sentido.
   2.2. Si $[sign(\gamma_k(P))<0]_{j+1}$ entonces comenzar la búsqueda de un triángulo que cumpla el Lema 4.3 por el triángulo $T_{k\oplus -1}$ y en ese sentido.

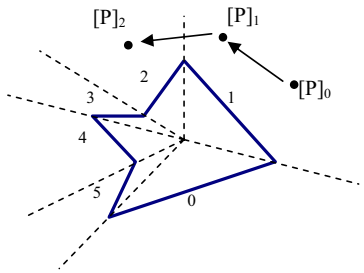
Cuando no es posible determinar la inclusión del punto, dado un triángulo del recubrimiento, esta heurística define cómo actuar para encontrar un triángulo que cumpla el Lema 4.3, si lo hay, basándonos en el signo de las coordenadas $\beta_k(P)$ ó $\gamma_k(P)$. Así, si $sign(\beta_k(P))<0$, el siguiente triángulo que debemos utilizar para comprobar que cumpla dicho lema será el $T_{k\oplus 1}$, y si no se cumple el $T_{k\oplus -1}$. Una vez establecida la dirección de la búsqueda, ésta se realiza en el mismo sentido, sin la necesidad de realizar comprobaciones adicionales. Si no existe ningún triángulo que cumpla el Lema 4.3, determinamos que no se produce la inclusión del punto en el polígono.

A continuación describimos el funcionamiento del algoritmo desarrollado en base a esta heurística que determina la DC entre un punto y un polígono con recubrimiento positivo respecto al centroide (Algoritmo 4.4 y Figura 4.9):

- Se calcula en primer lugar si $[P \; in \; F_+]$. Para realizar esto, se comprueba si no existe ningún triángulo del recubrimiento T_i que cumpla $[sign(\alpha_i(P))<0 \wedge sign(\beta_i(P))\geq 0 \wedge sign(\gamma_i(P))\geq 0]_0$.

- Si existe algún triángulo T_i que cumpla la condición anterior, utilizaremos ese triángulo como referencia para el siguiente frame.

- En los frames posteriores, se comprueba si $[sign(\alpha_i(P))<0 \wedge sign(\beta_i(P))\geq 0 \wedge sign(\gamma_i(P))\geq 0]$ utilizando el triángulo T_i anterior; si se cumple esta condición, el punto está fuera del polígono, si no, se comprueba si $[P \; in \; T_k]$ lo que implica que $[P \; in \; F_+]$, y en caso de que no se cumpla, se comprueban los triángulos contiguos $T_{i\oplus 1}$ y $T_{i\oplus -1}$ (en función del signo de la coordenada baricéntrica β) hasta encontrar algún triángulo T_k que cumpla que $[sign(\alpha_k(P))<0 \wedge sign(\beta_k(P))\geq 0 \wedge sign(\gamma_k(P))\geq 0]$ o que $[P \; in \; T_k]$.



$[sign(\alpha_1(P))=-1 \wedge sign(\beta_1(P)) \geq 0 \wedge sign(\gamma_1(P)) \geq 0]_0$, el punto está fuera

$[sign(\alpha_1(P))=-1 \wedge sign(\beta_1(P)) \geq 0 \wedge sign(\gamma_1(P)) \geq 0]_1$, el punto está fuera

$[sign(\alpha_1(P))=+1]_2$, comprobamos si $[sign(\beta_1(P))<0]_2$, como no es así la siguiente arista es la $1 \oplus 1=2$

$[sign(\alpha_2(P))=-1 \wedge sign(\beta_2(P)) \geq 0 \wedge sign(\gamma_2(P)) \geq 0]_2$, el punto está fuera

Figura 4.9: Funcionamiento del algoritmo para polígonos con recubrimiento positivo.

```
bool poligonoPositivo::testDeteccionColision2D(punto P, int & numtriangulo) {
   //Inclusion en la circunferencia envolvente
   if (this.circunferenciaEnvolvente().inclusion(P)==false) return false

   s = sign(this.triangulo(numtriangulo).α(P))
   t = sign(this.triangulo(numtriangulo).β(P))
   u = sign(this.triangulo(numtriangulo).γ(P))

   if (s<0 && t>=0 && u>=0)
      return false
   if (s>=0 && t>=0 && u>=0)
      return true
   for (tri=0;tri<this.numeroTriangulos();tri++) {
      if (t<0) indice = numtriangulo ⊕ (tri+1)
      else if (u<0) indice = numtriangulo ⊕ -(tri+1)
      s' = sign(this.triangulo(numtriangulo).α(P))
      t' = sign(this.triangulo(numtriangulo).β(P))
      u' = sign(this.triangulo(numtriangulo).γ(P))
      if (s'<0 && t'>=0 && u>=0) {
         numtriangulo = indice
         return false
      }
      if (s'>=0 && t'>=0 && u>=0) {
         numtriangulo = indice
         return true
      }
   }
   return true
}
```

Algoritmo 4.4: Detección de colisión Punto/Polígono con recubrimiento positivo.

4.3. Algoritmo de Detección de Colisión Circunferencia/Polígono

Una circunferencia puede usarse como volumen envolvente de un objeto 2D; o puede representar, por ejemplo, parte de un sistema de partículas 2D. Para éstas y otras aplicaciones es conveniente desarrollar un algoritmo de detección de colisión entre una circunferencia y un polígono. Este algoritmo será también la base de un algoritmo de detección de colisión entre polígonos como veremos en la Sección 4.4.

Vamos a definir a continuación los conceptos de offset, área de influencia, y área de influencia extendida de una arista. Mediante estos conceptos realizaremos una DC entre una circunferencia y una arista o un polígono realizando una DC entre el centro de la circunferencia y la arista extendida o el polígono extendido respectivamente.

Estas definiciones nos permitirán también establecer una medida de proximidad entre un punto y una arista, pudiendo utilizarse además para una detección de colisión aproximada, en la que se permite una determinada tolerancia. Para ello se establece un *offset*[*] del polígono a una distancia ε, siendo este ε el error permitido a la hora de obtener la colisión del punto con el polígono.

Básicamente, el algoritmo de detección de colisión entre una circunferencia y un polígono realiza una DC entre el centro de la circunferencia y el polígono. Si en este caso no se produce colisión realiza un test de intersección segmento/circunferencia, pero sólo con determinadas aristas, aquellas en cuya área de influencia extendida se encuentra la circunferencia. Veamos a continuación los conceptos necesarios para el desarrollo de este algoritmo.

4.3.1. Offset de una Arista

Antes de definir el concepto de *área de influencia* es necesario definir el de *offset* de una arista o segmento (Figura 4.10). El offset nos permitirá definir además el concepto de área de influencia extendida y posteriormente en la sección 4.5 los conceptos de triángulo y tri-cono extendidos.

[*] Se ha decidido mantener el término *offset* en inglés para no causar confusión debido a su amplio uso en Informática Gráfica y Detección de Colisiones.

Definición 4.2: Dado un polígono F con recubrimiento C_F y un triángulo $T_i=V_oV_1V_2$ tal que $T_i \in C_F$, definimos el *offset signado* de la arista V_1V_2 a una distancia r, al conjunto formado por las dos rectas que distan r unidades de la recta soporte de V_1V_2. Llamaremos *offset positivo* a la recta que se encuentra en la dirección de la normal, orientada hacia el exterior del polígono F, respecto de la arista V_1V_2; y *offset negativo* a la recta que se encuentra en la dirección opuesta.

Notación: Dado un polígono F con recubrimiento C_F y un triángulo $T_i=V_oV_1V_2$ tal que $T_i \in C_F$, notaremos el *offset positivo* de la arista V_1V_2 a una distancia r como $offset^+(V_1V_2,r)$ o también como $offset^+_i(r)$. El *offset negativo* lo notaremos como $offset^-(V_1V_2,r)$ o también como $offset^-_i(r)$. *

Debido a que el offset signado depende de la orientación del triángulo tenemos offsets distintos dependiendo de si el triángulo es positivo o negativo (Figura 4.10). Si el triángulo es positivo, el offset positivo se encuentra en el lado opuesto a V_o respecto de la arista V_1V_2. Si el triángulo es negativo, el offset positivo se encuentra en el mismo lado que V_o respecto de la arista V_1V_2.

El concepto de offset signado nos permitirá delimitar ciertas zonas del espacio, que posteriormente llamaremos áreas de influencia y áreas de influencia extendidas de una arista. Estas zonas serán útiles para obtener un algoritmo de detección de colisión entre una circunferencia y un polígono de forma parecida a como se lleva a cabo la DC entre un punto y un polígono. Además permitirá extender un polígono de manera que se pueda realizar la detección de colisión con una determinada tolerancia o error (Figura 4.11).

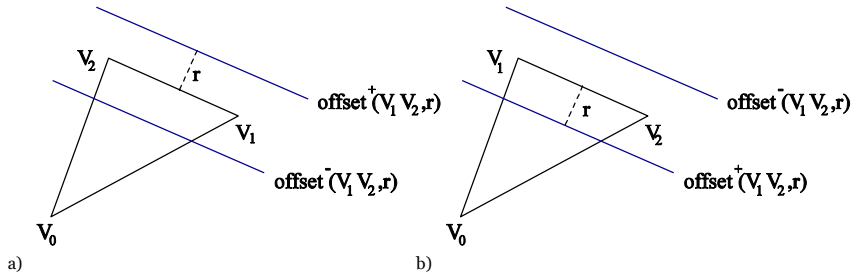


Figura 4.10: Offset de una arista, offset positivo y negativo. En el primer caso de un triángulo positivo y en el segundo de un triángulo negativo.

* Nos ha interesado definir el concepto de *offset* en relación a un triángulo del recubrimiento para poder utilizar una notación basada en el subíndice que representa dicho triángulo. Este mismo concepto puede ser definido para un triángulo que no pertenezca al recubrimiento del polígono, o directamente para un segmento. Del mismo modo que se ha definido para la arista V_1V_2, se puede definir el offset de la arista V_oV_1 ó V_2V_o.

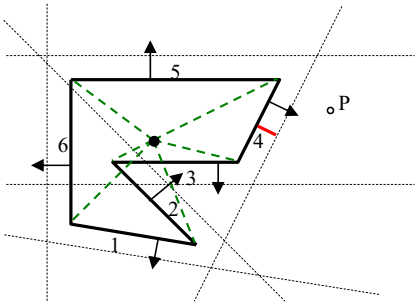


Figura 4.11: Offset positivo de las aristas de un polígono. Por ejemplo, se cumple que [*P in* $Offset_5(r)$] y [*not (P in* $Offset_4(r)$)].

4.3.2. Área de influencia de una Arista

Una vez definido el concepto de offset de una arista, pasamos a definir el de área de influencia de una arista (Figura 4.12). Esta será una región del espacio infinita delimitada por dos rectas como reflejamos en la siguiente definición:

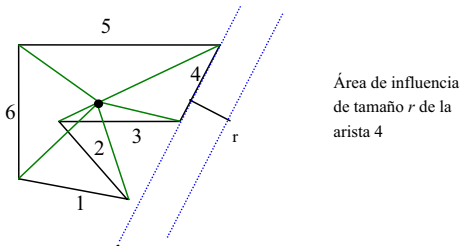


Figura 4.12: Área de Influencia de una arista.

Definición 4.3: Dado un polígono F y una arista $V_1V_2 \in$F, definimos el *área de influencia de tamaño r* de la arista V_1V_2, a la zona del espacio formada por todos los puntos comprendidos entre la recta soporte de la arista V_1V_2 y $offset^+(V_1V_2,r)$.

Notación: Dado un polígono F con recubrimiento C_F y un triángulo $T_i=V_0V_1V_2$ tal que $T_i \in C_F$, notaremos el área de influencia de tamaño *r* de la arista V_1V_2 como $AI(V_1V_2,r)$ o también como $AI_i(r)$.

Para determinar si un punto se encuentra incluido en el área de influencia de una arista podemos utilizar las coordenadas baricéntricas (Figura 4.13):

Propiedad 4.5: Sea F un polígono con recubrimiento C_F, $T_i=V_0V_1V_2$ un triángulo tal que $T_i \in C_F$ y P_{off} un punto tal que $P_{off} \in offset^+(V_1V_2,r)$. Entonces para todo punto *P* se cumple que [*P in* $AI(V_1V_2,r)$] sii $\alpha_i(P) \in [0, \alpha_i(P_{off})]$.

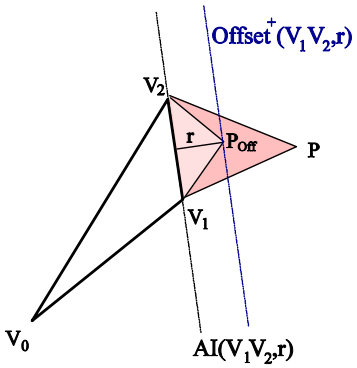


Figura 4.13: Cálculo de la inclusión de un punto en el área de influencia de una arista.

4.3.3. Área de influencia extendida de una arista

Definiremos el área de influencia extendida utilizando el concepto de offset positivo y negativo de una arista. El concepto de área de influencia extendida (Figura 4.14) es necesario para tratar adecuadamente los puntos cercanos a las concavidades de los polígonos, y para poder determinar la posición de la circunferencia respecto a la recta soporte de una arista.

Definición 4.4: Dado un polígono F y una arista $V_1V_2 \in F$, definimos el *área de influencia extendida de tamaño r* de la arista V_1V_2, a la zona del espacio formada por todos los puntos comprendidos entre $offset^+(V_1V_2,r)$ y $offset^-(V_1V_2,r)$.

Notación: Dado un polígono F con recubrimiento C_F y un triángulo $T_i=V_0V_1V_2$ tal que $T_i \in C_F$, notaremos el área de influencia extendida de tamaño *r* de la arista V_1V_2 como $AIE(V_1V_2,r)$ o también como $AIE_i(r)$.

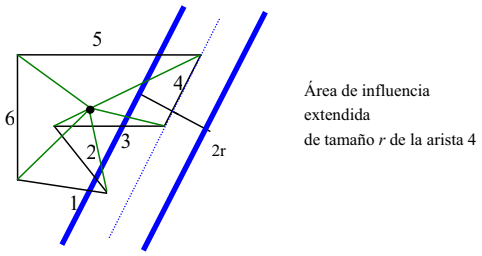


Figura 4.14: Área de influencia extendida de una arista.

Para determinar si un punto se encuentra incluido en el área de influencia extendida de una arista podemos utilizar las coordenadas baricéntricas de forma similar a como hicimos para el caso de áreas de influencia:

Propiedad 4.6: Sea F un polígono con recubrimiento C_F, $T_i=V_0V_1V_2$ un triángulo tal que $T_i \in C_F$, y P_{off} un punto tal que $P_{off} \in offset^+(V_1V_2,r)$. Entonces para todo punto P se cumple que [P *in* $AIE(V_1V_2,r)$] sii $\alpha_i(P) \in [-\alpha_i(P_{off}), \alpha_i(P_{off})]$.

El área de influencia extendida de tamaño r de una arista nos permitirá conocer si una circunferencia en movimiento interseca con una arista. Esto sólo puede ocurrir cuando el centro de la circunferencia se encuentre en el área de influencia de tamaño igual al radio de la circunferencia respecto a una arista, como veremos en la Sección 4.3.5.

4.3.4. Área de influencia extendida limitada

Los algoritmos que hemos desarrollado en este capítulo funcionan correctamente utilizando áreas de influencia y áreas de influencia extendidas. Sin embargo podemos obtener mejores tiempos en la detección de colisión si acotamos estas áreas aún más. Dependiendo de la inclinación del segmento respecto a la horizontal utilizaremos un par de rectas paralelas al eje X o al eje Y para realizar dicha acotación. Estas rectas se encuentran a una distancia r de los extremos de la arista sobre la que queremos calcular el área de influencia extendida de tamaño r (Figura 4.15.a).

El objetivo final de la utilización de las áreas de influencia consiste en obtener si se produce o no intersección entre una arista y una circunferencia. Para ello podríamos optar por calcular un área alrededor de la arista en la que se comprobara la inclusión del centro de la circunferencia y, si se produce esta inclusión, determinar que la circunferencia interseca con la arista. A este área se le conoce como *suma de Minkowski* [LGLM00] de un segmento y una circunferencia de radio r (Figura 4.15.b).

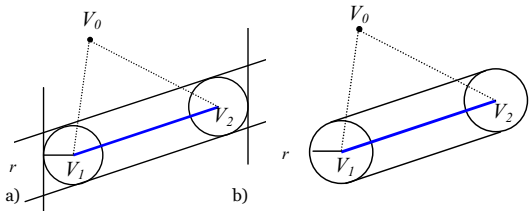


Figura 4.15: a) Área de Influencia extendida y limitada de tamaño r. b) Suma de Minkowski de un segmento y una circunferencia de radio r.

Otra alternativa menos costosa consiste en obtener un área de tamaño próximo a la suma de Minkowski del segmento y la circunferencia, pero mucho más eficiente en su cálculo, y una vez determinado que un punto se encuentra en dicha área, realizar un test de intersección segmento/circunferencia. Nosotros hemos optado por esta última opción, pues se obtienen mejores tiempos en la detección de colisión que con la solución anterior.

Definición 4.5: Dado un polígono F y una arista $V_1V_2 \in F$, definimos el área de influencia extendida y limitada de tamaño r de la arista V_1V_2, a la zona del espacio de todos los puntos comprendidos entre $AIE(V_1V_2,r)$ y un par de rectas a una distancia r de los extremos de la arista V_1V_2, de forma que dicha arista quede en el interior del área delimitada. Dichas rectas serán paralelas al eje X en caso de que la arista tenga pendiente mayor que 45º respecto al eje X, ó paralelas al eje Y en caso contrario.

Notación: Dado un polígono F con recubrimiento C_F y un triángulo $T_i=V_OV_1V_2$ tal que $T_i \in C_F$, notaremos el área de influencia extendida limitada de tamaño r de la arista V_1V_2 como $AIEL(V_1V_2,r)$ o también como $AIEL_i(r)$.

Posteriormente definiremos ciertos volúmenes envolventes que también acotarán las áreas de influencia definidas en secciones previas. De este modo no será necesario la utilización de áreas de influencia extendidas limitadas, pues el propio volumen envolvente servirá para limitar dichas áreas de extensión infinita.

4.3.5. Coherencia temporal y geométrica

Para comprobar la colisión entre una circunferencia y un polígono utilizaremos el algoritmo de DC Punto/Polígono, para ello ampliaremos el polígono usando las áreas de influencia extendidas de sus aristas y comprobaremos la colisión del centro C de la circunferencia de radio r con el polígono, lo que nos asegurará que se produce colisión entre la circunferencia y el polígono. Cuando C se encuentre fuera del polígono, comprobaremos la colisión de la circunferencia con aquellas aristas en cuya área de influencia extendida limitada de tamaño r se encuentre C.

Hay una serie de propiedades evidentes que se cumplen cuando se produce intersección entre una circunferencia y un polígono. Estas propiedades relacionan la intersección de una circunferencia y una arista del polígono con las áreas de influencia extendidas:

Dados un polígono F, una arista $V_1V_2 \in$ F, y una circunferencia E de centro *C* y radio *r*:

- **Propiedad 4.7:** Si [E $\cap$ $V_1V_2 \neq \emptyset$] entonces [*C in AIE*(V_1V_2,*r*)].

- **Propiedad 4.8:** Si [E $\cap$ F $\neq \emptyset$] entonces existe al menos una arista $V_jV_{j\oplus 1} \in$ F que cumple que [*C in AIE*($V_jV_{j\oplus 1}$,*r*)].

- **Propiedad 4.9:** Si [*C in* F] entonces [E *in* F]*.

Como consecuencia de las propiedades anteriores, podemos decir que cuando se produce colisión entre una circunferencia y un polígono, o bien se produce la inclusión del centro de la circunferencia en el polígono, o bien se produce la intersección de alguna arista del polígono con la circunferencia. Además las aristas que pueden estar implicadas en la colisión sólo pueden ser aquellas en cuya área de influencia extendida se encuentra el centro de la circunferencia (Figura 4.16).

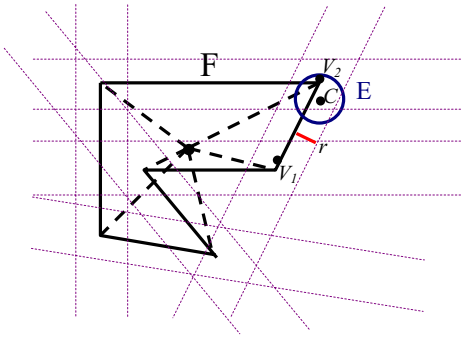


Figura 4.16: Para que se de intersección entre un polígono y una circunferencia E de radio *r*, el centro *C* de la circunferencia debe estar en el $AIE_i(r)$ de alguna arista.

El siguiente lema nos proporciona un mecanismo para aprovechar la coherencia temporal entre frames sucesivos para el caso de una circunferencia y un polígono.

Lema 4.4: Sea F un polígono con recubrimiento $C_F=\{T_i\}$, *i=1..n*. Sea E una circunferencia de centro *C* y radio *r*. Si $[C \text{ in } F]_j$ y $[sign(\alpha_i(C))]_j=[sign(\alpha_i(C))]_{j+1}$ para todo *i=1..n*, entonces $[E \text{ in } F]_{j+1}$. De igual modo, si se cumple que $[not\ (C \text{ in } F)]_j$ y que $[E \cap F \neq \emptyset]_{j+1}$, entonces debe existir algún triángulo $T_i=V_oV_1V_2$ tal que $T_i \in C_F$ y que $[V_1V_2 \cap E \neq \emptyset]_{j+1}$.

* Evidentemente [E *in* F] es equivalente a [E $\cap$ F $\neq \emptyset \vee$ ((*C in* F) $\wedge$ (E $\cap$ F = $\emptyset$))]

Demostración: Si el centro *C* de la circunferencia E se encuentra en el polígono y no cambia el signo de la coordenada baricéntrica α respecto a ninguno de los triángulos del recubrimiento entre un frame y el siguiente, *C* seguirá estando en el interior del polígono, debido al Lema 4.1. Al estar *C* dentro del polígono, o bien lo estará la circunferencia completa o bien interseca con alguna de las aristas del polígono. Si *C* no está en el interior del polígono, para que la circunferencia interseque con el polígono, debe existir una arista no original de un triángulo del recubrimiento que interseque con la circunferencia. □

Evidentemente en este lema podemos sustituir el término de área de influencia extendida (*AIE*) por el de área de influencia extendida limitada (*AIEL*). Además podemos aplicarlo al algoritmo de DC Circunferencia/Polígono como veremos en la siguiente sección.

4.3.6. Detección de Colisión Circunferencia/Polígono

En esta sección vamos a presentar el algoritmo genérico para la detección de colisión entre una circunferencia y un polígono no convexo (Algoritmo 4.5 y Figura 4.17). Este algoritmo es bastante sencillo, pues aplica el algoritmo de detección de colisión Punto/Polígono entre el centro de la circunferencia y el polígono. Sólo cuando el algoritmo Punto/Polígono determina que el centro *C* de la circunferencia está fuera del polígono se realiza la intersección entre la circunferencia y ciertas aristas, aquellas en cuya área de influencia de tamaño el radio de la circunferencia se encuentra el punto *C*.

Al igual que el resto de los algoritmos, se comprueba previamente si se produce intersección entre la circunferencia y la circunferencia envolvente del polígono mediante un sencillo test. Hemos incluido el algoritmo de intersección segmento/circunferencia para mostrar la sencillez del algoritmo global, en el que no es necesario realizar cálculos complejos.

```
static resultadoAnterior = false

bool Poligono::testDeteccionColision2D(Circunferencia c) {
   p = c.centro()
   radio = c.radio()
   // Intersección entre circunferencias envolventes.
   if (c.interseccion(this.circunferenciaEnvolvente())==false) return false

   // Deteccion de colision con el centro de la circunferencia
   resultado = this.testDeteccionColision2D(p)
   if (resultado==EQUAL_STATE && resultadoAnterior==true) return true
   resultadoAnterior = resultado
   if (resultado==true) return true

   // Cálculo de las aristas con las que puede intersectar
   for (i=0;i<this.numeroTriangulos();i++) {
      if (this.triangulo(i).AIEL(p,radio) == true) {
         if (c.interseccion(this.triangulo(i).V1V2)==true) return true
      }
   }
   return false
}

bool Circunferencia::interseccion(arista V1V2) {
   if (this.inclusion(V1)==true || this.inclusion(V2)) return true
   num = (this.centro()-V1) · (V2-V1)
   den = (V2-V1) · (V2-V1)
   if (num<0) return false
   if (num>den) return false
   return true
}
```

Algoritmo 4.5: Detección de Colisión Circunferencia/Polígono.

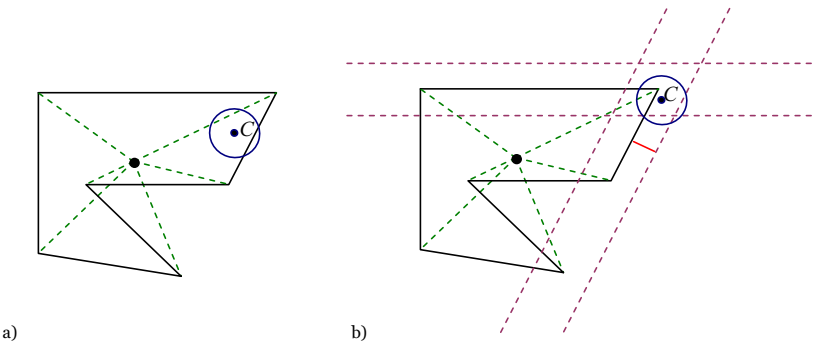


Figura 4.17: Modo de operación del algoritmo. a) El centro de la circunferencia está dentro del polígono. El algoritmo devuelve el estado de inclusión del centro en el polígono. b) El centro de la circunferencia se encuentra fuera del polígono. Se calcula la intersección con las aristas en cuya área de influencia extendida se encuentra el centro de la circunferencia.

4.4. Algoritmo de Detección de Colisión Polígono/Polígono

Para llevar a cabo la detección de colisión entre polígonos hemos desarrollado una serie de algoritmos auxiliares que sirven para realizar tareas básicas como la intersección entre segmentos, o la intersección entre un segmento y un polígono. Estos algoritmos se mostrarán en las secciones siguientes.

Básicamente, dados dos polígonos F_1 y F_2, el algoritmo realiza la detección de colisión entre la circunferencia envolvente de F_2 y el polígono F_1. Sólo cuando se produce colisión entre estos elementos se realiza un test detallado de intersección entre ciertas aristas de F_2 y F_1*. El algoritmo que presentamos es una optimización del algoritmo publicado en [JSF03] y [JSF04].

Antes de describir el algoritmo de detección de colisión entre polígonos, veamos una serie de algoritmos que realizan tareas básicas para la detección de colisión. Entre ellos un nuevo algoritmo de intersección entre segmentos mediante el uso de coordenadas baricéntricas.

4.4.1. Intersección Segmento/Segmento 2D

Basándonos de nuevo en las coordenadas baricéntricas de un punto respecto a un triángulo hemos obtenido un algoritmo de intersección entre un par de segmentos. Hemos utilizando este test, además de por su eficiencia, por homogeneidad en cuanto a los cálculos realizados basados en coordenadas baricéntricas.

Para describir adecuadamente este algoritmo es necesario definir el concepto de tri-cono. Este concepto no sólo será utilizado en el algoritmo de intersección entre segmentos, sino que será la base para la descomposición del recubrimiento de un polígono, como veremos en los algoritmos de detección de colisión desarrollados en la Sección 4.6.

Definición 4.6: Un *Tri-Cono* es la región angular definida por dos semi-rectas con un origen común, y dadas en un orden determinado. La región delimitada es aquella resultante de realizar un barrido angular desde la primera semi-recta a la segunda (Figura 4.18.a).

---

* Se comprueba la intersección entre las aristas que representan las áreas de influencia extendidas de F_1 en las que se encuentra F_2, con ciertas aristas de F_2, aquellas en las que se produce un cambio de signo en la coordenada baricéntrica α respecto al segmento utilizado para las áreas de influencia extendidas de F_1.

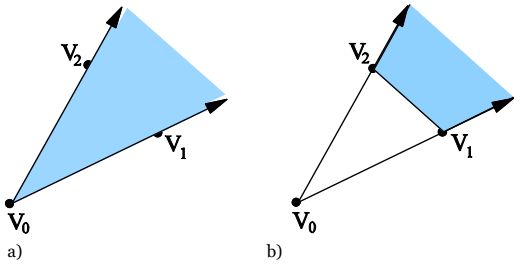


Figura 4.18: a) Tri-cono $\angle V_0V_1V_2$. b) Tri-cono truncado $\angle\backslash V_0V_1V_2$.

Un tri-cono puede definirse utilizando un triángulo si se establece uno de los vértices como el origen de las semi-rectas y los restantes vértices definen la dirección de las mismas, dadas en un orden determinado. Si los vértices del triángulo se dan en orden anti-horario, la región angular definida es la que contiene al triángulo.

Por tanto, podemos asociar un tri-cono a un triángulo de manera que este triángulo puede ser utilizado para definir el tri-cono. Para ello debe establecerse cuál es el vértice origen de las semi-rectas, que para nuestros cálculos será siempre el punto de referencia del recubrimiento del polígono, y el orden de las semi-rectas, que será el dado en la definición del triángulo[*]. Para nuestros fines utilizaremos siempre tri-conos asociados a triángulos positivos, por lo que en adelante nos referiremos exclusivamente a tri-conos cuyas semi-rectas forman un ángulo menor de 180 grados.

Notación: Sea T=$V_0V_1V_2$ un triángulo positivo, notaremos como $\angle$T, al tri-cono con origen en V_0 asociado a dicho triángulo

Notación: Sea T=$V_0V_1V_2$ un triángulo positivo y $\angle$T el tri-cono asociado. Notaremos que un punto P pertenece a $\angle$T como [P *in* $\angle$T].

Nótese que según las Propiedades 4.1 y 4.2 para todo punto P se cumple que [P *in* $\angle$T] sii [$sign(\beta(P)) \geq 0 \wedge sign(\gamma(P)) \geq 0$] respecto a T.

Definición 4.7: Sea T=$V_0V_1V_2$ un triángulo positivo y $\angle$T el tri-cono asociado. Definimos *tri-cono truncado* asociado al triángulo T, que notaremos $\angle\backslash$T, como la parte del tri-cono $\angle$T delimitada por la recta soporte de V_1V_2 de manera que V_0 es exterior a dicha región (Figura 4.18.b).

[*] Por ejemplo, dado el triángulo $V_0V_1V_2$, el tri-cono tendrá origen en V_0, con semi-rectas dadas en el orden V_0V_1 y V_0V_2.

Notación: Sea T=$V_0V_1V_2$ un triángulo positivo y $\angle\backslash$T el tri-cono truncado asociado a T. Notaremos que un punto *P* pertenece a $\angle\backslash$T como [*P in* $\angle\backslash$T].

Del mismo modo, según las Propiedades 4.1 y 4.2, para todo punto *P* se cumple que [*P in* $\angle\backslash$T] sii [$sign(\alpha(P)) \leq 0 \wedge sign(\beta(P)) \geq 0 \wedge sign(\gamma(P)) \geq 0$] respecto a T (Figura 4.18.b), o lo que es lo mismo: [$sign(\alpha(P)) \leq 0 \wedge$ *P in* $\angle$T].

A continuación vamos a establecer un teorema que nos permita comprobar si dos segmentos intersecan en el espacio bidimensional. Este teorema puede verse como una generalización del teorema postulado por Feito [FTU95], en el que se utiliza el área signada de triángulos formados con los extremos de los segmentos. En cambio nosotros formamos un triángulo con tres de los vértices de los segmentos y comprobamos si el vértice restante está incluido en el tri-cono truncado asociado a este triángulo (Algoritmo 4.6 y Figura 4.19).

Teorema 4.1: Sean *PQ y TU* dos segmentos tales que *Q* no está alineado con el segmento *TU*. El segmento *PQ* interseca con el segmento *TU*, es decir, [$PQ \cap TU \neq \emptyset$] sii [*P in* $\angle\backslash QTU$].

Demostración: Las coordenadas baricéntricas de *P* respecto al triángulo *QTU* nos determinan la posición en la que se encuentra el punto *P* respecto a dicho triángulo. Para que se produzca intersección entre los segmentos, el punto *P* debe estar en la región definida por las coordenadas $\alpha \geq 0$, $\beta \leq 0$ y $\gamma \leq 0$, que por definición expresan la zona del espacio definida por $\angle\backslash QTU$, pues para que se produzca intersección entre segmentos, el segmento *PQ* debe atravesar la recta soporte de *TU* ($\alpha \geq 0$), y no debe atravesar ni la recta soporte de *QT* ($\beta < 0$) ni la recta soporte de *QU* ($\gamma < 0$), aunque puede estar sobre dichas rectas soporte ($\beta = 0$ ó $\gamma = 0$). □

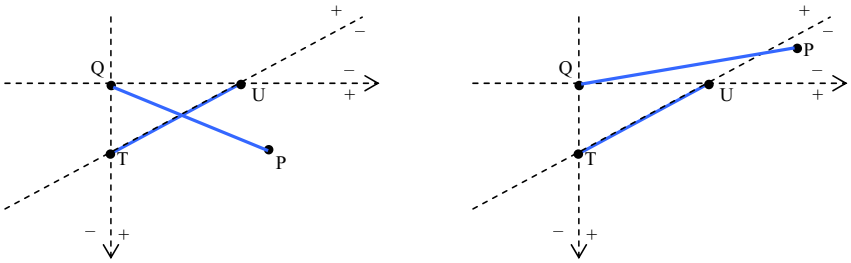


Figura 4.19: Intersección entre dos segmentos.

```
bool interseccion2D(punto p1,p2,q1,q2) {
// suponemos que q1 no está en la misma recta de p1p2
   T = triangulo(q1,p1,p2)
   s = sign(T.α(q2))
   if (s<=0) {
      t = sign(T.β(q2))
      if (t>=0) {
         u = sign(T.γ(q2))
         if (u>=0) {
            return true
         }
      }
   }
   return false
}
```

Algoritmo 4.6: Intersección entre segmentos.

En el caso de que *Q* esté alineado con el segmento *TU*, podemos utilizar el extremo *P* para formar $\angle\backslash PTU$ y comprobar la inclusión del punto *Q* en $\angle\backslash PTU$. Si el punto *P* estuviese alineado con *TU*, el segmento completo *PQ* se encontraría alineado con *TU*. Dependiendo del tipo de aplicación, podemos descartar el segmento, es decir, no considerar si se produce o no intersección entre segmentos. En el caso de intersección entre polígonos, habrá otra arista del polígono que intersecará con la arista en cuestión, con lo que no será necesario realizar dicha comprobación (Figura 4.20).

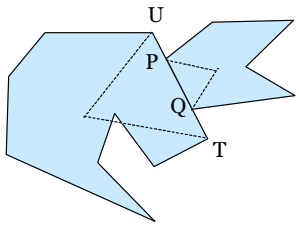


Figura 4.20: En el caso de segmentos alineados no es necesario comprobar su intersección pues habrá otro segmento que interseque con uno de los segmentos del polígono.

4.4.2. Intersección Segmento/Polígono 2D

La intersección entre un segmento y un polígono puede realizarse iterando un algoritmo de intersección entre segmentos, utilizando el segmento dado y todas las aristas del polígono. Si usamos el algoritmo de intersección entre segmentos definido

en la sección anterior, debemos tener en cuenta el caso en el que tres puntos estén alineados; para ello debemos seleccionar los extremos adecuados de los segmentos de manera que formen un triángulo correcto. Si esta situación no es posible, los segmentos están sobre la misma recta, y es necesario comprobar si intersecan en 1D. Para ello se puede proyectar sobre uno de los ejes *x* ó *y*, y realizar un sencillo test de solapamiento de intervalos [Ant92].

4.4.3. Intersección Polígono/Polígono 2D

Para realizar la intersección entre polígonos, podemos recorrer todas las aristas de un polígono F_1 y utilizar el algoritmo de intersección segmento/polígono anterior con cada una de las aristas de F_1 y el polígono F_2 [Vat92] [RF00]. Este algoritmo que acabamos de describir puede optimizarse para el caso de detección de colisión; para llevar a cabo esta optimización utilizaremos varias técnicas, como el uso de la coherencia y volúmenes envolventes, como veremos en las siguientes secciones.

4.4.4. Detección de Colisión Polígono/Polígono

Uno de los objetivos finales de esta memoria consiste en obtener un algoritmo de DC entre polígonos utilizando el esquema de representación de objetos mediante recubrimientos simpliciales. Vamos a describir este algoritmo, para el que utilizamos los conceptos desarrollados a lo largo de este capítulo:

Dados dos polígonos F_1 y F_2, realizaremos una DC circunferencia/polígono entre la circunferencia envolvente de F_2 y el polígono F_1, pero en lugar de comprobar la intersección entre la circunferencia y las aristas de F_1, se comprueba la intersección entre sólo ciertas aristas de F_2 y ciertas aristas de F_1, concretamente aquellas aristas de F_2 en las que se produce un cambio de signo en la coordenada α de sus vértices respecto a los triángulos del recubrimiento de F_1 en cuya área de influencia extendida se encuentre el centro de la circunferencia envolvente de F_2 (Figura 4.21). Para realizar esta tarea eficientemente se ha modificado el algoritmo de intersección entre segmentos (Algoritmo 4.6), de manera que se tenga en cuenta que los extremos de un segmento de F_2 están situados cada uno a un lado distinto respecto de un segmento de F_1.

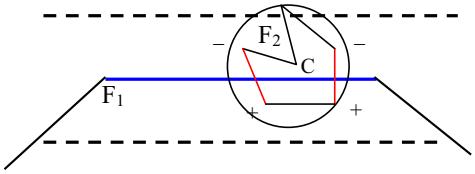


Figura 4.21: Cambio de signo de las aristas del polígono F_2 (en rojo) respecto a una de las aristas (en azul) de F_1 en cuya AIE se encuentra la circunferencia envolvente de F_2.

Además de lo anterior, para que el algoritmo funcione correctamente, es necesario que el centro *C* de la circunferencia envolvente de F_2 se encuentre dentro del propio polígono F_2, pues se comprueba la inclusión de *C* en F_1, y se determina que se produce colisión entre F_1 y F_2 cuando *C* se encuentra en F_1. En el caso de que *C* no esté dentro de F_2 podemos utilizar cualquier otro punto que pertenezca al polígono, como por ejemplo un vértice de F_2.

Este nuevo algoritmo (Algoritmo 4.7 y Figura 4.22) detecta la intersección entre dos polígonos aún cuando haya aristas de los dos polígonos que sean colineales o haya triángulos del recubrimiento degenerados.

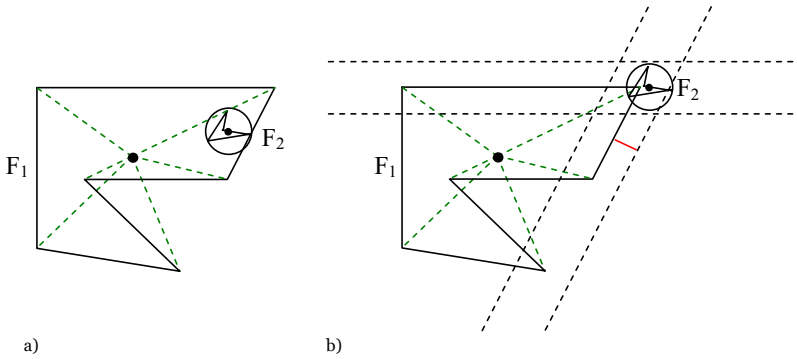


Figura 4.22: Modo de operación del algoritmo. a) El centro de la circunferencia envolvente de F_2 está dentro del polígono F_1. El algoritmo devuelve el estado de inclusión del centro en el polígono F_1. b) El centro de la circunferencia envolvente de F_2 se encuentra fuera del polígono F_1. Se calcula la intersección del polígono F_2 con las aristas de F_1 en cuya área de influencia extendida se encuentra el centro de la circunferencia, y sólo con aquellas aristas de F_2 en las que cambia el signo de α.

```
static resultadoAnterior = false

bool Poligono::testDeteccionColision2D(Poligono f) {
   p = f.circunferenciaEnvolvente().centro()
   radio = f.circunferenciaEnvolvente().radio()
   // Intersección entre circunferencias envolventes.
   if (f.circunferenciaEnvolvente().interseccion(this.circunferenciaEnvolvente())==false)
      return false
   // DC con el centro de la circunferencia o un punto que pertenezca al polígono
   resultado = this.testDeteccionColision2D(p)
   if (resultado==EQUAL_STATE && resultadoAnterior==true) return true
   resultadoAnterior = resultado
   if (resultado==true) return true
   // Cálculo de las aristas con las que puede intersectar
   for (i=0;i<this.numeroTriangulos();i++) {
      if (this.triangulo(i).AIEL(p,radio) == true) {
         if (this.triangulo(i).interseccionV1V2(f)==true) return true
      }
   }
   return false
}

bool Triangulo::interseccionV1V2(Poligono f) {
   if (this.degenerado()==true) {
      for (i=0;i<f.numeroVertices();i++)
         if (interseccion2D(f.vertice(i),f.vertice(i⊕1),this.V1,this.V2)==true)
            return true
      return false
   } else {
      s = sign(this.α(f.vertice(0)))
      for (i=0;i<f.numeroVertices();i++) {
         s' = sign(this.α(f.vertice(i⊕1)))
         if (s!=s')
            if (interseccion2Dmodificado(f.vertice(i),f.vertice(i⊕1),this.V1,this.V2)==true)
               return true
         s = s'
      }
      return false
   }
}

bool interseccion2Dmodificado(punto p1,p2,q1,q2) {
// suponemos que p1 y p2 están cada uno a un lado del segmento q1q2
   T = triangulo(q1,p1,p2)
   t = sign(T.β(q2))
   if (t>=0) {
      u = sign(T.γ(q2))
      if (u>=0) {
         return true
      }
   }
   return false
}
```

Algoritmo 4.7: Detección de Colisión Polígono/Polígono.

4.5. Descomposición mediante Tri-Trees

En esta sección definiremos una nueva estructura de datos que llamaremos *Tri-Tree*, utilizada para descomponer recursivamente el espacio y clasificar los triángulos del recubrimiento de un polígono. De este modo reduciremos el número de triángulos con los que tratar en la detección de colisión.

Utilizaremos conceptos como *tri-cono*, *árbol de tri-conos* y *triángulo envolvente* en el ámbito de un tri-cono. Aplicaremos estos conceptos a los algoritmos considerados en secciones anteriores para así obtener algoritmos más eficientes.

Los métodos de detección de colisión habituales suelen realizar un pre-procesamiento que consiste en descomponer un polígono complejo en partes más sencillas, posiblemente convexas, realizar una triangulación, o clasificar las características del polígono mediante diversos métodos de descomposición espacial como rejillas o quadtrees, según vimos en el Capítulo 2. De esta forma, realizar un test detallado de colisión implica realizarlo entre un menor número de características.

Muchos de los métodos clásicos utilizados para clasificar las características de un polígono requieren de estructuras de datos especiales que pueden ser difíciles de actualizar entre frames, o en el caso de volúmenes envolventes, puede que no se ajusten adecuadamente a los objetos a representar. En ambos casos se incrementa el coste de detección de colisión, en primer lugar debido a la actualización de las estructuras de datos utilizadas, y en segundo lugar debido al mayor número de pares de características a tratar, a causa de un ajuste inadecuado del volumen envolvente.

Para aumentar la eficiencia de los algoritmos desarrollados en secciones anteriores utilizaremos una nueva estructura de descomposición espacial denominada *árbol de tri-conos* o *Tri-Tree*. Un Tri-Tree está formado por *tri-conos* que se dividen de forma recursiva en nuevos tri-conos. Esta estructura de datos está compuesta por un conjunto de tri-conos con un origen común, el centroide del polígono[*], de manera que estos tri-conos cubren todo el espacio sin solapamiento entre ellos, salvo en la frontera. Un tri-cono puede subdividirse, utilizando nuevamente tri-conos, y así formar un árbol de tri-conos o tri-tree. Los triángulos del recubrimiento de un polígono se clasifican en los nodos de cada nivel del árbol de tri-conos generado. Además, asociado a cada tri-cono tendremos un triángulo y una circunferencia envolvente que se utilizarán para acotar aún más el espacio delimitado por un tri-cono.

[*] En nuestro caso particular, haremos coincidir el origen de los tri-conos con el origen del recubrimiento del polígono.

Utilizar esta estructura de datos supone una serie de ventajas frente a otras estructuras, como la invariabilidad cuando se aplican traslaciones, rotaciones y escalados uniformes. Además esta estructura se ajusta de una forma muy natural a la geometría del polígono, siendo muy rápida de calcular además de necesitar poco espacio de almacenamiento.

4.5.1. Definición de Tri-Tree

En la Sección 4.4.1. hemos definido un tri-cono utilizando un triángulo, de manera que un vértice del triángulo junto a los dos restantes definen las dos semi-rectas que forman dicho tri-cono. La región delimitada es aquella en la que se encuentra el triángulo considerado.

Construiremos una estructura jerárquica que descomponga el espacio usando tri-conos. Realizaremos esta descomposición para cada uno de los objetos sobre los que deseemos obtener la detección de colisión. De este modo, crearemos un conjunto de tri-conos con origen en el centroide del objeto, y estos tri-conos se irán subdividiendo mediante nuevos tri-conos hasta alcanzar una profundidad máxima o cumplir algún otro criterio de parada. El recubrimiento del polígono debe tener también como origen el centroide del objeto.

Definición 4.8: Un árbol de tri-conos o *Tri-Tree* es una estructura de datos jerárquica en forma de árbol de cuyo nodo raíz parten cuatro tri-conos hijos ($\angle T_0, \angle T_1, \angle T_2, \angle T_3$) todos con origen común y que cubren la totalidad del espacio sin solapamientos entre ellos, salvo en la frontera. Cada uno de estos cuatro tri-conos de primer nivel se descompone en dos sub-tri-conos hijos (Figura 4.23), de manera que el tri-cono $\angle T_i$ tendrá dos tri-conos de siguiente nivel ($\angle T_{i0}, \angle T_{i1}$). Del mismo modo, cada tri-cono se subdivide recursivamente en dos sub-tri-conos, hasta alcanzar un criterio de parada predefinido. Cada sub-tri-cono se construye a partir del tri-cono padre $\angle T_i$ definido por los vértices V_0, V_1, V_2 de la siguiente forma:

$$\angle T_{i0} = (V_0, V_1, V')$$
$$\angle T_{i1} = (V_0, V', V_2)$$

siendo $V' = (V_1+V_2)/2$ como norma general, o siendo V' cualquier otro punto incluido en el tri-cono que se quiere subdividir.

El criterio de parada que utilizaremos consiste en alcanzar una profundidad máxima en el árbol o que el número de triángulos del recubrimiento clasificados en un tri-cono sea menor o igual que un valor mínimo establecido de antemano.

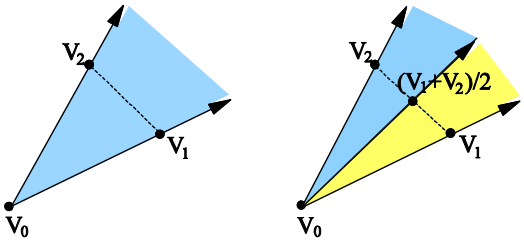


Figura 4.23: Un tri-cono y sus dos sub-tri-conos.

Dado un polígono construiremos un tri-tree formado por un conjunto de tri-conos que lo envuelvan completamente y sin que se produzca solapamiento entre ellos, salvo en la frontera. Este conjunto de tri-conos no es único, pero cualquiera de ellos es válido para nuestros fines.

Por razones de eficiencia, para cada polígono crearemos inicialmente cuatro tri-conos de primer nivel, alineados con los ejes y todos ellos con origen en el centroide del polígono, de manera que si realizamos una traslación del polígono al origen de coordenadas cada uno de estos tri-conos delimita uno de los cuatro cuadrantes en los que se divide el espacio (Figura 4.24 y 4.25):

- El tri-cono $\angle T_0$ de primer nivel representa el cuadrante ($x \geq 0$, $y \geq 0$), estando T_0 formado por los vértices $V_0=(0,0)$, $V_1=(1,0)$ y $V_2=(0,1)$.

- El tri-cono $\angle T_1$ de primer nivel representa el cuadrante ($x \leq 0$, $y \geq 0$), estando T_1 formado por los vértices $V_0=(0,0)$, $V_1=(0,1)$ y $V_2=(-1,0)$.

- El tri-cono $\angle T_2$ de primer nivel representa el cuadrante ($x \leq 0$, $y \leq 0$), estando T_2 formado por los vértices $V_0=(0,0)$, $V_1=(-1,0)$ y $V_2=(0,-1)$.

- El tri-cono $\angle T_3$ de primer nivel representa el cuadrante ($x \geq 0$, $y \leq 0$), estando T_3 formado por los vértices $V_0=(0,0)$, $V_1=(0,-1)$ y $V_2=(1,0)$.

Una vez construido el primer nivel de tri-conos se clasifican los triángulos del recubrimiento del polígono en cada uno de estos tri-conos. Los tri-conos se subdividen en nuevos sub-tri-conos hasta alcanzar el criterio de parada definido para el tri-tree, clasificando los triángulos del recubrimiento en los tri-conos de cada nuevo nivel del árbol. El conjunto de triángulos del recubrimiento a clasificar en los tri-conos hijos está restringido a los triángulos clasificados en el tri-cono padre. Evidentemente cuando se aplica cualquier tipo de transformación al objeto, se aplica también al tri-tree completo.

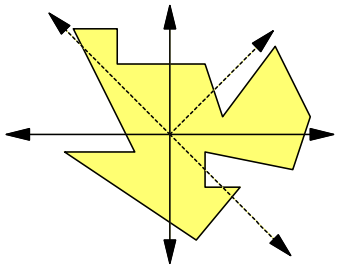


Figura 4.24: Tri-conos del tri-tree, primer nivel en trazo continuo. Segundo nivel en trazo discontinuo.

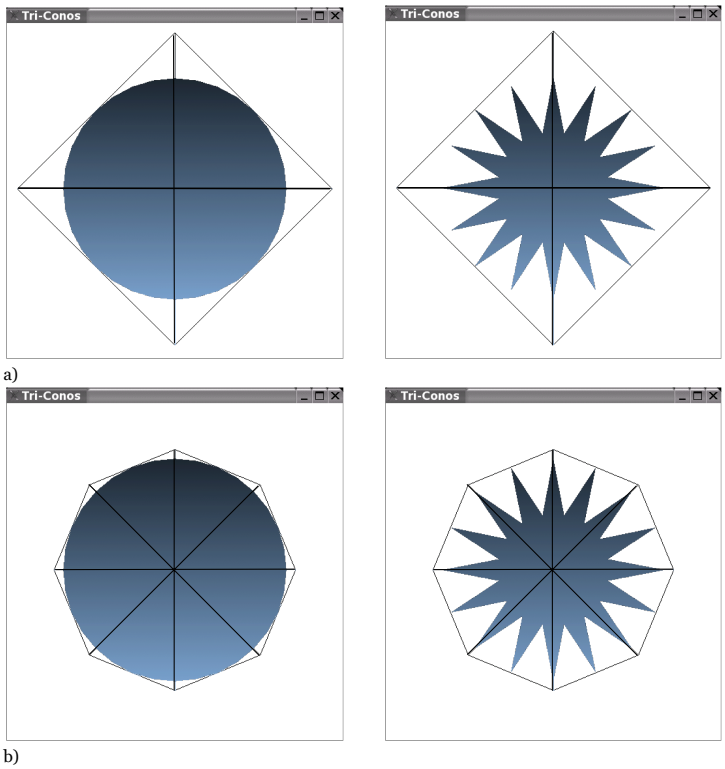


Figura 4.25: Ejemplo de polígonos (circunferencia y polígono estrellado, en color sólido) y triángulos utilizados para la definición de los tri-conos de un nivel determinado. a) Triángulos que definen los tri-conos de primer nivel (4 divisiones del espacio). b) Triángulos que definen los tri-conos de segundo nivel (8 divisiones del espacio).

4.5.2. Criterios para la clasificación de triángulos en un Tri-Tree

Vamos a establecer la manera de clasificar un triángulo del recubrimiento de un polígono en un tri-cono del tri-tree asociado al polígono. Teniendo en cuenta que todos los triángulos del recubrimiento y todos los tri-conos del tri-tree asociado tienen origen en el centroide del polígono, realizaremos una clasificación de dichos triángulos del recubrimiento. Hablaremos de *triángulos no contenidos* en el tri-cono cuando no intersecan con el espacio del tri-cono (salvo, claro está, el vértice común del recubrimiento); y de *triángulos total o parcialmente contenidos* en el tri-cono cuando intersecan con el tri-cono.

A continuación vamos a describir cada una de las situaciones posibles en las que se puede encontrar un triángulo del recubrimiento de un polígono respecto a un tri-cono del tri-tree del polígono. Supondremos que tanto el recubrimiento como el tri-tree asociado a un polígono tienen como origen el centroide del polígono:

Sea F un polígono con recubrimiento C_F y tri-tree asociado TT_F. Sea $S \in C_F$ y $\angle T \in TT_F$:

- **Definición 4.9:** Diremos que el triángulo S *no está contenido* en el tri-cono $\angle T$ cuando el único punto en común entre el triángulo S y el tri-cono $\angle T$ es el vértice original del tri-cono (Figura 4.26.a).

- **Definición 4.10:** Diremos que el triángulo S *está totalmente contenido* en el tri-cono $\angle T$ cuando el triángulo S está dentro del tri-cono $\angle T$ (Figura 4.26.b).

- **Definición 4.11:** Diremos que el triángulo S *está parcialmente contenido* en el tri-cono $\angle T$ cuando hay parte del triángulo S que está dentro del tri-cono $\angle T$ y parte del triángulo S que está fuera del tri-cono $\angle T$ (Figura 4.26.c-d).

Definición 4.12: Dado un triángulo S y un tri-cono $\angle T$, diremos que S está *clasificado* en $\angle T$, y lo notaremos como [S *clasif* $\angle T$], a la situación en la que el triángulo S está total o parcialmente contenido en el tri-cono $\angle T$.

Según las definiciones previas, cuando un lado del triángulo yace sobre una de las semi-rectas del tri-cono, estamos considerando que el triángulo está parcial o totalmente contenido en el tri-cono, según corresponda. Esto es correcto para nuestros propósitos, pues el objetivo de esta clasificación consiste en obtener todos los triángulos que tienen alguna parte en común con el tri-cono, aparte del origen del recubrimiento.

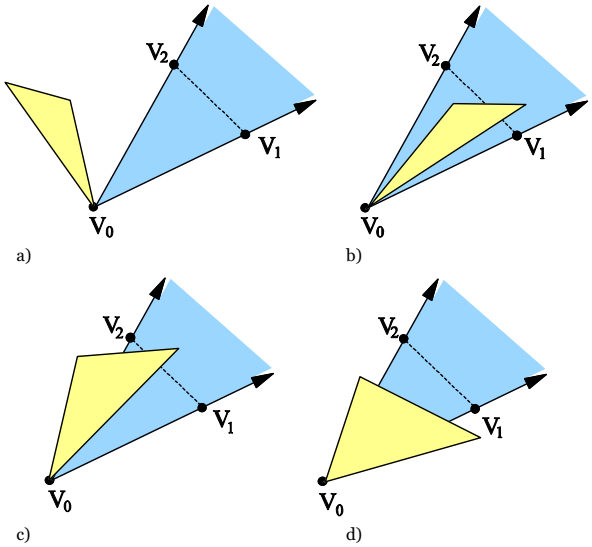


Figura 4.26: Relación entre un Triángulo (en amarillo) y un Tri-cono (en azul). a) Triángulo no contenido en $\angle V_0V_1V_2$, b) Triángulo totalmente contenido en $\angle V_0V_1V_2$. c) y d) Triángulos parcialmente contenidos.

Podemos utilizar las siguientes propiedades para obtener eficientemente los triángulos del recubrimiento de un polígono que están clasificados en un tri-cono:

Sea F un polígono con recubrimiento C_F y tri-tree asociado TT_F. Sea $S \in C_F$ y $\angle T \in TT_F$:

- **Propiedad 4.10:** Si alguno de los vértices no originales de S se encuentra en el tri-cono $\angle T$, entonces [S *clasif* $\angle T$].

  Esta propiedad contempla el caso de triángulos totalmente contenidos (Figura 4.26.b) y parcialmente contenidos con algún vértice en el tri-cono (Figura 4.26.c).

- **Propiedad 4.11:** Si alguno de los vértices no originales de T se encuentra en el tri-cono $\angle S$, entonces [S *clasif* $\angle T$].

  Al intercambiar los papeles entre el triángulo a clasificar y el tri-cono obtenemos los triángulos que intersecan con el tri-cono en el caso de la Figura 4.26.d.

El algoritmo que clasifica los triángulos del recubrimiento de un polígono en un tri-tree comprueba las dos condiciones anteriores para determinar la inclusión de cada

triángulo en cada uno de los tri-conos de un nivel determinado. Cuando se han clasificado todos los triángulos del recubrimiento del polígono en los tri-conos de un nivel del tri-tree, se clasifican nuevamente en los tri-conos del siguiente nivel, hasta llegar a la profundidad máxima del árbol o hasta que el número de triángulos clasificados en un tri-cono sea menor o igual que un valor mínimo. Evidentemente, el conjunto de triángulos candidatos a ser clasificados en un tri-cono está restringido a los triángulos clasificados en el tri-cono padre.

4.5.3. Triángulo envolvente asociado a un Tri-Cono

Vamos a definir el concepto de *triángulo envolvente* asociado a un tri-cono. Este triángulo envolvente pondrá un límite superior a los vértices de todos los triángulos del recubrimiento clasificados en el tri-cono, de manera que todos estos vértices tendrán coordenada baricéntrica α positiva o cero respecto al triángulo envolvente (Figura 4.27).

Definición 4.13: Sea F un polígono con tri-tree asociado TT_F. Sea $T=V_0V_1V_2$ un triángulo tal que $\angle T \in TT_F$. Definimos *triángulo envolvente* de la geometría de F contenida en $\angle T$ como el triángulo $T_{env}= V_0V'_1V'_2$ tal que:

1. V'_1 está sobre la semi-recta definida por V_0V_1.
2. V'_2 está sobre la semi-recta definida por V_0V_2.
3. Todo vértice V_i de los triángulos del recubrimiento de F clasificados en $\angle T$ cumplen que $sign(\alpha(V_i)) \geq 0$ respecto a T_{env}.

Dado un tri-cono, existe más de un posible triángulo envolvente, por lo que utilizaremos un triángulo envolvente que se ajuste en la medida de lo posible a los triángulos del recubrimiento clasificados en su tri-cono asociado, según veremos al final de esta sección.

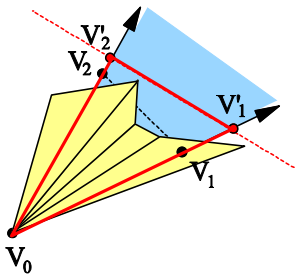


Figura 4.27: Triangulo envolvente $V_0V'_1V'_2$ asociado a un tri-cono $\angle V_0V_1V_2$ y triángulos incluidos en el mismo.

Para clasificar un triángulo del recubrimiento de un polígono respecto a un triángulo envolvente utilizaremos la coordenada baricéntrica α de los vértices de dicho triángulo respecto al triángulo envolvente.

Definición 4.14. Sea F un polígono con tri-tree asociado TT_F. Sea $S=V_0V_1V_2$ un triángulo tal que $S \in C_F$. Sea $\angle T$ un tri-cono tal que $\angle T \in TT_F$, y sea T_{env} un triángulo envolvente de la geometría de F contenida en $\angle T$. Diremos que S está *clasificado* en T_{env}, y lo notaremos como [S *clasif* T_{env}], si se cumple que [S *clasif* $\angle T$] y además $sign(\alpha(V_i)) \geq 0$, $i=0..2$.

Es posible que el triángulo envolvente no contenga a todos los vértices de los triángulos del recubrimiento clasificados en el tri-cono, pues existen vértices que pueden pertenecer a triángulos parcialmente contenidos en el tri-cono. Por este motivo hemos utilizado la coordenada baricéntrica α para delimitar el triángulo envolvente (apartado 3 de la Definición 4.13), pues aunque hay vértices no incluidos en el tri-cono, si sólo consideráramos los vértices incluidos en el mismo, parte de la geometría del polígono podría quedar fuera del triángulo envolvente. De este modo nos aseguramos que el triángulo envolvente lo sea de toda la geometría del polígono incluida en un tri-cono (Figura 4.27).

Para ajustar un triángulo envolvente a la geometría del polígono se ha utilizado un algoritmo iterativo que parte de un triángulo envolvente isósceles (uno con una distancia de V_1 y V_2 respecto a V_0 igual a la mayor de las distancias entre V_0 y cada uno de los vértices de los triángulos del recubrimiento clasificados en el tri-cono). A partir de este triángulo envolvente, se calcula el punto medio de una de las aristas que contienen a V_0. Si el nuevo triángulo, formado por dicho punto y el resto de los vértices del triángulo envolvente anterior, es un triángulo envolvente, se itera de nuevo el algoritmo con el nuevo triángulo envolvente generado, hasta un número determinado de veces. Si el triángulo formado no es un triángulo envolvente, el nuevo vértice es el punto medio entre el vértice anterior y el calculado. Cuando se ha terminado con una de las dos aristas del triángulo envolvente que contiene a V_0, se empieza con la siguiente y se realiza el mismo proceso (Figura 4.28, 4.29 y 4.30).

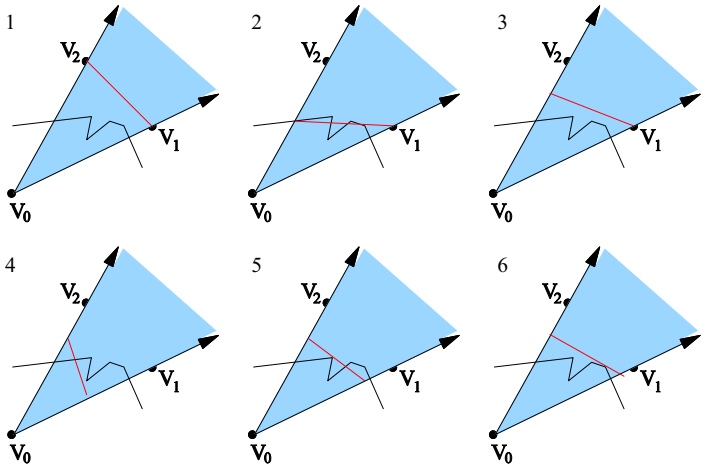


Figura 4.28: Iteraciones realizadas para calcular un triángulo envolvente.

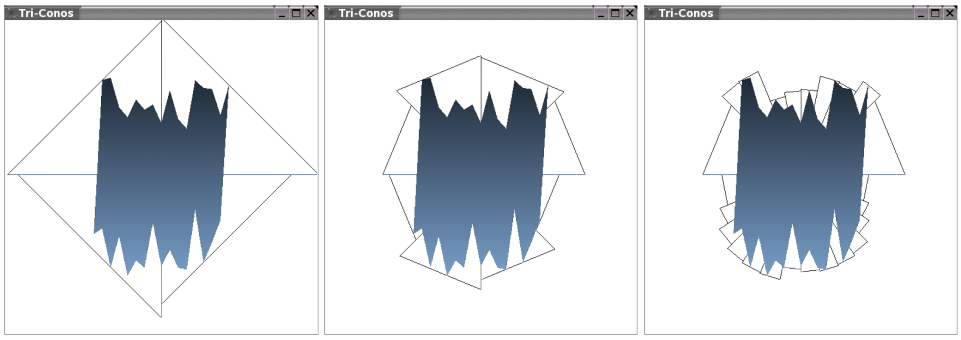
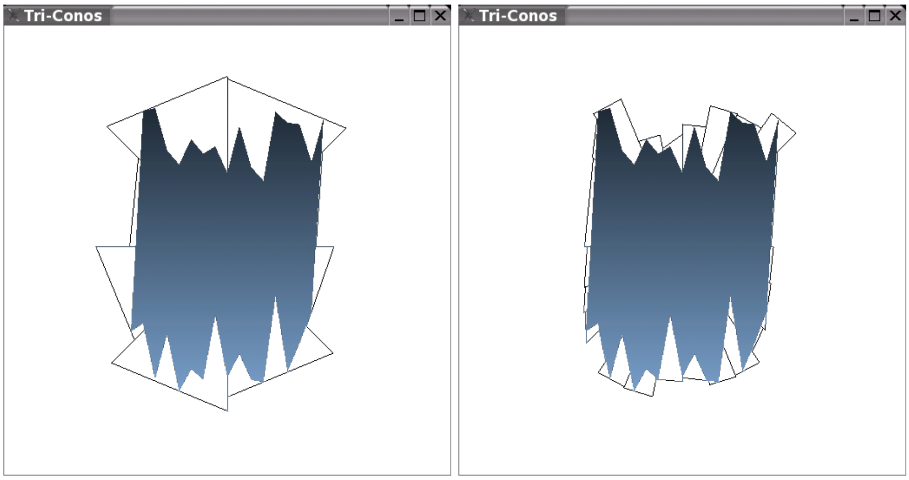
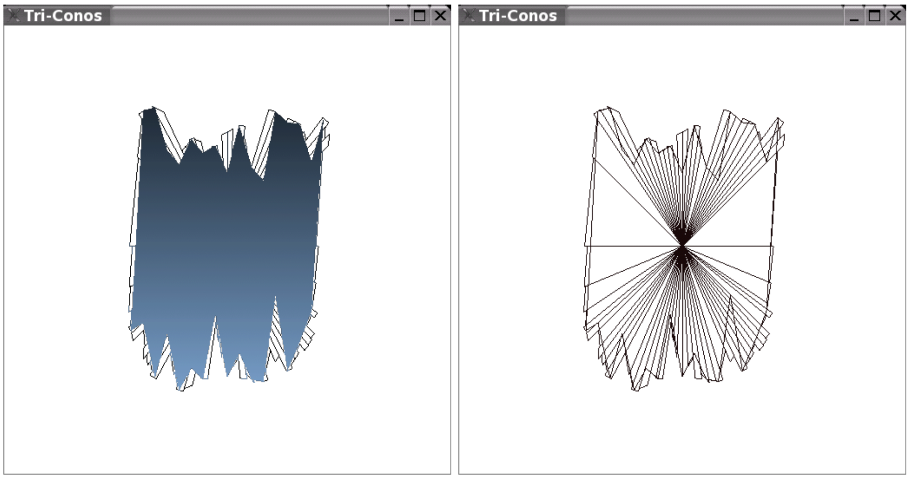


Figura 4.29: Triángulos envolventes de nivel 1, 2 y 4 respectivamente. En ninguno de los tres casos se ha iterado el algoritmo de ajuste del triángulo envolvente, por lo que el triángulo mostrado se corresponde con el triángulo isósceles de partida. En algunos casos no se ha subdividido el tri-cono correspondiente por tener un número de triángulos del recubrimiento muy pequeño.



a) Triángulos envolventes de nivel 2. b) Triángulos envolventes de nivel 4.



c) Triángulos envolventes de nivel 6. d) Triángulos envolventes de nivel 6.

Figura 4.30: Triángulos envolventes de nivel 2, 4 y 6 respectivamente (la última figura se ha duplicado para mostrar los triángulos envolventes de una forma más clara). En todos los casos se ha iterado el algoritmo de ajuste de los triángulos envolventes.

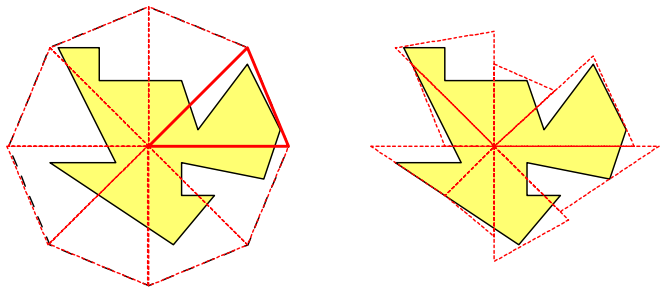
4.5.4. Envolvente regular de un polígono

A continuación vamos a definir una *envolvente regular* para un polígono, de manera que podamos aprovechar ciertas propiedades que permitan descartar eficientemente triángulos envolventes no involucrados en la colisión. Dicha envolvente regular estará formada por determinados triángulos envolventes isósceles asociados a los tri-conos de un nivel del tri-tree, que deben cumplir una serie de propiedades adicionales como veremos a continuación. Tendremos por tanto una envolvente regular del polígono por cada nivel de profundidad del tri-tree correspondiente.

Definición 4.15: Sea F un polígono con tri-tree asociado TT_F. Sea TT_{env} el conjunto de triángulos envolventes asociados a un nivel de TT_F. Sea d_{max} el máximo de las distancias entre V_o y cada uno de los vértices V'_1 y V'_2 de todos los triángulos de TT_{env}. Definimos la *envolvente regular* de F, $TT_{env\text{-}reg}$, como el conjunto de triángulos envolventes resultantes de substituir en cada triángulo envolvente $T_{env} \in TT_{env}$ los puntos V'_1 y V'_2 por V''_1 y V''_2 situados sobre las semi-rectas $V_oV'_1$ y $V_oV'_2$ respectivamente y a una distancia d_{max} de V_o.

Notación: Dado un polígono F con envolvente regular $TT_{env\text{-}reg}$, notaremos como $T_{env\text{-}reg}$ a un triángulo envolvente que forme parte de la envolvente regular $TT_{env\text{-}reg}$.

Intuitivamente, la definición de envolvente regular nos permite obtener para cada nivel del tri-tree asociado a un polígono, un conjunto de triángulos envolventes cuyas aristas no originales forman una envolvente convexa del polígono (Figura 4.31).



a) b)

Figura 4.31: a) Triángulos envolventes que forman parte de la envolvente regular de nivel 2 del polígono. b) Triángulos envolventes asociados a los tri-conos de nivel 2 del polígono.

4.5.5. Circunferencia envolvente asociada a un Tri-Cono

Podemos asociar a cada tri-cono, además de un triángulo envolvente, una circunferencia envolvente. Dicha circunferencia contendrá todos los vértices no

originales de los triángulos del recubrimiento clasificados en un tri-cono (Figura 4.32). Mediante este volumen envolvente descartaremos de manera más rápida triángulos envolventes sobre los que realizar un test de intersección en la detección de colisión entre polígonos.

Definición 4.16: Dado un polígono F y un tri-cono $\angle T \in TT_F$, definimos la *circunferencia envolvente* de los vértices no originales de los triángulos del recubrimiento de F clasificados en dicho tri-cono $\angle T$, a la circunferencia $E_{env}(C,r)$ tal que *C* es el centroide de dichos vértices y *r* es el máximo de las distancias de *C* a cada uno de los vértices.

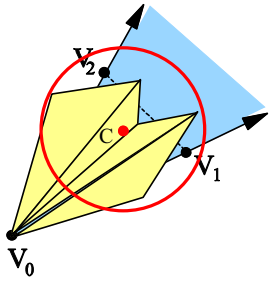


Figura 4.32: Circunferencia envolvente de los vértices no originales de los triángulos clasificados en un tri-cono.

Esta circunferencia envolvente junto con el triángulo envolvente definido previamente se utilizan para acotar la geometría incluida en el tri-cono al que van asociados (Figura 4.33).

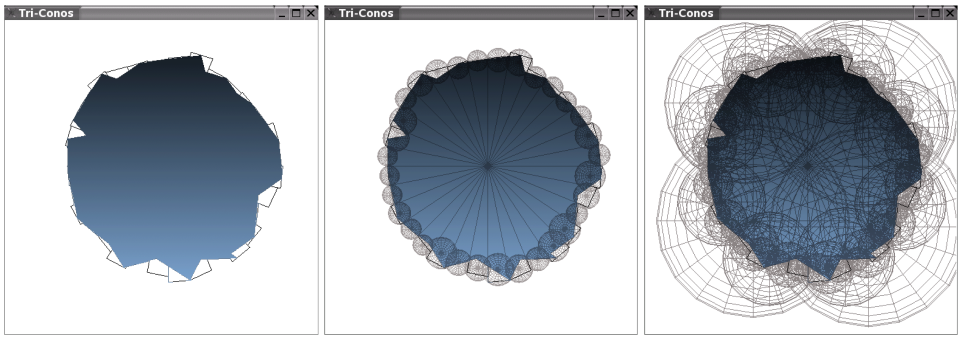


Figura 4.33: a) Polígono y triángulos envolventes de nivel 4. b) Polígono, circunferencias envolventes y tri-conos envolventes de nivel 4. c) Polígono y circunferencias envolventes asociadas a los niveles 1 a 4.

4.6. Algoritmos de Detección de Colisión utilizando Tri-Trees

En esta sección modificaremos y actualizaremos los algoritmos de detección de colisión mostrados en secciones anteriores para adaptarlos a la utilización de tri-trees. Gracias a la clasificación de los triángulos del recubrimiento realizada utilizando tri-trees, podremos aplicar los conceptos y técnicas anteriores, pero restringidas sólo a partes del polígono, es decir, aquellos triángulos del recubrimiento clasificados en determinados tri-conos.

4.6.1. Algoritmo de Detección de Colisión Punto/Polígono

Veamos a continuación el caso de la detección de colisión entre un punto y un polígono. Pero antes vamos a describir cómo realizar un uso adecuado de la coherencia tanto espacial como temporal, es decir, cómo restringir los cálculos a determinados triángulos del recubrimiento clasificados en un tri-cono, y cómo comenzar la detección de colisión entre frames con el mismo tri-cono del frame anterior.

4.6.1.1. Coherencia espacial

Partiendo de la definición de recubrimiento simplicial de un polígono, hemos desarrollado en secciones anteriores sendos algoritmos de inclusión y de colisión que calculan una sumatoria signada para obtener su resultado, utilizando para ello la posición relativa del punto respecto a cada uno de los triángulos del recubrimiento.

Si de alguna forma aseguramos que no se produce inclusión en ciertos triángulos del recubrimiento, podemos no considerarlos en esa sumatoria signada (al valer cero su inclusión). Podemos establecer una serie de propiedades que, de cumplirse, nos aseguran que se pueden descartar ciertos triángulos del recubrimiento y tratar sólo con determinados triángulos para obtener la inclusión del punto en el polígono. Para esto vamos a utilizar el concepto de tri-cono definido previamente, de manera que sólo trataremos con los triángulos clasificados en un determinado tri-cono, aquel en el que se encuentra el punto sobre el que queremos conocer su inclusión o su colisión con el polígono.

Definición 4.17: Dado un polígono F con tri-tree asociado TT_F, un punto *P* y un tri-cono $\angle T$ tal que $\angle T \in TT_F$, diremos que el punto *P se encuentra en la parte del recubrimiento de* F *clasificada en el tri-cono* $\angle T$, y lo notaremos como [*P in* $C_{\angle T}$], si

se cumple que *P* está en el tri-cono $\angle T$ y además *P* está incluido en la parte del polígono representada por los triángulos del recubrimiento clasificados en el tri-cono $\angle T$.

A continuación mostramos estas propiedades:

- **Propiedad 4.12:** Sea F un polígono y TT_F su tri-tree asociado. Sea $\angle T$ un tri-cono tal que $\angle T \in TT_F$. Sea *P* un punto tal que [*P in* $\angle T$]. Entonces [*P in* F] sii [*P in* $C_{\angle T}$].

  Esto es así debido a que si un punto se encuentra en un determinado tri-cono, sólo puede estar incluido en los triángulos del recubrimiento clasificados en dicho tri-cono. Para demostrar esto podemos descomponer el recubrimiento del polígono en dos conjuntos de triángulos, uno el de los triángulos que están clasificados en el tri-cono y otro el del resto de triángulos. Como el punto se encuentra en el tri-cono, no puede haber inclusión en ningún triángulo no contenido en el tri-cono, por tanto la inclusión sólo puede darse en triángulos clasificados en el tri-cono.

- **Propiedad 4.13:** Sea F un polígono y TT_F su tri-tree asociado. Entonces para todo punto *P* existe un tri-cono de primer nivel $\angle T \in TT_F$ tal que [*P in* $\angle T$].

  Esta propiedad, nos indica que siempre existe un tri-cono perteneciente al tri-tree asociado a un polígono en el que se encuentra un punto. Esto es evidente pues en la propia definición de tri-tree se dice que éste cubre todo el espacio 2D.

- **Propiedad 4.14:** Sea F un polígono y TT_F su tri-tree asociado. Sea $\angle T_i$ un tri-cono tal que $\angle T_i \in TT_F$ y que $\angle T_i$ no es un nodo hoja de TT_F. Sea *P* un punto que cumple que [*P in* $\angle T_i$]. Entonces existe un tri-cono $\angle T_j \in TT_F$ de siguiente nivel e hijo de $\angle T_i$, que cumple que [*P in* $\angle T_j$].

  De esta forma podemos descender por el subárbol de un tri-cono para obtener un tri-cono más pequeño en el que se encuentra el punto. Como un tri-cono se subdivide en dos partes que lo contienen completamente, un punto clasificado en un tri-cono estará siempre incluido en uno de sus sub-tri-conos hijos.

Si un punto se encuentra en la frontera de un tri-cono, podría clasificarse en dos tri-conos de un mismo nivel. En realidad es indiferente el tri-cono en el que se clasifique, pues en cualquiera de ellos estará cubierto por los triángulos del recubrimiento correctos. El criterio utilizado será el de clasificar el punto en el primer tri-cono en el que se detecte su inclusión.

4.6.1.2. Coherencia temporal

A continuación vamos a establecer una heurística que utilizaremos en nuestro algoritmo de detección de colisión, mediante la cual de un frame al siguiente comprobaremos en primer lugar la inclusión del punto en el tri-cono en el que se encontraba en el frame anterior. Evidentemente el punto estará en el mismo tri-cono la mayor parte del tiempo debido a la coherencia temporal, pero en caso de que el punto no esté incluido en dicho tri-cono, debemos localizar nuevamente el tri-cono en el que se encuentra.

Heurística 4.3: Dado un polígono F, su tri-tree asociado TT_F, un tri-cono $\angle T$ tal que $\angle T \in TT_F$ y un punto *P* incluido en dicho tri-cono en un frame determinado, $[P \ in \ \angle T]_j$, entonces en presencia de un alto grado de coherencia temporal lo más probable es que el punto siga estando en el mismo tri-cono en el siguiente frame, es decir $[P \ in \ \angle T]_{j+1}$

Por medio de esta heurística podemos guardar y comprobar entre frames el tri-cono en el que se encontraba el punto, para poder así comenzar la búsqueda en el siguiente frame por el tri-cono almacenado.

Podemos decir que gracias a la coherencia espacial y a la descomposición realizada mediante tri-conos y tri-trees, podemos substituir el término "inclusión en los triángulos del recubrimiento de un polígono" por su equivalente "inclusión en los triángulos del recubrimiento clasificados en un tri-cono" siempre que el punto se encuentre en dicho tri-cono. Así, si se cumple una determinada propiedad en el ámbito de un tri-cono podemos extender la afirmación al polígono completo. Por tanto, podemos aplicar esta afirmación a todas las definiciones dadas en la Sección 4.2.2.

4.6.1.3. Detección de Colisión Punto/Polígono

El nuevo algoritmo de detección de colisión (Algoritmo 4.8) presenta algunas ventajas respecto al algoritmo visto anteriormente que no utilizaba tri-trees (Algoritmo 4.2). En este nuevo algoritmo se clasifica el punto respecto al tri-tree, obteniendo un tri-cono sobre el que realizar la búsqueda. En el ámbito de ese tri-cono se puede realizar la detección de colisión sólo con los triángulos del recubrimiento clasificados en el mismo.

```
static previob_α = 0

bool poligono::testDeteccionColision2DtriTree(punto P) {
   // Inclusión del punto en la circunferencia envolvente del polígono
   if (this.circunferenciaEnvolvente().inclusion(P)==false) return false
   // Obtención del tri-cono en el que se encuentra el punto
   if (triConoAnterior.inclusion(P)==true) tc = triConoAnterior
   else triConoAnterior = tc = this.clasificaTriTree(P)
   // Inclusión del punto en el triángulo envolvente del tri-cono
   if (tc.trianguloEnvolvente().inclusion(P)==false) return false

   // Detección de colisión restringida al tri-cono
   // Cálculo de la máscara de bits alfa
   s_α = s_β = s_γ = b_α = b_β = b_γ = 0
   for (t=0;t<tc.numeroTriangulos();t++) {
      s_α[t] = sign(tc.triangulo(t).α(P))
      b_α[t] = (s_α[t]>=0)
   }
   // Si la mascara de bits es la misma que en la iteración anterior
   if (b_α == previob_α) return EQUAL_STATE
   previob_α = b_α
   // Cálculo de la máscara de bits beta
   for (t=0;t<tc.numeroTriangulos();t++)
      if (b_α[t]==1) {
         s_β[t] = sign(tc.triangulo(t).β(P))
         b_β[t] = (s_β[t]>=0)
      }
   // Cálculo de la máscara de bits gamma
   for (t=0;t<tc.numeroTriangulos();t++)
      if (b_β[t]==1) {
         s_γ[t] = sign(tc.triangulo(t).γ(P))
         b_γ[t] = (s_γ[t]>=0)
      }
   // Inclusión solo donde la mascara de bits gamma = 1
   inclusion = 0
   for (t=0;t<tc.numeroTriangulos();t++) {
      if (b_γ[t]==1) {
         if (tc.triangulo(t).degenerado()==true)
            posicion = tc.triangulo(t).posicionInclusion2D(P) // Contempla caso degenerado
         else
            posicion = posicionDeteccionColision2D(s_α[t], s_β[t], s_γ[t])
         switch (posicion) {
            V_0V_1V_2:            inclusion += 2*sign(tc.triangulo(t))
            V_1, V_2, V_1V_2:     return true
            V_0V_1, V_0V_2:       inclusion += sign(tc.triangulo(t))
            V_0:                  return V_0.inclusion()
         }
      }
   }
   return (inclusion == 2)
}
```

Algoritmo 4.8: Detección de Colisión Punto/Polígono con tri-trees.

Aprovechando la coherencia temporal y espacial, de un frame al siguiente, se comprueba en primer lugar la inclusión del punto en el tri-cono de la iteración anterior. Lo más probable es que el punto se encuentre en dicho tri-cono, con lo que no hay que clasificarlo a través de la estructura del tri-tree. Además se comprueba si dicho punto se encuentra dentro del triángulo envolvente asociado al tri-cono para descartarlo en caso de que no se encuentre dentro del mismo[*].

Los siguientes pasos del algoritmo son los mismos que los dados en la Sección 4.2.5 para el Algoritmo 4.2: Se calcula la máscara de bits correspondiente a la coordenada baricéntrica $\alpha_i(P)$ para todos los triángulos del recubrimiento clasificados en el tri-cono, en lugar de para todos los triángulos del recubrimiento. Esta máscara se compara con la del frame anterior y, si no cambia, se devuelve el estado de inclusión del frame previo. El resto de las coordenadas baricéntricas $\beta_i(P)$ y $\gamma_i(P)$ se calculan sólo cuando son necesarias, y su signo determina finalmente la colisión del punto con el polígono.

4.6.2. Algoritmo de Detección de Colisión Circunferencia/Polígono

Para el caso de detección de colisión entre una circunferencia y un polígono hemos extendido el Algoritmo 4.5 desarrollado en la Sección 4.3 para adaptarlo al uso de tri-trees.

Para obtener este algoritmo ha sido necesario definir nuevos conceptos como los de triángulo extendido y tri-cono extendido. La idea básica en estas definiciones consiste en obtener nuevos triángulos y tri-conos, mayores que los originales, de forma que para comprobar si se produce colisión entre una circunferencia y un polígono pueda utilizarse un algoritmo que compruebe la colisión entre un punto (el centro de la circunferencia) y esos triángulos y tri-conos extendidos, realizando así un test de colisión menos costoso.

[*] Podríamos comprobar previamente la inclusión en la circunferencia envolvente asociada al tri-cono, pero con la mera comprobación de la inclusión en el triángulo envolvente se obtiene un algoritmo más eficiente.

4.6.2.1. Triángulo y Tri-cono extendidos

Utilizaremos el concepto de offset (Sección 4.3.1) para definir la extensión de un triángulo y de un tri-cono (Figura 4.34). Estos nuevos conceptos nos permitirán realizar operaciones entre circunferencias y triángulos o tri-conos, mediante operaciones entre puntos y triángulos o tri-conos extendidos respectivamente.

Definición 4.18: Sea T=$V_0V_1V_2$ un triángulo con sus vértices dados en orden anti-horario (triángulo positivo), definimos la *extensión en r unidades del triángulo* T, como el triángulo delimitado por $offset^+(V_0V_1,r)$, $offset^+(V_1V_2,r)$ y $offset^+(V_2V_0,r)$. En caso de que los vértices del triángulo estén dados en orden horario (triángulo negativo), la *extensión en r unidades del triángulo* T estará delimitada por $offset^-(V_0V_1,r)$, $offset^-(V_1V_2,r)$ y $offset^-(V_2V_0,r)$.

Notación: La extensión en *r* unidades del triángulo T la notaremos como *Ext(*T,*r)*.

Podemos apreciar que un triángulo extendido *r* unidades es otro triángulo en el que cada una de sus aristas dista *r* unidades de la arista correspondiente del primer triángulo, siendo siempre el triángulo extendido mayor que el inicial y conteniendo además totalmente el triángulo extendido al inicial.

Definición 4.19: Dado un triángulo T, definimos la *extensión en r unidades del tri-cono* ∠T, como el tri-cono obtenido a partir del triángulo extendido *Ext(*T,*r)*.

Notación: La extensión en *r* unidades del tri-cono ∠T la notaremos como *Ext(*∠T,*r)*.

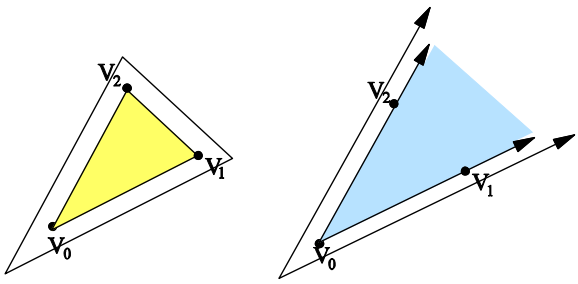


Figura 4.34: Triángulo y tri-cono extendidos.

Para determinar si un punto se encuentra en un triángulo o en un tri-cono extendido podemos utilizar las coordenadas baricéntricas. Como en un tri-tree los tri-conos y triángulos envolventes están asociados a triángulos positivos, sólo necesitaremos caracterizar la inclusión en base a que los triángulos de los que se parte son positivos.

Propiedad 4.15: Sea T=$V_0V_1V_2$ un triángulo positivo y *Ext*(T,*r*) la extensión en *r* unidades del triángulo T. Sean P_0, P_1 y P_2 puntos tales que $P_0 \in offset^+(V_1V_2,r)$, $P_1 \in offset^+(V_2V_0,r)$ y $P_2 \in offset^+(V_0V_1,r)$. Entonces para todo punto *P* se cumple que [*P in Ext*(T,*r*)] sii $\alpha(P) \leq \alpha(P_0) \wedge \beta(P) \leq \beta(P_1) \wedge \gamma(P) \leq \gamma(P_2)$.

Propiedad 4.16: Sea T=$V_0V_1V_2$ un triángulo positivo y *Ext*($\angle$T,*r*) la extensión en *r* unidades del tri-cono $\angle$T. Sean P_1 y P_2 puntos tales que $P_1 \in offset^+(V_2V_0,r)$ y $P_2 \in offset^+(V_0V_1,r)$. Entonces para todo punto *P* se cumple que [*P in Ext*($\angle$T,*r*)] sii $\beta(P) \leq \beta(P_1) \wedge \gamma(P) \leq \gamma(P_2)$.

Los valores de $\alpha(P_0)$, $\beta(P_1)$ y $\gamma(P_2)$ son constantes y pueden precalcularse, por lo que el cálculo de la inclusión de un punto en un triángulo o en un tri-cono extendido no supone un aumento de coste computacional en la detección de colisión.

4.6.2.2. Detección de Colisión Circunferencia/Polígono

La diferencia fundamental de este nuevo algoritmo con el Algoritmo 4.5 para la detección de colisión Circunferencia/Polígono (Sección 4.3.6) radica en la utilización de los tri-trees para obtener un conjunto más pequeño de triángulos sobre los que actuar. En el Algoritmo 4.9 se obtiene un conjunto de tri-conos en los que se encuentra la circunferencia (Figura 4.35), y no un solo tri-cono como ocurre en la detección de colisión punto/polígono. Para ello se utilizan los tri-conos extendidos y se realiza la inclusión del centro de la circunferencia en dichos tri-conos extendidos. El conjunto de tri-conos obtenido se reduce descartando aquellos tri-conos en cuyo triángulo envolvente extendido de tamaño *r* (el radio de la circunferencia) no se encuentra el centro *C* de la circunferencia.

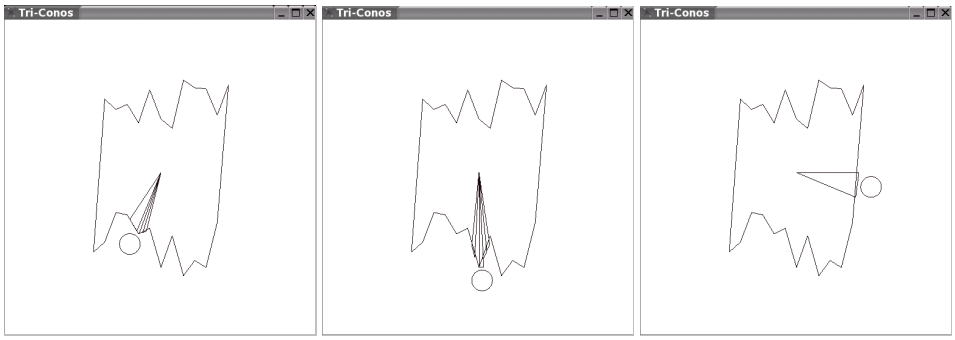


Figura 4.35: Conjunto de tri-conos en los que se encuentra la circunferencia (se muestran los triángulos envolventes de cada tri-cono).

Una vez reducido el conjunto de tri-conos, se realiza un test de colisión circunferencia/arista para los triángulos del recubrimiento clasificados en el conjunto de tri-conos, pero en lugar de utilizar el área de influencia extendida limitada de cada arista (*AIEL*) como se hace en el Algoritmo 4.5 (DC circunferencia/polígono), se utiliza el área de influencia extendida (*AIE*), pues el *AIEL* ya se encuentra limitada en el ámbito del tri-cono y del triángulo envolvente.

```
static resultadoAnterior = false

bool Poligono::testDeteccionColision2DtriTree(Circunferencia c) {
   p = c.centro()
   radio = c.radio()
   // Intersección entre circunferencias envolventes.
   if (c.interseccion(this.circunferenciaEnvolvente())==false) return false

   // Deteccion de colision con el centro de la circunferencia
   resultado = this.testDeteccionColision2D(p)
   if (resultado==EQUAL_STATE && resultadoAnterior==true) return true
   resultadoAnterior = resultado
   if (resultado==true) return true

   // Obtener la lista de TriConos en los que se encuentra la circunferencia
   LTC = this.obtenerListaTriConosExtendidosInclusion(p,radio)

   // Eliminar los TriConos en cuyo Triángulo envolvente no se encuentre la circunferencia
   for (i=0;i<LTC.end();i++) {
      if (LTC(i).trianguloEnvolvente.extension(radio).inclusion(p)==false)
         LTC(i).borrar()
   }

   // Eliminar los triángulos repetidos
   LT = LTC.eliminarTriangulosRepetidos()

   // Cálculo de las aristas con las que puede intersectar solo con la lista de triángulos
   for (i=0;i<LT.numeroTriangulos();i++) {
      if (LT.triangulo(i).AIE(p,radio) == true) {
         if (c.interseccion(LT.triangulo(i).V1V2)==true) return true
      }
   }
   return false
}
```

Algoritmo 4.9: Detección de Colisión Circunferencia/Polígono con tri-trees.

Hemos constatado que no es eficiente guardar el conjunto de tri-conos obtenidos en un frame para el siguiente, y comenzar comprobando la colisión utilizando dichos tri-conos al principio del algoritmo, pues el conjunto de tri-conos cambia considerablemente entre frames cuanto mayor es la circunferencia y mayor es la profundidad del tri-tree. Almacenar los tri-conos de niveles intermedios tampoco resuelve el problema, pues se añade un coste adicional al incrementar el número de triángulos a tratar, lo que hace que disminuya la eficiencia del algoritmo.

4.6.3. Algoritmo de Detección de Colisión Polígono/Polígono

Finalmente presentamos el algoritmo de DC entre polígonos que hace uso de volúmenes envolventes, tri-conos y tri-trees. Para poder utilizar estos conceptos en la detección de colisión entre polígonos ha sido necesario el desarrollo de pequeños algoritmos que resuelven problemas concretos, como la intersección entre segmentos (revisada), la intersección entre triángulos, y la intersección entre un segmento y un polígono; todos estos algoritmos adaptados a la detección de colisiones entre polígonos, como vamos a ver a continuación.

4.6.3.1. Intersección segmento/segmento

Se ha desarrollado un nuevo algoritmo de intersección entre segmentos algo diferente respecto al Algoritmo 4.6 visto en la Sección 4.4.1. En el nuevo algoritmo se han optimizado las operaciones implicadas y se ha contemplado el caso especial en el que los puntos seleccionados para formar un triángulo auxiliar sobre el que calcular las coordenadas baricéntricas estén alineados (Algoritmo 4.10).

```
bool interseccion2D(segmento V1V2, segmento Q1Q2)
     // El triángulo es V1V2V3
     A = Q1 - V2
     B = V1 - V2
     C = Q2 - V2
     w = A · B
     s = C · B
     if ( w > ε ) {
         if ( s > ε ) return false
         t = A · C
         if ( t < -ε ) return false
         if ( w < s + t ) return false
     } else if (w < -ε) {
         if ( s < -ε ) return false
         t = A · C
         if ( t > ε ) return false
         if ( w > s + t ) return false
     } else { // w=0, intercambio de Q1 y Q2
         if ( s > ε ) {
             t = C · A
             if ( t < -ε ) return false
             if ( -s < t ) return false
         } else if ( s < -ε ) {
             t = C · A
             if ( t > ε ) return false
             if ( -s > t ) return false
         } else return false // w=0, s=0, segmentos colineales
     }
     // Cálculos para obtener el punto de intersección si fuese necesario
     // div = 1 / w
     // α = s / div
     // β = t / div
     return true
}
```

Algoritmo 4.10: Intersección Segmento/Segmento (Algoritmo 4.6 revisado).

Basándonos en el Teorema 4.1, hemos optimizado los cálculos implicados como puede verse en el algoritmo. Para ello se utiliza la nomenclatura de la Figura 4.36.

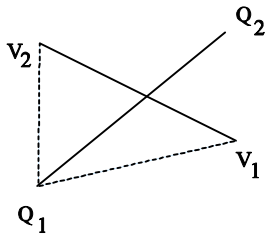


Figura 4.36: Creación de un triángulo auxiliar para el cálculo de intersección entre segmentos en 2D.

Los cálculos basados en las coordenadas baricéntricas de Q_2 respecto al triángulo $Q_1V_1V_2$, necesarios para obtener la intersección entre segmentos son expresados a continuación.

Sean las coordenadas baricéntricas:

$$\alpha = \frac{|Q_2V_1V_2|}{|Q_1V_1V_2|}, \beta = \frac{|Q_1Q_2V_2|}{|Q_1V_1V_2|}, \gamma = \frac{|Q_1V_1Q_2|}{|Q_1V_1V_2|}$$

Podemos calcular el área signada de la siguiente forma:

$$a_0 = |Q_1V_1V_2| = |V_2Q_1V_1| = |(Q_1 - V_2) \quad (V_1 - V_2)|$$
$$a_1 = |Q_2V_1V_2| = |V_2Q_2V_1| = |(Q_2 - V_2) \quad (V_1 - V_2)|$$
$$a_2 = |Q_1Q_2V_2| = -|V_2Q_2Q_1| = -|(Q_2 - V_2) \quad (Q_1 - V_2)|$$
$$a_3 = |Q_1V_1Q_2| = |V_1Q_2Q_1| = |(Q_2 - V_1) \quad (Q_1 - V_1)|$$

Si llamamos: $A = Q_1 - V_2 \quad B = V_1 - V_2 \quad C = Q_2 - V_2 \quad D = Q_2 - V_1 \quad E = Q_1 - V_1$

obtenemos:

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{|A \quad B|} \cdot \begin{bmatrix} |C \quad B| \\ -|C \quad A| \\ |D \quad E| \end{bmatrix} = \frac{1}{(A \cdot B)} \cdot \begin{bmatrix} (C \cdot B) \\ (A \cdot C) \\ (D \cdot E) \end{bmatrix}$$

además, como Q_1 no es colineal con V_1V_2 se cumple que $a_0 \neq 0$ y por tanto:

$$sign(a_i / a_0) = sign(a_i) / sign(a_0) = sign(a_i) \cdot sign(a_0), i = 1..3$$

tenemos:

$$\begin{bmatrix} sign(\alpha) \\ sign(\beta) \\ sign(\gamma) \end{bmatrix} = sign\left[(A \cdot B)\right] \cdot \begin{bmatrix} sign(C \cdot B) \\ sign(A \cdot C) \\ sign(D \cdot E) \end{bmatrix} = sign(w) \cdot \begin{bmatrix} sign(s) \\ sign(t) \\ sign(u) \end{bmatrix}$$

siendo:

$$s = C \cdot B\,,\ t = A \cdot C\,,\ u = D \cdot E \text{ y } w = A \cdot B$$

además, como:

$$\alpha + \beta + \gamma = 1$$

por tanto:

$$\frac{s}{w} + \frac{t}{w} + \frac{u}{w} = 1$$

$$\mathrm{u = w - s - t}$$

por este motivo no es necesario que el algoritmo calcule $D \cdot E$, ahorrándose estos cálculos.

Este algoritmo presenta una serie de ventajas que lo hacen adecuado para el caso de detección de colisión. Entre estas ventajas podemos destacar las siguientes:

- Se contempla el caso de que Q_1 sea colineal con V_1V_2, de manera que el algoritmo devuelve un resultado correcto. Cuando Q_1 es colineal, se intercambian los papeles de Q_1 y Q_2. Si nuevamente Q_2 es colineal con V_1V_2 el algoritmo resuelve que Q_1Q_2 es un segmento colineal con V_1V_2.

- Este algoritmo permite obtener el punto exacto de intersección si fuese necesario, calculando las coordenadas baricéntricas del punto de intersección. Para nuestros fines puede eliminarse este cálculo (que hemos sombreado en el algoritmo), sobre todo debido a que no se realizan operaciones de las que dependan otras previas, y a que dichas operaciones están localizadas al final del algoritmo.

- Es un algoritmo más robusto que otros, en cuanto a que no es necesario realizar ninguna división si lo único que deseamos es comprobar si se produce o no intersección. Además las comparaciones con el valor cero se realizan en un intervalo $[+\varepsilon,-\varepsilon]$.

Al igual que en el algoritmo de detección de colisión entre polígonos (Algoritmo 4.7), en el cual se modificó el algoritmo de intersección entre segmentos para contemplar el caso en el que se conoce a priori que los extremos de un segmento se encuentran cada uno a un lado de otro segmento, hemos modificado el algoritmo de intersección entre segmentos para reducir el número de cálculos. A continuación mostramos este algoritmo modificado (Algoritmo 4.11).

```
bool interseccion2Dmodificado(segmento V1V2, segmento Q1Q2)
    // Este algoritmo asume que Q1 y Q2 están cada uno a un lado del segmento V1V2
     A = Q1 − V2
     B = V1 − V2
     C = Q2 − V2
     w = A · B
     if ( w > ε ) {
         t = A · C
         if ( t < -ε ) return false
         s = C · B
         if ( w < s + t ) return false
     } else if (w < -ε) {
         t = A · C
         if ( t > ε ) return false
         s = C · B
         if ( w > s + t ) return false
     } else return false
     return true
}
```

Algoritmo 4.11: Intersección Segmento/Segmento revisado (Algoritmo 4.10 modificado para la detección de colisión).

4.6.3.2. Intersección triángulo/triángulo

Podemos utilizar el algoritmo de intersección segmento/triángulo para diseñar un algoritmo de intersección triángulo/triángulo que compruebe la intersección de las tres aristas de un triángulo con el otro, pero este algoritmo es claramente ineficiente. Por este motivo vamos a diseñar un algoritmo específico más rápido computacionalmente.

Vamos a emplear un enfoque similar al utilizado en el algoritmo de Cohen-Sutherland [FVFH92] para el recorte de líneas. Utilizaremos para ello las coordenadas baricéntricas de los vértices de un triángulo respecto del otro para determinar los códigos de región en el que se encuentra cada uno estos vértices. Asignamos un código de 3 bits para cada una de las regiones, de manera que el primer bit identifica las regiones que quedan a cada uno de los lados del segmento V_1V_2, es decir, si $sign(\alpha(P))<0$ el bit será igual a 1, y 0 en caso contrario. Este mismo planteamiento es extendido a las coordenadas baricéntricas β y γ del punto respecto al triángulo, correspondiéndose a los bits segundo y tercero respectivamente (Figura 4.37).

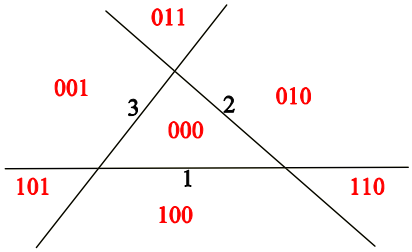


Figura 4.37: Códigos de región para la intersección triángulo/triángulo.

Dados dos triángulos T_1 y T_2=$V_0V_1V_2$, en primer lugar se determina el código de región de los vértices de T_2 respecto a T_1. Así se obtiene la situación de los vértices de V_0, V_1 y V_2 respecto a T_1, tras lo cual se procede como sigue a continuación (Figura 4.38):

- Si los tres vértices V_0,V_1,V_2 tienen el valor de 1 en el mismo bit, los vértices (y el triángulo T_2) están en el lado externo de la recta que determina dicho bit, por lo que no se produce la intersección de los triángulos.

- Si los tres vértices están en la región 000 podemos decir que el triángulo T_2 no interseca con el triángulo T_1, pero que T_2 se encuentra dentro de T_1.

- Si uno o dos vértices están en la región 000 podemos decir que se produce intersección entre triángulos.

- En el resto de los casos se realiza la operación AND bit a bit entre los códigos de región de cada dos vértices. En el caso de que el resultado sea igual a cero para la operación AND del código de región de un par de vértices, se comprueba la intersección del segmento formado por dichos vértices con los segmentos representados por uno de los códigos de región de esos dos vértices. Basta con realizar dicha comprobación con los segmentos asociados al código de región con menor número de unos, que como máximo será de un solo segmento, pues en este caso siempre tendremos un código de región con un único 1. Si se produce alguna intersección de segmento con segmento concluimos que ambos triángulos intersecan.

Si no se ha producido intersección, intercambiamos los papeles de T_1 y T_2 y realizamos el primer test de intersección, pues la única posibilidad restante es que T_1 esté totalmente incluido en T_2, con lo que el algoritmo terminará en el primer paso una vez intercambiados T_1 y T_2.

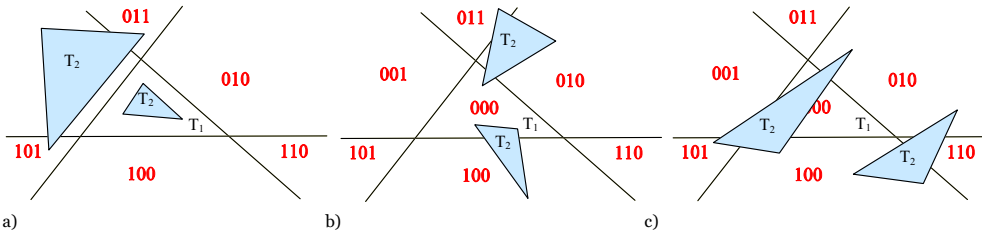


Figura 4.38: Distintas situaciones que pueden darse en la intersección triángulo/triángulo. a) Casos en los que se rechaza trivialmente la intersección. b) Casos en los que se acepta trivialmente la intersección. c) Casos en los que es necesario realizar comprobaciones adicionales.

Este algoritmo es más eficiente y rápido que la simple comprobación de intersección de todas las aristas de un triángulo con todas las del otro, pues se determina la intersección en muchos casos de forma trivial y con menor coste computacional.

A continuación podemos ver el algoritmo de intersección entre triángulos (Algoritmo 4.12). En este algoritmo realizamos la interpretación de que cuando un triángulo se encuentra dentro de otro se produce intersección, pues esta situación debemos contemplarla en el algoritmo de detección de colisión entre polígonos.

```
//Para comprobar la intersección entre T1 y T2 hay que llamar al algoritmo con T1,T2 y
posteriormente con T2,T1.

bool triangulo::interseccion(triangulo T2=V0V1V2) {
   T1 = this
   bits bit[3][3]

   bit0,i = { sign(T1.α(V0))<0, sign(T1.α(V1))<0, sign(T1.α(V2))<0 }
   if (bit0,i == 111) return false
   bit1,i = { sign(T1.β(V0))<0, sign(T1.β(V1))<0, sign(T1.β(V2))<0 }
   if (bit1,i == 111) return false
   bit2,i = { sign(T1.γ(V0))<0, sign(T1.γ(V1))<0, sign(T1.γ(V2))<0 }
   if (bit2,i == 111) return false

   if (bit1,0 == 000 || bit1,1 == 000 || bit1,2 == 000) return true

   if (this.interseccion(bit1,0, bit1,1, V0V1)==true) return true
   if (this.interseccion(bit1,0, bit1,2, V0V2)==true) return true
   if (this.interseccion(bit1,1, bit1,2, V1V2)==true) return true

   return false
}
bool triangulo::interseccion(bits b0, bits b1, segmento Q1Q2) {
   if (b0 & b1 == 000) {
      if (suma(b0)<suma(b1)) b=b0
      else b=b1
      for (i=0;i<3;i++)
         if (b[i]==1) return interseccion(this.segmento(i), Q1Q2)
   }
   return false
}
```

Algoritmo 4.12: Intersección entre triángulos.

4.6.3.3. Detección de Colisión Polígono/Polígono

Hemos visto en la Sección 4.4.3 que podemos comprobar si se produce intersección entre polígonos aplicando el algoritmo de intersección segmento/polígono para todos los segmentos de un polígono F_1 y el polígono F_2.

El algoritmo descrito puede mejorarse cuando se desea realizar una comprobación continua de intersección entre dos polígonos en movimiento o detección de colisión. Para realizar esto, utilizaremos tanto la coherencia temporal como la descomposición espacial dada por los tri-trees y los volúmenes envolventes asociados, como veremos en lo que resta de este capítulo.

A continuación, y basándonos en los algoritmos anteriores, vamos a describir el método utilizado para la detección de colisión entre polígonos. Antes de todo, vamos a establecer una propiedad que nos permite obtener la intersección entre dos polígonos si se da intersección entre dos triángulos envolventes.

Propiedad 4.17: Dados dos polígonos F y G, para que se produzca intersección entre F y G, debe producirse intersección entre un par de triángulos envolventes $T_{env,F} \in F$ y $T_{env,G} \in G$, o debe producirse la inclusión de uno de dichos triángulos envolventes en el otro.

Esto es así debido a que si se produce intersección entre dos polígonos*, debe haber intersección entre dos aristas, una de cada uno de los polígonos. Cada arista está incluida en un triángulo envolvente, por tanto debe producirse intersección entre los triángulos envolventes de ambas aristas o estar incluido un triángulo envolvente dentro del otro.

De esta propiedad podemos deducir que se produce intersección entre polígonos si se produce intersección entre las aristas incluidas en dos triángulos envolventes (Figura 4.39).

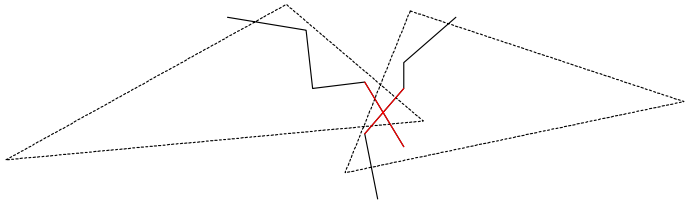


Figura 4.39: Ejemplo de test detallado de colisión entre aristas de dos triángulos envolventes.

El algoritmo de detección de colisión entre polígonos comprueba en primer lugar si se produce colisión entre las circunferencias envolventes de los polígonos, y a continuación se realiza una búsqueda entre las dos jerarquías de volúmenes envolventes asociadas a sus respectivos tri-trees. Dados dos nodos de sendos árboles, que representan a dos tri-conos, sólo si se produce intersección, primero entre sus circunferencias envolventes, y después entre sus triángulos envolventes, se desciende recursivamente a los nodos hijos; en primer lugar del volumen mayor, hasta encontrar dos nodos hoja, uno de cada árbol, en cuyo caso se realiza un test detallado de colisión entre los segmentos incluidos en ambos triángulos envolventes.

* En este caso particular no estamos considerando que uno de los polígonos esté incluido en el otro. Para tenerlo en cuenta podemos comprobar la inclusión de un vértice de cada uno de los polígonos en el otro.

La siguiente heurística define otro de los pasos dados en el algoritmo. Establece el modo de actuar cuando hay cierto grado de coherencia entre frames.

Heurística 4.4: Dados dos polígonos F y G, si en un determinado frame se ha producido intersección entre un par de aristas una de cada polígono, lo más probable es que en el siguiente frame se produzca también intersección entre ese par de aristas. Si no es así, la probabilidad de que se de intersección entre dos aristas pertenecientes a los mismos tri-conos del frame anterior es también muy alta.

Para hacer uso de esta heurística el algoritmo almacena entre frames si se ha producido o no colisión. Si se ha producido colisión se guarda además entre qué aristas, así como los tri-conos en los que se ha producido dicha colisión. En cada llamada al algoritmo se comprueba previamente si se da intersección entre las aristas almacenadas en el frame anterior. Si no se da se comprueba la intersección entre las aristas de los tri-conos almacenados. Debido a la coherencia, cuando tiene lugar una colisión, la probabilidad de que se ocurra entre las mismas aristas es bastante alta, suponiendo cambios pequeños en la posición relativa de los objetos. Es más, si no se produce colisión de nuevo entre este par de aristas, la siguiente colisión, si tiene lugar, es muy probable que ocurra entre aristas incluidas en los tri-conos entre los que hubo colisión en el frame anterior. Si no es así, se procede a buscar recursivamente entre los triángulos envolventes de los tri-trees asociados a cada uno de los polígonos, como describimos a continuación.

El siguiente teorema nos permite descartar rápidamente triángulos envolventes de los dos polígonos utilizando el concepto de triángulo envolvente que forma parte de una envolvente regular.

Teorema 4.2: Sean F y G dos polígonos con tri-trees asociados TT_F y TT_G. Sea $T_{env\text{-}reg}$ un triángulo envolvente que forma parte de una envolvente regular $TT_{env\text{-}reg}$ asociada a un nivel de TT_F. Para todo triángulo envolvente $T_{env}=V_0V_1V_2$ perteneciente a TT_G cuyos vértices V_i, $i=0..2$, cumplan que $sign(\alpha(V_i))<0$, respecto a $T_{env\text{-}reg}$, se cumple que dicho triángulo envolvente T_{env} no interseca con ningún triángulo envolvente de TT_F.

Demostración: Debido a que $T_{env\text{-}reg}$ forma parte de una envolvente regular y convexa del polígono F, todos los puntos del espacio que posean una coordenada α negativa respecto a $T_{env\text{-}reg}$ estarán a un lado de la recta soporte de la arista no original de $T_{env\text{-}reg}$, concretamente en el lado opuesto al que se encuentra el polígono F, pues todos los vértices de F tienen coordenada α positiva respecto a $T_{env\text{-}reg}$. Si los vértices de un triángulo envolvente T_{env} perteneciente al polígono G cumplen esta condición, dicho triángulo estará situado en el lado opuesto respecto al lado en el que

se encuentra F, y por tanto, no se producirá intersección entre ningún triángulo envolvente de TT_F y T_{env}. □

Definición 4.20: Dados dos polígonos F y F' con envolventes regulares $TT_{env\text{-}reg}$ y $TT'_{env\text{-}reg}$ respecto del último nivel de TT_F y $TT_{F'}$ respectivamente, y cuyos centroides son *C* y *C'*; diremos que $T_{env\text{-}reg} \in TT_{env\text{-}reg}$ y $T'_{env\text{-}reg} \in TT'_{env\text{-}reg}$ son *triángulos envolventes de referencia* si [*C in* $\angle T'_{env\text{-}reg}$] y [*C' in* $\angle T_{env\text{-}reg}$].

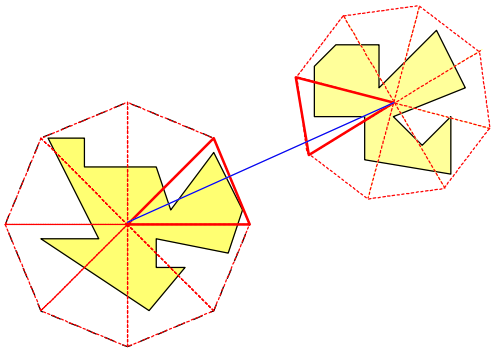


Figura 4.40: Triángulos que forman una envolvente regular (en rojo y trazo discontinuo) y triángulos envolventes de referencia (en rojo y trazo continuo). En azul podemos ver la recta que une el centroide de los polígonos que determina la elección de los triángulos envolventes de referencia.

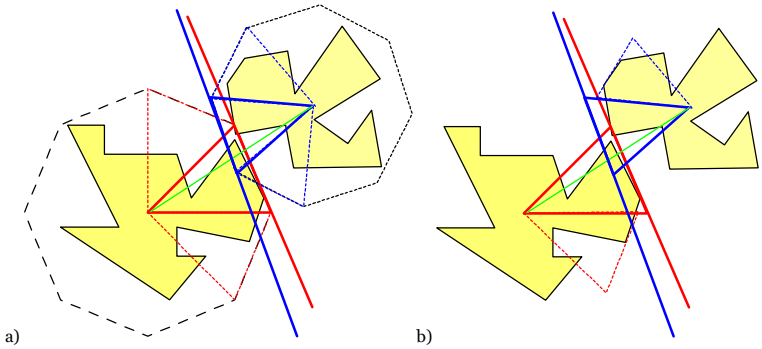


Figura 4.41: Triángulos envolventes tras aplicar el Teorema 4.2 (en color rojo y azul discontinuo) y triángulos envolventes de referencia (en color rojo y azul continuo) a) Se muestran los triángulos envolventes que forman parte de una envolvente regular. b) Se muestran los triángulos envolventes de cada polígono una vez iterado el algoritmo de ajuste.

Podemos acelerar los cálculos necesarios para la detección de colisión entre polígonos descartando algunos triángulos envolventes de forma rápida, sobre todo cuando los objetos no están muy próximos. Para ello aplicamos el Teorema 4.2 a los

triángulos envolventes de referencia de los polígonos (Figura 4.40) y a cada uno de los triángulos envolventes asociados a los tri-trees de los polígonos. De este modo obtenemos un conjunto de triángulos envolventes sobre los que luego comprobar la intersección (Figura 4.41), descartando el resto de triángulos envolventes.

A continuación presentamos un algoritmo recursivo que utiliza los conceptos anteriores para obtener la colisión entre polígonos (Algoritmo 4.13). En las siguientes figuras se muestran los triángulos envolventes involucrados en diversos ejemplos de detección de colisión entre polígonos (Figura 4.42 a 4.45).

```
static segmentoAnterior1, segmentoAnterior2, triConoAnterior1, triConoAnterior2, colisionAnter

bool poligono::testDeteccionColision2DtriTreeRecursivo(poligono P) {
   // Intersección entre circunferencias envolventes
   if (this.circunferenciaEnvolvente().interseccion(P.circunferenciaEnvolvente())==false)
      return false
   // Comprobación de interseccion entre tri-conos o segmentos anteriores
   if (colisionAnter == true) {
      if (interseccion2D(segmentoAnterior1, segmentoAnterior2) == true) return true
      if ((triConoAnterior1.testDeteccionColisionDetallado2D(triConoAnterior2) == true) ||
          (triConoAnterior2.testDeteccionColisionDetallado2D(triConoAnterior1) == true))
              return true
   }

   // Obtener los triángulos de referencia
   trianguloReferencia1 = this.obtenerTrianguloEnvolventeReferencia(P.centroide())
   trianguloReferencia2 = P.obtenerTrianguloEnvolventeReferencia(this.centroide())
   // Marcar los triángulos envolventes
   P.marcarTriangulosEnvolventes(trianguloReferencia1)
   this.marcarTriangulosEnvolventes(trianguloReferencia2)

   // Detección de Colisión recursiva entre tri-conos marcados
   for (triCono1 = 0; triCono1 < 4; triCono1++)
      if (this.hijo(triCono1).marcado()==true)
         for (triCono2 = 0; triCono2 < 4; triCono2++)
            if (P.hijo(triCono2).marcado()==true)
               if (this.hijo(triCono1).testDeteccionColision2D(P.hijo(triCono2))==true)
                  return true
   colisionAnter = false
   return false
}

bool triCono::testDeteccionColision2D(triCono tc) {
   // Intersección entre circunferencias envolventes
   if (this.circunferenciaEnvolvente().intersección(tc.circunferenciaEnvolvente())==false)
      return false

   // Intersección entre triángulos envolventes
   if (this.trianguloEnvolvente().interseccion(tc.trianguloEnvolvente())==false) &&
      (tc.trianguloEnvolvente().interseccion(this.trianguloEnvolvente())==false)
      return false

   // Descender por los subárboles
   if (this.hoja()==true && tc.hoja()==true) {
      if (this.testDeteccionColisionDetallado2D(tc)==true)
         return true
   } else if (this.hoja()==true) {
```

```
      for (triCono = 0; triCono < 2; triCono++)
         if (tc.hijo(triCono).marcado()==true)
            if (tc.hijo(triCono).testDeteccionColision2D(this)==true) return true
   } else if (tc.hoja()==true) {
      for (triCono = 0; triCono < 2; triCono++)
         if (this.hijo(triCono).marcado()==true)
            if (this.hijo(triCono).testDeteccionColision2D(tc)==true) return true
   } else {
      for (triCono1 = 0; triCono1 < 2; triCono1++)
         if (this.hijo(triCono1).marcado()==true)
            for (triCono2 = 0; triCono2 < 2; triCono2++)
               if (tc.hijo(triCono2).marcado()==true)
                  if (this.hijo(triCono1).testDeteccionColision2D(tc.hijo(triCono2))==true)
                       return true
   }
   return false
}
bool triCono::testDeteccionColisionDetallado2D(triCono tc) {
   for (i=0;i<this.numeroTriangulos();i++) {
      if (this.triangulo(i).degenerado()==true) {
         for (j=0;j<tc.numeroVertices();j++)
            if (interseccion2D((tc.vertice(j),tc.vertice(j⊕1),
                        this.triangulo(i).V1,this.triangulo(i).V2) == true) {
                  segmentoAnterior1 = (tc.vertice(j), tc.vertice(j⊕1))
                  segmentoAnterior2 = this.triangulo(i).V1V2
                  triConoAnterior1 = this; triConoAnterior2 = tc
                  colisionAnter = true
                  return true
            }
      } else {
         s = sign(this.triangulo(i).α(tc.vertice(0)))
         for (j=0;j<tc.numeroVertices();j++) {
            s' = sign(this.triangulo(i).α(tc.vertice(j⊕1)))
            if (s!=s') if (interseccion2Dmodificado(tc.vertice(j),tc.vertice(j⊕1),
                        this.triangulo(i).V1,this.triangulo(i).V2) == true) {
                  segmentoAnterior1 = (tc.vertice(j), tc.vertice(j⊕1))
                  segmentoAnterior2 = this.triangulo(i).V1V2
                  triConoAnterior1 = this; triConoAnterior2 = tc
                  colisionAnter = true
                  return true
            }
            s = s'
         }
      }
   }
   return false
}
void poligono::marcarTriangulosEnvolventes(triangulo T) {
   for (vert = 1; vert < 2; vert++)
      if (T.α(this.TrianguloEnvolvente().vertice(vert))>=0) {
         this.marcado = true
         for (triCono = 0; triCono < 2; triCono++)
            this.hijo(triCono).marcarTriangulosEnvolventes(T)
         break
      }
}
```

Algoritmo 4.13: Detección de Colisión Polígono/Polígono usando una intersección recursiva entre tri-trees.

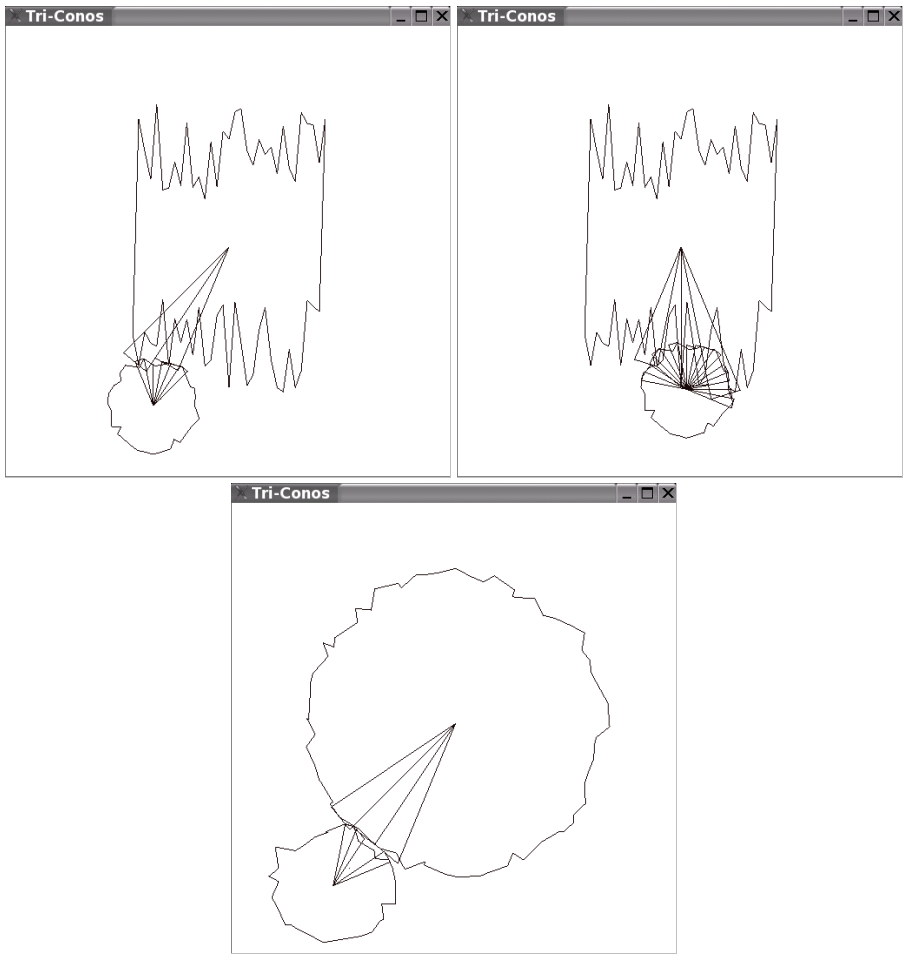


Figura 4.42: Totalidad de pares de triángulos envolventes que colisionan. El esquema de detección de colisión utilizado se detiene cuando se encuentra la primera colisión entre un par de aristas de los polígonos. Para obtener todas las características involucradas en la colisión deben comprobarse todos los pares de triángulos envolventes mostrados.

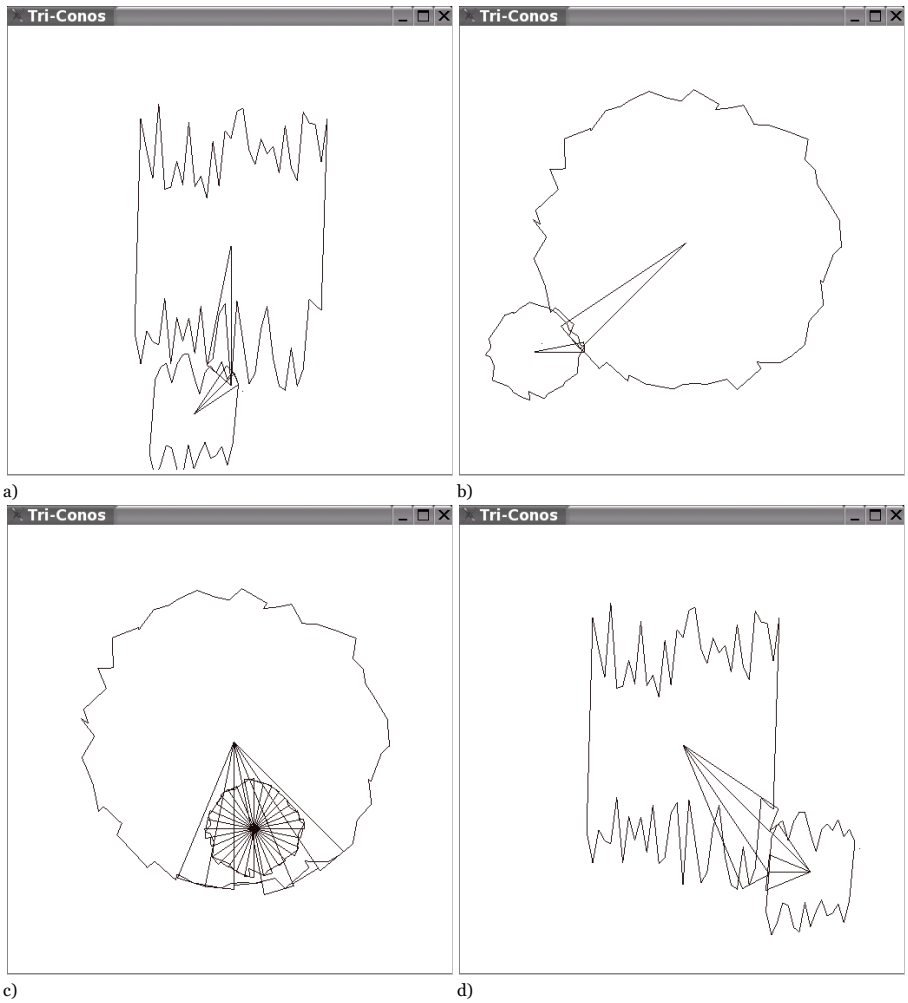


Figura 4.43: Triángulos envolventes involucrados en la detección de colisión. En las dos primeras figuras se produce colisión entre polígonos y se muestran los triángulos envolventes en cuyo interior se detecta la primera colisión. En la tercera figura, si el algoritmo no realiza internamente una detección de colisión punto/polígono, haría falta comprobar todos los pares de triángulos envolventes mostrados. En la cuarta figura no se produce colisión.

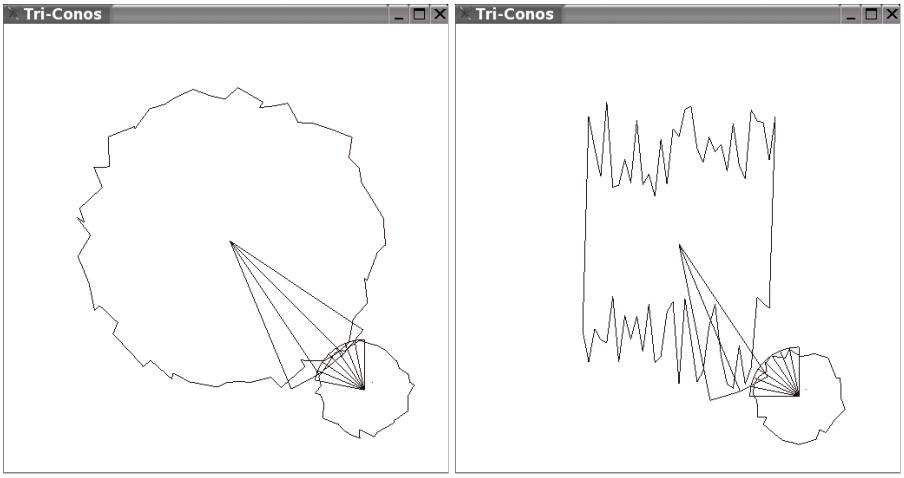


Figura 4.44: Triángulos envolventes que quedan tras aplicar el Teorema 4.2, el resto son descartados al principio del algoritmo.

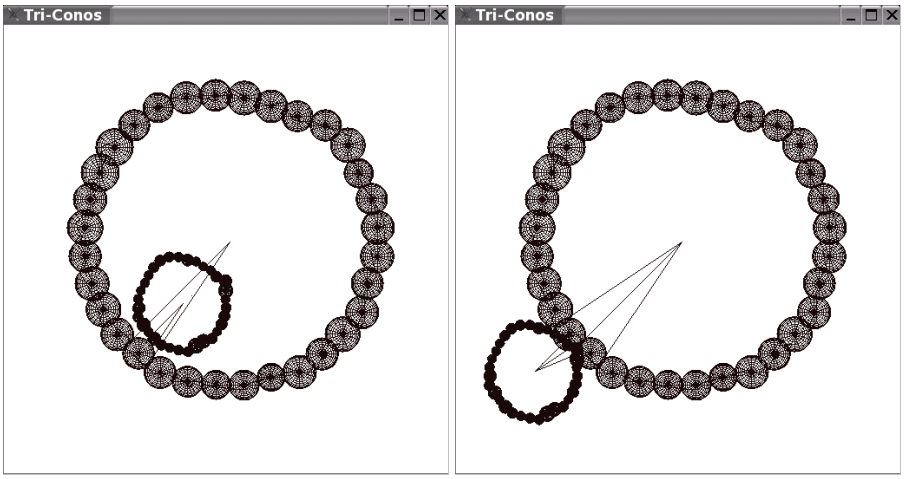


Figura 4.45: Circunferencias envolventes de último nivel y triángulos envolventes utilizados en la detección de colisión.

El algoritmo anterior es válido para cualquier pareja de polígonos. Sin embargo, podemos utilizar el algoritmo de detección de colisión circunferencia/polígono anterior (Algoritmo 4.9) y aplicarlo a la detección de colisión entre polígonos como hicimos en el Algoritmo 4.7, pero en este caso utilizando la descomposición mediante tri-trees. El algoritmo aquí expuesto es más eficiente cuando la relación entre el tamaño de las circunferencias envolventes de los polígonos es grande, es decir, cuando hay una diferencia significativa entre el tamaño de los polígonos. En esta situación el algoritmo es muy apropiado, pues su tiempo de ejecución es similar al del algoritmo de detección de colisión punto/polígono. A continuación mostramos este algoritmo (Algoritmo 4.14).

Al igual que el algoritmo en el que se basa, debemos utilizar un punto de referencia que se encuentre dentro del polígono sustituido por la circunferencia envolvente para que la detección de colisión punto/polígono incluida en el algoritmo funcione correctamente. Si el centro de la circunferencia envolvente no es interior al polígono puede usarse cualquier otro punto, como por ejemplo un vértice.

```
static resultadoAnterior = false

bool poligono::testDeteccionColision2DtriTree(poligono f) {
   p = f.circunferenciaEnvolvente().centro()
   radio = f.circunferenciaEnvolvente().radio()
   // Intersección entre circunferencias envolventes.
   if (c.interseccion(this.circunferenciaEnvolvente())==false) return false

   // DC con el centro de la circunferencia o un punto que pertenezca al polígono
   resultado = this.testDeteccionColision2D(p)
   if (resultado==EQUAL_STATE && resultadoAnterior==true) return true
   resultadoAnterior = resultado
   if (resultado==true) return true

   // Obtener la lista de TriConos en los que se encuentra la circunferencia
   LTC = this.obtenerListaTriConosExtendidosInclusion(p,radio)

   // Eliminar los TriConos en cuyo Triángulo envolvente no se encuentre la circunferencia
   for (i=0;i<LTC.end();i++) {
      if (LTC(i).trianguloEnvolvente.extension(radio).inclusion(p)==false)
         LTC(i).borrar()
   }
   // Eliminar los triángulos repetidos
   LT = LTC.eliminarTriangulosRepetidos()

   // Cálculo de las aristas con las que puede intersectar solo con la lista de triángulos
   for (i=0;i<LT.numeroTriangulos();i++) {
      if (LT.triangulo(i).AIE(p,radio) == true) {
         if (LT.triangulo(i).interseccionV1V2(f)==true) return true
      }
   }
   return false
}
```

Algoritmo 4.14: Detección de Colisión Polígono/Polígono mediante tri-trees no recursiva.

4.7. Algunas Consideraciones

Todos los algoritmos desarrollados pueden obtener las características implicadas en la colisión. Por ejemplo para la detección de colisión entre polígonos, cuando se produce intersección entre un par de aristas, en el caso de querer conocer todas las aristas implicadas, el algoritmo no debe parar y retornar indicando que se ha producido colisión con el primer par de aristas detectadas, sino continuar y almacenar los pares de características que intersecan. De igual forma, el algoritmo debe localizar todos los pares de triángulos envolventes que colisionan. Evidentemente, estos algoritmos son más costosos que los aquí expuestos.

También es posible obtener la detección de colisión entre objetos con una determinada tolerancia, por ejemplo, obtener colisión cuando un objeto se encuentra a una distancia menor que una distancia dada. Para ello simplemente hay que utilizar los offsets de los triángulos para determinar en que parte se encuentra un objeto respecto a un triángulo del recubrimiento. Igualmente será necesario utilizar offsets para los triángulos envolventes y para los tri-conos.

En cuanto a la eficiencia de los algoritmos, según veremos en el estudio temporal realizado en la siguiente sección, podemos considerar que son más eficientes que otros, en primer lugar al utilizar una descomposición espacial que ocupa menos memoria que otras y que es bastante rápida de calcular, pues se ajusta y adapta mejor a los objetos. En segundo lugar, gracias al aprovechamiento de la coherencia temporal y espacial, pues se utiliza la coherencia que pueda haber en sistemas interactivos dentro de sus posibilidades.

Si consideramos los aspectos de robustez de estos algoritmos, podemos decir que presentan un alto grado de robustez geométrica, puesto que son algoritmos válidos para muchos tipos de objetos. En otros algoritmos se pueden presentar problemas, como por ejemplo, su validez para objetos cóncavos, no-variedad, con agujeros, etc. Además estos algoritmos tienen un alto grado de robustez numérica en el sentido de que la mayor parte de las operaciones se realizan mediante sumas signadas con aritmética entera, y que cuando es necesario obtener exclusivamente el signo de una coordenada baricéntrica, éste no se calcula realizando la división que lleva implícita, estando el cálculo del signo limitado exclusivamente por la precisión en la evaluación de los determinantes [ABDPY97]. El signo es obtenido con una determinada tolerancia [GSS89], de manera que se considera cero cuando está en un intervalo $[+\varepsilon,-\varepsilon]$. Utilizar recubrimientos simpliciales y coordenadas baricéntricas, simplifica en gran medida los casos especiales, que no obstante, existen y se han considerado adecuadamente en los algoritmos.

4.8. Estudio de Tiempos

En esta sección estudiaremos el comportamiento de los algoritmos en distintas situaciones y para distintos tipos de polígonos complejos. Se ha realizado una implementación de los algoritmos descritos a lo largo de este capítulo de manera que podamos obtener los tiempos efectivos en los que se desarrolla la detección de colisión. Para realizar dicha implementación se ha seguido una filosofía orientada a objetos. Utilizaremos como medida en este estudio el número de frames por segundo, es decir el número de veces que se ejecuta el algoritmo de colisión por unidad de tiempo.

Se ha utilizado un procesador Intel Pentium IV a 1,6 GHz, con un 1GB de memoria RAM. La implementación se ha realizado en C++ con OpenGL en un sistema operativo Linux.

En este estudio obtendremos el comportamiento de los algoritmos para determinadas circunstancias como puede ser el grado de coherencia geométrica del polígono, la relación entre el tamaño de los polígonos o el tipo de trayectoria que sigue el movimiento de un objeto en relación a otro.

Por otra parte, compararemos los algoritmos que utilizan tri-trees con las versiones de los mismos que no hacen uso de esta descomposición espacial. Para el caso de DC entre puntos y polígonos compararemos además estos algoritmos con el algoritmo de inclusión crossing-count utilizando quadtrees según puede verse en [Hai94] y [SE03].

Hemos optado por describir en primer lugar las pruebas realizadas (secciones 4.8.2 a 4.8.4) para después obtener las conclusiones en relación a los tiempos obtenidos (sección 4.8.5). En la sección 4.8.6 se muestran todas las tablas y gráficos con los tiempos obtenidos. En cada una de las pruebas mostramos al final de la correspondiente sección una tabla resumen de los resultados.

En la siguiente sección obtendremos el tiempo efectivo de construcción de un tri-tree, tras lo cual mostraremos las pruebas y tiempos obtenidos para la DC entre diversos tipos de objetos.

4.8.1. Pruebas y Tiempos de Pre-procesamiento

Hemos medido el tiempo que se tarda en construir el tri-tree de un polígono complejo en función del número de subdivisiones espaciales realizadas. El número de subdivisiones espaciales depende de factores como la profundidad del árbol o el

número de niveles del tri-tree. Hemos comparado el tiempo de construcción del tri-tree con el tiempo de construcción de un quadtree. Se ha utilizado un quadtree debido sobre todo a que es una de las estructuras de datos adaptativa más utilizada en estos casos. Como cada nivel del árbol no tiene el mismo número de nodos en un quadtree y en un tri-tree, en las pruebas compararemos niveles con el mismo número de subdivisiones espaciales; por ejemplo el nivel 1 tiene 4 nodos en un tri-tree y en un quadtree, el nivel 3 de un tri-tree y el nivel 2 de un quadtree tienen 16 nodos, etc.

Debemos tener en cuenta que esto supone una penalización en el tiempo de construcción de un tri-tree respecto al de construcción de un quadtree, pues para el mismo número de subdivisiones espaciales el número de niveles del tri-tree es aproximadamente el doble que en el caso de un quadtree. Esto supone una penalización tanto en la construcción del tri-tree como en la obtención de un nodo del tri-tree; es necesario, en el primer caso clasificar las características aproximadamente el doble de veces, y en el segundo caso recorrer el doble número de nodos para obtener un tri-cono en el que se encuentre, por ejemplo, un punto.

Se ha utilizado un polígono complejo para mostrar el tiempo de construcción (Figura 4.46). Este tipo de polígono supone también una penalización para el algoritmo desarrollado pues la coherencia geométrica del polígono es muy pequeña. De este modo, al clasificar los triángulos del recubrimiento de este polígono en un tri-tree, muchos de ellos aparecen duplicados en más de un tri-cono. Aumentar la profundidad del tri-tree en estos casos puede suponer un inconveniente pues hay que descender hasta un nodo hoja en el árbol, obteniendo aproximadamente el mismo número de triángulos clasificados en ese nodo que en niveles superiores de la jerarquía.

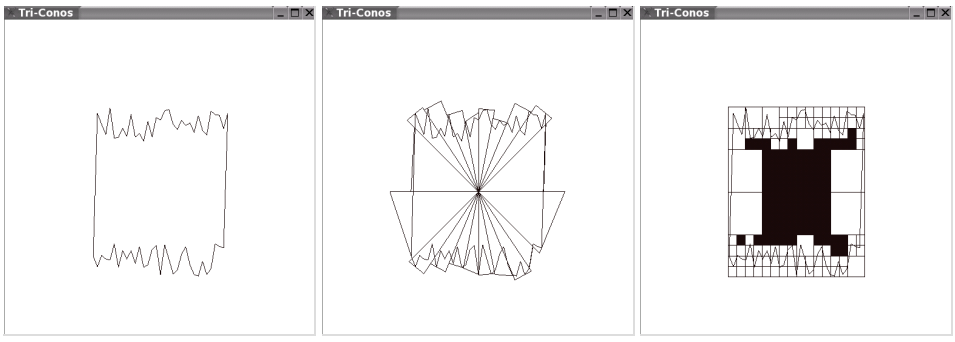


Figura 4.46: a) Polígono complejo. b) Polígono con descomposición espacial mediante tri-tree. c) Polígono con descomposición espacial mediante quadtree.

Para otros tipos de polígonos, se obtienen tiempos bastante mejores. Hemos preferido mostrar la construcción del tri-tree sólo para el peor caso[*], para compararlo con el tiempo de construcción de un quadtree.

En la Figura 4.47 podemos ver los tiempos de construcción de un tri-tree y un quadtree para un polígono complejo formado por 1.000 vértices y por 10.000 vértices respectivamente. En ambos casos no se ha iterado el algoritmo de ajuste de los triángulos envolventes del tri-tree, por lo que sus aristas originales tienen una longitud igual a la del vértice más distante del vértice original a cada uno de los vértices de los triángulos clasificados en su tri-cono. El tiempo se ha expresado en segundos y se utiliza una escala logarítmica.

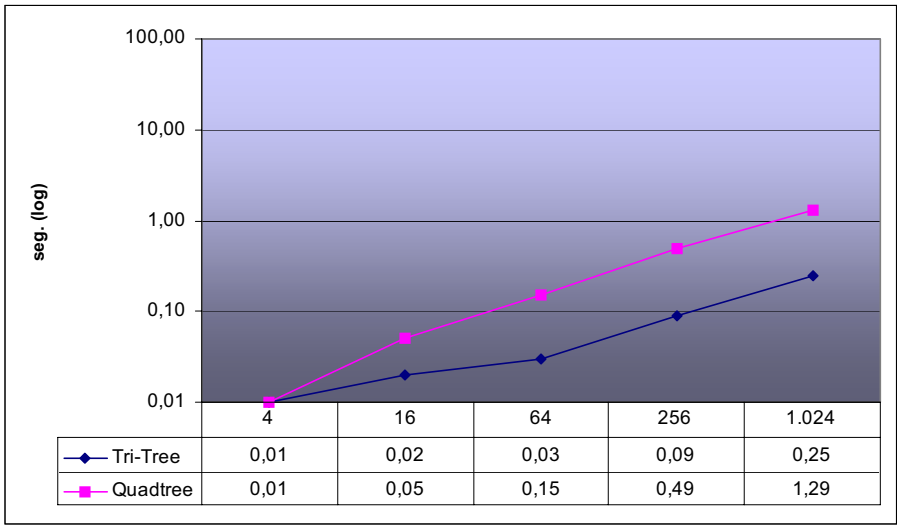
Podemos ver que en ambos casos (1.000 y 10.000 vértices) el tiempo de construcción de un tri-tree es siempre menor que el tiempo de construcción de un quadtree, siendo los tiempos más similares entre sí cuando el número de subdivisiones espaciales realizadas es bajo. Cuando el número de subdivisiones crece, la construcción de un tri-tree es mucho más rápida, pues podemos ver que la curva tiene una pendiente menor que el caso de un quadtree.

En la Figura 4.48 se muestra el tiempo obtenido al iterar el algoritmo de ajuste de los triángulos envolventes del tri-tree para el caso de un polígono formado por 1.000 vértices. Se utiliza una escala logarítmica y se ha comparado también con el algoritmo de construcción del quadtree.

En este caso se aprecia una importante penalización en el algoritmo de construcción del tri-tree al incrementar las iteraciones del algoritmo de ajuste, pero si lo comparamos con el tiempo de construcción del quadtree, podemos ver que la pendiente de la curva es menor en todos los casos. Si extrapolamos el gráfico para un gran número de subdivisiones, los tiempos obtenidos pueden ser mejores en los casos de 8 y 16 iteraciones respecto a los tiempos de construcción del quadtree. Debemos tener en cuenta que el número apropiado de iteraciones del algoritmo para cada polígono depende de factores como la coherencia geométrica del polígono y del número de subdivisiones del mismo. Es necesario un menor número de iteraciones cuanto mayor sea el número de subdivisiones realizadas y mayor sea por tanto la coherencia geométrica. La coherencia geométrica es mayor en el ámbito de un tri-cono pequeño que uno grande. Según las pruebas realizadas, para este tipo de polígonos el rango de iteraciones adecuado está comprendido entre 0 y 4 iteraciones (depende de la profundidad del árbol).

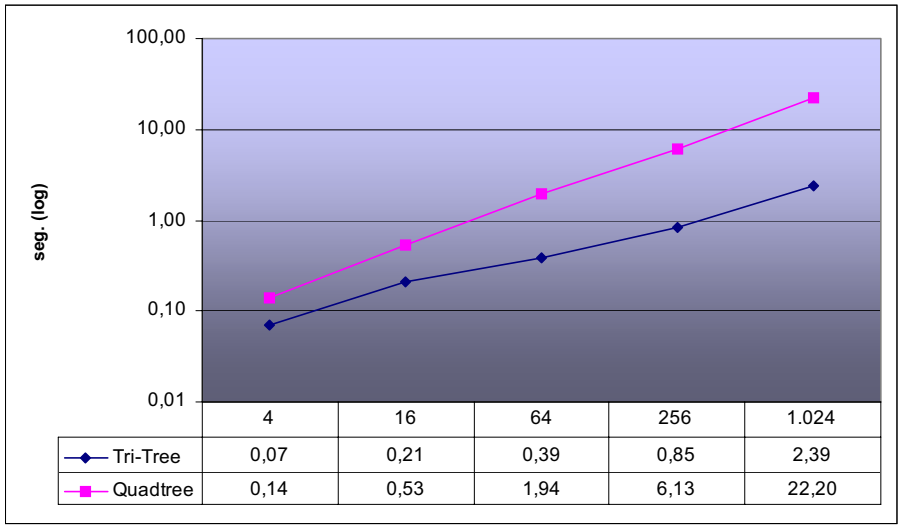
[*] Consideramos que el tipo de figura utilizado puede ser uno de los peores casos, debido a la irregularidad del objeto y a la proximidad de las aristas entre si.

a)



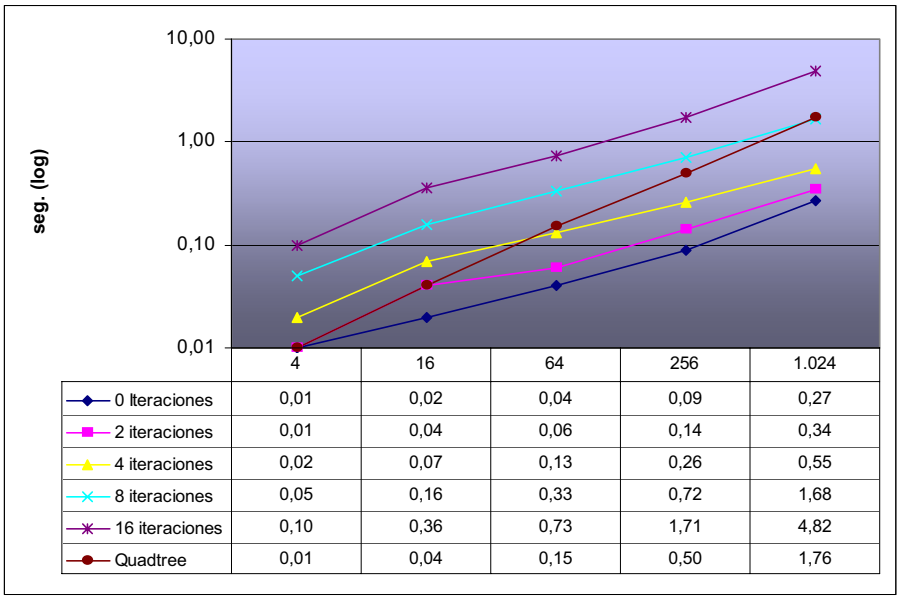
	4	16	64	256	1.024
Tri-Tree	0,01	0,02	0,03	0,09	0,25
Quadtree	0,01	0,05	0,15	0,49	1,29

b)



	4	16	64	256	1.024
Tri-Tree	0,07	0,21	0,39	0,85	2,39
Quadtree	0,14	0,53	1,94	6,13	22,20

Figura 4.47: Tiempo de construcción de un tri-tree y un quadtree para un polígono complejo. En el eje X se muestra el número de subdivisiones realizadas en el árbol correspondiente y en el eje Y el tiempo en segundos con escala logarítmica. a) Polígono de 1.000 vértices. b) Polígono de 10.000 vértices.



	4	16	64	256	1.024
0 Iteraciones	0,01	0,02	0,04	0,09	0,27
2 iteraciones	0,01	0,04	0,06	0,14	0,34
4 iteraciones	0,02	0,07	0,13	0,26	0,55
8 iteraciones	0,05	0,16	0,33	0,72	1,68
16 iteraciones	0,10	0,36	0,73	1,71	4,82
Quadtree	0,01	0,04	0,15	0,50	1,76

Figura 4.48: Tiempo de construcción de un tri-tree para un determinado número de iteraciones del algoritmo de ajuste de triángulos envolventes y construcción de un quadtree, ambos para un polígono complejo formado por 1.000 vértices. En el eje X se muestra el número de subdivisiones realizadas en el árbol correspondiente y en el eje Y el tiempo en segundos con escala logarítmica.

4.8.2. Pruebas para la Detección de Colisión Punto/Polígono

En esta sección mostramos las pruebas realizadas para la DC entre un punto y un polígono cuando el punto se mueve en una trayectoria definida en relación al polígono. Se han utilizado dos tipos de trayectorias, una circular y otra lineal, ambas cercanas al polígono.

La trayectoria circular (Figura 4.49.a) se ha generado en torno al polígono y muy próxima al mismo de manera que el punto vuelva al origen. La trayectoria está formada por 90.000 posiciones con una pequeña variación aleatoria en los ejes X e Y respecto a la circunferencia. En dicha trayectoria se producen colisiones en menos del 5% de los casos.

La trayectoria lineal (Figura 4.49.b) se ha generado mediante un desplazamiento en X de 9.000 posiciones más una pequeña variación aleatoria en el eje Y respecto a la horizontal. Esta trayectoria está también próxima al polígono, de manera que se producen colisiones entre un 10% y un 20% de los casos.

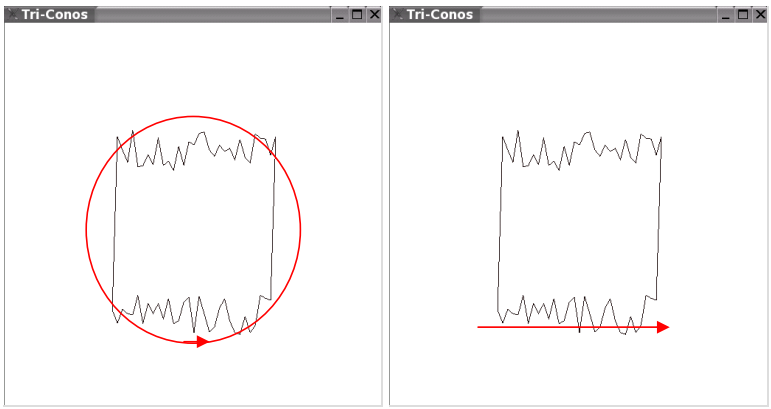


Figura 4.49: Trayectorias utilizadas. a) Circular. b) Lineal.

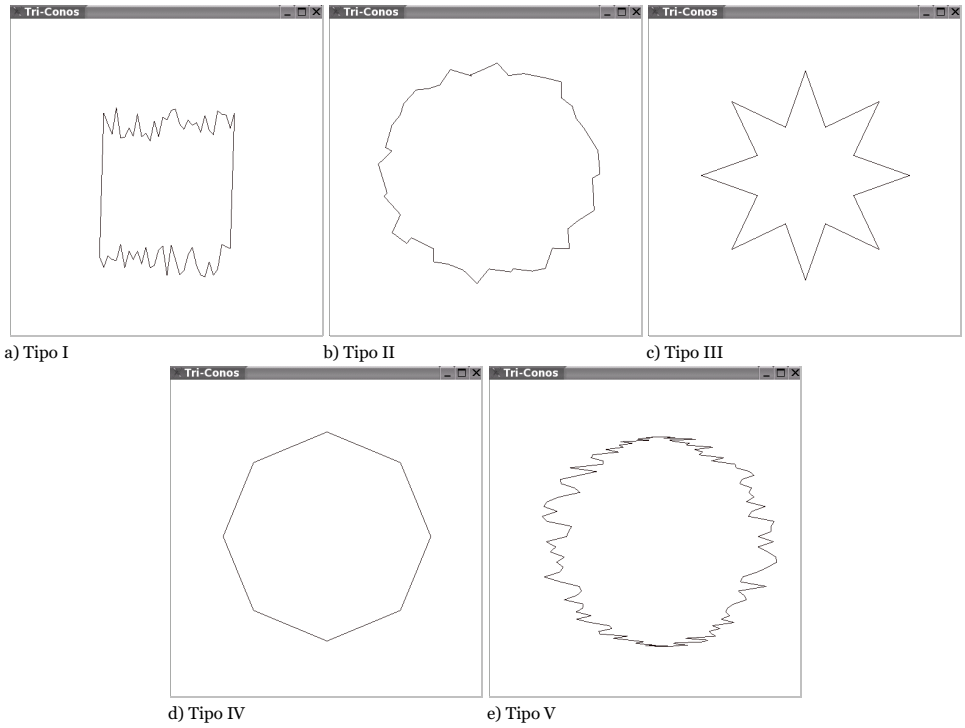


Figura 4.50: Polígonos utilizados en la DC. a) Tipo I: No convexo irregular. b) Tipo II: No convexo regular. c) Tipo III: Con recubrimiento positivo. d) Tipo IV: Convexo. e) Tipo V: No convexo regular.

En el caso de la trayectoria circular, su densidad, es decir el número de posiciones por unidad de área, es mayor que en el caso de la trayectoria lineal. Esta característica puede apreciarse en los tiempos obtenidos por los algoritmos, pues a mayor densidad mayor coherencia geométrica y mayor velocidad en la detección de colisión.

Se han utilizado distintos tipos de polígonos: no convexos irregulares, no convexos regulares, con recubrimiento positivo y convexos (ver Figura 4.50), y se han utilizando los algoritmos apropiados para cada caso. Para cada polígono se ha variado el número de vértices y se ha ajustado tanto el número de subdivisiones (profundidad del tri-tree), como el número de iteraciones del algoritmo de ajuste de triángulos envolventes. Estos parámetros se han fijado a valores adecuados para cada caso (el número de subdivisiones se ha igualado al número de vértices y el número de iteraciones se ha igualado a ocho). Podemos ver un resumen de las pruebas realizadas en la Tabla 4.1.

Esta tabla, así como las siguientes, está dividida en tres partes:

- **Datos de Entrada**: Son los diversos parámetros de los algoritmos considerados, como el tipo de polígono, el rango de vértices para cada polígono, número de subdivisiones espaciales realizadas en la descomposición mediante tri-tree, número de iteraciones del algoritmo de ajuste de triángulos envolventes, relación entre el tamaño de los objetos considerados, tipo de trayectoria y algoritmos considerados.

  La aparición de más de un elemento en una columna indica que se han realizado pruebas con cada uno de ellos. Por ejemplo, en la Tabla 4.1 se han realizado pruebas para los cuatro tipos de polígonos, cada uno con vértices comprendidos entre 16 y 1.024, y cada uno de ellos con número de subdivisiones comprendido entre 16 y 1.024, para dos tipos de trayectoria: circular y lineal. Todas estas pruebas se han realizado para los algoritmos DC, DC-TT y Quadtree para un valor fijo del número de iteraciones del algoritmo de ajuste (8) y una relación entre tamaños (1:1), parámetro que para esta prueba no es significativo y que lo será para la DC de circunferencias y polígonos en relación a polígonos.

- **Leyenda**: Resume la leyenda utilizada en los correspondientes gráficos. "Eje X" y "Eje Y" indican el parámetro asociado a cada eje en el gráfico (normalmente en el "Eje Y" se muestra el número de frames por segundo). En la columna "Gráficos" se indica el significado de cada tabla junto con su gráfico cuando la figura se ha subdividido en diversas secciones (a, b, c, d, etc.). Por último en la columna "Comportamiento Analizado" se resume la prueba realizada.

- **Resultados**: Indica los resultados obtenidos en función de los parámetros de entrada. Cuando un parámetro no es significativo no se muestra. El resultado puede ser la posición de cada algoritmo en las pruebas realizadas (1º el mejor, 2º, 3º, etc.) o una descripción verbal de dicho resultado. En la columna "Figura" mostramos una referencia a la figura que tiene los tiempos medidos.

Datos de Entrada						
Tipo Polígono	Vértices	Subdivisiones	Iteraciones A.A.	Relación Tamaño	Trayectoria	Algoritmos
	16 64 256 1.024	16 64 256 1.024	8	1:1	Circular Lineal	DC (Alg.4.2, 4.3 y 4.4) DC-TT (Alg. 4.8) QuadTree [Hai94]
Leyenda						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Nº Subdivisiones	(Frames/seg.) escala log.	a) Trayectoria circular b) Trayectoria lineal		DC en función del nº de subdivisiones, tipo de polígono y trayectoria		
Resultados						
Tipo Polígono	Trayectoria	Pos. Algoritmo		Figura		
	Circular y Lineal	1º DC-TT 2º Quadtree 3º DC (Alg. 4.2)		4.51.a-b		
	Circular	1º DC-TT 2º DC (Alg. 4.2) 3º Quadtree		4.52.a		
	Lineal	1º DC-TT 2º Quadtree 3º DC (Alg. 4.2)		4.52.b		
	Circular y Lineal	1º DC (Alg. 4.4) 2º DC-TT 3º Quadtree		4.53.a-b		
	Circular y Lineal	1º DC (Alg. 4.3) 2º DC-TT 3º Quadtree		4.54.a-b		

Tabla 4.1: Resumen de resultados para las pruebas de DC punto/polígono.

4.8.3. Pruebas para la Detección de Colisión Circunferencia/Polígono

En este caso se han realizado pruebas en las que se obtiene el tiempo de detección de colisión para una trayectoria circular (en frames por segundo), similar a la generada para la DC punto/polígono, pero esta vez depende de la relación entre el tamaño relativo del polígono y el de la circunferencia; lo que llamaremos escala relativa o relación de tamaño. Se ha utilizado para las pruebas un polígono no convexo irregular (Tipo I) con diferente número de vértices. El número de subdivisiones

realizadas mediante tri-trees se ha ajustado a un valor constante que depende del número de vértices y del tamaño relativo entre objetos. Por ejemplo para 1.024 vértices y un tamaño de 1:1 se ha utilizado un valor, mayor que para 64 vértices y tamaño 1:1, y menor que para 1.024 vértices y un tamaño relativo de 1:8.

El tamaño relativo entre objetos se ha medido utilizando el radio de la circunferencia envolvente del polígono en relación al de la circunferencia. Se han utilizado tamaños de 1:1, 1:2, 1:4, 1:8, 1:16 y 1:32, que indican la relación entre la circunferencia y el polígono, en el caso 1:1 tienen igual tamaño, en el caso 1:2 el radio de la circunferencia es la mitad del radio de la circunferencia envolvente del polígono.

El número de subdivisiones realizadas para cada polígono se ha establecido constante pero en función de la escala relativa y del número de vértices del polígono.

Para medir los tiempos se han utilizado los algoritmos de DC con tri-trees (DC-TT) y sin tri-trees (DC). En la Tabla 4.2 podemos ver resumidas las características de estas pruebas.

Datos de Entrada						
Tipo Polígono	Vértices	Subdivisiones	Iteraciones A.A.	Relación Tamaño	Trayectoria	Algoritmos
	16 64 256 1.024	Constante: Ajustado al nº de vértices y relación tamaño	8	1:1 1:2 1:4 1:8 1:16 1:32	Circular Lineal	DC (Alg.4.5) DC-TT (Alg. 4.9)
Leyenda Figura 4.55						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Relación Tamaño	(Frames/seg.) escala log.	a) 16 vértices b) 64 vértices c) 256 vértices d) 1.024 vértices		DC en función de la relación entre el tamaño de los objetos, el nº de vértices y el tipo de trayectoria.		
Resultados Figura 4.55						
Vértices	Trayectoria	Pos. Algoritmo		Figura		
16	Circular	1º DC 2º DC-TT		4.55		
64, 256, 1.024	Circular	1º DC-TT 2º DC		4.55		
16, 64, 256, 1.024	Lineal	Sin diferencias significativas		-		
Leyenda Figura 4.56						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Vértices	(Frames/seg.) escala log.	a) Algoritmo DC b) Algoritmo DC-TT		DC en función de la relación entre el tamaño de los objetos y el nº de vértices.		
Resultados Figura 4.56						
Algoritmo	Relación Tamaño	Pos. Algoritmo		Figura		
DC	1:1, 1:2, 1:4, 1:8, 1:16, 1:32	Resultados similares		4.56.a		
DC-TT	1:1, 1:2, 1:4, 1:8, 1:16, 1:32	A mayor diferencia en la relación entre tamaños se obtienen mejores tiempos		4.56.b		

Tabla 4.2: Resumen de resultados para las pruebas de DC circunferencia/polígono.

4.8.4. Pruebas para la Detección de Colisión Polígono/Polígono

A la hora de desarrollar un conjunto de tests para la detección de colisión entre polígonos nos encontramos por una parte con la variedad de polígonos, número de vértices de los mismos y escala relativa entre ambos; y por otra, el conjunto de algoritmos desarrollado. Además tenemos una alta variedad de parámetros de los que depende la DC, como la profundidad en el tri-tree, número mínimo de características clasificadas en un tri-cono, número de iteraciones del algoritmo de ajuste de triángulos envolventes, tipo de trayectoria, etc.

Podemos establecer un conjunto de pruebas sistemáticas que midan el tiempo de DC para todos estos parámetros y características, pero esto haría interminable el número de tablas de datos y gráficos. Hemos adoptado una estrategia que consiste en analizar comportamientos, por lo que nos basamos en los resultados de pruebas anteriores, como la DC entre una circunferencia y un polígono, o la DC entre un punto y un polígono, y en los parámetros analizados en dichas pruebas. Así obtenemos un subconjunto de pruebas adecuado para cada situación particular, sobre todo cuando deseamos obtener el comportamiento obtenido en la detección de colisión de situaciones reales y habituales.

Por ejemplo, es interesante conocer el comportamiento del algoritmo de DC en todas sus variantes para el caso de trayectorias circulares y lineales con un pequeño movimiento aleatorio que varíe un poco respecto a la trayectoria inicial. Éste es un comportamiento normal cuando se desea detectar la colisión entre objetos, pues o bien el movimiento de un objeto intenta rodear o ir por el contorno de otro, o bien es un movimiento lineal seguido de pequeños giros cuando un objeto se encuentra en las cercanías de otro objeto (cuando sus circunferencias envolventes intersecan en nuestro caso). Por este motivo, siempre analizaremos estos dos tipos de trayectorias en las pruebas realizadas. En la tabla 4.3. mostramos el comportamiento de los algoritmos para distintas trayectorias y combinaciones de tipos de polígonos en los que se han variado el número de vértices.

Otro factor importante a tener en cuenta es la profundidad del tri-tree o número de niveles. Mediremos para un tipo de polígono cuál es la profundidad óptima y cual es el comportamiento de los algoritmos al variar la profundidad del árbol (Tabla 4.4). En las pruebas restantes utilizaremos una profundidad que dependerá del tamaño de los objetos y de la relación entre el tamaño de ambos. También ajustaremos el mínimo número de triángulos clasificados en un tri-cono a una constante, así como la iteración del algoritmo de ajuste, que sólo interviene en el tiempo de pre-procesamiento.

Siempre tendremos en cuenta el tamaño (número de vértices) de los polígonos. En primer lugar realizaremos pruebas de comportamiento con polígonos del mismo

tamaño, y luego realizaremos combinaciones entre polígonos de distintos tamaños (Tabla 4.5).

Por último consideraremos el tamaño relativo entre los polígonos, sobre todo para establecer el comportamiento de los algoritmos con un mismo tipo de polígono pero con distintos tamaños (Tabla 4.6).

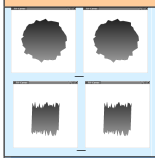
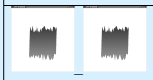
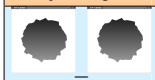
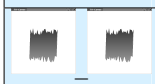
Datos de Entrada						
Parejas Polígonos	Vértices de cada par	Subdivisiones	Iteraciones A.A.	Relación Tamaño	Trayectoria	Algoritmos
	16, 16 64, 64 256, 256 1.024, 1.024	Constantes: Ajustadas al nº de vértices	8	1:1	Circular Lineal	DC (Alg.4.7) DC-TT (Alg. 4.13) DC-Lista (Alg. 4.14)
Leyenda						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Vértices de los dos polígonos	(Frames/seg.) escala log.		a) Trayectoria circular b) Trayectoria lineal	DC en función del nº de vértices, tipos de polígonos y trayectorias.		
			c) Trayectoria circular d) Trayectoria lineal			
Resultados						
Parejas Polígonos	Trayectoria	Pos. Algoritmo		Figura		
	Circular y Lineal	1º DC-TT 2º ≈ 3º DC-Lista, DC		4.57.a-b		
	Circular y Lineal	1º DC-TT 2º DC 3º DC-Lista		4.57.c-d		

Tabla 4.3: Resumen de resultados I para las pruebas de DC polígono/polígono.

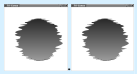
Datos de Entrada						
Pareja Polígonos	Vértices del par	Subdivisiones	Iteraciones A.A.	Relación Tamaño	Trayectoria	Algoritmos
	256, 256	4 8 16 32 64 128 256	8	1:1	Circular Lineal	DC (Alg.4.7) DC-TT (Alg. 4.13) DC-Lista (Alg. 4.14)
Leyenda						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Subdivisiones	(Frames/seg.) escala log.	a) Trayectoria circular b) Trayectoria lineal		DC en función del nº de subdivisiones y trayectorias.		
Resultados						
Trayectoria	Pos. Algoritmo			Figura		
Circular y Lineal	1º DC-TT 2º DC 3º DC-Lista			4.58		

Tabla 4.4: Resumen de resultados II para las pruebas de DC polígono/polígono.

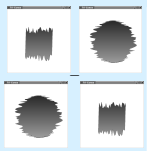
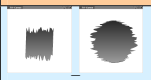
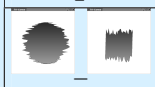
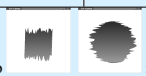
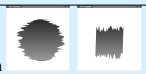
Datos de Entrada						
Parejas Polígonos	Vértices de cada par	Subdivisiones	Iteraciones A.A.	Relación Tamaño	Trayectoria	Algoritmos
	64, 64 64, 256 64, 1.024 256, 64 256, 256 256, 1.024 1.024, 64 1.024, 256 1.024, 1.024	Constantes: Ajustadas al nº de vértices	8	1:1	Circular Lineal	DC (Alg.4.7) DC-TT (Alg. 4.13) DC-Lista (Alg. 4.14)
Leyenda						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Vértices de los dos polígonos	(Frames/seg.) escala log.		a) Trayectoria circular b) Trayectoria lineal	DC en función del nº de vértices, tipos de polígonos y trayectorias.		
			c) Trayectoria circular d) Trayectoria lineal			
Resultados						
Vértices	Trayectoria	Pos. Algoritmo		Figura		
Pocos vértices (64)	Circular y Lineal	1º DC-TT 2º DC-Lista 3º DC		4.59		
Muchos vértices (>64)	Circular y Lineal	1º DC-TT 2º DC 3º DC-Lista		4.59		
En el caso  mejores tiempos que en 						

Tabla 4.5: Resumen de resultados III para las pruebas de DC polígono/polígono.

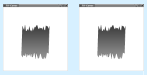
Datos de Entrada						
Pareja Polígonos	Vértices del par	Subdivisiones	Iteraciones A.A.	Relación Tamaño	Trayectoria	Algoritmos
	256, 256	Constantes: Ajustadas al nº de vértices y relación de tamaño	8	1:1 1:2 1:4 1:8 1:16	Circular Lineal	DC (Alg.4.7) DC-TT (Alg. 4.13) DC-Lista (Alg. 4.14)
Leyenda						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Relación Tamaño	(Frames/seg.) escala log.	a) Trayectoria circular b) Trayectoria lineal		DC en función de la relación entre el tamaño de los polígonos y las trayectorias.		
Resultados						
Trayectoria	Pos. Algoritmo		Figura			
Circular y Lineal	1º DC-Lista 2º DC 3º DC-TT		4.60			

Tabla 4.6: Resumen de resultados IV para las pruebas de DC polígono/polígono.

4.8.5. Resultados obtenidos

4.8.5.1. Detección de Colisión Punto/Polígono

Para el caso de polígonos no convexos irregulares (Tipo I), podemos apreciar en la Figura 4.51 una mejora significativa cuando se utiliza el algoritmo de DC con tri-trees (DC-TT) respecto al algoritmo de DC para polígonos no convexos sin el uso de tri-trees (DC), y también con respecto a la inclusión utilizando quadtrees para ambos tipos de trayectorias.

Según la Figura 4.52, en el caso de polígonos no convexos regulares (Tipo II) se aprecia una mejora en los tiempos obtenidos por los algoritmos respecto al caso de polígonos no convexos irregulares (Tipo I). El algoritmo de DC mediante el uso de tri-trees (DC-TT) es superior a los demás. Observamos también que el algoritmo de inclusión mediante quadtrees se comporta peor en el caso de trayectorias circulares que en el caso de trayectorias lineales.

Sin embargo en el caso de polígonos formados por un recubrimiento positivo (Tipo III), al utilizar un algoritmo específico adecuado a este caso (DC con el Algoritmo 4.4) obtenemos mejores tiempos que con el algoritmo que utiliza tri-trees (Figura 4.53).

En el caso de un polígono convexo (Tipo IV), podemos apreciar en la Figura 4.54 la gran mejora que supone el algoritmo específico para este caso (DC con el Algoritmo 4.3) respecto a los otros (DC-TT y Quadtree). También obtenemos una gran mejora en el caso del algoritmo de DC que utiliza tri-trees (DC-TT) respecto a los tiempos obtenidos para otro tipo de polígonos. Esto es así debido al alto grado de coherencia geométrica que posee un polígono convexo en relación a los otros tipos de polígonos considerados.

4.8.5.2. Detección de Colisión Circunferencia/Polígono

En la Figura 4.55 podemos ver el número de frames por segundo obtenido para un polígono con distinto número de vértices (16, 64, 256 y 1.024), en función del tamaño relativo entre objetos. Estos mismos datos podemos verlos en la Figura 4.56 en la que mostramos el número de frames por segundo para los algoritmos DC y DC-TT en dos gráficos, a distinta escala relativa y en función del tamaño del polígono. Podemos concluir que para polígonos con alto número de vértices es mejor el algoritmo DC-TT, sin embargo para un bajo número de vértices no merece la pena realizar la descomposición espacial mediante tri-trees, pues los tiempos obtenidos por el algoritmo DC-TT son peores que los obtenidos con el algoritmo DC.

Para el caso de una trayectoria lineal apreciamos resultados muy similares entre los métodos propuestos (DC y DC-TT), con variaciones poco significativas, por lo que no hemos considerado necesario incluir en este estudio los gráficos con los tiempos obtenidos en la DC.

4.8.5.3. Detección de Colisión Polígono/Polígono

En la Figura 4.57 podemos apreciar el número de frames por segundo obtenido para distintas trayectorias y combinaciones de polígonos en los que se ha variado el número de vértices. Podemos ver que en todos los casos el algoritmo de detección de colisión recursivo que utiliza tri-trees (DC-TT) es superior al resto para una escala relativa entre polígonos de 1:1. El algoritmo de detección de colisión no recursivo basado en el algoritmo circunferencia/polígono que utiliza listas de tri-conos (DC-Lista) obtiene unos tiempos similares a los del algoritmo que no utiliza tri-trees (DC). Después veremos que este algoritmo es más apropiado para el caso de polígonos con gran diferencia de tamaño.

En la Figura 4.58 podemos ver el resultado obtenido para el caso de detección de colisión entre dos polígonos con un número de vértices fijo, en el que se ha variado la profundidad del tri-tree (número de subdivisiones). Podemos apreciar que el algoritmo DC-TT obtiene mejores resultados a mayor número de subdivisiones que el resto. Se produce una disminución en el número de frames por segundo en el algoritmo cuando se aumenta demasiado la profundidad, pues en este caso hay que descender más en el árbol (con el incremento de tiempo que supone), obteniendo un número similar de triángulos clasificados en el último nivel que en niveles anteriores. Ocurre un comportamiento similar en el algoritmo DC-Lista.

La Figura 4.59 muestra el comportamiento de los algoritmos al variar el tamaño de los polígonos (número de vértices) y el tipo de polígono. Para polígonos con gran número de vértices, el comportamiento del algoritmo DC-TT es mejor que el resto. Para polígonos con un número de vértices pequeño no compensa utilizar tri-trees. El algoritmo DC-Lista es claramente ineficiente en el caso de una relación entre polígonos 1:1, pues es superado en la mayor parte de los casos incluso por el algoritmo DC que no usa tri-trees.

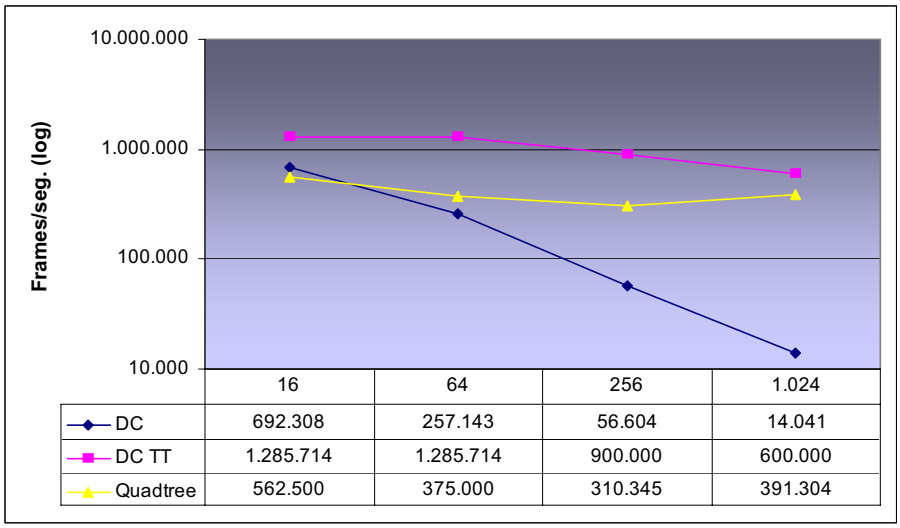
La última figura (Figura 4.60) nos presenta el comportamiento de los algoritmos ante la variación del tamaño relativo entre polígonos. Se ha fijado el tipo de polígono utilizado así como el número de vértices y se muestran los frames por segundo en función del tamaño relativo entre polígonos. Podemos apreciar un aumento en la eficiencia del algoritmo DC-Lista para polígonos con gran diferencia de tamaño.

Finalmente, como cada trayectoria tiene una densidad distinta en cuanto al número

de posiciones espaciales ocupadas, podemos ver que en el caso de trayectorias circulares obtenemos en general mejores tiempos que en el caso de trayectorias lineales, pues en el primer caso su densidad es mayor y por tanto se produce un aumento de la coherencia temporal que es aprovechada por los algoritmos.

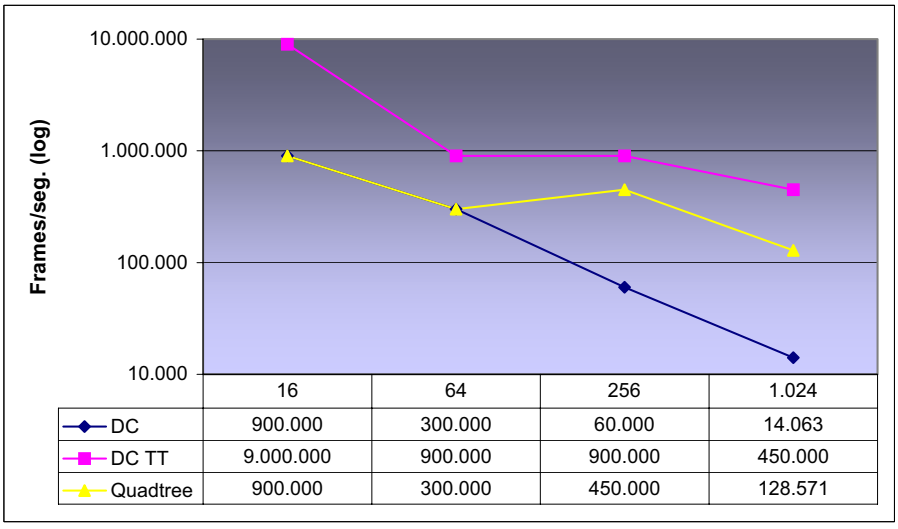
4.8.6. Figuras y Tablas con los tiempos obtenidos

a)



	16	64	256	1.024
DC	692.308	257.143	56.604	14.041
DC TT	1.285.714	1.285.714	900.000	600.000
Quadtree	562.500	375.000	310.345	391.304

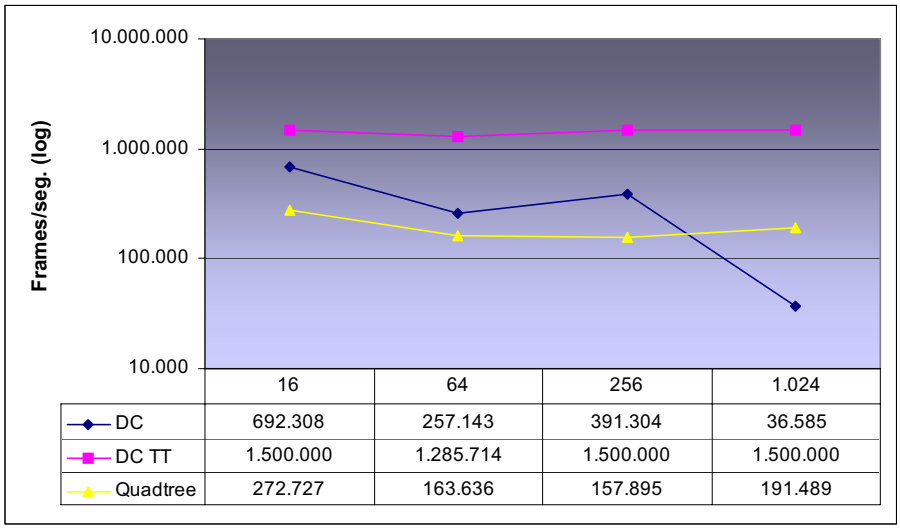
b)



	16	64	256	1.024
DC	900.000	300.000	60.000	14.063
DC TT	9.000.000	900.000	900.000	450.000
Quadtree	900.000	300.000	450.000	128.571

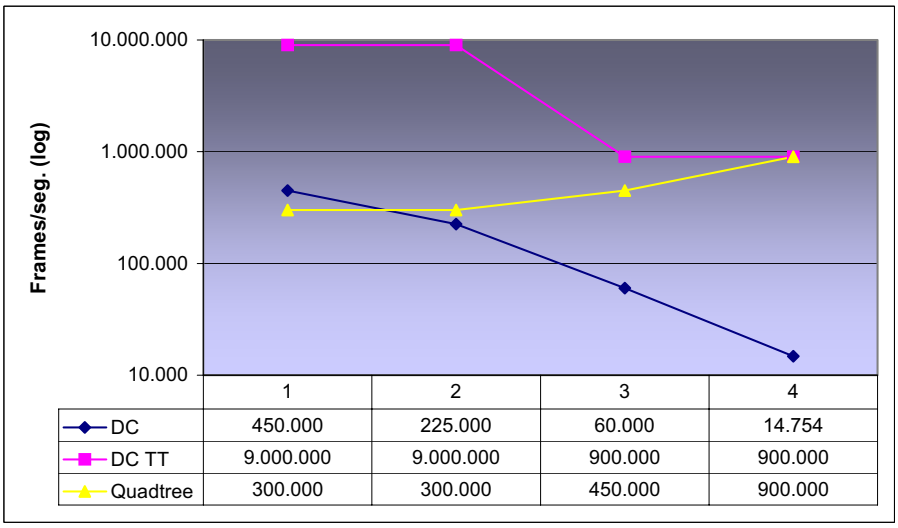
Figura 4.51: DC Punto/Polígono no convexo irregular (Tipo I). En el eje X se muestra el número de vértices del polígono y en el eje Y el número de frames por segundo (con escala logarítmica). a) Para trayectoria circular. b) Para trayectoria lineal.

a)



	16	64	256	1.024
DC	692.308	257.143	391.304	36.585
DC TT	1.500.000	1.285.714	1.500.000	1.500.000
Quadtree	272.727	163.636	157.895	191.489

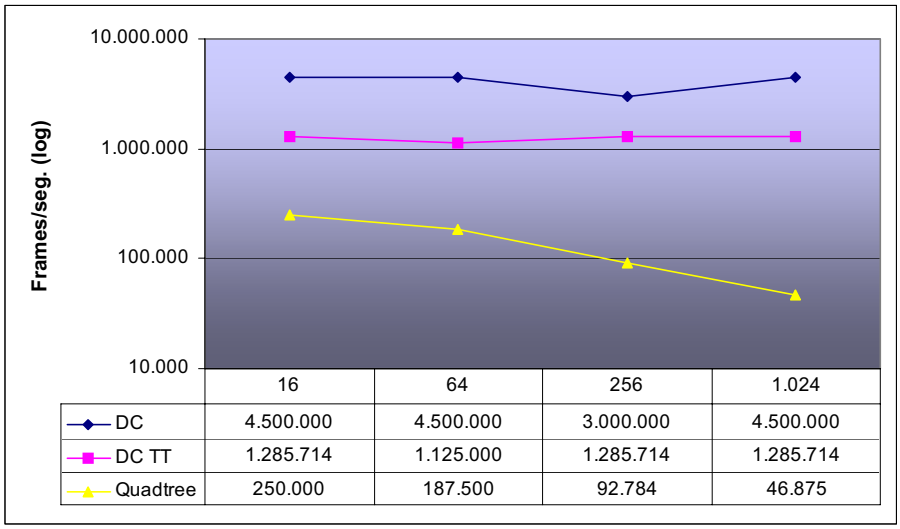
b)



	1	2	3	4
DC	450.000	225.000	60.000	14.754
DC TT	9.000.000	9.000.000	900.000	900.000
Quadtree	300.000	300.000	450.000	900.000

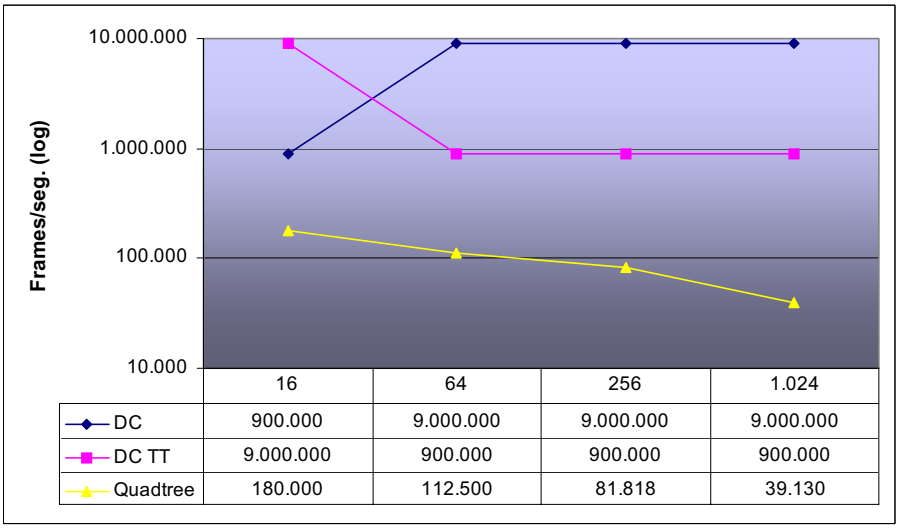
Figura 4.52: DC Punto/Polígono no convexo regular (Tipo II). En el eje X se muestra el número de vértices del polígono y en el eje Y el número de frames por segundo (con escala logarítmica). a) Para trayectoria circular. b) Para trayectoria lineal.

a)



	16	64	256	1.024
DC	4.500.000	4.500.000	3.000.000	4.500.000
DC TT	1.285.714	1.125.000	1.285.714	1.285.714
Quadtree	250.000	187.500	92.784	46.875

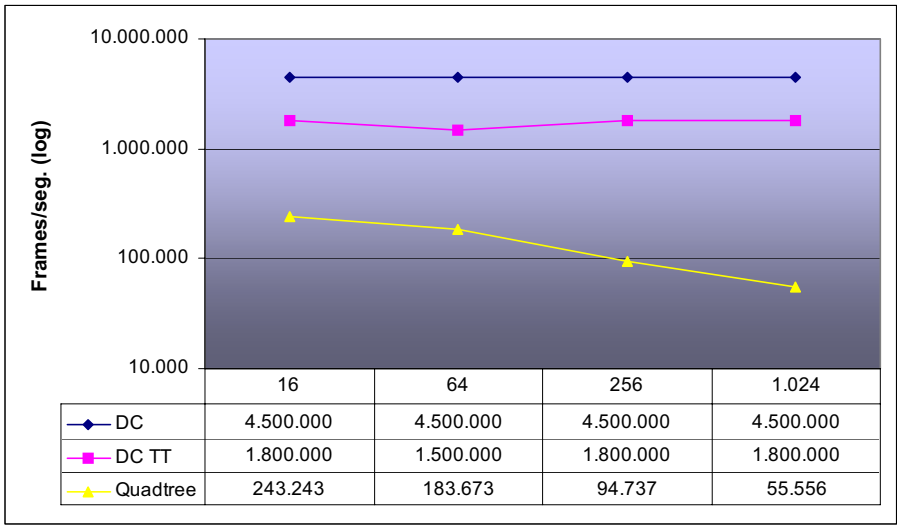
b)



	16	64	256	1.024
DC	900.000	9.000.000	9.000.000	9.000.000
DC TT	9.000.000	900.000	900.000	900.000
Quadtree	180.000	112.500	81.818	39.130

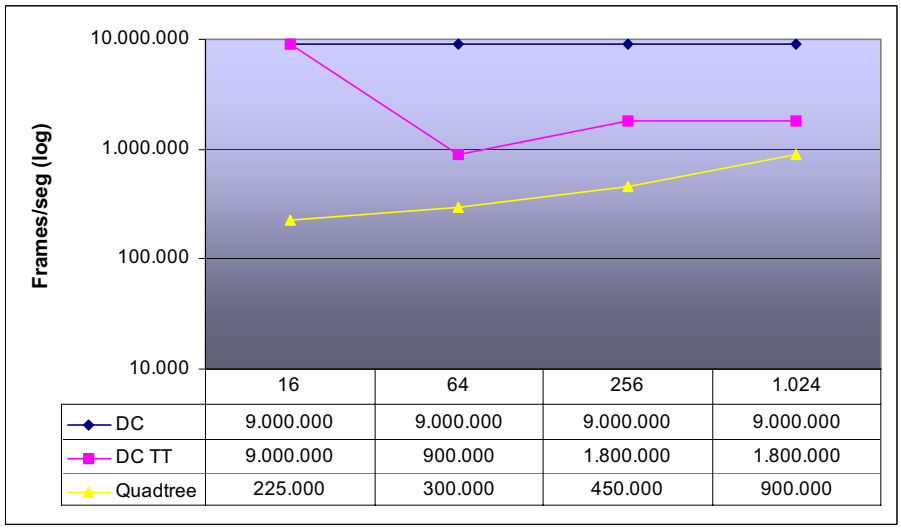
Figura 4.53: DC Punto/Polígono con recubrimiento positivo (Tipo III). En el eje X se muestra el número de vértices del polígono y en el eje Y el número de frames por segundo (con escala logarítmica). a) Trayectoria circular. b) Trayectoria lineal.

a)



	16	64	256	1.024
DC	4.500.000	4.500.000	4.500.000	4.500.000
DC TT	1.800.000	1.500.000	1.800.000	1.800.000
Quadtree	243.243	183.673	94.737	55.556

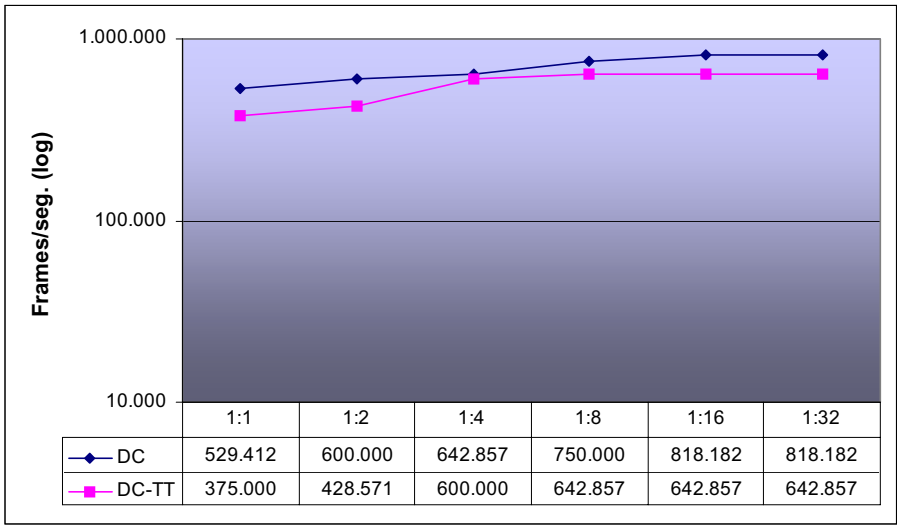
b)



	16	64	256	1.024
DC	9.000.000	9.000.000	9.000.000	9.000.000
DC TT	9.000.000	900.000	1.800.000	1.800.000
Quadtree	225.000	300.000	450.000	900.000

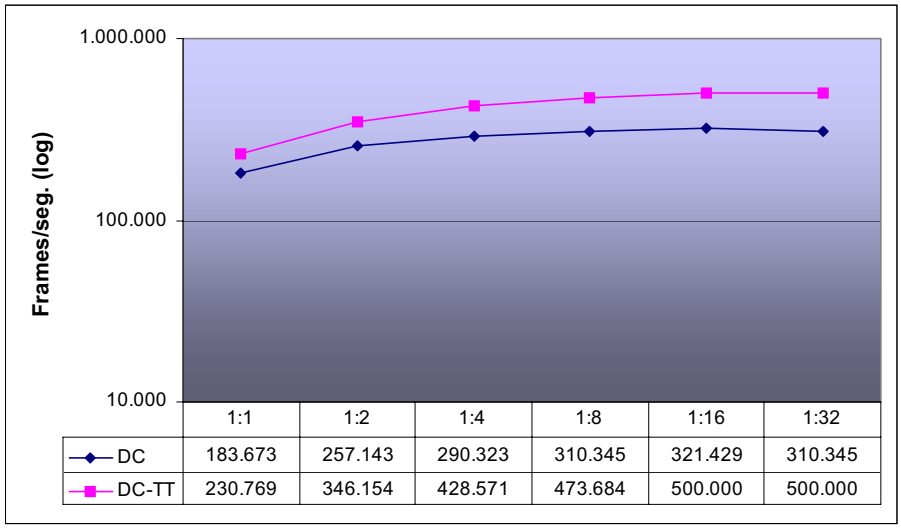
Figura 4.54: DC Punto/Polígono convexo (Tipo IV). En el eje X se muestra el número de vértices del polígono y en el eje Y el número de frames por segundo (con escala logarítmica). a) Trayectoria circular. b) Trayectoria lineal.

a)



	1:1	1:2	1:4	1:8	1:16	1:32
DC	529.412	600.000	642.857	750.000	818.182	818.182
DC-TT	375.000	428.571	600.000	642.857	642.857	642.857

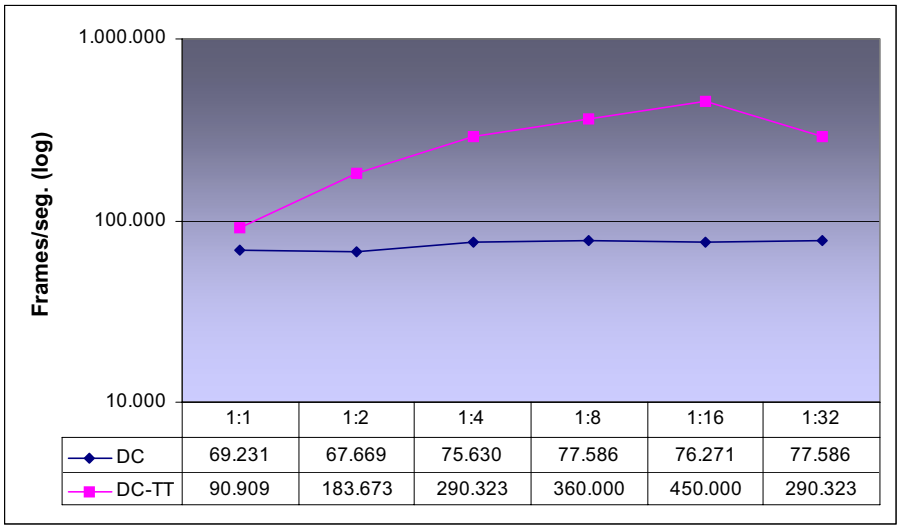
b)



	1:1	1:2	1:4	1:8	1:16	1:32
DC	183.673	257.143	290.323	310.345	321.429	310.345
DC-TT	230.769	346.154	428.571	473.684	500.000	500.000

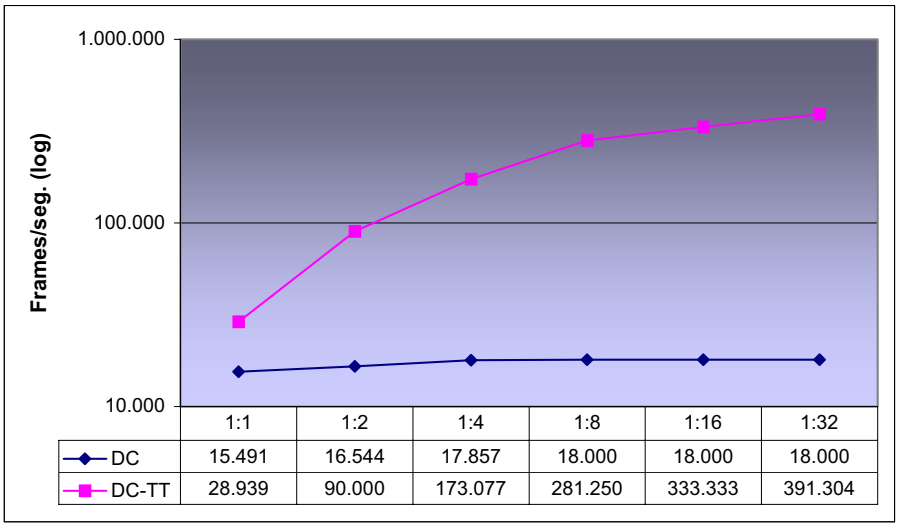
Figura 4.55: DC circunferencia/polígono no convexo irregular (Tipo I) para una trayectoria circular. Se ha variado el tamaño relativo entre el polígono y la circunferencia, parámetro que se muestra en el eje X; el número de frames por segundo con escala logarítmica se muestra en el eje Y. a) Para el caso de polígono de 16 vértices. b) 64 vértices. c) 256 vértices. d) 1.024 vértices.

c)



	1:1	1:2	1:4	1:8	1:16	1:32
DC	69.231	67.669	75.630	77.586	76.271	77.586
DC-TT	90.909	183.673	290.323	360.000	450.000	290.323

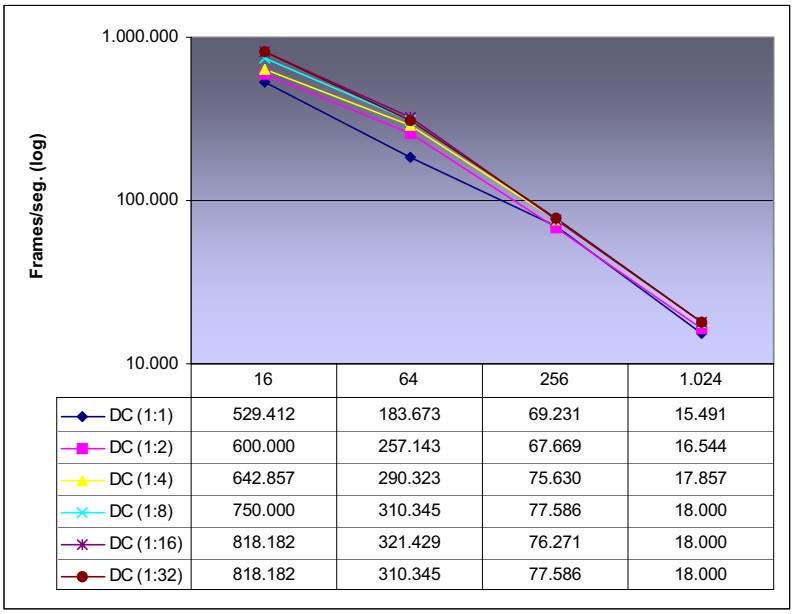
d)



	1:1	1:2	1:4	1:8	1:16	1:32
DC	15.491	16.544	17.857	18.000	18.000	18.000
DC-TT	28.939	90.000	173.077	281.250	333.333	391.304

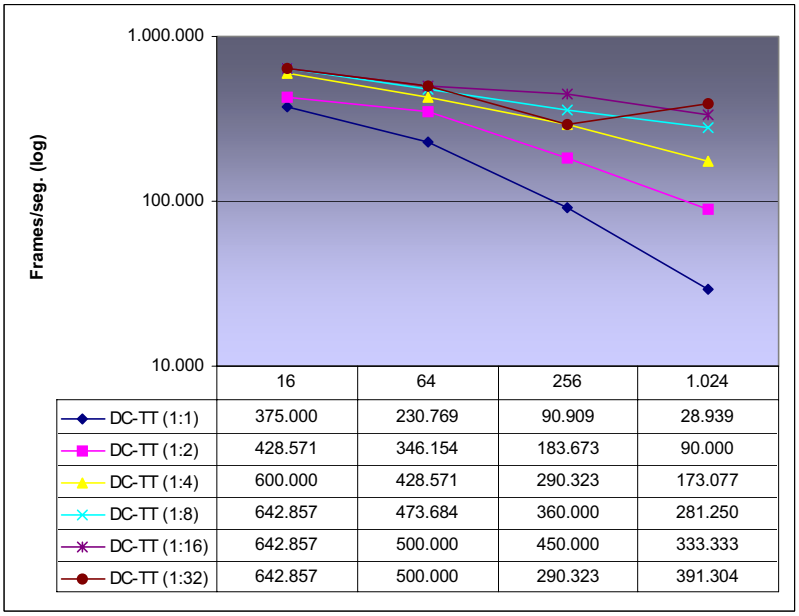
Figura 4.55: DC circunferencia/polígono no convexo irregular (Tipo I) para una trayectoria circular. Se ha variado el tamaño relativo entre el polígono y la circunferencia, parámetro que se muestra en el eje X; el número de frames por segundo con escala logarítmica se muestra en el eje Y. a) Para el caso de polígono de 16 vértices. b) 64 vértices. c) 256 vértices. d) 1.024 vértices.

a)



	16	64	256	1.024
DC (1:1)	529.412	183.673	69.231	15.491
DC (1:2)	600.000	257.143	67.669	16.544
DC (1:4)	642.857	290.323	75.630	17.857
DC (1:8)	750.000	310.345	77.586	18.000
DC (1:16)	818.182	321.429	76.271	18.000
DC (1:32)	818.182	310.345	77.586	18.000

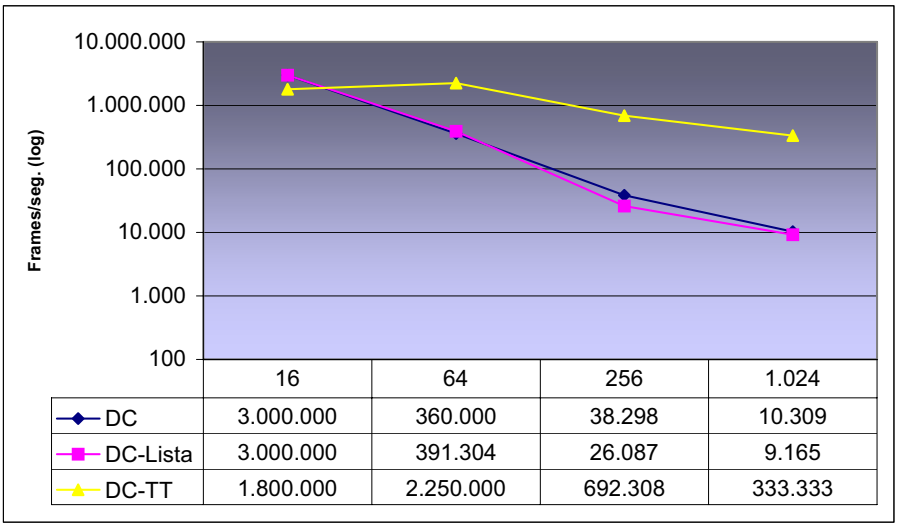
b)



	16	64	256	1.024
DC-TT (1:1)	375.000	230.769	90.909	28.939
DC-TT (1:2)	428.571	346.154	183.673	90.000
DC-TT (1:4)	600.000	428.571	290.323	173.077
DC-TT (1:8)	642.857	473.684	360.000	281.250
DC-TT (1:16)	642.857	500.000	450.000	333.333
DC-TT (1:32)	642.857	500.000	290.323	391.304

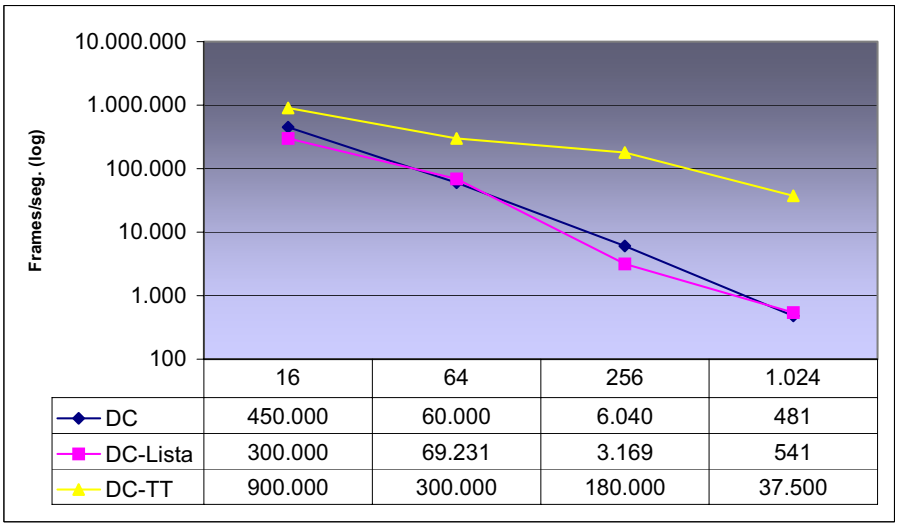
Figura 4.56: DC circunferencia/polígono no convexo irregular (Tipo I) para una trayectoria circular, en función de la escala relativa. En el eje X se muestra el tamaño del polígono, el número de frames por segundo se muestra en el eje Y con escala logarítmica. a) Algoritmo de DC sin tri-trees (DC). b) Algoritmo de DC con tri-trees (DC-TT).

a)



	16	64	256	1.024
DC	3.000.000	360.000	38.298	10.309
DC-Lista	3.000.000	391.304	26.087	9.165
DC-TT	1.800.000	2.250.000	692.308	333.333

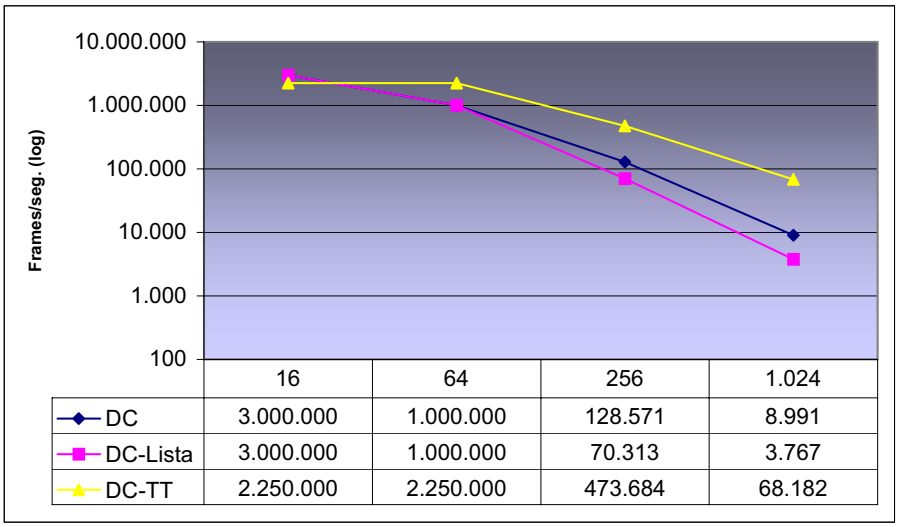
b)



	16	64	256	1.024
DC	450.000	60.000	6.040	481
DC-Lista	300.000	69.231	3.169	541
DC-TT	900.000	300.000	180.000	37.500

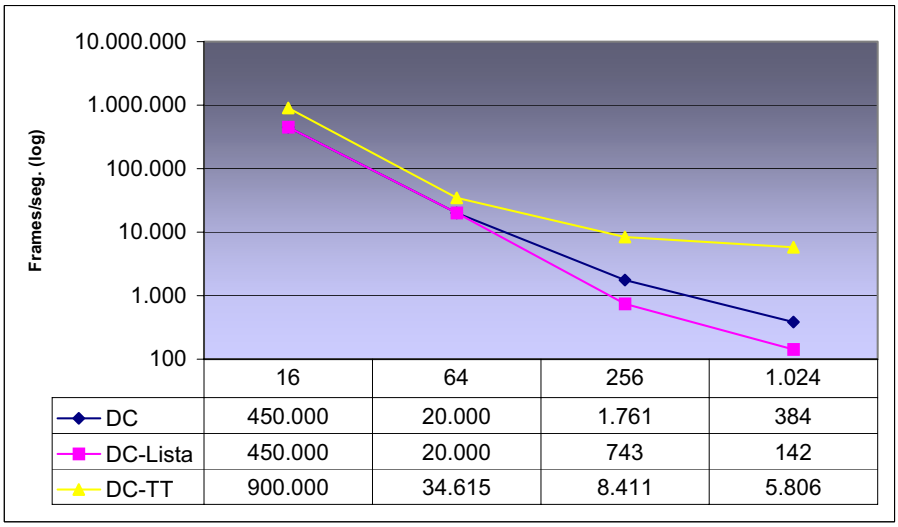
Figura 4.57: Variación del número de vértices en la DC entre polígonos. En el eje X se muestra el número de vértices común a los dos polígonos y en el eje Y el número de frames por segundo con escala logarítmica. a) Movimiento circular de dos polígonos no convexos regulares (Tipo II). b) Movimiento lineal de dos polígonos no convexos regulares (Tipo II). c) Movimiento circular de dos polígonos no convexos irregulares (Tipo I). d) Movimiento lineal de dos polígonos no convexos irregulares (Tipo I).

c)



	16	64	256	1.024
DC	3.000.000	1.000.000	128.571	8.991
DC-Lista	3.000.000	1.000.000	70.313	3.767
DC-TT	2.250.000	2.250.000	473.684	68.182

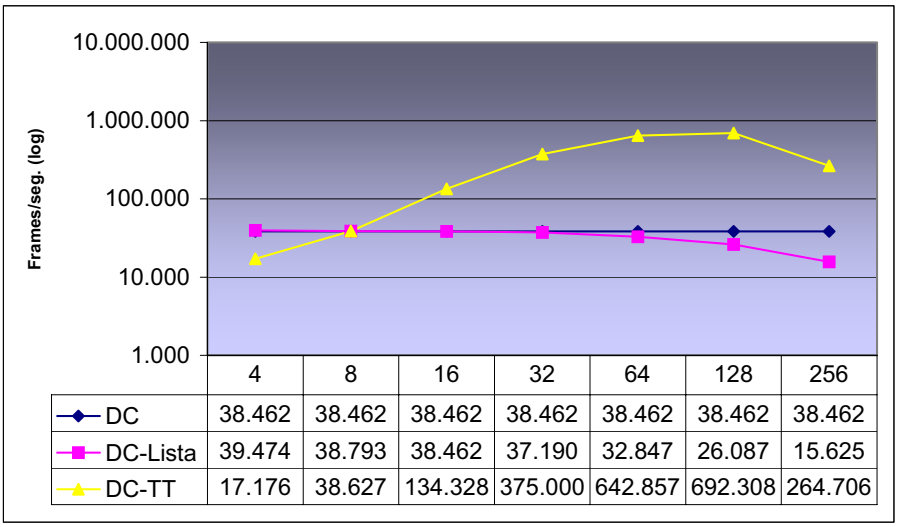
d)



	16	64	256	1.024
DC	450.000	20.000	1.761	384
DC-Lista	450.000	20.000	743	142
DC-TT	900.000	34.615	8.411	5.806

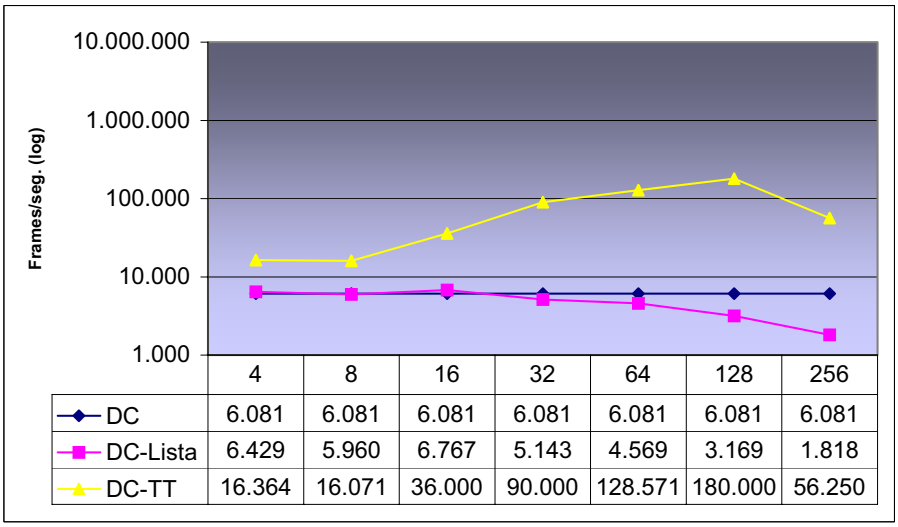
Figura 4.57: Variación del número de vértices en la DC entre polígonos. En el eje X se muestra el número de vértices común a los dos polígonos y en el eje Y el número de frames por segundo con escala logarítmica. a) Movimiento circular de dos polígonos no convexos regulares (Tipo II). b) Movimiento lineal de dos polígonos no convexos regulares (Tipo II). c) Movimiento circular de dos polígonos no convexos irregulares (Tipo I). d) Movimiento lineal de dos polígonos no convexos irregulares (Tipo I).

a)



	4	8	16	32	64	128	256
DC	38.462	38.462	38.462	38.462	38.462	38.462	38.462
DC-Lista	39.474	38.793	38.462	37.190	32.847	26.087	15.625
DC-TT	17.176	38.627	134.328	375.000	642.857	692.308	264.706

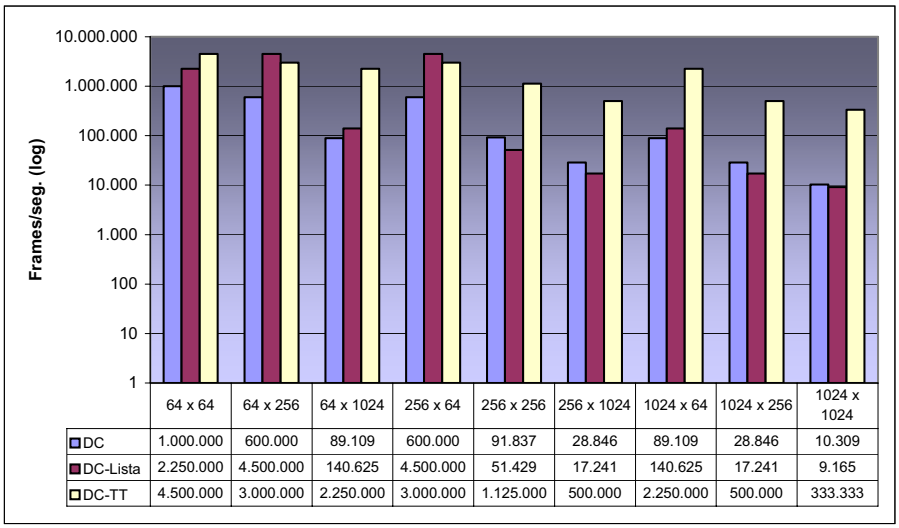
b)



	4	8	16	32	64	128	256
DC	6.081	6.081	6.081	6.081	6.081	6.081	6.081
DC-Lista	6.429	5.960	6.767	5.143	4.569	3.169	1.818
DC-TT	16.364	16.071	36.000	90.000	128.571	180.000	56.250

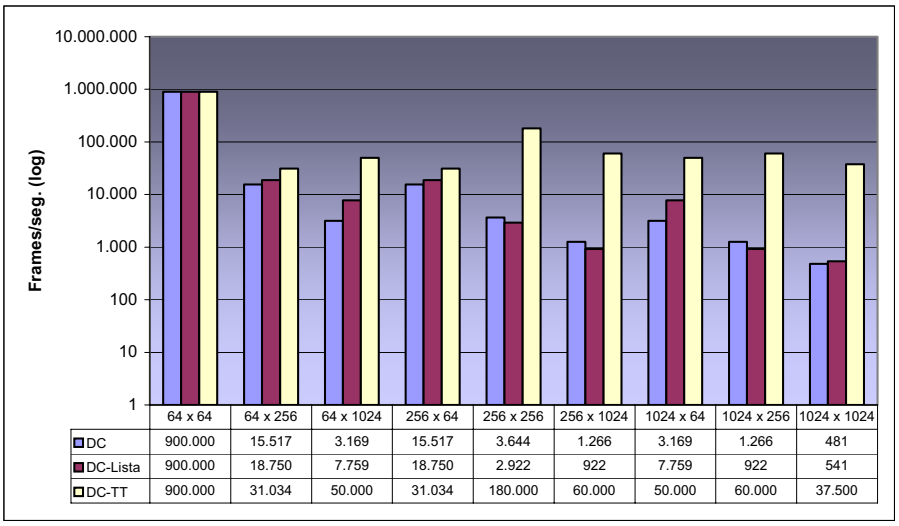
Figura 4.58: Variación de la profundidad del tri-tree para la DC entre dos polígonos no convexos regulares (Tipo V) de 256 vértices cada uno. En el eje X se muestra el número de subdivisiones del espacio y en el eje Y el número de frames por segundo con escala logarítmica. a) Movimiento circular. b) Movimiento lineal.

a)



	64 x 64	64 x 256	64 x 1024	256 x 64	256 x 256	256 x 1024	1024 x 64	1024 x 256	1024 x 1024
DC	1.000.000	600.000	89.109	600.000	91.837	28.846	89.109	28.846	10.309
DC-Lista	2.250.000	4.500.000	140.625	4.500.000	51.429	17.241	140.625	17.241	9.165
DC-TT	4.500.000	3.000.000	2.250.000	3.000.000	1.125.000	500.000	2.250.000	500.000	333.333

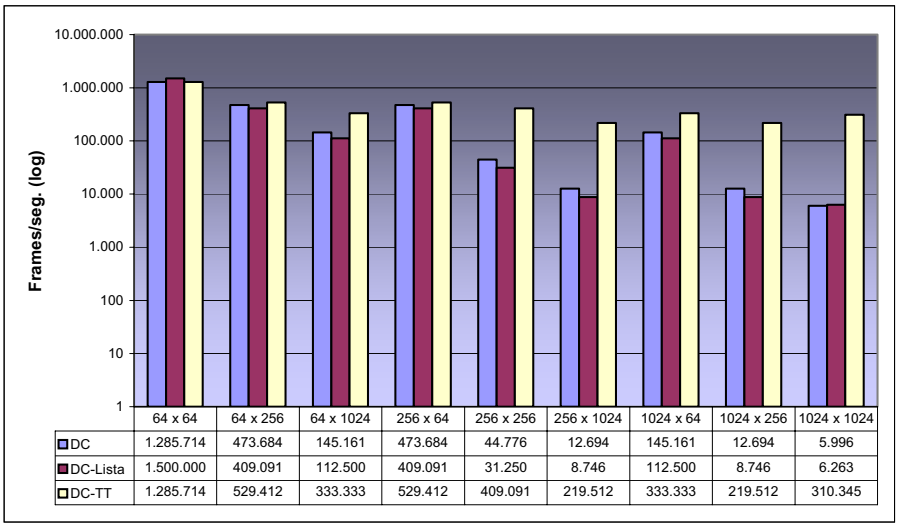
b)



	64 x 64	64 x 256	64 x 1024	256 x 64	256 x 256	256 x 1024	1024 x 64	1024 x 256	1024 x 1024
DC	900.000	15.517	3.169	15.517	3.644	1.266	3.169	1.266	481
DC-Lista	900.000	18.750	7.759	18.750	2.922	922	7.759	922	541
DC-TT	900.000	31.034	50.000	31.034	180.000	60.000	50.000	60.000	37.500

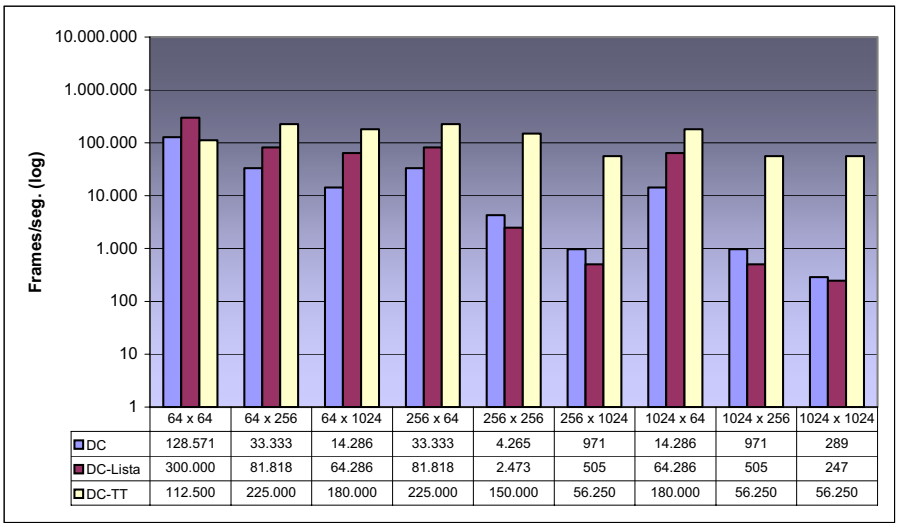
Figura 4.59: Variación del número de vértices en la DC entre polígonos de diverso tipo. En el eje X se muestra el número de vértices de los polígonos y en el eje Y el número de frames por segundo con escala logarítmica. a) Movimiento circular de un polígono no convexo regular (Tipo V) alrededor de un polígono no convexo irregular (Tipo I). b) Movimiento lineal de un polígono no convexo regular (Tipo V) en relación a un polígono no convexo irregular (Tipo I). c) Movimiento circular de un polígono no convexo irregular (Tipo I) alrededor de un polígono no convexo regular (Tipo V). d) Movimiento lineal de un polígono no convexo irregular (Tipo I) en relación a un polígono no convexo regular (Tipo V).

c)



	64 x 64	64 x 256	64 x 1024	256 x 64	256 x 256	256 x 1024	1024 x 64	1024 x 256	1024 x 1024
DC	1.285.714	473.684	145.161	473.684	44.776	12.694	145.161	12.694	5.996
DC-Lista	1.500.000	409.091	112.500	409.091	31.250	8.746	112.500	8.746	6.263
DC-TT	1.285.714	529.412	333.333	529.412	409.091	219.512	333.333	219.512	310.345

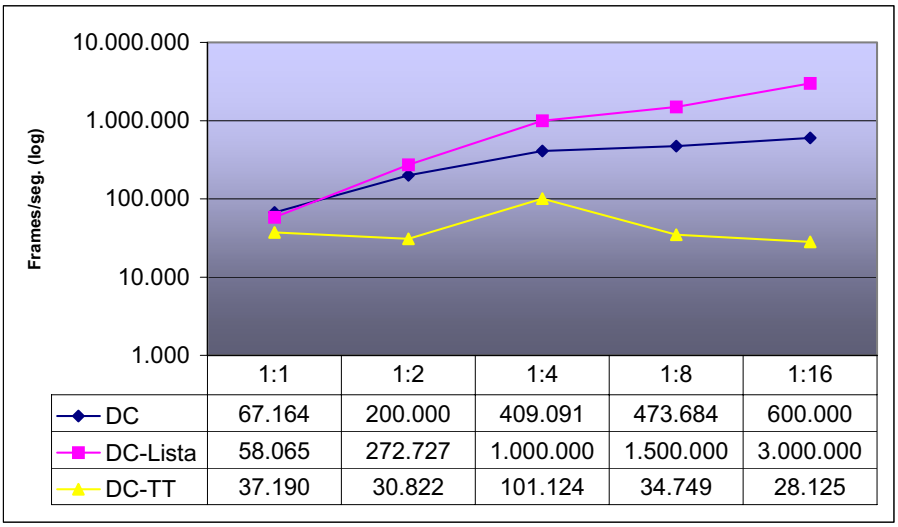
d)



	64 x 64	64 x 256	64 x 1024	256 x 64	256 x 256	256 x 1024	1024 x 64	1024 x 256	1024 x 1024
DC	128.571	33.333	14.286	33.333	4.265	971	14.286	971	289
DC-Lista	300.000	81.818	64.286	81.818	2.473	505	64.286	505	247
DC-TT	112.500	225.000	180.000	225.000	150.000	56.250	180.000	56.250	56.250

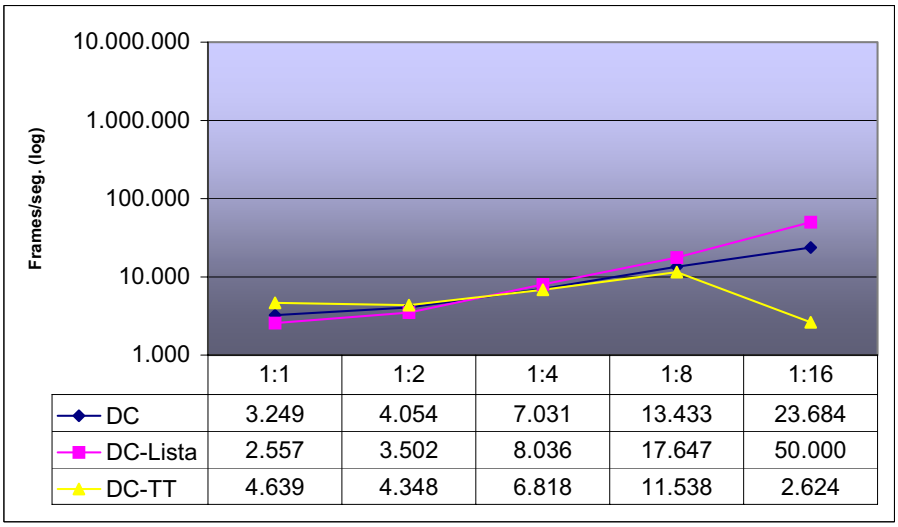
Figura 4.59: Variación del número de vértices en la DC entre polígonos de diverso tipo. En el eje X se muestra el número de vértices de los polígonos y en el eje Y el número de frames por segundo con escala logarítmica. a) Movimiento circular de un polígono no convexo regular (Tipo V) alrededor de un polígono no convexo irregular (Tipo I). b) Movimiento lineal de un polígono no convexo regular (Tipo V) en relación a un polígono no convexo irregular (Tipo I). c) Movimiento circular de un polígono no convexo irregular (Tipo I) alrededor de un polígono no convexo regular (Tipo V). d) Movimiento lineal de un polígono no convexo irregular (Tipo I) en relación a un polígono no convexo regular (Tipo V).

a)



	1:1	1:2	1:4	1:8	1:16
DC	67.164	200.000	409.091	473.684	600.000
DC-Lista	58.065	272.727	1.000.000	1.500.000	3.000.000
DC-TT	37.190	30.822	101.124	34.749	28.125

b)



	1:1	1:2	1:4	1:8	1:16
DC	3.249	4.054	7.031	13.433	23.684
DC-Lista	2.557	3.502	8.036	17.647	50.000
DC-TT	4.639	4.348	6.818	11.538	2.624

Figura 4.60: Variación de la relación entre el tamaño de los polígonos en la DC entre un par de polígonos no convexos irregulares (Tipo I) de 256 vértices cada uno. En el eje X se muestra la relación entre el tamaño de los polígonos y en el eje Y el número de frames por segundo con escala logarítmica. a) Movimiento circular. b) Movimiento lineal.

4.9. Conclusiones

En este capítulo hemos empleado un conjunto de técnicas y algoritmos para la detección de colisión en 2D utilizando recubrimientos simpliciales. Hemos desarrollado un nuevo tipo de descomposición espacial denominada Tri-Tree basada en tri-conos. Este tipo de descomposición espacial se ajusta a los polígonos en mejor medida que otro tipo de descomposiciones. Posteriormente hemos incorporado tri-trees a los algoritmos desarrollados. Todos estos conceptos los extenderemos a 3D para obtener algoritmos de detección de colisión.

Según las pruebas realizadas, la construcción de un tri-tree es bastante rápida y se realiza en tiempo de pre-procesamiento, no añadiendo un coste suplementario a los algoritmos de DC. Además se puede ajustar adecuadamente a la forma de un polígono por lo que podría utilizarse para el modelado de los mismos, siendo una estructura de datos que no varía cuando se producen transformaciones rígidas, por lo que no es necesario recalcularla. Su utilización en la detección de colisión es apropiada en cuanto a que el tiempo de acceso a sus elementos es menor comparado con otros tipos de descomposiciones espaciales.

En cuanto a la DC punto/polígono, hemos desarrollado algoritmos específicos para el caso de polígonos convexos y con recubrimiento positivo además del algoritmo genérico para polígonos no convexos. Estos algoritmos específicos obtienen tiempos casi constantes en la detección de colisión, independientemente del número de vértices de los polígonos, y tiempos mejores que con la utilización de tri-trees en los algoritmos. El uso de tri-trees para el resto de los casos optimiza el tiempo de detección de colisión en gran medida.

Hemos desarrollado también un algoritmo de DC circunferencia/polígono extendiendo conceptos desarrollados para la DC entre puntos y polígonos. Esto nos ha llevado a obtener tiempos cercanos a los obtenidos en la DC entre un punto y un polígono, sobre todo cuando la circunferencia es pequeña en relación al tamaño del polígono. Para el resto de los casos nos ofrece una aproximación a la DC entre polígonos, pues la circunferencia puede usarse como volumen envolvente de uno de los polígonos, de manera que obtenemos algoritmos de DC polígono/polígono más eficientes para el caso de polígonos con distinto tamaño relativo.

Por último para la detección de colisión entre polígonos, además del algoritmo basado en la DC circunferencia/polígono hemos desarrollado otro que recorre recursivamente los tri-trees de ambos polígonos. Este algoritmo es adecuado cuando los polígonos tienen un tamaño similar.

CAPÍTULO

5

Detección de Colisiones 3D

En este Capítulo extenderemos las principales técnicas vistas en el capítulo anterior para su aplicación a algoritmos de Detección de Colisión entre objetos tridimensionales, representados mediante recubrimientos simpliciales. Estudiaremos un nuevo tipo de descomposición espacial denominada Tetra-Tree, que junto al uso de la coherencia, nos permitirá acelerar los cálculos en los algoritmos de detección de colisión desarrollados.

CONTENIDOS:

5. Detección de Colisiones 3D

Una vez establecidos los conceptos básicos y definidos una serie de algoritmos para la detección de colisión en 2D, pasamos a extender dichos conceptos y aplicarlos a la construcción de algoritmos para objetos poliédricos complejos en 3D[*], representados mediante recubrimientos simpliciales. A lo largo de este capítulo describiremos las relaciones entre diversos tipos de *d*-símplices[†] y poliedros, y definiremos algunas propiedades que nos permitan el aprovechamiento de la coherencia para acelerar los cálculos. Desarrollaremos algoritmos nuevos para, por una parte, implementar operaciones de inclusión e intersección entre objetos, y por otra, operaciones de detección de colisión entre un punto y un poliedro, una esfera y un poliedro, y finalmente entre poliedros.

Debido al coste computacional que supone realizar una detección de colisión entre objetos tridimensionales, hemos optado por desarrollar directamente algoritmos de DC que utilizan una descomposición espacial basada en *tetra-conos*, que hemos llamado árbol de tetra-conos o *tetra-tree*. Para cada tetra-cono asociaremos también tetraedros y esferas envolventes.

En la fase de poda o fase ancha de la DC utilizaremos esferas envolventes para descartar pares de objetos distantes entre si, y reducir así el número de objetos a tratar. En éste capítulo nos centraremos sobre todo en la detección de colisión a bajo nivel (fase estrecha).

---

[*] Nos referimos al *sólido 3D* definido en el Capítulo 3. Utilizaremos el término *poliedro complejo* para hacer hincapié en que puede estar formado por caras no convexas, con agujeros o no variedad. En ésta definición se incluye el caso de *mallas cerradas de triángulos*, por lo que los algoritmos y técnicas descritas en éste capítulo son igualmente válidas para dichas mallas, aunque dichos algoritmos pueden optimizarse para este caso, como mostramos al final del capítulo.

[†] Puntos, segmentos, triángulos y tetraedros.

Utilizaremos un enfoque constructivo en el desarrollo de este capítulo mediante el cual describiremos en primer lugar las técnicas y algoritmos auxiliares, necesarios para el desarrollo de una detección de colisión eficiente, para terminar describiendo los algoritmos de detección de colisión que hacen uso de esas técnicas y algoritmos auxiliares.

Comenzaremos, igual que hicimos en 2D, por caracterizar la inclusión y la posición de un punto respecto a un tetraedro, operación básica para la DC. Después de esto, definiremos el concepto de tetra-tree y sus propiedades más relevantes de cara a la implementación de algoritmos de detección de colisión entre objetos. Seguiremos con los fundamentos y algoritmos auxiliares necesarios para cada caso específico de detección de colisión entre objetos, para mostrar finalmente los algoritmos de DC que se basan en dichos fundamentos.

Para finalizar el capítulo, realizaremos un estudio temporal que nos muestre la eficiencia de los métodos implementados, así como de la descomposición espacial utilizada, seguido de las conclusiones, que resumirán los logros alcanzados a lo largo de este capítulo.

5.1. Caracterización de la inclusión de puntos en tetraedros

El estudio de la posición entre distintos elementos geométricos, como el caso de un punto y un tetraedro, es la base para la mayor parte de los algoritmos de detección de colisión que utilizan recubrimientos simpliciales. De este modo reduciremos las operaciones para la detección de colisión a operaciones entre *d*-símplices.

Al igual que para el caso de triángulos, utilizaremos las coordenadas baricéntricas de un punto respecto a un tetraedro para determinar su inclusión y establecer la posición exacta del punto respecto al tetraedro en cuestión. Esto nos permitirá conocer si un punto se encuentra exactamente sobre un vértice, una arista o una cara concreta del tetraedro. También podremos saber si el punto se encuentra en el interior del tetraedro o fuera del mismo. De este modo, mediante operaciones simples entre puntos y tetraedros podremos obtener, por ejemplo, la inclusión de un punto en un poliedro complejo.

Existen numerosos métodos para el estudio de la inclusión de puntos en tetraedros. Por homogeneidad y debido a su bajo coste computacional utilizaremos las coordenadas baricéntricas signadas de un punto respecto a un tetraedro para determinar la inclusión de dicho punto en el tetraedro. Otros autores [FT97b] [SFMOT05] han utilizado volúmenes signados para este fin; nosotros vamos a

extender y generalizar ese método basándonos en las coordenadas baricéntricas de un punto respecto a un tetraedro.

A lo largo de este capítulo consideraremos que vértices, aristas, triángulos y tetraedros están definidos en $\Re^3$, salvo que digamos explícitamente lo contrario.

Utilizaremos la definición clásica de coordenadas baricéntricas signadas de un punto respecto a un tetraedro [Ebe04] y [Eri05]:

Definición 5.1: Sean V_0,V_1,V_2,V_3 los vértices ordenados de un tetraedro de volumen no nulo. Sea P un punto en $\Re^3$. Entonces existen unos únicos valores $\alpha,\beta,\gamma,\delta \in \Re$, de manera que cumplen que $\alpha + \beta + \gamma + \delta = 1$ y que $P = \alpha V_0 + \beta V_1 + \gamma V_2 + \delta V_3$. Estos valores son:

$$\begin{aligned} \alpha &= |PV_1V_2V_3| \,/\, |V_0V_1V_2V_3| \\ \beta &= |V_0PV_2V_3| \,/\, |V_0V_1V_2V_3| \\ \gamma &= |V_0V_1PV_3| \,/\, |V_0V_1V_2V_3| \\ \delta &= |V_0V_1V_2P| \,/\, |V_0V_1V_2V_3| \end{aligned}$$

Los valores $\alpha,\beta,\gamma,\delta$ son las *coordenadas baricéntricas signadas* del punto *P* respecto al tetraedro $V_0V_1V_2V_3$ (Figura 5.1). En adelante las llamaremos simplemente *coordenadas baricéntricas* de un punto respecto a un tetraedro.

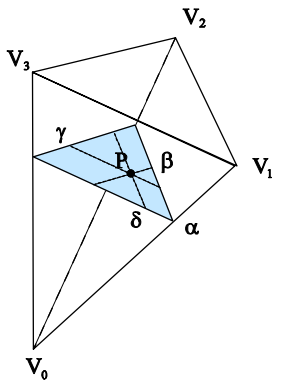


Figura 5.1: Coordenadas baricéntricas de un punto respecto a un tetraedro.

Podemos determinar si un punto se encuentra o no dentro de un tetraedro calculando el signo de sus coordenadas baricéntricas. Para ello podemos hacer uso de la siguiente propiedad:

Propiedad 5.1: Sea P un punto con coordenadas baricéntricas $(\alpha,\beta,\gamma,\delta)$ respecto al tetraedro definido por los vértices V_0,V_1,V_2,V_3. El punto P está dentro del tetraedro $V_0V_1V_2V_3$ si y sólo si $0 \leq \alpha, \beta, \gamma, \delta \leq 1$.

Es posible realizar una interpretación geométrica de la Propiedad 5.1 para comprobar en qué posición se encuentra un punto respecto a un tetraedro. Para llevar a cabo dicha interpretación examinamos el signo de los determinantes que definen las coordenadas baricéntricas del punto. Dependiendo de la orientación del tetraedro tendremos dos situaciones distintas:

- Cuando el tetraedro $V_0V_1V_2V_3$ tiene volumen signado positivo*, es decir $|V_0V_1V_2V_3|>0$. En este caso la coordenada baricéntrica α tiene valor 0 cuando el volumen signado es $|PV_1V_2V_3|=0$. Esto ocurre exclusivamente cuando el punto P se encuentra sobre el plano soporte† del triángulo $V_1V_2V_3$. Además α es negativa cuando $|PV_1V_2V_3|<0$, es decir, cuando P se encuentra en el lado opuesto a V_0 respecto al plano soporte de $V_1V_2V_3$; y positiva cuando $|PV_1V_2V_3|>0$, es decir, cuando P se encuentra en el mismo lado que V_0 respecto al plano soporte de $V_1V_2V_3$.

- Cuando el tetraedro $V_0V_1V_2V_3$ tiene volumen signado negativo, es decir $|V_0V_1V_2V_3|<0$. En esta situación α es negativa cuando $|PV_1V_2V_3|>0$, lo que indica que P se encuentra en el lado opuesto a V_0 respecto al plano soporte de $V_1V_2V_3$. Del mismo modo, α es positiva cuando $|PV_1V_2V_3|<0$, lo cual indica que P se encuentra en el mismo lado que V_0 respecto al plano soporte de $V_1V_2V_3$. Cuando $|PV_1V_2V_3|=0$ entonces la coordenada baricéntrica α es cero, por lo que P está sobre el plano soporte de $V_1V_2V_3$.

Podemos llegar a conclusiones similares para las coordenadas baricéntricas β, γ y δ del punto respecto al tetraedro. En estos casos, el signo de cada coordenada baricéntrica β, γ y δ respecto a los triángulos $V_0V_3V_2$, $V_0V_1V_3$, y $V_0V_2V_1$ respectivamente, representa el lado en el que se encuentra P respecto a los planos soporte de dichos triángulos.

Una consecuencia derivada de la Propiedad 5.1 nos permite conocer si un punto se encuentra fuera de un tetraedro:

---

* Al tetraedro que cumple esta propiedad lo hemos llamado *tetraedro positivo*.

† Dado un triángulo $V_1V_2V_3$, llamaremos *plano soporte* de dicho triángulo, al plano que pasa V_1,V_2 y V_3.

Propiedad 5.2: Sea P un punto con coordenadas baricéntricas $(\alpha,\beta,\gamma,\delta)$ respecto al tetraedro $V_0V_1V_2V_3$. El punto P está fuera del tetraedro $V_0V_1V_2V_3$ si y sólo si $\alpha<0 \vee \beta<0 \vee \gamma<0 \vee \delta<0$.

A continuación, caracterizaremos la posición de un punto P respecto a un tetraedro $V_0V_1V_2V_3$ utilizando las coordenadas baricéntricas $(\alpha,\beta,\gamma,\delta)$ de ese punto respecto al tetraedro (Figura 5.2). En nuestro caso particular nos interesa conocer, además de si un punto está dentro o fuera del tetraedro, si está sobre un vértice, una arista, o una cara concreta, y si no es así, si se encuentra en el interior del tetraedro. Todas estas expresiones pueden substituirse por expresiones equivalentes que consideren únicamente el signo de la correspondiente coordenada baricéntrica. Mediante el signo de las coordenadas baricéntricas podremos obtener algoritmos de detección de colisión más eficientes y robustos, basados en aritmética entera. Las distintas situaciones de un punto respecto a un tetraedro son las siguientes*:

- El punto P está sobre el vértice V_0 sii $\alpha = 1$ y $\beta,\gamma,\delta = 0$
- El punto P está sobre el vértice V_1 sii $\beta = 1$ y $\alpha,\gamma,\delta = 0$
- El punto P está sobre el vértice V_2 sii $\gamma = 1$ y $\alpha,\beta,\delta = 0$
- El punto P está sobre el vértice V_3 sii $\delta = 1$ y $\alpha,\beta,\gamma = 0$

- El punto P está en el interior de la arista V_0V_1 sii $\gamma,\delta = 0$ y $\alpha,\beta > 0$
- El punto P está en el interior de la arista V_0V_2 sii $\beta,\delta = 0$ y $\alpha,\gamma > 0$
- El punto P está en el interior de la arista V_0V_3 sii $\beta,\gamma = 0$ y $\alpha,\delta > 0$
- El punto P está en el interior de la arista V_1V_2 sii $\alpha,\delta = 0$ y $\beta,\gamma > 0$
- El punto P está en el interior de la arista V_2V_3 sii $\alpha,\beta = 0$ y $\gamma,\delta > 0$
- El punto P está en el interior de la arista V_3V_1 sii $\alpha,\gamma = 0$ y $\beta,\delta > 0$

- El punto P está en el interior de la cara $V_0V_1V_2$ sii $\delta = 0$ y $\alpha,\beta,\gamma > 0$
- El punto P está en el interior de la cara $V_0V_2V_3$ sii $\beta = 0$ y $\alpha,\gamma,\delta > 0$
- El punto P está en el interior de la cara $V_0V_3V_1$ sii $\gamma = 0$ y $\alpha,\beta,\delta > 0$
- El punto P está en el interior de la cara $V_1V_2V_3$ sii $\alpha = 0$ y $\beta,\gamma,\delta > 0$

- El punto P está dentro del tetraedro $V_0V_1V_2V_3$ sii $\alpha,\beta,\gamma,\delta \geq 0$
- El punto P está fuera del tetraedro $V_0V_1V_2V_3$ sii $\alpha,\beta,\gamma,\delta < 0$
- El punto P está en el interior del tetraedro $V_0V_1V_2V_3$ sii $\alpha,\beta,\gamma,\delta > 0$

* Recordemos que los vértices del tetraedro están ordenados, y la definición de las coordenadas baricéntricas de un punto respecto a un tetraedro se ha realizado en base al orden de dichos vértices.

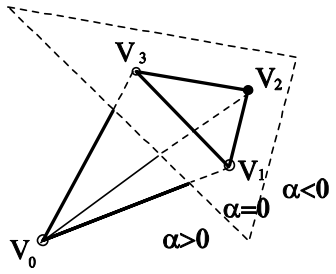


Figura 5.2: Relación entre un punto y un tetraedro en base a la coordenada baricéntrica α. Un punto se encontrará a un lado u otro del plano soporte de $V_1V_2V_3$ en función de *sign*(α).

Podríamos diseñar un algoritmo genérico que obtuviese la posición relativa de un punto respecto a un tetraedro mediante la descripción anterior. Sin embargo, un algoritmo concreto puede necesitar sólo cierta información, como que el punto esté sobre un triángulo de la frontera o una arista concreta, por lo que este algoritmo podría optimizarse para contemplar sólo los casos necesarios. En función de las necesidades se usará un algoritmo específico que detecte sólo los casos necesarios.

Notación: Sea T_i el tetraedro $V_oV_1V_2V_3$, llamaremos $\alpha_i(P),\beta_i(P),\gamma_i(P),\delta_i(P)$ a las coordenadas baricéntricas del punto P respecto a T_i, siendo $\alpha_i = |PV_1V_2V_3| \,/\, |V_oV_1V_2V_3|$, $\beta_i = |V_oPV_2V_3| \,/\, |V_oV_1V_2V_3|$, $\gamma_i = |V_oV_1PV_3| \,/\, |V_oV_1V_2V_3|$ y $\delta_i = |V_oV_1V_2P| \,/\, |V_oV_1V_2V_3|$.

A lo largo de este capítulo daremos los vértices de los tetraedros en el orden adecuado para que α siempre se corresponda con la coordenada baricéntrica del punto respecto al triángulo no original del tetraedro del recubrimiento*. De esta forma, dicha coordenada baricéntrica nos proporcionará un mecanismo para decidir si un punto se encuentra a un lado u otro del plano soporte del triángulo no original del tetraedro del recubrimiento con respecto al centroide.

5.2. Descomposición mediante Tetra-Trees

Para evitar tener que realizar un test de colisión entre todos los pares de características de dos objetos poliédricos suelen utilizarse jerarquías de volúmenes envolventes o algún método de descomposición espacial como octrees [ABJN85] [Swa93] o *k*-d trees [HKM95] (según vimos en la Sección 2.3), y clasificar dichas

* Para que esto ocurra, basta con comenzar por el vértice origen del recubrimiento V_o (el centroide), de manera que el tetraedro se dará en la forma $V_oV_1V_2V_3$.

características en estas estructuras de datos. Además cuando tratamos con objetos no convexos, se suele realizar un pre-procesamiento que consiste en descomponer dichos objetos en partes convexas, de manera que en lugar de un único objeto, obtenemos un conjunto de partes convexas sobre las que realizar la DC.

Sin embargo, la mayor parte de estas estructuras de datos, o bien son difíciles de actualizar entre frames, o bien no se ajustan adecuadamente a los objetos a representar, aumentando en el primer caso el tiempo en determinar la colisión, por el coste añadido en la actualización de la estructura de datos utilizada; y aumentando este tiempo también en el segundo caso, debido sobre todo a que se descarta un menor numero de pares de características, debido a que se produce un peor ajuste de la estructura de datos utilizada a los objetos.

Por este motivo vamos a utilizar una descomposición espacial nueva denominada *Tetra-Tree*, basada en una subdivisión recursiva del espacio mediante *tetra-conos*. Un *Tetra-Cono* es una región del espacio delimitada por tres planos que intersecan en un punto común. Se puede definir mediante un tetraedro en el que uno de los vértices es el origen del tetra-cono; este vértice junto a los restantes definen los planos que delimitan dicho tetra-cono. Un tetra-tree estará formado por varios tetra-conos con un origen común, el centroide* del poliedro, de manera que estos tetra-conos cubran la totalidad del espacio sin solapamiento entre ellos, salvo en la frontera. Conforme se construye el tetra-tree, los tetraedros del recubrimiento del poliedro se clasifican en cada uno de sus tetra-conos. Cada tetra-cono se subdividirá en nuevos tetra-conos de forma recursiva, normalmente hasta una profundidad máxima en el árbol, o hasta que el número de tetraedros clasificados en un tetra-cono sea menor a un número mínimo preestablecido.

Utilizar esta descomposición nos permitirá tratar con una parte del recubrimiento del poliedro, sólo con los tetraedros clasificados en los tetra-conos involucrados. Además para cada tetra-cono incorporaremos tetraedros y esferas envolventes, de esta manera nos facilitará la clasificación de características y se simplificarán las operaciones a realizar.

Una de las ventajas que supone utilizar esta estructura de datos consiste en evitar tener que clasificar de nuevo los tetraedros del recubrimiento cuando se producen transformaciones rígidas. Además los volúmenes envolventes utilizados se ajustan de una manera más natural a la forma de los objetos, por lo que se obtienen algoritmos más eficientes que con otro tipo de volúmenes envolventes. El cálculo de un tetra-

* Salvo que se diga lo contrario, tanto el recubrimiento de un poliedro como su tetra-tree tendrán como origen el centroide del objeto. Además, el recubrimiento mediante triángulos de una cara tendrá como origen el centroide de dicha cara. Todas las definiciones y algoritmos desarrollados en este capítulo se basan en dicho tipo de recubrimiento.

tree es más rápido que otro tipo de estructuras de datos, como por ejemplo un octree, no necesitando mucho espacio de almacenamiento [JFS04] [JFSO06].

5.2.1. Definición de Tetra-Cono

Un cono tetraédrico o *Tetra-Cono* es una región del espacio delimitada por tres planos con un único punto en común. Utilizaremos un tetraedro para definir esa región, de manera que el vértice original del tetraedro sea ese punto en común, y el vértice original junto a cada uno de los dos vértices restantes del tetraedro definan cada uno de esos tres planos. La región delimitada es aquella en la que el tetraedro considerado está dentro de la misma.

Definición 5.2: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo. Definimos *Tetra-Cono* asociado al tetraedro T, que notaremos como $\angle$T, a la región del espacio delimitada por los planos soporte de $V_0V_1V_2$, $V_0V_2V_3$ y $V_0V_3V_1$, de manera que T es interior a dicha región (Figura 5.3).

Siempre utilizaremos tetraedros positivos (con volumen signado positivo) para definir un tetra-cono. El origen del tetra-cono (el punto en común de los tres planos) coincidirá siempre con el primer vértice del tetraedro, que para nuestros propósitos será el origen del recubrimiento de los objetos.

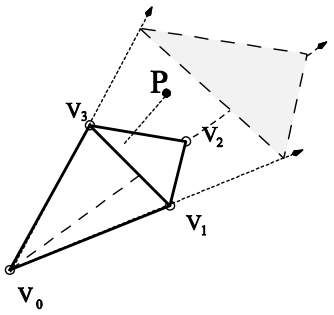


Figura 5.3: Definición de un tetra-cono. En este caso el punto *P* no está en el tetraedro $V_0V_1V_2V_3$ pero sí en el tetra-cono asociado a dicho tetraedro.

Notación: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo y $\angle$T el tetra-cono asociado. Notaremos que un punto *P* pertenece a $\angle$T como [*P in* $\angle$T].

Nótese que según las Propiedades 5.1 y 5.2, para todo punto *P* se cumple que [*P in* $\angle$T] sii $sign(\beta(P))\geq 0 \wedge sign(\gamma(P))\geq 0 \wedge sign(\delta(P))\geq 0$ respecto a T.

Definición 5.3: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo y $\angle$T el tetra-cono asociado. Definimos *tetra-cono truncado* asociado al tetraedro T, que notaremos $\angle\backslash$T, como la parte del tetra-cono $\angle$T delimitada por el plano soporte de $V_1V_2V_3$ de manera que V_0 es exterior a dicha región (Figura 5.4).

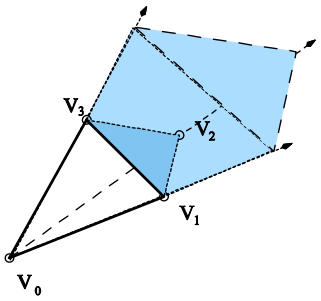


Figura 5.4: Tetra-cono truncado.

Notación: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo y $\angle\backslash$T el tetra-cono truncado asociado a T. Notaremos que un punto *P* pertenece a $\angle\backslash$T como [*P in* $\angle\backslash$T].

Del mismo modo que en el caso de un tetra-cono, según las Propiedades 5.1 y 5.2, para todo punto *P* se cumple que [*P in* $\angle\backslash$T] sii [*P in* $\angle$T] $\wedge$ $sign(\alpha(P))\leq 0$ respecto a T.

5.2.2. Definición de Tetra-Tree

Una vez definido el concepto de tetra-cono, construiremos una estructura jerárquica que descomponga el espacio para cada uno de los objetos considerados en la detección de colisión. Para ello crearemos un conjunto de tetra-conos con origen en el centroide del poliedro. Dichos tetra-conos se subdividirán recursivamente en nuevos tetra-conos, en los que clasificaremos los tetraedros del recubrimiento, o bien hasta alcanzar una profundidad máxima en el árbol, o bien hasta que el número de tetraedros clasificados en un tetra-cono esté por debajo de un umbral mínimo. Esta estructura forma un tetra-tree, que es una generalización del tri-tree definido en 2D.

Definición 5.4: Un árbol de tetra-conos o *Tetra-Tree* es una estructura de datos jerárquica en forma de árbol de cuyo nodo raíz parten ocho tetra-conos hijos ($\angle T_0,\angle T_1,\angle T_2,\angle T_3,\ \angle T_4,\angle T_5,\angle T_6,\angle T_7$), todos con origen común y que cubren la totalidad del espacio sin solapamientos entre ellos, salvo en la frontera. Cada uno de estos ocho tetra-conos de primer nivel se descompone en cuatro sub-tetra-conos hijos (Figura 5.5), de manera que el tetra-cono $\angle T_i$ tendrá cuatro tetra-conos de siguiente nivel ($\angle T_{i0},\angle T_{i1},\angle T_{i2},\angle T_{i3}$). Del mismo modo, cada tetra-cono se subdivide

recursivamente en cuatro sub-tetra-conos, hasta alcanzar un criterio de parada predefinido. Cada sub-tetra-cono se construye a partir del tetra-cono padre $\angle T_i$ definido por los vértices V_0, V_1, V_2, V_3 de la siguiente forma:

$$\angle T_{i0} = (V_0, V_1, (V_1+V_2)/2, (V_3+V_1)/2)$$
$$\angle T_{i1} = (V_0, (V_1+V_2)/2, V_2, (V_2+V_3)/2)$$
$$\angle T_{i2} = (V_0, (V_3+V_1)/2, (V_2+V_3)/2, V_3)$$
$$\angle T_{i3} = (V_0, (V_1+V_2)/2, (V_2+V_3)/2, (V_3+V_1)/2)$$

El criterio de parada que utilizaremos consiste en alcanzar una profundidad máxima en el árbol o que el número de tetraedros del recubrimiento clasificados en un tetra-cono sea menor o igual a un valor mínimo establecido de antemano.

Dada una figura poliédrica compleja es posible construir un tetra-tree formado por un conjunto de tetra-conos que la envuelva completamente, sin solapamientos entre los tetra-conos que forman un mismo nivel de profundidad en el árbol, salvo en su frontera. Este conjunto de tetra-conos no es único, pero cualquiera es válido para el caso que nos ocupa.

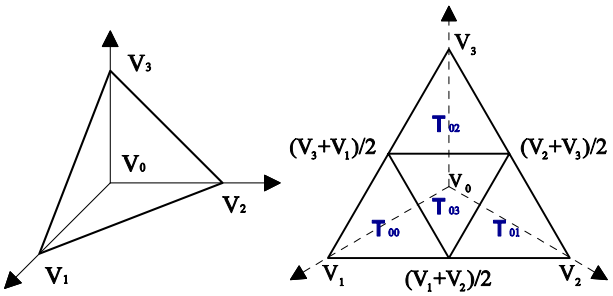


Figura 5.5: Un tetra-cono y sus cuatro sub-tetra-conos.

Para cada poliedro hemos creado inicialmente ocho tetra-conos de primer nivel, todos ellos con origen en el centroide del objeto, de manera que si realizamos una traslación del centroide al origen de coordenadas, cada uno de estos tetra-conos delimita uno de los ocho octantes en los que se divide el espacio. Por ejemplo, el tetra-cono $\angle T_0$ de primer nivel representa al octante ($x \geq 0$, $y \geq 0$, $z \geq 0$), y T_0 está formado por los vértices V_0=(0,0,0), V_1=(0,0,1), V_2=(1,0,0) y V_3=(0,1,0) (Figura 5.6). El resto de tetra-conos de primer nivel se construye de forma similar al caso expuesto para el tetra-cono $\angle T_0$, de manera que cada uno de ellos represente un octante y los vértices de los tetraedros estén situados sobre los ejes coordenados a una distancia unitaria del origen.

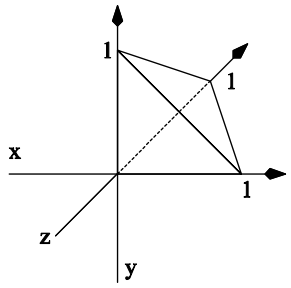


Figura 5.6: Tetra-cono que representa el octante (x≥0, y≥0, z≥0).

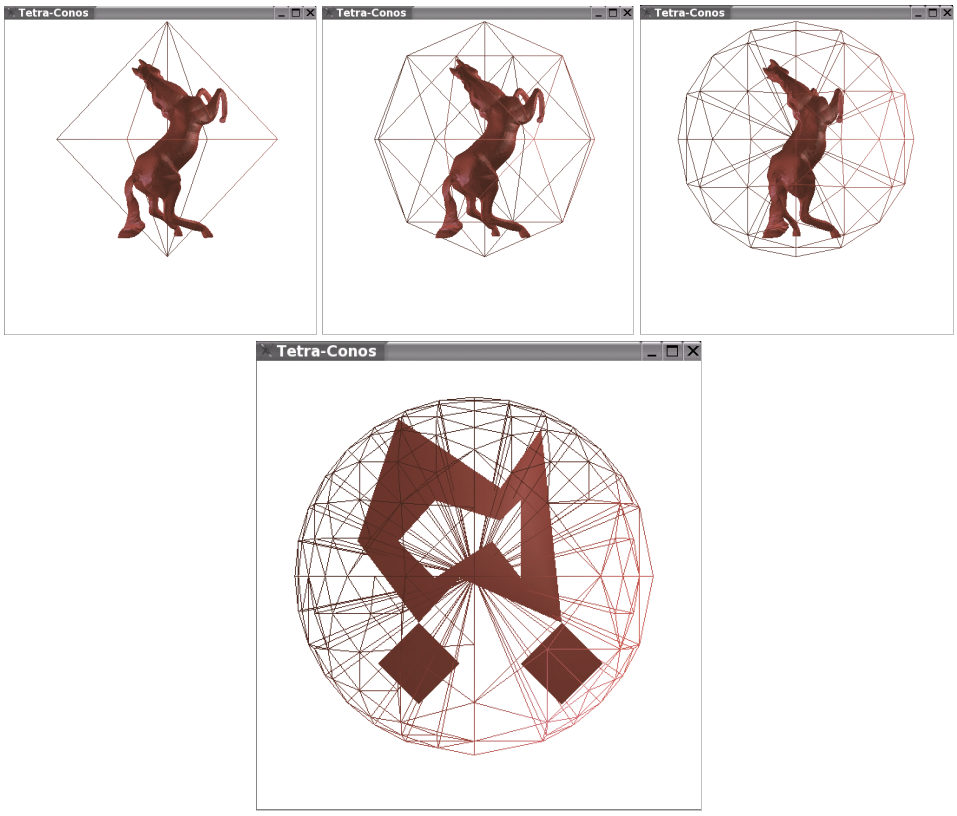


Figura 5.7: Descomposición de varios objetos utilizando tetra-trees.

Una vez definidos los ocho tetra-conos de primer nivel, se clasifican los tetraedros del recubrimiento en dichos tetra-conos. Cada tetra-cono se subdivide en cuatro nuevos sub-tetra-conos y se clasifican nuevamente los tetraedros del recubrimiento clasificados en el tetra-cono padre de cada uno de ellos, recursivamente y hasta alcanzar el criterio de parada. Como podemos ver, la construcción de los tetra-conos que forman el tetra-tree son una generalización del caso 2D.

En la Figura 5.7 se muestran los tetraedros asociados a cada tetra-cono de la descomposición mediante tetra-trees de una malla de triángulos (figura del "caballo") y de un objeto poliédrico complejo. En la figura del "caballo", se representan los tetraedros asociados a los tetra-trees de nivel 1, nivel 2 y nivel 3 (8, 32 y 128 divisiones del espacio en cada uno de los niveles). En la figura poliédrica compleja se representan los tetraedros asociados al tetra-tree de nivel 4 (512 subdivisiones). Podemos apreciar que en ciertas partes de dicha figura no se ha descendido a la profundidad máxima en el árbol, pues el número de tetraedros clasificados en determinados tetra-conos está por debajo del umbral mínimo preestablecido.

5.2.3. Criterios para la clasificación de tetraedros en un Tetra-Tree

En esta sección estableceremos una serie de criterios para obtener la clasificación de los tetraedros del recubrimiento de un poliedro en los tetra-conos del tetra-tree asociado a dicho poliedro. Debemos tener en cuenta que tanto los tetraedros del recubrimiento como los tetra-conos tienen origen en el centroide del objeto. En base a esto, clasificaremos los tetraedros en dos categorías básicas: tetraedros *no contenidos* en un tetra-cono (Figura 5.8.a), que serán aquellos tetraedros que no intersecan con el tetra-cono (salvo el vértice común del recubrimiento); y tetraedros *contenidos* en un tetra-cono, que serán aquellos tetraedros que intersecan con el tetra-cono.

Supongamos que tanto el tetra-tree como el recubrimiento del objeto tienen origen en el centroide del mismo. En este caso, las posibles situaciones en las que se puede encontrar un poliedro respecto a un tetra-cono son las siguientes:

Sea F un poliedro con recubrimiento C_F y tetra-tree asociado TT_F. Sea $S \in C_F$ y $\angle T \in TT_F$:

- **Definición 5.5:** Diremos que el tetraedro S *no está contenido* en el tetra-cono $\angle T$ cuando el único punto en común entre el tetraedro S y el tetra-cono $\angle T$ es el vértice original del tetra-cono $\angle T$ (Figura 5.8.a).
- **Definición 5.6:** Diremos que el tetraedro S *está totalmente contenido* en el tetra-cono $\angle T$ cuando el tetraedro S está dentro del tetra-cono $\angle T$ (Figura 5.8.b).
- **Definición 5.7:** Diremos que el tetraedro S *está parcialmente contenido* en el tetra-cono $\angle T$ cuando hay parte del tetraedro S que está dentro del tetra-cono $\angle T$ y parte del tetraedro S que está fuera del tetra-cono $\angle T$ (Figura 5.8.c-e).

Definición 5.8: Dado un tetraedro S y un tetra-cono ∠T, diremos que S está *clasificado* en ∠T, y lo notaremos como [S *clasif* ∠T], a la situación en la que el tetraedro S está total o parcialmente contenido en el tetra-cono ∠T.

Los casos en los que parte de la frontera de un tetraedro (una arista o una cara) interseca con parte de la frontera de un tetra-cono*, son considerados como parcial o totalmente contenidos, según sea el caso, dependiendo de si el resto del tetraedro se encuentra en el exterior del tetra-cono o en el interior del mismo respectivamente. La idea básica consiste en clasificar en el tetra-cono todos los tetraedros que tienen alguna parte en común con el tetra-cono.

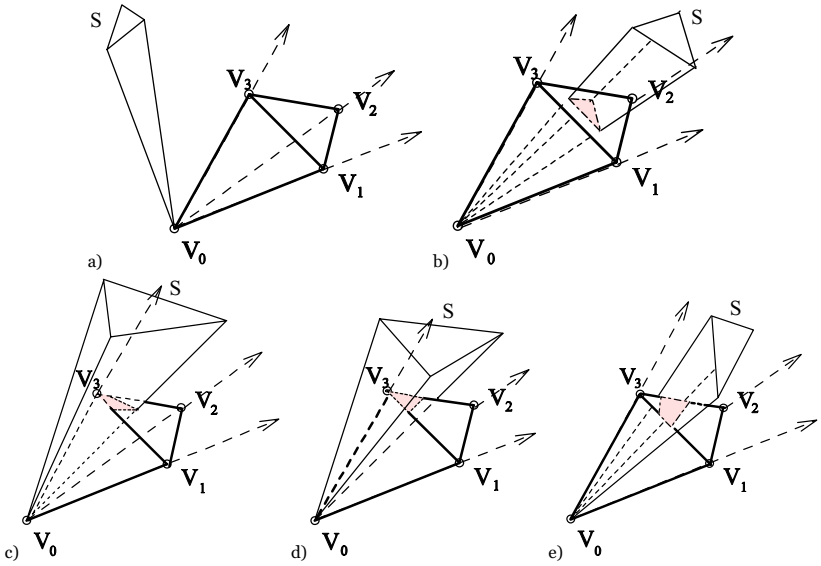


Figura 5.8: Relación tetraedro/tetra-cono ∠T, T=$V_0V_1V_2V_3$. a) Tetraedro no contenido en ∠T. b) Tetraedro totalmente contenido en ∠T. c),d),e) Tetraedros parcialmente contenidos en ∠T.

Podemos utilizar las siguientes propiedades para obtener eficientemente los tetraedros del recubrimiento de un poliedro que están clasificados en un tetra-cono:
Sea F un poliedro con recubrimiento C_F y tetra-tree asociado TT_F. Sea $S \in C_F$ y $\angle T \in TT_F$:

* Podemos considerar que la frontera de un tetra-cono está formada por tres tri-conos con un origen común.

- **Propiedad 5.3:** Si alguno de los vértices no originales de S se encuentra en el tetra-cono $\angle$T, entonces [S *in* $\angle$T].

  Esta propiedad contempla el caso de tetraedros totalmente contenidos (Figura 5.8.b) y parcialmente contenidos con algún vértice en el tetra-cono (Figura 5.8.c).

- **Propiedad 5.4:** Si algún vértice no original del tetraedro T se encuentra en el tetra-cono formado por $\angle$S, entonces [S *in* $\angle$T].

  Es decir, si intercambiamos los papeles entre el tetraedro a clasificar y el tetra-cono obtenemos los tetraedros que intersecan con el tetra-cono para el caso contemplado en la Figura 5.8.d.

- **Propiedad 5.5:** Si alguna arista no original del tetraedro S interseca con algún tri-cono de la frontera del tetra-cono, entonces [S *in* $\angle$T].

  Esta situación contempla el caso de la Figura 5.8.e.

Estas propiedades nos proporcionan un mecanismo completo para clasificar un tetraedro respecto a un tetra-cono, por lo que no es necesario realizar comprobaciones adicionales para obtener dicha clasificación. Estas propiedades deben comprobarse en el orden expuesto, para poder clasificar los tetraedros de una forma eficiente y rápida. Cuando se han clasificado todos los tetraedros del recubrimiento en los tetra-conos de un determinado nivel pasan a clasificarse en los tetra-conos del siguiente, evidentemente restringiendo el conjunto de tetraedros a clasificar solamente a aquellos que están contenidos en el tetra-cono padre.

5.2.4. Intersección segmento/tetra-cono

Una de las situaciones en las que es necesario estudiar si se produce intersección entre un segmento y un tetra-cono tiene lugar en la clasificación de tetraedros del recubrimiento respecto a los tetra-conos de un tetra-tree (Propiedad 5.5).

Podemos utilizar la noción de coordenadas baricéntricas de un punto respecto a un tetraedro (Definición 5.1) para determinar si se produce o no intersección entre una arista y cada uno de los tri-conos que forman parte de la frontera del tetra-cono.

Propiedad 5.6: Dado un tetra-cono $\angle$T y un segmento P_1P_2, el segmento P_1P_2 interseca con $\angle$T si y solamente si interseca con alguno de los tri-conos que forman parte de la frontera del tetra-cono (Figura 5.9), o los dos extremos del segmento P_1P_2 se encuentran dentro del tetra-cono.

Si seguimos los pasos dados en la sección anterior a la hora de clasificar un tetraedro del recubrimiento del poliedro respecto de un tetra-cono, no será necesario comprobar si los extremos de un segmento se encuentran dentro del tetra-cono, pues esto ya se hace al aplicar las Propiedades 5.3 y 5.4 a dichos vértices.

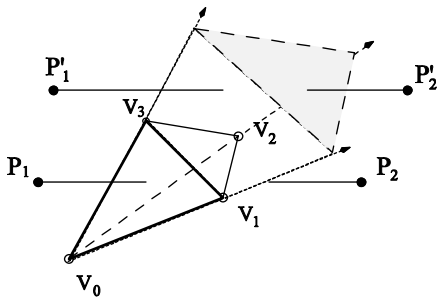


Figura 5.9: Intersección Segmento/Tetra-Cono.

En estas condiciones podemos establecer el siguiente teorema para determinar si se produce intersección entre un segmento y un tri-cono de la frontera de un tetra-cono. Este teorema da soporte a la determinación de intersección entre un segmento y un tetra-cono, si se aplica a cada uno de los tres tri-conos que forman la frontera del tetra-cono.

Teorema 5.1: Sea $\angle T$ el tetra-cono asociado al tetraedro T con vértices $V_0V_1V_2V_3$. Sea P_1P_2 un segmento tal que P_1 no se encuentra sobre el plano soporte de $V_0V_1V_3$. Entonces el segmento P_1P_2 interseca con el tri-cono $\angle V_0V_1V_3$ de la frontera del tetra-cono $\angle T$ sii las coordenadas baricéntricas de P_2 respecto al tetraedro $P_1V_0V_1V_3$ cumplen que $sign(\alpha(P_2))\leq 0 \wedge sign(\gamma(P_2))\geq 0 \wedge sign(\delta(P_2))\geq 0$, siendo indiferente el valor de $sign(\beta(P_2))$ (Figura 5.10).

Demostración: Los puntos P_1 y P_2 están situados cada uno a un lado del plano soporte de $V_0V_1V_3$ debido a que $sign(\alpha(P_2))\leq 0$. Además el punto de intersección debe estar en el mismo lado que V_3 respecto al plano soporte de $P_1V_0V_1$ o sobre ese mismo plano, pues $sign(\gamma(P_2))\geq 0$, y en el mismo lado que V_1 respecto al plano soporte de $P_1V_3V_0$ o sobre ese mismo plano, pues $sign(\delta(P_2))\geq 0$. Respecto a la cara $P_1V_1V_3$ no es necesaria ninguna condición, pues el punto de intersección puede estar en cualquiera de los dos lados del plano soporte de $P_1V_1V_3$ o sobre el mismo plano, al no imponer ninguna restricción sobre el valor de $sign(\beta(P_2))$. □

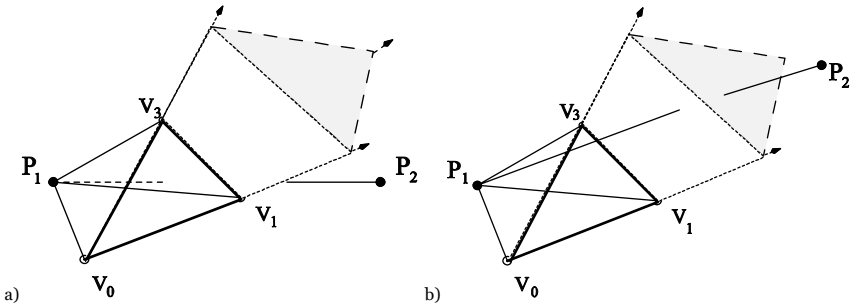


Figura 5.10: Construcción de un tetraedro auxiliar para la intersección entre un segmento y un tri-cono de la frontera de un tetra-cono. a) El segmento interseca con la cara del tetraedro que define el tetra-cono. b) El segmento no interseca con dicha cara, pero interseca con el tri-cono de la frontera del tetra-cono.

Para comprobar la intersección con cada uno de los dos tri-conos restantes de la frontera del tetra-cono podemos utilizar un planteamiento similar, formando tetraedros auxiliares con cada uno de triángulos originales del tetraedro utilizado en la definición del tetra-cono.

En el caso de que el extremo del segmento con el que se forma el tetraedro auxiliar estuviese sobre el plano soporte de $V_0V_1V_3$, sería necesario utilizar el otro extremo del segmento para formar dicho tetraedro auxiliar. Sólo en el caso de que este punto también se encuentre sobre dicho plano soporte, sería necesario realizar un test bidimensional, para lo cual habría que proyectar el tri-cono $\angle V_0V_1V_3$ y el segmento P_1P_2 sobre uno de los planos coordenados y comprobar si se produce intersección entre el segmento y el tri-cono. Este test consiste en primer lugar en comprobar si alguno de los vértices del segmento P_1P_2 está en el tri-cono $\angle V_0V_1V_3$ pero en 2D. Si ninguno de los extremos de P_1P_2 está en el tri-cono $\angle V_0V_1V_3$, se forma un nuevo tri-cono $\angle V_0P_1P_2$ y se comprueba si V_1 ó V_3 se encuentran en $\angle V_0P_1P_2$.

5.2.5. Tetraedro envolvente asociado a un Tetra-Cono

En esta sección vamos a definir el concepto de *tetraedro envolvente* asociado a un tetra-cono. Este tetraedro tendrá el mismo vértice original que el tetra-cono y cada una de sus tres caras originales estarán situadas sobre cada uno de los tri-conos de la frontera del tetra-cono. Este tetraedro envolvente pondrá un límite superior al tetra-cono de manera que todos los vértices de los tetraedros del recubrimiento del poliedro clasificados en el tetra-cono tendrán coordenada baricéntrica α positiva o cero respecto a dicho tetraedro envolvente (Figura 5.11.a). Así cualquier consulta, ya sea de inclusión o interferencia, afectará exclusivamente a la parte del poliedro incluida en el tetraedro envolvente.

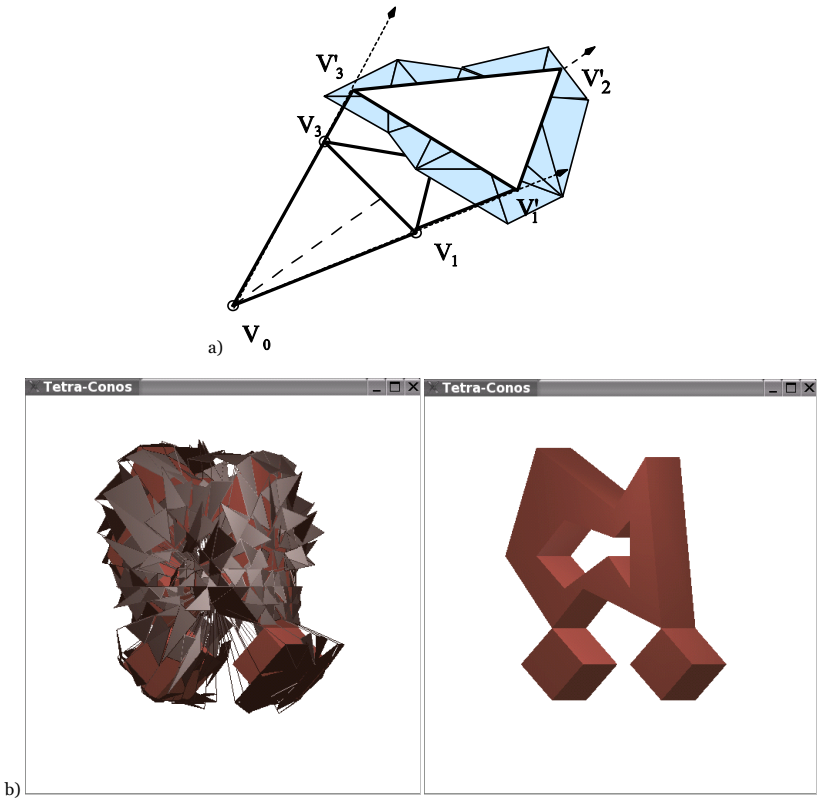


Figura 5.11: a) Tetraedro envolvente $T_{env}=V_oV'_1V'_2V'_3$ asociado a un tetra-cono $\angle T$, y triángulos no originales de los tetraedros del recubrimiento de un objeto clasificados en $\angle T$. b) Tetraedros envolventes de nivel 5 de una figura compleja junto a la figura original.

Definición 5.9: Sea F un poliedro con tetra-tree asociado TT_F. Sea $T=V_oV_1V_2V_3$ un tetraedro tal que $\angle T \in TT_F$. Definimos *tetraedro envolvente* de la geometría de F contenida en $\angle T$ como el tetraedro $T_{env}=V_oV'_1V'_2V'_3$ tal que (Figura 5.11):

1. V'_1 está sobre la semi-recta definida por V_oV_1.
2. V'_2 está sobre la semi-recta definida por V_oV_2.
3. V'_3 está sobre la semi-recta definida por V_oV_3.
4. Todo vértice V_i de los tetraedros del recubrimiento de F clasificados en $\angle T$ cumplen que $sign(\alpha(V_i)) \geq 0$ respecto a T_{env}.

Dado un tetra-cono, existe más de un posible tetraedro envolvente, por lo que utilizaremos un tetraedro envolvente que se ajuste en la medida de lo posible a los tetraedros del recubrimiento clasificados en su tetra-cono asociado. Se ha utilizado

un algoritmo que ajusta dicho tetraedro envolvente a la geometría del poliedro, similar al empleado en 2D (Sección 4.5.3) pero extendido a tres dimensiones.

Para clasificar un tetraedro del recubrimiento de un poliedro respecto a un tetraedro envolvente utilizaremos la coordenada baricéntrica α de los vértices de dicho tetraedro respecto al tetraedro envolvente.

Definición 5.10. Sea F un poliedro con tetra-tree asociado TT_F. Sea $S=V_0V_1V_2V_3$ un tetraedro tal que $S \in C_F$. Sea $\angle T$ un tetra-cono tal que $\angle T \in TT_F$, y sea T_{env} un tetraedro envolvente de la geometría de F contenida en $\angle T$. Diremos que S está *clasificado* en T_{env}, y lo notaremos como [S *clasif* T_{env}], si se cumple que [S *clasif* $\angle T$] y además $sign(\alpha(V_i)) \geq 0$, $i=0..3$.

Es posible que el tetraedro envolvente asociado a un tetra-cono no contenga a todos los vértices de los tetraedros del recubrimiento clasificados en él, pues existen vértices que pueden pertenecer a tetraedros parcialmente contenidos en el tetra-cono. Por este motivo hemos utilizado la coordenada baricéntrica α para delimitar el tetraedro envolvente (apartado 4 de la Definición 5.9), pues aunque existen vértices no incluidos en el tetra-cono, si sólo consideramos los vértices incluidos en el mismo, parte de la geometría del poliedro podría quedar fuera del tetraedro envolvente. De este modo nos aseguramos que el tetraedro envolvente lo sea de toda la geometría del poliedro incluida en un tetra-cono (Figura 5.12).

Nótese que si utilizamos para obtener el tetraedro envolvente la distancia del vértice más lejano de los tetraedros clasificados en el tetra-cono al centroide, éste tetraedro no se ajustaría demasiado a la geometría del poliedro. Esta situación se da cuando no se itera el algoritmo de ajuste señalado anteriormente, como puede apreciarse en la Figura 5.13 y 5.14.

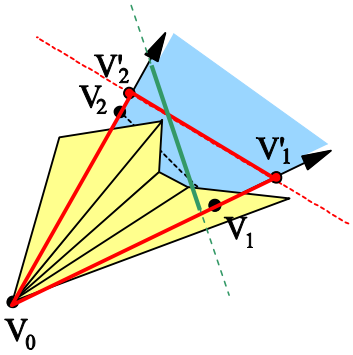


Figura 5.12: Vista en 2D de la geometría del poliedro contenida en un tetraedro envolvente (en rojo). Si utilizáramos solo los vértices contenidos en el tetra-cono, parte de la geometría del poliedro se quedaría fuera de ese posible tetraedro envolvente (en verde).

Figura 5.13: Tetraedros envolventes (niveles 1 al 6) sin iterar el algoritmo de ajuste.

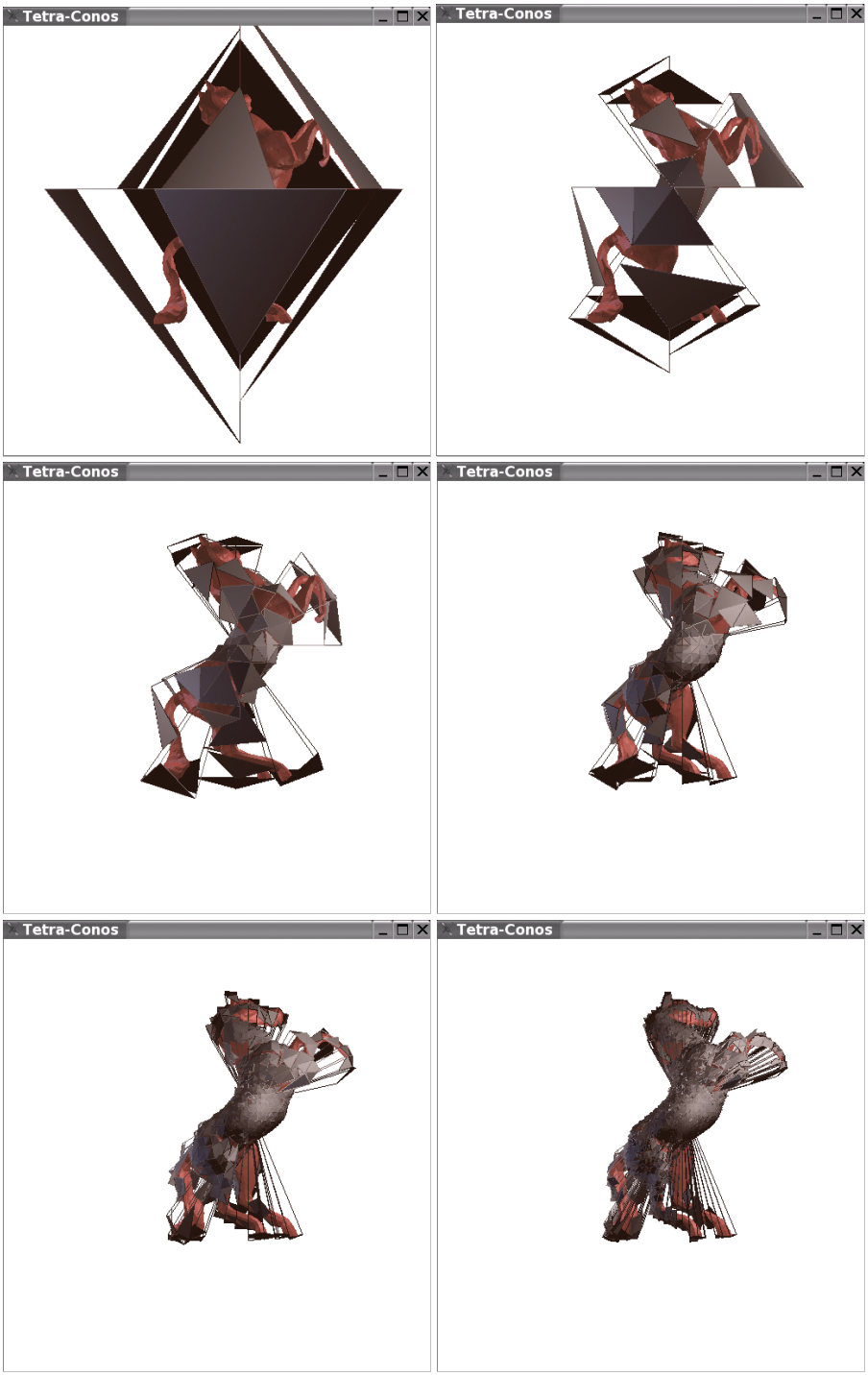


Figura 5.14: Tetraedros envolventes (niveles 1 al 6) iterando el algoritmo de ajuste.

5.2.6. Envolvente regular de un poliedro

A continuación vamos a definir el concepto de *envolvente regular* de un poliedro, de manera que nos ayude a descartar tetraedros envolventes en la colisión entre poliedros de una forma más rápida, aprovechando que esta envolvente es regular y convexa. Dicha envolvente regular estará formada por una serie de tetraedros envolventes asociados a los tetra-conos de un nivel del tetra-tree. Tendremos por tanto una envolvente regular por cada nivel de profundidad del tetra-tree correspondiente.

Definición 5.11: Sea F un poliedro con tetra-tree asociado TT_F. Sea TT_{env} el conjunto de tetraedros envolventes asociados a un nivel de TT_F. Sea d_{max} el máximo de las distancias entre V_o y cada uno de los vértices V'_1, V'_2 y V'_3 de todos los tetraedros de TT_{env}. Definimos la *envolvente regular* de F, $TT_{env\text{-}reg}$, como el conjunto de tetraedros envolventes resultantes de substituir en cada tetraedro envolvente $T_{env} \in TT_{env}$ los puntos V'_1, V'_2 y V'_3 por V''_1, V''_2 y V''_3 situados sobre las semi-rectas $V_oV'_1$, $V_oV'_2$ y $V_oV'_3$ respectivamente y a una distancia d_{max} de V_o.

Notación: Dado un poliedro F con envolvente regular $TT_{env\text{-}reg}$, notaremos como $T_{env\text{-}reg}$ a un tetraedro envolvente que forme parte de la envolvente regular $TT_{env\text{-}reg}$.

Intuitivamente, la definición de envolvente regular nos permite obtener para cada nivel del tetra-tree de un poliedro, un conjunto de tetraedros envolventes cuyos triángulos no originales forman una envolvente convexa del poliedro (Figura 5.15).

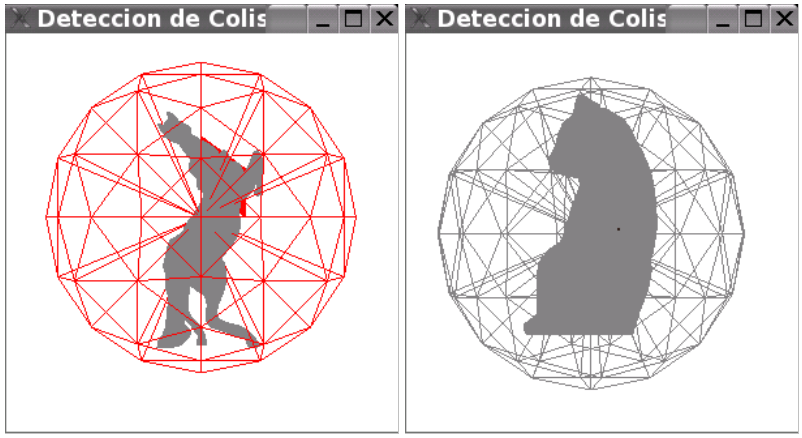


Figura 5.15: Tetraedros envolventes que forman parte de una envolvente regular.

5.2.7. Esfera envolvente asociada a un Tetra-Cono

Además de utilizar un tetraedro envolvente asociado a cada tetra-cono, usaremos también una esfera envolvente. Cada esfera envolvente contiene a los vértices no originales de los tetraedros del recubrimiento clasificados en un tetra-cono. Mediante estas esferas envolventes descartaremos también tetraedros envolventes sobre los que realizar un test de intersección en la DC entre poliedros. Esto es debido a que si no intersecan dichas esferas envolventes no intersecarán los triángulos no originales de los tetraedros clasificados en el tetra-cono correspondiente, con lo que no será necesario comprobar la intersección entre sus tetraedros envolventes, que es un test más costoso que el test de intersección entre esferas.

Definición 5.12: Dado un poliedro F y un tetra-cono $\angle T \in TT_F$, definimos la *esfera envolvente* de los vértices no originales de los tetraedros del recubrimiento de F clasificados en dicho tetra-cono $\angle T$, a la esfera $E_{env}(C,r)$ tal que C es el centroide de dichos vértices y r es el máximo de las distancias de C a cada uno de los vértices. (Figura 5.16 y 5.17).

Esta esfera envolvente junto con el tetraedro envolvente asociado al tetra-cono son utilizados para acotar la geometría incluida en dicho tetra-cono, de manera que podemos considerar que el volumen tratado es la intersección entre el tetraedro envolvente y la esfera. Por ejemplo, en el algoritmo de DC entre poliedros, en primer lugar se comprueba la intersección de las esferas envolventes, y de producirse, la intersección entre tetraedros envolventes.

Este volumen resultante se ajusta en mayor grado a la geometría del poliedro, pues normalmente los triángulos de los tetraedros del recubrimiento que forman la frontera del poliedro se encuentran próximos al triangulo no original del tetraedro envolvente, dejando mucho espacio libre entre el origen del tetraedro y la frontera de dicho poliedro.

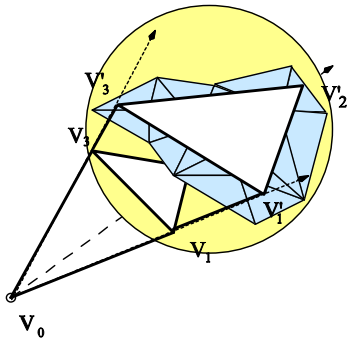


Figura 5.16: Esfera envolvente de los triángulos no originales clasificados en un tetra-cono.

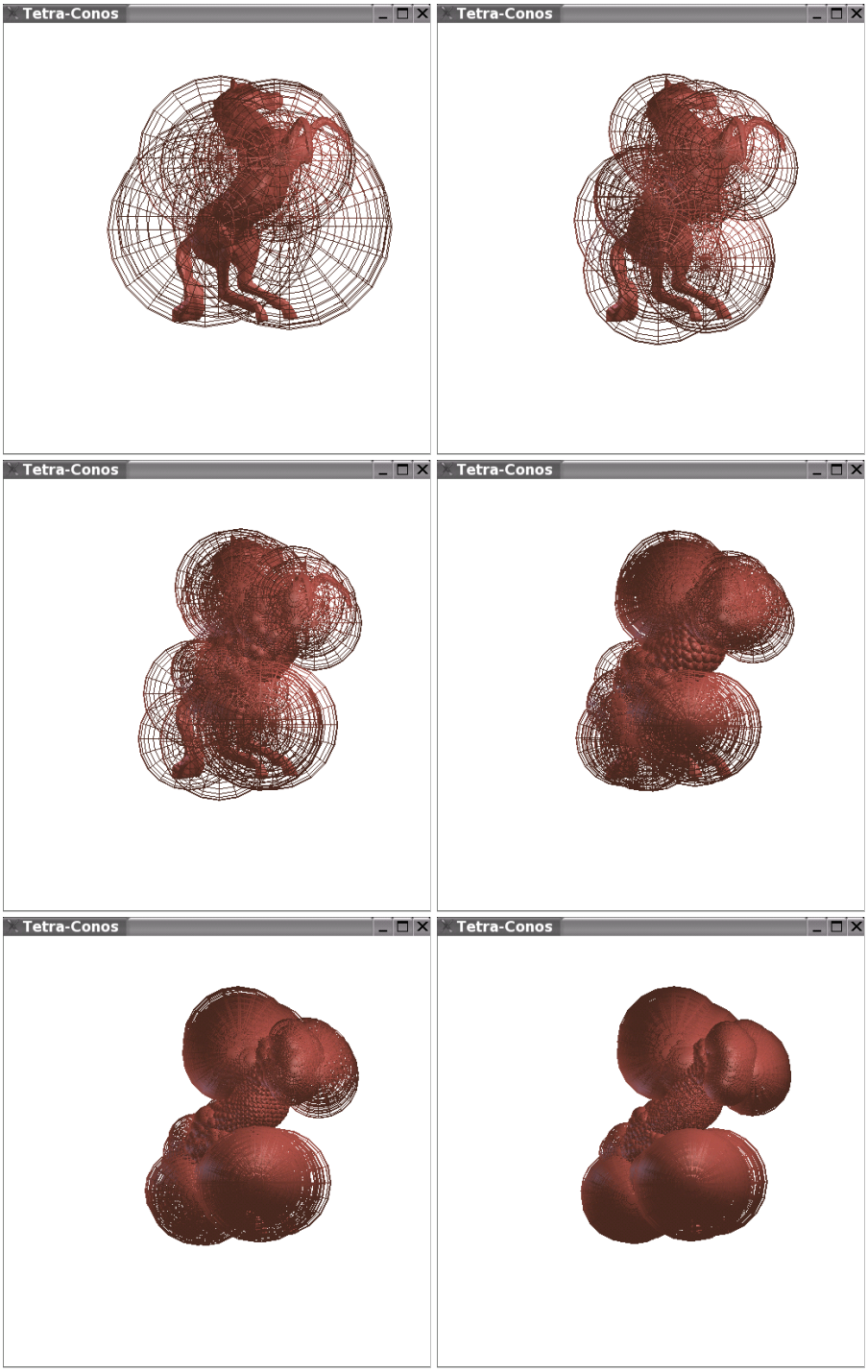


Figura 5.17: Esferas envolventes asociadas a un tetra-cono. Niveles 1 al 6.

5.3. Algoritmo de Detección de Colisión Punto/Poliedro

Son diversas las aplicaciones que posee un sistema de detección de colisiones entre un punto y un poliedro. Entre otras podemos destacar las simulaciones de sistemas de partículas, los entornos inmersivos y las aplicaciones de realidad virtual. Un algoritmo de DC entre un punto y un poliedro puede utilizarse en la DC entre poliedros, para ello, antes de realizar un test detallado de colisión, se pueden situar puntos en lugares estratégicos de ambos poliedros, y aplicarles dicho algoritmo para descartar rápidamente una posible colisión entre objetos.

A lo largo de esta sección definiremos algunos conceptos que nos permitirán desarrollar un algoritmo de detección de colisión entre un punto y un poliedro. Muchos de estos conceptos se utilizarán o extenderán a algoritmos posteriores, como el de detección de colisión entre una esfera y un poliedro, o el algoritmo de detección de colisión entre poliedros.

La idea básica del algoritmo de detección de colisión entre un punto y un poliedro consiste en clasificar el punto en el tetra-tree del poliedro, y obtener un tetra-cono de último nivel en el que se encuentra dicho punto. La búsqueda comenzará en el tetra-cono en el que se encontraba el punto en el frame anterior, y si no se encuentra en éste, partiremos del nodo raíz del tetra-tree y buscaremos recursivamente el tetra-cono de último nivel en el que se encuentre el punto. Una vez hallado el tetra-cono, se determina si el punto está en su tetraedro envolvente. Si está en su interior, utilizaremos la coherencia para disminuir el número de cálculos necesarios para obtener la inclusión del punto en los tetraedros del recubrimiento clasificados en el tetra-cono.

Comenzaremos por describir cómo utilizar la coherencia para restringir los cálculos a determinados tetraedros del recubrimiento haciendo uso de los tetra-trees, y para localizar el tetra-cono en el que se encuentra el punto buscando en primer lugar en los tetra-conos utilizados en el frame anterior.

5.3.1. Coherencia espacial

En esta sección vamos a describir una serie de propiedades para la detección de colisión entre un punto y un poliedro complejo. Estas propiedades nos permiten utilizar el tetra-tree asociado a un objeto de manera que podamos restringir los cálculos de colisión a los tetraedros del recubrimiento clasificados en un tetra-cono, en lugar de utilizar todos los tetraedros del recubrimiento. Estas propiedades nos permitirán descartar tetraedros de manera que podamos utilizar el algoritmo de

inclusión de puntos en poliedros sólo con los tetraedros clasificados en un determinado tetra-cono, aquél en el que se encuentra el punto en cuestión.

Definición 5.13: Dado un poliedro F con tetra-tree asociado TT_F, un punto *P* y un tetra-cono $\angle T$ tal que $\angle T \in TT_F$, diremos que el punto *P se encuentra en la parte del recubrimiento de* F *clasificada en el tetra-cono* $\angle T$, y lo notaremos como [*P in* $C_{\angle T}$], si se cumple que *P* está en el tetra-cono $\angle T$ y además *P* está incluido en la parte del poliedro representada por los tetraedros del recubrimiento clasificados en el tetra-cono $\angle T$.

Veamos las propiedades que nos permiten utilizar la coherencia espacial:

- **Propiedad 5.7:** Sea F un poliedro y TT_F su tetra-tree asociado. Sea $\angle T$ un tetra-cono tal que $\angle T \in TT_F$. Sea *P* un punto tal que [*P in* $\angle T$]. Entonces [*P in* F] si y solamente si [*P in* $C_{\angle T}$].

  Esto es así debido a que si un punto se encuentra en un determinado tetra-cono, sólo puede estar contenido en los tetraedros del recubrimiento clasificados en dicho tetra-cono. Para probar esto podemos descomponer el recubrimiento del poliedro en dos conjuntos de tetraedros, uno el de los tetraedros que están clasificados en el tetra-cono y otro el del resto de tetraedros (aquellos que hemos definido como no contenidos en el tetra-cono). Como el punto se encuentra en el tetra-cono, no puede haber inclusión en ningún tetraedro no contenido en dicho tetra-cono, por tanto la inclusión sólo puede darse en tetraedros clasificados en ese tetra-cono.

- **Propiedad 5.8:** Sea F un poliedro y TT_F su tetra-tree asociado. Entonces para todo punto *P* existe un tetra-cono de primer nivel $\angle T \in TT_F$ tal que [*P in* $\angle T$].

  Siempre existe un tetra-cono de primer nivel perteneciente al tetra-tree de un poliedro en el cual podemos clasificar cualquier punto del espacio. Esto es evidente debido a que el tetra-tree de un objeto cubre todo el espacio tridimensional. Los tetra-conos que lo componen en un determinado nivel cubren siempre la totalidad del espacio sin solapamientos entre ellos, salvo en la frontera.

- **Propiedad 5.9:** Sea F un poliedro y TT_F su tetra-tree asociado. Sea $\angle T_i$ un tetra-cono tal que $\angle T_i \in TT_F$ y que $\angle T_i$ no es un nodo hoja de TT_F. Sea *P* un punto que cumple que [*P in* $\angle T_i$]. Entonces existe un tetra-cono $\angle T_j \in TT_F$ de siguiente nivel e hijo de $\angle T_i$, que cumple que [*P in* $\angle T_j$].

> Haciendo uso de esta propiedad podemos descender por el subárbol de un tetra-cono y obtener un tetra-cono de menor tamaño, presumiblemente con menor número de tetraedros del recubrimiento clasificados en él. Como un tetra-cono se subdivide en cuatro tetra-conos que lo contienen completamente y sin solapamientos, salvo en la frontera, un punto clasificado en un tetra-cono estará contenido en uno de sus sub-tetra-conos hijos.

En el caso de que un punto se encuentre en la frontera de un tetra-cono, el punto podría clasificarse en más de un tetra-cono del mismo nivel, en todos aquellos que comparten esa frontera. En realidad no importa el tetra-cono que escojamos, pues en todos los tetra-conos que comparten la frontera se han clasificado los mismos tetraedros del recubrimiento, por lo que la inclusión será la correcta utilizando cualquiera de estos tetra-conos. El criterio que hemos utilizado para elegir el tetra-cono consiste en utilizar el primer tetra-cono en el que detectemos la inclusión del punto.

Podemos decir que, gracias a la coherencia espacial y a la descomposición realizada mediante tetra-conos y tetra-trees, podemos substituir el término "inclusión en los tetraedros del recubrimiento de un poliedro" por su equivalente "inclusión en los tetraedros del recubrimiento clasificados en un tetra-cono" siempre que el punto se encuentre en dicho tetra-cono. Por este motivo, las definiciones y propiedades de la siguiente sección sirven tanto para el recubrimiento completo de un poliedro como para la parte del recubrimiento clasificada en un tetra-cono.

5.3.2. Coherencia temporal

Vamos a estudiar una serie de propiedades que nos permitirán utilizar la coherencia temporal en los algoritmos desarrollados para la detección de colisión entre un punto y un poliedro. Por una parte, para determinar sobre qué elementos buscar en primer lugar en base a colisiones anteriores, y por otra, para determinar si el estado de colisión es el mismo en el frame actual que en el frame previo, debido a que el punto se encuentra en determinadas zonas espaciales.

Mediante la siguiente heurística aprovecharemos la coherencia temporal para, en un frame determinado, comprobar en primer lugar si el punto se encuentra en el mismo tetra-cono en el que se encontraba el punto en el frame anterior. Esta situación, gracias a la coherencia temporal, se producirá en repetidas ocasiones, no siendo necesario descender recursivamente por el tetra-tree para encontrar el tetra-cono en el que se encuentra el punto. Evidentemente es necesario comprobar si el punto está en dicho tetra-cono para localizar el tetra-cono correcto en caso de que no sea así.

Heurística 5.1: Dado un poliedro F, su tetra-tree asociado TT_F, un tetra-cono $\angle T \in TT_F$ y un punto *P* incluido en dicho tetra-cono en un frame determinado, $[P \text{ in } \angle T]_j$, entonces, en presencia de un alto grado de coherencia temporal, lo más probable es que el punto siga estando en el mismo tetra-cono en el siguiente frame, es decir $[P \text{ in } \angle T]_{j+1}$.

Gracias a esta heurística podemos guardar el tetra-cono en el que se encontraba el punto en el frame anterior, para en el siguiente comenzar buscando en primer lugar en dicho tetra-cono.

Las siguientes propiedades sirven tanto para poliedros descompuestos mediante tetra-trees como para poliedros completos sobre los que no se ha realizado dicha descomposición, debido sobre todo a las propiedades sobre coherencia espacial expuestas en la sección anterior. Por tanto, todas estas propiedades, que expresaremos en relación al poliedro completo, podemos restringirlas al tetra-cono en el que se encuentra el punto y a los tetraedros clasificados en dicho tetra-cono.

Veamos en primer lugar un lema que nos permite obtener la inclusión de un punto en un poliedro en base al estado de inclusión anterior y al valor de la coordenada baricéntrica α del punto respecto a los tetraedros del recubrimiento, sin que sea necesario obtener el resto de las coordenadas baricéntricas.

Lema 5.1: Sea *P* un punto cualquiera y F un poliedro con recubrimiento $C_F=\{T_i\}$ / $i=1..n$. Si se cumple que $[not\ (P\ in\ F)]_j$ y que $[sign(\alpha_i(P))]_j=[sign(\alpha_i(P))]_{j+1}$ para todo $i=1..n$, entonces $[not\ (P\ in\ F)]_{j+1}$. De igual modo, si se cumple que $[P\ in\ F]_j$ y que $[sign(\alpha_i(P))]_j= [sign(\alpha_i(P))]_{j+1}$ para todo $i=1..n$, entonces $[P\ in\ F]_{j+1}$.

Demostración: Para que un punto *P* que se encuentra fuera de un poliedro en un frame determinado, pase a estar dentro del poliedro en el siguiente frame, debe atravesar una de las caras que forman el poliedro. Esto implica un cambio de signo (de negativo a positivo o cero) de la coordenada baricéntrica α respecto a uno de los tetraedros del recubrimiento de esa cara. Por este motivo, si no se produce ningún cambio de signo de la coordenada α respecto a ningún tetraedro del recubrimiento, el punto no habrá entrado en el mismo y permanecerá fuera.

Del mismo modo, para que un punto que está dentro de un poliedro pase a estar fuera, debe haber un cambio de signo de α respecto a alguno de los tetraedros del recubrimiento (un cambio de positivo a cero o negativo). Si no se produce ningún cambio de signo el punto permanecerá dentro.

Podemos considerar el caso especial en el que el punto se mueva por el plano soporte de una cara del poliedro, incluyendo la situación en la que se mueva por una arista de

la cara. En este caso, si el punto se encuentra fuera (o dentro en un segundo caso) en un momento, para pasar a dentro (o fuera en el segundo caso) del poliedro debe atravesar otra de las caras del poliedro, con el consiguiente cambio de signo respecto a otro tetraedro del recubrimiento, por lo que el planteamiento anterior es válido también para esta situación. □

Según este lema, es necesario que se produzca un cambio en el signo de α para alguno de los tetraedros del recubrimiento cuando el punto cambia su estado de inclusión respecto al poliedro. Por tanto, podemos guardar el signo de α respecto a todos los tetraedros del recubrimiento en el frame anterior y compararlo con el signo en el frame actual. Si no se produce ningún cambio de signo, el estado de inclusión actual del punto será el mismo que el estado anterior. De este modo, dado un tetraedro T_i del recubrimiento, no será necesario calcular las coordenadas baricéntricas β_i, γ_i y δ_i cuando no cambie el signo de α_i.

Gracias a la Propiedad 5.2, si cualquiera de las coordenadas baricéntricas de un punto respecto del tetraedro es negativa, el punto se encuentra fuera del tetraedro. Esto nos lleva a calcular β sólo cuando $\alpha \geq 0$, γ sólo cuando $\beta \geq 0$, y δ sólo cuando $\gamma \geq 0$.

A partir del Lema 5.1 y de la Propiedad 5.1 podemos deducir las siguientes propiedades que nos permiten aprovechar la coherencia de forma similar:

Propiedad 5.10: Sea F un poliedro con recubrimiento $C_F=\{T_i\}/\ i=1..n$. Para todo punto P tal que $[not\ (P\ in\ F)]_j$ y $[P\ in\ F]_{j+1}$, existe algún $k \in [1,n]$ tal que $[\alpha_k(P) < 0]_j$ y $[\alpha_k(P) \geq 0]_{j+1}$.

Propiedad 5.11: Sea F un poliedro con recubrimiento $C_F=\{T_i\}/\ i=1..n$. Para todo punto P tal que $[P\ in\ F]_j$ y $[not\ (P\ in\ F)]_{j+1}$, existe algún $k \in [1,n]$ tal que $[\alpha_k(P) \geq 0]_j$ y $[\alpha_k(P) < 0]_{j+1}$.

Según estas nuevas propiedades, cuando un punto que se encontraba fuera del poliedro en un frame entra en el poliedro en el siguiente frame, se produce un cambio de negativo a cero o positivo en el signo de α respecto a algún tetraedro del recubrimiento. Podemos aprovechar esta propiedad en el algoritmo de detección de colisión desarrollado en esta sección. Para esto guardamos en una máscara de bits si se cumple la relación $\alpha_i(P) \geq 0$ para todos los tetraedros $T_i \in C_F$, $i=1..n$. Cada bit b_i valdrá 1 si $\alpha_i(P) \geq 0$ y 0 en caso contrario. Esta codificación divide el espacio que ocupa el poliedro en distintas zonas, cada una de las cuales tiene asociada una máscara de bits y un estado de inclusión (Figura 5.18).

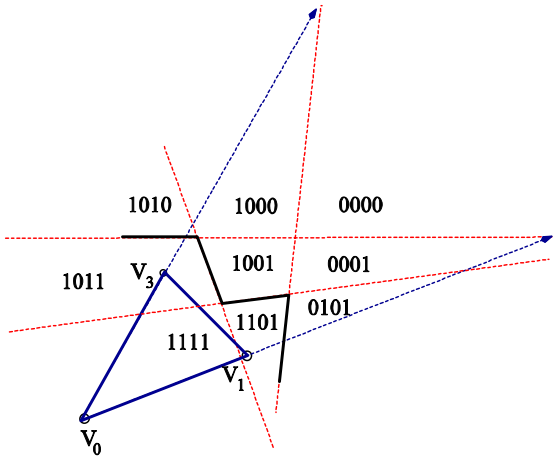


Figura 5.18: Interpretación geométrica (vista en 2D) del cambio de signo de α en el ámbito de un tetra-cono (en negro se representa el trozo de poliedro clasificado en el tetra-cono), división del espacio en zonas según $sign(\alpha_i)$. Los bits están representados por $b_i = (\alpha_i(P) \geq 0)$.

5.3.3. Cálculos incrementales

Para calcular la máscara de bits correspondiente a la coordenada baricéntrica α respecto a los tetraedros del recubrimiento del poliedro o respecto a aquellos clasificados en el tetra-cono correspondiente podemos utilizar un cálculo incremental entre frames.

Para ello se calcula en primer lugar el incremento de α para incrementos unitarios de x, y, z, es decir, se calcula cuánto se incrementa α cuando $\Delta x=1$, cuánto se incrementa α cuando $\Delta y=1$ y cuánto se incrementa α cuando $\Delta z=1$. A estos tres valores los hemos llamado $\Delta_x\alpha$, $\Delta_y\alpha$ e $\Delta_z\alpha$ respectivamente. Estos valores constantes son distintos para cada uno de los tetraedros del recubrimiento y se calculan y almacenan una sola vez. Estos incrementos se obtienen como sigue:

Sea $T=(V_0,V_1,V_2,V_3)$ un tetraedro, y sea $P=(x,y,z)$ un punto con coordenadas baricéntricas $(\alpha,\beta,\gamma,\delta)$ respecto a T. Vamos a realizar los cálculos necesarios para obtener $\Delta_x\alpha$, $\Delta_y\alpha$ y $\Delta_z\alpha$ cuando el punto P pase a la posición $P'=(x+\Delta x,y+\Delta y,z+\Delta z)$ con coordenadas baricéntricas $(\alpha',\beta',\gamma',\delta')$ respecto a T, siendo $\Delta x=1$, $\Delta y=1$ e $\Delta z=1$.

Si llamamos $P'_{\Delta x=1}=(x+1,y,z)$, $P'_{\Delta y=1}=(x,y+1,z)$, $P'_{\Delta z=1}=(x,y,z+1)$, entonces:

$$\Delta_x\alpha = (|P'_{\Delta x=1},V_1,V_2,V_3| - |P,V_1,V_2,V_3|) / |V_0,V_1,V_2,V_3|$$
$$\Delta_y\alpha = (|P'_{\Delta y=1},V_1,V_2,V_3| - |P,V_1,V_2,V_3|) / |V_0,V_1,V_2,V_3|$$
$$\Delta_z\alpha = (|P'_{\Delta z=1},V_1,V_2,V_3| - |P,V_1,V_2,V_3|) / |V_0,V_1,V_2,V_3|$$

Cuando se mueve el punto, se calcula el incremento que se ha producido en x, y, z, es decir Δx, Δy, Δz, respecto a su posición en el frame anterior, y se calcula la nueva coordenada baricéntrica α' para el frame actual en base a dichos incrementos y a los incrementos precalculados de $\Delta_x\alpha$, $\Delta_y\alpha$ e $\Delta_z\alpha$, para ello utilizamos la siguiente fórmula:

$$\alpha' = \alpha + \Delta x \cdot \Delta_x\alpha + \Delta y \cdot \Delta_y\alpha + \Delta z \cdot \Delta_z\alpha.$$

Estos cálculos pueden extenderse a las restantes coordenadas baricéntricas (β,γ,δ) de un punto respecto a un tetraedro. Sin embargo, realizar estos cálculos para todas las coordenadas es más costoso que el cálculo no incremental, pues hay que actualizar todas las coordenadas baricéntricas del punto respecto a todos los tetraedros del recubrimiento clasificados en el tetra-cono cuando se mueve el punto, aunque no se necesiten. Cuando se hace uso de la coherencia según hemos visto en la Sección 5.3.1 y 5.3.2, la mayor parte del tiempo sólo necesitamos la coordenada α, por lo que no es necesario calcular β,γ y δ.

Por consiguiente, aprovecharemos este cálculo incremental únicamente para la coordenada baricéntrica α, que se calcula siempre para todos los tetraedros del recubrimiento clasificados en un tetra-cono, y calcularemos las restantes coordenadas baricéntricas cuando sea necesario mediante el cálculo tradicional. Se han realizado diversas pruebas que corroboran que este cálculo combinado es más rápido que el tradicional o que el incremental para todas las coordenadas baricéntricas de un punto, siendo el cálculo combinado el realmente utilizado en la implementación final de los algoritmos.

5.3.4. Inclusión Punto/Poliedro

El algoritmo de detección de colisión que hemos desarrollado debe comprobar en algunos casos la inclusión del punto en los tetraedros del recubrimiento de un tetra-cono y la posición relativa del punto respecto a dichos tetraedros. Para ello partimos de un algoritmo desarrollado por Feito [FT97b], que hemos modificado para utilizar las coordenadas baricéntricas de un punto respecto a un tetraedro. Existen muchas otras soluciones para determinar la inclusión de puntos en poliedros. Haines [Hai94] propone un método para polígonos que puede ser extendido al espacio tridimensional [SE03]. En el artículo [OSF05] se hace una revisión de los métodos más representativos de inclusión de puntos en sólidos.

Para no contabilizar indebidamente la inclusión del punto cuando ésta tiene lugar en partes compartidas por dos o más tetraedros es necesario conocer la posición del punto respecto a cada tetraedro. La idea consiste en contabilizar media inclusión cuando ésta se da en una cara compartida por dos tetraedros, y contabilizar una sola vez la inclusión para el conjunto de tetraedros cuando se produce en una arista original. En el resto de los casos se contabiliza un valor de uno multiplicado por el signo del tetraedro. Los cálculos se han multiplicado por dos para poder utilizar aritmética entera, y se utiliza la denominación expresada en la Figura 5.19 y en la Tabla 5.1. A continuación podemos ver de forma resumida los pasos necesarios para determinar la inclusión (Algoritmo 5.1):

- Inicializar a cero una variable para contabilizar la inclusión: *Inclusión* = 0
- Repetir los siguientes pasos para cada tetraedro del recubrimiento:
  - Si el punto se encuentra sobre la arista V_2V_3 o en los vértices V_2 ó V_3, entonces el punto está sobre el poliedro y devolvemos *DENTRO*.
  - Si el punto se encuentra en el interior del triángulo no original $V_1V_2V_3$ o en el interior del tetraedro $V_0V_1V_2V_3$ (partes del tetraedro que no son compartidas con ninguna otra), contabilizamos la inclusión una sola vez, sumando o restando de acuerdo al signo del tetraedro:

    *Inclusión += 2*signo(tetraedro)*
  - Si el punto se encuentra en el interior de un triángulo original ($V_3V_2V_0$, $V_3V_0V_1$, $V_0V_1V_2$) o en una arista interior del triángulo no original del tetraedro (V_1V_2, V_3V_1), contabilizamos la inclusión como la mitad del signo del tetraedro:

    Inclusión += signo(tetraedro)
  - En el resto de los casos (arista V_0V_1 o vértice V_1) debemos contabilizar una sola vez la inclusión en la frontera del tetraedro, pues es compartido por un número indeterminado de tetraedros. Para esto se almacenan dos conjuntos de características (vértices y aristas) visitadas, uno para las positivas y otro para las negativas. Cada vez que determinamos la inclusión en estos casos, consultamos si ya ha sido visitada o no la correspondiente característica, mirando en el conjunto de acuerdo al signo del tetraedro, para en caso de no haber sido visitada, contabilizarla según su signo y añadirla al correspondiente conjunto para que no vuelva a contabilizarse:

    *Inclusión += 2*sign(tetraedro)*
- El punto está dentro del poliedro si y solamente si se cumple que *Inclusión=2*, en otro caso está fuera del poliedro.

Para el caso de tetraedros del recubrimiento degenerados (que no forman propiamente un tetraedro), el algoritmo de inclusión de puntos en tetraedros devuelve la posición del punto considerado respecto al triángulo no original del tetraedro, considerando sólo las posiciones del punto respecto a dicho triángulo. Si dicho triángulo no original es también degenerado, el algoritmo de inclusión devuelve la posición respecto a la arista que representa.

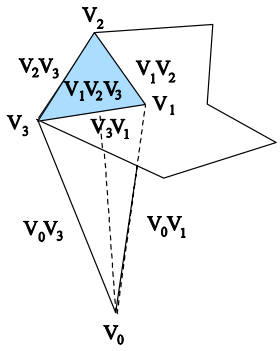


Figura 5.19: Denominación de las partes de un tetraedro del recubrimiento de un poliedro.

Características / bit	b_0	b_1	b_2	b_3	Nombre
Vértice V_0	1	0	0	0	Vértice original
Vértice V_1	0	1	0	0	Vértice no original, origen de la cara, vértice interno de la cara
Vértice V_2	0	0	1	0	Vértice no original; Vértice frontera
Vértice V_3	0	0	0	1	Vértice no original; Vértice frontera
Arista V_0V_1	1	1	0	0	Arista original o interior
Arista V_0V_2	1	0	1	0	Arista original o interior
Arista V_0V_3	1	0	0	1	Arista original o interior
Arista V_1V_2	0	1	1	0	Arista no original o exterior
Arista V_2V_3	0	0	1	1	Arista no original o exterior; Arista frontera
Arista V_3V_1	0	1	0	1	Arista no original o exterior
Cara $V_1V_2V_3$	0	1	1	1	Cara o triángulo, no original o exterior
Cara $V_3V_2V_0$	1	0	1	1	Cara o triángulo, original o interior
Cara $V_3V_0V_1$	1	1	0	1	Cara o triángulo, original o interior
Cara $V_0V_1V_2$	1	1	1	0	Cara o triángulo, original o interior
Tetraedro $V_0V_1V_2V_3$	1	1	1	1	Interior del tetraedro
bit b_i = 1 if sign(b_i(P))>0 **bit b_i = 0 if sign(b_i(P))=0**					

Tabla 5.1: Denominación de los elementos de un tetraedro del recubrimiento de un poliedro.

```
bool poliedro::inclusion(punto P) {
   positiveSet = negativeSet = Ø
   inclusion = 0
   for (t=0;t<this.numeroTetraedros();t++) {
      posicion = this.tetraedro(t).posicionInclusion(P)
      switch (posicion) {
         V0V1V2V3, V1V2V3:                    inclusion += 2*sign(this.tetraedro(t))
         V2V3, V2, V3 :                       return true
         V3V2V0, V3V0V1, V0V1V2, V1V2, V3V1:  inclusion += sign(this.tetraedro(t))
         V0:                                  return V0.inclusion()
         else {
            if (sign(this.tetraedro(t))==1)
               if (positiveSet.dentro(posicion)==false) {
                    inclusion += 2
                    positiveSet.inserta(posicion)
               }
            if (sign(this.tetraedro(t))==-1)
               if (negativeSet.dentro(posicion)==false) {
                    inclusion -= 2
                    negativeSet.inserta(posicion)
               }
         }
      }
   }
   return (inclusion == 2)
}
```

Algoritmo 5.1: Inclusión punto/poliedro.

También debemos considerar la situación especial en la que el punto sobre el que queremos obtener la detección de colisión esté situado sobre el centroide del poliedro. En este caso, como el punto es el origen del recubrimiento, no es posible determinar si dicho punto está o no dentro del poliedro. La solución adoptada consiste en clasificar el punto en tiempo de pre-procesamiento, utilizando cualquier método de clasificación de puntos en poliedros*. En el caso de que el punto esté sobre el centroide, se devuelve el valor de inclusión previamente almacenado para dicho punto.

5.3.5. Detección de Colisión Punto/Poliedro

En esta sección describimos el algoritmo de DC entre un punto y un poliedro desarrollado [JFSO06]. Este algoritmo utiliza la coherencia espacial para obtener un subconjunto de tetraedros menor que el original y la coherencia temporal para disminuir el número cálculos a realizar entre frames.

* Puede ser este mismo método, cambiando el origen del recubrimiento a otro punto distinto del centroide.

Se ha establecido un orden en los tetraedros del recubrimiento clasificados en un tetra-cono. Así podemos utilizar una máscara de bits en el ámbito de un tetra-cono para cada punto P sobre el que deseamos obtener la colisión con el poliedro. En dicha máscara, cada bit b_i, $i=1..n_j$, se corresponde con uno de los n_j tetraedros del recubrimiento clasificados en el tetra-cono $\angle T_j$ y representa si $\alpha_i(P) \geq 0$.

Optimizaremos el uso de las máscaras de bits reduciendo el número de tetraedros considerados pues es posible utilizar un único tetraedro del recubrimiento para cada cara del poliedro. Esto es debido a que todos los tetraedros del recubrimiento de una cara tienen el mismo valor para el signo de la coordenada baricéntrica α (Figura 5.20). Por tanto, podemos utilizar un único tetraedro para conocer el signo de la coordenada α de todos los tetraedros del recubrimiento de dicha cara, y por tanto conocer la posición del punto en relación al plano soporte del triángulo no original de ese tetraedro, y por extensión de la cara completa.

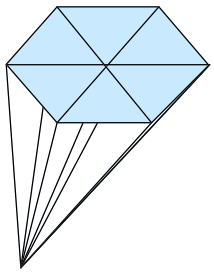


Figura 5.20: Respecto a una determinada cara, un punto tiene el mismo valor en el signo de α para todos los tetraedros de su recubrimiento, pues todos los triángulos no originales se encuentra en el mismo plano.

El algoritmo que estamos describiendo calcula dicha máscara de bits en cada frame, utilizando un único bit por cada cara del poliedro clasificada en un tetra-cono, y compara dicha máscara con la máscara calculada en el frame anterior. Si ambas máscaras son iguales, el punto se encuentra en la misma zona espacial que en el frame previo, por lo que su estado de inclusión en el poliedro es el mismo que en el frame anterior, y no es necesario realizar cálculos adicionales para obtener su colisión en el frame actual.

Cuando las máscaras de bits son distintas entre dos frames consecutivos, es necesario calcular el signo de las restantes coordenadas baricéntricas para obtener la inclusión del punto en cada tetraedro[*], y determinar así la inclusión en el poliedro. Los valores de $\beta_k(P)$ sólo se calculan si el correspondiente bit b_k de la máscara que

[*] En este caso es necesario calcular las coordenadas baricéntricas para todos los tetraedros clasificados en el correspondiente tetra-cono y no sólo para un tetraedro representativo de cada cara.

representa $\alpha_i(P)$ es igual a 1[*]. Del mismo modo, los valores de $\gamma_k(P)$ sólo se calculan si $\beta_k(P) \geq 0$, y los valores de $\delta_k(P)$ sólo cuando $\gamma_k(P) \geq 0$.

Debido a la coherencia temporal, los puntos sobre los que obtenemos la colisión cambian poco de posición en el espacio, estando la mayor parte del tiempo en una misma zona, por lo que la máscara de bits que representa el signo de $\alpha_i(P)$ cambia menos veces cuanto mayor sea la coherencia temporal, con el consiguiente aumento de eficiencia en el algoritmo. Además al restringir los cálculos a los tetraedros clasificados en un tetra-cono, el número de zonas espaciales que se generan disminuye bastante respecto al número de zonas del poliedro completo, aumentando por tanto la coherencia y la eficiencia de este método.

A continuación mostramos de forma resumida los pasos dados (Algoritmo 5.2 y Figura 5.21):

- Gracias a la coherencia temporal, en primer lugar se comprueba si el punto se encuentra en el mismo tetra-cono que en el frame anterior. Si no es así se obtiene el tetra-cono en el que se encuentra el punto, clasificándolo en el árbol que representa el tetra-tree. A continuación se comprueba si el punto se encuentra en el tetraedro envolvente asociado a dicho tetra-cono[†]. A partir de este momento, todos los cálculos se efectúan en el ámbito del tetra-cono asociado.

- A continuación se comprueba si la coordenada baricéntrica $\alpha_i(P)$ es igual a la del frame anterior para todo $i=1..n_j$, siendo n_j el número de tetraedros clasificados en el tetra-cono $\angle T_j$.[‡] Si la máscara que representa dicha coordenada es igual a la máscara previa, el punto tendrá el mismo estado de inclusión y no será necesario realizar comprobaciones adicionales.

- Posteriormente sólo se calcula, y por orden, las coordenadas baricéntricas $\beta_i(P)$, $\gamma_i(P)$ y $\delta_i(P)$ para aquellos tetraedros del recubrimiento del tetra-cono que cumplan que en la coordenada baricéntrica previa tuvieran un valor positivo o igual a cero.

- Por último se obtiene la inclusión del punto en los tetraedros del tetra-cono en los que $sign(\delta_i(P))>0$.

---

[*] Cuando el valor de este bit es 0 indica que el signo de la coordenada baricéntrica α es negativo, y debido a la Propiedad 5.2 el punto se encuentra fuera del tetraedro.

[†] Podríamos comprobar previamente la inclusión en la esfera envolvente asociada al tetra-cono, pero hemos constatado que se obtienen mejores resultados comprobando únicamente la inclusión en el tetraedro envolvente.

[‡] Debemos tener en cuenta que podemos optimizar este paso utilizando únicamente un tetraedro por cada cara clasificada en el tetra-cono.

El caso de tetraedros degenerados se ha tratado de forma similar a como se hace en el algoritmo de inclusión (Sección 5.3.4), obteniendo la posición del punto considerado respecto al triángulo no original del tetraedro, en lugar de respecto al tetraedro completo, o la posición del punto respecto a la arista que forma parte del poliedro en caso de que el triángulo no original sea también degenerado.

El algoritmo de DC que acabamos de ver se ha simplificado y se ha primado la legibilidad sobre la eficiencia, al igual que en el resto de algoritmos que describimos en este capítulo.

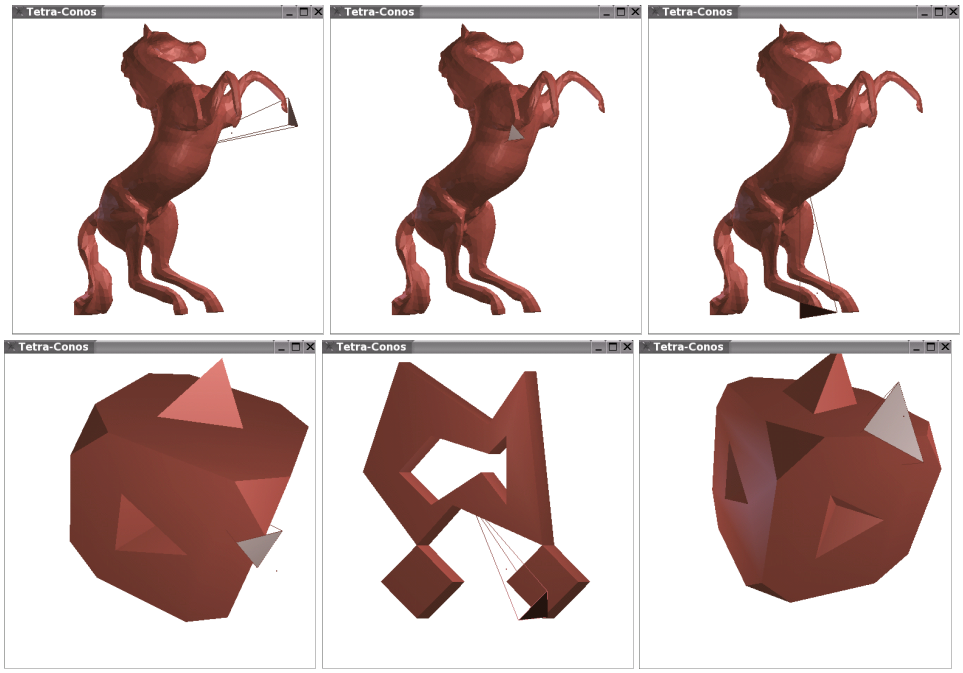


Figura 5.21: Detección de colisión punto/poliedro. Se muestra el tetraedro envolvente asociado al tetra-cono en el que se encuentra el punto.

```
static previob_α = 0

bool poliedro::testDeteccionColision(punto P) {
   //Inclusion en la esfera envolvente
   if (this.esferaEnvolvente().inclusion(P)==false) return false
   // Obtención del tetra-cono en el que se encuentra el punto
   if (tetraConoAnterior.inclusion(P)==true) tc = tetraConoAnterior
   else tetraConoAnterior = tc = this.clasificaTetraTree(P)
   // Inclusión del punto en el tetraedro envolvente del tetra-cono
   if (tc.tetraedroEnvolvente().inclusion(P)==false) return false
   // Detección de colisión restringida al tetra-cono
   // Cálculo de la máscara de bits alfa
   s_α = s_β = s_γ = s_δ = b_α = b_β = b_γ = b_δ = 0
```

```
   for (t=0;t<tc.numeroTetraedros();t++) { // Se puede optimizar usando un tetraedro por cara
      sα[t] = sign(tc.tetraedro(t).α(P))
      bα[t] = (sα[t]>=0)
   }
   // Si la mascara de bits es la misma que en la iteración anterior
   if (bα == previobα) return EQUAL_STATE
   previobα = bα
   // Cálculo de la máscara de bits beta
   for (t=0;t<tc.numeroTetraedros();t++)
      if (bα[t]==1) {
         sβ[t] = sign(tc.tetraedro(t).β(P))
         bβ[t] = (sβ[t]>=0)
      }
   // Cálculo de la máscara de bits gamma
   for (t=0;t<tc.numeroTetraedros();t++)
      if (bβ[t]==1) {
         sγ[t] = sign(tc.tetraedro(t).γ(P))
         bγ[t] = (sγ[t]>=0)
      }
   // Cálculo de la máscara de bits delta
   for (t=0;t<tc.numeroTetraedros();t++)
      if (bγ[t]==1) {
         sδ[t] = sign(tc.tetraedro(t).δ(P))
         bδ[t] = (sδ[t]>=0)
      }

   // Inclusión sólo donde la mascara de bits delta = 1
   positiveSet = negativeSet = ∅
   inclusion = 0
   for (t=0;t<tc.numeroTetraedros();t++)
      if (bδ[t]==1) {
         if (tc.tetraedro(t).degenerado()==true)
            posicion = tc.tetraedro(t).posicionInclusion(P) // Contempla el caso degenerado
         else
            posicion = posicionDeteccionColision(sα[t], sβ[t], sγ[t], sδ[t])
         switch (posicion) {
            V0V1V2V3, V1V2V3:                  inclusion += 2*sign(tc.tetraedro(t))
            V2V3, V2, V3 :                     return true
            V3V2V0, V3V0V1, V0V1V2, V1V2, V3V1:  inclusion += sign(tc.tetraedro(t))
            V0:                                return V0.inclusion()
            else {
               if (sign(tc.tetraedro(t))==1)
                  if (positiveSet.dentro(posicion)==false) {
                       inclusion += 2
                       positiveSet.inserta(posicion)
                  }
               if (sign(tc.tetraedro(t))==-1)
                  if (negativeSet.dentro(posicion)==false) {
                       inclusion -= 2
                       negativeSet.inserta(posicion)
                  }
            }
         }
      }
   return (inclusion == 2)
}
```

Algoritmo 5.2: Detección de Colisión Punto/Poliedro.

5.4. Algoritmo de Detección de Colisión Esfera/Poliedro

Existen numerosas aplicaciones en las que puede ser necesario un método de detección de colisión entre una esfera y un poliedro, por ejemplo en sistemas de partículas, en los que las dimensiones de las mismas son significativas, o en aplicaciones en las que se utilizan esferas como volúmenes envolventes de los objetos o partes de los mismos.

Es posible representar objetos utilizando puntos o esferas. Klein et. al [KZ04] presentan un esquema de detección de colisión basado en nubes de puntos. Para desarrollar este método construyen una jerarquía de puntos en la que en cada nodo guardan un número suficiente de puntos de muestreo así como una esfera como parte de la superficie representada. Kim et al. [KGS98] utilizan un esquema de subdivisión espacial jerárquica para la detección de colisión entre esferas en movimiento. En cuanto a la detección de colisión entre una esfera y un poliedro podemos destacar el trabajo de Vemuri et al. [VCC98] basado en un octree, y el de Karabassi et. al [KPT99] basado en un test de intersección entre un cilindro y un triángulo.

El algoritmo de detección de colisión entre una esfera y un poliedro [JFS04] [JFSO06] desarrollado en esta sección utiliza recubrimientos simpliciales y tetra-trees, y será usado también como base para uno de los algoritmos de detección de colisión entre poliedros que veremos al final de este capítulo.

Para llevar buen término este algoritmo es necesario el desarrollo de ciertos conceptos que permitirán extender aquellos usados en el algoritmo de detección de colisión punto/poliedro. La idea básica consiste en extender los tetraedros y los tetra-conos para que la detección de colisión entre la esfera y el poliedro pueda realizarse utilizando la detección de colisión entre un punto (el centro de la esfera) y esos tetraedros y tetra-conos transformados o extendidos, realizando así un test de colisión menos costoso. Para ello utilizaremos conceptos como área de influencia y extensión de un tetraedro o de un tetra-cono.

Además, los conceptos de extensión de un tetraedro o de un tetra-cono nos permitirán establecer una medida de la proximidad entre un punto y una cara del poliedro, permitiendo a su vez obtener la inclusión del punto en el poliedro con una determinada tolerancia, es decir, permitiendo establecer un error ε a la hora de obtener la colisión del punto con el poliedro.

De forma resumida, el algoritmo aquí desarrollado obtiene en primer lugar la colisión del centro de la esfera con el poliedro. Si no se produce colisión, obtiene los

tetra-conos en los que se encuentra la esfera, descartando posteriormente aquellos en cuyo tetraedro envolvente extendido no se encuentra el centro de la esfera. Finalmente realiza un test de intersección entre la esfera y la parte de las caras del poliedro clasificadas en el conjunto de tetra-conos obtenido anteriormente.

El área de influencia de un triángulo, junto con el concepto de offset, nos permitirá definir el concepto de área de influencia extendida y los conceptos de tetraedro y tetra-cono extendidos.

5.4.1. Offset de un triángulo

Para definir el área de influencia de un triángulo necesitamos definir el concepto de *offset* de un triángulo[*].

Notación: Dada una cara H con n_k aristas, que forma parte de un poliedro F, notaremos C_H al recubrimiento mediante tetraedros $\{T_i\}$, $i=1,\ldots,n_k$, de la cara H respecto al origen del recubrimiento de F.

Definición 5.14: Sea F un poliedro y sea H una cara tal que $H \in F$. Sea $T_i=V_0V_1V_2V_3$ un tetraedro tal que $T_i \in C_H$. Definimos el *offset signado* del triángulo $V_1V_2V_3$ a una distancia r, al conjunto formado por los dos planos que distan r unidades del plano soporte de $V_1V_2V_3$. Llamaremos *offset positivo* al plano que se encuentra en la dirección de la normal a la cara H, y *offset negativo* al plano que se encuentra en la dirección opuesta.

Notación: Dado un poliedro F con recubrimiento C_F y un tetraedro $T_i=V_0V_1V_2V_3$ tal que $T_i \in C_F$, notaremos el *offset positivo* del triángulo $V_1V_2V_3$ a una distancia r como $offset^+(V_1V_2V_3,r)$ o también como $offset^+_i(r)$. El *offset negativo* lo notaremos como $offset^-(V_1V_2V_3,r)$ o también como $offset^-_i(r)$.[†]

---

[*] Se ha preferido utilizar el concepto de *offset de un triángulo* en lugar del de *offset de una cara*, debido sobre todo a que los cálculos que permiten obtener la posición de un punto respecto al offset se realizan en base al tetraedro en el que se encuentra dicho triángulo. Sin embargo, el concepto de *offset*, y por consiguiente los de *área de influencia* y *área de influencia extendida*, pueden definirse en base a una cara del poliedro.

[†] Otros motivos que nos llevan a utilizar el concepto de *offset de un triángulo* en lugar de *offset de una cara* están relacionados con la notación y su uso para tetraedros y tetra-conos extendidos. Por una parte, podemos utilizar el subíndice que representa el tetraedro, y por otra podemos utilizar el concepto de offset de un triángulo original para definir los tetraedros y tetra-conos extendidos, pues del mismo modo que se ha definido para el triángulo $V_1V_2V_3$, se puede definir el offset de los triángulos $V_0V_1V_2$, $V_0V_2V_3$ ó $V_0V_3V_1$.

Definición 5.15: Sea F un poliedro cuyo recubrimiento C_F tiene origen en V_o. Sea H una cara de F con vector normal $\boldsymbol{n}$. Definimos el *signo* de la cara H, *sign(*H*),* como +1 si V_o se encuentra en la dirección opuesta a $\boldsymbol{n}$, –1 si V_o se encuentra en la dirección de $\boldsymbol{n}$, y 0 si se encuentra en el plano soporte de H.

Debido a que el offset signado depende de la orientación de la cara a cuyo recubrimiento pertenece el tetraedro $T=V_oV_1V_2V_3$, tenemos offsets distintos dependiendo de si dicha cara es positiva o negativa. Si la cara es positiva (Figura 5.22), el offset positivo se encuentra en el lado opuesto a V_o respecto al triángulo $V_1V_2V_3$. Si la cara es negativa, el offset positivo se encuentra en el mismo lado que V_o respecto al triángulo $V_1V_2V_3$.

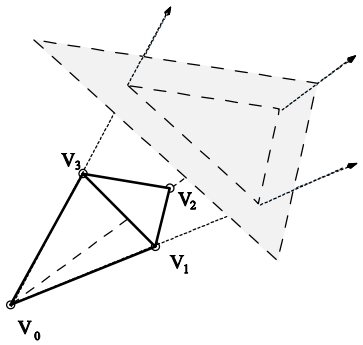


Figura 5.22: Offset positivo del triángulo $V_1V_2V_3$ (en la dirección de la normal de la cara).

El concepto de offset signado nos permitirá delimitar ciertas zonas del espacio, que posteriormente llamaremos áreas de influencia y áreas de influencia extendida de un triángulo. Estas zonas serán útiles para obtener un algoritmo de detección de colisión entre una esfera y un poliedro de forma parecida a como se lleva a cabo la DC entre un punto y un poliedro. Además permitirá extender un poliedro de manera que se pueda realizar la detección de colisión con una determinada tolerancia o error.

5.4.2. Área de influencia de un triángulo

A continuación vamos a definir una nueva región del espacio comprendida entre dos planos paralelos, el plano soporte de un triángulo y su offset positivo (Figura 5.23). Llamaremos a esta región área de influencia.

Definición 5.16: Sea F un poliedro con recubrimiento C_F. Sea $T_i=V_oV_1V_2V_3$ un tetraedro tal que $T_i \in C_F$. Definimos el *área de influencia de tamaño r* del triángulo $V_1V_2V_3$, a la zona del espacio formada por todos los puntos comprendidos entre el plano soporte de $V_1V_2V_3$ y $offset^+(V_1V_2V_3,r)$.

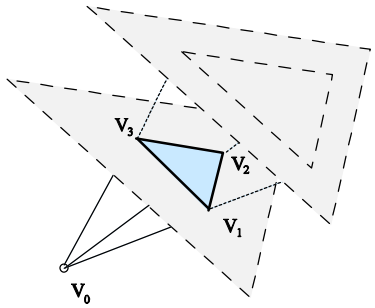


Figura 5.23: Área de influencia de un triángulo que forma parte de una cara con signo positivo.

Notación: Dado un poliedro F con recubrimiento C_F y un tetraedro $T_i=V_0V_1V_2V_3$ tal que $T_i \in C_F$, notaremos área de influencia de tamaño r del triángulo $V_1V_2V_3$ como $AI(V_1V_2V_3,r)$, o también como $AI_i(r)$.

Para determinar si un punto se encuentra incluido en el área de influencia de un triángulo podemos utilizar las coordenadas baricéntricas:

Propiedad 5.12: Sea F un poliedro con recubrimiento C_F. Sea $T_i=V_0V_1V_2V_3$ un tetraedro tal que $T_i \in C_F$, y P_{off} un punto tal que $P_{off} \in offset^+(V_1V_2V_3,r)$. Entonces para todo punto P se cumple que [P *in* $AI(V_1V_2V_3,r)$] sii $\alpha_i(P) \in [0, \alpha_i(P_{off})]$.

5.4.3. Área de influencia extendida de un triángulo

El área de influencia extendida nos permitirá determinar la posición de una esfera respecto al plano en el que se encuentra un triángulo no original del recubrimiento de una cara, de forma similar a como se determina dicha posición entre un punto y el plano soporte de un triángulo.

De forma similar a como hemos definido el área de influencia, definiremos el área de influencia extendida (Figura 5.24), para poder tratar adecuadamente los puntos cercanos a las concavidades de los poliedros. Esta zona del espacio está también limitada por dos planos paralelos a una distancia r respecto al plano que define el triángulo y será la base para el desarrollo del algoritmo de detección de colisión entre una esfera y un poliedro, así como de uno de los algoritmos de detección de colisión entre poliedros.

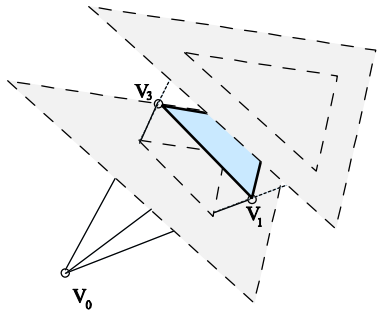


Figura 5.24: Área de influencia extendida de un triángulo.

Definición 5.17: Sea F un poliedro con recubrimiento C_F. Sea $T_i=V_0V_1V_2V_3$ un tetraedro tal que $T_i \in C_F$. Definimos el *área de influencia extendida de tamaño r* del triángulo $V_1V_2V_3$, a la zona del espacio formada por todos los puntos comprendidos entre $offset^+(V_1V_2V_3,r)$ y $offset^-(V_1V_2V_3,r)$.

Notación: Dado un poliedro F con recubrimiento C_F y un tetraedro $T_i=V_0V_1V_2V_3$ tal que $T_i \in C_F$, notaremos área de influencia extendida de tamaño r del triángulo $V_1V_2V_3$ como $AIE(V_1V_2V_3,r)$, o también como $AIE_i(r)$.

Para determinar si un punto se encuentra incluido en el área de influencia extendida de un triángulo podemos utilizar las coordenadas baricéntricas de forma similar a como hicimos para el caso de áreas de influencia:

Propiedad 5.13: Sea F un poliedro con recubrimiento C_F. Sea $T_i=V_0V_1V_2V_3$ un tetraedro tal que $T_i \in C_F$, y P_{off} un punto tal que $P_{off} \in offset^+(V_1V_2V_3,r)$. Entonces para todo punto P se cumple que $[P \text{ in } AIE(V_1V_2V_3,r)]$ sii $\alpha_i(P) \in [-\alpha_i(P_{off}), \alpha_i(P_{off})]$.

El área de influencia extendida de tamaño r de un triángulo nos permitirá conocer si una esfera en movimiento colisiona con una cara del poliedro. Para que esto ocurra debe cumplirse en primer lugar que el centro de la esfera se encuentre en el área de influencia extendida de tamaño el radio de la esfera respecto a un triángulo no original del tetraedro del recubrimiento.

En el capítulo anterior utilizamos el área de influencia extendida limitada (Sección 4.3.4) para acotar aún más la región definida por el área de influencia extendida. Al utilizar directamente el concepto de tetra-tree en los algoritmos de este capítulo no es necesario limitar esta región como hicimos anteriormente, pues esta zona del espacio queda acotada directamente al utilizar tetra-conos y tetraedros envolventes.

5.4.4. Tetraedro y Tetra-Cono extendidos

Vamos a utilizar los conceptos anteriores para definir el concepto de extensión de un tetraedro y extensión de un tetra-cono (Figura 5.25). Esto nos permitirá realizar operaciones entre esferas y tetraedros o tetra-conos, mediante operaciones entre puntos y tetraedros o tetra-conos extendidos.

Definición 5.18: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo. Definimos la *extensión en r unidades del tetraedro* T, como el tetraedro delimitado por $offset^+(V_1V_2V_3,r)$, $offset^+(V_0V_2V_1,r)$, $offset^+(V_0V_3V_2,r)$ y $offset^+(V_0V_1V_3,r)$, definidos dichos offsets en base a las caras del tetraedro T. En el caso de que T sea negativo, el tetraedro extendido está delimitado por $offset^-(V_1V_2V_3,r)$, $offset^-(V_0V_2V_1,r)$, $offset^-(V_0V_3V_2,r)$ y $offset^-(V_0V_1V_3,r)$.

Notación: La extensión en *r* unidades del tetraedro T la notaremos como *Ext(*T,*r)*.

Podemos considerar que un tetraedro extendido *r* unidades es otro tetraedro delimitado por una serie de planos a *r* unidades de sus caras, de manera que el tetraedro extendido es mayor que el inicial y además el tetraedro extendido contiene completamente al tetraedro inicial.

Definición 5.19: Dado un tetraedro T, definimos la *extensión en r unidades del tetra-cono* ∠T, al tetra-cono obtenido a partir del tetraedro extendido *Ext(*T,*r)*.

Notación: La extensión en *r* unidades del tetra-cono ∠T la notaremos como *Ext(*∠T,*r)*.

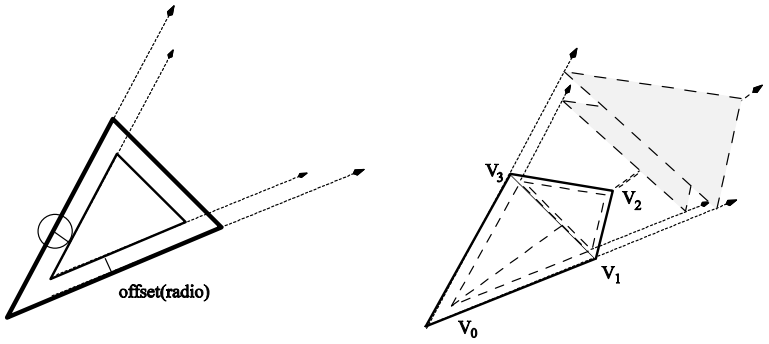


Figura 5.25: Tetraedro y tetra-cono extendidos.

Para determinar si un punto se encuentra en un tetraedro o en un tetra-cono extendido podemos utilizar las coordenadas baricéntricas. Como en un tetra-tree tanto los tetra-conos como los tetraedros envolventes están asociados a tetraedros positivos, sólo necesitaremos caracterizar la inclusión en base a que los tetraedros de los que se parte son positivos.

Propiedad 5.14: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo y *Ext*(T,*r*) la extensión en *r* unidades del tetraedro T. Sean P_0, P_1, P_2 y P_3 puntos tales que $P_0 \in offset^+(V_1V_2V_3,r)$, $P_1 \in offset^+(V_0V_2V_3,r)$, $P_2 \in offset^+(V_0V_3V_1,r)$ y $P_3 \in offset^+(V_0V_1V_2,r)$. Entonces para todo punto *P* se cumple que [*P in Ext*(T,*r*)] sii $\alpha(P) \leq \alpha(P_0) \wedge \beta(P) \leq \beta(P_1) \wedge \gamma(P) \leq \gamma(P_2) \wedge \delta(P) \leq \delta(P_3)$.

Propiedad 5.15: Sea T=$V_0V_1V_2V_3$ un tetraedro positivo y *Ext*($\angle$T,*r*) la extensión en *r* unidades del tetra-cono $\angle$T. Sean P_1, P_2 y P_3 puntos tales que $P_1 \in offset^+(V_0V_2V_3,r)$, $P_2 \in offset^+(V_0V_3V_1,r)$ y $P_3 \in offset^+(V_0V_1V_2,r)$. Entonces para todo punto *P* se cumple que [*P in Ext*($\angle$T,*r*)] sii $\beta(P) \leq \beta(P_1) \wedge \gamma(P) \leq \gamma(P_2) \wedge \delta(P) \leq \delta(P_3)$.

Los valores de $\alpha(P_0)$, $\beta(P_1)$, $\gamma(P_2)$ y $\delta(P_3)$ son constantes y pueden precalcularse, por lo que la obtención de la inclusión de un punto en un tetraedro o en un tetra-cono extendido no supone un aumento de coste computacional en la detección de colisión.

5.4.5. Intersección Esfera/Poliedro

Antes de describir el algoritmo de detección de colisión esfera/poliedro veamos cómo se calcula la intersección entre una esfera y un poliedro, necesaria para el desarrollo del mismo.

Definición 5.20: Sea F un poliedro y T un tetraedro de su recubrimiento. Definimos *elementos frontera* de T a aquellas aristas, vértices y triángulos (en el caso de poliedros con caras triangulares) que forman parte de la frontera de F.

Los elementos frontera de un tetraedro (Figura 5.26), como pertenecientes a la frontera del poliedro, son los elementos con los que si se produce la intersección de la esfera entonces se produce directamente la intersección con el poliedro.

En el caso de poliedros formados por caras triangulares, dichas caras junto con el vértice original forman directamente los tetraedros del recubrimiento, por lo que los vértices no originales, aristas no originales y triángulo no original de esos tetraedros, son todos ellos elementos que pertenecen a la frontera del poliedro. Para estos casos, si la esfera colisiona con dichos elementos de la frontera, se produce colisión con el poliedro.

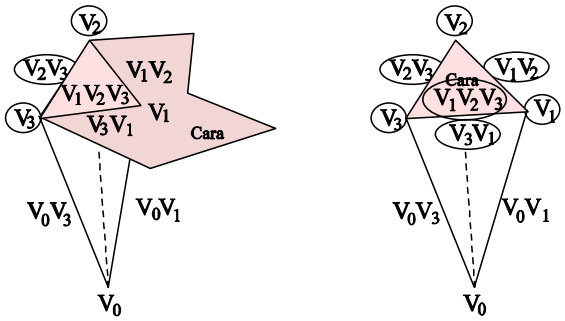


Figura 5.26: Elementos frontera de un tetraedro (rodeados por un círculo). a) Respecto a una cara no triangular. b) Respecto a una cara triangular.

Cuando un poliedro tiene caras no triangulares, no todos los vértices y aristas no originales de los tetraedros del recubrimiento de dichas caras se consideran elementos frontera, pues algunos de ellos no pertenecen a la frontera del poliedro, pudiendo darse el caso de que la esfera colisione con algún vértice o arista no original, y no lo haga con la cara. Para esta situación se ha desarrollado un algoritmo especial que comprueba la intersección entre una cara y una esfera como veremos a lo largo de esta sección.

A continuación mostramos unas propiedades que nos permiten calcular la intersección entre una esfera y los elementos frontera de un tetraedro y poder así concluir que se produce intersección entre la esfera y el poliedro.

Sea F un poliedro, sea E una esfera de centro C y radio r, y T=$V_0V_1V_2V_3$ un tetraedro tal que T$\in$C$_F$. Podemos determinar la intersección de la esfera E con el poliedro F a través de los elementos frontera del tetraedro T de la siguiente forma [JFSO06]:

- **Propiedad 5.16:** Sea V_i un vértice frontera de T. Si [V_i *in* E] entonces [E $\cap$ F $\neq \varnothing$].
- **Propiedad 5.17:** Sea V_iV_j una arista frontera de T. Si [$V_iV_j \cap$ E $\neq \varnothing$] entonces [E $\cap$ F $\neq \varnothing$].
- **Propiedad 5.18:** Sea Q el punto más cercano sobre el plano soporte del triángulo $V_1V_2V_3$ al punto C. Si [Q *in* E] $\wedge$ [Q *in* $V_1V_2V_3$] entonces [E $\cap$ $V_1V_2V_3$ $\neq \varnothing$].

Si comprobamos estas propiedades en el orden dado obtenemos un algoritmo eficiente que obtiene intersección entre la esfera y el poliedro cuando alguna de las Propiedades 5.16 ó 5.17 se cumple. Si no se cumplen estas dos propiedades y se

cumple la Propiedad 5.18. debemos comprobar si se produce la intersección de la esfera con la cara asociada al triángulo*, para en caso de que ocurra esta intersección, obtener que la esfera interseca con el poliedro.

Para determina la intersección cara/esfera (Algoritmo 5.3) es necesario contabilizar el número de intersecciones que se producen entre la esfera y los triángulos no originales de los tetraedros del recubrimiento, de manera similar a como se hace en el algoritmo de inclusión punto/poliedro (Algoritmo 5.1) en la situación en la que el punto se encuentra sobre una cara del poliedro. Para ello se ha desarrollado el siguiente teorema:

Teorema 5.2: Sea F un poliedro y E una esfera de centro C y radio r. Sea H una cara de F con recubrimiento $C_H=\{S_j\}$, $j=1..n_k$. Sea Q el punto más cercano sobre el plano soporte de H al punto C. Sea $\{T_i=V_0V_1V_{i,2}V_{i,3}\}\in C_H$, $i=1..m_k$, $m_k\leq n_k$ el subconjunto de tetraedros de C_H que cumplen que $[Q\ in\ E] \wedge [Q\ in\ V_1V_{i,2}V_{i,3}]$ (Propiedad 5.18). Entonces $[E \cap H \neq \emptyset]$ y por extensión $[E \cap F \neq \emptyset]$ si y solamente si se cumple alguna de las siguientes condiciones:

a) Existe algún tetraedro S_j, $j\in[1..n_k]$, que cumple que un vértice frontera de S_j está incluido en la esfera E (Propiedad 5.16).

b) Existe algún tetraedro S_j, $j\in[1..n_k]$, que cumple que una arista frontera de S_j interseca con la esfera E (Propiedad 5.17).

c) $Q\neq V_1 \wedge sign(H)\cdot\sum_{i=1}^{m_k} sign(T_i)\cdot In(Q,V_1V_{i,2}V_{i,3}) = 2$

siendo $In(Q,V_1V_{i,2}V_{i,3})$ una función que devuelve:

- 2 si Q está en el interior del triángulo no original $V_1V_{i,2}V_{i,3}$.
- 1 si Q está sobre las aristas que no son frontera de T_i.
- 0 si Q no está en ninguna de las situaciones anteriores.

d) $Q=V_1 \wedge [V_1\ in\ H]$.

Demostración: Trivialmente si un vértice del poliedro está dentro de la esfera, o una arista del poliedro interseca con la esfera, se produce intersección esfera-poliedro. En el caso de que el punto más cercano Q del plano soporte de la cara al centro de la esfera se encuentre dentro de la esfera E y de uno o más triángulos del recubrimiento de la cara, es necesario comprobar la inclusión de este punto Q en la cara H, pues si se encuentra en el interior de la cara se produce intersección de la esfera con la cara. En la inclusión de Q en H deben considerarse de modo especial las aristas de los tetraedros del recubrimiento que no son frontera de la cara y se encuentran sobre su plano soporte, pues estos elementos son compartidos por un par

* Cuando la cara está formada por un único triángulo, entonces concluimos que se produce colisión entre la esfera y la cara y por extensión entre la esfera y el poliedro.

de tetraedros; así como el vértice V_1, común a todos los tetraedros del recubrimiento de la cara y que se encuentra sobre su plano soporte; para tratarlos adecuadamente de forma similar a como se hace la inclusión de puntos en polígonos en 2D, es decir, ponderando la inclusión en cada caso, obteniendo también inclusión cuando $Q=V_1$ y V_1 está dentro de la cara H. Hay que tener en cuenta el signo de la cara, o lo que es lo mismo, su orientación respecto al origen del recubrimiento, pues dependiendo de esto, los tetraedros del recubrimiento tendrán un signo u otro; por este motivo es necesario multiplicar finalmente por el signo de la cara para obtener un resultado correcto. Además puede ser que el punto Q no esté en el interior de la cara y que la esfera efectivamente interseque con dicha cara, pero en este caso la esfera interseca con alguna de las aristas de la cara, o algún vértice de la cara está incluido en la esfera, casos ya contemplados anteriormente. □

Evidentemente este teorema es de aplicación a los tetraedros del recubrimiento de una cara clasificados en un tetra-cono, de manera que si se produce intersección en el ámbito del tetra-cono se produce colisión entre la esfera y el poliedro.

A continuación vamos a describir los pasos necesarios para comprobar las distintas situaciones descritas en el Teorema 5.2, como la inclusión de un vértice en una esfera, la intersección entre una arista y una esfera y la determinación del punto más cercano en el plano soporte de un triángulo a una esfera. Para ello enunciamos las siguientes propiedades:

- **Inclusión de los vértices frontera del tetraedro en una esfera:**

  Propiedad 5.19: Dado un vértice V y una esfera E de centro C y radio R. Entonces $dist(V - C)^2 < R^2$ si y solamente si [V *in* E].

- **Intersección entre una arista frontera y una esfera:**

  Propiedad 5.20: Sea V_iV_j una arista y E la esfera de centro C y radio r. Sea $u = (C-V_i)\cdot(V_j-V_i) \,/\, (V_j-V_i)\cdot(V_j-V_i)$. Entonces [$V_iV_j \cap E \neq \emptyset$] si y solamente si $u \in [0,1]$ y $u \leq r$ [SE03].

- **Intersección entre un triángulo y una esfera:**

  Propiedad 5.21: Sea E una esfera de centro C y radio r y sea $T=V_1V_2V_3$ un triángulo. Sea Q el punto más cercano sobre el plano soporte de T al punto C, y b la distancia de Q a C. Entonces [$T \cap E \neq \emptyset$] si y solamente si $|b| \leq r$.

  El punto más cercano Q sobre el plano soporte del triángulo T al punto C, se calcula de la siguiente forma [SE03]:

Dado el plano soporte de T mediante la ecuación implícita normalizada $P \cdot \boldsymbol{n} + d = 0$, en la que P es un punto cualquiera sobre el plano, $\boldsymbol{n}$ es la normal al plano y d es una constante, podemos obtener el punto Q como $Q = C - b \cdot \boldsymbol{n}$, siendo b la distancia entre Q y C según la expresión $b = (C - P) \cdot \boldsymbol{n}$ (Figura 5.27).

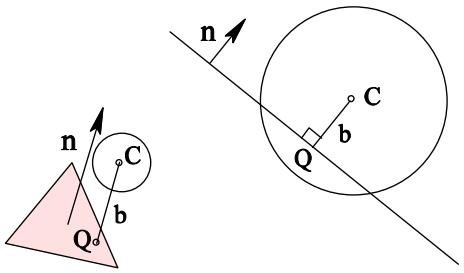


Figura 5.27: Cálculo del punto Q (el punto más cercano en el plano al centro de una esfera).

```
bool esfera::interseccion(cara F) {
   inclusion = 0
   calculadoPuntoMasCercanoQ = false
   for (T=0;T < F.numeroTetraedros();T++) {
      if (this.inclusion(F.tetraedro(T).V2)==true || this.inclusion(F.tetraedro(T).V3)==true)
         return true
      if (this.interseccion(F.tetraedro(T).V2V3)==true)
         return true
      if (calculadoPuntoMasCercanoQ == false) {
         calculadoPuntoMasCercanoQ = true
         if (this.calculaPuntoMasCercano(F.plano(), &Q)==false)
             return false
      }
      posicion = F.tetraedro(T).V1V2V3.posicionRelativa(Q)
      switch (posicion) {
         V2V3, V2, V3 : return true
         V1V2V3:        inclusion += 2*sign(F.tetraedro(T))
         V1V2, V3V1:    inclusion += sign(F.tetraedro(T))
         V1:            return V1.inclusión()
      }
   }
   return (inclusion == 2)
}
```

Algoritmo 5.3: Intersección Esfera/Poliedro.

5.4.6. Coherencia temporal y geométrica

Utilizaremos un algoritmo basado en la detección de colisión Punto/Poliedro para comprobar la colisión entre una esfera y un poliedro. Para ello extenderemos el poliedro usando las áreas de influencia de los triángulos no originales del recubrimiento de sus caras.

Podemos deducir las siguientes propiedades que relacionan la intersección de una esfera y un poliedro con las áreas de influencia extendidas:

Sea F un poliedro y $T=V_0V_1V_2V_3$ un tetraedro tal que $T \in C_F$. Sea E una esfera de centro *C* y radio *r*:

- **Propiedad 5.22:** Si $[E \cap V_1V_2V_3 \neq \emptyset]$ entonces [*C in AIE*($V_1V_2V_3$,*r*)].

- **Propiedad 5.23:** Si $[E \cap F \neq \emptyset]$ entonces existe al menos un tetraedro $T_j=V_0V_1V_2V_3$, tal que $T_j \in C_F$, y que cumple que [*C in AIE*($V_1V_2V_3$,*r*)].

- **Propiedad 5.24:** Si [*C in* F] entonces [E *in* F]*.

Estas propiedades nos permiten desarrollar un algoritmo de detección de colisión entre una esfera y un poliedro que compruebe en primer lugar si el centro de la esfera se encuentra en el poliedro, y si no es así, que compruebe la intersección de la esfera con las caras del poliedro en cuya área de influencia extendida se encuentre el centro de la esfera.

A continuación mostramos un lema que nos permite aprovechar la coherencia temporal entre frames consecutivos para el caso que nos ocupa:

Lema 5.2: Sea F un poliedro con recubrimiento $C_F=\{T_i\}$, $i=1..n$. Sea E una esfera de centro *C* y radio *r*. Si $[C \text{ in } F]_j$ y $[sign(\alpha_i(C))]_j=[sign(\alpha_i(C))]_{j+1}$ para todo $i=1..n$, entonces $[E \text{ in } F]_{j+1}$. De igual modo, si se cumple que $[not\ (C \text{ in } F)]_j$ y que $[E \cap F \neq \emptyset]_{j+1}$ entonces debe existir algún tetraedro $T_i=V_0V_1V_2V_3$ tal que $T_i \in C_F$ y que $[V_1V_2V_3 \cap E \neq \emptyset]_{j+1}$.

Demostración: Si el centro *C* de la esfera E se encuentra en el poliedro F y no cambia el signo de la coordenada baricéntrica α respecto a ninguno de los tetraedros del recubrimiento entre frames consecutivos, *C* seguirá estando en el interior del poliedro debido al Lema 5.1. Al estar *C* dentro del poliedro, o bien lo está la esfera completa o bien la esfera interseca con alguna de las caras del poliedro. Si *C* no está en el interior del poliedro, para que la esfera interseque con el poliedro, debe existir un triángulo no original de un tetraedro del recubrimiento que interseque con la esfera. □

* Evidentemente [E *in* F] es equivalente a $[E \cap F \neq \emptyset \vee ((C \text{ in } F) \wedge (E \cap F = \emptyset))]$

Es más, conociendo la particularidad de que en los poliedros formados por caras planas un punto tiene el mismo signo en su coordenada baricéntrica α respecto a todos los tetraedros del recubrimiento de una cara, sólo es necesario utilizar un tetraedro del recubrimiento de la cara en representación de toda la cara, pues el signo de α respecto a ese tetraedro determinará el lado respecto al plano soporte de la cara en el que se encuentra ese punto.

5.4.7. Detección de Colisión Esfera/Poliedro

Una diferencia fundamental entre el algoritmo de detección de colisión (Algoritmo 5.4) y el algoritmo de inclusión (Algoritmo 5.1) radica en la utilización de tetra-trees para obtener un conjunto menor de tetraedros sobre los que actuar.

A continuación resumimos los pasos más importantes del algoritmo:

- En primer lugar, se detecta si se produce intersección entre la esfera envolvente del poliedro y la esfera, y a continuación se realiza la detección de colisión entre el centro de la esfera y el poliedro, utilizando el algoritmo de *detección de colisión punto/poliedro* (Algoritmo 5.2).

- El siguiente paso consiste en obtener la *lista de tetra-conos* en los que se encuentra la esfera (Figura 5.28). Para ello se utilizan los tetra-conos extendidos, obteniendo la inclusión del centro *C* de la esfera en dichos tetra-conos extendidos. Este conjunto de tetra-conos se reduce descartando aquellos en cuyo tetraedro envolvente extendido de tamaño *r* no se encuentra *C*. Una vez obtenido este conjunto reducido de tetra-conos, por una parte se obtiene el conjunto de caras cuyos tetraedros del recubrimiento están incluidos en el conjunto de tetra-conos, y por otra para cada cara se obtiene el conjunto de tetraedros del recubrimiento que la forman incluidos en ese conjunto de tetra-conos.

- Por último se realiza un test de *intersección esfera/cara* para el conjunto de caras obtenidas. Para dicho test se utilizan únicamente los tetraedros incluidos en el correspondiente conjunto de tetra-conos, no teniendo en cuenta al resto, aunque pertenezcan al recubrimiento de la correspondiente cara.

Hemos constatado que no es eficiente guardar este conjunto de tetra-conos entre frames, para que el algoritmo comience por comprobar la colisión de la esfera con los elementos de dichos tetra-conos al principio del mismo. Esto es debido a que el conjunto de tetra-conos cambia considerablemente entre frames cuanto mayor es la

esfera y mayor es la profundidad del tetra-tree. Almacenar los tetra-conos de niveles intermedios tampoco resuelve el problema, pues se añade un coste adicional al incrementar el número de tetraedros a tratar, lo que hace que disminuya la eficiencia del algoritmo.

```
static resultadoAnterior = false

bool poliedro::testDeteccionColision(esfera E) {
   C = E.centro()
   R = E.radio()

   // Interseccion entre esferas envolventes
   if (E.interseccion(this.esferaEnvolvente())==false) return false

   // Detección de colisión con el centro de la esfera
   resultado = this.testDeteccionColision(C)
   if (resultado==EQUAL_STATE && resultadoAnterior==true) return true
   resultadoAnterior = resultado
   if (resultado==true) return true

   // Obtener la lista de Tetra-Conos en los que está la esfera
   LTC = this.obtenerListaTetraConosExtendidosInclusion(C,R)

   // Eliminar los Tetra-Conos en cuyo Tetraedro envolvente no se encuentre la esfera
   for (Tc=0;Tc<LTC.end();Tc++)
      if (LTC(Tc).tetraedroEnvolvente.extension(R).inclusion(C)==false)
         LTC(Tc).borrar()

   // Obtener la lista de caras incluidas en los tetraconos.
   // Cada cara tiene una lista de tetraedros incluidos en el conjunto de tetraconos LTC
   LF = this.obtenerListaCarasConSusTetraedros(LTC)

   // Cálculo de la intersección esfera/cara con las caras en cuya AIE se encuentra C
   for (F=0;F<LF.numeroCaras();F++)
      if (sign(LF.cara(F).tetraedroRepresentativo().AIE(C,R)) >= 0)
         if (E.interseccion(LF.cara(F)) == true) return true
   return false
}
```

Algoritmo 5.4: Detección de Colisión Esfera/Poliedro.

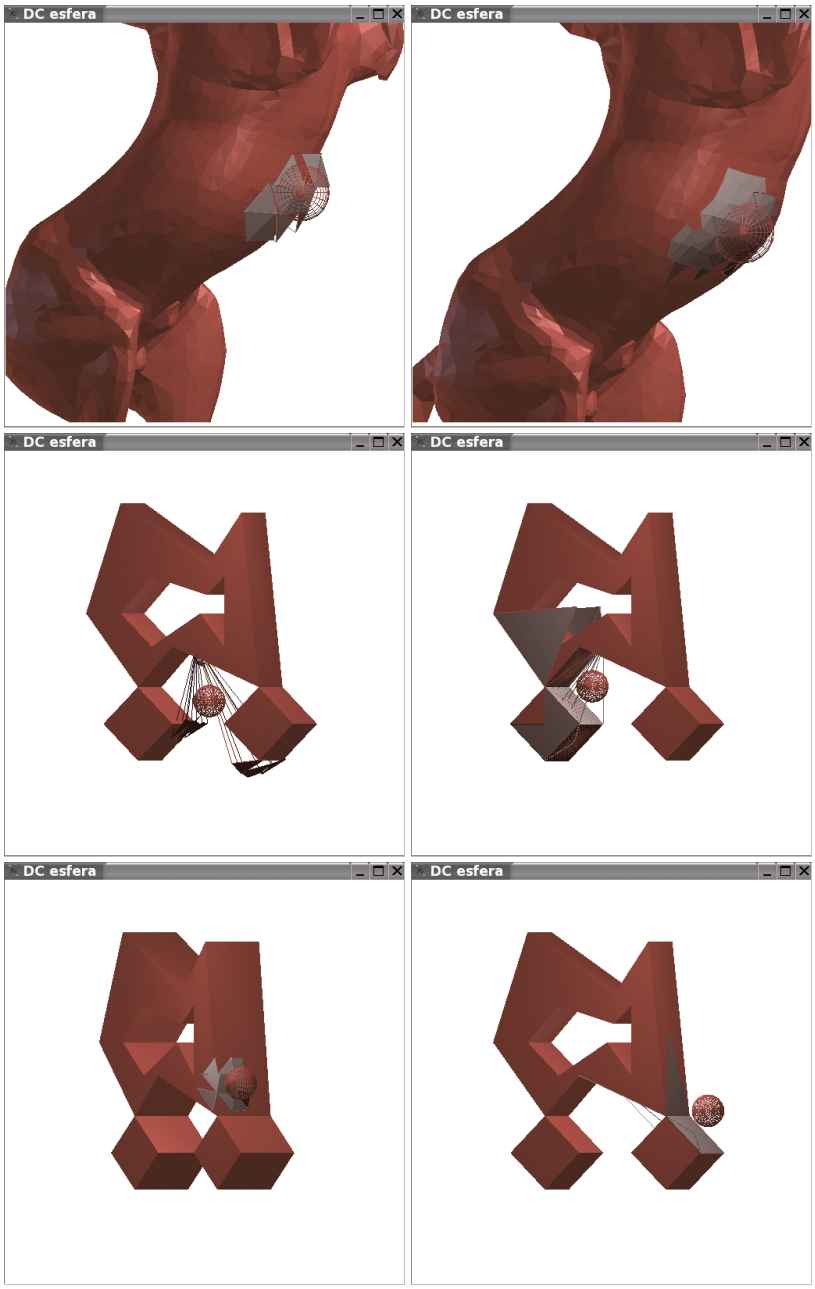


Figura 5.28: Detección de colisión esfera/poliedro. En la parte izquierda se muestran los tetraedros envolventes asociados a los tetra-conos en los que se encuentra la esfera. En la parte derecha se muestran los tetraedros del recubrimiento implicados (clasificados en el conjunto de tetra-conos correspondiente).

5.5. Algoritmo de Detección de Colisión Poliedro/Poliedro

El objetivo final de este trabajo consiste en la elaboración de un algoritmo de detección de colisión entre poliedros. Perseguimos que éste sirva para objetos poliédricos complejos sin que sea necesario descomponerlos en partes convexas como la mayor parte de las soluciones aportadas por otros investigadores.

Para llevar a buen término este algoritmo ha sido necesario desarrollar pequeños algoritmos auxiliares que resuelven problemas muy concretos, como la intersección entre un segmento y un triángulo, de cuya eficiencia depende el tiempo de cálculo del algoritmo final. Estos algoritmos además de ser eficientes deben ser robustos en la medida de lo posible. A continuación veremos el conjunto de algoritmos soporte desarrollados para la DC, como los algoritmos de intersección segmento/cara compleja, segmento/tetraedro o tetraedro/tetraedro, enfocados todos ellos a su utilización posterior en el algoritmo de detección de colisión entre poliedros.

En líneas generales, el algoritmo de detección de colisión entre poliedros se basa en recorrer las jerarquías de tetra-trees de ambos poliedros, para obtener finalmente los tetra-conos involucrados en la colisión debido a que sus tetraedros y esferas envolventes asociados intersecan, reduciendo así el número de pares de tetra-conos sobre los que tratar, y reduciendo por tanto el número de características de los poliedros sobre las que comprobar su intersección. Aprovecharemos también la coherencia para no repetir cálculos innecesarios. Finalmente mostraremos otro método basado en el algoritmo de detección de colisión esfera/poliedro, método adecuado cuando la diferencia entre el tamaño relativo de los objetos es grande.

Veamos en primer lugar el nuevo algoritmo de intersección entre un segmento y un triángulo, método básico para los algoritmos que veremos a lo largo de las siguientes secciones. Por otra parte, el algoritmo de intersección segmento/cara compleja será el utilizado finalmente en el algoritmo de detección de colisión a nivel de tetra-conos, para determinar la intersección entre poliedros. El algoritmo de intersección segmento/tetraedro y el de intersección tetraedro/tetraedro se utilizarán para obtener los tetraedros envolventes que interfieren. Finalmente el algoritmo de intersección entre un segmento y un poliedro nos indicará cómo utilizar los algoritmos anteriores para poder construir un algoritmo de intersección entre poliedros, restringido únicamente a las características de los tetra-conos que intersecan de ambos poliedros.

5.5.1. Intersección segmento/triángulo

Existen numerosos problemas en los que debe calcularse la intersección entre un rayo o un segmento y un triángulo. Estos cálculos son necesarios en ray-casting, ray-tracing [JS00] [JSF01], tests de inclusión, operaciones booleanas entre sólidos, y en el caso que nos ocupa de detección de colisiones entre sólidos. En este último, dicho test es necesario para la comprobación de intersección entre las caras de dos poliedros, algoritmo que en última instancia debe comprobar la intersección segmento/cara. En definitiva, en este tipo de aplicaciones dicho test debe realizarse numerosas veces, por lo que debe estar optimizado.

Existen buenas soluciones a este problema. Badouel [Bad90] diseñó una solución basada en el estudio de la intersección entre el rayo y el plano del triángulo, su proyección sobre el plano *xy*, *yz* ó *zx*, y el estudio de la inclusión del punto en el triángulo utilizando coordenadas baricéntricas.

Por otra parte, Möller [MT97] desarrolló un algoritmo basado en la resolución de un sistema de ecuaciones formado por la ecuación del rayo y la ecuación del punto de intersección entre el rayo y el triángulo, utilizando para ello las coordenadas baricéntricas del triángulo.

Segura [SF01] propuso un algoritmo de intersección segmento/triángulo. Este método realiza el estudio del signo del volumen de los tetraedros formados por los vértices del triángulo y los extremos del segmento. El algoritmo determina si se produce o no intersección, pero el punto de intersección debe calcularse mediante un algoritmo clásico de intersección rayo/plano [JSF01].

Proponemos un nuevo algoritmo para la determinación de intersección entre un segmento y un triángulo basado en las coordenadas baricéntricas, y especialmente apropiado para la detección de colisión entre sólidos, debido a sus características especiales [JSFO06].

La idea básica del algoritmo consiste en obtener el signo de las coordenadas baricéntricas de un extremo del segmento respecto al tetraedro obtenido uniendo los vértices del triángulo y el otro extremo del segmento. La eficacia de este método viene dada por la optimización realizada, debido sobre todo al orden de los cálculos, a la eliminación de cálculos innecesarios, y a la posibilidad de almacenamiento de aquellos que pueden ser reutilizados.

El siguiente teorema da soporte a este método (Algoritmo 5.5), en el que hemos utilizado la nomenclatura de la Figura 5.29. Este algoritmo puede verse como una generalización del algoritmo de Segura et. al [SF01] que utiliza volúmenes signados de tetraedros. En el algoritmo que presentamos en esta sección utilizamos las coordenadas baricéntricas de un punto respecto a un tetraedro, en lugar de hacerlo respecto a un triángulo.

```
bool triangulo::interseccion(segmento Q1Q2)
     // El triángulo es V1V2V3
     A = Q1 – V3
     B = V1 – V3
     C = V2 – V3
     W1 = B x C
     w = A · W1
     D = Q2 – V3
     s = D · W1
     if ( w > ε ) {
         if ( s > ε ) return false
         W2 = A x D
         t = W2 · C
         if ( t < -ε ) return false
         u = - W2 · B
         if ( u < -ε ) return false
         if ( w < s + t + u ) return false
     } else if (w < -ε) {
         if ( s < -ε ) return false
         W2 = A x D
         t = W2 · C
         if ( t > ε ) return false
         u = - W2 · B
         if ( u > ε ) return false
         if ( w > s + t + u ) return false
     } else { // w=0, intercambio de Q1 y Q2
         if ( s > ε ) {
             W2 = D x A
             t = W2 · C
             if ( t < -ε ) return false
             u = - W2 · B
             if ( u < -ε ) return false
             if ( -s < t + u ) return false
         } else if ( s < -ε ) {
             W2 = D x A
             t = W2 · C
             if ( t > ε ) return false
             u = - W2 · B
             if ( u > ε ) return false
             if ( -s > t + u ) return false
         } else return false // w=0, s=0, segmentos coplanarios
     }
     // Cálculos para obtener el punto de intersección si fuese necesario
     //tt = 1 / (w – s)      //t_param = w / div
     //α = tt * t
     //β = tt * u
     return true
}
```

Algoritmo 5.5: Intersección Segmento/Triángulo.

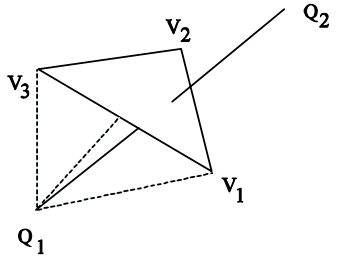


Figura 5.29: Creación de un tetraedro auxiliar para el cálculo de intersección segmento/triángulo.

Teorema 5.3: Sea $V_1V_2V_3$ un triángulo y sea Q_1Q_2 un segmento tal que Q_1 no es coplanario con el plano soporte de $V_1V_2V_3$. El segmento Q_1Q_2 interseca con el triángulo $V_1V_2V_3$, es decir, $[Q_1Q_2 \cap V_1V_2V_3 \neq \emptyset]$ sii $[Q_2 \; in \; \angle\backslash Q_1V_1V_2V_3]$ o lo que es lo mismo $sign(\alpha(Q_2))\leq 0 \wedge sign(\beta(Q_2))\geq 0 \wedge sign(\gamma(Q_2))\geq 0 \wedge sign(\delta(Q_2))\geq 0$ respecto de $Q_1V_1V_2V_3$.

Demostración: Los puntos Q_1 y Q_2 están situados uno a cada lado del plano soporte de $V_1V_2V_3$, o Q_2 está sobre dicho plano, debido a que $sign(\alpha(Q_2))\leq 0$. Además el punto de intersección debe estar en el mismo lado que V_1 respecto al plano soporte de $Q_1V_3V_2$, o sobre dicho plano soporte, pues $sign(\beta(P))\geq 0$; y en el mismo lado que V_2 respecto al plano soporte de $Q_1V_3V_1$, o sobre dicho plano, debido a que $sign(\gamma(P))\geq 0$; asimismo estará en el mismo lado que V_3 respecto del plano soporte de $Q_1V_2V_1$, o sobre dicho plano, pues $sign(\delta(Q_2))\geq 0$. □

Basándonos en el teorema anterior, se han optimizado los cálculos utilizados como puede verse en el algoritmo. Estos cálculos son los mostrados a continuación:

Sean las coordenadas baricéntricas:

$$\alpha = \frac{|Q_2V_1V_2V_3|}{|Q_1V_1V_2V_3|} = \frac{\det(Q_2V_1V_2V_3)}{\det(Q_1V_1V_2V_3)} \quad \beta = \frac{|Q_2Q_1V_3V_2|}{|Q_1V_1V_2V_3|} = \frac{\det(Q_2Q_1V_3V_2)}{\det(Q_1V_1V_2V_3)}$$

$$\gamma = \frac{|Q_2Q_1V_1V_3|}{|Q_1V_1V_2V_3|} = \frac{\det(Q_2Q_1V_1V_3)}{\det(Q_1V_1V_2V_3)} \quad \delta = \frac{|Q_2Q_1V_2V_1|}{|Q_1V_1V_2V_3|} = \frac{\det(Q_2Q_1V_2V_1)}{\det(Q_1V_1V_2V_3)}$$

Podemos calcular el área signada de la siguiente forma:

$$a_0 = |Q_1V_1V_2V_3| = -|V_3Q_1V_1V_2| = -|(Q_1 - V_3) \quad (V_1 - V_3) \quad (V_2 - V_3)|$$

$$a_1 = |Q_2V_1V_2V_3| = -|V_3Q_2V_1V_2| = -|(Q_2 - V_3) \quad (V_1 - V_3) \quad (V_2 - V_3)|$$

$$a_2 = |Q_2Q_1V_3V_2| = -|V_3Q_2V_2Q_1| = -|(Q_2 - V_3) \quad (V_2 - V_3) \quad (Q_1 - V_3)|$$

$$a_3 = |Q_2Q_1V_1V_3| = -|V_3Q_2Q_1V_1| = -|(Q_2 - V_3) \quad (Q_1 - V_3) \quad (V_1 - V_3)|$$

$$a_4 = |Q_2Q_1V_2V_1| = -|V_1Q_2Q_1V_2| = -|(Q_2 - V_1) \quad (Q_1 - V_1) \quad (V_2 - V_1)|$$

Si llamamos:

$$A = Q_1 - V_3 \quad B = V_1 - V_3 \quad C = V_2 - V_3 \quad D = Q_2 - V_3$$
$$E = Q_2 - V_1 \quad F = Q_1 - V_1 \quad G = V_2 - V_1$$

obtenemos:

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \frac{1}{-\begin{vmatrix} A & B & C \end{vmatrix}} \cdot \begin{bmatrix} -\begin{vmatrix} D & B & C \end{vmatrix} \\ -\begin{vmatrix} D & C & A \end{vmatrix} \\ -\begin{vmatrix} D & A & B \end{vmatrix} \\ -\begin{vmatrix} E & F & G \end{vmatrix} \end{bmatrix} = \frac{1}{A \cdot (B \times C)} \cdot \begin{bmatrix} D \cdot (B \times C) \\ -(D \times A) \cdot C \\ (D \times A) \cdot B \\ (E \times F) \cdot G \end{bmatrix}$$

además, como Q_1 no es coplanario con $V_1V_2V_3$ podemos decir que $a_0 \neq 0$ y por tanto:

$$sign(a_i / a_0) = sign(a_i) / sign(a_0) = sign(a_i) \cdot sign(a_0), i = 1..4$$

tenemos:

$$\begin{bmatrix} sign(\alpha) \\ sign(\beta) \\ sign(\gamma) \\ sign(\delta) \end{bmatrix} = sign\left[A \cdot (B \times C)\right] \cdot \begin{bmatrix} sign\left[D \cdot (B \times C)\right] \\ sign\left[-(D \times A) \cdot C\right] \\ sign\left[(D \times A) \cdot B\right] \\ sign\left[(E \times F) \cdot G\right] \end{bmatrix} = sign(w) \cdot \begin{bmatrix} sign(s) \\ sign(t) \\ sign(u) \\ sign(v) \end{bmatrix}$$

siendo:

$$s = D \cdot (B \times C)\,,\ t = -(D \times A) \cdot C\,,\ u = (D \times A) \cdot B\,,$$
$$v = (E \times F) \cdot G \text{ y } w = A \cdot (B \times C)$$

además, como:

$$\alpha + \beta + \gamma + \delta = 1$$

por tanto:

$$\frac{s}{w} + \frac{t}{w} + \frac{u}{w} + \frac{v}{w} = 1$$
$$\text{v} = \text{w} - \text{s} - \text{t} - \text{u}$$

por este motivo no es necesario que el algoritmo calcule $(E \times F) \cdot G$, con el consiguiente ahorro computacional.

Este algoritmo presenta una serie de ventajas que lo hacen adecuado para el caso de detección de colisión. Entre estas ventajas podemos destacar las siguientes [JSFO06]:

- Se contempla el caso de que Q_1 sea coplanario con $V_1V_2V_3$, de manera que el algoritmo devuelve un resultado correcto. Cuando Q_1 es coplanario, se intercambian los papeles de Q_1 y Q_2. Si nuevamente Q_2 es coplanario, el algoritmo resuelve que Q_1Q_2 es un segmento coplanario con $V_1V_2V_3$.

- Este algoritmo permite obtener el punto exacto de intersección si fuese necesario, calculando el parámetro *t* o las coordenadas baricéntricas del punto de intersección. Para la detección de colisión no es necesario calcular este punto y por eso en el algoritmo hemos sombreado y puesto bajo la forma de un comentario estos cálculos, esto puede hacerse debido a que ningún otro cálculo depende de éstos.
- Para poliedros que utilizan un recubrimiento simplicial o para mallas de triángulos, se pueden reutilizar muchos de los cálculos involucrados, pues son comunes a las aristas compartidas por los triángulos que forman parte del poliedro.

Además de lo anterior, este algoritmo tiene una serie de propiedades que además de hacerlo más robusto lo hacen eficiente para otras aplicaciones distintas a la detección de colisión:

- El vector normal al triángulo se calcula al principio del algoritmo, como en los algoritmos de Badouel y Segura. Si este valor se calcula y almacena previamente, dependiendo de la aplicación concreta, pueden ahorrarse bastantes cálculos.
- Es un algoritmo más robusto que otros, en cuanto a que no es necesario realizar ninguna división si lo único que deseamos es comprobar si se produce o no intersección. Además las comparaciones con el valor cero se realizan en un intervalo $[+\varepsilon,-\varepsilon]$ lo que hace que sea menos propenso a errores [GSS89].

5.5.2. Intersección segmento/cara

El método clásico de intersección entre un segmento y una cara compleja (polígono no convexo) está basado en el cálculo del punto de intersección entre el segmento y el plano en el que se encuentra el polígono y la posterior comprobación de la inclusión del punto en el polígono, aplicando el algoritmo de crossing-count [Hai94] tras realizar una proyección a 2D.

Para comprobar si se produce intersección entre un segmento cualquiera y una cara compleja, vamos a utilizar un método basado en la suma signada de las intersecciones del segmento con cada uno de los triángulos no originales que forman parte de los tetraedros del recubrimiento de la cara. Este método está basado en el descrito por Segura [SF98], el cual ha sido modificado para adaptarlo al uso de coordenadas baricéntricas respecto a los tetraedros del recubrimiento del poliedro, en lugar de respecto a los triángulos del recubrimiento de la cara. También ha sido optimizado en el sentido de que se utiliza el algoritmo de *intersección*

segmento/triángulo (Algoritmo 5.5) visto en la sección previa, el cual ha sido modificado y adaptado a este caso concreto para reducir el número de cálculos necesarios.

Teorema 5.4: Sea F un poliedro y H una cara de F. Sea C_H el recubrimiento de H formado por el conjunto de tetraedros $\{T_i\}$, $i=1,\ldots,n_k$, con $T_i=V_0V_1V_{i,2}V_{i,3}$. Entonces para todo segmento Q_1Q_2 no coplanario con H se produce intersección entre dicho segmento y la cara H, es decir, $[Q_1Q_2 \cap H \neq \emptyset]$ si y solamente si se cumple alguna de las siguientes condiciones:

a) $[Q_1Q_2 \cap H \neq V_{i,1}] \wedge sign(H) \cdot \sum_{i=1}^{n_k} sign(T_i) \cdot Inter(Q_1Q_2, V_1V_{i,2}V_{i,3}) = 2$

siendo $Inter(Q_1Q_2, V_1V_{i,2}V_{i,3})$ una función que devuelve (Figura 5.30):

- 0 si no se produce intersección entre Q_1Q_2 y el triángulo $V_1V_{i,2}V_{i,3}$.
- 1 si se produce intersección entre Q_1Q_2 y alguna de las aristas que no son frontera $V_1V_{i,2}$ ó $V_{i,3}V_1$, o en los vértices frontera $V_{i,2}$ ó $V_{i,3}$.
- 2 si se produce intersección entre Q_1Q_2 y el interior de $V_1V_{i,2}V_{i,3}$, o en la arista frontera $V_{i,2}V_{i,3}$.

b) $[Q_1Q_2 \cap H=V_{i,1}] \wedge [V_{i,1}\ in\ H]$.

c) $[Q_1Q_2 \cap V_{i,2}V_{i,3} \neq \emptyset]$.*

Demostración: La intersección entre el plano en el que se encuentra una cara y un segmento no coplanario con ella es un punto, por tanto determinar si se produce intersección entre el segmento y la cara es equivalente a determinar si se produce la inclusión de dicho punto en la cara, de manera similar al caso 2D. Hay que tener en cuenta el signo de la cara, o lo que es lo mismo, su orientación respecto al origen del recubrimiento, pues dependiendo de esto, los tetraedros del recubrimiento tendrán un signo u otro; por este motivo es necesario multiplicar finalmente por el signo de la cara para obtener un resultado correcto. □

Este método puede optimizarse para el caso de caras convexas, pues bastaría con encontrar la intersección del segmento con alguno de los triángulos que forman parte del recubrimiento de la cara, deduciendo de este modo que se produce intersección entre el segmento y la cara.

Hemos acelerado los cálculos para el caso en el que se produce la intersección en la arista $V_{i,2}V_{i,3}$, pues en esta situación tiene lugar intersección entre el segmento y la cara, pues la arista $V_{i,2}V_{i,3}$ (incluyendo los vértices) está en la frontera de dicha cara (Figura 5.30.a).

* Este caso incluye la intersección del segmento Q_1Q_2 y los vértices $V_{i,2}$ ó $V_{i,3}$.

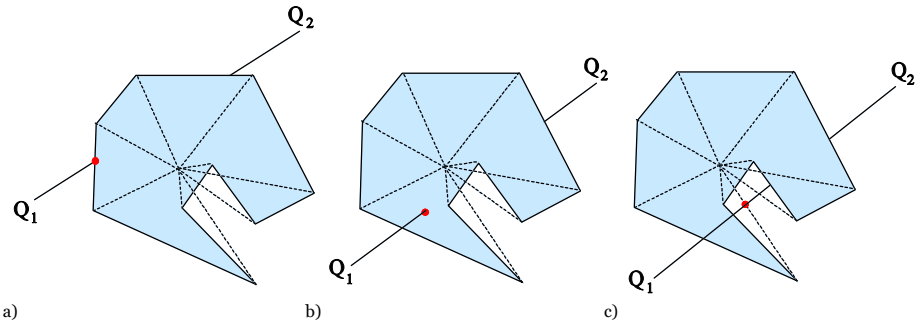


Figura 5.30: Intersección segmento/cara compleja. a) Se produce intersección en una arista frontera. El algoritmo no comprueba la intersección con otros triángulos del recubrimiento. b) Se produce intersección en el interior del triángulo no original de un tetraedro del recubrimiento. c) Se produce intersección en una arista no frontera.

Al igual que en el caso de la inclusión de puntos en 2D, cuando se produce intersección en el origen del recubrimiento de la cara (vértice V_1), el algoritmo devuelve el valor de inclusión previamente almacenado para ese punto, pues utilizando este método no es posible determinar el estado de inclusión del punto cuando se da esta circunstancia.

Es posible que el triángulo no original del tetraedro del recubrimiento de la cara sea degenerado. En esta situación es necesario comprobar la intersección de la arista frontera de dicho tetraedro con el segmento considerado (Figura 5.31).

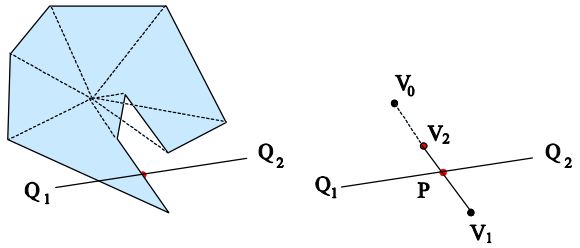


Figura 5.31: Intersección de Q_1Q_2 con la arista V_1V_2 del triángulo degenerado $V_0V_1V_2$.

Veamos a continuación de forma resumida el algoritmo desarrollado para determinar si se produce intersección entre un segmento y una cara compleja (Algoritmo 5.6):

- Antes de calcular la intersección entre el segmento y la cara, se comprueba la posición relativa de los extremos del segmento respecto al plano soporte del triángulo no original de uno de los tetraedros del recubrimiento de la cara. En el caso de que ambos extremos estén sobre el plano, se comprueba si se produce la intersección bidimensional entre el segmento y la cara, tras proyectar sus vértices a uno de los planos x=0, y=0 ó z=0.

- Si ambos extremos se encuentran en el mismo lado respecto al plano soporte de la cara deducimos que no se produce intersección.

- En el resto de los casos, debemos determinar si alguno de los extremos del segmento se encuentra sobre el plano soporte de la cara, para no utilizarlo en la formación de tetraedros auxiliares (Figura 5.32). Sea Q_1 el extremo que no se encuentra sobre el plano, formaremos el tetraedro $Q_1V_1V_2V_3$ para aplicar el Teorema 5.3 y así obtener si se produce o no intersección entre el segmento y cada uno de los triángulos no originales del recubrimiento de la cara. Finalmente aplicamos el Teorema 5.4 para obtener si se ha producido intersección entre el segmento y la cara.

```
bool cara::interseccion(segmento Q1Q2) {
   // Calcular la posición de los extremos del segmento respecto al plano de la cara
   this.tetrahedro(0).V1V2V3.calcularPosicion(Q1Q2, w, s)
   If ((w > -ε && s < ε) || (w < ε && s > -ε)) // Distinto signo, incluye (w = 0 y s = 0)
      If (w < -ε || w > ε) // No hacer nada: w <> 0
      else
         if (s < -ε || s > ε) // s <> 0 se produce el intercambio de los extremos del segmento
            Intercambiar(Q1,Q2)
         else  // w = 0, s = 0 se realiza una intersección cara/segmento en 2D
            return this.interseccion2D(Q1Q2)

   // Realizar la intersección utilizando los tetraedros T1=V1,0V1,1V1,2V1,3 de CH
   suma = 0
   For (t=0;t<this.numero_tetraedros();t++) {
      if (this.tetraedro(t).V1V2V3.degenerado()==true)
         if (interseccion(this.tetraedro(t).V2V3, Q1Q2)==true) return true
      else
         switch (this.tetraedro(t).V1V2V3.interseccionModificado(Q1Q2)) {
            1: suma += 2*sign(this.tetraedro(t))          // En el interior de V1V2V3
            2: suma += sign(this.tetraedro(t))            // Aristas V1V2 o V1V3
            3: return true                                // Arista V2V3, vért. V2 o V3
            4: return this.tetraedro(t).V1.inclusion()    // Vértice V1
         }
   }
   If (suma*sign(this)==2) return true
   else return false
}
```

Algoritmo 5.6: Intersección segmento/cara compleja.

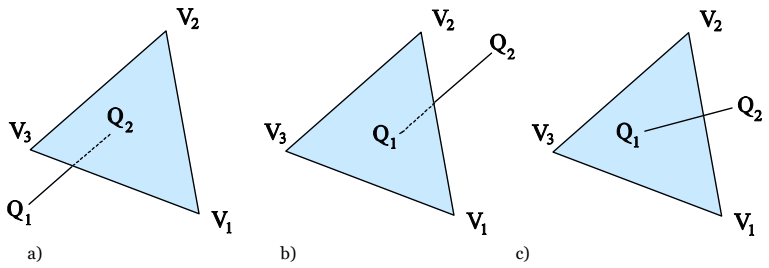


Figura 5.32: Casos especiales en la intersección segmento/triángulo. a) Q_2 se encuentra en el mismo plano que el triángulo. b) Q_1 se encuentra en el mismo plano que el triángulo. c) Q_1 y Q_2 se encuentran en el mismo plano que el triángulo.

El algoritmo de *intersección segmento/triángulo* visto en la sección anterior (Algoritmo 5.5) ha sido modificado y optimizado para la intersección segmento/cara compleja, de manera que no compruebe si el extremo Q_1 del segmento se encuentra en el mismo plano que el triángulo, con el consiguiente intercambio de papeles entre Q_1 y Q_2. Esta comprobación debe realizarse una sola vez para la cara y no para todos los triángulos no originales de los tetraedros del recubrimiento de la cara. Por tanto, dicha comprobación es realizada una sola vez en el propio algoritmo de intersección segmento/cara compleja (Algoritmo 5.6).

Los cálculos realizados por el antiguo algoritmo de intersección segmento/triángulo (Algoritmo 5.5) se basaban en el cálculo de las variables w y s (volúmenes signados de los tetraedros $Q_1V_1V_2V_3$ y $Q_2V_1V_2V_3$ respectivamente). Los valores de w y s son únicos para cada par segmento/triángulo del recubrimiento de una cara. Sin embargo su signo es el mismo para todos los pares formados por un segmento y los distintos triángulos no originales del recubrimiento de una cara. Debido a esto, el algoritmo de intersección segmento/cara compleja (Algoritmo 5.6) comprueba inicialmente la posición de los extremos del segmento respecto al plano soporte de la cara por medio de un único tetraedro de su recubrimiento, utilizando para ello los valores del signo de w y s de ese tetraedro.

A continuación mostramos este algoritmo de *intersección segmento/triángulo modificado* (Algoritmo 5.7) en el que se conoce a priori que los extremos del segmento están cada uno a un lado del plano. Este algoritmo modificado devuelve además si se produce intersección en una arista frontera del tetraedro. Hemos incluido también el algoritmo que determina a qué lado del plano se encuentra el segmento (cálculo de las variables w y s), así como el algoritmo de intersección segmento/cara compleja bidimensional.

```
void triangulo::calcularPosicion(segmento Q1Q2 real & w, real & s) {
   // El triangulo es V1V2V3
   A = Q1 – V3
   B = V1 – V3
   C = V2 – V3
   W1 = B x C
   w = A · W1
   D = Q2 – V3
   s = D · W1
}

int triangulo::interseccionModificado(segmento Q1Q2) {
   // El triángulo es V1V2V3
   A = Q1 – V3
   B = V1 – V3
   C = V2 – V3
   W1 = B x C
   w = A · W1
   D = Q2 – V3
   if ( w > ε ) {
       W2 = A x D
       t = W2 · C
       if ( t < -ε ) return 0
       u = - W2 · B
       if ( u < -ε ) return 0
       s = D · W1
       v = s + t + u
       if ( w < v ) return 0
   } else {
       W2 = A x D
       t = W2 · C
       if ( t > ε ) return 0
       u = - W2 · B
       if ( u > ε ) return 0
       s = D · W1
       v = s + t + u
       if ( w > v ) return 0
   }
   if (t >= -ε && t <= ε) return 3                          // Arista V2V3, vért. V2 o V3
   if ((u >= -ε && u <= ε) && (v >= -ε && v <= ε)) return 4 // Vértice V1
   if ((u >= -ε && u <= ε) || (v >= -ε && v <= ε)) return 2 // Aristas internas V1V2 o V1V3
   return 1
}

bool cara::interseccion2D (segmento Q1Q2) {
   //Proyectar todos los puntos sobre el plano x=0, y=0 ó z=0
   segmento P1P2=Proyección(Q1Q2)
   For (t=0;t<this.num_tetraedros();t++) {
      Triangulo T = Proyección(this.tetraedro(t).V1V2V3)
      If (T.V2V3.Interseccion2D(P1P2)==true) return true
   }
   cara2D = Proyeccion(this)
   if (cara2D.inclusion2D(Q1)==true || cara2D.inclusion2D(Q2)==true) return true
   return false
}
```

Algoritmo 5.7: Intersección segmento/triángulo modificado y algoritmos auxiliares para la intersección segmento/cara compleja.

5.5.3. Intersección segmento/tetraedro

Para determinar si se produce intersección entre un segmento y la frontera de un tetraedro vamos a desarrollar un algoritmo basado en el estudio del signo de cada uno de los extremos del segmento respecto a los planos que definen al tetraedro. Este algoritmo es la base para el algoritmo de intersección entre tetraedros desarrollado en la siguiente sección.

Es evidente que dado un segmento Q_1Q_2 y un tetraedro T=$V_0V_1V_2V_3$, el segmento interseca con la frontera del tetraedro si y solamente si interseca con alguno de los triángulos que forman dicha frontera. No consideraremos el caso en el que el segmento se encuentre en el interior del tetraedro, pues este caso ya es tenido en cuenta en el algoritmo de intersección entre tetraedros que veremos en la siguiente sección.

A partir del enunciado anterior podemos obtener un algoritmo que compruebe la intersección del segmento con cada uno de los triángulos del tetraedro. Estos cálculos pueden optimizarse de manera que se compruebe la intersección con el mínimo número de caras del tetraedro, como veremos a continuación.

Además es posible descartar el segmento, es decir, determinar que no se produce colisión, en determinados casos. En aquellos en los que el segmento se encuentre en la parte exterior del plano soporte de uno de los triángulos que forman el tetraedro respecto del restante vértice del tetraedro.

Vamos a utilizar un enfoque similar al utilizado en el algoritmo de Cohen-Sutherland para el recorte de líneas. Utilizaremos para ello las coordenadas baricéntricas de los extremos del segmento respecto al tetraedro para determinar los códigos de región en el que se encuentra cada uno de dichos extremos. Asignaremos un código de 4 bits para cada una de las regiones, de manera que el primer bit identifique las regiones que quedan a cada uno de los lados del plano soporte del triángulo $V_1V_2V_3$, es decir, si $sign(\alpha(P))<0$ el bit será igual a 1 y 0 en caso contrario. Este mismo planteamiento es extendido a las coordenadas baricéntricas β, γ y δ del punto respecto al tetraedro, representando a los bits segundo, tercero y cuarto respectivamente (Figura 5.33).

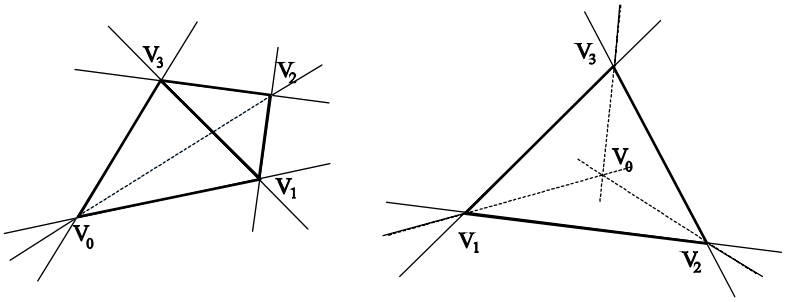


Figura 5.33: Regiones del espacio divididas por los planos que forman un tetraedro.

Utilizando esta codificación podemos determinar la situación de un segmento Q_1Q_2 respecto a un tetraedro $V_0V_1V_2V_3$ obteniendo los códigos de región de los extremos de dicho segmento, por medio de las coordenadas baricéntricas (Algoritmo 5.8):

- Si los dos puntos Q_1 y Q_2 tienen el valor de 1 en el mismo bit, los dos puntos están en el lado externo del plano, es decir, no intersecan con el tetraedro.

- Si los dos puntos Q_1 y Q_2 están en la región 0000 podemos decir que el segmento no interseca con el tetraedro, además el segmento se encuentra dentro del tetraedro.

- Si sólo uno de los puntos Q_1 ó Q_2 está en la región 0000 podemos decir que el segmento interseca con el tetraedro.

- En el resto de los casos hay que comprobar la intersección del segmento con los triángulos que forman el tetraedro, representados sus extremos por el correspondiente código de región. Basta con realizar esta comprobación con el conjunto de triángulos asociados al código de región con menor número de unos, que como máximo estará formado por dos triángulos.

El algoritmo descrito realiza aproximadamente el mismo número de cálculos que otro algoritmo destinado al mismo fin, el cual compruebe la intersección del segmento con los cuatro triángulos de un tetraedro. Esto es debido a que es necesario calcular en primer lugar las coordenadas baricéntricas de los extremos del segmento respecto al tetraedro, y posteriormente realizar la intersección del segmento con dos triángulos del tetraedro como máximo. El algoritmo aquí expuesto tiene la ventaja de que permite obtener trivialmente la intersección de ciertos segmentos con pocos cálculos. Además, como veremos a continuación, puede extenderse fácilmente para obtener eficientemente si se produce intersección entre dos tetraedros.

```
bool tetraedro::interseccion(segmento Q1Q2) {
   bits array[4][2] bit

   bit0,i = { sign(this.α(Q1))<0, sign(this.α(Q2))<0 }
   if (bit0,i == 11) return false
   bit1,i = { sign(this.β(Q1))<0, sign(this.β(Q2))<0 }
   if (bit1,i == 11) return false
   bit2,i = { sign(this.γ(Q1))<0, sign(this.γ(Q2))<0 }
   if (bit2,i == 11) return false
   bit3,i = { sign(this.δ(Q1))<0, sign(this.δ(Q2))<0 }
   if (bit3,i == 11) return false

   if (biti,0 == 0000 && biti,1 == 0000) return false
   if (biti,0 == 0000 || biti,1 == 0000) return true

   if (this.interseccion(biti,0, biti,1, Q1Q2)==true) return true

   return false
}

bool tetraedro::interseccion(bits b0, bits b1, segmento Q1Q2) {
   if (b0 & b1 == 0000) {
      this.obten_minimo_numero_triangulos(b0, b1, num_tri, lista_tri)
      for (j=0;j<num_tri;j++)
         if (lista_tri(j).interseccion(Q1Q2)==true) return true
   }
   return false
}

void tetraedro::obten_minimo_numero_triangulos(bits b0, bits b1, int & n, triangulo & listaT)
{
   n=0
   if (suma(b0)<suma(b1)) b=b0
   else b=b1

   for (i=0;i<4;i++)
      if (b[i]=1) {
         listaT.insertar(this.triangle(i))
         n++;
      }
}
```

Algoritmo 5.8: Intersección segmento/tetraedro.

5.5.4. Intersección tetraedro/tetraedro

Podemos utilizar un algoritmo de *intersección segmento/tetraedro* para diseñar un algoritmo de *intersección tetraedro/tetraedro* iterándolo con todas las aristas de un tetraedro. Para ello es necesario comprobar la intersección de las seis aristas de un tetraedro con el otro y viceversa. Como este algoritmo es claramente ineficiente vamos a diseñar un algoritmo específico, extendiendo el algoritmo de intersección

segmento/tetraedro visto en la sección anterior. Este algoritmo de intersección entre tetraedros será utilizado sobre todo para descartar pares de tetraedros envolventes. Utilizaremos un enfoque similar al utilizado en la Sección 5.3 para el Algoritmo 5.8. En primer lugar determinaremos el código de región de los vértices de un tetraedro (T_2=$V_0V_1V_2V_3$) respecto al otro (T_1). Utilizando estos códigos podemos determinar la situación de los vértices de T_2 respecto a T_1 y actuar como sigue a continuación:

- Si los cuatro vértices V_0,V_1,V_2,V_3 tienen el valor 1 en el mismo bit, los vértices (y el tetraedro) están en el lado externo del plano que representa dicho bit, es decir, no se produce la intersección de los tetraedros.

- Si los cuatro vértices están en la región 0000 podemos decir que el tetraedro T_2 no interseca con el tetraedro T_1, pero que T_2 se encuentra dentro de T_1.

- Si uno, dos o tres vértices están en la región 0000 podemos decir que se produce intersección entre tetraedros.

- En el resto de los casos se realiza la operación AND bit a bit entre los códigos de región de cada dos vértices. En el caso de que el resultado de la operación AND del código de región de un par de vértices sea igual a cero, se comprueba la intersección del segmento formado por dichos vértices con los triángulos representados por uno de los códigos de región de esos dos vértices. Basta con realizar dicha comprobación con el conjunto de triángulos asociados al código de región con menor número de unos, que estará formado como máximo por dos triángulos. Si se produce intersección entre el segmento y uno de los triángulos concluimos que ambos tetraedros intersecan.

Si no se ha producido intersección, intercambiamos los papeles de T_1 y T_2 y realizamos de nuevo este test de intersección.

A continuación podemos ver el algoritmo de intersección entre tetraedros (Algoritmo 5.9). En este algoritmo consideramos que cuando un tetraedro está dentro de otro se produce intersección, pues este caso es necesario para el funcionamiento correcto del algoritmo de detección de colisión entre poliedros.

```
//Para comprobar la intersección entre T1 y T2 debe llamarse al algoritmo con T1,T2 y
posteriormente con T2,T1.

bool tetraedro::interseccion(tetrahedro T2=V0V1V2V3) {
   T1 = this
   bits bit[4][4]

   bit0,i = { sign(T1.α(V0))<0, sign(T1.α(V1))<0, sign(T1.α(V2))<0, sign(T1.α(V3))<0 }
   if (bit0,i == 1111) return false
   bit1,i = { sign(T1.β(V0))<0, sign(T1.β(V1))<0, sign(T1.β(V2))<0, sign(T1.β(VE))<0 }
   if (bit1,i == 1111) return false
   bit2,i = { sign(T1.γ(V0))<0, sign(T1.γ(V1))<0, sign(T1.γ(V2))<0, sign(T1.γ(V3))<0 }
   if (bit2,i == 1111) return false
   bit3,i = { sign(T1.δ(V0))<0, sign(T1.δ(V1))<0, sign(T1.δ(V2))<0, sign(T1.δ(V3))<0 }
   if (bit3,i == 1111) return false

   if (biti,0 == 0000 || biti,1 == 0000 || biti,2 == 0000 || biti,3 == 0000) return true

   if (this.interseccion(biti,0, biti,1, V0V1)==true) return true
   if (this.interseccion(biti,0, biti,2, V0V2)==true) return true
   if (this.interseccion(biti,0, biti,3, V0V3)==true) return true
   if (this.interseccion(biti,1, biti,2, V1V2)==true) return true
   if (this.interseccion(biti,1, biti,3, V1V3)==true) return true
   if (this.interseccion(biti,2, biti,3, V2V3)==true) return true

   return false
}
```

Algoritmo 5.9: Intersección entre tetraedros.

5.5.5. Intersección segmento/poliedro

Para comprobar si un segmento interseca con un poliedro podemos iterar el algoritmo de *intersección segmento/cara compleja* (Algoritmo 5.6) visto en la sección 5.5.2, utilizando todas las caras del poliedro y dicho segmento. El algoritmo de DC entre poliedros realizará un test similar pero sólo con las partes del poliedro incluidas en un determinado tetra-cono.

5.5.6. Intersección poliedro/poliedro

Podemos comprobar si se produce intersección entre poliedros aplicando un algoritmo de *intersección segmento/poliedro* entre todos los segmentos del primer poliedro con el segundo, y en caso de que no se produzca intersección, intercambiando los papeles de los dos poliedros para aplicar de nuevo esta comprobación.

Este algoritmo puede mejorarse cuando deseamos obtener la intersección entre dos poliedros en movimiento. Para esto utilizaremos tanto la coherencia temporal y geométrica, como la descomposición espacial obtenida por los tetra-trees y los volúmenes envolventes asociados, como veremos en lo que resta de este capítulo.

5.5.7. Detección de Colisión Poliedro/Poliedro

A continuación, y basándonos en los algoritmos anteriores, vamos a describir el método utilizado para la detección de colisión entre poliedros. Pero antes vamos a mostrar una propiedad gracias a la cual utilizaremos tetraedros envolventes para descartar partes del poliedro en la detección de colisión.

Propiedad 5.25: Dados dos poliedros F y G, para que se produzca intersección entre F y G debe producirse intersección entre un par de tetraedros envolventes $T_{env,F} \in F$ y $T_{env,G} \in G$ ó debe producirse la inclusión de uno de dichos tetraedros envolventes en el otro.

Esto es debido a que si se produce intersección entre dos poliedros*, debe haber intersección entre una arista y una cara, cada una de ellas de un poliedro. Cada uno de estos elementos estará incluido en un tetraedro envolvente, por tanto debe producirse intersección entre los tetraedros envolventes de ambos elementos (Figura 5.34).

Básicamente, el algoritmo de detección de colisión entre poliedros realiza una búsqueda entre las dos jerarquías de volúmenes envolventes asociadas a sus respectivos tetra-trees. Dados dos nodos de sendos tetra-trees, sólo si se produce intersección entre sus volúmenes envolventes, primero entre las esferas envolventes (Figura 5.35) y después entre los tetraedros envolventes, se desciende recursivamente a los nodos hijos, en primer lugar del volumen mayor, hasta encontrar dos nodos hoja de los respectivos árboles, en cuyo caso se realiza un test detallado de colisión entre los segmentos clasificados en un tetraedro envolvente y los triángulos que forman las caras clasificadas en el otro tetraedro envolvente y viceversa.

Gracias a la siguiente heurística podemos aprovechar la coherencia temporal cuando se produce una colisión.

Heurística 5.2: Dados dos poliedros F y G, si en un determinado frame se ha producido intersección entre un segmento de F y una cara de G, lo más probable es que en el siguiente frame se produzca intersección entre ese mismo segmento y esa cara. Si no ocurre dicha intersección, la probabilidad de que se produzca intersección entre un segmento y una cara de los mismos tetra-conos donde se encontraban en el frame anterior es también alta.

---

* En este caso particular no estamos considerando que uno de los poliedros esté incluido en el otro. Para tenerlo en cuenta podemos comprobar la inclusión de un vértice de cada uno de los poliedros en el otro.

Para hacer uso de esta heurística el algoritmo debe almacenar si se ha producido o no colisión en el frame anterior. Si se ha producido colisión debe guardar además entre qué arista y cara se ha producido, así como los tetra-conos involucrados en dicha colisión. En cada llamada al algoritmo debe comprobar inicialmente si se ha producido intersección entre la arista y la cara almacenadas en el frame anterior. Si no hay intersección entre estos elementos debe comprobarse si la hay entre las caras y aristas de los tetra-conos almacenados.

Debido a la coherencia, cuando tiene lugar una colisión, la probabilidad de que ocurra entre el mismo segmento y la misma cara es bastante alta, suponiendo cambios pequeños en la posición relativa de los objetos. Es más, si no tiene lugar la colisión de nuevo entre ese par de características, la siguiente colisión, si tiene lugar, es muy probable que ocurra entre las caras y aristas incluidas en los tetra-conos entre los que hubo colisión en el frame anterior. Si no es así, se procederá a buscar recursivamente entre los tetraedros envolventes de los tetra-trees asociados a cada uno de los poliedros.

El siguiente teorema nos permite descartar rápidamente tetraedros envolventes de ambos poliedros utilizando los tetraedros envolventes que forma parte de una envolvente regular.

Teorema 5.5: Sean F y G dos poliedros con tetra-trees asociados TT_F y TT_G. Sea $T_{env\text{-}reg}$ un tetraedro envolvente que forma parte de una envolvente regular $TT_{env\text{-}reg}$ asociada a un nivel de TT_F. Para todo tetraedro envolvente $T_{env}=V_0V_1V_2V_3$ perteneciente a TT_G cuyos vértices V_i, $i=0..3$, cumplan que $sign(\alpha(V_i))<0$, respecto a $T_{env\text{-}reg}$, se cumple que dicho tetraedro envolvente T_{env} no interseca con ningún tetraedro envolvente de TT_F.

Demostración: Debido a que $T_{env\text{-}reg}$ forma parte de una envolvente regular y convexa del poliedro F, todos los puntos del espacio que posean una coordenada α negativa respecto a $T_{env\text{-}reg}$ estarán a un lado del plano soporte del triángulo no original de $T_{env\text{-}reg}$, concretamente en el lado opuesto al que se encuentra el poliedro F, pues todos los vértices de F tienen coordenada α positiva respecto a $T_{env\text{-}reg}$. Si los vértices de un tetraedro envolvente T_{env} perteneciente al poliedro G cumplen esta condición, dicho tetraedro estará situado en el lado opuesto respecto al lado en el que se encuentra F, y por tanto, no se producirá intersección entre ningún tetraedro envolvente de TT_F y T_{env}. □

Figura 5.34: Tetraedros envolventes involucrados en el test de colisión entre objetos.

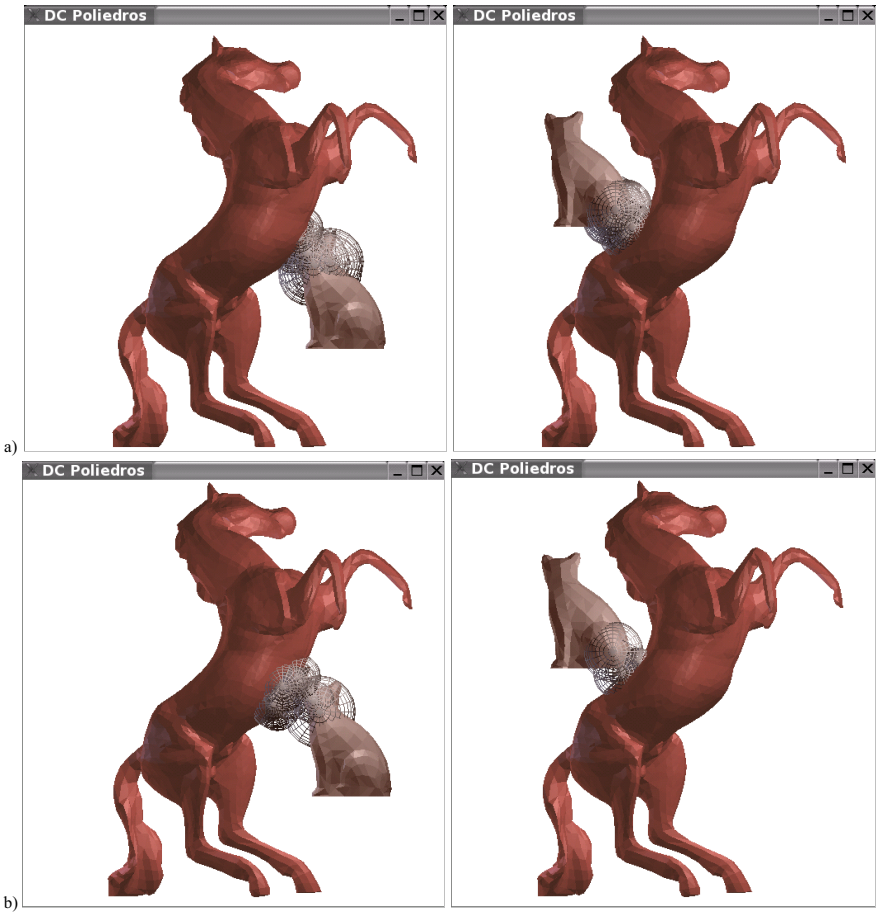


Figura 5.35: Esferas envolventes involucradas en la detección de colisión. a) Esferas envolventes que colisionan. b) Esferas envolventes asociadas a los tetraedros envolventes que colisionan.

Definición 5.21: Dados dos poliedros F y F' con envolventes regulares $TT_{env\text{-}reg}$ y $TT'_{env\text{-}reg}$ respecto del último nivel de TT_F y $TT_{F'}$ respectivamente, y cuyos centroides son *C* y *C'*; diremos que $T_{env\text{-}reg} \in TT_{env\text{-}reg}$ y $T'_{env\text{-}reg} \in TT'_{env\text{-}reg}$ son *triángulos envolventes de referencia* si [*C in* $\angle T'_{env\text{-}reg}$] y [*C' in* $\angle T_{env\text{-}reg}$].

Es posible acelerar los cálculos necesarios para la detección de colisión entre poliedros descartando algunos tetraedros envolventes de forma rápida, sobre todo cuando los objetos no están muy próximos. Para ello aplicamos el Teorema 5.5 a los tetraedros envolventes de referencia de los poliedros (Figura 5.36) y a cada uno de los tetraedros envolventes asociados a los tetra-trees de los poliedros. De este modo obtenemos un conjunto de tetraedros envolventes sobre los que comprobar la intersección (Figura 5.37), descartando al resto.

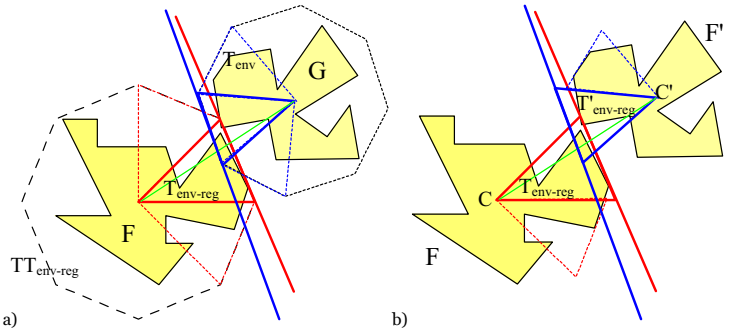


Figura 5.36: Simplificación 2D de los tetraedros envolventes que cumplen el Teorema 5.5 (en color rojo y azul discontinuo) y tetraedros envolventes de referencia (en color rojo y azul continuo). a) Se muestran los tetraedros envolventes que forman parte de una envolvente regular. b) Se muestran los tetraedros envolventes una vez iterado el algoritmo de ajuste.

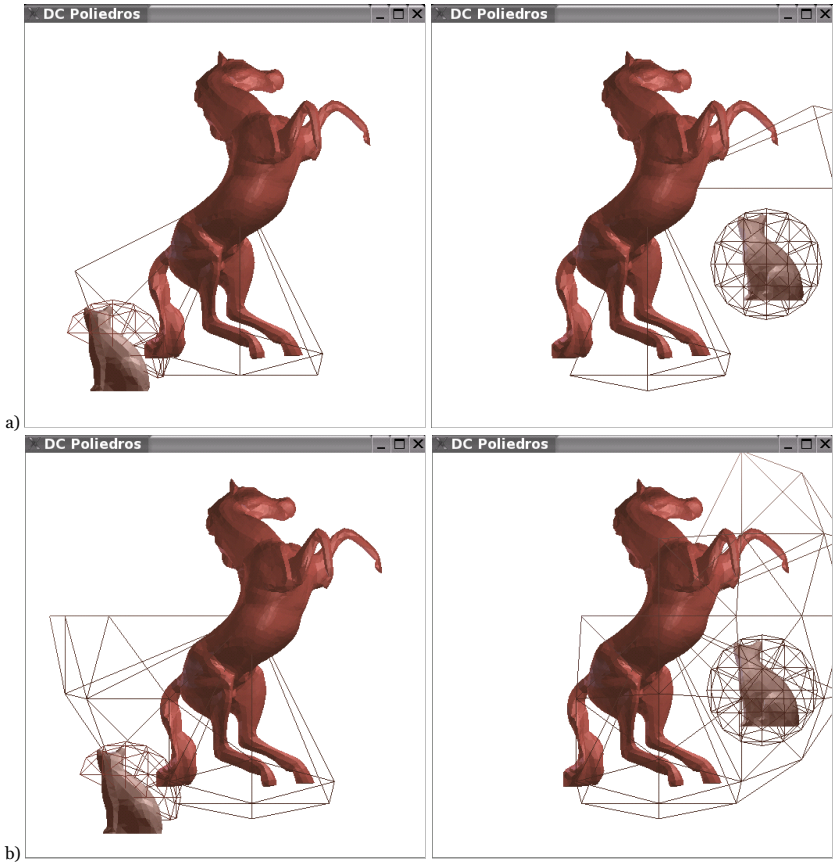


Figura 5.37: Diferencia entre utilizar $T_{env\text{-}reg}$ y T_{env} para la DC. Mostramos los tetraedros envolventes que forman parte de la envolvente regular de los poliedros como representación de los tetra-conos involucrados en la DC. a) Se aplica el Teorema 5.5 a los tetraedros envolventes de los poliedros. b) Se aplica dicho Teorema a los tetraedros que forman parte de la envolvente regular de los poliedros.

En el algoritmo desarrollado se ha tenido en cuenta que es posible que la intersección entre un segmento y una cara se produzca fuera del ámbito del tetra-cono en el que se encuentra la cara, pues esta intersección puede darse en alguno de los triángulos no originales de los tetraedros parcialmente contenidos o incluso de un tetraedro no contenido en el tetra-cono (Figura 5.38). Por este motivo es necesario comprobar si el punto de intersección se encuentra dentro del tetra-cono considerado, para en caso de que no sea así, localizar el tetra-cono en el que se encuentra dicha intersección y poder comprobar la inclusión de dicho punto de intersección en los triángulos del nuevo tetra-cono.

A continuación mostramos el algoritmo recursivo que utiliza los conceptos anteriores para la detección de colisión entre poliedros complejos (Algoritmo 5.10).

```
static segmentoAnterior, caraAnterior, tetraConoAnterior1, tetraConoAnterior2

static colisionAnterior

bool poliedro::testDeteccionColisionRecursivo(poliedro P) {
   // Intersección entre esferas envolventes
   if (this.esferaEnvolvente().intersección(P.esferaEnvolvente())==false) return false

   // Comprobación de intersección entre segmento-cara anterior o tetra-conos anteriores
   if (colisionAnterior == true) {
      if (caraAnterior.interseccion(segmentoAnterior) == true) return true
      if ((tetraConoAnterior1.testDeteccionColisionDetallado(tetraConoAnterior2) == true) ||
          (tetraConoAnterior2.testDeteccionColisionDetallado(tetraConoAnterior1) == true))
               return true
   }

   // Obtener los tetraedros de referencia
   tetraedroReferencia1 = this.obtenerTetraedroEnvolventeReferencia(P.centroide())
   tetraedroReferencia2 = P.obtenerTetraedroEnvolventeReferencia(this.centroide())

   // Marcar los tetraedros envolventes
   P.marcarTetraedrosEnvolventes(tetraedroReferencia1)
   this.marcarTetraedrosEnvolventes(tetraedroReferencia2)

   // Detección de colisión recursiva entre tetra-conos marcados
   for (tetraCono1 = 0; tetraCono1 < 8; tetraCono1++)
      if (this.hijo(tetraCono1).marcado()==true)
         for (tetraCono2 = 0; tetraCono2 < 8; tetraCono2++)
            if (P.hijo(tetraCono2).marcado()==true)
               if (this.hijo(tetraCono1).testDeteccionColision(P.hijo(tetraCono2))==true)
                  return true
   colisionAnterior = false
   return false
}

bool tetraCono::testDeteccionColision(tetraCono tc) {
   // Intersección entre esferas envolventes
   if (this.esferaEnvolvente().intersección(tc.esferaEnvolvente())==false) return false
   // Intersección entre tetraedros envolventes
   if (this.tetraedroEnvolvente().interseccion(tc.tetraedroEnvolvente())==false) &&
      (tc.tetraedroEnvolvente().interseccion(this.tetraedroEnvolvente())==false) return false
```

```
   // Descender por los subárboles
   if (this.hoja()==true && tc.hoja()==true) {
      paresVolumenes.insertar(this,tc)
      if (this.testDeteccionColisionDetallado(tc)==true ||
            tc.testDeteccionColisionDetallado(this)==true) return true
   } else if (this.hoja()==true) {
      for (tetraCono = 0; tetraCono < 4; tetraCono++)
         if (tc.hijo(tetraCono).marcado()==true)
            if (tc.hijo(tetraCono).testDeteccionColision(this)==true) return true
   } else if (tc.hoja()==true) {
      for (tetraCono = 0; tetraCono < 4; tetraCono++)
         if (this.hijo(tetraCono).marcado()==true)
            if (this.hijo(tetraCono).testDeteccionColision(tc)==true) return true
   } else {
      for (tetraCono1 = 0; tetraCono1 < 4; tetraCono1++)
         if (this.hijo(tetraCono1).marcado()==true)
            for (tetraCono2 = 0; tetraCono2 < 4; tetraCono2++)
               if (tc.hijo(tetraCono2).marcado()==true)
                  if (this.hijo(tetraCono1).testDeteccionColision(tc.hijo(tetraCono2))==true)
                       return true
   }
   return false
}

bool tetraCono::testDeteccionColisionDetallado(tetraCono tc) {
   for (seg = 0; seg < tc.numeroSegmentos(); seg++) {
      Q1Q2 = tc.segmento(seg)
      TCInterseccion = this->poliedro()->tetraConoInterseccion(Q1Q2)
      for (c = 0; c < TCInterseccion.numeroCaras(); c++) {
         if (TCInterseccion.cara(c).interseccion(Q1Q2)==true) {
            segmentoAnterior = Q1Q2
            caraAnterior = TCInterseccion.cara(c)
            tetraConoAnterior1 = this
            tetraConoAnterior2 = TCInterseccion
            colisionAnterior = true
            return true
         }
      }
   }
   return false
}

void poliedro::marcarTetraedrosEnvolventes(tetraedro T) {
   for (vert = 1; vert < 3; vert++)
      if (T.α(this.TetraedroEnvolvente().vertice(vert))>=0) {
         this.marcado = true
         for (tetraCono = 0; tetraCono < 4; tetraCono++) {
            this.hijo(tetraCono).marcarTetraedrosEnvolventes(T)
         }
         break
      }
}
```

Algoritmo 5.10: Detección de Colisión Poliedro/Poliedro recursiva.

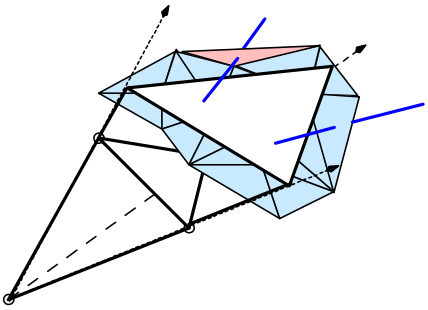


Figura 5.38: El punto de intersección entre el segmento de un poliedro y la cara de otro poliedro puede encontrarse en otro tetra-cono distinto del considerado, pues esta intersección puede darse en el triángulo no original de un tetraedro parcialmente contenido (en azul) o incluso no contenido (en rojo) en el tetra-cono.

En la implementación realizada del Algoritmo 5.10 se han efectuado diversas optimizaciones en el ámbito de los tetra-conos. Como es necesario realizar un test de *intersección segmento/triángulo* entre todos los pares de aristas y triángulos no originales de los tetraedros del recubrimiento clasificados en un par de tetra-conos, podemos pre-calcular cierta información que es común para muchos de estos cálculos, y luego reutilizarla sin tener que calcularla de nuevo cada vez que se necesita. Entre estos cálculos compartidos se encuentran los siguientes:

- Para cada triángulo se puede almacenar el valor de los vectores: $B = V_1 - V_3$ y $C = V_2 - V_3$. Dado un triángulo, este cálculo es común para todas las aristas con las que se comprueba la intersección arista/triángulo.

- Para cada par vértice-del-triángulo/extremo-de-la-arista (V_i, Q_j), j=*1..2*, de cada uno de los triángulos no originales de los tetraedros clasificados en un tetra-cono, se puede almacenar el valor de los vectores: $A = Q_j - V_i$, y precalcular $w_{i,j} = A \cdot (B \times C)$. A la hora de utilizar estos cálculos necesitaremos para cada par triángulo/arista el valor de $A = Q_1 - V_n$, siendo n el vértice que corresponda al vértice V_3 del triángulo en cuestión.

- Con los cálculos anteriores obtenemos también los valores de $D = Q_2 - V_n$ y de $s = D \cdot (B \times C)$, pues éstos se corresponden con los valores de A y w almacenados para el par (V_i, Q_2).

El algoritmo descrito en esta sección es válido para cualquier pareja de poliedros sobre los que realizar la detección de colisión. Sin embargo, podemos utilizar el algoritmo de *detección de colisión esfera/poliedro* (Algoritmo 5.4) y aplicarlo a la detección de colisión entre poliedros. Este algoritmo es más eficiente cuando la relación entre el tamaño de las esferas envolventes de los poliedros es grande, es decir, hay una diferencia significativa en el tamaño de los poliedros. En esta situación el algoritmo es muy apropiado, pues su tiempo de ejecución es similar al del algoritmo de detección de colisión punto/poliedro (Algoritmo 5.2).

Al igual que el algoritmo en el que se basa, en el nuevo algoritmo se debe utilizar un punto que se encuentre dentro del poliedro para que la detección de colisión punto/poliedro incluida en el mismo funcione correctamente. Si el centro de la esfera envolvente no pertenece al poliedro puede usarse cualquier otro punto, como por ejemplo un vértice.

A continuación podemos ver este nuevo algoritmo para la detección de colisión entre poliedros (Algoritmo 5.11).

5.6. Optimizaciones para mallas de triángulos

Si en lugar de utilizar objetos modelados mediante caras complejas se utilizan mallas de triángulos, los fundamentos y algoritmos desarrollados siguen siendo válidos. Sin embargo es posible realizar algunas optimizaciones:

- En primer lugar, no es necesario realizar un recubrimiento de las caras. Podemos utilizar los propios triángulos sin la necesidad de dividirlos más.

- Para todos los casos, se consideran elementos frontera el interior del triángulo $V_1V_2V_3$, las aristas V_1V_2, V_3V_1 y V_2V_3, así como los vértices V_1, V_2 y V_3, es decir, el triángulo $V_1V_2V_3$ completo.

- En el caso de la *inclusión y detección de colisión punto/poliedro* se obtiene la inclusión directa del punto en el tetraedro si se encuentra sobre algún elemento frontera.

- Para la *intersección y detección de colisión esfera/poliedro* se obtiene la intersección directa de la esfera y el tetraedro cuando se produce la intersección con las aristas frontera o la inclusión de los vértices frontera en la esfera. A estos efectos también se obtiene la intersección directa cuando el punto más cercano Q sobre el plano soporte del triángulo a la esfera se encuentra sobre el triángulo $V_1V_2V_3$.

- Para el caso de *intersección y detección de colisión entre poliedros* no es necesario un algoritmo de intersección segmento/cara, pues basta con utilizar un algoritmo segmento/triángulo.

```
static resultadoAnterior = false

bool poliedro::testDeteccionColision(poliedro P) {
   E = P.esferaEnvolvente()
   C = E.centro()
   R = E.radio()
   // Interseccion entre esferas envolventes
   if (E.interseccion(this.esferaEnvolvente())==false) return false

   // Detección de colisión con el centro de la esfera
   resultado = this.testDeteccionColision(C)
   if (resultado==EQUAL_STATE && resultadoAnterior==true) return true
   resultadoAnterior = resultado
   if (resultado==true) return true
   // Obtener la lista de Tetra-Conos en los que está la esfera
   LTC = this.obtenerListaTetraConosExtendidosInclusion(C,R)
   // Eliminar los Tetra-Conos en cuyo Tetraedro envolvente no se encuentre la esfera
   for (Tc=0;Tc<LTC.end();Tc++)
      if (LTC(Tc).tetraedroEnvolvente.extension(R).inclusion(C)==false)
         LTC(Tc).borrar()
   // Obtener la lista de caras incluidas en los tetraconos.
   // Cada cara tiene una lista de tetraedros incluidos en el conjunto de tetraconos LTC
   LF = this.obtenerListaCarasConSusTetraedros(LTC)
   // Cálculo de la intersección esfera/cara con las caras en cuya AIE se encuentra C
   for (F=0;F<LF.numeroCaras();F++)
      if (sign(LF.cara(F).tetraedroRepresentativo().AIE(C,R)) >= 0)
         if (E.interseccion(LF.cara(F)) == true)
            if (P.intersección(LF.cara(F)) == true)
               return true
   return false
}

bool poliedro::intersección(cara F) {
   // Todos los segmentos del poliedro con la cara F
   for (seg = 0; seg < this.numeroSegmentos(); seg++) {
      Q1Q2 = this.segmento(seg)
      if (F.interseccion(Q1Q2)==true)
         return true
   }
   // Todos los segmentos de la cara F con todas las caras del poliedro
   for (seg = 0; seg < F.numeroSegmentos(); seg++) {
      Q1Q2 = F.segmento(seg)
      for (c = 0; c < this.numeroCaras(); c++) {
         if (this.cara(c).interseccion(Q1Q2)==true)
            return true
         }
   }
   return false
}
```

Algoritmo 5.11: Detección de colisión Poliedro/Poliedro no recursiva.

5.7. Algunas Consideraciones

Los algoritmos desarrollados pueden obtener las características implicadas en la colisión. Por ejemplo, para la detección de colisión entre poliedros, cuando se produce intersección entre una arista y una cara, en el caso de querer conocer todos los elementos implicados, el algoritmo no debe parar y retornar que se ha producido colisión, sino continuar y almacenar los pares de características implicadas. De igual forma, el algoritmo no debe parar cuando se da intersección entre los elementos de un único par de tetraedros envolventes, sino localizar todos los pares de tetraedros envolventes que intersecan. Evidentemente, estos algoritmos son más costosos que los aquí expuestos.

También es posible obtener la detección de colisión entre objetos con una determinada *tolerancia*, por ejemplo obtener colisión cuando un objeto se encuentra a una distancia menor que una distancia dada. Para ello simplemente hay que utilizar los offsets de los tetraedros para determinar el lado en el que se encuentra un determinado elemento respecto a un tetraedro del recubrimiento. Igualmente será necesario utilizar offsets para los tetraedros envolventes y para los tetra-conos.

En cuanto a la *eficiencia* de los algoritmos, según veremos en el estudio temporal realizado en la siguiente sección, podemos considerar que son más eficientes que otros, en primer lugar al utilizar una descomposición espacial que ocupa menos memoria que otras y que es bastante rápida de calcular, pues se ajusta y adapta mejor a los objetos que otras. En segundo lugar, gracias al aprovechamiento de la coherencia temporal y espacial, pues se utiliza la coherencia que pueda haber en sistemas interactivos.

Si consideramos los aspectos de *robustez* de estos algoritmos podemos decir que presentan un alto grado de robustez geométrica, pues son algoritmos válidos para muchos tipos de objetos que en otros métodos pueden presentar problemas, como por ejemplo para objetos cóncavos, no-variedad, con agujeros, etc. Además tienen un alto grado de robustez numérica en el sentido de que la mayor parte de las operaciones se realizan mediante sumas signadas que utilizan aritmética entera. Cuando es necesario obtener exclusivamente el signo de una coordenada baricéntrica, éste no se calcula realizando la división que lleva implícita sino realizando una sencilla operación entre signos. El cálculo del signo se encuentra limitado exclusivamente por la precisión en la evaluación de los determinantes [ABDPY97] siendo calculado con una determinada tolerancia [GSS89], de manera que se considera cero cuando está en un intervalo $[+\varepsilon,-\varepsilon]$. Utilizar recubrimientos simpliciales y coordenadas baricéntricas, simplifica en gran medida los casos especiales, que no obstante, existen y se han considerado adecuadamente en los algoritmos.

5.8. Estudio de Tiempos

A lo largo de esta sección mostraremos la eficacia de los algoritmos desarrollados para la detección de colisión de puntos, esferas y poliedros en relación a poliedros. En primer lugar mediremos el tiempo de construcción de un tetra-tree para distintos tipos de poliedros y lo compararemos con el tiempo de construcción de un octree. Se ha utilizado este tipo de estructura de datos pues es una de las estructuras adaptativas más utilizadas y que además permite acelerar los cálculos de inclusión de puntos en poliedros utilizando el algoritmo de crossing-count, según se deduce de [Hai94] y [SE03].

Posteriormente mediremos el tiempo de detección de colisión para cada uno de los algoritmos descritos en este capítulo (punto/poliedro, esfera/poliedro y poliedro/poliedro en la versión recursiva y en la versión basada en el algoritmo esfera/poliedro). Utilizaremos poliedros de distinto tamaño, algunos de ellos mallas de triángulos. Usaremos como medida el número de frames por segundo, es decir, el número de veces que se ejecuta el algoritmo de detección de colisión por unidad de tiempo.

Cada prueba la realizaremos para dos tipos de trayectorias, una circular y una de contorno. Se variarán determinados parámetros para ajustar en lo posible el tetra-tree a los poliedros y ajustar dichos parámetros al tipo de algoritmo utilizado, para tratar de obtener mejores tiempos en la detección de colisión. Veremos la influencia de cada uno de estos parámetros tanto en la construcción del tetra-tree como en la detección de colisión entre objetos.

Al igual que en el caso 2D se ha utilizado un procesador Intel Pentium IV a 1,6 GHz con 1 MB de RAM para llevar a cabo las pruebas. Los algoritmos se han implementado en C++ con OpenGL en el sistema operativo Linux, siguiendo una filosofía orientada o objetos.

5.8.1. Pruebas y Tiempos de Pre-procesamiento

Aunque para la detección de colisión el tiempo necesario para construir las estructuras de datos, en nuestro caso los tetra-trees, no es significativo, consideramos interesante compararlo con otros métodos como la construcción de un octree, pues este nuevo tipo de estructura de datos creemos que puede utilizarse eficientemente para otros tipos de aplicaciones como por ejemplo el modelado de objetos. El tiempo de construcción no afecta a la DC pues un tetra-tree es invariante cuando se producen transformaciones rígidas sobre los poliedros y, una vez construidos estos tetra-trees, no existe penalización debido a su posible actualización

durante la detección de colisión. Para el caso de una detección de colisión entre objetos deformables, aunque el tema no es objeto de esta memoria, si que sería necesaria una actualización.

Mediremos el tiempo necesario para la construcción de un tetra-tree en el caso de poliedros formados por distinto número de vértices. Los parámetros necesarios para la construcción de un tetra-tree son tres: la profundidad en el árbol (o número de subdivisiones espaciales realizadas), el mínimo número de tetraedros del recubrimiento clasificados en un tetra-cono para no realizar la subdivisión del tetra-cono, y el número de iteraciones del algoritmo de ajuste del tetraedro envolvente de cada tetra-cono.

Variaremos estos parámetros para obtener el tiempo de construcción de un tetra-tree. Para compararlo con un octree utilizaremos el número de subdivisiones espaciales realizadas, en lugar de la profundidad en el árbol, pues un octree y un tetra-tree tienen un número de nodos distinto en cada nivel. El criterio de parada para la subdivisión (mínimo número de tetraedros, o de triángulos/caras en el caso de un octree) será el mismo para tetra-trees y octrees. El número de iteraciones del algoritmo de ajuste sólo afecta al tetra-tree.

Los poliedros utilizados son de dos tipos, poliedros complejos con caras planas no convexas formadas por más de tres vértices y algunas con agujeros en su interior, y poliedros formados por mallas de triángulos. Se han utilizado dos poliedros del primer tipo y dos del segundo; los primeros con menor número de vértices que los segundos. Los poliedros considerados son los expuestos en la Figura 5.39. En la Figura 5.40 podemos ver la descomposición realizada mediante octrees y tetra-trees a diversos de estos poliedros.

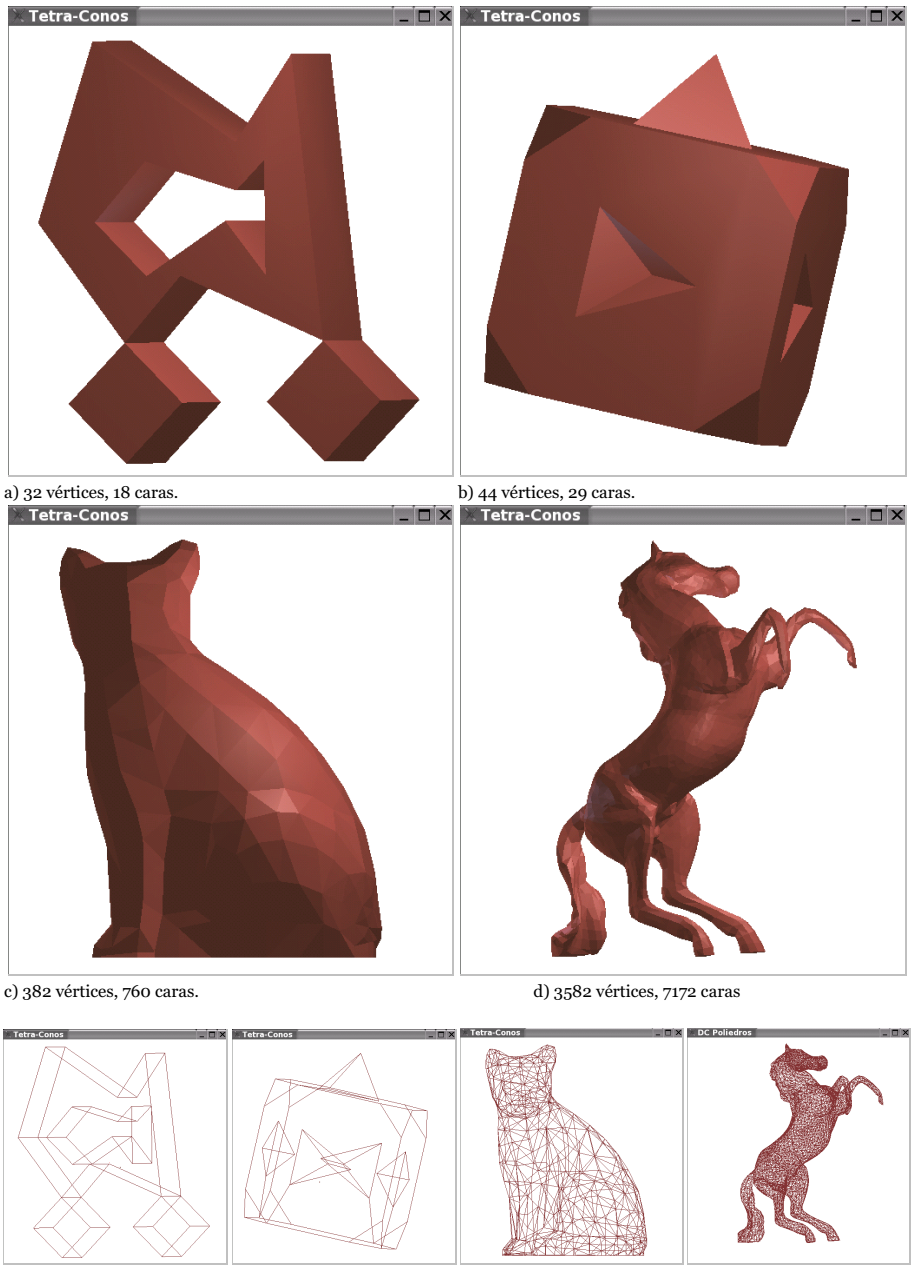
En la Figura 5.41 y 5.42 mostramos el tiempo de construcción de un tetra-tree y un octree en función del número de subdivisiones espaciales para los objetos considerados. La Figura 5.41 muestra los tiempos obtenidos en la construcción del árbol correspondiente hasta la profundidad máxima. La Figura 5.42 muestra los tiempos obtenidos en la construcción del árbol, en el que no se desciende si el número de tetraedros clasificados en un tetra-cono en el caso de un tetra-tree, o el número de caras clasificadas en una celda para un octree, es menor o igual a cinco. En la Figura 5.43 mostramos los tiempos de la Figura 5.41 para el *Poliedro Complejo II* y el *Caballo*. En dicha figura se muestran juntos los tiempos de construcción del tetra-tree y del octree. Debemos tener en cuenta que hay subdivisiones que no pertenecen al tetra-tree o al octree, por lo que en las tablas no aparece dicha información.

A la vista de estos datos, podemos concluir que la construcción de un tetra-tree es más rápida que la construcción de un octree tanto para poliedros formados por muchas caras como para el caso de existir un gran número de subdivisiones espaciales. La pendiente de la curva obtenida para la construcción de un tetra-tree es menor que la obtenida para la construcción de un octree.

Sin embargo, para poliedros con pocas caras o poliedros en los que se realizan pocas subdivisiones espaciales, la construcción de un octree es más rápida. Esto es así debido sobre todo a que los poliedros considerados (*Poliedro Complejo I* y *II*) están formados por caras poligonales de más de tres vértices, por lo que es necesario realizar un recubrimiento de la cara y son más los tetraedros que hay que clasificar en el tetra-tree que caras a clasificar en el octree. Además observamos que los tiempos obtenidos para el *Poliedro Complejo II* son mejores que los obtenidos para el *Poliedro Complejo I*. Esto es debido a que, a pesar de que el *Poliedro Complejo II* tiene un número mayor de caras, el número de tetraedros del recubrimiento es menor que en el caso del *Poliedro Complejo I*, pues muchas de las caras de este objeto son triangulares y no ha sido necesario realizar un recubrimiento de las mismas.

Debemos tener en cuenta que aunque para estos casos concretos el tiempo de construcción de un tetra-tree es peor que el de un octree, en general los tiempos obtenidos en la detección de colisión son optimizados con el uso de tetra-trees. Para poliedros formados por muchas caras, los tiempos de construcción y de detección de colisión son mucho mejores utilizando tetra-trees que mediante el uso de octrees.

En la Figura 5.44 podemos apreciar el tiempo de construcción de un tetra-tree para el caso del poliedro denominado *Caballo* en función del número de subdivisiones espaciales realizadas y del número de iteraciones del algoritmo de ajuste. En la Figura 5.45 podemos ver el tiempo necesario para la construcción del tetra-tree de cada uno de los poliedros considerados anteriormente, para un numero de iteraciones del algoritmo de ajuste constante e igual a ocho.



a) 32 vértices, 18 caras.

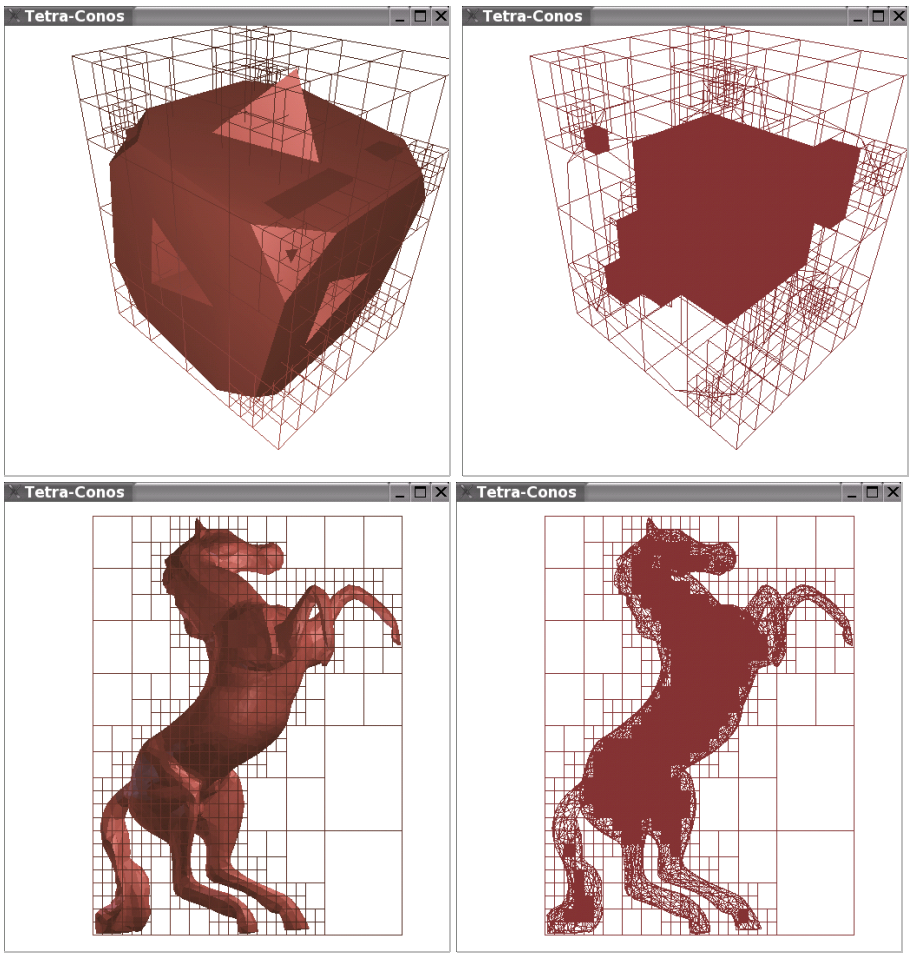
b) 44 vértices, 29 caras.

c) 382 vértices, 760 caras.

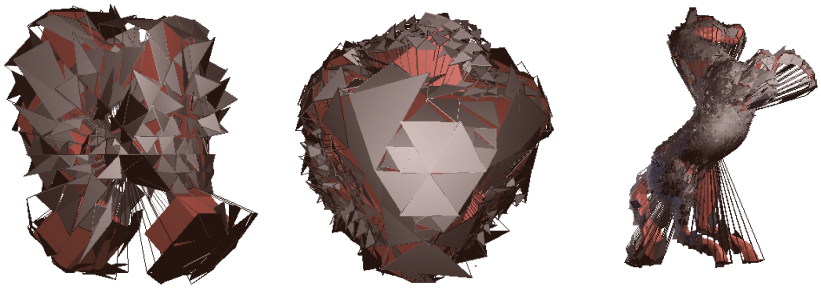
d) 3582 vértices, 7172 caras

e)

Figura 5.39: Poliedros utilizados en las pruebas. a) Poliedro complejo I. b) Poliedro complejo II. c) Malla de triángulos III (*gato*). d) Malla de triángulos IV (*caballo*). e) Modelos en modo de alambre.



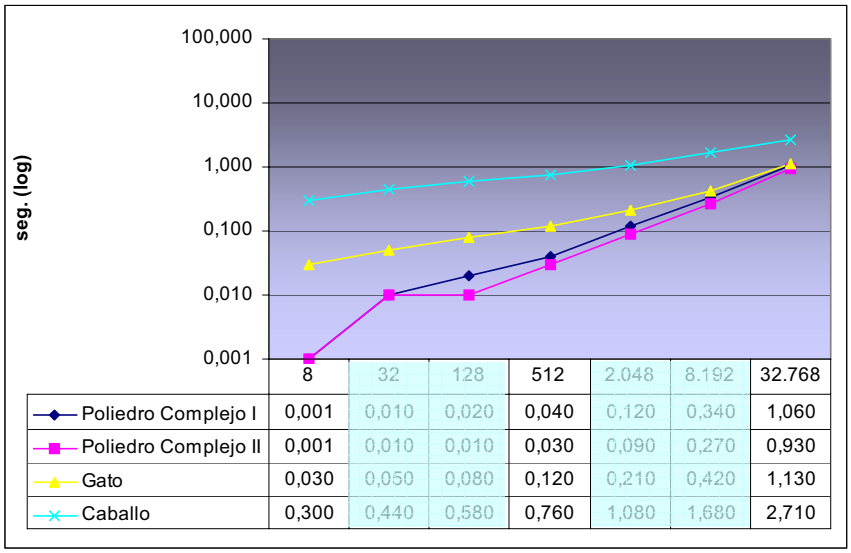
a)



b)

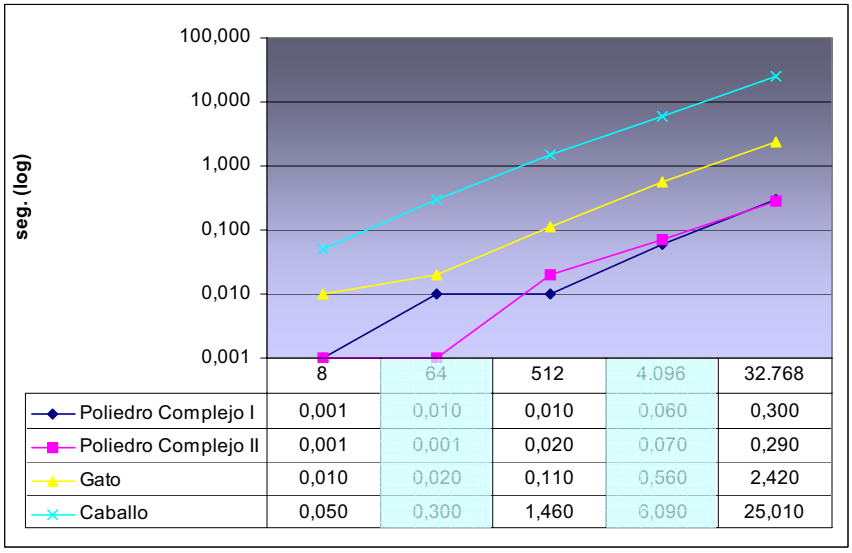
Figura 5.40: Descomposición de poliedros mediante: a) Octrees. b) Tetra-Trees.

a)



	8	32	128	512	2.048	8.192	32.768
Poliedro Complejo I	0,001	0,010	0,020	0,040	0,120	0,340	1,060
Poliedro Complejo II	0,001	0,010	0,010	0,030	0,090	0,270	0,930
Gato	0,030	0,050	0,080	0,120	0,210	0,420	1,130
Caballo	0,300	0,440	0,580	0,760	1,080	1,680	2,710

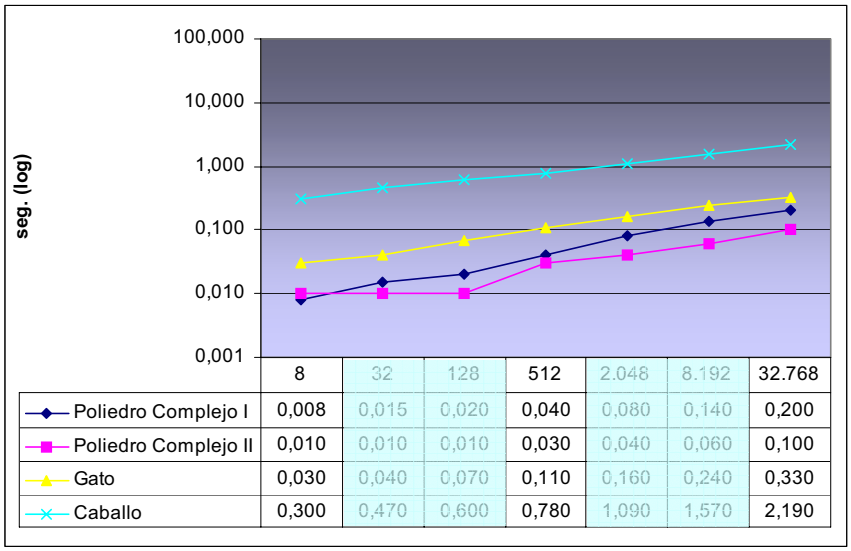
b)



	8	64	512	4.096	32.768
Poliedro Complejo I	0,001	0,010	0,010	0,060	0,300
Poliedro Complejo II	0,001	0,001	0,020	0,070	0,290
Gato	0,010	0,020	0,110	0,560	2,420
Caballo	0,050	0,300	1,460	6,090	25,010

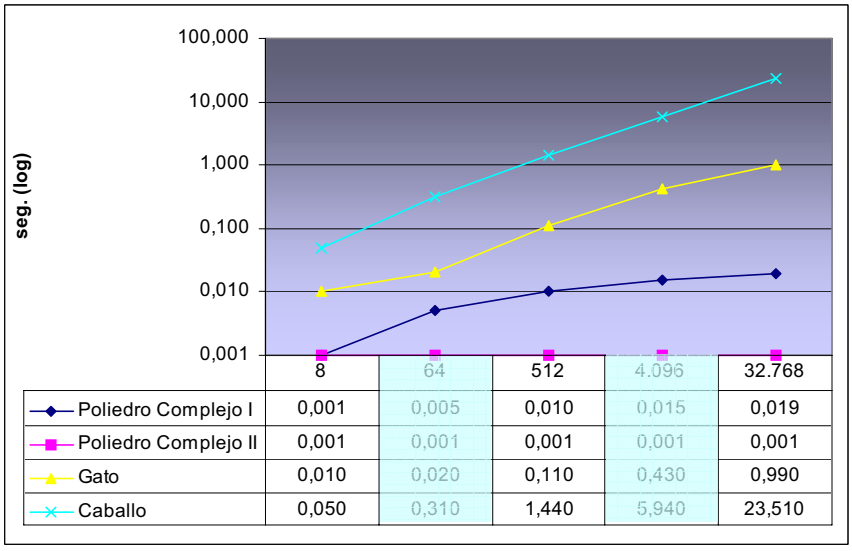
Figura 5.41: Tiempo en segundos con escala logarítmica para construir la descomposición espacial de los objetos considerados. En el eje X se muestra el número de subdivisiones espaciales. a) Construcción de un Tetra-Tree hasta su profundidad máxima, sin iterar el algoritmo de ajuste de tetraedros envolventes. b) Construcción de un Octree hasta su profundidad máxima.

a)



	8	32	128	512	2.048	8.192	32.768
Poliedro Complejo I	0,008	0,015	0,020	0,040	0,080	0,140	0,200
Poliedro Complejo II	0,010	0,010	0,010	0,030	0,040	0,060	0,100
Gato	0,030	0,040	0,070	0,110	0,160	0,240	0,330
Caballo	0,300	0,470	0,600	0,780	1,090	1,570	2,190

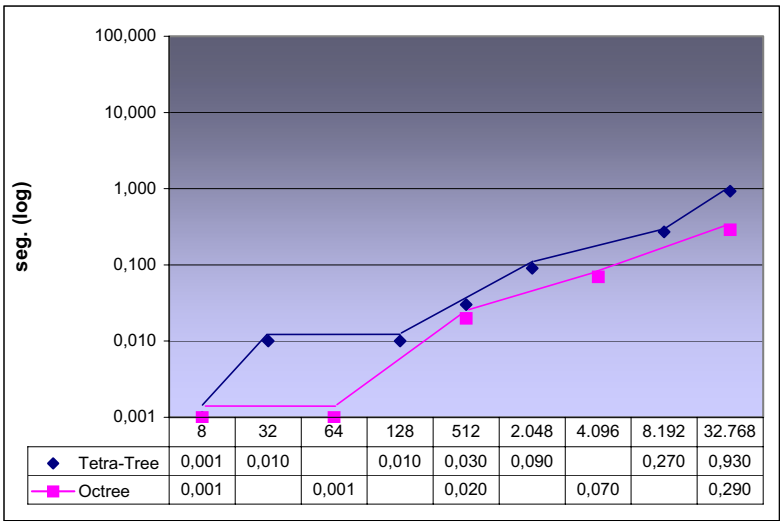
b)



	8	64	512	4.096	32.768
Poliedro Complejo I	0,001	0,005	0,010	0,015	0,019
Poliedro Complejo II	0,001	0,001	0,001	0,001	0,001
Gato	0,010	0,020	0,110	0,430	0,990
Caballo	0,050	0,310	1,440	5,940	23,510

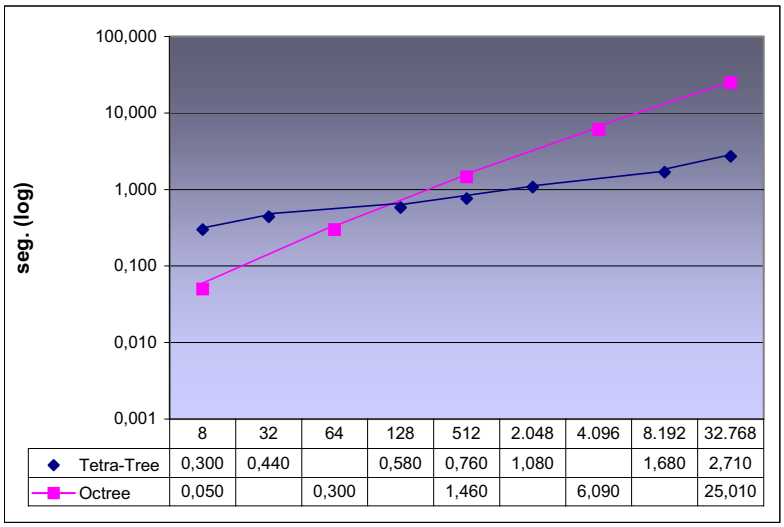
Figura 5.42: Tiempo en segundos con escala logarítmica para construir la descomposición espacial de los objetos considerados. En el eje X se muestra el número de subdivisiones espaciales. a) Construcción de un Tetra-Tree con mínimo número de tetraedros clasificados por tetra-cono igual a cinco, y sin iterar el algoritmo de ajuste de tetraedros envolventes. b) Construcción de un Octree con mínimo número de caras clasificadas en cada celda igual a cinco.

a)



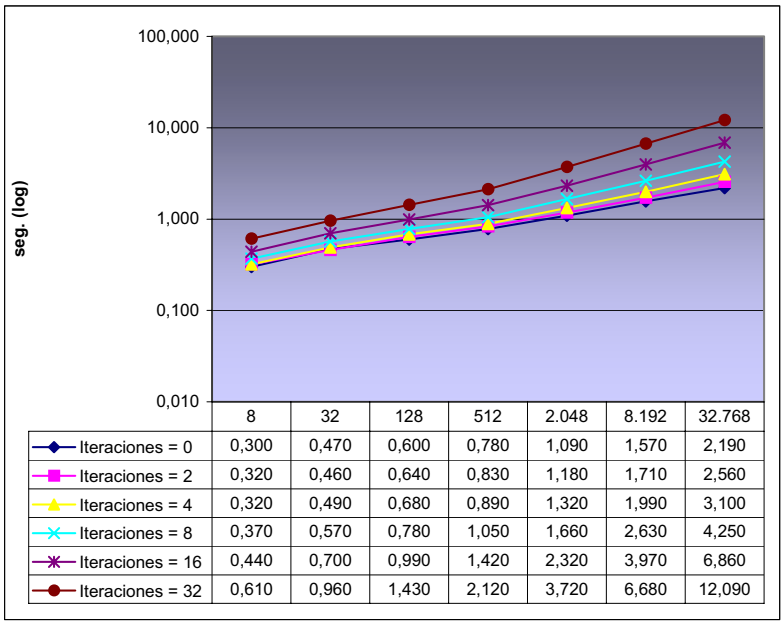
	8	32	64	128	512	2.048	4.096	8.192	32.768
Tetra-Tree	0,001	0,010		0,010	0,030	0,090		0,270	0,930
Octree	0,001		0,001		0,020		0,070		0,290

b)



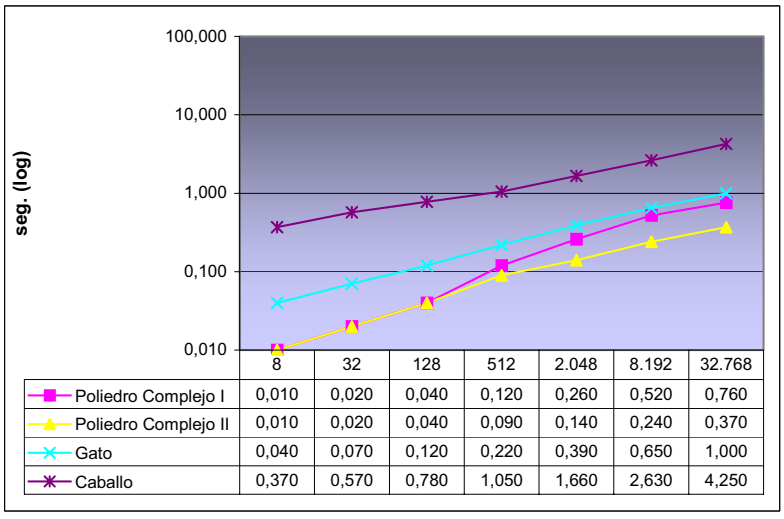
	8	32	64	128	512	2.048	4.096	8.192	32.768
Tetra-Tree	0,300	0,440		0,580	0,760	1,080		1,680	2,710
Octree	0,050		0,300		1,460		6,090		25,010

Figura 5.43: Tiempo en segundos con escala logarítmica para construir la descomposición espacial de los objetos considerados hasta la profundidad máxima del árbol, sin iterar el algoritmo de ajuste. En el eje X se muestra el número de subdivisiones espaciales. a) Para el *Poliedro Complejo II*. b) Para la malla de triángulos denominada *Caballo*.



	8	32	128	512	2.048	8.192	32.768
Iteraciones = 0	0,300	0,470	0,600	0,780	1,090	1,570	2,190
Iteraciones = 2	0,320	0,460	0,640	0,830	1,180	1,710	2,560
Iteraciones = 4	0,320	0,490	0,680	0,890	1,320	1,990	3,100
Iteraciones = 8	0,370	0,570	0,780	1,050	1,660	2,630	4,250
Iteraciones = 16	0,440	0,700	0,990	1,420	2,320	3,970	6,860
Iteraciones = 32	0,610	0,960	1,430	2,120	3,720	6,680	12,090

Figura 5.44: Tiempo en segundos con escala logarítmica para construir el tre-tree del poliedro *Caballo* en función del número de subdivisiones (eje X) y del número de iteraciones del algoritmo de ajuste.



	8	32	128	512	2.048	8.192	32.768
Poliedro Complejo I	0,010	0,020	0,040	0,120	0,260	0,520	0,760
Poliedro Complejo II	0,010	0,020	0,040	0,090	0,140	0,240	0,370
Gato	0,040	0,070	0,120	0,220	0,390	0,650	1,000
Caballo	0,370	0,570	0,780	1,050	1,660	2,630	4,250

Figura 5.45: Tiempo en segundos con escala logarítmica para construir un tetra-tree para los poliedros considerados en función del número de subdivisiones (eje X) y del tipo de poliedro. El algoritmo de ajuste se ha fijado a 8 iteraciones.

5.8.2. Pruebas para la Detección de Colisión punto/poliedro

Veamos a continuación las pruebas realizadas para la detección de colisión entre un punto y un poliedro. Los tiempos obtenidos por el algoritmo descrito en este capítulo (Algoritmo 5.2) que hemos llamado DC-TT, se compararán con los tiempos obtenidos por un algoritmo de detección de colisión similar que no utiliza Tetra-Trees, denominado DC, y con sendos algoritmos de inclusión mediante tetra-trees (Inc-TT) y octrees (Inc-Oct).

Para las pruebas se han utilizado los poliedros de la Figura 5.39 en los que se han construido los correspondientes tetra-trees para niveles en el intervalo [1-7], es decir con número de subdivisiones igual a 8, 32, 128, 512, 2.048, 8.192 y 32.768. Se han construido también octrees para los niveles 1, 3 y 5 pues en estos niveles coincide el número de subdivisiones con los niveles 1, 4 y 7 del tetra-tree respectivamente.

Se han generado dos tipos de trayectorias, una circular sobre un plano aleatorio que pasa por el centroide del poliedro, con centro en ese punto y un radio adaptado al tipo de poliedro de manera que se produzcan colisiones en un porcentaje menor del 5% y esté lo más próxima posible al poliedro. Esa trayectoria se ha modificado con una pequeña variación aleatoria respecto de la circunferencia original.

El segundo tipo de trayectoria es la que llamamos de contorno. Esta trayectoria se ha generado recorriendo el contorno del poliedro a una determinada distancia del mismo con una pequeña desviación aleatoria. En este movimiento no se producen colisiones entre el punto y el poliedro.

La densidad de puntos generados es mayor en el caso de la trayectoria de contorno que en el de la trayectoria circular. Además en la trayectoria de contorno, la mayor parte del tiempo encontramos que el punto está incluido en algún tetraedro envolvente del tetra-tree, pues está muy próximo al poliedro. En cambio en el caso de la trayectoria circular, esto ocurre en menos del 40% de los casos.

Se ha medido también la influencia del criterio de parada utilizado para no dividir un tetra-cono. Éste consiste en tener ya clasificados en ese tetra-cono un número mínimo de tetraedros. También se ha medido la influencia del número de iteraciones del algoritmo de ajuste, pero no se han encontrado diferencias significativas para ninguno de los poliedros considerados. Esto es debido a la naturaleza de las trayectorias generadas, pues en un caso el punto considerado está casi siempre en un tetraedro envolvente, independientemente del ajuste considerado, y en el otro, el porcentaje de veces que el punto está fuera de todos los tetraedros envolventes del poliedro es similar, independientemente del ajuste.

En las Tablas 5.2 y 5.3 puede verse un resumen de las pruebas realizadas y de los resultados obtenidos.

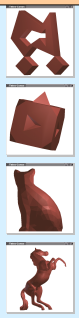
Datos de Entrada					
Tipo Poliedro	Subdivisiones	Iteraciones A.A.	Mín. Tetra. Clasif	Trayectoria	Algoritmos
	8 32 128 512 2.048 8.192 32.768	8	3	Circular Contorno	DC (Det.Col sin Tetra-Tree) DC-TT (Algoritmo 5.2) Inc-TT (Inclusión con Tetra-Tree) Inc-Oct (Inclusión con Octree)
Leyenda					
Eje X	Eje Y	Gráficos		Comportamiento analizado	
Nº Subdivisiones	Frames/seg	a) Trayectoria de contorno b) Trayectoria circular		DC en función del nº de subdivisiones, tipo de poliedro y trayectoria.	
Resultados (Resumen en Figura 5.50)					
Tipo Poliedro	Trayectoria	Pos. Algoritmo		Figura	
	Contorno	1º DC-TT 2º DC 3º Inc-Oct 4º Inc-TT		5.46.a	
	Circular	1º DC-TT 2º Inc-Oct 3º DC 4º Inc-TT		5.46.b	
	Contorno	1º DC-TT 2º DC 3º Inc-TT 4º Inc-Oct		5.47.a	
	Circular	1º DC-TT 2º Inc-TT 3º Inc-Oct 4º DC		5.47.b	
	Contorno	1º DC-TT 2º DC 3º Inc-Oct 4º Inc-TT		5.48.a	
	Circular	1º DC-TT 2º Inc-Oct 3º Inc-TT 4º DC		5.48.b	
	Contorno	1º DC-TT 2º Inc-TT 3º Inc-Oct 4º DC		5.49.a	
	Circular	1º DC-TT 2º Inc-Oct 3º Inc-TT 4º DC		5.49.b	

Tabla 5.2: Resumen de resultados I para las pruebas de DC punto/poliedro.

Datos de Entrada					
Tipo Poliedro	Subdivisiones	Iteraciones A.A.	Mín. Tetra. Clasif	Trayectoria	Algoritmos
	512	8	0 2 4 8 16	Circular Contorno	DC-TT (Algoritmo 5.2) Inc-TT (Inclusión con Tetra-Tree) Inc-Oct (Inclusión con Octree)
Leyenda					
Eje X	Eje Y	Gráficos		Comportamiento analizado	
Min.Tetra.Clasif.	Frames/seg	a) Trayectoria de contorno b) Trayectoria circular		DC en función del nº mínimo de tetraedros clasificados en un tetra-cono, tipo de poliedro y trayectoria.	
Resultados					
Tipo Poliedro	Trayectoria	Comportamiento		Figura	
	Contorno y Circular	Mínimo nº tetraedros clasificados: No es conveniente un valor alto		5.51.a-b	
	Contorno	Mínimo nº tetraedros clasificados: No es significativo		5.52.a-b	

Tabla 5.3: Resumen de resultados II para las pruebas de DC punto/poliedro.

5.8.3. Pruebas para la Detección de Colisión esfera/poliedro

A continuación mostraremos las pruebas realizadas para la detección de colisión entre una esfera y un poliedro. Se ha utilizado el Algoritmo 5.3 para los poliedros de la Figura 5.39 y esferas de distinto tamaño. Se ha variado el tamaño relativo de la esfera en relación al tamaño de la esfera envolvente del poliedro, es decir, un tamaño relativo de 1:32 por ejemplo, indica que el radio de la esfera es 1/32 parte del radio de la esfera envolvente del poliedro. Los tamaños relativos considerados han sido 1:1, 1:2, 1:4, 1:8, 1:16, 1:32, 1:64 y 1:1.024, representando este último valor a una esfera puntual.

Se han utilizado los dos tipos de trayectorias consideradas en la sección anterior: una trayectoria circular y una trayectoria por el contorno del poliedro.

Para cada poliedro se ha construido su correspondiente tetra-tree a una profundidad fija, ajustada al número de vértices del poliedro y a la relación entre los tamaños de los objetos. El número de iteraciones del algoritmo de ajuste se ha fijado a 8, y el mínimo número de tetraedros clasificados en un tetra-cono se ha fijado a 3.

En la Tabla 5.4 puede verse un resumen de las pruebas realizadas y de los resultados obtenidos.

Datos de Entrada						
Tipo Poliedro	Subdivisiones	Iteraciones A.A.	Mín. Tetra. Clasif	Relación Tamaño	Trayectoria	Algoritmos
a)  b)  c)  d) 	Constante: Apropiado para el tamaño del poliedro y la relación entre tamaños	8	3	1:1 1:2 1:4 1:8 1:16 1:32 1:64 1:1.024	Circular Contorno	DC-TT (Algoritmo 5.3)
Leyenda						
Eje X	Eje Y	Gráficos		Comportamiento analizado		
Relación Tamaño	Frames/seg	a)  b)  c)  d) 		DC en función de la relación entre el tamaño de los objetos y el tipo de trayectoria.		
Resultados						
Tipo Poliedro	Trayectoria	Comportamiento				Figura
a)  b)  c)  d) 	Contorno y Circular	El algoritmo es mejor mientras más pequeña sea la esfera en relación al tamaño del poliedro. Se comporta mejor en la trayectoria circular que en la de contorno como era de esperar.				5.53 y 5.54

Tabla 5.4: Resumen de resultados para las pruebas de DC esfera/poliedro.

5.8.4. Pruebas para la Detección de Colisión poliedro/poliedro

Finalmente hemos desarrollado una serie de pruebas para la detección de colisión entre poliedros. Utilizamos los algoritmos de DC recursivo y el basado en esfera/poliedro mediante lista de tetra-conos (Algoritmos 5.10 y 5.11 respectivamente) que hemos denominado DC-TT y DC-Lista.

En primer lugar se ha medido el número de frames por segundo obtenidos en la DC entre los poliedros considerados en las pruebas anteriores con una relación de tamaño de 1 a 1, es decir poliedros con un tamaño similar.

Se han utilizado dos tipos de trayectorias, una trayectoria de contorno, en la que se produce colisión entre poliedros en un porcentaje comprendido entre el 5% y el 10% de los casos, de forma similar a la trayectoria considerada en pruebas anteriores; y una trayectoria circular en la que se produce colisión en un porcentaje menor al 5% de los casos.

Para cada poliedro se ha construido su correspondiente tetra-tree a una profundidad constante, ajustada al número de vértices del poliedro y a la relación entre los tamaños de los objetos. El número de iteraciones del algoritmo de ajuste se ha fijado a 8, así como el mínimo número de tetraedros clasificados en un tetra-cono, que se ha fijado a 3.

También se ha comparado la eficacia de los algoritmos para el caso de distintas escalas relativas entre poliedros, para valores de 1:1, 1:2, 1:4, 1:8, 1:16 y 1:32.

En las Tablas 5.5 y 5.6 puede verse un resumen de las pruebas realizadas y de los resultados obtenidos.

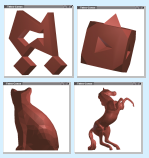
<table>
<tr><th colspan="7">Datos de Entrada</th></tr>
<tr><th colspan="2">Par de Poliedros</th><th>Subdivisiones</th><th>Iteraciones A.A.</th><th>Mín. Tetra. Clasif</th><th>Relación Tamaño</th><th>Trayectoria</th><th>Algoritmos</th></tr>
<tr><td colspan="2">Todas las combinaciones entre:
  </td><td>Constante: Apropiado para el tamaño del poliedro y la relación entre tamaños</td><td>8</td><td>3</td><td>1:1</td><td>Circular
Contorno</td><td>DC-TT (Algoritmo 5.10)
DC-Lista (Algoritmo 5.11)</td></tr>
<tr><th colspan="8">Leyenda</th></tr>
<tr><th>Eje X</th><th>Eje Y</th><th colspan="2">Gráficos</th><th colspan="4">Comportamiento analizado</th></tr>
<tr><td>Pareja de poliedros</td><td>Frames/seg</td><td colspan="2">a) Contorno
b) Circular</td><td colspan="4">DC en función del par de poliedros y el tipo de trayectoria.</td></tr>
<tr><th colspan="8">Resultados</th></tr>
<tr><th colspan="2">Trayectoria</th><th colspan="5">Comportamiento</th><th>Figura</th></tr>
<tr><td colspan="2">Contorno y Circular</td><td colspan="5">Frames/seg en relación al tamaño de los poliedros (tiempo real: >50 Fra./seg).
Mejores tiempos obtenidos por el algoritmo DC-TT que DC-Lista.</td><td>5.55 y 5.56</td></tr>
</table>

Tabla 5.5: Resumen de resultados I para las pruebas de DC poliedro/poliedro.

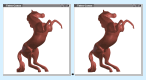
<table>
<tr><th colspan="7">Datos de Entrada</th></tr>
<tr><th colspan="2">Par de Poliedros</th><th>Subdivisiones</th><th>Iteraciones A.A.</th><th>Mín. Tetra. Clasif</th><th>Relación Tamaño</th><th>Trayectoria</th><th>Algoritmos</th></tr>
<tr><td colspan="2">Pareja de poliedros:
  </td><td>Constante: Apropiado para el tamaño del poliedro y la relación entre tamaños</td><td>8</td><td>3</td><td>1:1
1:2
1:4
1:8
1:16</td><td>Circular
Contorno</td><td>DC-TT (Algoritmo 5.10)
DC-Lista (Algoritmo 5.11)</td></tr>
<tr><th colspan="8">Leyenda</th></tr>
<tr><th>Eje X</th><th>Eje Y</th><th colspan="2">Gráficos</th><th colspan="4">Comportamiento analizado</th></tr>
<tr><td>Relación Tamaño</td><td>Frames/seg</td><td colspan="2">a) Contorno
b) Circular</td><td colspan="4">DC en función de la relación de tamaño entre poliedros y el tipo de trayectoria.</td></tr>
<tr><th colspan="8">Resultados</th></tr>
<tr><th colspan="2">Trayectoria</th><th colspan="5">Comportamiento</th><th>Figura</th></tr>
<tr><td colspan="2">Contorno y Circular</td><td colspan="5">Para tamaños similares entre poliedros se comporta mejor el algoritmo DC-TT, sin embargo, cuando un poliedro es mucho más pequeño que otro, se comporta mejor el algoritmo DC-Lista</td><td>5.57</td></tr>
</table>

Tabla 5.6: Resumen de resultados II para las pruebas de DC poliedro/poliedro.

5.8.5. Resultados obtenidos

5.8.5.1. Detección de Colisión Punto/Poliedro

Cuando observamos el comportamiento de los algoritmos para cada tipo de objeto a distintas profundidades en el tetra-tree (número de subdivisiones), y con distintos tipos de trayectorias del punto en relación al poliedro (de contorno y circular), podemos ver que en el caso del *Poliedro Complejo I* y la trayectoria de contorno, se obtienen mejores resultados con el algoritmo DC-TT que con el resto de algoritmos para 128 y 512 subdivisiones (Figura 5.46.a). A partir de 2.048 subdivisiones todos los algoritmos se estabilizan y se igualan entre sí. Estos resultados son debidos al escaso número de caras que forman el poliedro y al tamaño de las mismas. En este caso, subdividir mucho el poliedro empeora los tiempos obtenidos. Ocurre algo similar para el caso de la trayectoria circular (Figura 5.46.b), obteniendo mejores tiempos el algoritmo DC-TT en este caso que en el caso de la trayectoria de contorno.

En cambio para el caso del *Poliedro Complejo II*, se observa una mejora considerable del algoritmo DC-TT al aumentar el número de subdivisiones, estabilizándose a partir de un determinado valor (Figura 5.47). Esto es debido a la forma del poliedro, más regular que la del poliedro anterior, por lo que se produce un mejor ajuste de los tetraedros envolventes; y es debido también al tamaño de las caras, la mayor parte de ellas más pequeñas que en el caso del *Poliedro Complejo I*. El algoritmo de inclusión pierde efectividad a partir de una determinada profundidad, debido a que aumenta el número de nodos a consultar para descender a la profundidad máxima, sin que por ello disminuya el número de tetraedros clasificados en niveles más profundos del árbol.

Para un poliedro con un número intermedio de vértices, como el caso del *Gato*, podemos ver que el algoritmo DC-TT obtiene muy buenos resultados para un determinado valor en el número de subdivisiones para cada uno de los dos tipos de trayectorias consideradas, y que a partir de ese valor, el incremento de subdivisiones no supone ventaja alguna (Figura 5.48).

Cuando se incrementa el número de vértices del poliedro, como el caso del *Caballo*, es necesaria una mayor profundidad en el tetra-tree para obtener los mejores resultados (Figura 5.49). Además de influir factores como la forma o la irregularidad del poliedro en los tiempos obtenidos, influye el tamaño de los triángulos que forman el poliedro en relación al tamaño de la trayectoria generada.

En general se obtienen mejores resultados para la trayectoria circular que para la trayectoria de contorno. Esto es debido a que se detecta rápidamente que no se produce colisión entre el punto y el poliedro porque en muchas ocasiones el punto no

se encuentra en el tetraedro envolvente del correspondiente tetra-cono, pues en la trayectoria circular el punto se encuentra más alejado del poliedro muchas más veces que en el caso de una trayectoria de contorno, que está siempre muy próxima al objeto.

En la Figura 5.50 podemos ver un resumen de los tiempos obtenidos en figuras anteriores, pero en función del número de vértices de los poliedros, y esta vez sólo para el caso de 512 subdivisiones.

Por último, en la Figura 5.51 y 5.52 mostramos la influencia del criterio de parada utilizado para no dividir un tetra-cono o una celda en el caso de un octree (mínimo número de tetraedros o de caras clasificados respectivamente), para dos de los objetos considerados en las pruebas (*Poliedro Complejo II* y *Caballo*). Podemos apreciar que para poliedros con pocas caras no es conveniente utilizar un valor alto para el mínimo de triángulos o caras clasificadas por tetra-cono o celda. Sin embargo, para poliedros con muchas caras este valor no tiene una relevancia significativa en los tiempos obtenidos.

5.8.5.2. Detección de Colisión Esfera/Poliedro

El algoritmo utilizado en estas pruebas (Algoritmo 5.3) se comporta mejor cuanto mayor es la diferencia de tamaño entre el poliedro y la esfera (cuanto menor es la esfera, ver Figura 5.53). Se obtienen mejores resultados para la trayectoria circular que para la de contorno. Esto era de esperar, pues en la trayectoria circular hay ocasiones en las que la esfera está más alejada del poliedro que en el caso de la trayectoria de contorno, en la que la esfera siempre está cerca del poliedro.

Si comparamos entre sí los tiempos obtenidos con una determinada relación de tamaño para todos los poliedros (Figura 5.54), podemos establecer que el algoritmo depende no sólo del número de tetraedros del recubrimiento de un poliedro, sino de la complejidad del mismo, pues en el caso del *Poliedro Complejo I* y el *Poliedro Complejo II*, aunque el primero tiene menor número de caras y de vértices que el segundo, obtenemos peores tiempos debido sobre todo al tamaño de algunas de sus caras y la complejidad de las mismas (debido a que tienen agujeros o a su forma más irregular). Además podemos observar que para el caso del *Caballo*, cuando la esfera sigue una trayectoria circular, se obtienen mejores tiempos que para el caso del *Gato*. Esto es debido a la forma de los objetos, más irregular en el caso del *Caballo*, por lo que la trayectoria que sigue la esfera está más alejada de éste que en el caso del *Gato*, que tiene una forma más regular.

Estos resultados nos inducen a plantear la posibilidad de utilización del algoritmo esfera/poliedro para el caso de detección de colisión entre poliedros (Algoritmo 5.11) cuando el tamaño relativo entre los objetos es considerable.

5.8.5.3. Detección de Colisión Poliedro/Poliedro

Hemos medido el número de frames por segundo obtenidos en la detección de colisión entre todas las parejas que pueden formarse con los poliedros considerados en pruebas anteriores (*Poliedro Complejo I*, *Poliedro Complejo II*, *Gato* y *Caballo*), para una trayectoria de contorno y una circular (Figura 5.55). Utilizando el Algoritmo 5.10 (DC-TT) hemos obtenido resultados satisfactorios, pues en todos los casos se ejecuta el algoritmo de detección de colisión en tiempo real (superior a 50 frames/seg.).

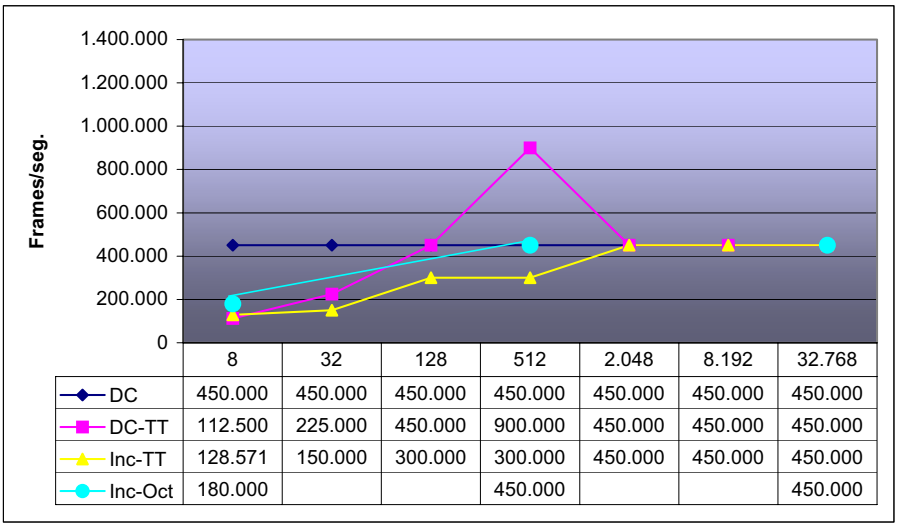
También observamos que se obtienen mejores resultados para una trayectoria de contorno que para una trayectoria circular, debido a que en esta trayectoria se produce mayor número de colisiones que en la trayectoria circular, y a que se aprovecha en mejor medida la coherencia temporal. Al igual que en otros casos descritos anteriormente, se obtienen mejores resultados para el *Poliedro Complejo I* en relación al *Poliedro Complejo II* debido a la complejidad de este último.

En la Figura 5.56 se han comparado los tiempos obtenidos por el algoritmo anterior (DC-TT) y la versión del algoritmo basada en el algoritmo esfera/poliedro (Algoritmo 5.11 ó DC-Lista) para ambos tipos de trayectorias. Vemos que en el caso de una relación 1:1 entre tamaños el algoritmo DC-TT es notablemente superior al algoritmo DC-Lista.

Sin embargo, cuando variamos la relación entre el tamaño de los objetos (Figura 5.57) obtenemos resultados similares para los dos algoritmos cuando la relación es de 1:4 y mejores para el algoritmo DC-Lista cuando esta relación es superior. Cuando los objetos tienen un tamaño similar funciona mejor el algoritmo DC-TT. Estos resultados se encuentran en armonía con los resultados obtenidos para el caso esfera/poliedro.

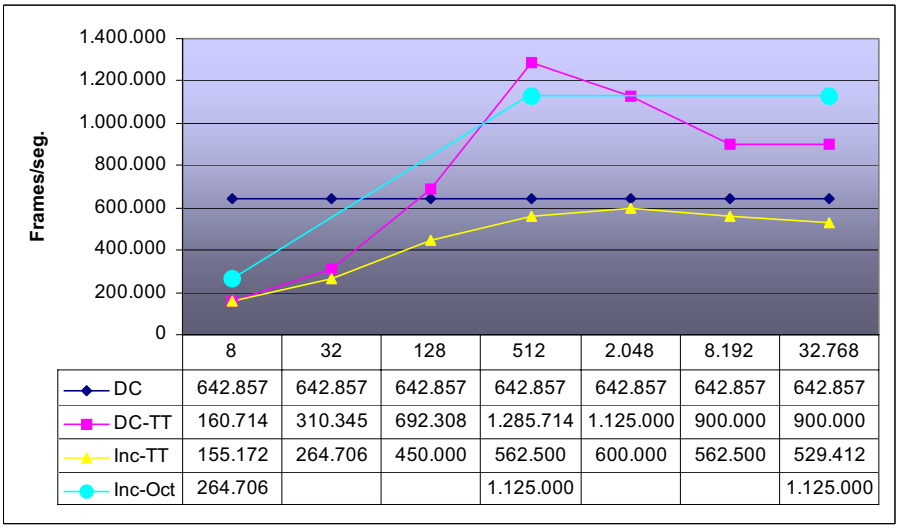
5.8.6. Figuras con los tiempos obtenidos

a)



	8	32	128	512	2.048	8.192	32.768
DC	450.000	450.000	450.000	450.000	450.000	450.000	450.000
DC-TT	112.500	225.000	450.000	900.000	450.000	450.000	450.000
Inc-TT	128.571	150.000	300.000	300.000	450.000	450.000	450.000
Inc-Oct	180.000			450.000			450.000

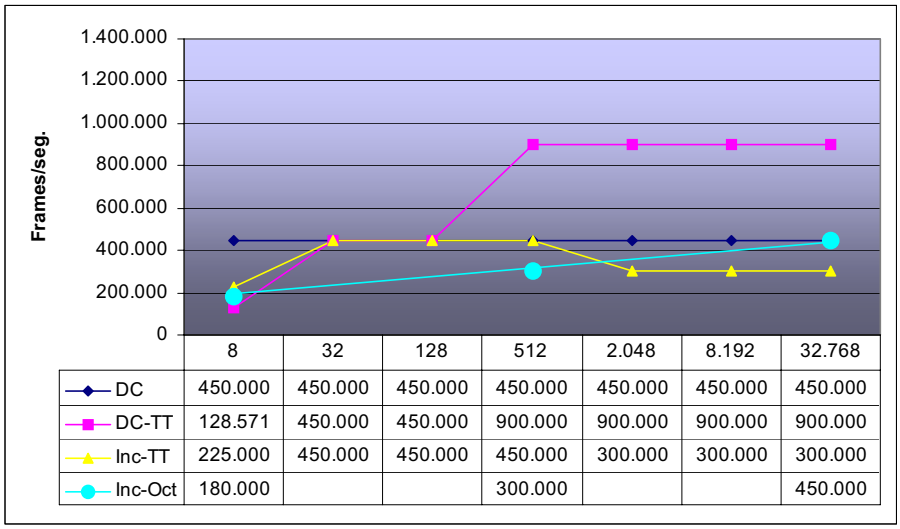
b)



	8	32	128	512	2.048	8.192	32.768
DC	642.857	642.857	642.857	642.857	642.857	642.857	642.857
DC-TT	160.714	310.345	692.308	1.285.714	1.125.000	900.000	900.000
Inc-TT	155.172	264.706	450.000	562.500	600.000	562.500	529.412
Inc-Oct	264.706			1.125.000			1.125.000

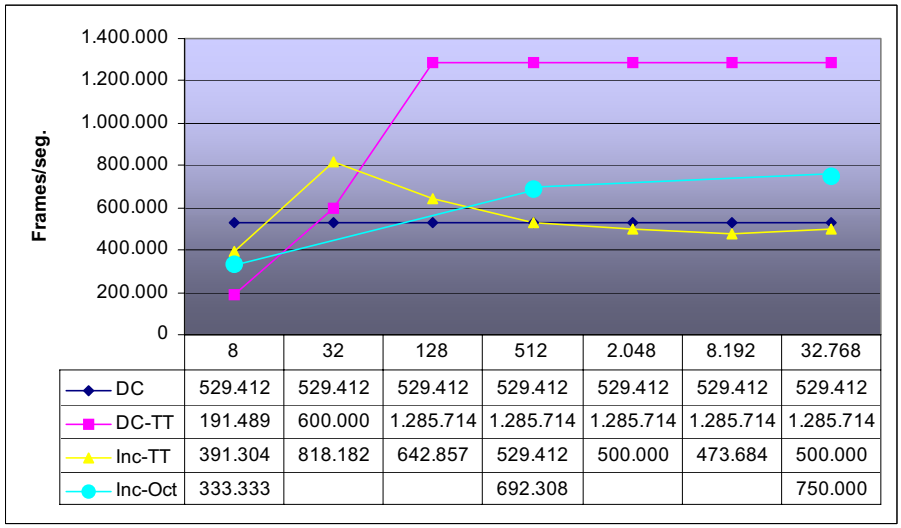
Figura 5.46: DC entre un Punto y el *Poliedro Complejo I*. En el eje X se muestra el número de subdivisiones y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

a)



	8	32	128	512	2.048	8.192	32.768
DC	450.000	450.000	450.000	450.000	450.000	450.000	450.000
DC-TT	128.571	450.000	450.000	900.000	900.000	900.000	900.000
Inc-TT	225.000	450.000	450.000	450.000	300.000	300.000	300.000
Inc-Oct	180.000			300.000			450.000

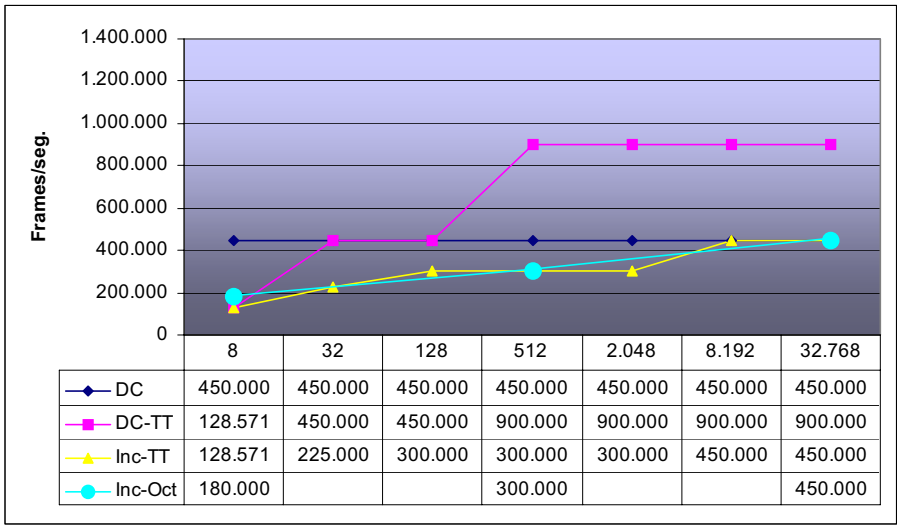
b)



	8	32	128	512	2.048	8.192	32.768
DC	529.412	529.412	529.412	529.412	529.412	529.412	529.412
DC-TT	191.489	600.000	1.285.714	1.285.714	1.285.714	1.285.714	1.285.714
Inc-TT	391.304	818.182	642.857	529.412	500.000	473.684	500.000
Inc-Oct	333.333			692.308			750.000

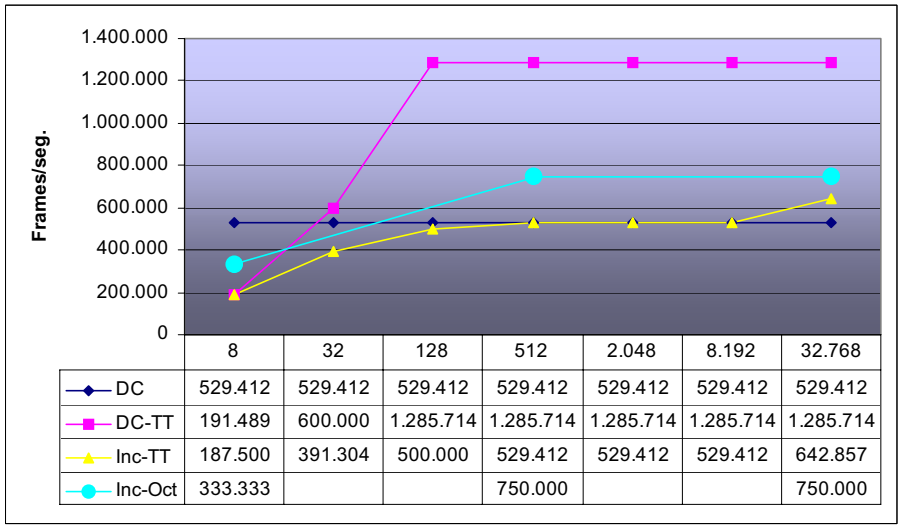
Figura 5.47: DC entre un Punto y el *Poliedro Complejo II*. En el eje X se muestra el número de subdivisiones y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

a)



	8	32	128	512	2.048	8.192	32.768
DC	450.000	450.000	450.000	450.000	450.000	450.000	450.000
DC-TT	128.571	450.000	450.000	900.000	900.000	900.000	900.000
Inc-TT	128.571	225.000	300.000	300.000	300.000	450.000	450.000
Inc-Oct	180.000			300.000			450.000

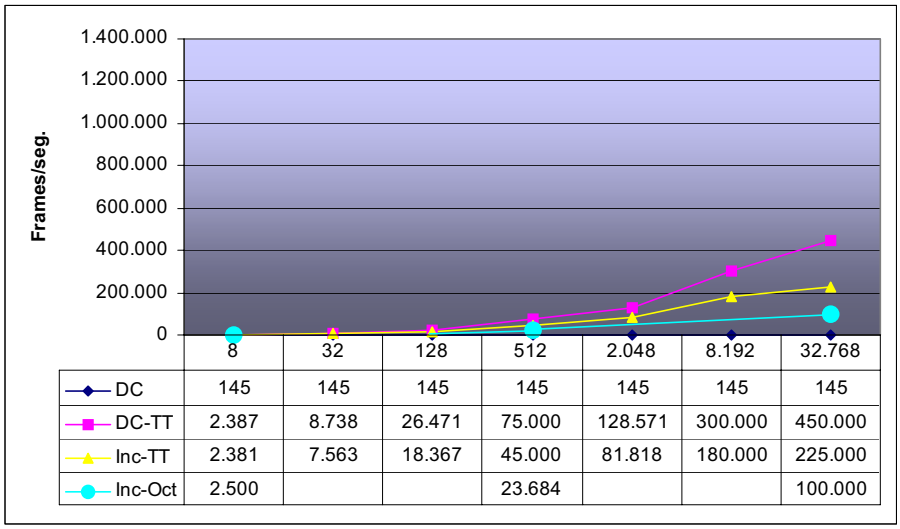
b)



	8	32	128	512	2.048	8.192	32.768
DC	529.412	529.412	529.412	529.412	529.412	529.412	529.412
DC-TT	191.489	600.000	1.285.714	1.285.714	1.285.714	1.285.714	1.285.714
Inc-TT	187.500	391.304	500.000	529.412	529.412	529.412	642.857
Inc-Oct	333.333			750.000			750.000

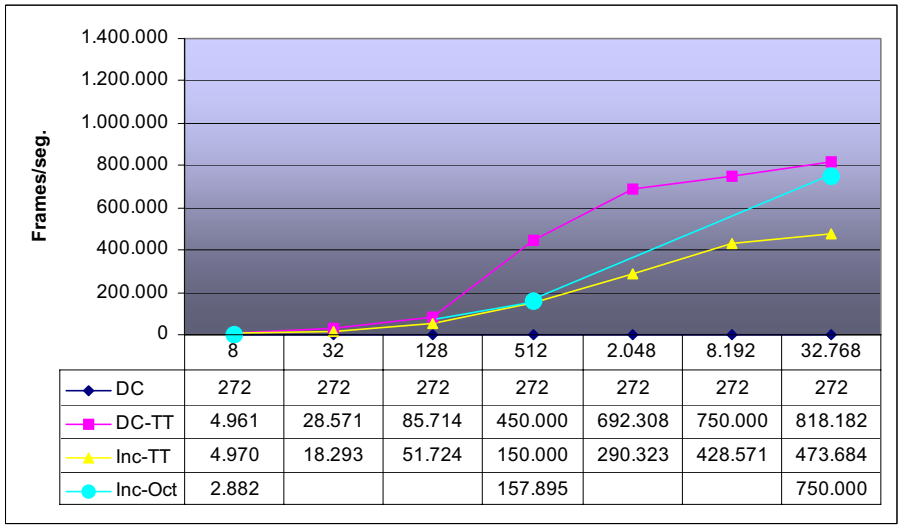
Figura 5.48: DC entre un Punto y el *Gato*. En el eje X se muestra el número de subdivisiones y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

a)



	8	32	128	512	2.048	8.192	32.768
DC	145	145	145	145	145	145	145
DC-TT	2.387	8.738	26.471	75.000	128.571	300.000	450.000
Inc-TT	2.381	7.563	18.367	45.000	81.818	180.000	225.000
Inc-Oct	2.500			23.684			100.000

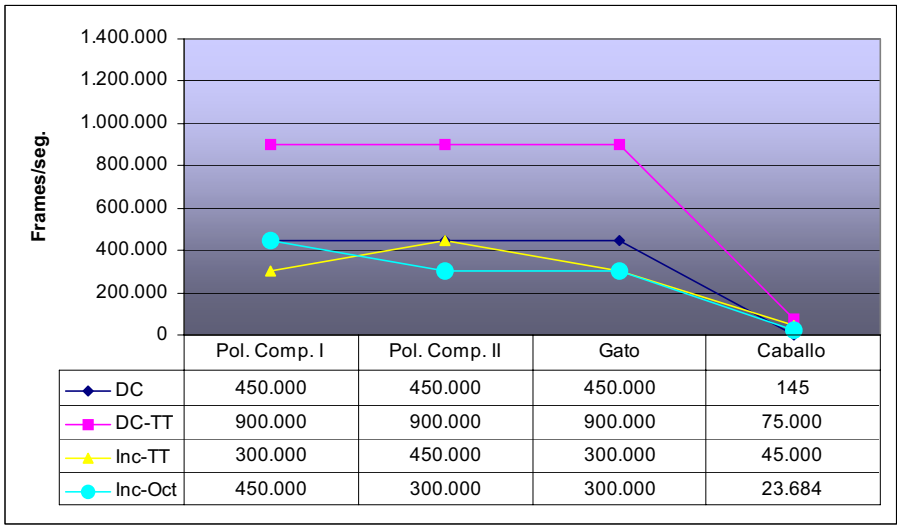
b)



	8	32	128	512	2.048	8.192	32.768
DC	272	272	272	272	272	272	272
DC-TT	4.961	28.571	85.714	450.000	692.308	750.000	818.182
Inc-TT	4.970	18.293	51.724	150.000	290.323	428.571	473.684
Inc-Oct	2.882			157.895			750.000

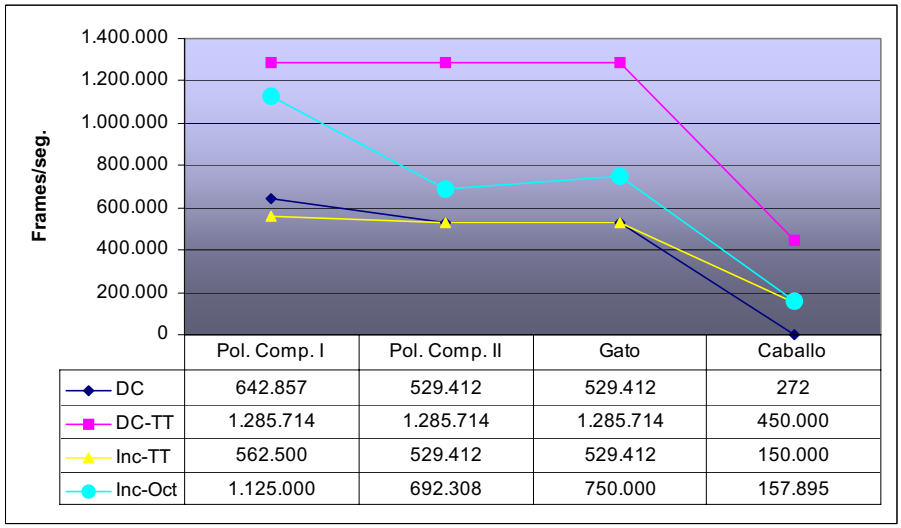
Figura 5.49: DC entre un Punto y el *Caballo*. En el eje X se muestra el número de subdivisiones y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

a)



	Pol. Comp. I	Pol. Comp. II	Gato	Caballo
DC	450.000	450.000	450.000	145
DC-TT	900.000	900.000	900.000	75.000
Inc-TT	300.000	450.000	300.000	45.000
Inc-Oct	450.000	300.000	300.000	23.684

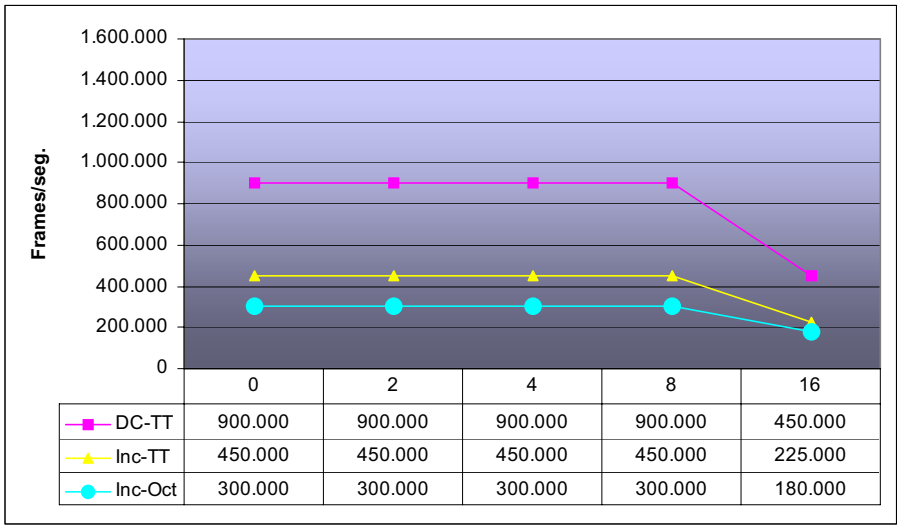
b)



	Pol. Comp. I	Pol. Comp. II	Gato	Caballo
DC	642.857	529.412	529.412	272
DC-TT	1.285.714	1.285.714	1.285.714	450.000
Inc-TT	562.500	529.412	529.412	150.000
Inc-Oct	1.125.000	692.308	750.000	157.895

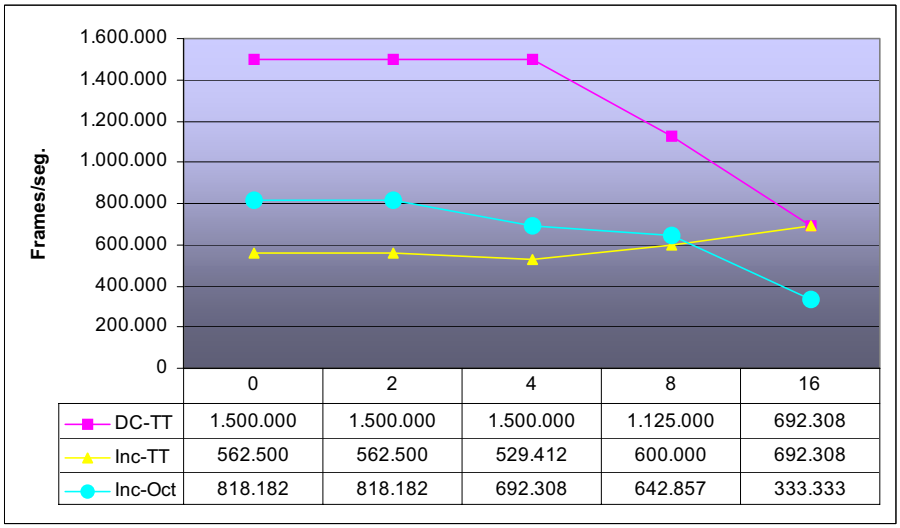
Figura 5.50: DC entre un Punto y los distintos poliedros utilizados en las pruebas. En el eje X se muestra el poliedro en orden ascendente en cuanto número de vértices y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

a)



	0	2	4	8	16
DC-TT	900.000	900.000	900.000	900.000	450.000
Inc-TT	450.000	450.000	450.000	450.000	225.000
Inc-Oct	300.000	300.000	300.000	300.000	180.000

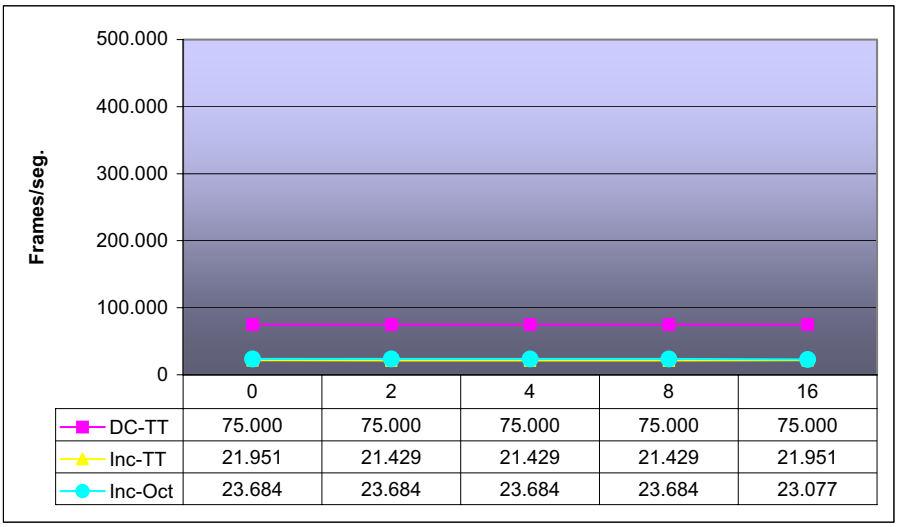
b)



	0	2	4	8	16
DC-TT	1.500.000	1.500.000	1.500.000	1.125.000	692.308
Inc-TT	562.500	562.500	529.412	600.000	692.308
Inc-Oct	818.182	818.182	692.308	642.857	333.333

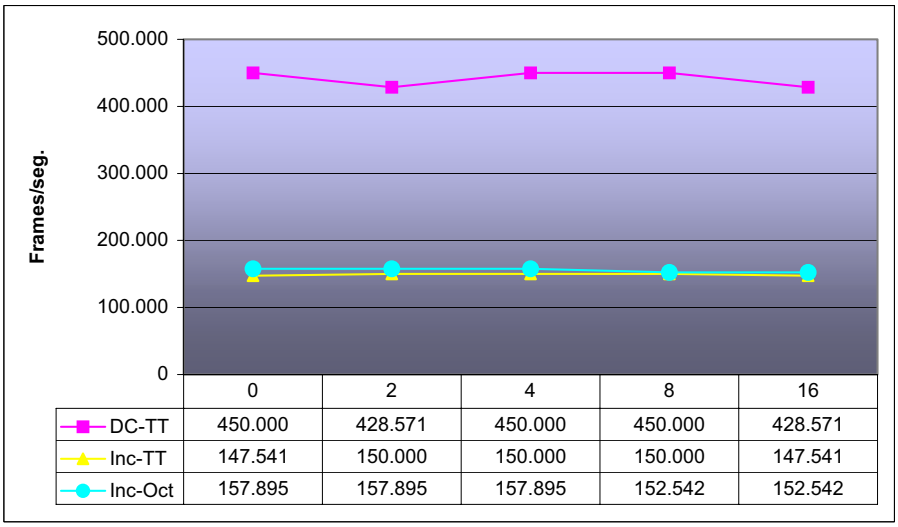
Figura 5.51: DC entre un Punto y el *Poliedro Complejo II* para 512 subdivisiones del espacio. En el eje X se muestra el mínimo número de tetraedros clasificados por tetra-cono y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

a)



	0	2	4	8	16
DC-TT	75.000	75.000	75.000	75.000	75.000
Inc-TT	21.951	21.429	21.429	21.429	21.951
Inc-Oct	23.684	23.684	23.684	23.684	23.077

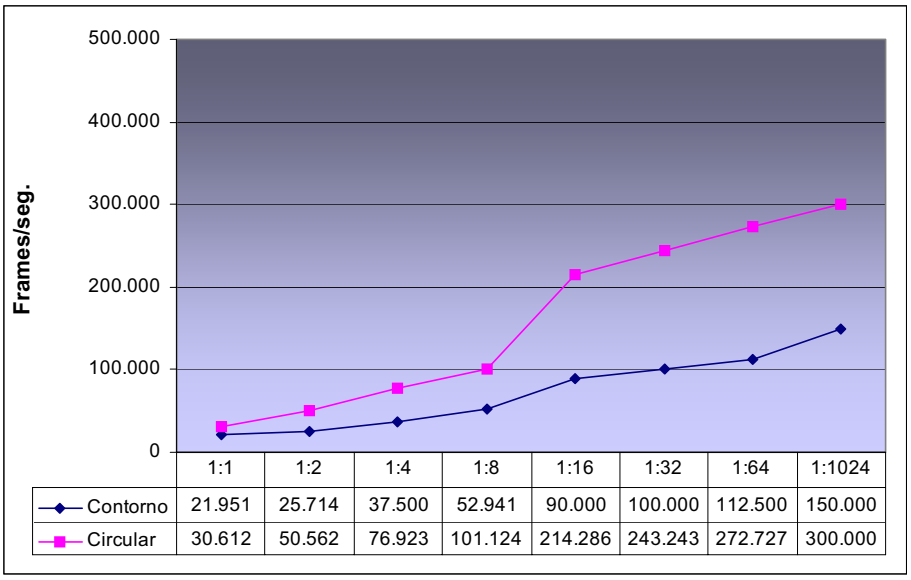
b)



	0	2	4	8	16
DC-TT	450.000	428.571	450.000	450.000	428.571
Inc-TT	147.541	150.000	150.000	150.000	147.541
Inc-Oct	157.895	157.895	157.895	152.542	152.542

Figura 5.52: DC entre un Punto y el *Caballo* para 512 subdivisiones del espacio. En el eje X se muestra el mínimo número de tetraedros clasificados por tetra-cono y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.

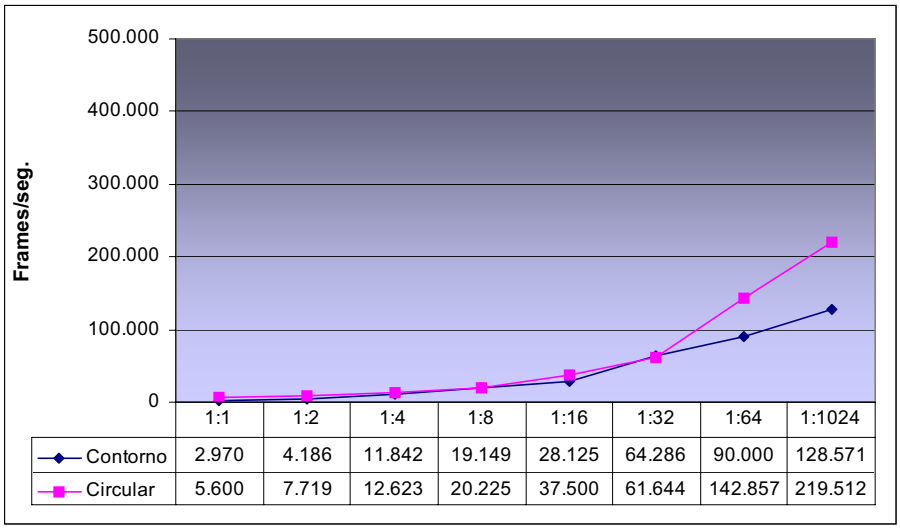
a)



b)

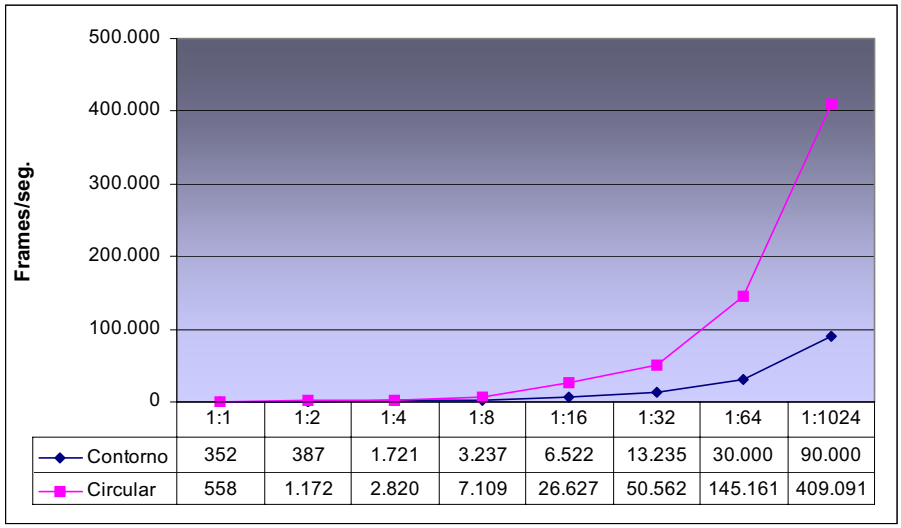
Figura 5.53: DC esfera/poliedro para trayectoria de contorno y circular. En el eje X se muestra el tamaño relativo entre objetos y en el eje Y el número de frames por segundo. a) Figura Compleja I. b) Figura Compleja II. c) Gato. d) Caballo.

c)



	1:1	1:2	1:4	1:8	1:16	1:32	1:64	1:1024
Contorno	2.970	4.186	11.842	19.149	28.125	64.286	90.000	128.571
Circular	5.600	7.719	12.623	20.225	37.500	61.644	142.857	219.512

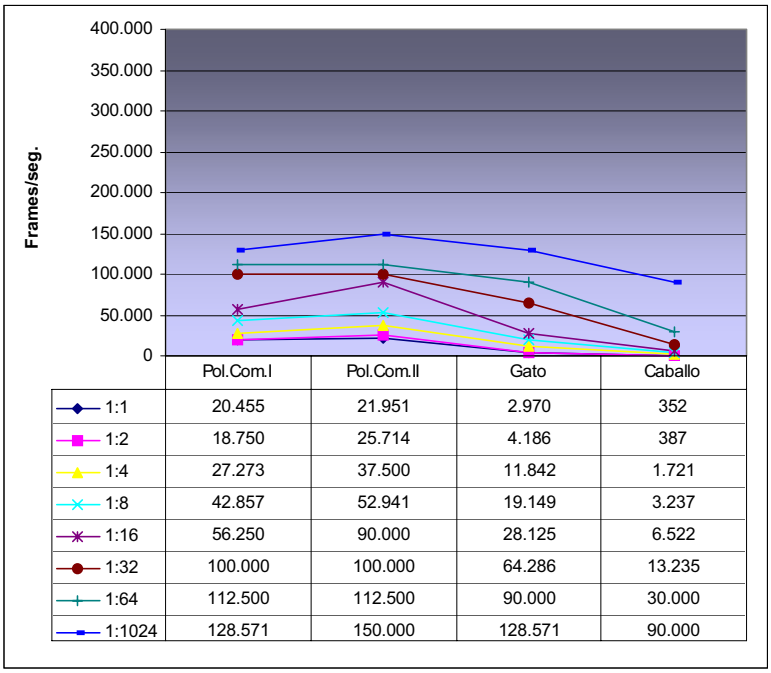
d)



	1:1	1:2	1:4	1:8	1:16	1:32	1:64	1:1024
Contorno	352	387	1.721	3.237	6.522	13.235	30.000	90.000
Circular	558	1.172	2.820	7.109	26.627	50.562	145.161	409.091

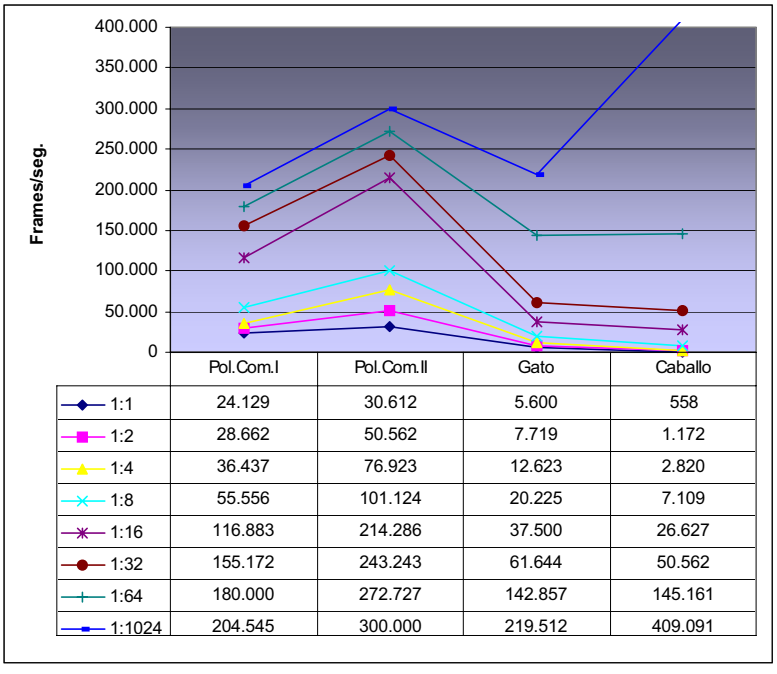
Figura 5.53: DC esfera/poliedro para trayectoria de contorno y circular. En el eje X se muestra el tamaño relativo entre objetos y en el eje Y el número de frames por segundo. a) Figura Compleja I. b) Figura Compleja II. c) Gato. d) Caballo.

a)



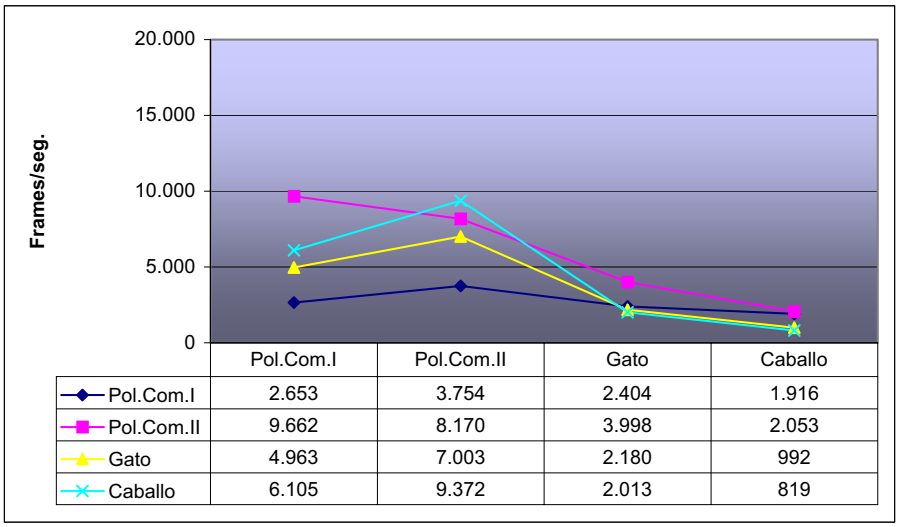
	Pol.Com.I	Pol.Com.II	Gato	Caballo
1:1	20.455	21.951	2.970	352
1:2	18.750	25.714	4.186	387
1:4	27.273	37.500	11.842	1.721
1:8	42.857	52.941	19.149	3.237
1:16	56.250	90.000	28.125	6.522
1:32	100.000	100.000	64.286	13.235
1:64	112.500	112.500	90.000	30.000
1:1024	128.571	150.000	128.571	90.000

b)



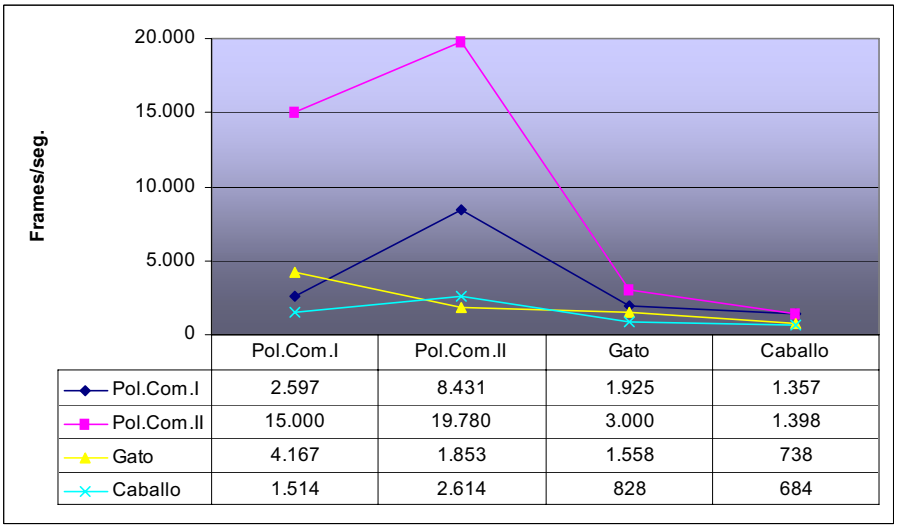
	Pol.Com.I	Pol.Com.II	Gato	Caballo
1:1	24.129	30.612	5.600	558
1:2	28.662	50.562	7.719	1.172
1:4	36.437	76.923	12.623	2.820
1:8	55.556	101.124	20.225	7.109
1:16	116.883	214.286	37.500	26.627
1:32	155.172	243.243	61.644	50.562
1:64	180.000	272.727	142.857	145.161
1:1024	204.545	300.000	219.512	409.091

Figura 5.54: DC esfera/poliedro para distintas relaciones de tamaño entre objetos. En el eje X se muestra el tipo de poliedro y en el eje Y el número de frames por segundo. a) Para trayectoria de contorno. b) Para trayectoria circular.



	Pol.Com.I	Pol.Com.II	Gato	Caballo
Pol.Com.I	2.653	3.754	2.404	1.916
Pol.Com.II	9.662	8.170	3.998	2.053
Gato	4.963	7.003	2.180	992
Caballo	6.105	9.372	2.013	819

a)

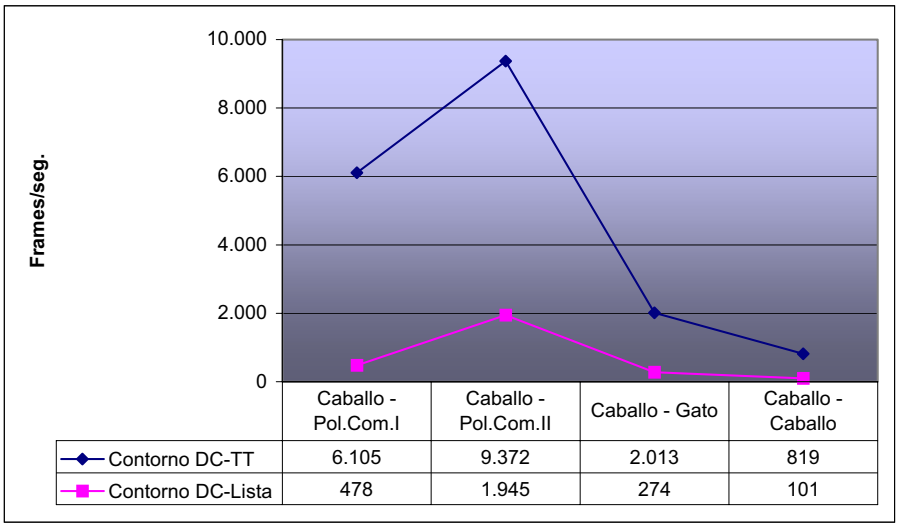


	Pol.Com.I	Pol.Com.II	Gato	Caballo
Pol.Com.I	2.597	8.431	1.925	1.357
Pol.Com.II	15.000	19.780	3.000	1.398
Gato	4.167	1.853	1.558	738
Caballo	1.514	2.614	828	684

b)

Figura 5.55: DC entre poliedros con el algoritmo DC-TT para el par *Poliedro 1-Poliedro 2*. En la tabla, se representa en filas el *Poliedro 1* y en columnas el *Poliedro 2*. En el eje Y se muestra el número de frames por segundo. a) Para una trayectoria de contorno. b) Para una trayectoria circular.

a)



b)

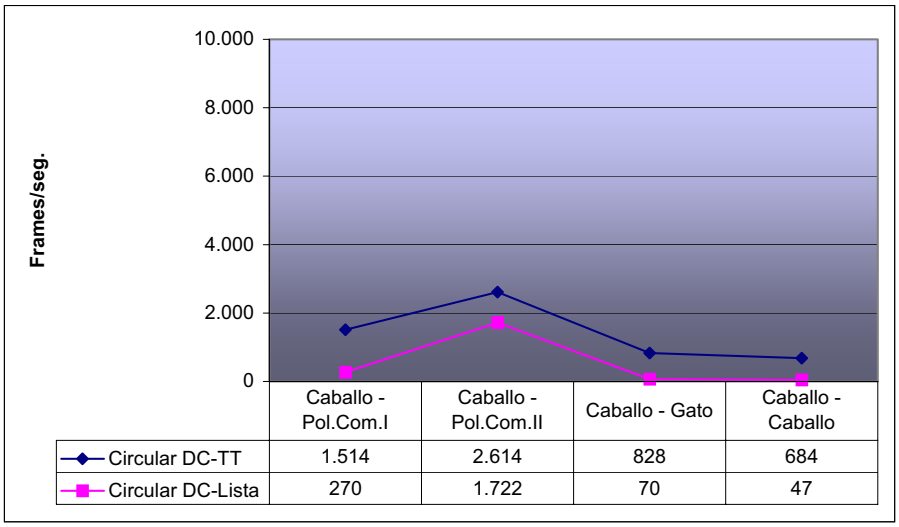
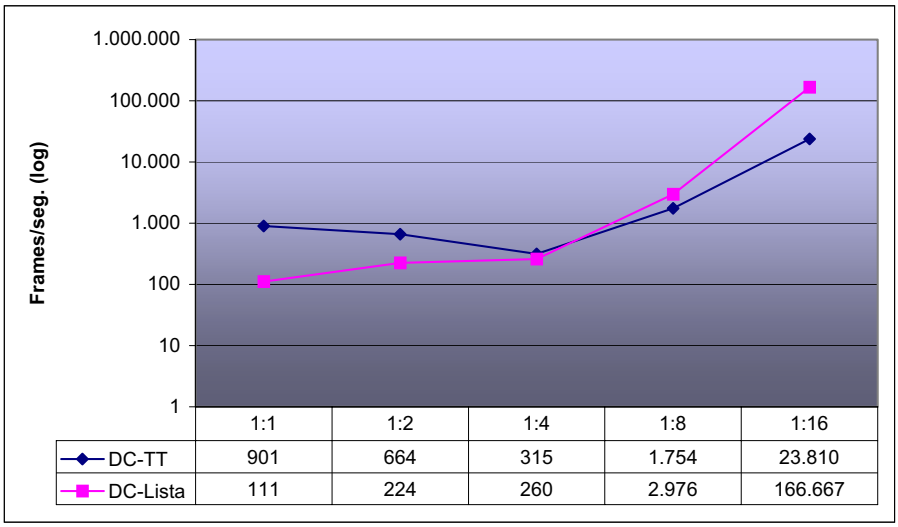


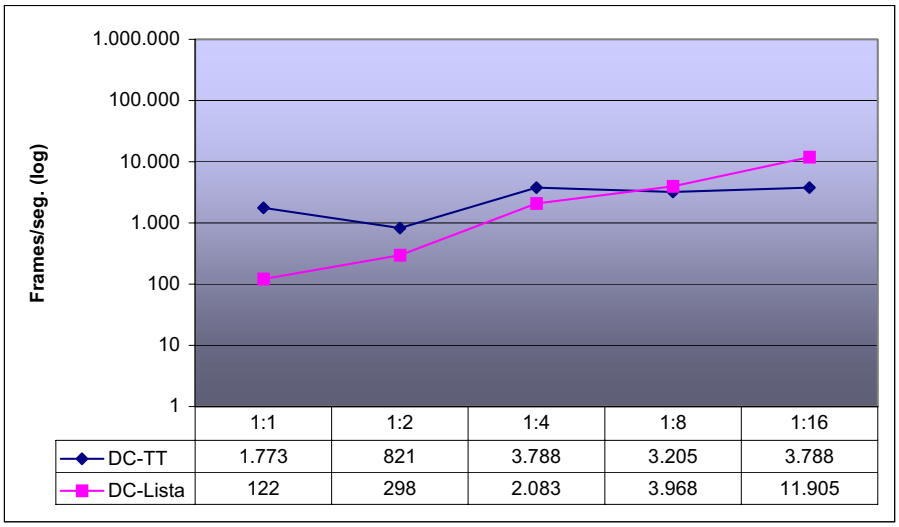
Figura 5.56: Comparación de la DC entre poliedros con el algoritmo DC-TT y DC-Lista. En el eje X se muestra el par de poliedros utilizados y en el eje Y el número de frames por segundo. a) Para una trayectoria de contorno. b) Para una trayectoria circular.

a)



	1:1	1:2	1:4	1:8	1:16
DC-TT	901	664	315	1.754	23.810
DC-Lista	111	224	260	2.976	166.667

b)



	1:1	1:2	1:4	1:8	1:16
DC-TT	1.773	821	3.788	3.205	3.788
DC-Lista	122	298	2.083	3.968	11.905

Figura 5.57: DC entre el par de poliedros *Caballo-Caballo* con distintas relaciones de tamaño. En el eje X se muestra la relación entre el tamaño de los poliedros y en el eje Y el número de frames por segundo con escala logarítmica. a) Para una trayectoria de contorno. b) Para una trayectoria circular.

5.9. Conclusiones

En este capítulo hemos desarrollado un conjunto de técnicas y algoritmos para la detección de colisión en 3D utilizando recubrimientos simpliciales. Hemos desarrollado un nuevo tipo de descomposición espacial denominada Tetra-Tree basada en tetra-conos que hemos usado en la DC. Este tipo de descomposición espacial se ajusta a los poliedros en mejor medida que otro tipo de descomposiciones.

Según las pruebas realizadas la construcción de un tetra-tree es bastante rápida, realizándose en tiempo de pre-procesamiento, y no añadiendo un coste suplementario a los algoritmos de DC. Un tetra-tree se puede ajustar adecuadamente a la forma de un poliedro por lo que podría utilizarse para el modelado, siendo una estructura de datos que no varía cuando se producen transformaciones rígidas, por lo que no es necesario recalcularla. Su utilización en la detección de colisión es apropiada en cuanto a que el tiempo de acceso a sus elementos es menor comparado con otros tipos de descomposiciones espaciales como octrees.

En cuanto a la DC punto/poliedro, hemos desarrollado un algoritmo eficiente que, mediante el uso de tetra-trees y gracias a la coherencia temporal, localiza rápidamente la posición espacial de un punto y obtiene la colisión utilizando sólo los tetraedros del recubrimiento clasificados en un tetra-cono.

Hemos desarrollado también un algoritmo de DC esfera/poliedro, extendiendo conceptos desarrollados para la DC entre puntos y poliedros. Esto nos ha llevado a obtener tiempos cercanos a los obtenidos con este último método, sobre todo cuando la esfera es pequeña en relación al tamaño del poliedro. Para el resto de situaciones nos ofrece una aproximación a la DC entre poliedros, utilizando la esfera como volumen envolvente de uno de los poliedros, permitiendo obtener algoritmos de DC poliedro/poliedro más eficientes para el caso de poliedros con gran diferencia de tamaño.

Por último para la detección de colisión entre poliedros, además del algoritmo basado en la DC esfera/poliedro, hemos desarrollado otro algoritmo que recorre recursivamente los tetra-trees de ambos poliedros, localizando rápidamente los tetraedros envolventes que intersecan. Este algoritmo es adecuado cuando los poliedros tienen un tamaño similar.

CAPÍTULO

6

Hardware Gráfico y Detección de Colisiones

En este capítulo hablaremos de distintas técnicas utilizadas para la detección de colisión mediante el uso de tarjetas gráficas programables. Por último presentaremos una implementación basada en la GPU de un algoritmo de detección de colisión entre una nube de partículas y un poliedro.

CONTENIDOS:

6. Hardware Gráfico y Detección de Colisiones

Últimamente ha aumentado el uso de la Unidad de Procesamiento Gráfico (GPU) para la resolución de problemas de ámbito general no dedicados a la propia visualización. Los avances en las tarjetas gráficas y los lenguajes de programación adecuados a las mismas llevan a plantear la resolución de problemas de detección de colisión mediante el uso compartido de la GPU y la CPU. A lo largo de este capítulo estudiaremos por una parte la resolución de problemas de detección de colisión en la GPU, y por otra particularizaremos para el problema de detección de colisión entre una nube de partículas y un poliedro.

Uno de los motivos que nos lleva a utilizar la GPU para este tipo de problemas reside en su capacidad para realizar cálculos en forma de flujo de datos formados por vectores y a su alto grado de paralelismo en las operaciones realizadas, siendo posible además utilizar conjuntamente la CPU y la GPU para resolver este tipo de problemas. Otros motivos son: el alto ancho de banda en las transferencias de información en la memoria de la tarjeta gráfica; el menor coste de la GPU; y la potencial paralelización de tarjetas gráficas, aspecto que nos hace pensar en un mayor rendimiento futuro cuando el procesador coopere con varias tarjetas gráficas.

6.1. Detección de colisiones en la GPU

Para determinar la colisión entre objetos existen dos técnicas principales que utilizan la GPU. La primera consiste en utilizar la GPU para obtener la intersección entre objetos en el espacio de la imagen, y la segunda en utilizar la GPU como un procesador más que acelere cálculos matemáticos o geométricos.

Las técnicas basadas en el espacio de la imagen suelen ser de fácil implementación pues trabajan sobre un conjunto de primitivas rasterizables y no necesitan de estructuras de datos complejas sobre las que operar. Debido a que los tests de colisión dependen de la resolución de los buffers sobre los que se dibujan los objetos, estos métodos son aproximados. Además la colisión puede o no detectarse dependiendo de la proximidad de dichos objetos al plano de visión.

En cuanto al segundo tipo de técnicas es necesario codificar la información de la geometría de los objetos en la forma de texturas, y adaptar los algoritmos de manera que puedan ejecutarse en la forma de flujos de datos sobre procesadores de vértices y fragmentos.

Algunos de los algoritmos más destacados para la detección de colisión comprenden el de Govindaraju et. al [GLM06] que utiliza una técnica de visibilidad para mallas de triángulos junto al algoritmo CULLIDE; el de Choi et. al [CKK06] que utiliza la GPU para obtener las auto-colisiones entre objetos deformables, usando para ello un esquema de autoalimentación jerárquico que optimiza las transferencias de la GPU a la CPU; el algoritmo de Fan et. al [FWG03] que emplea una técnica en el espacio de la imagen basada en la detección de colisiones por medio de jerarquías de cajas envolventes orientadas; o el algoritmo de Heidelberger [HTG04] que se fundamenta en la utilización de imágenes de profundidad por capas y jerarquías de cajas envolventes alineadas con los ejes para la detección de colisión y de auto-colisión entre objetos deformables.

Veamos a continuación un método representativo que opera para objetos convexos y otro para objetos cóncavos, ambos en el espacio de la imagen y que utilizan la GPU para determinar la colisión entre un par de objetos.

6.1.1. Test de colisión entre objetos convexos

Veamos el método utilizado por [MOK95] y [BWS99]. La idea básica consiste en considerar cada píxel del "frame-buffer" sobre el que se va a dibujar como un rayo perpendicular al plano de visión y lanzado hacia los objetos. Cuando el rayo interseca con un objeto, la intersección se describe mediante un intervalo formado por la primera y la última intersección del rayo. Si tenemos dos objetos convexos, existen nueve posibles casos diferentes de solapamiento de intervalos.

El test de intersección puede llevarse a cabo en dos pasadas sin necesidad de realimentar a partir del "frame-buffer", utilizando para ello técnicas de oclusión. En la primera pasada, se dibuja el primer objeto en el buffer de profundidad, utilizando el test de profundidad *menor-o-igual*. A continuación, el test se cambia a *mayor-que*,

la actualización del buffer de profundidad se desactiva, y se dibuja el segundo objeto con el test de oclusión habilitado. Si el test de oclusión devuelve que no hay píxeles del segundo objeto visibles, este objeto estará totalmente delante del primero, y los objetos no colisionan. En otro caso es necesaria una segunda pasada idéntica a la primera pero intercambiando los dos objetos. Al principio de las dos pasadas se inicializa el buffer de profundidad al valor de Z más lejano. La actualización del buffer de color se desactiva durante el test. No es necesario inicializar el buffer de profundidad en la segunda pasada si la visualización al principio de cada pasada implica sólo a las caras delanteras de los objetos y el test de profundidad está establecido en el valor *siempre*. Del mismo modo, los test de oclusión pueden optimizarse utilizando sólo las caras traseras de los objetos.

Debemos tener en cuenta que si aplicamos este test a objetos cóncavos y obtenemos que los dos objetos no intersecan, realmente no intersecan. En cambio si este método devuelve que los dos objetos cóncavos intersecan, puede ser que realmente intersequen o no. Por tanto, podemos tratar este método como conservativo para objetos cóncavos.

6.1.2. Test de colisión entre objetos cóncavos

Otro método para objetos cóncavos* debido a [KP03] [Kno03] puede utilizarse considerando que dos objetos colisionan si y sólo si tienen al menos un punto en común. Si los objetos son poliédricos, intersecan si y sólo si parte de alguna arista de uno de los objetos atraviesa el volumen del otro objeto. De esta forma, si los dos objetos intersecan, existe al menos un punto en alguna arista de uno de los objetos que pertenece a ambos objetos. Un rayo lanzado desde ese punto en cualquier dirección, pero en particular en la dirección del observador, pasa a través de la frontera de uno de los dos objetos un número impar de veces [Oro94]. Para asegurar que se produce o no colisión debe lanzarse un rayo para todos los puntos en todas las aristas de un objeto contra el volumen del otro. Si no se detecta intersección, debe realizarse de nuevo este test intercambiando los objetos.

Utilizando el hardware gráfico, todos los test de intersección de una pasada pueden llevarse a cabo en paralelo con la ayuda del "stencil-buffer"†. Los pasos a dar son los siguientes:

---

* Evidentemente el método también es válido para objetos convexos.

† Hemos preferido utilizar el término en ingles, debido a su amplio uso.

- Inicialización:
    - Desactivar la escritura en el buffer de color.
    - Inicializar el buffer de profundidad y el "stencil-buffer".
    - Habilitar la comprobación del buffer de profundidad.
- Dibujar las aristas:
    - Habilitar la actualización del buffer de profundidad.
    - Establecer que los píxeles pasen siempre.
    - Dibujar todas las aristas del segundo objeto, escribiendo su profundidad en el buffer de profundidad.
- Dibujar las caras:
    - Deshabilitar la actualización del buffer de profundidad.
    - Establecer que los píxeles pasen si están más próximos que el valor almacenado de profundidad.
    - Habilitar el test del "stencil-buffer".
    - Dibujar las caras delanteras del primer objeto. Incrementar el "stencil-buffer" para los píxeles que pasen.
    - Dibujar las caras traseras del segundo objeto. Decrementar el "stencil-buffer" para los píxeles que pasen.

Después de estos tres pasos, el "stencil-buffer" contiene valores distintos de cero si los dos objetos están en colisión.

Este método tiene algunos inconvenientes que hacen que falle en ciertas situaciones como cuando desde la posición del observador se ve un objeto totalmente contenido en otro.

6.2.Detección de colisión entre una nube de partículas y un poliedro

En esta sección resolveremos un problema de detección de colisión clásico que consiste en obtener las colisiones que se producen entre una nube de partículas y un poliedro. Una nube de partículas es un conjunto de elementos considerados puntuales, de los que nos interesa otro tipo de propiedades como su posición, su masa, movimiento, fuerza aplicada, etc. El movimiento de estas partículas puede ser controlado, caótico o estar inducido por determinadas leyes de la física. [Ebe04].

Para la mayoría de las aplicaciones, un sistema de partículas puede verse desde dos puntos de vista, bien como la representación de un conjunto de puntos en un medio continuo, bien como la descripción del estado dinámico de un sólido. En el contexto del modelado, las partículas se utilizan fundamentalmente de la primera manera señalada, de manera que un sistema de partículas describe la discretización de algún

medio continuo, existiendo múltiples trabajos que así lo consideran, [TK88] [Sta99] [JP02]. Las partículas también pueden ser utilizadas para muestreo de superficies implícitas [WH94].

La mayor parte de los trabajos existentes relacionados con la detección de colisiones de sistemas de partículas mediante el uso de hardware gráfico resuelven el problema en el espacio de la imagen, al estar diseñados para la visualización de las partículas. Así, Kolb et. al [KLR04] plantean un algoritmo basado en el uso de mapas de profundidad para representar la frontera del sólido, almacenando los valores de distancia y normales. Estos mapas se codifican mediante texturas de 8 bits. Knott [Kno03] propone un método de detección de colisiones con sistemas de partículas genérico, basado en el uso de hardware gráfico, si bien plantea el uso de una única etapa del pipeline de la GPU y restringe el número de partículas al número de bucles soportados por la misma. Baciu et. al [BW03] proponen un método de detección de colisiones entre sólidos en el espacio de la imagen. Para ello, modifica el algoritmo Z-buffer y almacena los valores de profundidad en una matriz, siendo necesario más de una pasada para determinar las colisiones. Govindaraju et. al [GLM06] presentan un algoritmo de detección de colisión basado en las extensiones* de ocultación de las nuevas tarjetas gráficas.

Para resolver el problema planteado en esta sección utilizaremos una adaptación del algoritmo de detección de colisión punto/poliedro y lo aplicaremos a cada una de las partículas, si bien podría considerarse utilizar el algoritmo de detección de colisión esfera/poliedro en el caso de que las dimensiones de las partículas fuesen importantes para una aplicación concreta. La inclusión aislada de puntos en poliedros ha sido también resuelta utilizando el hardware gráfico en otros trabajos similares, obteniendo resultados prometedores como puede verse en [OJSF05].

Nuestro algoritmo de detección de colisión [JOSF05] [JOSF06] se ha dividido en dos etapas. En una primera etapa se clasifica cada partícula en el tetra-tree del poliedro, obteniendo un tetra-cono de último nivel. Además se obtiene si la partícula se encuentra en el correspondiente tetraedro envolvente asociado a ese tetra-cono. En la segunda etapa se realiza la inclusión en los tetraedros clasificados en el tetra-cono correspondiente, siempre que la partícula se encuentre incluida en su tetraedro envolvente. Cada una de estas dos etapas se ha implementado en el procesador de vértices y de fragmentos respectivamente (Figura 6.1).

A continuación veremos cual es la información suministrada a la GPU para que pueda realizar esta detección de colisiones en dos etapas, así como la codificación utilizada para almacenar la información de las partículas y del tetra-tree del poliedro.

---

* Son nuevas posibilidades añadidas a la librería gráfica OpenGL, que aprovechan mejor las características particulares de las tarjetas gráficas.

Veremos también la implementación de cada etapa de detección de colisión en la GPU. Hemos utilizado el lenguaje Cg [FK03] para la implementación. Este lenguaje de alto nivel es específico para la programación de la GPU. Las características más importantes de Cg, así como las de la GPU, su funcionamiento, y utilización para problemas de propósito general, pueden consultarse en el Apéndice, necesario para comprender la implementación realizada en este capítulo.

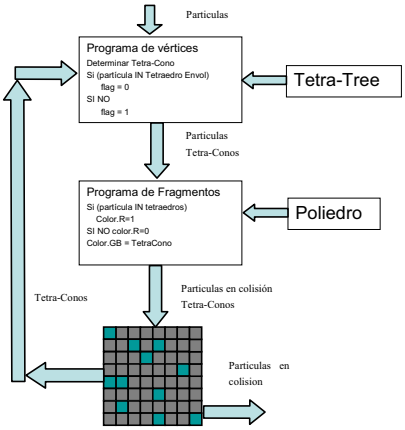


Figura 6.1: Esquema general que muestra las distintas etapas inducidas en la detección de colisión entre una nube partículas y un poliedro.

6.2.1. Intercambio de información

Uno de los principales inconvenientes que tiene el realizar una implementación en la GPU consiste en que hay que adaptar las estructuras de datos y el modelo de programación a unas características particulares, y a un modelo de programación enfocado a la visualización, mucho más restringido que en la programación clásica. Además debe mantenerse un flujo bidireccional de información entre la CPU y la GPU en una forma poco habitual, con la consiguiente penalización en el tiempo de ejecución debido a ese trasiego de información.

El algoritmo de detección de colisión debe tener en cuenta que la información de entrada a la GPU está formada por el conjunto de partículas y ciertas propiedades asociadas a las mismas, en nuestro caso la posición de la partícula en el espacio, por la información geométrica del poliedro, y por otra información adicional como el tetra-tree en la forma de una jerarquía de tetra-conos y tetraedros envolventes, así como los tetraedros del recubrimiento que forman parte de cada tetra-cono. Asimismo debe suministrarse información de realimentación, es decir, información calculada en la GPU que servirá de entrada entre instantes de tiempo consecutivos o

frames. Esta información de realimentación consiste básicamente en el tetra-cono en el que se encontraba dicha partícula en el frame anterior.

Como deseamos que para cada partícula se invoque una vez al algoritmo de detección de colisión con el poliedro, asimilamos el concepto de partícula con el de vértice, de manera que se envía un vértice a la GPU por cada partícula considerada. De este modo se utilizan algunas de las propiedades de los vértices para subministrar la información de entrada acerca de la posición de la partícula y del tetra-cono en el que se encontraba en la iteración anterior. Utilizaremos para ello las *variables de entrada* de Cg*.

Para realizar una implementación eficiente del algoritmo se puede utilizar un array de vértices o de normales de OpenGL para subministrar esta información acerca de las posiciones de las partículas y los tetra-conos en los que se encontraban en el frame anterior. Para el resto de propiedades asociadas a cada partícula, si fuesen necesarias, podrían codificarse en otros arrays de propiedades asociadas a los vértices, como por ejemplo el color.

La información geométrica del poliedro y de su tetra-tree se codifica en la forma de un conjunto de texturas que se subministran una única vez a la GPU, por lo que sólo se producirá una penalización en tiempo de pre-procesamiento al principio del algoritmo, para ello usaremos *variables constantes*. Por comodidad subministramos también el vértice origen del recubrimiento, en nuestro caso el centroide del poliedro, así como el número de niveles del tetra-tree del poliedro, pero mediante sendas *variables de entrada uniformes*.

Al suponer que el poliedro puede moverse libremente (y con él el tetra-tree asociado), es necesario proporcionar también información acerca de la transformación aplicada al mismo entre frames. Esto se hace a través de *variables de entrada uniformes* de Cg. Así cuando se mueve el poliedro, tras obtener la información de los vértices almacenada en la textura, se le aplica la transformación requerida antes de su procesamiento.

La información de salida de la GPU la obtenemos a través del "frame-buffer", en el que se escribe tanto la información acerca del tetra-cono en el que se encuentra dicha partícula y que sirve de realimentación al siguiente frame, como información de su estado de colisión. La información de salida puede obtenerse desde la CPU a través de la lectura del "frame-buffer" o de alguna textura si se utiliza como salida.

---

* Una descripción de los distintos tipos de variables de este lenguaje puede verse en la Sección A.3 del Apéndice.

Otra información de entrada a la GPU, necesaria para poder redirigir la información de salida, está formada por las coordenadas en el "frame-buffer" del píxel donde se desea escribir dicha información para cada partícula, que se realiza mediante *variables de entrada*. Esta información, que no es relevante para la aplicación en sí, lo es para poder asociar el resultado de detección de colisión a cada una de las partículas. También es necesario enviar a través de *variables de entrada uniformes* la matriz de modelado y de visión que se aplicará a las posiciones de los píxeles a los que queremos enviar la información de salida.

A continuación podemos ver un esquema que resume de forma gráfica el intercambio de información que se produce entre CPU y GPU (Figura 6.2 y Tabla 6.1).

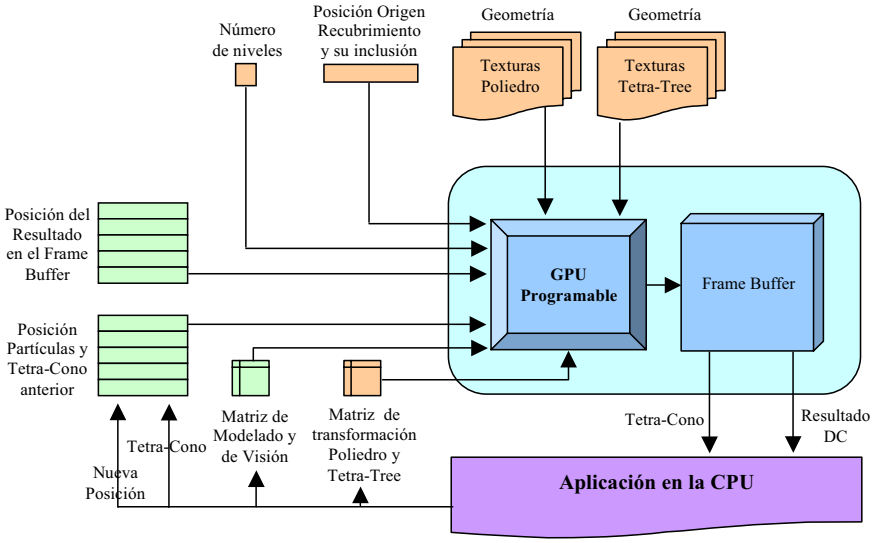


Figura 6.2: Información circulante entre elementos de la CPU y GPU.

Información	*Tipo Variable*	*Formato*	*Origen*
Posición partícula y tetra-cono anterior	*Entrada (flujo datos)*	*Array de Elementos Vector*	*Aplicación y frame-buffer*
Posición resultado en "frame-buffer"	*Entrada (flujo datos)*	*Array de Elementos Vector*	*Aplicación (constante)*
Matriz Modelado y Visión	*Entrada Uniforme*	*Matriz*	*Aplicación*
Geometría Poliedro	*Entrada Constante*	*Texturas*	*Aplicación (constante)*
Geometría Tetra-Tree	*Entrada Constante*	*Texturas*	*Aplicación (constante)*
Origen Recubrimiento y su inclusión	*Entrada Uniforme*	*Vector*	*Aplicación (constante)*
Matriz Transformación Poliedro y Tetra-Tree	*Entrada Uniforme*	*Matriz*	*Aplicación*
Numero niveles	*Entrada Uniforme*	*Real*	*Aplicación (constante)*
Resultado DC	*Salida*	*Píxel frame-buffer real 8bits*	*GPU*
Tetra-Cono	*Salida*	*Píxel frame-buffer real 16bits*	*GPU*

Tabla 6.1: Información de entrada y salida de la GPU.

6.2.2. Pre-procesamiento: codificación de la información

Veamos cómo representar la información que necesita la GPU para el proceso de DC entre una nube de partículas y un poliedro, así como la información de salida hacia la CPU que se obtiene como resultado de dicho proceso.

Información acerca de las partículas

Para cada partícula se proporciona la posición de la partícula, junto con el tetra-cono en el que se encontraba en el frame anterior, y se almacena en una variable de tipo vector con semántica NORMAL*, en la que se guarda la posición de la partícula seguida del código de tetra-cono en el que se encontraba en el frame anterior, en la forma: (pos_x, pos_y, pos_z, cod_tetra_cono). No se ha utilizado la semántica POSITION, pues está ligada a la posición final a la que irá el resultado de salida en el "frame-buffer" y es necesario que siempre se active una posición para cada partícula.

Podría subministrarse información de la transformación sufrida por la partícula, de manera individual o global para todas las partículas, pero hemos considerado que es la propia aplicación en la CPU la que proporciona el movimiento de las mismas, de manera que la posición de las partículas de entrada a la GPU representa su posición real, no siendo necesario ninguna transformación. Pueden considerarse múltiples variantes, por ejemplo que en la propia GPU se produzca el movimiento de cada partícula de forma aleatoria, o bien subministrar información acerca de las fuerzas actuantes para modificar también en la GPU la posición de las mismas.

También se subministra información acerca de la posición de salida en el "frame-buffer" del resultado de la detección de colisión. Se almacena en una variable de tipo vector con semántica POSITION, en la que se guarda la posición (x,y) en el "frame-buffer" donde queremos obtener el resultado de la detección de colisión de la partícula con el poliedro. La coordenada z del vector no es necesaria, si bien debe subministrase una coordenada z de manera que la posición (x,y,z) quede dentro del volumen de visión definido para que se ejecute el programa de vértices almacenado en la GPU para esta partícula.

La posición de salida en el "frame-buffer" subministrada para cada partícula debe multiplicarse por la matriz de modelado y de visión definida por la aplicación en la CPU, para ello se subministra una matriz de transformación bajo la forma de una *variable uniforme*.

---

* La semántica de una variable se utiliza para establecer la correspondencia entre variables no uniformes en la GPU y variables en la CPU, por ejemplo la semántica NORMAL relaciona una variable en la GPU con la variable que representa el vector normal de un vértice en la CPU (véase el Apéndice para más información).

El tamaño del "frame-buffer" limita el número de partículas sobre las que puede aplicarse la detección de colisión. Para un tamaño de 1.024x768 sería de 786.432 partículas.

Información acerca del poliedro

Para la detección de colisión es necesario subministrar información acerca del poliedro en la forma de vértices y tetraedros de su recubrimiento, y del conjunto de tetra-conos que forman el tetra-tree a distintos niveles de profundidad, así como de los tetraedros clasificados en cada tetra-cono. Esta información se almacena en texturas constantes como veremos más adelante.

Por comodidad, el vértice origen del recubrimiento es subministrado a la GPU mediante una *variable uniforme* de tipo vector, codificada su posición mediante las coordenadas (x,y,z), seguido de su valor de inclusión en el poliedro, para el caso en el que la partícula se encuentre situada en dicho punto. También utilizaremos una *variable uniforme* de tipo real para almacenar el número máximo de niveles en el tetra-tree.

En este caso resulta necesario subministrar información acerca de la transformación del poliedro cuando éste se mueve, pues la información del mismo, como hemos dicho, es constante. La matriz de transformación se da mediante una *variable uniforme*.

Veamos el método utilizado en la codificación de la información relativa al poliedro y a su tetra-tree mediante el uso de texturas (Figura 6.3).

En primer lugar, se podría utilizar una textura 1D de vértices, en la que cada componente estaría formada por un vector de reales y representaría la posición de un vértice en el espacio. Como el tamaño de este tipo de texturas está muy limitado, hemos optado por utilizar una textura 2D de las mismas características. En esta textura se almacenan todos los vértices generados en el recubrimiento del poliedro, a excepción del origen del recubrimiento que se almacena en una *variable uniforme* como hemos visto.

Se utiliza otra textura 2D para representar los tetraedros del recubrimiento clasificados en cada tetra-cono de último nivel. Para ello se almacena en cada posición de la textura información de un tetraedro mediante los 3 índices, que representan los 3 vértices exteriores almacenados en la textura anterior. Como la textura es 2D, y se almacena un solo índice por vértice, y éste representa la posición del vértice en una textura 1D, al acceder a ella hay que convertir este índice al formato (fila,columna). Estos tres vértices junto con el origen del recubrimiento

representan el tetraedro. Los índices se almacenan en las componentes (x,y,z) de cada posición de la textura, guardándose además en la cuarta componente el signo del tetraedro calculado previamente. Cada fila de la textura representa a los tetraedros del recubrimiento clasificados en un tetra-cono de último nivel, es decir, para conocer los tetraedros que pertenecen a un tetra-cono *i* es necesario obtener los tetraedros de la fila *i* de la textura. La primera columna de cada fila se ha reservado para almacenar información acerca del número de tetraedros clasificados en dicho tetra-cono. De esta forma quedan posiciones libres en cada fila. Existe una limitación en el número de tetraedros clasificados por tetra-cono y en el número de tetra-conos, impuesta por el tamaño de la textura. Esta representación facilitará enormemente el acceso a los tetraedros del recubrimiento de un tetra-cono.

Debido a que el tamaño de las texturas debe ser una potencia de 2 (número de filas o columnas), podremos almacenar un poliedro cuyo recubrimiento esté formado por un máximo de 262.144 vértices (256 x 1.024) y por un máximo de 2.048 tetra-conos de último nivel, es decir 5 niveles de profundidad (8 x 4^4 tetra-conos). El número de tetraedros clasificados permitidos en cada tetra-cono será de 511 como máximo.

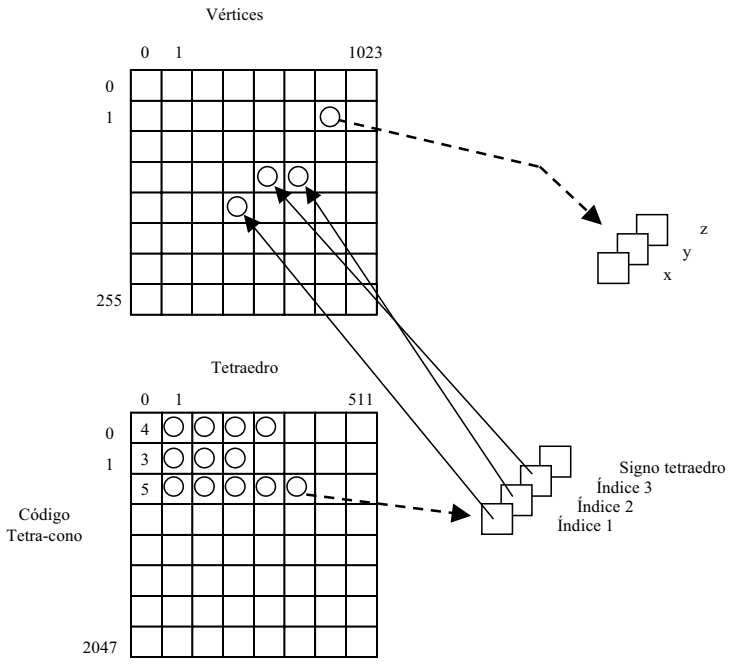


Figura 6.3: Representación de la geometría de un poliedro utilizando texturas.

En cuanto a la información acerca de la representación del tetra-cono, se ha guardado de forma similar a la anterior (Figura 6.4). En primer lugar se utiliza una textura 2D para representar los vértices del tetraedro envolvente de cada tetra-cono. No se almacena el vértice origen (común a todos los tetra-conos), pues es el mismo vértice que se subministra como *parámetro uniforme* y que representa al origen del recubrimiento. En cuanto a la textura 2D de índices, que representa a los tetraedros envolventes y por extensión a los tetra-conos, en cada componente se almacenan tres índices que son una referencia a los vértices almacenados en la textura anterior. En cada fila de la textura se almacena un nivel de profundidad en el tetra-tree, es decir, el conjunto ordenado de tetra-conos de un nivel determinado.

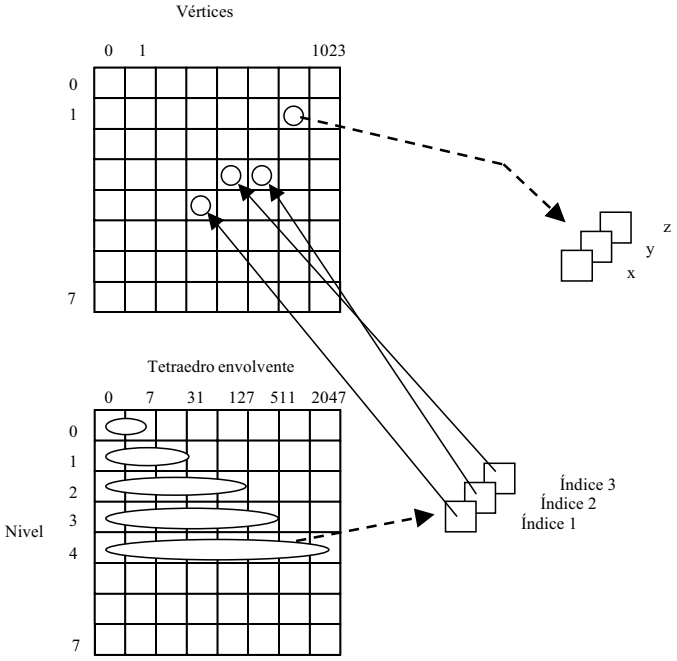


Figura 6.4: Representación del tetra-tree asociado a un poliedro utilizando texturas.

En cada nivel habrá un número fijo de tetra-conos, distinto del número de tetra-conos del nivel anterior, por lo que habrá posiciones no utilizadas en cada fila, menor cuanto mayor sea el nivel de profundidad del tetra-tree o fila en la textura. Nuevamente tendremos restricciones en cuanto al número de niveles permitidos, impuesta por el tamaño de la textura utilizada. Debido a que el tamaño de las texturas debe ser una potencia de 2 (número de filas o columnas), para almacenar 5 niveles es necesaria una textura de 8 filas y 2.048 columnas. Por tanto, en esta implementación, el nivel máximo de profundidad permitido será de 5. El número de

tetra-conos es de 2.728 = 2.048+512+128+32+8, por tanto son necesarios 8.184 vértices (2.728 x 3) para representar los tetra-conos en el caso de incluir vértices repetidos, por lo que se utiliza una textura de 8 filas y 1.024 columnas (8.224 vértices).

Pasar de un nivel a otro en esta representación se realiza mediante una sencilla fórmula, a través de la cual se conoce qué tetra-conos de un nivel corresponden a un tetra-cono de un nivel superior (qué hijos corresponden a un tetra-cono padre):

[Codigo tetra-cono hijo] = 4 x [código tetra-cono padre] + [número de hijo]

Este código de tetra-cono es el que se utiliza para acceder a la fila correspondiente de la textura que representa a los tetraedros del recubrimiento de cada tetra-cono.

Almacenar los datos en las distintas texturas de la forma señalada nos proporciona un alto grado de localidad en los datos, pues el acceso a las texturas con información de índices se suele hacer por filas y secuencialmente, al igual que el acceso a las texturas con la información de vértices, pues hemos almacenado consecutivamente los vértices de un mismo tetraedro aunque se haya duplicado información.

Información de salida

En el píxel correspondiente a cada partícula obtenemos si se ha producido colisión, así como el código de tetra-cono en el que se encuentra la partícula, codificado en un color (r,g,b,a) con semántica COLOR, que es la asociada a la información de salida en el "frame-buffer". Como cada componente de color es un valor entre 0.0 y 1.0, y se utilizan 8 bits para su representación, hay que codificar el tetra-cono. Convenimos en que la primera componente representa si se ha producido o no colisión (valor 1 ó 0 respectivamente) y las compenentes (g,b) representan el código de tetra-cono módulo 256, y el código de tetra-cono resto de la división entera entre 256 respectivamente (esta información es devuelta a la CPU mediante una representación en el intervalo [0.0-1.0] con 8 bits de precisión).

Información entre etapas

Entre las dos etapas programables del pipeline gráfico (del procesador de vértices al procesador de fragmentos) se subministrará la siguiente información:

- Coordenadas de salida en el "frame-buffer" en un vector con semántica POSITION, de forma similar a la información de entrada con semántica POSITION.

- Coordenadas de la partícula, junto con el nuevo tetra-cono calculado, almacenado en un vector con semántica TEXCOORDo, de forma similar a la información de entrada con semántica NORMAL. Es necesario enviar la información acerca de los vértices (en nuestro caso partículas) al procesador de fragmentos, pues éste no es capaz de obtenerla del mismo modo que lo hace el procesador de vértices.

- Información sobre la inclusión del punto en el tetraedro envolvente asociado al tetra-cono en el que se encuentra. Esta información se guardará en la primera componente de un vector con semántica TEXCOORD1, representando el valor 0 la inclusión en el tetra-cono y el valor 1 la no inclusión.

Ha sido necesario incluir esta nueva variable con semántica TEXCOORD1 a pesar de quedar posiciones libres en el vector con semántica POSITION, pues esta información no es una información de entrada al programa de fragmentos, sino que sirve simplemente para activar la ejecución del programa almacenado en dicho procesador, una vez para cada partícula.

En la siguiente figura podemos resumir el modelo de comunicación global entre etapas programables de la GPU y la CPU (Figura 6.5).

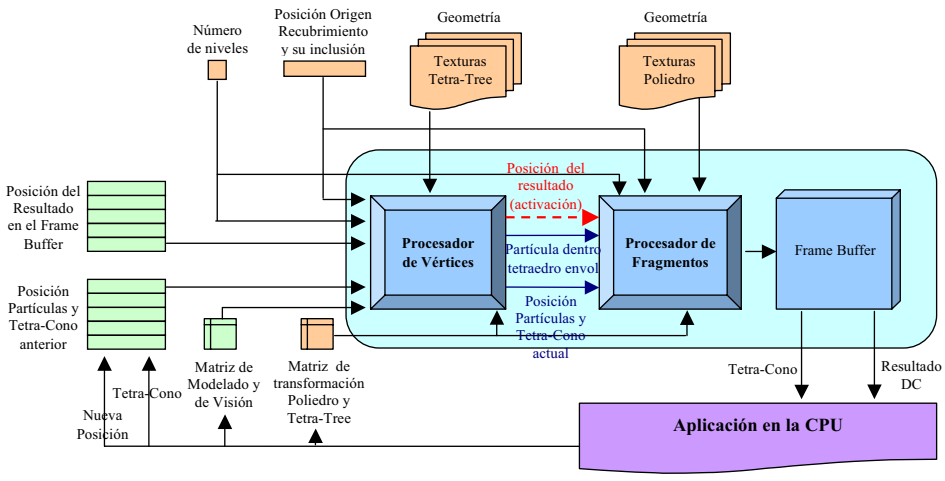


Figura 6.5: Esquema global de información que circula entre etapas programables de la GPU y la CPU.

6.2.3. Etapa 1: Clasificación en el Tetra-Tree

Para determinar si se produce o no colisión entre una partícula y un poliedro, en esta etapa se obtiene el tetra-cono en el que se encuentra dicha partícula y si se encuentra en el tetraedro envolvente de ese tetra-cono. Esta información se pasa a la segunda etapa.

La etapa de clasificación, implementada en el procesador de vértices (Algoritmo 6.1), comprueba en primer lugar si la partícula se encuentra en el tetra-cono y en el tetraedro envolvente del frame anterior. Si es así se envía esta información a la siguiente etapa. Si no es así, la partícula se clasifica en los tetra-conos de primer nivel.

```
#define MAX_COL_VERTICES 1024

#define getVertex(tetracono, nivel) { \
   v = texRECT( indexTT, float2( tetracono, nivel )).xyz; \
   v_fil = v / MAX_COL_VERTICES; \
   v_col = v % MAX_COL_VERTICES; \
   v1 = texRECT( vertexTT, float2( v_col.x, v_fil.x )).xyz; \
   v2 = texRECT( vertexTT, float2( v_col.y, v_fil.y )).xyz; \
   v3 = texRECT( vertexTT, float2( v_col.z, v_fil.z )).xyz; \
   v1 = mul(transf,float4(v1.xyz,1)).xyz; \
   v2 = mul(transf,float4(v2.xyz,1)).xyz; \
   v3 = mul(transf,float4(v3.xyz,1)).xyz; \
}

int is_in_tetra_cone ( float3 v0, float3 v1, float3 v2, float3 v3, float3 particle ) {
   int s1, s2, s3;
   int s0;
   float3 v0_particle, v1_particle, v2_particle, v3_particle;

   v0_particle = v0 - particle;
   v1_particle = v1 - particle;
   v2_particle = v2 - particle;
   v3_particle = v3 - particle;

   int result = 0;   // Fuera del tetracono
   s1 = sign( determinant( float3x3( v0_particle, v3_particle, v2_particle )));
   if ( s1>=0 ) {
      s2 = sign( determinant( float3x3( v3_particle, v0_particle, v1_particle )));
      if ( s2>=0 ) {
         s3 = sign( determinant( float3x3( v0_particle, v2_particle, v1_particle )));
         if ( s3>=0 ) {
            s0 = sign( determinant( float3x3( v1_particle, v2_particle, v3_particle )));
            if ( s0 < 0 ) {
               result = 2; // Está fuera del tetraedro envolvente pero en el tetracono
            } else
               result = 1; // Está dentro del tetraedro envolvente y del tetracono
         }
      }
   }
   return result;
}
```

```
void main (
      float3 position : POSITION,    // (x,y,0) coords de salida "frame-buffer"
      float4 particle : NORMAL,      // (x,y,z,tetracono) coords particula y tetracono anter.

      uniform float4x4 ModelViewProj,// matriz de modelado y visión
      uniform float4 v0,             // origen de todos los tetraedros y su inclusión
      uniform float level,           // Nivel de profundidad
      uniform float4x4 transf,       // Transformación del Poliedro y del Tetra-Tree

      const samplerRECT indexTT : texunit3,  // Indices vértices tetraconos (iv1,iv2,iv3)
      const samplerRECT vertexTT : texunit4,// Coordenadas de los vertices (x,y,z)

      out float4 oPosition  : POSITION,// Coords salida "frame-buffer" (x,y,0,1)
      out float4 oParticle : TEXCOORD0,// Coords Particula y tetracono (x,y,z,tetracono)
      out float4 oInfo : TEXCOORD1 )   // (x,0,0,0), x indica dentro o fuera tetraedro envol
{
   int tetracono, tcono, lev, i, is_in;
   float3 v1, v2, v3;
   int3 v, v_fil, v_col;

   oPosition = mul(ModelViewProj,float4(position.xy,0,1));
   oInfo.w = 0;
   tetracono = particle.w;
   v0 = mul(transf,float4(v0.xyz,1));

   getVertex( tetracono, level );
   is_in = is_in_tetra_cone( v0.xyz, v1, v2, v3, particle.xyz );
   if ( is_in == 0 ) {
      for (i=0;i<8;i++) {
         getVertex( i, 0 );
         is_in = is_in_tetra_cone( v0.xyz, v1, v2, v3, particle.xyz );
         if ( is_in >=1 ) {
            tetracono = i;
            break;
         }
      }
      for (lev = 1; lev<=level; lev++) {
         for (i=0;i<4;i++) {
            tcono = 4 * tetracono + i ;
            getVertex( tcono, lev );
            is_in = is_in_tetra_cone( v0.xyz, v1, v2, v3, particle.xyz );
            if ( is_in >= 1 ) {
               tetracono = tcono;
               break;
            }
         }
      }
   }

   if (is_in == 2) oInfo.x = 1; // desestimar la particular, fuera del tetraedro envolvente
   oParticle = float4( particle.xyz, tetracono );
}
```

Algoritmo 6.1: Programa de vértices para la etapa 1.

A continuación se clasifica en los tetraedros hijos o de segundo nivel en relación al tetra-cono de primer nivel en el que se encontraba. La partícula se clasifica de esta forma por el tetra-tree hasta llegar al tetra-cono de último nivel. En cualquier nivel, si la partícula no se encuentra en el tetraedro envolvente, es descartada para que no pase a la siguiente etapa, marcándose como tal en la correspondiente variable de salida. Estas partículas podrían ser descartadas simplemente aplicándoles una transformación que las dejase, por ejemplo, detrás del plano trasero de recorte, con lo que el propio pipeline de visualización excluiría la partícula para procesos posteriores. Pero de esta forma no obtendríamos como salida el tetra-cono en el que se encuentra la partícula para el siguiente frame.

En esta etapa no se han incluido las esferas envolventes a nivel de tetra-cono. Bastaría con añadir la información de las esferas envolventes codificada en textura y comprobar previamente si la partícula se encuentra en la esfera envolvente asociada al tetra-cono.

6.2.4. Etapa 2: Inclusión en un Tetra-Cono

Esta etapa se ha implementado en el procesador de fragmentos (Algoritmo 6.2). Cada partícula que ha pasado la primera etapa debe clasificarse respecto a los tetraedros del recubrimiento del tetra-cono obtenido. Para ello se obtiene de la etapa anterior la información del tetra-cono en el que se encuentra una partícula y si está o no incluida en el tetraedro envolvente. Si no está incluida en el tetraedro envolvente, se escribe en el "frame-buffer" que no se ha producido colisión y el código de tetra-cono en el que se encuentra dicha partícula para el siguiente frame. En otro caso se comprueba la inclusión de la partícula en los tetraedros del recubrimiento del correspondiente tetra-cono para determinar si se produce o no la inclusión en el poliedro. Como vimos en el Algoritmo 5.1 y 5.2, es necesario implementar sendos conjuntos de características visitadas positivas y negativas, pero actualmente esto no es posible debido a las limitaciones en cuanto a las estructuras de datos permitidas por la propia GPU y el lenguaje Cg. Para solucionar este problema se devuelve un valor de ½ como código de colisión de esa partícula, indicando que el programa residente en la CPU, que obtiene y maneja el resultado de la colisión calculada en la GPU, debe calcular la inclusión en el tetra-cono obtenido desde la GPU. La probabilidad de que esto ocurra es muy baja, por lo que en la mayor parte de los casos no es necesario realizar estos cálculos.

A diferencia del algoritmo de detección de colisión mostrado en el capítulo anterior (Algoritmo 5.2), este algoritmo no hace uso de la coherencia temporal expresada en el Lema 5.1 pues sería necesario guardar información entre frames acerca de la máscara de bits que representa el signo de la coordenada baricéntrica α, y esto no es posible por el momento, debido de nuevo a las limitaciones impuestas por la GPU.

```
void main (
      float4 particle : TEXCOORD0,  // (x,y,z,tetracono) coords particula y tetracono
      float4 info : TEXCOORD1,      // (x,0,0,0), x indica dentro o fuera tetraedro envol.

      uniform float4 v0,            // origen de todos los tetraedros, inclusión
      uniform float level,          // Nivel de profundidad
      uniform float4x4 transf,      // Transformación del Poliedro y del Tetra-Tree

      const samplerRECT index: texunit1,    // Indices vertices tetraedros (i1,i2,i3,signo)
      const samplerRECT vertex: texunit2,   // Coordenadas de los vertices (x,y,z)

      out float4 oResult  : COLOR)  // (r,g,b,0) r=resultado DC, gb=codigo tetracono
{
   int tetracono = particle.w;
   oResult = float4( 0, (tetracono / 256) / 256.0, (tetracono % 256) / 256.0, 0 );
   if (info.x==0) {
      int4 index_v;     //indices de v1,v2,v3
      float3 v[4];
      float3 v0_particle, v1_particle, v2_particle, v3_particle;
      int signo, pos_x, suma;
      int s0, s1, s2, s3;
      float numTetra = texRECT( index, float2( 0, tetracono )).r;  // (y,x).r
      v[0] = mul(transf,float4(v0.xyz,1)).xyz;
      suma = 0;
      for (int i=1;i<=numTetra;i++) {
         index_v = texRECT( index, float2( i, tetracono ));
         pos_x = index_v.x / 1024;
         index_v = index_v % 1024;
         signo = index_v.w;
         if ( signo == 2 ) signo = -1;
         v0_particle = v[0] - particle.xyz;
         v[1] = texRECT( vertex, float2( index_v.x, pos_x )).rgb;
         v[1] = mul(transf,float4(v[1],1)).xyz;
         v[2] = texRECT( vertex, float2( index_v.y, pos_x )).rgb;
         v[2] = mul(transf,float4(v[2],1)).xyz;
         v[3] = texRECT( vertex, float2( index_v.z, pos_x )).rgb;
         v[3] = mul(transf,float4(v[3],1)).xyz;
         v1_particle = v[1] - particle.xyz;
         v2_particle = v[2] - particle.xyz;
         v3_particle = v[3] - particle.xyz;
         float b0 = determinant( float3x3( v1_particle, v2_particle, v3_particle ));
         s0 = signo * sign(b0);
         if ( s0 >= 0 ) {
            float b1 = determinant( float3x3( v0_particle, v3_particle, v2_particle ));
            s1 = signo * sign(b1);
            if ( s1 >= 0 ) {
               float b2 = determinant( float3x3( v3_particle, v0_particle, v1_particle ));
               s2 = signo * sign(b2);
               if ( s2 >= 0 ) {
                  float b3 = determinant( float3x3( v0_particle, v2_particle, v1_particle ));
                  s3 = signo * sign(b3);
                  if ( s3 >= 0 ) {
                     int vert = -1;
                     if ( s1 > 0 && s2 > 0 && s3 > 0 ) // interior de v0v1v2v3, cara v1v2v3
                        suma += 2 * signo;
                     else if ( s0==0 && s1==0 ) { // vért. v2, v3, arista v2v3
                        oResult.r = 1;
                        break;
                     }
                     else if ( s1>0 && s2==0 && s3==0 )           // vért. v1, arista v0v1
```

```
                    vert = 1;
                 else if ( s0>0 && s1==0 && s2>0 && s3==0 )    // arista v0v2
                    vert = 2;
                 else if ( s0>0 && s1==0 && s2==0 && s3>0 )    // arista v0v3
                    vert = 3;
                 else if ( s0>0 && s1==0 && s2==0 && s3==0 ) { // vért. v0
                    oResult.r = v0.w;
                    break;
                 }
                 else suma += signo;     // Cara v3v2v0, v3v0v1, v0v1v2, arista v1v2, v2v1
                 if ( vert != -1 ) {     // Casos especiales, se tratan fuera de la GPU
                    oResult.r = 0.5;
                    suma = 0;
                    break;
                 }
              }
           }
        }
     }
     if ( suma==2 ) oResult.r = 1;
  }
}
```

Algoritmo 6.2: Programa de fragmentos para la etapa 2.

6.3. Conclusiones

En esta sección, a modo de conclusión, describiremos las posibilidades de mejora de estos programas, así como algunos problemas que presenta esta implementación. Finalmente realizaremos un estudio temporal que muestre la eficiencia de la implementación realizada.

6.3.1. Algunos problemas

Hemos observado que se produce un problema de sincronización en el acceso a texturas entre el procesador de fragmentos y el de vértices. Por una parte, el procesador de fragmentos debe esperar a que el procesador de vértices envíe información, y por otra, no se aprovecha el alto grado de paralelismo del procesador de fragmentos, normalmente con mayor número de unidades de procesamiento que el procesador de vértices. En el caso considerado en este capítulo hay una ejecución del procesador de fragmentos por cada ejecución del procesador de vértices.

Por otra parte, al intentar ambos procesadores acceder de forma simultánea a información sobre texturas, se produce un problema de sincronización que ralentiza el proceso global.

6.3.2. Posibles mejoras

A continuación mostramos algunas de las mejoras que pueden realizarse en la implementación de este algoritmo, así como posibles alternativas al problema propuesto:

- Es posible modificar la posición de las partículas dentro de la GPU sin necesidad de obtener su posición desde la CPU. En el caso de un movimiento aleatorio de las mismas, cada posición sería variada mediante la generación de un número aleatorio. Si todas las partículas tienen un movimiento simultáneo simplemente se enviaría información sobre su transformación global. En el caso de movimiento independiente, podría subministrarse por ejemplo información de la fuerza actuante sobre cada partícula. Además es posible hacer que el movimiento de la partícula dependa de la colisión producida. Para todos los casos es necesario enviar información sobre la posición de las partículas en el primer frame y desactivarlo para los siguientes frames. La nueva posición puede escribirse en una textura de flotantes de forma que sirva de realimentación para el siguiente frame.

- En el caso de que haya algún tipo de dependencia entre partículas, no es posible realizar operaciones en base a los cálculos realizados a otras partículas hasta que no haya finalizado el procesamiento de todas ellas, y siempre mediante el acceso al "frame-buffer" o a alguna textura con la información que necesitan.

- El tetra-cono resultado de la detección de colisión puede almacenarse también en una textura para servir de realimentación a la siguiente ejecución para el conjunto de partículas.

- Sería posible optimizar la representación de la geometría de los objetos de manera que se obtenga una mayor localidad en el acceso a la información.

6.3.3. Estudio de tiempos

Son múltiples las alternativas de implementación que pueden realizarse del problema de DC planteado. Hemos optado por mostrar tres implementaciones para la DC entre una nube de partículas y un poliedro que tratan de minimizar los inconvenientes planteados en la Sección 6.3.1. Éstas son:

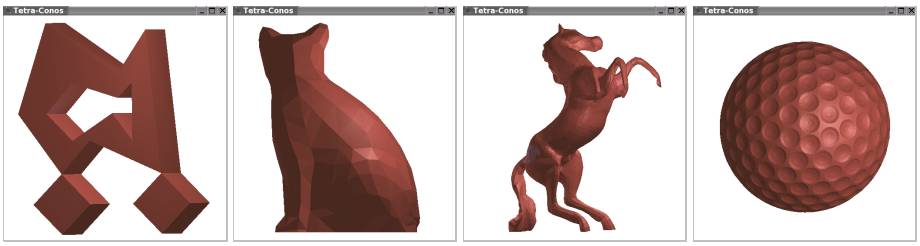
- **Implementación FS**: Esta implementación realiza las dos etapas mostradas en la Sección 6.2 en una solo programa incluido en el procesador de fragmentos (Fragment Shader). Esto es posible gracias a la utilización de una tarjeta gráfica de última generación (NVidia GeForce 6600) y el perfil Shader Model 3.0, que permite la utilización de ciclos y programas con mayor número de instrucciones.

- **Implementación CPU+FS**: Se realiza la primera etapa en la CPU y la segunda en el procesador de Fragmentos.

- **Implementación CPU**: Se realizan las operaciones descritas en las dos etapas exclusivamente en la CPU, sin utilizar el procesador gráfico.

Para las pruebas se ha codificado en textura tanto la geometría como los tetra-trees de los poliedros mostrados en la Figura 6.6. Hemos incorporado un nuevo poliedro con mayor número de vértices que los utilizados en el capítulo anterior, y mostrar así la eficiencia de la implementación para objetos de tamaño considerable.

Se ha generado un numero determinado de partículas, cada una de ellas en una posición aleatoria y dentro de la caja envolvente del poliedro correspondiente. Estas partículas se mueven también aleatoriamente dentro de dicha caja.



a) 32 vértices, 18 caras. b) 382 vértices, 760 caras. c) 3.582 vértices, 7.172 caras d) 23.363 vértices, 46.205 caras

Figura 6.6: Poliedros considerados para las pruebas. a) Poliedro Complejo I. b) Gato. c) Caballo. d) Golf.

Las pruebas realizadas han sido las siguientes:

- Obtención del tiempo de detección de colisión en frames/seg. para los distintos tipos de objetos, todos ellos a una profundidad de tetra-tree fija (2.048 subdivisiones espaciales), y para 1.024 partículas (Figura 6.7).

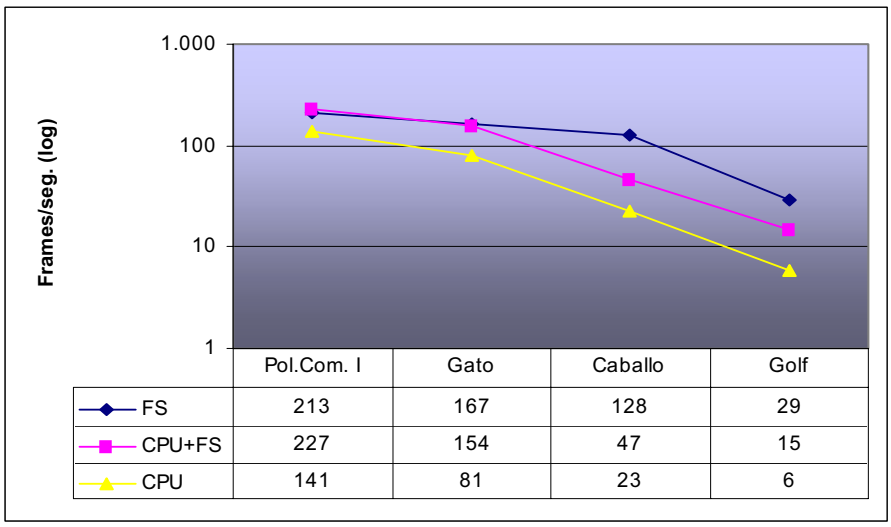
- Obtención del tiempo de detección de colisión en frames/seg. para el poliedro *Gato* y 1.024 partículas. En esta prueba se ha variado el número de subdivisiones del Tetra-Tree (Figura 6.8).

- Obtención del tiempo de detección de colisión en frames/seg. para el poliedro *Caballo* con una profundidad de tetra-tree fija (2.048 subdivisiones espaciales), en el que se ha variado el número de partículas (Figura 6.9).

Según los resultados obtenidos podemos ver que el algoritmo implementado en el Procesador de Vértices obtiene mejores resultados, seguido del algoritmo híbrido implementado entre la CPU y el Procesador de Fragmentos.

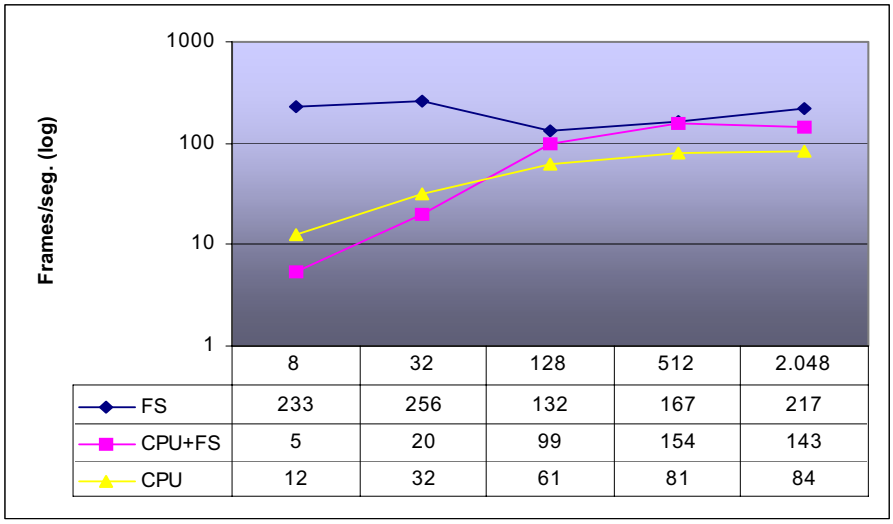
Según podemos ver en la Figura 6.8, con poca profundidad en el tetra-tree el algoritmo FS obtiene muy buenos resultados, mejores que a profundidades mayores. Esto es debido al coste que le supone al algoritmo acceder varias veces a posiciones con baja localidad espacial en las texturas, sobre todo cuando hay una alta profundidad en el tetra-tree.

Apreciamos también un escalonamiento en el gráfico que representa la variación del número de partículas (Figura 6.9) para el algoritmo FS. Esto puede deberse a la forma de enviar la información desde la CPU a la GPU, que se realiza en bloques, siendo uno de los cuellos de botella de este tipo de implementación. Por este motivo tarda aproximadamente lo mismo en realizarse la DC para 256 partículas que para 1.024, pues en los dos casos la cantidad de información enviada a la memoria de la GPU se realiza en el mismo número de bloques.



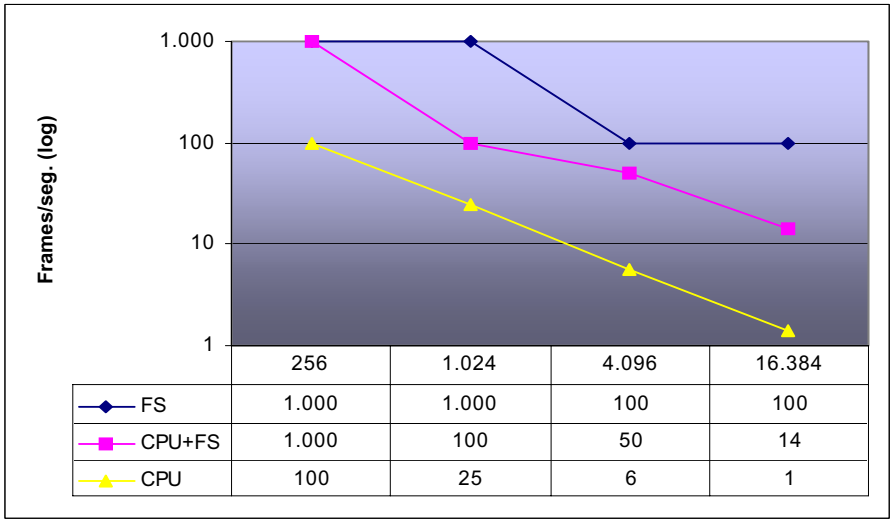
	Pol.Com. I	Gato	Caballo	Golf
FS	213	167	128	29
CPU+FS	227	154	47	15
CPU	141	81	23	6

Figura 6.7: DC entre nube de 1.024 partículas y distintos tipos de poliedros. En el eje X se nuestra el poliedro, y en el eje Y el número de frames por segundo obtenidos con escala logarítmica. El número de subdivisiones se ha fijado en 2.048.



	8	32	128	512	2.048
FS	233	256	132	167	217
CPU+FS	5	20	99	154	143
CPU	12	32	61	81	84

Figura 6.8: DC entre una nube de 1.024 partículas y el poliedro *Gato*, en el que se ha variado la profundidad de su tetra-tree. En el eje X se nuestra el número de subdivisiones realizadas, y en el eje Y el número de frames por segundo obtenidos con escala logarítmica.



	256	1.024	4.096	16.384
FS	1.000	1.000	100	100
CPU+FS	1.000	100	50	14
CPU	100	25	6	1

Figura 6.9: DC entre una nube de partículas y el poliedro *Caballo*, en el que se ha variado el número de partículas. En el eje X se nuestra el número de partículas, y en el eje Y el número de frames por segundo obtenidos con escala logarítmica. El número de subdivisiones se ha fijado en 2.048.

CAPÍTULO

7

Principales Aportaciones y Futuros Trabajos

En este capítulo, a modo de conclusión, repasamos las aportaciones realizadas en esta memoria así como las posibles ampliaciones y trabajos por desarrollar.

7. Principales Aportaciones y Futuros Trabajos

A lo largo de esta memoria se han establecido una serie de principios y técnicas para el desarrollo de métodos de detección de colisión entre diversos tipos de objetos, tanto en 2D como en 3D. Estos métodos, basados en una representación mediante recubrimientos simpliciales, son adecuados tanto para objetos poliédricos complejos* como para objetos definidos por mallas de triángulos, en cuyo caso los algoritmos han sido optimizados. Esto supone un avance en el sentido de que no es necesario descomponer un objeto en piezas convexas o realizar triangulaciones como realizan la mayor parte de los métodos habituales.

Los algoritmos desarrollados son sencillos, más eficientes y robustos que otros, en el sentido de que las operaciones realizadas son operaciones entre símplices, para lo que se utilizan las coordenadas baricéntricas. De ésta forma se minimizan los casos especiales a tratar y se proporciona robustez al considerar el signo de las coordenadas baricéntricas con una determinada tolerancia o error.

Se ha utilizado la coherencia temporal, en el sentido de que se aprovechan cálculos entre frames; así como la coherencia espacial, entendiendo como tal la utilización de una nueva estructura de datos de descomposición espacial denominada tetra-tree†. Los tetraedros del recubrimiento de los objetos son clasificados en sus respectivos tetra-trees de manera que no es necesario tratar con toda la complejidad de los objetos para determinar si están o no en colisión.

---

* Nos referimos tanto a polígonos como poliedros, pero en este capítulo utilizaremos el término 3D genérico.

† Nos referimos también al término en 2D tri-tree y a sus elementos constituyentes como tri-conos, triángulos y circunferencias envolventes.

Esta nueva estructura de datos es adecuada para la detección de colisión pues se ajusta bien a la forma de los objetos y permite localizar fácilmente las partes del objeto en colisión, siendo una estructura que necesita poco espacio de almacenamiento y más rápida de calcular que otras, como puede ser un octree. Además creemos que puede ser utilizada para otro tipo de aplicaciones de forma efectiva, como es el caso de modelado de objetos, modelado a distinto nivel de detalle, terrenos, objetos deformables, etc.

Los algoritmos desarrollados permiten la detección de colisión booleana* entre objetos. Si se desea obtener la determinación de colisión, entendiendo como tal la obtención de las partes de los objetos implicadas en la colisión, estos algoritmos pueden modificarse para que no se detengan de forma inmediata y almacenar los pares de características de los objetos que intersecan. En cuanto a la respuesta a la colisión, es una línea de investigación que queda abierta, sobre todo en lo relacionado a la deformación de los objetos, para lo que pensamos que un tetra-tree puede ser adecuado al permitir delimitar la zona de respuesta de manera sencilla. Todos los algoritmos desarrollados pueden utilizarse para la detección de colisión con una determinada tolerancia o error, es decir, la obtención de si se produce o no colisión cuando los objetos se encuentra a una distancia menor que un valor de tolerancia, utilizando para ello el concepto de área de influencia.

Entre los algoritmos desarrollados podemos destacar los nuevos algoritmos de detección de colisión para objetos de distinta naturaleza además de para polígonos no convexos, como un algoritmo punto/polígono específico para polígonos convexos y otro para polígonos con recubrimiento positivo. Estos algoritmos específicos obtienen un tiempo de detección de colisión casi constante gracias a la coherencia temporal. Además se han desarrollado nuevos algoritmos para la detección de colisión entre una circunferencia y un polígono y para la detección de colisión entre polígonos.

En el caso 3D se han desarrollado algoritmos para la detección de colisión de puntos, esferas y poliedros en relación a poliedros. El algoritmo *punto/poliedro* y el algoritmo *esfera/poliedro* puede ser utilizado en la detección de colisión entre sistemas de partículas o para la detección de colisión entre poliedros, para lo cual se situarían partículas de manera estratégica sobre los poliedros y serviría como método previo a la detección de colisión detallada.

* La determinación de si se produce o no colisión entre objetos.

Ligado al desarrollo de éstos algoritmos ha sido necesario implementar algoritmos más sencillos que operan sobre símplices. Se ha optado por el desarrollo de nuevos algoritmos basados en coordenadas baricéntricas, sobre todo por eficiencia y por homogeneidad con las operaciones realizadas a lo largo de la memoria. Se han realizado implementaciones alternativas utilizando otros algoritmos clásicos que se han mostrado menos eficientes y robustas que las realizadas mediante coordenadas baricéntricas, por lo que hemos utilizado los nuevos algoritmos desarrollados. Ejemplos de estos nuevos algoritmos son los algoritmos de intersección entre segmentos o entre triángulos en 2D, o los algoritmos de intersección entre segmento y tetra-cono, segmento y triángulo, segmento y tetraedro, o tetraedro y tetraedro en 3D.

También se ha estudiado el comportamiento de éstos algoritmos de detección de colisión para circunstancias específicas, implementando algoritmos más adecuados para algunas de éstas situaciones, como por ejemplo algoritmos más rápidos cuando los objetos tienen tamaños muy dispares, o para cuando tienen tamaños similares.

Para mostrar su eficiencia estos algoritmos han sido implementados y se ha realizado un completo estudio temporal para diversos de los parámetros relacionados, tanto con la detección de colisión en sí (trayectorias de los objetos, etc.), como con la construcción de los tetra-trees correspondientes. Para todos los casos se han obtenido resultados más que satisfactorios, incluyendo una detección de colisión en tiempo real para poliedros con gran número de vértices.

La implementación de estos algoritmos utilizando una filosofía orientada a objetos nos ha posibilitado un marco de trabajo para el desarrollo de nuevos algoritmos para la detección de colisión, así como para la comparación de nuevos algoritmos y la mejora de los ya existentes. Hemos preferido mostrar en esta memoria los algoritmos de una manera clara a realizar una optimización que obscurezca la comprensión de los mismos. No obstante, tanto los algoritmos como la implementación realizada son susceptibles de mejorarse, de manera que queda abierta esta posibilidad como futuro trabajo a realizar.

Finalmente, como puede verse en el Capítulo 6 dedicado a la detección de colisiones utilizando el hardware gráfico programable, es posible adaptar estos algoritmos para que puedan utilizar toda la potencia de las tarjetas gráficas de última generación, mediante una programación de propósito general. Se ha implementado un algoritmo de detección de colisión entre una nube de partículas y un poliedro en el que se obtienen resultados más que notables en relación a una implementación clásica realizada en la CPU.

Además de las posibles líneas de trabajo que quedan abiertas y las futuras ampliaciones descritas en párrafos anteriores, hay algunas posibles líneas de trabajo futuro que consideramos importantes enumerar de forma resumida, como son:

- Mejora general de los algoritmos desarrollados.

- Desarrollo de un algoritmo de ajuste de tetraedros envolventes más eficiente en cuanto a su velocidad y al ajuste realizado.

- Utilización de Tetra-Trees para otro tipo de aplicaciones como modelado geométrico, niveles de detalle, terrenos, ray-tracing, objetos deformables, etc.

- Desarrollo de nuevos algoritmos basados en los anteriores, como un algoritmo de detección de colisión entre poliedros utilizando partículas situadas en lugares estratégicos.

- Desarrollo de algoritmos para la respuesta a colisión. Entre éstos debemos destacar algoritmos de deformación y de actualización de los tetra-trees.

- Adaptación de los algoritmos y estructuras de datos para su utilización en entornos de realidad virtual tales como VRML y su posterior visualización en sistemas estéreo.

- En cuanto a la implementación en la GPU consideramos necesario la mejora en la codificación de la información por medio de texturas para obtener una mejor localidad espacial en el acceso a los datos.

- Otra futura línea de trabajo que amplia la anterior consiste en la implementación y optimización de los algoritmos planteados, tanto para la GPU como para dispositivos móviles, donde pueden ser especialmente adecuados los algoritmos y las estructuras de datos expuestas, debido sobre todo a su sencillez y a la necesidad de pocos recursos de memoria, limitación importante en tales dispositivos.

- Por último, pensamos que es importante la implementación de estos métodos en aplicaciones de tiempo real, donde se muestre su potencia y robustez.

Apéndice

A continuación presentamos algunas de las características fundamentales de las tarjetas gráficas programables. Esto es necesario para una correcta utilización de la GPU en la resolución de problemas de propósito general, distintos de la visualización, para lo que presentamos un conjunto de técnicas y principios generales. Por último repasamos brevemente el lenguaje Cg, utilizado en la programación de la GPU.

CONTENIDOS:

A. Hardware Gráfico

La posibilidad de utilizar la GPU* como un procesador más en el que se pueden realizar operaciones no dedicadas a la visualización ha sido motivado entre otros factores por su potencia de cálculo y su alto ancho de banda en las transferencias de memoria realizadas en la propia tarjeta. La aparición de la serie *GeForce 6* de *NVIDIA* ha supuesto un gran avance debido a la eliminación de algunas restricciones en la programación de la GPU. Esto unido a la aparición de lenguajes de alto nivel adecuados para la programación de la tarjeta gráfica, como el lenguaje *Cg*, que facilita en gran medida la implementación de ciertas operaciones, nos ha impulsado en cierta forma a adaptar ciertos algoritmos de propósito general para que puedan ser implementados en la GPU. A lo largo de este apéndice estudiaremos las características de la GPU que la hacen interesante para la implementación de problemas distintos al de la visualización.

A.1. Características de la GPU

Una diferencia fundamental entre una GPU y una CPU actual consiste en que los procesadores gráficos tienen una arquitectura vectorial, distinta de la arquitectura de Von Neumann que es en serie. Un procesador vectorial aplica una función a un vector de datos de entrada produciendo un conjunto de resultados en paralelo.

Los datos de entrada son introducidos en el procesador gráfico en forma de flujo de datos o vector, son procesados utilizando algún tipo de función establecida, y se guarda el resultado en memoria gráfica. Cada elemento de ese flujo de entrada es procesado de forma independiente utilizando un cierto grado de paralelismo dependiendo del número de unidades de ejecución que posea el procesador (la GPU).

---

* Unidad de Procesamiento Gráfico.

En esta sección veremos, por una parte el esquema de funcionamiento de la GPU en la forma de pipeline gráfico, y por otra la jerarquía de memoria y su utilización en relación a la CPU [FK03].

A.1.1. Etapas del pipeline

Las GPU's actuales utilizan una secuencia de etapas para el procesamiento en paralelo de instrucciones, comúnmente denominada pipeline. Cada etapa recibe entradas de la etapa anterior y produce salidas para la etapa siguiente. En la Figura A.1 podemos ver el pipeline de las tarjetas gráficas de hoy en día. La entrada de la primera etapa del pipeline suele estar formada por un flujo de vértices que definen la geometría de los objetos. Unidos a cada vértice, además de su posición, se subministra otro tipo de información como el color, una coordenada de textura, o el vector normal.

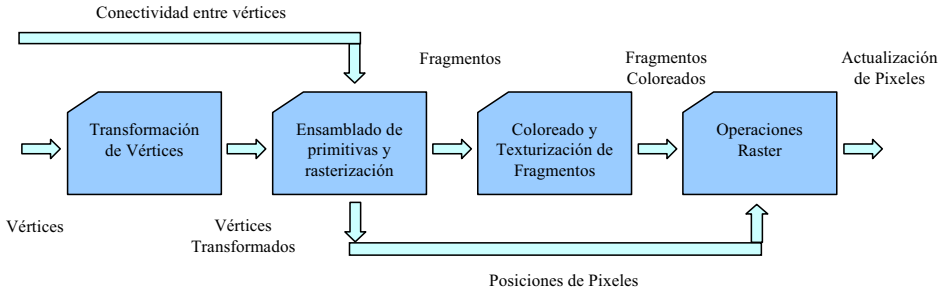


Figura A.1: El pipeline de las tarjetas gráficas.

A continuación resumimos brevemente cada una de las etapas del pipeline gráfico:

- **Transformación de vértices**: realiza una serie de operaciones matemáticas para cada vértice. Entre estas operaciones destacan la transformación de la posición, la generación de coordenadas de textura y la determinación del color del vértice mediante los parámetros de iluminación.

- **Ensamblado de primitivas y rasterización:** los vértices transformados en la etapa anterior son ensamblados en primitivas geométricas, según la información de la primitiva que forma el flujo de vértices. Se obtiene una secuencia de triángulos, segmentos o puntos, que pueden necesitar el recorte con el volumen de visión. Los polígonos que pasan este proceso de recorte son pasterizados, y como consecuencia se obtiene un flujo de posiciones de píxeles y un flujo de fragmentos. La diferencia entre píxel y fragmento consiste en que

un fragmento es un estado potencial para actualizar un píxel al final del proceso. Los fragmentos que pasen una serie de filtros a través del pipeline de la tarjeta actualizarán un determinado píxel en el "frame-buffer". Como consecuencia de esta etapa se obtiene un flujo de fragmentos posiblemente distinto del número de vértices de entrada.

- **Interpolación, aplicación de textura y coloreado:** los parámetros de los fragmentos de la etapa anterior son interpolados mediante una serie de operaciones matemáticas y de textura para determinar el color final del fragmento. Esta etapa puede calcular una nueva profundidad del fragmento o descartarlo, por lo que al final de la etapa puede obtenerse un fragmento o ninguno.

- **Operaciones raster:** realiza un conjunto de operaciones por fragmento justo antes de actualizar el "frame-buffer". Por ejemplo, se eliminan superficies ocultas o se realizan las operaciones de blending. En esta etapa se realizan operaciones que se habilitan desde OpenGL o DirectX, como los test Alpha, Stencil, Dithering, etc.* o también operaciones lógicas. Como resultado, algunos de los fragmentos no actualizarán el "frame-buffer".

Estas etapas que tienen lugar en las tarjetas gráficas actuales pueden verse modificadas en el sentido de que se pueden programar dos etapas adicionales entre ellas, dando lugar al pipeline gráfico programable (Figura A.2). En este pipeline se ha añadido una etapa de procesamiento de vértices y una etapa de procesamiento de fragmentos totalmente programables y configurables por el programador. Veamos a continuación estas nuevas etapas:

- **Procesador de vértices:** en esta etapa se obtiene como entrada los atributos de cada vértice (posición, color, coordenadas de textura, etc.) y son transformados de acuerdo a un conjunto de instrucciones dadas por el programador. Estos atributos alimentan a la siguiente etapa del pipeline. Normalmente las operaciones que se pueden realizar están un tanto limitadas pero optimizadas para tratar con las operaciones más comunes en la visualización, como operaciones entre vectores, operaciones trigonométricas u operaciones especiales para obtener iluminación, simulación de niebla, etc.; en general, son operaciones geométricas. Otra limitación viene impuesta por el número de registros y los formatos de representación utilizados.

---

* Hemos preferido utilizar los términos en inglés, debido a que son comúnmente utilizados y su traducción pude dar lugar a confusión.

- **Procesador de fragmentos:** a partir de fragmentos de una etapa previa se modifican los atributos de los mismos, que pasan a la siguiente etapa. Al igual que antes, el número de operaciones está limitado y suele ser un poco más restrictivo que el permitido para el procesador de vértices, al igual que los formatos de representación, que tienen menor precisión. A diferencia del procesador de vértices, se permite el acceso a texturas por medio de coordenadas de textura, aunque en las tarjetas más modernas se ha habilitado también un acceso a textura un poco más restrictivo por parte del procesador de vértices.

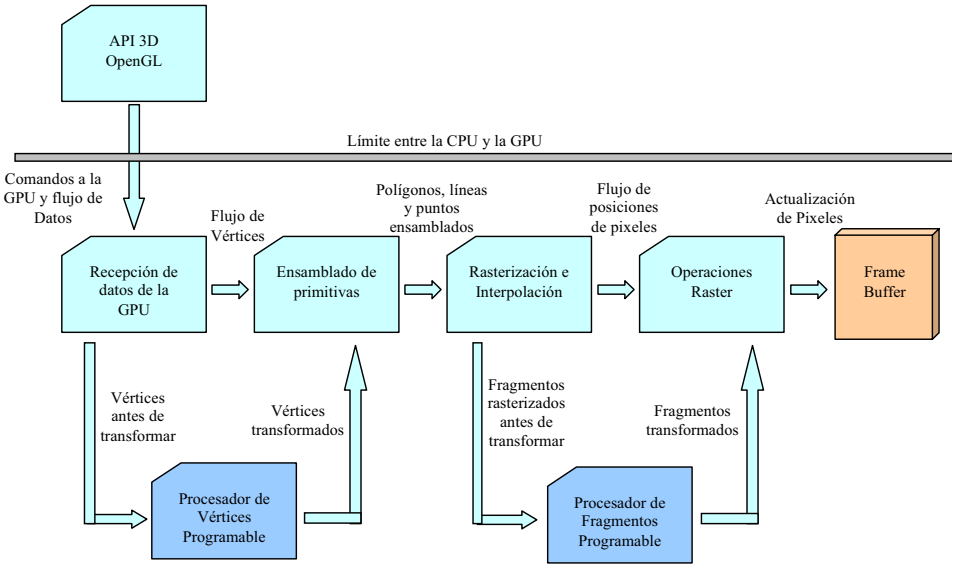


Figura A.2: Pipeline gráfico programable.

A.1.2. El modelo de memoria de la GPU

Un procesador gráfico tiene su propia jerarquía de memoria similar a la de los procesadores secuenciales, que incluye una memoria principal, memoria caché y registros. Sin embargo esta jerarquía de memoria está diseñada para acelerar las operaciones gráficas basadas en el modelo de programación de flujos de elementos. Veamos en líneas generales cómo está estructurado el modelo de memoria en la GPU [Fero5].

En la Figura A.3 podemos ver una representación de este modelo combinado con el modelo de memoria de la CPU. La GPU tiene su propia memoria caché y registros para acelerar el acceso a la información durante la ejecución de un programa. La

GPU tiene su propia memoria principal con su propio espacio de direcciones, de manera que los programadores deben copiar explícitamente los datos en la memoria de la GPU antes de empezar la ejecución de un programa. Esta transferencia de información es un cuello de botella para muchas aplicaciones debido a la asimetría en las transferencias en el bus AGP. Este problema se soluciona parcialmente en arquitecturas con bus PCI-Express.

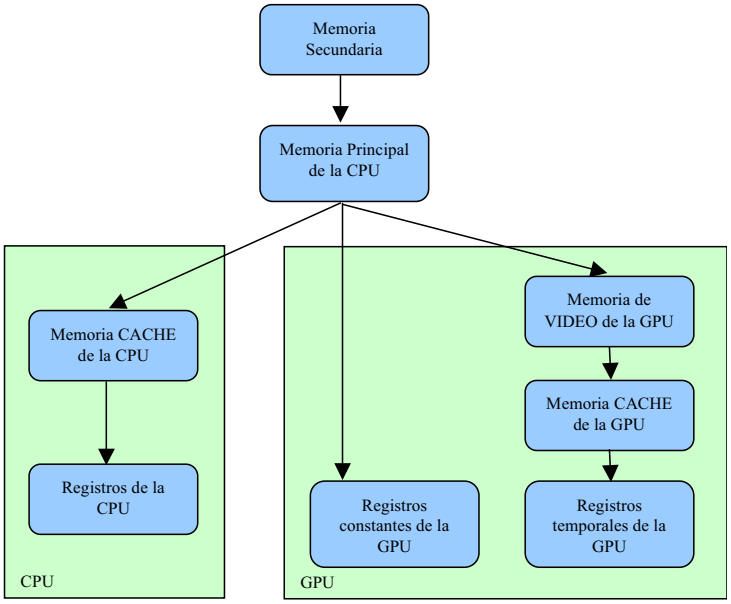


Figura A.3: Jerarquía de memoria de la CPU y GPU.

La utilización de la memoria de la GPU tiene una serie de restricciones de uso. Hay tres tipos de flujos de datos que puede manejar un programador de la GPU, éstos son los flujos de vértices, los flujos de "frame-buffers" y los flujos de texturas (Figura A.4).

El procesador de vértices obtiene información de los vértices (array de vértices) como las posiciones, color, normal, etc. En el perfil* *Vertex Shader 3.0* también se puede leer de las texturas almacenadas en memoria. El procesador de fragmentos obtiene información de fragmentos y de texturas y puede escribir en el "frame-buffer".

* Para aprovechar las características particulares de un conjunto de tarjetas gráficas se definen determinados perfiles para cada conjunto. Cada perfil permite realizar un conjunto de operaciones u otro. Una determinada tarjeta gráfica puede programarse utilizando un subconjunto de perfiles, pero debe especificarse uno sólo para programarla.

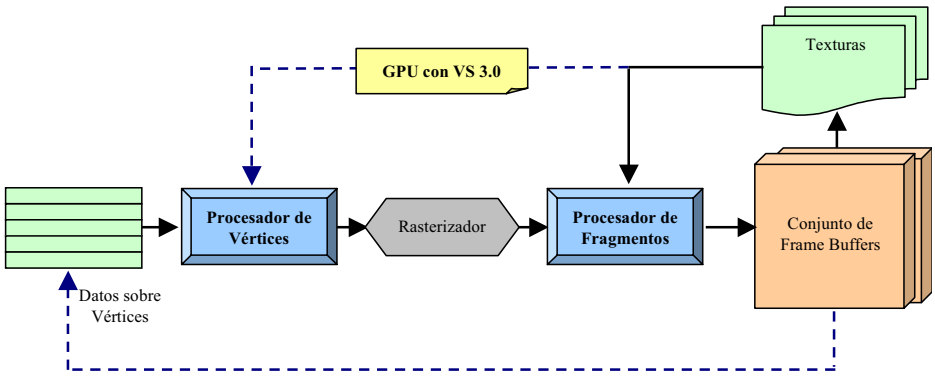


Figura A.4: Flujos de Datos que puede utilizar el programador en las modernas GPU's.

A.1.3. Recursos computacionales en la GPU

Las tarjetas gráficas modernas tienen múltiples procesadores de vértices (las tarjetas *NVIDIA GeForce 6800 Ultra* y la *ATI Radeon XT* tienen seis de ellos). Las primitivas básicas para los cálculos 3D son vectores de cuatro coordenadas (x,y,z,w) y colores (r,g,b,a) por lo que los procesadores de vértices tienen hardware específico para procesar vectores de cuatro componentes.

La mayor parte de los procesadores de vértices sólo son capaces de obtener información del vértice actual del flujo de vértices de entrada, no pudiendo leer información de otros vértices distintos del actual. Las tarjetas *NVIDIA GeForce de la Serie 6* tienen una nueva característica llamada *vertex textura fetch (VTF)* que permite el acceso aleatorio a memoria de manera que se puede acceder a parte de los vértices o a todos almacenándolos previamente en una textura de vértices.

Las GPU's modernas tienen también múltiples procesadores de fragmentos vértices (las tarjetas *NVIDIA GeForce 6800 Ultra* y la *ATI Radeon XT* tienen dieciséis). Los procesadores de vértices pueden leer información de texturas, sin embargo la dirección de salida de un fragmento está determinada antes del procesamiento del fragmento, por lo que no se puede cambiar el píxel al que va dirigido. No pueden por tanto escribir información en memoria.

La imagen generada en la GPU puede enviarse al "frame-buffer" para visualizarla o puede escribirse en una textura en memoria. Éste es el único mecanismo para realimentar directamente la GPU con nuevas entradas sin tener que pasar por la CPU. Debemos tener en cuenta que en la ejecución de un programa de fragmentos en una unidad, se puede leer de textura tantas veces como queramos a lo largo del

programa, pero sólo se escribe al finalizar este programa en todas las unidades de ejecución y una sola vez.

La GPU tiene mayores limitaciones en cuanto a los tipos de datos posibles, pues aunque se utilice un lenguaje de alto nivel para la programación, la GPU sólo utiliza números reales en distintos formatos.

A.2. Utilización de la GPU para problemas de propósito general

Uno de los principales motivos que nos lleva a pensar en la utilización de la GPU en lugar de la CPU es su mayor potencial para realizar cálculos en coma flotante, sobre todo debido al tratamiento vectorial que hace de los datos. Los datos en la GPU son introducidos en forma de flujo de datos, procesados según los programas cargados en el procesador de vértices y/o de fragmentos y el resultado es escrito normalmente en la memoria gráfica o "frame-buffer". Es posible también paralelizar algunos procesos utilizando de manera conjunta la CPU y la GPU.

Normalmente la información que recibe un programa almacenado en la GPU consiste en un flujo de vértices y propiedades asociadas a los vértices, como la normal, color, coordenadas de textura, etc., finalmente el programa genera una serie de píxeles de un determinado color en el "frame-buffer". Si queremos realizar operaciones que necesiten más información, podemos codificarla en una o más texturas, pero a sabiendas de que la velocidad de los algoritmos va a depender en gran medida del tiempo de acceso a la textura y de los retardos ocasionados por un acceso aleatorio a la misma.

Otra razón que nos lleva a plantearnos el uso de la GPU para problemas de propósito general es el mayor ancho de banda en las transferencias en la propia memoria de la tarjeta gráfica, mucho mayor que el ancho de banda de la memoria principal del ordenador.

Por último, el menor coste de la GPU en relación a la CPU, la posibilidad de paralelización de tarjetas y la evolución más rápida de las mismas, nos llevan a pensar que algoritmos hoy implementados ineficientemente en la GPU, en el futuro obtengan muchos mejores resultados. Además todo parece indicar que la evolución del hardware gráfico será más rápida que la de las CPU's, siendo en cualquier caso posible utilizar los dos procesadores simultáneamente para maximizar el rendimiento.

En esta sección llevaremos a cabo un estudio sobre el tipo de algoritmos que pueden ser llevados a una implementación en la GPU, tras lo cual veremos las analogías de programación entre la CPU y la GPU. También daremos una serie de normas sobre cómo implementar algoritmos generales en la GPU. Por otra parte veremos las características principales de las tarjetas gráficas avanzadas que hemos utilizado para la implementación del algoritmo de detección de colisión visto en el Capítulo 6.

A.2.1. ¿Qué tipo de algoritmos pueden ir bien a la GPU?

Para poder utilizar la GPU en una determinada implementación es necesario que los datos de entrada sean paralelizables, y que se les aplique los mismos cálculos de manera repetida y de forma independiente entre sí [Fer05]. No todos los problemas son susceptibles de ser resueltos de esta forma, por lo que no todos los algoritmos pueden implementase en la GPU de manera eficiente. La clave está en el paralelismo y la independencia, es decir, no sólo se deben realizar los mismos cálculos a un flujo de elementos, sino que los cálculos de cada elemento debe tener una dependencia pequeña o ninguna con otros elementos.

Una propiedad de los programas conocida como *intensidad aritmética* consiste en la relación entre el número de cálculos y el ancho de banda, formalmente la *Intensidad aritmética = operaciones / palabras transferidas*. La programación en la GPU exige una alta intensidad aritmética [Fer05].

Las salidas disponibles en el procesador gráfico están limitadas a una serie de valores para modificar el color del "frame-buffer", por lo que no se disponen de las suficientes salidas de datos para la mayoría de los algoritmos de propósito general. Es previsible que esta limitación desaparezca en el futuro.

Por otra parte, los valores de salida de los algoritmos implementados en la GPU quedan almacenados en un buffer de memoria, que en algunos casos hay que recuperar desde la propia tarjeta gráfica, lo que ralentiza el proceso global. Cuando se usa un bus AGP esto es un problema debido a la asimetría de la conexión. Sin embargo, en las placas con PCI-Express, esta limitación se reduce considerablemente.

Los datos de entrada a los algoritmos residentes en el procesador gráfico deben ser adaptados a un conjunto de tipos de datos disponibles. Esto, junto a que los lenguajes de programación actuales de las tarjetas gráficas limitan el conjunto de estructuras de datos a utilizar, reduce en cierto modo el tipo de algoritmos que pueden adaptarse al modelo de programación en la GPU.

A.2.2. Analogías en la programación GPU-CPU

Todas las ventajas de uso que hemos visto vienen limitadas actualmente por una serie de inconvenientes que nos llevan a transformar ciertos problemas o algoritmos que están implementados de una forma más natural en la CPU, a una forma en la que puedan usarse de acuerdo a las características tan particulares de la GPU. A continuación pasamos a enumerar algunas de estas limitaciones en comparación con el método natural utilizado en la CPU, y cómo resolver éstas [Fer05]:

- Para almacenar datos en la GPU se utilizan texturas, tal y como se hace de forma natural en los arrays en la CPU:

  Para almacenar información se puede utilizar una textura o un array de vértices. Estos elementos son leídos por los procesadores programables de la GPU y pueden utilizarse como almacenamiento de información tal y como se hace en un Array. Existen diversos tipos de texturas, 1D, 2D, 3D, y cada uno de estos tipos tiene un tamaño determinado, siendo el tipo de textura más versátil la textura 2D.

- Los programas de fragmentos en la GPU simulan los bucles internos en la CPU:

  En la CPU se puede utilizar un bucle para iterar operaciones sobre elementos almacenados en un array. En la GPU podemos utilizar operaciones similares en un programa de fragmentos que se aplicará a todos los elementos de un flujo de fragmentos de entrada. La cantidad de paralelismo depende del número de procesadores en la GPU, pero también de la utilización del paralelismo de las operaciones sobre vectores.

- Escribir en el "frame-buffer" o en una textura es el modo de obtener realimentación:

  Si deseamos que los datos de salida sirvan de entrada para nuevos cálculos debemos escribir estos datos en la memoria gráfica.

- La rasterización de la geometría provoca los cálculos en la GPU:

  Para poder utilizar los procesadores programables de la GPU es necesario dibujar, de manera que se llame al programa de vértices con un flujo de vértices y se genere un flujo de fragmentos que pase al programa de fragmentos.

- Las coordenadas de textura son el dominio de entrada de una aplicación:

  Un programa posee un dominio de entrada y un rango de salida. En muchos casos este dominio de entrada tiene una dimensión distinta al de los datos de entrada. La forma de tratar con esto es utilizar coordenadas de textura que se almacenan en los vértices. Estas coordenadas de textura se interpolan para generar un conjunto de coordenadas para cada fragmento. Estas coordenadas interpoladas se pasan como entrada al procesador de fragmentos. Podemos utilizar estas coordenadas de textura como índices de arrays para controlar así el dominio de entrada para los cálculos.

- Las coordenadas de vértices controlan el rango de salida de una aplicación:

  Como los programas de vértices no son capaces de modificar las coordenadas del píxel sobre el que escribir un fragmento, los programas de vértices, junto a los vértices de entrada, determinan los píxeles que se generarán. De esta forma las coordenadas de los vértices controlan el rango de salida, típicamente pasando los vértices sin ninguna transformación al programa de fragmentos.

A.2.3. Reglas para obtener un programa adecuado para la GPU

Para aumentar la eficiencia de un programa es necesario la presencia de localidad en las referencias a los datos de memoria, es decir, que los datos referenciados en un breve periodo de tiempo se encuentren en posiciones cercanas de memoria. En general la GPU obtiene un ancho de banda mayor en los accesos secuenciales a memoria que la CPU. En cuanto al acceso a la memoria caché, debido a que la memoria caché se utiliza para acelerar el filtrado de texturas en la GPU, la caché de la GPU necesita ser tan solo del tamaño del filtro de texturas, que normalmente está formado por pocos texels, por tanto la memoria caché no es muy adecuada para la programación de propósito general.

Cuando se inicia una aplicación es necesario enviar los datos bajo la forma de texturas o arrays de vértices a la GPU; si el resultado de los cálculos de la GPU es requerido por parte de la CPU, es necesario obtener estos datos de la GPU. Realizar todos los cálculos directamente en la CPU no precisa de estas transferencias de información.

Debemos considerar que existen distintos formatos para los píxeles y adecuar nuestros datos a ellos. Además no todos los formatos de los elementos de textura y del "frame-buffer" operan a la misma velocidad, y en algunos casos es necesaria una conversión de formato interna en la CPU.

A.2.4. Características avanzadas de la serie GeForce 6.

La aplicación de procesadores gráficos para la programación de aplicaciones genéricas (no de visualización) necesita del conocimiento de la arquitectura de estos procesadores. La GPU está diseñada para la visualización, operación que tiene un alto grado de paralelismo en los cálculos realizados, pues obtiene flujos de píxeles coloreados a partir de flujos de elementos independientes en la forma de vértices. Para esto la GPU posee varios procesadores programables, como hemos visto, que aplican una serie de cálculos a flujos de elementos en paralelo.

A continuación mostramos una serie de características que facilitan la programación de problemas de propósito general en las tarjetas gráficas de la serie *GeForce 6*. El perfil *Shader Model 3.0* permite una serie de características avanzadas que proporcionan mayor flexibilidad en los programas y una mayor capacidad de procesamiento. Entre otras, las características de este modelo son:

Procesador de vértices:

- Incremento del número de instrucciones: Se permiten 512 instrucciones estáticas y 65.536 instrucciones dinámicas. Las instrucciones estáticas hacen referencia al total de instrucciones antes de compilar un programa. Las instrucciones dinámicas hacen referencia al total de instrucciones resultantes después de la compilación.
- Mayor número de registros temporales: hasta 32 registros.
- Soporte para la instanciación.
- Control de flujo dinámico: los saltos y ciclos tienen un coste de dos ciclos.
- Texturización de vértices: se permite utilizar texturas, pero sólo el acceso a las posiciones de los vecinos más próximos está implementado en el hardware. Se puede acceder a cuatro texturas como máximo.

Procesador de fragmentos:

- Incremento del número de instrucciones: Se permiten 65.535 instrucciones estáticas y 65.535 instrucciones dinámicas.
- Múltiples destinos de renderización: El procesador de fragmentos puede obtener su salida como máximo en cuatro buffers de color separados.
- Control de flujo dinámico: Se permite el salto condicional y los bucles, haciendo más flexibles los programas.

- Indexado de atributos: Se puede utilizar un registro de índices para seleccionar los atributos a procesar, permitiendo realizar mediante bucles la misma operación para distintas entradas.

- Soporte para flotantes de 32 y 16 bits de precisión.

- Operaciones en paralelo sobre vectores: Se pueden realizar dos instrucciones en paralelo (una para tres componentes del vector y otra para el restante componente).

- Cada pipeline de una unidad es capaz de realizar 4 operaciones. En el caso de la tarjeta GeForce 6800 se pueden realizar 16 operaciones en el pipeline de cada unidad.

A.3. El lenguaje Cg

Cg [FK03] es un lenguaje de alto nivel que elimina la necesidad de programar en ensamblador las distintas etapas programables de la GPU. Permite su utilización desde un lenguaje anfitrión como *C++*, así como utilizar *OpenGL* o *DirectX* para enviar comandos al pipeline programable para la carga de programas en *Cg*, así como para enviar los vértices y sus parámetros como entrada al procesador de vértices. Tiene una simbología similar al lenguaje de programación *C*.

Este lenguaje permite la utilización de determinados perfiles, cada uno de los cuales especifica el conjunto de operaciones a realizar además del conjunto de datos de entrada y salida de cada etapa programable. Cada perfil se adapta a un conjunto determinado de tarjetas, siendo más complejo mientras más avanzada sea la tarjeta gráfica, como lo son las tarjetas de cuarta generación entre las que se incluyen las tarjetas *NVIDIA GeForce FX* y las *ATI Radeon 9700*.

Veamos cómo dar soporte a los distintos tipos comunicación entre la CPU y la GPU utilizando el lenguaje Cg a través de distintos tipos de variables:

- **Variables de entrada**: Están formadas por los vértices de entrada junto con información asociada a cada vértice, como la Normal o el Color del mismo entre otros. Provocan la ejecución del programa de vértices por cada vértice subministrado.

- **Variables de entrada uniformes**: Subministran a la GPU la información que es común para el flujo de vértices de entrada y que es variable entre frames.

- **Variables constantes**: Subministran información común para el flujo de vértices de entrada y que es fija entre frames (a no ser que sea modificada internamente en la GPU), como es el caso de las texturas.

- **Variables de salida**: Envían información de salida, resultado de los cálculos, a la siguiente etapa del pipeline, al "frame-buffer", o a una textura.

La correspondencia entre variables no uniformes de la GPU y la CPU se realiza mediante la asignación de una semántica, por ejemplo la semántica POSITION indica que se le asociará su correspondiente variable la posición de cada vértice, la semántica NORMAL asociará a su variable la normal de cada vértice. Las variables uniformes se asocian por nombre en lugar de por semántica.

Bibliografía

Bibliografía

[ABDPY97] Avanaim, F.; Boissonnat, J.D.; Devillers, O.; Preparata, F.P.; Yvinec, M. Evaluating sings of determinants using simple-precision arithmetic. *Algorithmica*, Vol. 17, pp. 111 – 132, 1997.

[ABJN85] Ayala, D.; Brunet, P.; Juan, R.; Navazo, I. Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics*, Vol. 4, num. 1, pp. 41 – 59, 1985.

[AH02] Akenine-Möller, T.; Haines, E. *Real-Time Rendering*. 2nd edition. A K Peters, 2002.

[Ant92] Antonio, F. Faster Line Segment intersection. *Graphics Gems III,* Academic Press, pp. 199-202, 1992.

[Bad90] Badouel, F. An efficient Ray-Polygon intersection. *Graphics Gems*. Academic Press, London, pp. 390 – 393, 1990.

[Bar92] Baraff, D. *Dynamic simulation of non-penetrating rigid body simulation*. Ph.D. Thesis, Cornell University, 1992.

[BCGMT96] Barequet, G.; Chazelle, B.; Guibas, L.; Mitchell, J.; Tal, A. Boxtree: A hierarchical representation of surfaces in 3D. *Proceedings of Eurographics'96*.

[Ber04] Bergen, G. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann publishers, Elsevier, 2004.

[Ber97] Bergen, G. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools*, Vol 2, num. 4, pp. 1-14, 1997.

[BKSS90] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. *In Proceedings of ACM SIG-MOD Conf. On Management of Data*, pp. 322 – 331, 1990.

[BO04] Bradshaw, G.; O'sullivan, C. Adaptative medial-axis approximation for sphere-tree construction. *ACM Transactions on Graphics*, Vol. 23, num. 1, pp. 1 – 26, 2004.

[BWS99] Baciu, G.; Wong, W.; Sun, H. RECODE: An Image-Based Collision Detection Algorithm. *Journal of Visualization and Computer Animation*.Vol. 10, num. 4, pp. 181 – 192, 1999.

[Cam97] Cameron, S. Enhancing GJK: Computing minimum penetration distances betweeen convex polyhedra. *Proceedings of the International Conference on Robotics and Automation*, pp. 3112 – 3117, 1997.

[Can86] Canny, J.F. Collision Detection for moving polyhedra. *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 8, num. 2, pp. 200 – 209, 1986.

[Cap01] Capens, N. Smallest Enclosing Spheres. *Flipcode Code of the Day Forum*, June 29, 2001.

[Cha91] Chazelle, B. Triangulating a simple polygon in linear time. *Discrete Computational Geometry*, num. 6, pp 485 – 524, 1991.

[Chu96] Chung, K. An efficient collision detection algorithm for polytopes in virtual environments. *Ph.D. Thesis, University of Hong Kong*, 1996.

[CK86] Culley, R.K.; Kempf, K.G. A Collision Detection Algorithm Based on Velocity and distance bounds. *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, Vol. 2, pp. 1064 – 1069, 1986.

[CKK06] Choi, Y.-J.K.; Kim, Y.J; Kim, M.-H. Rapid pairwise intersection tests using programmable GPUs. *Visual Computer*, Vol 22. pp. 80 – 89, 2006.

[CL03] Chien, Y.R; Liu, J.S. Improvements to Ellipsoidal Fit Based Collision Detection. *Technical Report TR-IIS-03-001, Institute of Information Science, Academia Sinica, Taiwan*, 2003.

[CLMP95] Cohen, J.D; Lin, M.C.; Manocha, D.; Ponamgi, M. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. *Proceedings of the ACM Symposium on Interactive 3D graphics*, pp. 189 – 196, 1995.

[CW96] Chung, K.; Wang, W. Quick Collision Detection of Polytopes in Virtual Environments. *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST 96)*, pp. 125 – 131, 1996.

[Dey91] Dey, K. Triangulations and CSG representation of polyhedra with arbitrary genus. *The 7th ACM symposium on Computational Geometry*. ACM Press, pp. 364 – 372, 1991.

[DK90] Dobkin, D.P.; Kirkpatrick, D.G. Determining the separation of preprocessed polyhedra – a unified approach. *In Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science*. Springer-Verlag, Vol. 443, pp. 400 – 413, 1990.

[Duf92] Duff, T. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics*, Vol. 26, num. 2, pp. 131 – 139, 1992.

[Ebe02] Eberly, D. Intersection of a Sphere and a Cone. *Technical Report, Magic Software*, January 25, 2002.

[Ebe04] Eberly, D.H. *Game Physics*. Morgan Kaufmann publishers, Elsevier, 2004.

[EL00] Ehmann, S.A.; Lin, M.C. SWIFT: Accelerated Proximity Queries Using Multi-Level Voronoi Marching. *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2000.

[EL01] Ehmann, S.A.; Lin, M.C. Accurate and Fast Proximity Queries Between Polyhedra using Convex Surface Decomposition. *Computer Graphics Forum*, Vol. 20, num. 3, pp. 500 – 521, 2001.

[Eri05] Ericson C. *Real-Time Collision Detection*. Morgan Kaufmann publishers, Elsevier, 2005.

[Eri95] Erikson, C. Error Correction of a Large Architectural Model: The Henderson County Courthouse. *Technical Report TR95-013, Department of Computer Science, University of North Carolina at Chapel Hill*, 1995.

[Fei95] Feito, F.R. *Modelado de sólidos y Álgebra de Objetos Gráficos*. Ph.D. thesis. Depto. de Lenguajes y Sistemas Informáticos. Universidad de Granada, 1995.

[Fer05] Fernando, R. *GPU-Gems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, Pearson Education, 2005.

[Fiu89] Fiume, E.L. *The Mathematical Structure of Raster Graphics*. Academic Press, 1989.

[FK03] Fernando, R.; Kilgard, M.J. *The Cg Tutorial. The Definitive Guide to Programmable Real-Time Graphics*. NVIDIA, Addison-Wesley, 2003.

[FT97] Feito, F.R.; Torres, J.C. Boundary representation of polyhedral heterogeneous solids in the context of a graphic object algebra. *The Visual Computer,* Vol. 13, pp. 64 – 77, 1997.

[FT97b] Feito, F.R.; Torres, J.C. Inclusion Test for General Polyhedra. *Computer & Graphics*, Vol. 21, num. 1, pp. 23 – 30, 1997.

[FTU95] Feito, F.R.; Torres, J.C.; Ureña, A. Orientation, Simplicity and Inclusion Test for Planar Polygons. *Computer & Graphics*, Vol. 19, num. 4, pp. 595 – 600, 1995.

[FVFH92] Foley, J.D.; van Dam, A.; Feiner, S.K.; Hughes, J.F. *Computer Graphics. Principles and Practice.* 2nd edition. Addison-Wesley, pp. 113-117, 1992.

[FW93] Fortune, S.; Wyk, C.J. Van. Efficient exact arithmetic for computational geometry. *Proceedings 9th Annual ACM Symposium Foundations on Computational Geometry*, pp. 163 – 172, 1993.

[FWG03] Fan, Z.; Wan, H.; Gao, S. IBCD: a fast collision detection algorithm based on image space using OBB. *The journal of visualization and computer animation*, Vol. 14, pp. 169 – 181, 2003.

[GJK88] Gilbert, E; Johnson, D.; Keerthi, S.S. A Fast Procedure for Computing the Distance Between Complex Objects in Three-dimensional Space. *IEEE Journal of Robotics and Automation*, Vol. 4, num. 2, pp. 193 – 203, 1988.

[GLGT99] Gregory, A.; Lin, M.; Gottschalk, S.; Taylor, R. H-Collide: A Framework for Fast and Accurate Collision Detection for Haptic Interaction, *Proceedings of IEEE Virtual Reality Conference*, 1999.

[GLM05] Govindaraju, N.; Lin, M.C.; Manocha, D. Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling Using Graphics Hardware. *IEEE VR*, 2005.

[GLM06] Govindaraju, N.; Lin, M.C.; Manocha, D. Fast and Reliable Collision Culling Using Graphics Hardware. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, num. 2, pp. 143 – 154, 2006.

[GLM96] Gottcshalk, S.; Lin, M.C.; Manocha, D. OBB-Tree: A hierarchical structure for rapid interference detection. *Proceedings of the ACM SIGGRAPH 96*, pp. 171 – 180, 1996.

[Got96] Gottschalk, S. Separating Axis Theorem. *Technical Report TR96-024*, Dept. of Computer Science, University of North Carolina Chapel Hill, 1996.

[GRLM03] Govindaraju, N.; Redon, S.; Lin, M.C.; Manocha, D. CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 25 – 32, 2003.

[GSF94] García-Alonso, A.; Serrano, N.; Flaquer, J. Solving the Collision Detection Problem. *IEEE Computer Graphics and Applications*, Vol. 13, num. 3, pp. 36 – 43, 1994.

[GSS89] Guibas, L.; Salesin, D.; Stolfi, J. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. *Proceedings of the Fifth Annual ACM Symposium on Computational Geometry*, pp. 208 – 217, 1989.

[Hai94] Haines, E. Point in Polygon Strategies. *Graphics Gems II*, Academic Press, pp. 24 – 46, 1994.

[He99] He, T. Fast Collision Detection Using QuOSPO Trees. *Proceedings of then 1999 Symposium on Interactive 3D Graphics*, pp. 55 – 62, 1999.

[Hel97] Held, M.; ERIT: A Collection of Efficient and Reliable Intersection Tests. *Journal of Graphics Tools*, Vol. 2, num. 4, pp. 25-44, 1997.

[Her86] Herman, M. Fast Three-Dimensional Collision-Free Motion Planning. *Proceedings of the 1986 IEEE International Conference on Robotics and Automation*, Vol. 2, pp. 1056 – 1063, 1986.

[HKM95] Held, M.; Klosowski, J.; Mitchell, J. Evaluation of collision detection methods for virtual reality fly-throughs. *Proceedings 7th Canadian Conference Computer Geometry*, pp. 205 – 210, 1995.

[HLCGM97] Hudson, T.; Lin, M.C.; Cohen, J.; Gottschalk, S.; Manocha, D. V-COLLIDE: Accelerated collision detection for VRML. *Proceedings of VRML 1997*, pp.117 – 123, 1997.

[Hof89] Hoffmann, C. *Geometric and Solid Modelling. An Introduction*. Morgan Kaufmann Publishers, 1989.

[HS97] Huang, C.; Shih, T. On the Complexity of Point-in-Polygon Algorithms. *Computer & Geosciences*, Vol. 23, num. 1, pp. 109 – 118, 1997.

[HTG04] Heidelberger, B.; Teschner, M.; Gross, M. Detection of Collisions and Self-collisions Using Image-space Techniques. *Journal of WSCG*, Vol. 12. Num. 1-3, UNION Agency – Science Press, 2004.

[Hub93] Hubbard, P.M. Space-Time Bounds for Collision Detection, *Technical Report CS-93-04, Department of Computer Science, Brown University*, 1993.

[Hub95] Hubbard, P.M. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, Vol 1, num. 3, pp. 218 – 230, 1995.

[Hub96] Hubbard, P.M. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, Vol. 15, num. 3, pp. 179 – 210, 1996.

[HZLM01] Hoff III, K.E.; Zaferakis, A.; Lin, M.; Manocha, D. Fast and Simple 2D Geometric Proximity Queries using Graphics Hardware. *In Symposium on Interactive 3D Graphics (I3D)*, 2001.

[JFS03] Jiménez, J.J.; Feito, F.R.; Segura, F.R. Algoritmos para la detección de colisión 2D. *XIII Congreso Español de Informática Gráfica (CEIG'03)*, A Coruña, Spain, 2003.

[JFS04] Jiménez, J.J.; Feito, F.R.; Segura, R.J. Utilización de Tetra-Trees y Recubrimientos Simpliciales para la Detección de Colisión entre Esfera y Poliedro Complejo. *XIV Congreso Español de Informática Gráfica (CEIG'04)*, Sevilla, Spain, pp. 243 – 256, 2004.

[JFSO06] Jiménez, J.J.; Feito, F.R.; Segura, R.J.; Ogáyar, C.J. Particle Oriented Collision Detection using Simplicial Coverings and Tetra-Trees. *Computer Graphics Forum*, Vol. 25, num. 1, pp. 53 – 68, 2006.

[JOSF05] Jiménez, J.J; Ogáyar, C.J.; Segura, R.J.; Feito, F.R. Detección de Colisión entre un Sólido y una Nube de Partículas mediante el uso de Hardware Gráfico Programable. *XV Congreso Español de Informática Gráfica (CEIG'05)*, Granada, Spain, pp. 181 – 190, 2005.

[JOSF06] Jiménez, J.J; Ogáyar, C.J.; Segura, R.J.; Feito, F.R. Collision Detection between a Complex Solid and a Particle Cloud assisted by Programmable GPU. *3rd Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS 2006"*, Madrid, Spain, 2006.

[JP02] James, D.L.; Pai, D.K. Dynamic response textures for real time deformation simulation with graphics hardware. *ACM Transactions on Graphics*, Vol. 21, num. 3, pp. 582 – 585, 2002.

[JS00] Jiménez, J.J.; Segura, R.J. Trazado de rayos optimizado para piezas poliédricas complejas. *II Jornadas Andalusíes de Informática Gráfica*, Sevilla, Spain, 2000.

[JSF01] Jiménez, J.J.; Segura, R.J.; Feito, F.R. An optimized raytracing for complex solids. *Proceedings of the WSCG'01,9th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. Ed. V. Skala, UNION Agency, Science Press, Plzen, Czech Republic, 2001.

[JSF02] Jiménez, J.J.; Segura, F.R.; Feito, F.R. Tutorial sobre detección de colisiones en informática gráfica. *Novática (ATI)*, nº Mayo-Junio, pp. 55-58, 2002.

[JSF02b] Jiménez, J.J.; Segura, F.R.; Feito, F.R. Clasificación de métodos de detección de colisión. *III Jornadas Regionales de Informática Gráfica*, Jaén, Spain, 2002.

[JSF02c] Jiménez, J.J.; Segura, F.R.; Feito, F.R. Algorithms for Point-Polygon Collision Detection in 2D. *1st Ibero-American Symposium on Computer Graphics*, Guimaraes, Portugal, 2002.

[JSF03] Jiménez, J.J.; Segura, F.R.; Feito, F.R. Polygon-Polygon Collision Detection in 2D. *Proceedings of Eurographics'03. Poster Presentations*. Granada, Spain, 2003.

[JSF04] Jiménez, J.J.; Segura, F.R.; Feito, F.R. Efficient Collision Detection between 2D Polygons. *Journal of WSCG*, Vol. 12, UNION Agency-Science Press, 2004.

[JSFO06] Jiménez, J.J.; Segura, F.R.; Feito, F.R., Ogáyar, C.J. A robust Segment/Triangle intersection algorithm for interference tests. Efficiency study, *Technical Report, Departamento de Informática, Universidad de Jaén*, 2006.

[JTT01] Jiménez, P.; Thomas, F.; Torras, C. 3D Collision Detection: a survey. *Computer & Graphics*, Vol. 25, pp. 269 – 285, 2001.

[KGLMP98] Krishnan, S; Gopi, M.; Lin, M.C.; Manocha, D.; Pattekar, A. Rapid and accurate contact determination between spline models using shelltrees. *Proceedings of Eurographics'98*, 1998.

[KGS98] Kim, D.; Guibas, L.J.; Shin, S. Fast Collision Detection among Multiple Moving Spheres. *IEEE Transactions of Visualization and Computer Graphics*, Vol. 4, num. 3, pp. 230 – 242, 1998.

[KHMSZ98] Klosowski, J.T.; Held, M.; Mitchell, J.S.B; Sowizral, H.; Zikan, K. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, num. 1, pp. 21 – 36, 1998.

[KK86] Kay, T.; Kajiya J. Ray Tracing Complex Scenes. *Computer Graphics (SIGGRAPH 1986 Proceedings)*, Vol. 20, num. 4, pp. 269-278, 1986.

[KLM02] Kim, Y.J.; Lin, M.C.; Manocha, D. DEEP: Dual-space expansion for estimating penetration depth between convex polytopes. *Proceedings of the IEEE International Conference on Robotics and Automation*, 2002.

[KLR04] Kolb, A.; Latta, L.; Rezk-Salama, C. Hardware-based Simulation and Collision Detection for Large Particle Systems. *Graphics Hardware*, 2004.

[Kno03] Knott, D. *CinDeR: Collision and Interference Detection in Real Time Using Graphics Hardware*. PhD. Thesis, Department of Computer Science, University of British Columbia, 2003.

[Knu97] Knuth, D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3d Edition. Addison-Wesley, 1997.

[KP03] Knott, D.; Pai, D. CinDeR: Collision and Interference Detection in Real Time Using Graphics Hardware. *Proceedings of Graphics Interface*, 2003.

[KPT99] Karabassi, E.; Papaioannou, G.; Theoharis, T. Intersection Test for Collision Detection in Particle Systems. *Journal of Graphics Tools*. Vol. 4, num. 1, pp. 25 – 37, 1999.

[KZ04] Klein, J.; Zachmann, G. Point Cloud Collision Detection. *Computer Graphics Forum*. Vol. 23, num. 3, pp. 567 – 576, 2004.

[Lar95] Larcombe, M. Re: Arbitrary Bounding Box Intersection. *Comp. graphics algorithms usenet newsgroup article*, October 11, 1995.

[LC91] Lin, M.C.; Canny, J.F. Efficient algorithms for incremental distance computation. *IEEE Conference on Robotics and Automation*, pp. 1008 – 1014, 1991.

[LG98] Lin, M.C.; Gottschalk, S. Collision detection between geometric models: a survey. *IMA Conference on Mathematics of Surfaces*, 1998.

[LGLM00] Larse, E.; Gottschalk, S.; Lin, M.; Manocha, D. Fast distance queries with rectangular swept sphere volumes. *IEEE International Conference on Robotics and Automation*, 2000.

[Lub91] Lubachevsky, B.D. How to Simulate Billiards and Similar Systems. *Journal of Computational Physics*, Vol 91, num. 1, pp. 255 – 283, 1991.

[Man88] Mäntylä, M. *An introduction to Solid Modeling*. Computer Science Press, 1998.

[Mir98] Mirtich, B. V-Clip: Fast and Robust Polyhedral Collision Detection. *ACM Transactions on Graphics*. Vol 17, num. 3., pp. 177 – 208, 1998.

[MOK95] Myszkowski, K.; Okunev, O.; Kunii, T. Fast Collision Detection Between Computer Solids Using Rasterizing Graphics Hardware. *The Visual Computer*, Vol. 11, num. 9, pp. 497 – 511, 1995.

[Moo66] Moore, R. *Interval Analysis*. Prentice-Hall, 1966.

[MT97] Moller, T.; Trumbore, B. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, Vol. 2, num. 1, pp. 21 – 28, 1997.

[MW88] Moore, M.; Wilhelms, J. Collision Detection and Response for Computer Animation. *Proceedings of SIGGRAPH'88, Computer Graphics*, Vol 22, num. 4, pp. 289 – 298, 1988.

[NAT90] Naylor, B.; Amanatides, J.; Thibault, W. Merging bsp trees yield polyhedral modeling results. *Proceedings of ACM Siggraph*, pp. 115 – 124, 1990.

[Nay93] Naylor, B. Constructing Good Partitioning Trees. *Proceedings of Graphics Interface 1993*, pp. 181 – 191, 1993.

[OF05] Ortega, L.; Feito, F.R. Collision Detection using polar diagrams. *Computer & Graphics*, Vol. 29, pp. 726 – 737, 2005.

[OJF04] Ortega, L.; Jiménez, J.J; Feito, F.R. Detección de Colisiones en 2D. *Plataforma Avanzada de Modelado Paramétrico en CAD.* Ed. Joan, R; Torres, J.C.; Feito, F.R.; Jaén (Spain). ISBN 84-609-2575-7, 2004.

[OJSF05] Ogáyar, C.J.; Jiménez, J.J; Segura, R.J.; Feito, F.R. Inclusión de Puntos en Sólidos mediante el uso de Hardware Gráfico Programable. *XV Congreso Español de Informática Gráfica (CEIG'05)*, Granada, Spain, pp. 127 – 136, 2005.

[Oro94] O'Rourque, J. *Computational Geometry in C.* Cambridge University, p.2, 1994.

[OSF05] Ogayar, C.J.; Segura, R.J.; Feito, F.R. Point in solid strategies. *Computer & Graphics*. Vol. 29, pp 616 – 624, 2005.

[PRS93] Paoluzzi, A.; Ramella, M.; Santarelli, A. Dimension-independent modelling with simplicial complexes. *ACM TOG*, Vol. 12, pp. 56 – 120, 1993.

[RF00] Rivero, M.L.; Feito, F.R. Boolean Operations on General Planar Polygons. *Computer & Graphics,* Vol. 24, num. 6, pp. 881-896, 2000.

[SE03] Schneider, P.J.; Eberly, D.H. *Geometric Tools for Computer Graphics*. Morgan Kaufmann Publishers, Elsevier Science, San Francisco, 2003.

[Seg01] Segura, R.J. *Modelado de Sólidos mediante Recubrimientos Simpliciales*. Ph.D. thesis. Depto. de Lenguajes y Sistemas Informáticos. Universidad de Granada, 2001.

[SF01] Segura, R.J.; Feito, F.R. Algorithms to test Ray-Triangle Intersection. Comparative Study. *Journal of WSCG*. Vol. 9, num. 3, pp. 76 – 81, 2001.

[SF98] Segura, R.J.; Feito, F.R. An algorithm for determining intersection segment-polygon in 3D. *Computer & Graphics*, Vol. 22, num. 5, pp. 587-592, 1998.

[SFMOT05] Segura, R.J.; Feito, F.R.; Miras, J.R.; Ogáyar, C.; Torres, J.C. An Efficient Point Classification Algorithm for Triangle Meshes. *Journal of Graphics Tools*. Vol. 10, num. 3, pp. 27 – 35, 2005.

[She96] Shewchuk, J.R. Robust adaptive floating-point geometric predicates. *In proceedings of the Twelfth Annual Symposium on Computational Geometry, ACM*, pp. 141 – 150, 1996.

[She97] Shewchuk, J.R. Adaptive floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, Vol. 18, pp. 305 – 363, 1997.

[SHH98] Suri, S.; Hubbard, P.M.; Hughes, J.F. Collision Detection in Aspect and Scale Bounded Polyhedra. *Proceedings of the 9th annual ACM-SIAM symposium on Discrete algorithms*, pp. 127 – 136, 1998.

[Sny92] Snyder, J.M. Interval Analysis for Computer Graphics. *Proceedings of SIGGRAPH'92, Computer Graphics*, Vol. 26, num. 2, pp. 121 – 130, 1992.

[SSTY00] Schömer, E.; Sellen, J.; Teichmann, M.; Yap, C.K. Smallest Enclosing Cylinders. *Algorithmica*, Vol. 27, num. 2, pp. 170-186, 2000.

[ST85] Samet, H.; Tamminen, M Bintrees, CSGtrees, and time. *Proceedings of SIGGRAPH'85, Computer Graphics*, Vol 19, num. 3, pp. 121 – 130, 1985.

[Sta99] Stam, J. Stable fluids. *Proceedings of SIGGRAPH 99, Computer Graphics Proceedings*. pp. 121 – 128, 1999.

[Swa93] Swan, J. E. II Octree-based collision detection with fast neighbor finding. *OSU/ACCAD Technical Report, OSU/ACCAD-12/93-TR7*, 1993.

[TC93] Torres, J.C.; Clarés, B. Graphic objects: a mathematical abstract model for computer graphics. *Computer Graphics Forum*. Vol. 12, num. 5, pp. 311 – 327, 1993.

[TK88] Terzopoulos, D.; Fleischer, K. Deformable models. *The Visual Computer*. Vol. 4, num. 6, 1988.

[TKH*04] Teschner, M.; Kimmerle, S.; Heidelberg, B.; Zachmann, G.; Raghupathi, L.; Fuhrmann, A.; Cani, M.P.; Faure, F.; Magnenat-Thalmann, N.; Strasser, W.; Volino, P. Collision detection for deformable objects. *Computer Graphics Forum*, Vol. 23, num. 1, pp. 61 – 81, 2004.

[Tor92] Torres, J.C. *Representación Abstracta de Objetos Gráficos. Teoría de Objetos Gráficos*. Ph.D. thesis. Depto. de Lenguajes y Sistemas Informáticos. Universidad de Granada, 1992.

[Vat92] Vatti, B. A generic solution to polygon clipping. *Communications of the ACM,* Vol 35, num. 7, pp. 56-63, 1992.

[VCC98] Vemuri, B.C., Cao, Y.; Chen, L. Fast Collision Detection Algorithms with Applications to Particle Flow. *Computer Graphics Forum*. Vol. 17, num. 2, pp. 121 – 134, 1998.

[WCCKW02] Wang, W.; Choi, Y.K.; Chan, B.; Kim, M.S.; Wang, J. Efficient Collision Detection for Moving Ellipsoids Based on Simple Algebraic Test and Separating Planes. *Technical Report TR-2002-16, Department of Computer Science and Information Systems, University of Hong Kong*, 2002.

[Wel91] Welzl, E. Smallest Enclosing Disks (Balls and Ellipsoids). *Lecture Notes in Computer Science,* Vol. 555, Springer-Verlag, pp. 359-370, 1991.

[WH94] Witkin, A.P.; Heckbert, P.S. Using particles to sample and control implicit surfaces. *Proceedings of SIGGRAPH 94. Computer Graphics Proceedings*, pp 269 – 278, 1994.

[WLML99] Wilson, A.; Larsen, E.; Manocha, D.; Lin, M.C. Partitioning and Handling Massive Models for Interactive Collision Detection. *Eurographics'99*, Vol. 18, num. 3, 1999.

[Zac98] Zachmann, G. Rapid Collision Detection by Dynamically Aligned DOP-Trees. *Proceedings of IEEE Virtual Reality Annual International Symposium VRAIS'98*, pp. 90 – 97, 1998.