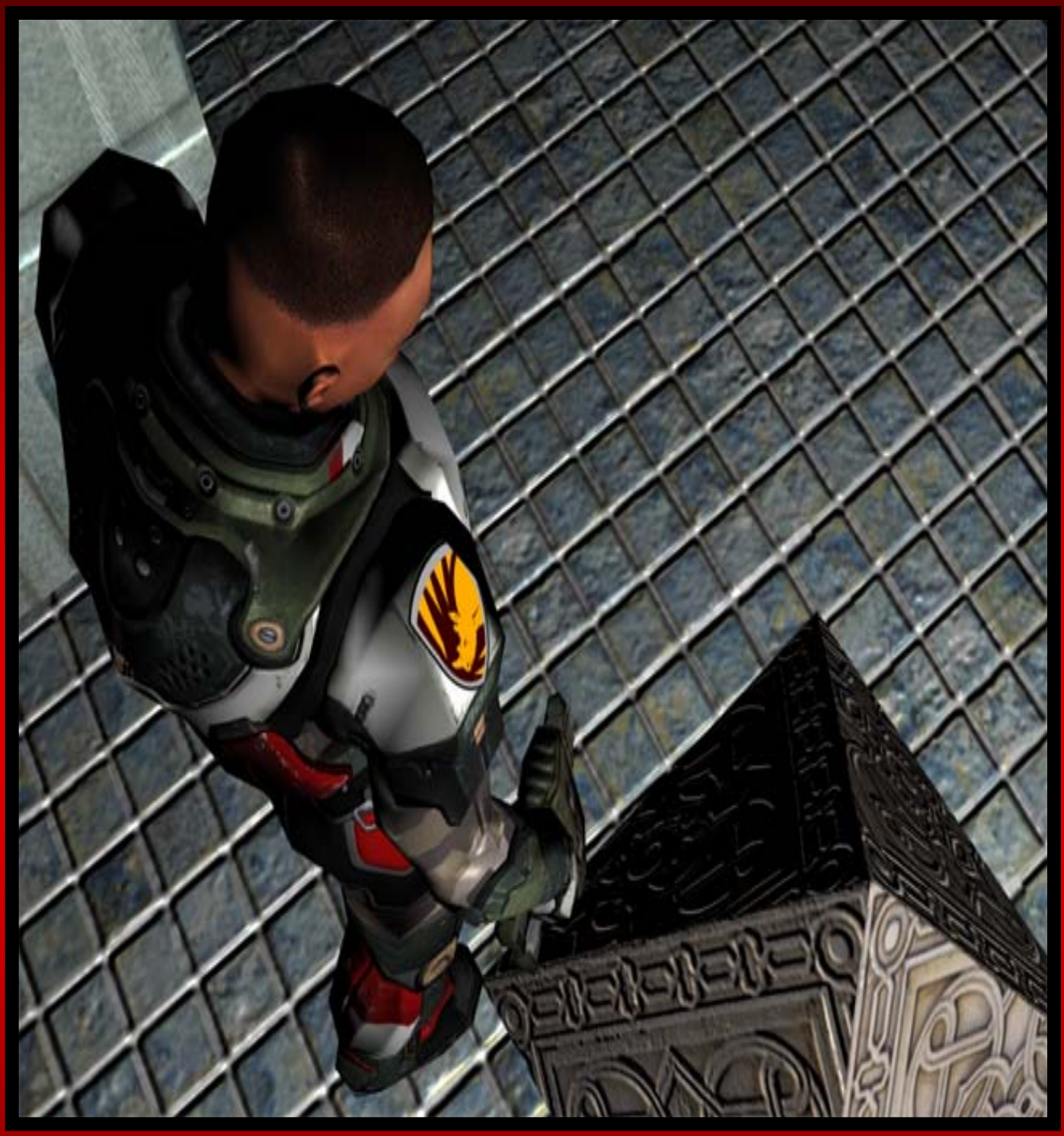


Proyecto de Grado

FingER

Diseño e implementación de un motor gráfico 3D en tiempo real

Integrantes
Javier Filippini
Gabriel Acosta
Aldo Filippini

Tutores
Ing. Eduardo Fernández
Ing. Tomás Laurenzo

Instituto de Computación – Facultad de Ingeniería – Universidad de la República
Montevideo – Uruguay

Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo - Uruguay

Proyecto de grado

FingER: Diseño e implementación de un motor gráfico 3D en tiempo real

Javier Filippini
Gabriel Acosta
Aldo Filippini

Julio de 2007

Tutores: Ing. Eduardo Fernández e Ing. Tomás Laurenzo

Resumen

Este trabajo de proyecto de grado presenta el diseño e implementación de un motor 3d en tiempo real del que se ha implementado un sistema de división espacial, un sistema visualización y un sistema de colisiones y respuesta física. En el diseño de cada uno de estos módulos se persiguió el objetivo de maximizar su rendimiento para lo cual se implementaron distintas técnicas de *optimización*: un *grafo celda-portal*, un sistema de optimización de cambios de estados del *renderer* y un sistema de reducción de verificaciones de situaciones de contacto entre objetos. El diseño del sistema de visualización tuvo en cuenta las características del hardware gráfico que generalmente se utiliza en este tipo de aplicaciones.

Palabras clave: motor 3d, tiempo real, división espacial, visualización, colisiones, respuesta física, renderer, hardware gráfico.

Agradecimientos

A todos los que me vieron desaparecer durante el transcurso de este trabajo, pero muy especialmente a las familias de: Solange Moreira, Norma Echeverría, Marcelo Melluso, Alejandro Bonnecarrere, Juan José Cabrera "El Cala", Diego Valiñas, Emiliano Mazza y Alejandro Díaz. También a la exagerada paciencia y apoyo de los tutores Tomás Laurenzo y Eduardo Fernández.

-Aldo Filippini

Este trabajo no hubiera sido posible sin el apoyo incondicional de mi familia, en especial de mi esposa Alejandra, que se encargó de tantas cosas durante tanto tiempo, de mi hijo Mateo, mi hermosa fuente de inspiración y de Tania, cuya generosidad agradezco todos los días. Además debo especial gratitud a los tutores Eduardo Fernández y Tomás Laurenzo por haberse involucrado con tanta dedicación en todo este proceso de investigación y desarrollo.

- Javier Filippini

A mi hijo Matías, que nació durante el proyecto y el cual ha cedido su tiempo de atención para que su padre pudiera terminarlo.

-Gabriel Acosta

Tabla de Contenido

1

Introducción

1.1 La computación gráfica en tiempo real

La computación gráfica es un área que se ha venido desarrollando en forma activa desde hace varias décadas y sus resultados se han aplicado en áreas muy diversas, desde el cine y los videojuegos hasta la visualización de datos y el análisis médico.

Inicialmente, los mayores esfuerzos fueron realizados con el objetivo de lograr imágenes realistas a partir de descripciones matemáticas de objetos geométricos, superficies de terreno, etc., tratando que su representación gráfica se pareciera lo más posible a lo que se obtendría si se sacaran fotos de objetos reales [FOLE90]. Sin embargo los algoritmos necesarios para la generación de estas imágenes son cada vez más complejos y requieren sistemas que no están al alcance del usuario común.

Por otra parte la industria de los videojuegos, inaugurada en 1972 con la aparición del Pong, se fue desarrollando con gran ímpetu, lo que hizo que aumentara no sólo la demanda sino la cantidad de personas dedicadas a investigar y desarrollar aplicaciones gráficas en tiempo real.

Uno de los resultados de este desarrollo fue la aparición de hardware específicamente diseñado para la generación de gráficos: las tarjetas aceleradoras. Los primeros modelos solo eran capaces de mostrar gráficos bidimensionales. Luego aparecieron modelos capaces de dibujar gráficos tridimensionales así como de realizar muchos de los cálculos accesorios necesarios, liberando a la CPU de esta tarea y aumentando el poder general de los sistemas hogareños [ECCL00]

De esta manera surge la computación gráfica en tiempo real: rama de la computación gráfica que se encarga del estudio de los algoritmos que permiten optimizar la generación de gráficos con el objetivo de utilizarlos en aplicaciones interactivas en tiempo real. Se trata de un área muy activa, cuyo desarrollo se nutre de resultados obtenidos en áreas muy diversas, entre las que cabe mencionar cálculo numérico, investigación operativa, simulación y *optimización*. Como ejemplo cabe mencionar los trabajos de Teller [TELL91], cuyos algoritmos fueron implementados en el videojuego Quake, que resultó ser uno de los más influyentes de la historia [IDSO03].

La complejidad de algunos de estos algoritmos y la necesidad de reutilizar el código de generación de gráficos en muchas aplicaciones, impulsó el desarrollo de arquitecturas donde los componentes gráficos se agrupan en una estructura denominada motor gráfico en tiempo real o *realtime computer graphics engine* [EBER02e].

Este proyecto se ha enfocado en la investigación e implementación de algunas de las técnicas que se utilizan actualmente en los motores gráficos aplicados en la industria de los videojuegos. Este objetivo surge del alcance del proyecto y de las primeras etapas de diálogo entre los estudiantes y los tutores. La industria de los videojuegos es de las más dinámicas y sus técnicas y herramientas se aplican frecuentemente en otras áreas de conocimiento. Algunos ejemplos son: visualización de terreno [SNOO03], modelado arquitectónico [ARCO03], análisis de información médica [KOCH98] y visualización de datos [MERC03]

En el resto de capítulo se da un breve pantallazo sobre las principales funciones de los motores gráficos estudiados y se detallan los objetivos generales del proyecto así como los resultados esperados y obtenidos.

1.2 Los motores gráficos 3D en tiempo real

Los motores gráficos 3D en tiempo real basan su funcionamiento en la generación de gráficos tridimensionales, procesando escenas compuestas por conjuntos de objetos geométricos ubicados en un mundo tridimensional que es observado por una *cámara* virtual encargada de definir que es lo que se ve.

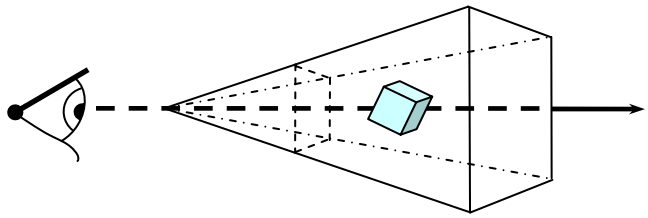


Figura 1.1 *Modelo de cámara virtual*

Según Eberly [EBER02d], la visión intuitiva clásica de lo que hace un motor gráfico 3D en tiempo real es la de dibujar polígonos en el dispositivo de salida. Si bien esta es una de sus tareas, ciertamente no es la única. Las escenas pueden estar compuestas por cientos de objetos geométricos, muchos de los cuales pueden no ser vistos por el *observador* en un determinado momento, ya sea porque están fuera de su campo visual o porque se encuentran ocluidos por otros objetos más cercanos. Este hecho hace que sea posible realizar una optimización en función del campo visual y la posición de los objetos que permita dibujar solo aquellos que realmente se están observando.

Otra de las tareas que comúnmente se le asigna a un motor gráfico está relacionada con la animación de los objetos. Estos pueden moverse de muy diversa forma e incluso estar construidos a partir de jerarquías que les permiten poseer animación propia. El motor debe no solo proveer la funcionalidad de animación sino además la de detectar las colisiones entre los objetos, reportando los puntos exactos de colisión así como otros atributos importantes como la velocidad lineal y angular.

Adicionalmente el motor debe realizar otras tareas como controlar los eventos generados por los dispositivos de entrada, proveer una API suficientemente flexible para que el desarrollador de aplicaciones pueda hacer uso de toda su funcionalidad y estar diseñado de manera que esta pueda ser extendida.

Como puede apreciarse un motor gráfico 3D en tiempo real es una "compleja entidad que consiste en mucho más que una simple capa que dibuja polígonos." [EBER02e]. Es incluso mucho más que una colección de algoritmos de optimización ya que estos deben funcionar en forma simultánea adaptándose a las restricciones de performance general que se imponen en las aplicaciones en tiempo real [BLOW01].

1.3 Motivación y objetivos

Los motores gráficos 3D en tiempo real se han venido utilizando desde hace ya una década y, si bien han visto aumentar sus funcionalidades, sus características generales son muy similares a las descritas en el punto anterior. Existen productos como el Doom 3 engine [IDSO03], Unreal3 engine [UNRE03] y Source [VALV03] que se comercializan con el fin de proveer todas las herramientas necesarias para desarrollar videojuegos de última generación. Estos productos proveen, además de las funcionalidades básicas descritas, módulos de red para crear juegos multijugador, módulos de sonido, de simulación física, sistemas de guiones e incluso editores de gráficos y mapas. Sus costos varían desde unas decenas de miles de dólares a varios cientos de miles.

Por otro lado, al ser ésta una rama con tanto desarrollo, existen muchas implementaciones libres de uso gratuito que incluyen características similares a los motores 3D profesionales. Algunos ejemplos son el Irrlicht [IRRL03], GameBryo [GBRY03], Ogre [OGRE03], aunque existen muchos otros [3DEN03].

La pregunta que surge en forma natural, es ¿por qué es interesante implementar un motor 3D?

La implementación completa de un motor 3D en tiempo real permite comprender en mayor profundidad cuales son las técnicas que forman su núcleo, su dificultad de implementación y su rendimiento en aplicaciones reales. Esto tiene fines tanto teóricos como relacionados con el ejercicio profesional. La mejor comprensión facilita los avances académicos así como aporta al desarrollo de la incipiente industria de videojuegos en el Uruguay, industria que a nivel global mueve miles de millones de dólares anualmente [DFCI03].

Dentro de los objetivos generales planteados en el proyecto se encuentra el de presentar un tema que plantea interés académico para el área de computación gráfica de la Facultad de Ingeniería, ya que si bien gran parte de los algoritmos y técnicas son estudiados en materias optativas, nunca se ha implementado un motor 3D que los combine en un solo producto.

1.4 Resultados esperados

- Desarrollar un motor 3D completo que permita crear aplicaciones interactivas, teniendo especial cuidado en crear una arquitectura que permita expandir su funcionalidad.
- Conocer en detalle cuáles son las técnicas de optimización que se utilizan en los motores 3D en tiempo real e implementar aquellas que permitan obtener, a nuestro criterio, un buen rendimiento general en escenas cuya geometría fija sea altamente ocluyente.
- Estudiar e implementar distintas técnicas que permitan agregarle riqueza visual a los objetos, entre los que se destacan efectos de *iluminación*, reflexión especular, *bump* m*apping* y transparencias.
- Desarrollar un sistema que le permita al motor utilizar geometría creada con un editor de uso libre de manera de facilitar la creación de escenas. El editor elegido fue el GTK Radiant [GTKR00].
- Implementar un sistema de *detección* y *respuesta a colisiones* que pueda ser extendido para incluir simulación física realista. Este sistema incluye un editor de colisiones que permite analizar el rendimiento del sistema.

- Implementar un sistema de animación de caracteres flexible, preferentemente de uso libre, que permita agregar personajes animados a las escenas

1.5 Organización de la documentación

Este documento cumple la función de presentar todos los aspectos relacionados con el proyecto de grado implementado, yendo de lo general a lo particular, mostrando desde la arquitectura general del motor hasta los detalles técnicos necesarios para extender su funcionalidad.

En el capítulo dos se presenta un estudio somero sobre las características generales de los motores 3D en tiempo real y se presenta la base teórica de las técnicas implementadas.

En el capítulo tres se describe la arquitectura general del sistema implementado explicando brevemente cuáles son las funciones de los módulos principales.

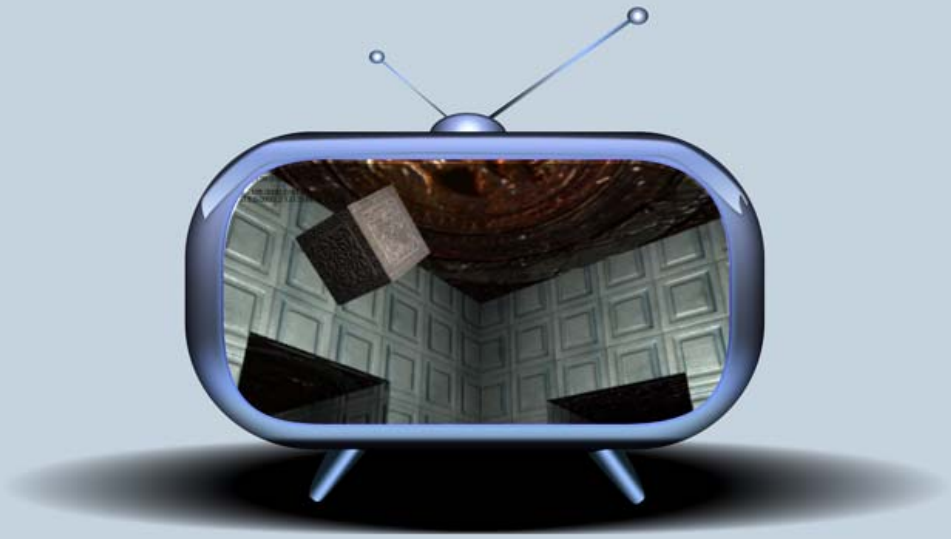
Los capítulos cuatro, cinco y seis desarrollan los aspectos técnicos de las tres áreas centrales de nuestro trabajo: el manejador de *grafo de escena* y *renderer* , el sistema de chequeo y *respuesta a colisiones* y el sistema de creación de mapas y optimización espacial.

El capítulo siete presenta los resultados obtenidos y los trabajos futuros.

Por último, los apéndices detallan aspectos técnicos referentes a cada una de las tres grandes áreas del proyecto: A1 a A7 detallan el *grafo de escena*, animación de caracteres y los algoritmos de optimización implementados; B1 a B7 detallan todos los algoritmos correspondientes al sistema de *detección de colisiones* y respuesta física; C1 a C7 describen los algoritmos de partición espacial y unión con el editor GTK Radiant [GTKR00].

2

Motores Gráficos 3D en Tiempo Real



2.1 Introducción

Según se mencionó en el capítulo 1, los motores 3D son herramientas que permiten el desarrollo de aplicaciones gráficas tridimensionales cuyo rendimiento permita una interacción fluida con el usuario. Para lograr esto, su arquitectura está diseñada de manera de generar imágenes en forma continua, estableciéndose un ciclo en el cual la acción del usuario afecta la generación de la imagen siguiente.

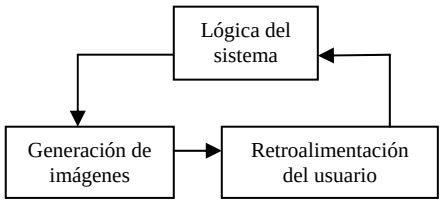


Figura 2.1 *Luego de la generación de cada imagen la acción del usuario retroalimenta la etapa lógica que decide qué imagen generar a continuación.*

La velocidad a la que se generan las imágenes se mide en *cuadros por segundo*, o *frames per second* (*FPS*) en inglés. Se considera que una aplicación debe generar imágenes a por lo menos 15 *FPS* para brindar una sensación de movimiento fluido, mientras que a partir de 72 *FPS* las diferencias en la calidad de la fluidez son indetectables [MOLL02h].

La cantidad de *FPS* que la aplicación es capaz de generar depende de varios factores, entre los que se encuentran la complejidad de la escena y el aspecto de cada objeto constitutivo.

La complejidad de una escena se mide en la cantidad de polígonos totales procesados. Se estima que una escena medianamente compleja puede estar compuesta por cientos de objetos, cada uno conteniendo a su vez cientos o miles de polígonos [MOLL02i].

En cuanto al aspecto se puede decir que generalmente se busca riqueza visual en las imágenes, para lo cual los objetos deben ser procesados utilizando distintas técnicas que les permitan agregarles, entre otros, efectos de brillos especulares, *texturas* y sombras.

En este contexto, el requerimiento de velocidad de generación se fusiona con el de riqueza visual definiendo un concepto más general que permite medir la calidad en la generación de gráficos en tiempo real.

La conclusión a la que se llega es que cuanto más se pueda acelerar el procesamiento de los objetos en la escena mayor será la cantidad de *FPS* que se podrá generar y mayor su riqueza visual. Ambos aspectos impactan positivamente en la calidad general de la aplicación desarrollada.

2.2 Arquitectura general

Dentro del contexto relacionado con el desarrollo de videojuegos, Blow [BLOW01] indica que la parte más difícil de su implementación es la relacionada con el diseño de la arquitectura. Esto se debe a dos factores: la complejidad y el tamaño del proyecto involucrado. En la siguiente figura pueden apreciarse los módulos involucrados en un juego 3D estándar actual:

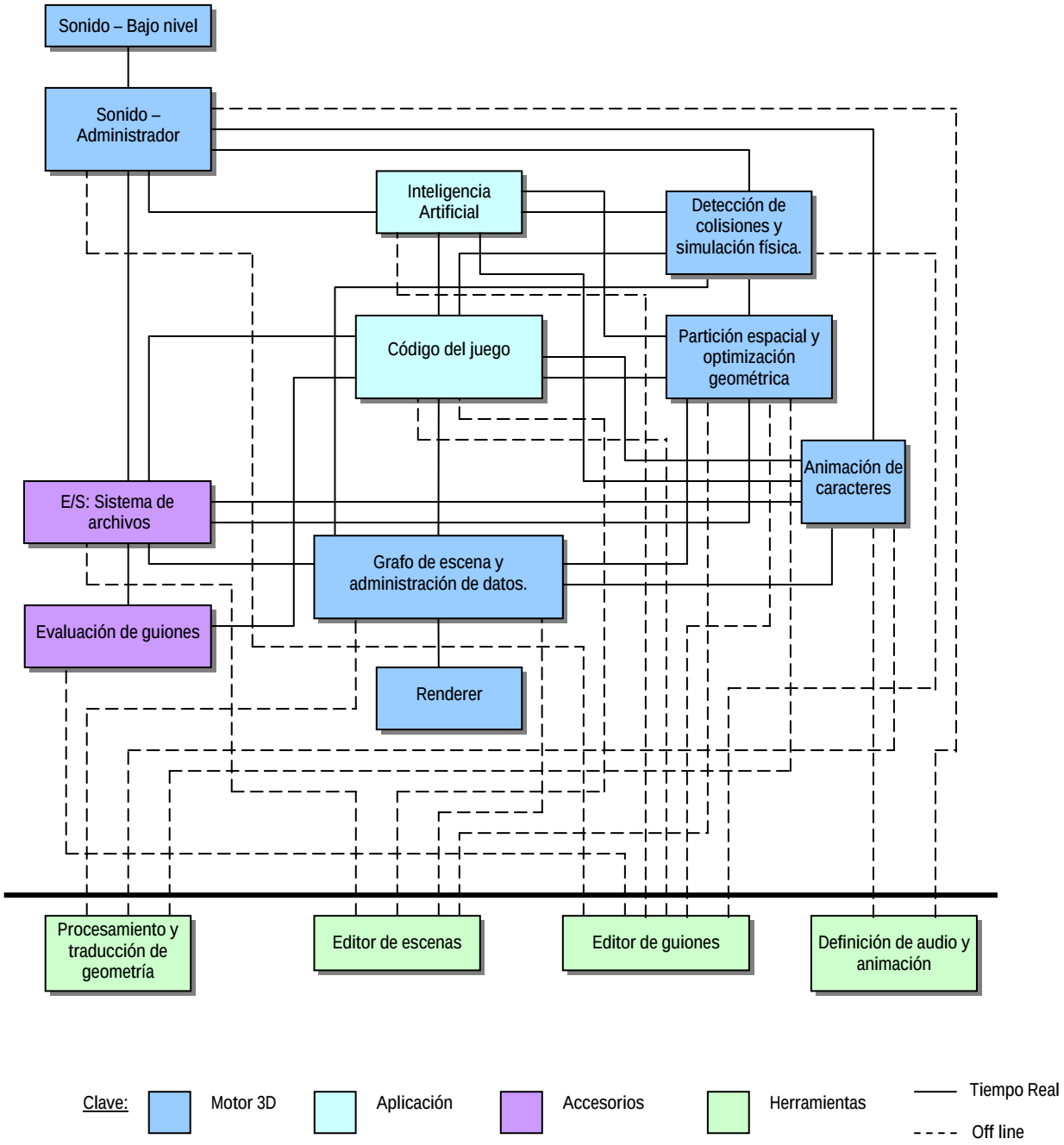


Figura 2.2 *Módulos en un juego 3D estándar actual. Cada nodo del grafo representa un área de importancia. Las líneas indican una relación de acoplamiento entre módulos. Solo los módulos de la aplicación corresponden al trabajo del desarrollador. Sacado [BLOW04]*

Los módulos mostrados en la figura 2.2 se pueden clasificar de la siguiente manera:

- Módulos del motor 3D: Comprende los módulos encargados de la generación de gráficos y sonido. Dentro de las tareas que realiza cabe destacar la de administrar los recursos de la aplicación, optimizar el procesamiento y generación de imágenes y la animación de caracteres. El sonido, al ser una fuente ubicua, se procesa también a este nivel.

- Módulos accesorios: Comprende los módulos que proveen acceso a recursos secundarios como el sistema de archivos, red y el procesamiento de guiones que automatizan algunos aspectos de la generación de escenas animadas.

- Módulos de la Aplicación: Comprende todos los módulos que el desarrollador de aplicaciones debe implementar para crear su aplicación. En el caso de los videojuegos incluye toda la lógica de procesamiento y, dependiendo de la aplicación, módulos encargados de aportar inteligencia artificial a los caracteres.

- Módulos de herramientas: Comprende todos los programas accesorios de apoyo al desarrollo de aplicaciones. Su función fundamental es generar archivos con datos gráficos y sonoros compatibles con los estándares impuestos por el diseño del motor.

La cantidad de trabajo extra-aplicación necesaria para implementar un juego 3D es, como puede apreciarse, muy significativa [BLOW01]. Es por ello que existen paquetes de desarrollo comerciales encargados de brindar las funcionalidades gráficas de animación y sonido, permitiendo a los desarrolladores centrarse en la codificación de la aplicación en sí.

Dado que nuestro proyecto se centró en la generación de gráficos y animación, a continuación se presentan algunos de los algoritmos más utilizados en el diseño de los módulos asociados.

2.3 Grafos de escena y optimización geométrica

Visto como una caja negra, un motor 3D se comporta como un consumidor-productor: consume polígonos y produce gráficos en el dispositivo de salida [EBER02d]. En este contexto, el consumidor puede verse afectado por dos situaciones: ser alimentado con demasiados datos o quedarse esperando por períodos de tiempo hasta que la aplicación lo requiera. Se necesita, por lo tanto, un módulo que actúe de *front-end* y que administre de manera eficiente los datos que el módulo de dibujo o *renderer* procesa:

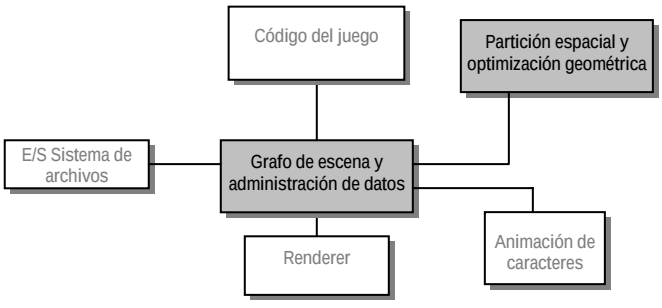


Figura 2.3 *Grafo de escena y optimización geométrica dentro de la arquitectura de un videojuego 3D. Se encargan de administrar los datos que el módulo de dibujo, o Renderer, procesa previo a la generación de la imagen final.*

A este módulo se lo denomina g*rafo de escena* y es donde toda la información de la escena es cargada y procesada para generar cada imagen.

Una de las técnicas más usadas para acelerar el proceso de dibujado consiste en no dibujar lo que no se ve. Para ello se debe realizar una verificación de los objetos geométricos de la escena que permita determinar si son o no vistos por el observador. Este trabajo, que se realiza en el módulo de optimización geométrica, consiste en chequear si el volumen de los objetos intersecan o no el *volumen de vista* observado por la *cámara* virtual en todo momento.

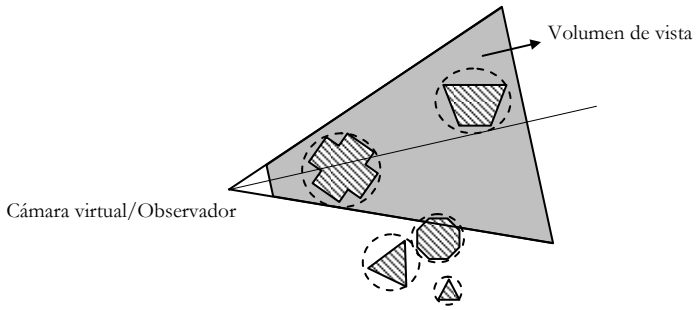


Figura 2.4 *La optimización de la etapa de dibujado se basa en no procesar los objetos que no se ven.*

Una manera eficiente de realizar este trabajo es organizar los objetos en una estructura de árbol donde cada nivel incluya los elementos del nivel inferior de manera que, para n objetos, las verificaciones se puedan realizar en O(log n), en promedio. La información contenida en los nodos del árbol consiste, fundamentalmente, en el volumen que ocupan los objetos geométricos asociados. Adicionalmente se enriquece esta información con atributos que permiten describir el aspecto final de los objetos, como por ejemplo luces, texturas y colores. Un detalle destacable de este sistema es que provee la posibilidad de agrupar naturalmente conjuntos de objetos geométricos relacionados, dada la coherencia espacial generalmente existente en las escenas. Por ejemplo, para que una *luz* afecte solamente a un conjunto de objetos, simplemente se la ubica en un nodo superior a estos dentro del árbol.

Por lo tanto se define *grafo de escena* como la estructura de árbol de los objetos de una escena, enriquecida con atributos descriptivos. Este grafo cumple a la vez las funciones de descripción y optimización de los chequeos de *visibilidad* y colisión de los objetos que conforman una escena.

A continuación se introducen tres conceptos básicos correspondientes a tres estructuras de agrupación espacial comúnmente utilizadas en la construcción de los grafos de escena: las jerarquías de volúmenes acotantes, los *octrees* y los *árboles BSP*.

2.3.1 Jerarquías de volúmenes acotantes

Se define *volumen acotante* como el volumen que contiene un conjunto de objetos. En general, los volúmenes acotantes son objetos geométricos simples: esferas, cajas alineadas con los ejes de coordenadas o cajas orientadas arbitrariamente.

En las jerarquías de volúmenes acotantes, el *grafo de escena* es un árbol formado a partir de nodos que contienen un *volumen acotante*, que engloba toda la geometría del subárbol que lo tiene como raíz. Así, la raíz de todo el árbol contiene el *volumen acotante* de toda la escena y las hojas, el *volumen acotante* de los elementos más simples.

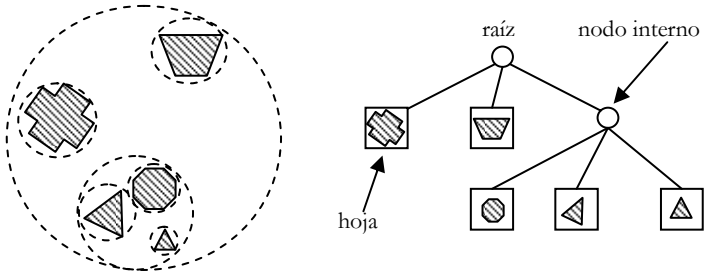


Figura 2.5 *Jerarquía de volúmenes acotantes. La imagen de la izquierda muestra una escena con cinco objetos, cada uno englobado por un volumen acotante circular. Cada objeto es agrupado en un círculo mayor hasta que toda la escena queda englobada por el círculo mayor. La imagen de la derecha muestra el árbol que modela esta escena*

Durante la etapa de dibujado, se utiliza el árbol de jerarquías para determinar cuáles son los objetos que el observador está viendo. Para ello, se verifica si los volúmenes acotantes definidos en la escena intersecan al *volumen de vista* definido por el observador. Como esta verificación se realiza en forma jerárquica, logra eficientemente descartar rápidamente aquellos objetos que se encuentran fuera del *volumen de vista*. Detalles de esta estrategia pueden encontrarse en [MOLL02j]

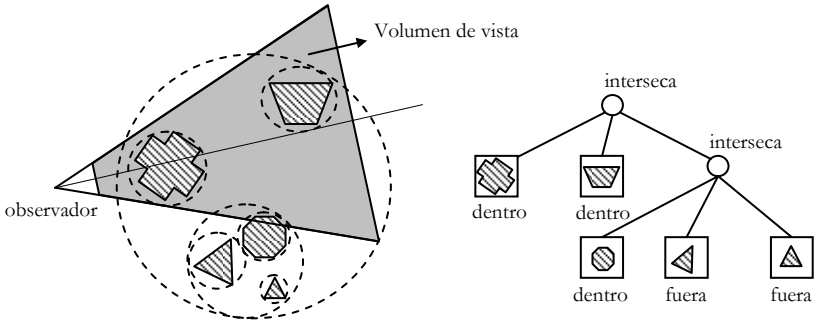


Figura 2.6 *Algoritmo de oclusión jerárquica. La jerarquía de volúmenes acotantes se interseca con el volumen de vista definido en función de la posición del observador. El volumen acotante de la raíz del árbol interseca al volumen de vista y la verificación continúa hacia los nodos internos determinando recursivamente qué objetos hay que dibujar.*

2.3.2 Octrees

La idea general de los *octrees* es similar a la de las jerarquías de volúmenes acotantes. La diferencia está en que en los *octrees*, los volúmenes acotantes son cajas orientadas en forma paralela a los ejes coordenados. Para construirlo, se parte de una caja que ocupa el volumen de toda la escena y se la divide simultáneamente con planos que pasan por su centro y son paralelos a los tres ejes coordenados. Este proceso continúa en forma recursiva hasta que algún criterio de detención es alcanzado. De esta manera, en cada división se crean ocho nuevas cajas idénticas, con un volumen igual a un octavo del original. Algunos de los criterios de detención que se utilizan son que el tamaño de las *celdas* lleguen a determinado valor, que el nivel recursión llegue a determinado valor o que la cantidad de objetos/polígonos dentro de las cajas sea menor que determinado valor.

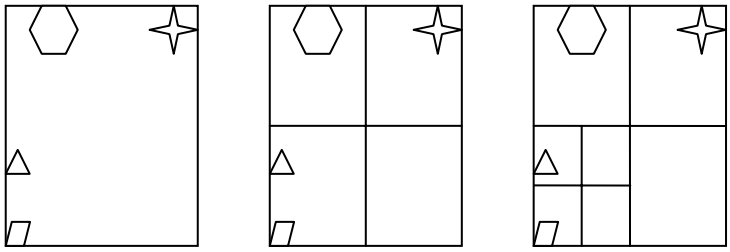


Figura 2.7 *Construcción de un quadtree: versión 2D de un octree. Se parte de un rectángulo que contiene a toda la escena y en un primer paso se la subdivide según dos ejes paralelos a los ejes coordenados, generando cuatro nuevas cajas. El proceso continúa recursivamente mientras las cajas contengan más de un objeto.*

El algoritmo de visibilidad utiliza el *volumen de vista* para recorrer el *grafo de escena* y determinar cuales son las cajas que el observador ve en todo momento, procediendo luego a dibujar los objetos contenidos en ellas.

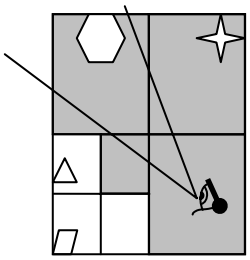


Figura 2.8 *Visualización por celdas en un quadtree. Sólo las celdas grises se cortan con el volumen de vista por lo que las figuras de abajo a la izquierda no son enviadas al renderer, optimizando el proceso de dibujo.*

Cada vez que un objeto atraviesa una celda se debe reflejar su nueva relación de pertenencia dentro del *grafo de escena*, por lo que el sistema debe recorrer todo el grafo a efectos de encontrar la celda que lo contiene.

2.3.3 Árboles BSP

La idea detrás de los *árboles BSP* consiste en dividir recursivamente la geometría fija de la escena con planos de corte, de modo que cada plano separe a los polígonos en dos conjuntos: los que quedan en el semiespacio positivo y los que quedan en el semiespacio negativo [MOLL02j].

Una de las estrategias de elección de planos de corte consiste en utilizar planos coplanares con polígonos de la escena. En estos árboles, la subdivisión resultante divide al entorno en celdas. Las celdas son porciones del espacio delimitadas por geometría y portales. Los portales son las áreas de conexión entre celdas y se corresponden con lo que intuitivamente se asocia a las puertas y ventanas que conectan dos habitaciones (ver figura 2.9).

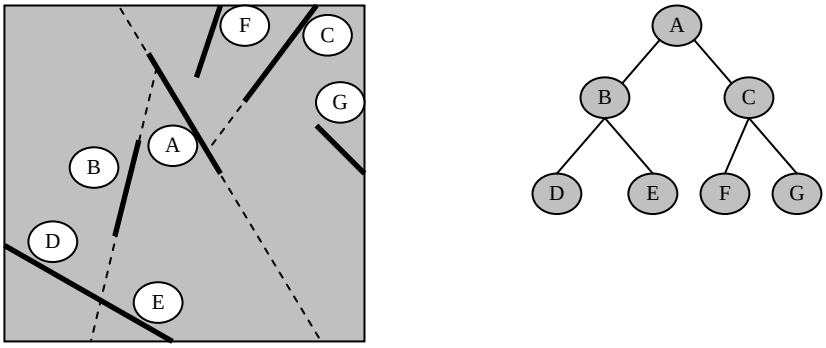
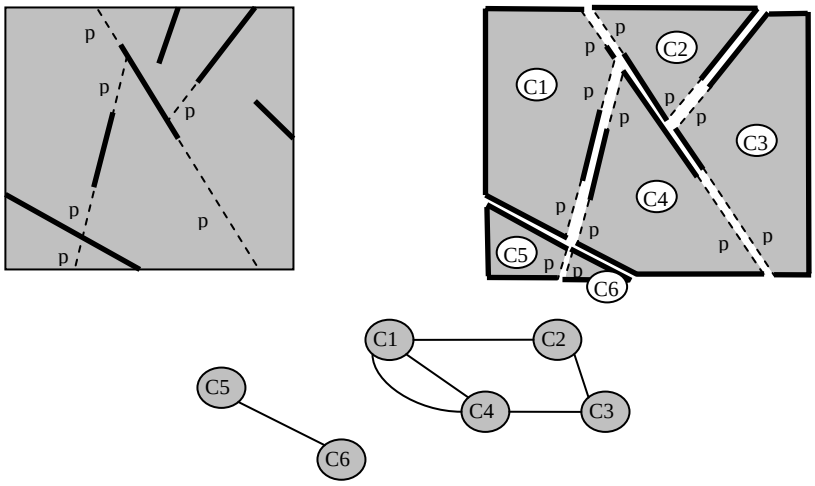


Figura 2.9 *Árbol BSP alineado a polígonos. El espacio es inicialmente dividido utilizando el polígono A. Luego, cada semiespacio es dividido por B y C. El proceso continúa hasta considerar todos los polígonos de la escena, o cuando se alcance una cantidad establecida de polígonos en el semiespacio.*

La relación existente entre celdas y portales se modela utilizando una estructura de grafo denominada *grafo celda-portal.*



Grafo celda-portal

Figura 2.10 *Construcción de un grafo celda-portal a partir de un árbol BSP, partiendo de la subdivisión con planos de corte coplanares con polígonos de la escena. El algoritmo reconoce las zonas de conexión entre celdas (marcados 'p' en las figuras) y las utiliza para definir las celdas disjuntas. Luego se determinan los portales comunes a las celdas para definir celdas conexas y armar el grafo celda-portal.*

El *grafo celda-portal* puede ser utilizado para optimizar el proceso de visualización, dado que se visualizarán en cada instante solamente los objetos contenidos en el conjunto de celdas visibles. A partir de la ubicación del observador en el grafo y de su dirección de observación, el algoritmo encuentra todas aquellas otras celdas que el observador visualiza a través de los portales. Si las escenas están compuestas por geometría fija altamente ocluyente[1], entonces desde una celda se podrán ver pocas celdas adyacentes, optimizando mucho el proceso de dibujado [TELL91; TELL92].

[1] Podemos definir geometría altamente ocluyente como aquella en la que el área de la geometría de las celdas es mucho mayor que el área de sus portales, haciendo que desde una posición cualquiera se pueda observar un número pequeño de celdas.

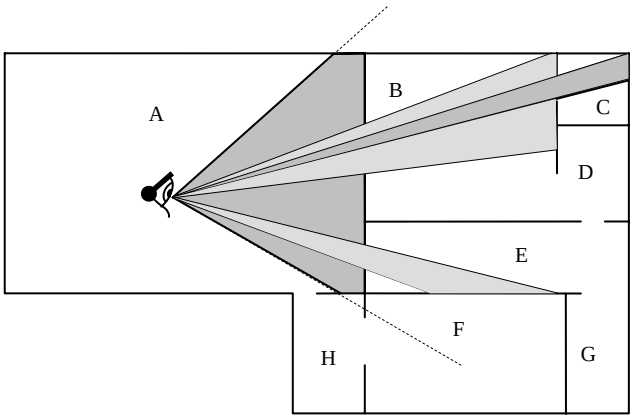


Figura 2.11 *Visibilidad por portales. Las celdas están nombradas de A a H. Los portales son las aberturas que conectan celdas. Solo las celdas que se ven a través de los portales son dibujadas. En el ejemplo A, B, C y E.*

Las mismas ideas pueden aplicarse para realizar la optimización de colisiones dado que cualquier objeto móvil solamente podrá interactuar con objetos que se encuentren en la o las mismas celdas que él. Utilizando estructuras y algoritmos que permitan determinar rápidamente la ubicación de los objetos en el grafo, es posible restringir la verificación de colisiones a un subconjunto de objetos de cardinalidad muy inferior a la totalidad de objetos que componen una escena.

2.3.4 Otras técnicas de optimización geométrica

La idea fundamental detrás de estas técnicas es reducir el número de polígonos que el SGM envía al *renderer* en cada *cuadro* de animación.

Tanto las jerarquías de volúmenes acotantes como los *octrees* son estructuras genéricas diseñadas para ser aplicadas en cualquier tipo de escenas por lo que su eficiencia depende fuertemente del tamaño y organización de los objetos que la componen.

Los *árboles BSP* forman parte de otro tipo de técnicas que aprovechan ciertas características de las escenas para obtener mayores niveles de rendimiento. Dentro de estas cabe destacar las denominadas de oclusión u *Occlusion Culling* que pretenden optimizar el dibujado eliminando el procesamiento de objetos que se encuentran obstruidos por otros. Por más información referirse a [MOLL02e; WON00; MOLL02c; MOLL02d].

Otra importante técnica de aceleración que se aplica directamente a objetos geométricos, se basa en determinar qué *nivel de detalle* (*LOD*: *level of detail*) es el más adecuado para dibujar un objeto que se encuentre a una distancia dada del observador. Cuanto más cerca se encuentre un objeto, será necesario dibujar una cantidad mayor de los polígonos que lo definen, ya que se requiere un mayor detalle visual del objeto. Sin embargo, si el objeto se encuentra lejos del observador, seguramente sea suficiente dibujar una cantidad reducida de ellos (ver Figura 2.11)

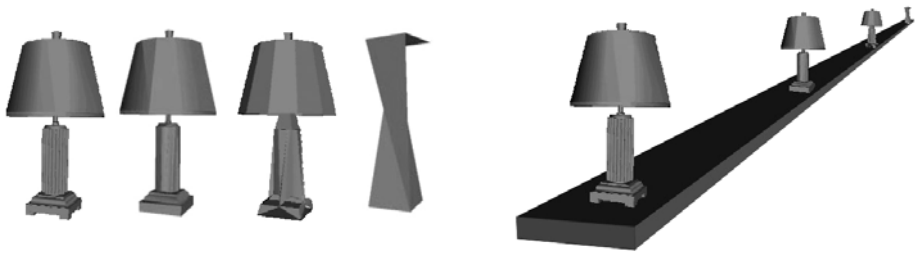


Figura 2.12 *Técnica de nivel de detalle aplicada a un objeto. En la imagen izquierda se muestra cada nivel de detalle a dibujar de acuerdo a diferentes distancias al observador. El resultado, en la derecha, resulta visualmente convincente a la vez que se disminuye la cantidad de polígonos a dibujar en función de la distancia.*

Existen varias maneras diferentes de implementar los niveles de detalle, cada una con sus ventajas y desventajas, pero básicamente se diferencian en la manera como se eligen los distintos niveles y en la forma en que se realizan las transiciones de un nivel a otro [MOLL02k].

2.4 Renderer y riqueza visual

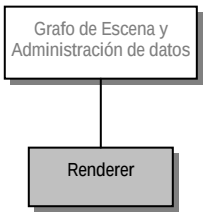


Figura 2.13 *Renderer dentro de la arquitectura de un motor 3D. Corresponde a la última etapa de procesamiento de datos. Es el encargado de dibujar los gráficos en el dispositivo de salida.*

El aspecto final de cada objeto dibujado por un motor 3D no solo depende de su geometría, sino también de un conjunto de atributos que involucran características propias del objeto, como su color y material, y de las escenas en la que están ubicados, como luces y sombras.

Una de las formas de modelar materiales consiste en asignar, a cada objeto, propiedades físicas de absorción, reflexión y refracción de la *luz* para luego calcular cuál es su aspecto final en función de la cantidad de *luz* que recibe cada punto en su superficie [MOLL02g]. Si bien esto es posible, generalmente la cantidad de cálculos necesarios es demasiado grande para realizarse en tiempo real con escenas medianamente complejas, por lo que se debe recurrir a otras técnicas, aproximativas, para acelerar el proceso de dibujado.

La técnica más usada para la modelación de materiales en tiempo real se denomina *texture mapping*. Consiste en la utilización de imágenes que se "mapean" a los polígonos de los objetos geométricos definiendo, de esta manera, el color de cada punto de su superficie. (Figura 2.12).

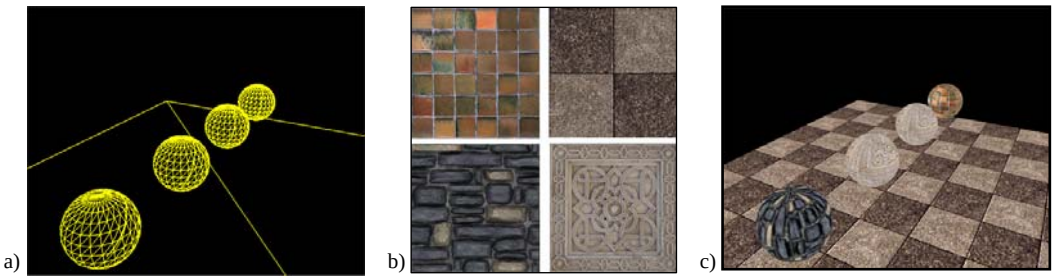


Figura 2.14 *Texture Mapping. a) Geometría de la escena. b) Texturas. c) Escena con mapeo de texturas. No incluye las sombras, brillos y demás efectos relacionados con la iluminación.*

El poder expresivo de las técnicas de aplicación de texturas es muy grande. Es por ello que los diseñadores de hardware gráfico han hecho especial énfasis en acelerar su performance y enriquecer sus posibilidades. De esta manera, las tarjetas aceleradoras han pasado de poder aplicar una textura por polígono a aplicar más de una textura a la vez, pudiendo definir funciones de combinación entre cada una de ellas y, últimamente, definiendo un sistema que permite expandir las funciones de combinación hasta transformarse en un verdadero lenguaje de programación denominado lenguaje de *sombreado* o *shader lenguaje* [NVID02; NVID04].

Utilizando este sistema se pueden crear efectos que simulan, entre otros, la aplicación de luces con reflexiones difusas y especulares. (Figura 2.13).

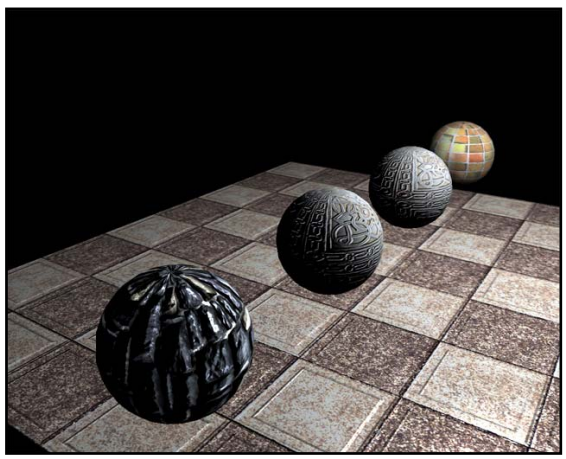


Figura 2.15 *Ejemplos de materiales. La aplicación de texturas unido a las posibilidades de los lenguajes de sombreado permite aumentar significativamente el poder de expresión. Compárese esta imagen con las de la figura 2.12*

2.4.1 Los lenguajes de sombreado o *Shader Languages*

Las imágenes generadas por computadora para películas de animación como Toy Story o Cars no son generadas en tiempo real sino que son el producto de horas de cálculo off-line realizados por procesadores estándar. Dado que la velocidad no es un determinante principal, la ventaja de utilizar CPUs consiste en que los artistas y programadores pueden crear prácticamente cualquier efecto, pudiendo generar imágenes de gran calidad y realismo [NVID02].

Tradicionalmente, el diseño de los sistemas en tiempo real permitía la aplicación de una serie de efectos que estaban más o menos preprogramados en el hardware, limitando las posibilidades de expresión a los desarrolladores de aplicaciones.

La diferencia fundamental entre estos dos sistemas de generación de imágenes radica en la posibilidad de programar el aspecto de cada punto utilizando un lenguaje de alto nivel [NVID02].

Para eliminar esta limitación, los diseñadores de hardware desarrollaron una nueva generación capaz de aceptar programas que se ejecutan en la etapa de rasterización de los polígonos, aumentando significativamente las posibilidades de expresión respecto a los sistemas anteriores [NVID02].

La aparición de los modelos GeForce 3 y GeForce 4 de NVidia marcó los primeros pasos en esta nueva etapa. Tomando ideas del lenguaje RenderMan [APOD99], utilizado en sistemas de rendereo offline, desarrollaron un conjunto de unidades programables que podían ser utilizadas para realizar cálculos de iluminación y combinación de texturas en forma flexible. Estas unidades se denominaron *Register Combiners* y conformaron las primeras versiones de los lenguajes de *sombreado* en tiempo real: *Shader* model 1.1 y 1.5 respectivamente [NVID04b].

Las versiones posteriores de este diseño ampliaron significativamente la posibilidades de programación, por lo que se desarrollaron tres lenguajes especialmente diseñados para utilizar las unidades lógicas de procesamiento: el Cg desarrollado por NVIDIA, el HLSL desarrollado por Microsoft para ser usado con DirectX y el GLSL desarrollado para ser utilizado con OpenGL.

En los apéndices A.4 y A.5 se desarrollan ejemplos de implementaciones de ecuaciones de *luz* y efectos de reflexión especular utilizando el *Shader* model 1.5.

2.5 Colisiones y respuesta física

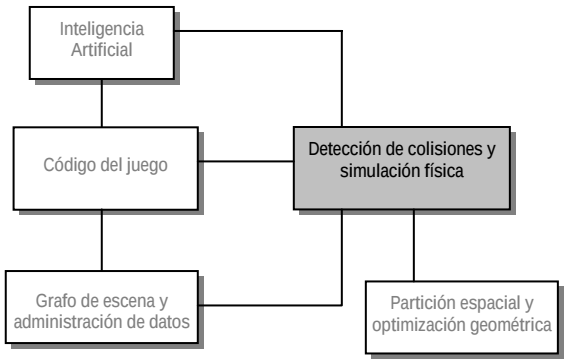


Figura 2.16 Módulo de detección de colisiones y respuesta física dentro de la arquitectura de un motor 3D.

2.5.1 Introducción

El poder actual de las tarjetas graficas para producir efectos visuales de alta calidad, es enorme. Mucho del trabajo que tradicionalmente se hacia programando la CPU, ha sido absorbido por el hardware gráfico o por el API, típicamente OpenGL o DirectX, construido sobre el. Esto ha permitido que los programadores se concentren en aspectos gráficos de más alto nivel. Desde el punto de vista visual los diseñadores de juegos y grafistas tienen en sus manos el hardware necesario para poder crear mundos de apariencia realista. Pero, en este camino, el realismo visual es solamente una mitad del asunto. El realismo físico es la otra mitad. Un personaje bien diseñado visualmente y con una animación convincente, llamaría mucho la atención si atravesara las paredes de una habitación. También si el personaje no pudiera interactuar con los elementos que lo rodean, de una forma físicamente realista, la aplicación no seria demasiado interesante.

Hasta que llegue el día en que el soporte de física sea liberado de las responsabilidades de la CPU [AGE02], será necesario implementar este tipo de comportamiento usando la CPU. Y será necesario implementarlo en forma eficiente, para no perder la sensación de animación: Si la etapa de render es capaz de producir imágenes a 60 *FPS*, pero el sistema físico no es capaz de manejar interacciones entre objetos de forma eficiente, la cadencia de imágenes disminuirá notoriamente. Entonces, si este tipo de realismo físico es lo que se busca, es necesario entender como modelarlo de modo que sea rápido, preciso y robusto.

El modelo de colisiones y respuesta adoptado en el proyecto se divide en 2 partes bien diferenciadas: por un lado se implementaron algoritmos especializados en la *detección de colisiones* entre objetos rígidos convexos, y por otro, se implementó un módulo de física para simular una respuesta realista a las colisiones detectadas. Este es un modelo comúnmente utilizado [BARA92; MIRT96; MIRT98a; GUEN03; BEND06].

Para poder modelar una colisión, y siguiendo el modelo de numerosos autores [BARA92; REDO04a; BERG99a; MIRT98a; SCHM04], fue necesario adoptar previamente un modelo de movimiento para los objetos. Utilizando el modelo de Mirtich [MIRT96], asociamos a estos, dos vectores, uno que describe su traslación (vector velocidad lineal) y otro que describe su rotación alrededor de un eje (vector velocidad angular). Stephane Redon [REDO04a], hace una taxonomía de los métodos de *detección de colisiones* en dos grandes categorías de acuerdo a como manejan la interpenetración de objetos. Los modelos llamados *discretos*, obtienen las posiciones de los objetos móviles en instantes discretos de tiempo y reportan únicamente la interpenetración entre ellos. Por otro lado, los modelos de *detección continua* (o CCD por Continuous Collision Detection) garantizan que durante toda la simulación no sucedan interpenetraciones entre objetos.

Los métodos más comunes de simulación física [ERLE05a; ERLE05b], utilizan la información dada por los modelos discretos o continuos para determinar nuevos vectores de movimiento para aquellos objetos que han colisionado o se encuentran en contacto. Existen básicamente tres formas de abordar el problema: los métodos de penalización [MOOR88; BARZ96; HAZE03], los métodos impulsivos [MIRT96] y los métodos matriciales de restricciones [BARA92; BARA89; BARA96; STEW97; ANIT96].

2.5.2 Sistemas discretos de detección de colisiones

Los sistemas de detección discretos [BARA92; BERG99a; BERG99b; BERG01], se basan en determinar si existe o no interpenetración entre objetos en movimiento, luego de haberlos movido durante un cierto lapso de tiempo respetando el modelo de movimiento elegido. Según el modelo de vectores, la posición y orientación final de un objeto móvil están dadas por:

$$Posición \; + = v * \Delta t$$
$$Orientación \; + = w * \Delta t$$

siendo Δt el lapso de tiempo elegido y *v, w* sus vectores velocidad lineal y angular. Como resultado, el sistema actualizará en forma instantánea la posición y orientación de cada objeto móvil, sin considerar que algunos de ellos puedan quedar en una situación de interpenetración.

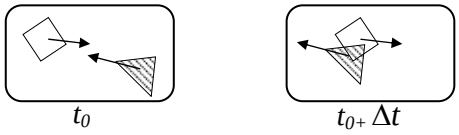


Figura 2.17 *Interpenetración como consecuencia de hacer avanzar el sistema durante un lapso de tiempo Δt. Los sistemas discretos de detección de colisiones consideran la interpenetración una vez ocurridas, declarándolas como colisiones, y resolviéndolas en una etapa posterior mediante backtracking.*

Debido a la naturaleza de paso discreto impuesta por el lapso de tiempo, el momento exacto en el que un choque sucede, muy probablemente sea durante el transcurso de ese tiempo y no en su principio o fin. Como consecuencia, al ser un sistema que detecta las posiciones de los objetos en los extremos de lis pasos, debe realizarse un procedimiento de búsqueda para encontrar el momento exacto en que ha ocurrido una colisión.

Una limitación importante del método es que si en dos pasos consecutivos no se detecta colisión, se supone que tampoco la hubo en todo el intervalo de tiempo. Así, dos objetos con elevada velocidad relativa podrían atravesarse sin que el sistema detecte su colisión.

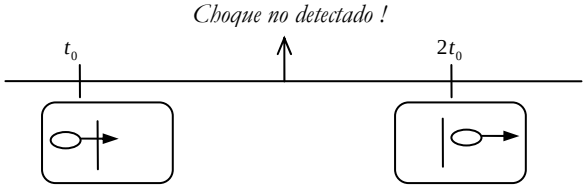


Figura 2.18 *Una situación problemática en sistemas de detección discretos. El lapso de tiempo fijo t_0 utilizado para determinar la posición de los objetos durante la simulación, trae como consecuencia la imposibilidad de detectar la colisión entre pares de objetos que se acercan a cierta velocidad. Para el sistema no ha ocurrido ninguna interpenetración y por lo tanto no es capaz de detectar la colisión asociada (Efecto "tunneling")*

La interpenetración detectada entre dos objetos, entonces, necesita ser resuelta. Esto se fundamenta en que, en general, este tipo de situaciones es muy notoria al observador generando resultados visuales no deseados. Por otro lado, es incoherente al modelo de cuerpos rígidos que se generalmente se adopta junto con estos métodos. La idea es entonces encontrar el tiempo en que ha ocurrido el impacto antes de la interpenetración.

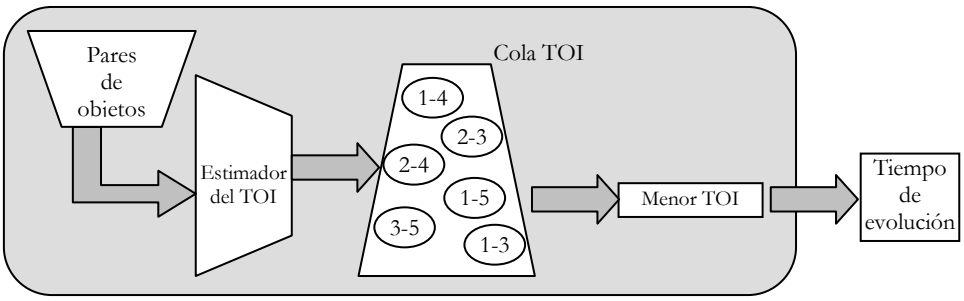
Para detectar el *tiempo de impacto*, los métodos discretos utilizan la estrategia de backtracking. Esta estrategia consiste en encontrar este tiempo utilizando recursión. Asumiendo que en el lapso actual de tiempo es $[t_n, t_{n+1}]$ y que en t_{n+1} se ha detectado interpenetración entre 2 objetos, se estima el tiempo de colisión t_c por métodos de interpolación (por ejemplo se toma $t_c = (t_n + t_{n+1})/2$). Luego, la posición de los objetos de todo el sistema es calculada nuevamente para t_c y la verificación por interpenetración se realiza nuevamente. Dependiendo si algún objeto interpenetra o no, el algoritmo decide si el primer impacto ocurrió en $[t_n, t_e]$ o en $[t_e, t_{n+1}]$ repitiendo el proceso en el nuevo intervalo. Este proceso termina cuando la cantidad de interpenetración es menor a un cierto valor (en el caso en que esta medida se pueda realizar) o cuando el intervalo de tiempo es menor a un cierto valor.

Si bien esos métodos de backtracking pueden acarrear un costo computacional elevado cuando los objetos móviles son complejos o cuando la interpenetración es profunda, son aplicados únicamente cuando se ha detectado interpenetración. Mientras no se detecte interpenetración, los objetos pueden moverse libremente (en cada paso de tiempo) en el entorno geométrico.

Cabe destacar que algunos métodos de simulación física, como los métodos de penalización, no requieren la utilización de backtracking para obtener el *tiempo de impacto*. Estos métodos utilizan la distancia de interpenetración detectada entre los objetos para modelar una reacción adecuada. Sin embargo, pueden llevar a un movimiento oscilatorio y hasta resonante en determinadas configuraciones [GREG00].

2.5.3 Sistemas continuos de detección de colisiones

Los sistemas de detección continua de colisiones garantizan en todo momento la no interpenetración entre objetos, eliminando el paso discreto que gobierna su movimiento al estimar conservativamente [MIRT96; MIRT98a] o exactamente [REDO02; REDO04a; REDO04b], el tiempo del primer impacto (llamado *TOI*) entre cada par de objetos que puedan colisionar en un sistema. Utilizando una *cola de prioridad* en los tiempos estimados, el algoritmo es capaz de determinar el tiempo de evolución de todo el sistema en el que se garantice la no interpenetración entre cualquier objeto durante todo el período. Luego de transcurrido ese tiempo, se calcula la distancia entre el par de objetos que lo determinaron. Si esa distancia es menor que un cierto valor, las rutinas de respuesta son aplicadas y un nuevo tiempo estimado de impacto entre ellos y todos los objetos involucrados con ellos son recalculados en la *cola de prioridad*. Luego el sistema avanza de acuerdo al nuevo *tiempo de impacto* indicado en el tope de la cola.



Estos sistemas necesitan un estimador de la distancia entre los objetos involucrados en una posible colisión. Mirtich trata el problema con algoritmos sobre objetos poliédricos

Figura 2.19 *Diagrama general de los sistemas de detección continua. Para los pares de objetos cercanos se calcula su tiempo de impacto mediante funciones de estimación. Los tiempos se mantienen en una cola de prioridad de la cual se obtiene el próximo tiempo en el que el sistema puede evolucionar sin riesgo de interpenetraciones.*

convexos utilizando V-clip [MIRT98b] aunque otros autores indican que el algoritmo *GJK* es más eficiente [CAME97; BERG99a]. En su algoritmo, Mirtich utiliza también V-Clip entre pares de objetos para realizar un estimativo de su de impacto, *TOI*, utilizando consideraciones dinámicas (fuerza de la gravedad, aceleración y velocidades actuales de los objetos). Este estimativo puede hacer que, una vez llegado al *tiempo de impacto* predicho, los objetos involucrados aún se encuentren a una distancia demasiado lejana como para aplicar funciones de respuesta. En estos casos el *tiempo de impacto* entre ellos se vuelve a calcular y a insertar en el *heap*. Debido a que este recálculo se hace con una cercanía mayor de los objetos involucrados, el nuevo *tiempo de impacto* encontrado será más cercano a su valor exacto. Mirtich muestra que sus funciones de estimación son suficientemente buenas como para que la cantidad de iteraciones necesarias para obtener el tiempo en un acercamiento complejo (con velocidad angular entre los objetos involucrados), sea baja.

Por su parte, Redon [REDO02] trabaja directamente con la geometría (vértices y aristas) de los objetos involucrados para obtener el tiempo exacto de colisión entre ellos evitando así el carácter aproximativo de Mirtich. Utilizando características dinámicas de los objetos y asumiendo que su velocidad angular es suficientemente baja (como para que en el paso de tiempo estimado, el desplazamiento angular no involucre un giro de más de 180 grados) su algoritmo utiliza métodos de interpolación (Newton-Rapson) para encontrar el tiempo exacto de colisión entre los pares de objetos obteniendo el *tiempo de impacto* posible entre cada arista y entre cada vértice y cara de ellos. Este algoritmo reporta también los puntos de contacto y distancia entre los objetos por lo que evita también el uso de algoritmos complejos como GJK o V-Clip para este propósito.

Para obtener un funcionamiento eficiente, estos algoritmos deben ser usados junto con otros que permitan evitar el cálculo costoso del *tiempo de impacto* entre cada par de objetos que componen un sistema. Este cálculo es de orden cuadrático en el número de objetos. Para evitarlo se utilizan métodos de partición espacial como *BSP* u *Octrees*, como también rutinas que utilizan la dinámica de los objetos como son *Sweep & Prune* [BARA92] o tablas de hash jerárquico [MIRT96; BERG00]. Otras técnicas de agrupamiento jerárquico en la geometría de objetos complejos también son comúnmente utilizadas [GOTT96; KLOS98a; BERG97].

2.5.4 Métodos de simulación física

Una vez detectada la colisión entre 2 objetos, será necesario actuar sobre ellos de modo que su movimiento posterior no haga que los objetos se interpenetren. En forma general, los sistemas de simulación física buscan encontrar nuevas vectores velocidad que determinen una separación de los puntos de colisión en el movimiento futuro de los objetos [BARA92; MIRT96]. La forma de encontrar estos vectores determinan los distintos métodos de simulación.

Los *métodos de penalización* son comúnmente utilizados con sistemas de detección discretos, en los que la *detección de colisiones* se relaciona directamente con una interpenetración de los objetos colisionantes. El método se basa en penalizar esa interpenetración ubicando un resorte, adecuadamente dimensionado, entre los puntos de penetración máxima, que tiende a eliminarla en instantes posteriores:

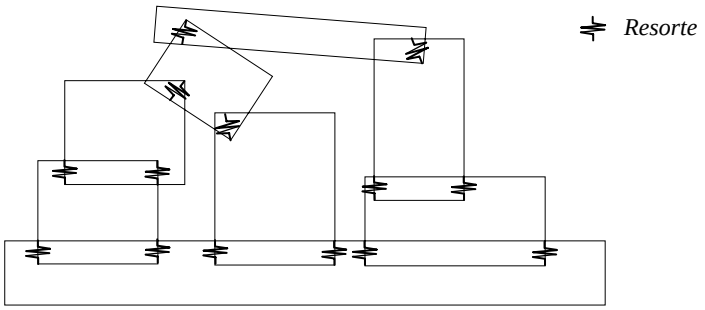


Figura 2.20 *Método de penalización: Cada interpenetración detectada se penaliza con un resorte adecuadamente dimensionado.*

Utilizando estos resortes, que se traducen en fuerzas aplicadas sobre puntos específicos de los objetos, se integran sus ecuaciones de movimiento (Ver apéndice B7), para obtener la nueva ubicación de cada uno de ellos en el siguiente paso discreto de tiempo. La principal ventaja de este método es la relación lineal que aparece entre cada una de las colisiones detectadas y el algoritmo de resolución de estas. Para k interpenetraciones, la respuesta toma un tiempo del orden O(k) con constantes lineales muy bajas [EBER04b]. Su desventaja es la dificultad en el dimensionado de los resortes asociados con las interpenetraciones y en la aparición de situaciones ecuaciones diferenciales con características resonantes o de costosa integración numérica (ver [EBER04b], puntos 5.5 y 5.6).

Los *métodos impulsivos,* que aparecen como tal en la tesis doctoral de Brian Mirtich [MIRT96], se basan en simular toda interacción física a través de impulsos de colisión. Este método es comúnmente utilizado con sistemas continuos de *detección de colisiones*. Los impulsos son aplicados en los puntos más cercanos entre los objetos colisionantes y las situaciones de contacto permanente (o micro colisiones) son simuladas con series de impulsos de colisión ocurriendo a una frecuencia elevada.

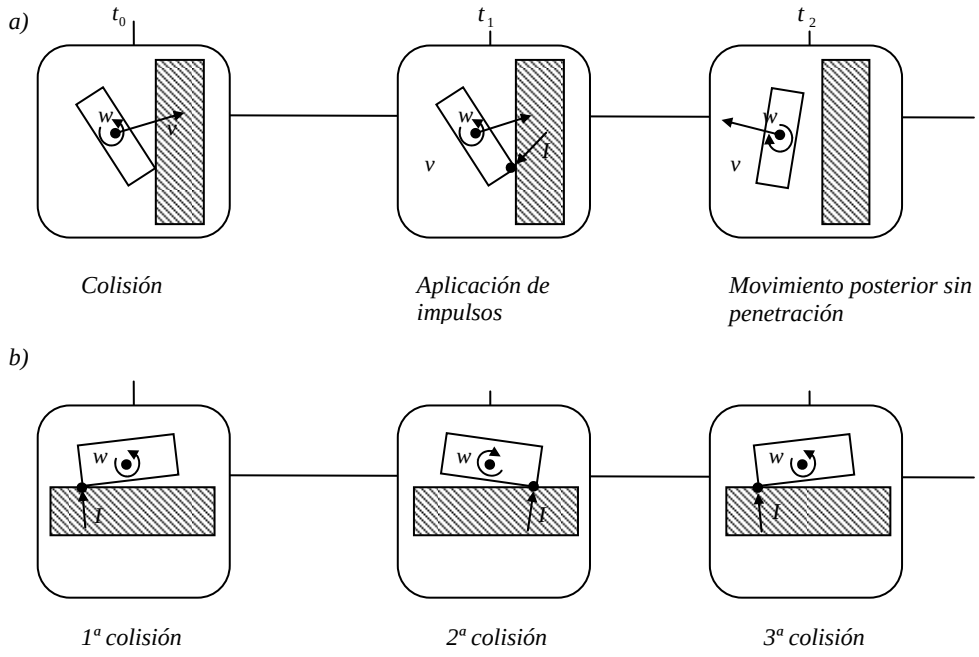


Figura 2.21 *Método impulsivo: A cada colisión detectada se aplica un impulso que determina su separación en instantes posteriores. En (b) una situación de contacto permanente (contacto estático o micro colisión) se simula mediante una serie de impulsos ocurriendo a una frecuencia elevada.*

La eficiencia de los métodos impulsivos depende en gran medida de la exactitud del método de detección continua que se elija. El método estimativo originalmente propuesto por Mirtich resulta eficaz en situaciones de colisión dinámica, como por ejemplo cuando dos objetos chocan en el aire. Sin embargo, situaciones extremas de contacto estático, como por ejemplo un conjunto de bloques apilado, requiere del sistema un numero elevado de iteraciones para resolver las múltiples micro colisiones que aparecen entre los objetos. Numerosas alternativas han surgido para solucionar estos problemas [REDO04a; GUEN03] haciendo este método eficiente incluso en estas situaciones. Otra desventaja radica en la dificultad de simular contactos permanentes entre objetos (por ejemplo una bisagra) ya que la idea principal de estos métodos es aplicar impulsos para separar objetos y no para acercarlos. Sin embargo Bender [BEND05; BEND06], aborda este problema con resultados satisfactorios. Una versión simple de este método es la que se desarrolló en este proyecto.

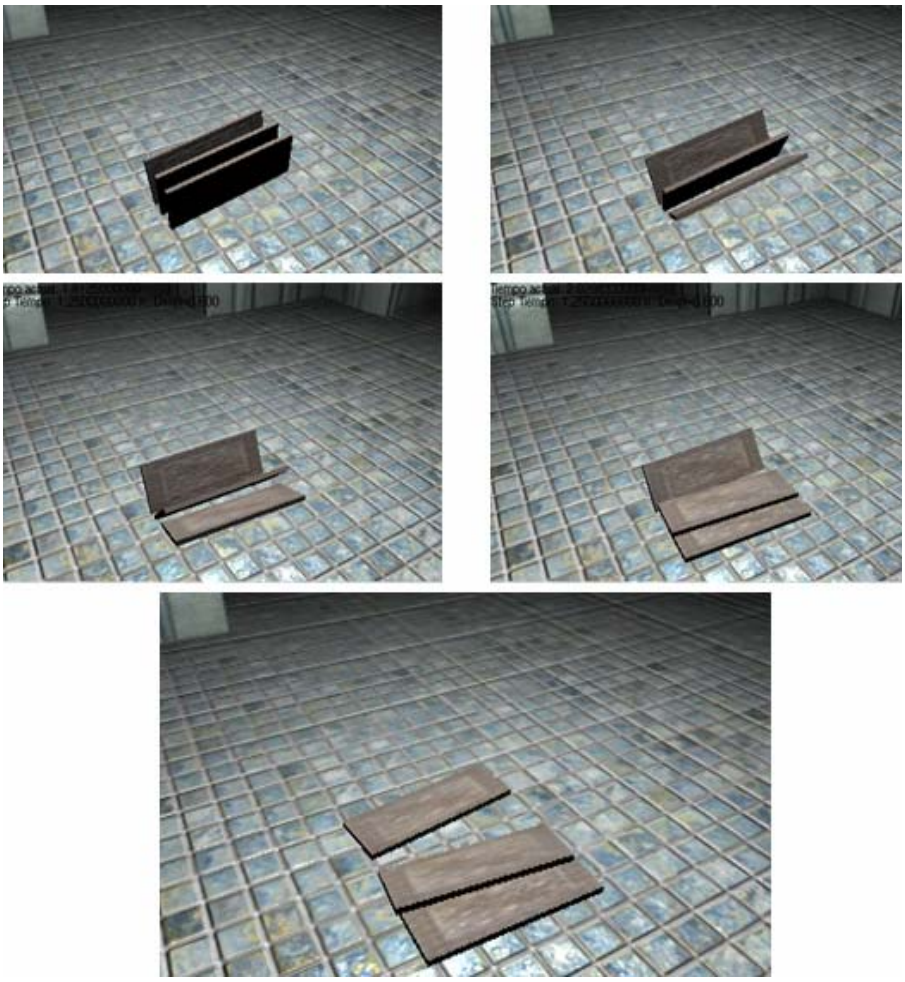


Figura 2.22 *Fing*ER *simulando la caída de objetos por efecto de la gravedad. La implementación de algoritmos de física ofrece resultados realistas.*

Los *métodos de restricciones*, originalmente introducidos por Baraff [BARA92] tratan de resolver específicamente el problema de múltiples contactos estáticos que aparece en una situación de equilibrio, utilizando métodos impulsivos para resolver colisiones dinámicas. Este problema resulta en la resolución de un problema de complementariedad lineal (o LCP, [MURT97]) que puede o no tener solución [BARA93]. Debido a esto, algunos autores [ANIT96; STEW97] han buscado representaciones alternativas del problema que garanticen la existencia de soluciones e incluso se ha encontrado la forma de representar cualquier situación del sistema (no solamente situaciones de equilibrio estático) mediante este tipo de formulación [ERLE05a]. Las ventajas de este método radican en la posibilidad de modelar contactos continuos entre objetos (como bisagras, contactos esféricos o contactos rígidos) y la posibilidad de paralelización debido a la representación inherentemente matricial del problema como un LCP. La desventaja radica en su compleja implementación y en el tiempo $O(k^3)$ necesario para resolver k contactos. Los últimos desarrollos, sin embargo, logran resolver el problema en forma aproximativa lo que permite obtener rápidas soluciones, pero imperfectas, en tiempos lineales. Esto trae como consecuencia lo que se denomina movimiento plausible que, para sistemas de animación en tiempo real, resultan suficientemente convincentes [CATT05].

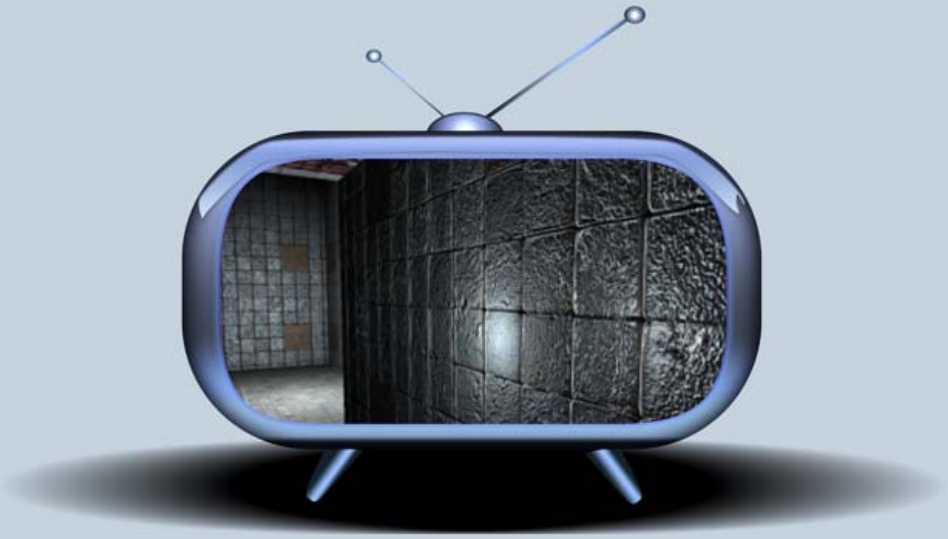
2.6 Resumen: clasificación de motores 3D

Hemos introducido los conceptos fundamentales que guían el diseño de los motores gráficos 3D en tiempo real. El funcionamiento de cada uno sus módulos puede ser optimizado mediante la implementación de diversas técnicas destinadas a acelerar una parte del funcionamiento general del motor: grafos de escena y optimización de datos basada en la posición del observador, modelado de efectos y materiales usando lenguajes de *sombreado* y optimización de chequeo y cálculo de colisiones para generar animaciones físicamente realistas.

Si bien todas las técnicas descritas podrían aplicarse en forma simultánea, llegar a este grado de flexibilidad trae problemas a la hora de lograr un diseño eficiente. Es por ello que la gran mayoría de motores 3D se centran en la implementación y aplicación de algunas de estas técnicas, definiendo así distintos tipos de motores. De esta manera, existen motores 3D basados en celdas y portales para ambientes cerrados, basados en *octrees* o en jerarquías de volúmenes acotantes para escenas abiertas, basados en horizontes de oclusión para escenas donde el observador está a nivel del piso, con o sin *LOD*, que utilizan oclusión por objetos, basados en modelos de iluminación por píxel, por vértice, con o sin *detección de colisiones*, con o sin simulación física, etc.

3

Implementación: Arquitectura General



3.1 Introducción

El trabajo realizado en este proyecto consistió en la creación de un motor 3D que optimizara su funcionamiento con escenas compuestas por geometría altamente ocluyente. Asimismo se proyectó implementar un sistema eficiente de *detección de colisiones* y respuesta física realista.

Para lograr estos objetivos, se construyó un *grafo de escena* basado en *árboles BSP* traducidos a partir de los mapas generados por el editor GTK Radiant [GTKR00], de uso libre, con el cual se pueden definir fácilmente los detalles de entornos arquitectónicos cerrados. En este contexto, se implementó un sistema de traducción que permite traducir los mapas generados a un formato propio más adecuado al diseño del motor 3D implementado.

En cuanto a la *detección* y *respuesta a colisiones*, el sistema utilizado en este proyecto se basó en la implementación de las rutinas estimativas de Mirtich descritas en su tesis doctoral [MIRT96] junto con la utilización de la partición en celdas y portales descrita en el capítulo 5.4 y el algoritmo de *Sweep & Prune* descrito en [BARA92]. Se optó, de esta forma, por la implementación de un sistema de *detección de colisiones* continua.

Se trabajó también, en la creación de un sistema que le permitiera al desarrollador crear aplicaciones fácilmente. Para ello se diseñó una API que da acceso a todas las funcionalidades del motor y una arquitectura que permite extender sus posibilidades para agregar nuevos efectos visuales e incluso nuevos algoritmos de optimización.

A continuación se presenta la arquitectura implementada y se da una breve introducción al diseño y funcionamiento de cada módulo, con el objetivo de dar una visión general del funcionamiento del motor.

3.2 Arquitectura

Dentro de los módulos generales de un motor 3D, descritos en el capítulo dos, se implementaron los relacionados con la generación de gráficos y animación en tiempo real:

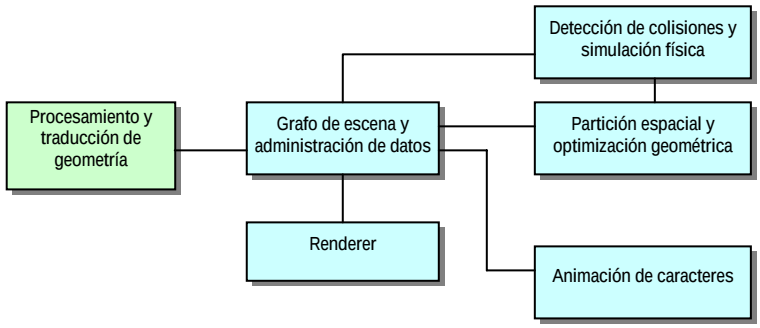


Figura 3.1 *Módulos de la arquitectura general de un motor 3D involucrados en la generación de gráficos en tiempo real.*

En nuestro proyecto, estos módulos tienen la siguiente estructura interna:

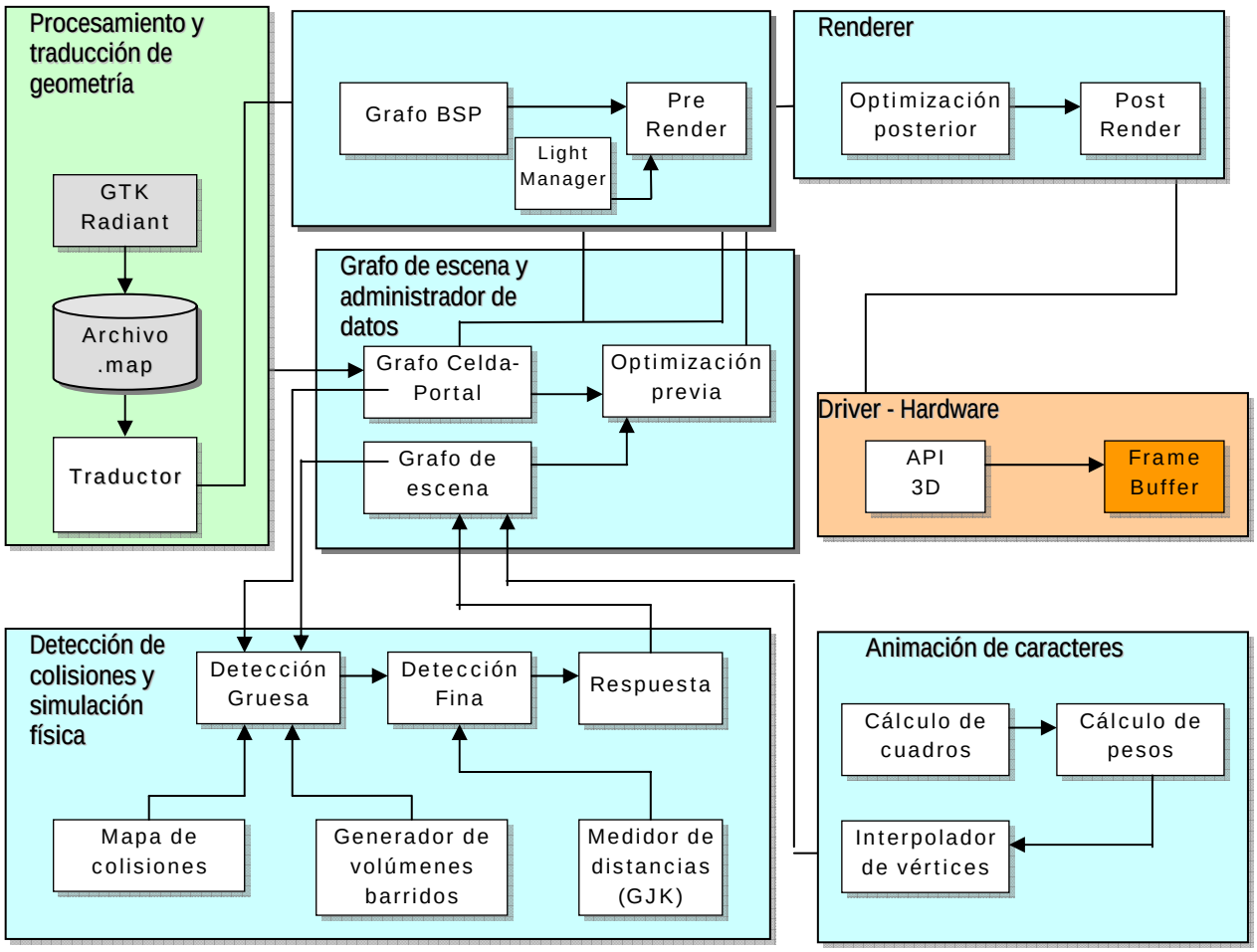


Figura 3.2 *Módulos de generación de gráficos en FingER. Se incluyen aquí las herramientas para generar la geometría fija.*

3.3 Descripción de funciones

3.3.1 Procesamiento y traducción de geometría

Dada la gran variedad de escenas 3D posibles, la primera decisión que se debió tomar fue elegir el algoritmo de optimización geométrica que se iba a implementar. En esta decisión, el principal factor que influyó fue la disponibilidad de datos de prueba. Las escenas formadas por ambientes altamente ocluyentes han sido utilizadas en muchos de los videojuegos más populares, existiendo abundante información al respecto y, sobre todo, muchos editores de escenas orientados a este tipo de datos. Dentro de los editores disponibles, el GTK Radiant [GTKR00] es uno de los que más desarrollo ha tenido, existiendo en la Web muchas escenas construidas por entusiastas de la comunidad. [Q3WO07].Se decidió, por lo tanto, utilizar esta información para implementar el algoritmo *BSP* de optimización geométrica descrito en el capítulo dos.

La segunda decisión que guió el diseño del motor fue la de soportar tecnologías de última generación para la creación de efectos visuales. Básicamente, la posibilidad de incluir programas escritos en lenguajes de *sombreado* que permitieran implementar modelos avanzados de iluminación, en particular aquellos que incluyen reflexiones especulares.

Lamentablemente estas dos decisiones colisionaron, debido a que el GTKRadiant no incluye la posibilidad de definir escenas que contengan elementos conteniendo programas de *sombreado*. Fue por esta razón que se tuvo que implementar un sistema de traducción que permitiera dividir la tarea de descripción de escenas en dos etapas:

- Datos geométricos del entorno.
- Luces, objetos móviles y objetos con animación de caracteres.

Los datos geométricos del entorno se construyen con el GTK Radiant, mientras que el resto de los elementos de la escena se construyen a partir del *grafo de escena* mediante la API provista por el motor.

El proceso de creación y traducción de los datos geométricos del entorno involucra a los siguientes módulos:

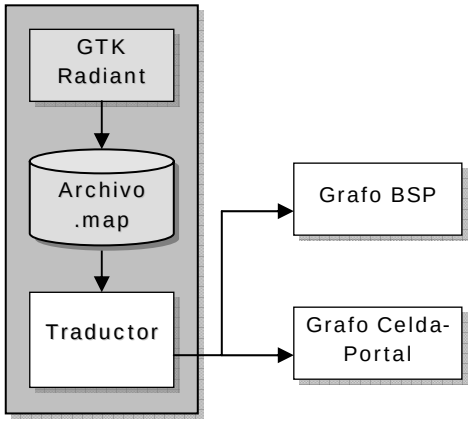


Figura 3.3 *Traducción de datos geométricos generados con el GTKRadiant*

- **GTK Radiant y Archivo .map:** Comprende al editor de niveles descrito y a los datos que genera. El formato .map es propietario de [IDSO03]
- **Traductor:** Comprende el módulo que, a partir de los datos creados en el editor, genera datos optimizados aplicando el algoritmo *BSP* explicado en el capítulo dos. La información traducida genera dos clases de datos:
    - **Grafo BSP:** Es la estructura que contiene el *árbol BSP* y que brinda los servicios de búsqueda en dicha estructura.
    - **Grafo celda-portal:** Es la estructura que contiene la información sobre la interconexión de las celdas a través de portales.

3.3.2 Grafo de escena y administrador de datos

Como se dijo en el punto anterior, los datos que el motor 3D debe procesar en todo momento están compuestos por dos conjuntos de datos: el entorno creado por el GTK Radiant y los objetos móviles y con animación creados a partir de un *grafo de escena* construido a partir de la API del motor.

El formato de cada uno de estos conjuntos, y especialmente el *grafo de escena*, fueron diseñados pensando en el usuario, teniendo en cuenta que el programador de aplicaciones necesita una manera sencilla de indicarle al motor cuales son los objetos que quiere poner en la escena. Este formato, detallado en el capítulo cuatro, no es compatible con los formatos que las tarjetas de video esperan recibir, por lo que se decidió realizar una optimización previa para transformarlos en datos que puedan ser rápidamente dibujados.

Esta tarea se realiza en el módulo denominado **Optimización Previa** el cual está acoplado a los siguientes otros módulos del motor:

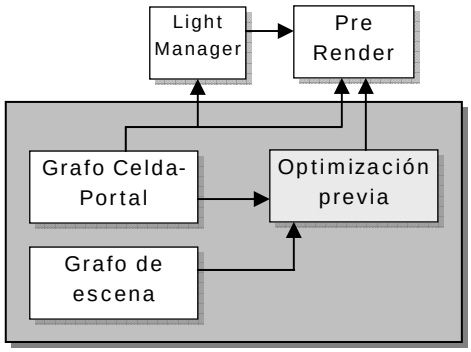


Figura 3.4 *Grafo de escena en el motor 3D.*

Como puede apreciarse en la figura 3.4, la etapa de optimización previa procesa todos los datos de la escena: los creados con el editor y los pertenecientes al *grafo de escena*. Su producto es un conjunto con la misma información, optimizado según los criterios que se detallan en el capítulo cuatro.

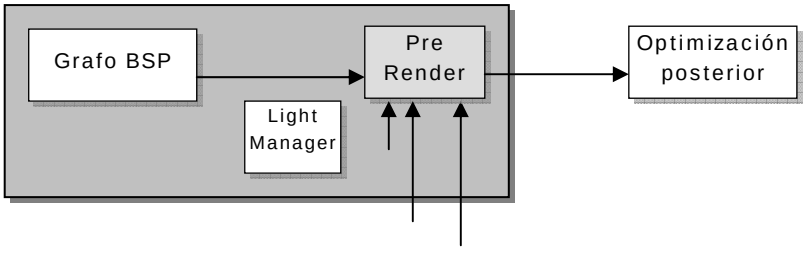
Dado que el algoritmo de optimización debe trabajar con un conjunto potencialmente grande de datos, este proceso se realiza en la etapa de inicialización del motor, por lo que, una vez realizado, no se vuelven a procesar nuevamente. Esto tiene como consecuencia inmediata el hecho de que todos los objetos de la escena deben ser cargados en memoria antes que el motor comience a funcionar. Esta estrategia, denominada *precaching,* no plantea problemas significativos si se dispone de memoria RAM suficiente para contener todos los datos de la escena.

3.3.3 Partición espacial y optimización geométrica

Una de las claves para aumentar el rendimiento del motor 3D consiste en optimizar los datos que éste debe procesar. Este problema tiene varias aproximaciones, una de las cuales consiste en minimizar la cantidad de polígonos que el motor debe dibujar, utilizando para ello alguna de las técnicas de agrupación espacial descritas en el capítulo dos.

En este proyecto se decidió la implementar el algoritmo *BSP* junto con un *grafo celda-portal* para que, a partir de la posición del observador, se pueda determinar rápidamente cuales son las celdas visibles.

Esta tarea es realizada por el módulo denominado **Pre Render**, que está acoplado a los siguientes módulos del motor:



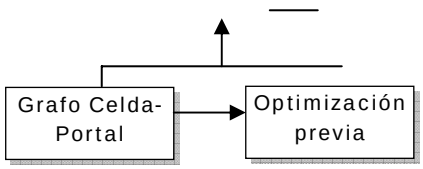


Figura 3.5 *Optimización geométrica en el motor 3D.*

Su tarea consiste en utilizar los servicios del grafo *BSP* para determinar, en tiempo real, en qué celda está ubicado el observador. Una vez determinada esta celda, se procede a consultar al *grafo celda-portal* para determinar cuales son las celdas adyacentes que deben ser dibujadas. Solo aquellas celdas adyacentes cuyos portales se proyecten en el plano de visión del observador serán procesadas. Este ciclo continúa en forma recursiva. Los detalles de implementación y algoritmos utilizados se detallan en el capítulo seis.

El módulo denominado **Light Manager** se encarga de realizar los test de *descarte (culling)* para optimizar los cálculos relacionados con las luces que afectan a los objetos. En los apéndices A3, A4 y A5 se detallan las características de estos cálculos y se justifica la creación de un módulo que optimice su aplicación.

3.3.4 Renderer

Una vez que el módulo de partición espacial determina el conjunto de celdas visibles, esta información pasa al módulo encargado de dibujarla, denominado r*enderer*, cuyo diseño es:

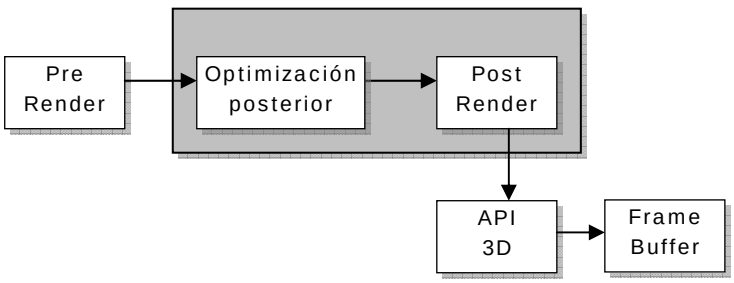


Figura 3.6 *Etapa final: Renderer.*

El estudio del funcionamiento y las características del hardware actualmente disponible permitió diseñar un módulo capaz de realizar una última optimización de los datos que son efectivamente dibujados. El módulo denominado **Optimización Posterior** es el encargado de realizar esta tarea. Como puede observarse en la figura 3.5, éste modulo trabaja directamente con los datos optimizados por el módulo de optimización geométrica, no con los de toda la escena. Este hecho permitió diseñar un algoritmo de optimización rápido, que es ejecutado cada vez que el motor debe dibujar la escena, o sea, en cada *cuadro* generado. Los detalles de este algoritmo se describen en el capítulo cuatro y en el apéndice A1.

La última etapa del pipeline gráfico implementado se encarga de enviar los datos a dibujar al *driver* de video por medio de la API 3D utilizada por el motor. La arquitectura implementada incluye una capa de abstracción que permite utilizar cualquier API: OpenGL o Direct3D. Esta capa la denominamos "Post Render".

3.3.5 Animación de caracteres

El *grafo de escena* implementado permite definir la posición de los distintos objetos geométricos en forma jerárquica. De esta manera, un cambio en la posición de un objeto afectará directamente a todos aquellos que estén subordinados a él. Esta característica hace que el motor esté especialmente apto para crear modelos articulados, donde la posición de una parte depende directamente de la de su padre.

Para que la animación de este tipo de modelos no produzca deformaciones visibles en las articulaciones, existen varias técnicas, una de las cuales, denominada *Vertex Skinning* fue la que se implementó en este proyecto. Para realizar pruebas se implementó un módulo capaz de cargar modelos articulados en formato MD5 que fuera diseñado por *id Software* para ser utilizado con el *Doom 3 engine*. Los detalles de la implementación se describen en el apéndice A5.

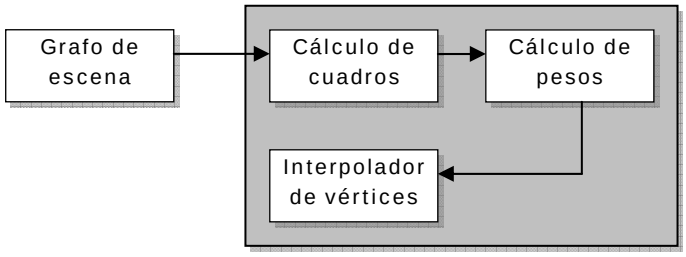


Figura 3.7 *Módulos relacionados con la animación de caracteres.*

Los datos del objeto geométrico articulado, cargado previamente en el *grafo de escena*, son procesados por el módulo de animación de caracteres que se encarga generar la animación siguiendo los siguientes pasos:

- **Cálculo de Cuadros:** Dado que las animaciones MD5 deben mostrarse a una velocidad determinada por el diseñador, el módulo de cálculo de *cuadros* se encarga de determinar, en función del tiempo real transcurrido, cual es el próximo *cuadro* de animación que debe ser procesado.
- **Cálculo de pesos:** Para poder interpolar las posiciones de los vértices entre el *cuadro* anterior y el actual, el algoritmo de *Vertex Skinning* necesita calcular cual es la influencia o **peso** relativo de ambas posiciones en cada vértice.
- **Interpolador de Vértices:** Es el módulo que genera la nueva posición para cada vértice del modelo, realizando una interpolación donde se tienen en cuenta los vértices del *cuadro* actual, los del próximo y los pesos de cada uno.

3.3.6 Detección de colisiones y simulación física.

Este conjunto de módulos conforma una de las áreas de mayor importancia dentro del motor, pues tiene influencia sobre todos los objetos. Su tarea es detectar cada colisión que pueda surgir durante la simulación y modificar las velocidades de los objetos aplicando leyes de la física para simular una reacción realista.

Para lograr esto, este *sistema de colisiones*, trata a todos los objetos que componen la escena (paredes, objetos fijos y objetos móviles) como objetos convexos independientes. La tarea de detectar y responder a colisiones entre cada uno de ellos requeriría un tiempo cuadrático en el numero de objetos. Es por ello que se diseñó una arquitectura que permitiera optimizar este trabajo.

La solución propuesta consta de los siguientes módulos:

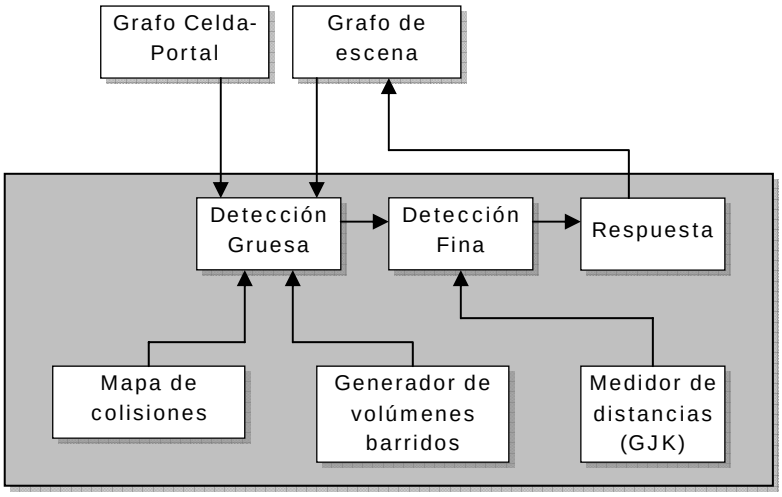


Figura 3.8 *Sistema de detección y respuesta a colisiones*

El proceso de detección y respuesta, detallado en el capítulo cinco, sigue los siguientes pasos:

- **Detección Gruesa:** Esta etapa se encarga de restringir el chequeo de colisiones sólo a aquellos objetos que se encuentran cerca. Para ello utiliza la partición en celdas y portales descartando la verificación de colisiones entre objetos que no estén en celdas comunes. Esta idea se aplica también para los objetos dentro de una misma celda. Utilizando una estructura denominada mapa de colisiones**,** se logra descartar la verificación de colisiones entre objetos lejanos interiores a una celda.

- **Detección Fina:** Aquellos pares de objetos cuya cercanía indique que se encuentran en colisión potencial se procesan en el módulo de detección fina, el cual aplica un algoritmo medidor de distancias con el que se obtiene la distancia y los puntos más cercanos.

- **Respuesta:** Los objetos que se encuentren a una distancia menor que un cierto valor se consideran en situación de colisión. El módulo de respuesta se encarga de modificar las características cinéticas de los objetos en función de sus propiedades físicas para garantizar la no interpenetración en el instante posterior a la colisión.

3.4 Funcionamiento general

Como se vio en el capítulo 2, los motores 3D están diseñados para generar imágenes en forma continua siguiendo el siguiente esquema:

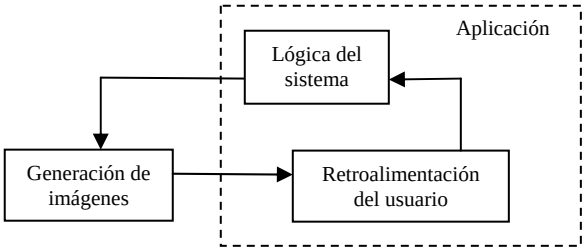


Figura 3.9 *Loop principal del motor 3D*

Los módulos denominados **Lógica del Sistema** y **Retroalimentación del usuario** forman parte de la **Aplicación** que utiliza los servicios del motor para visualizar sus imágenes. El *loop* que plantea la figura 3.8 implica que éste debe acoplarse profundamente con la aplicación a la que brinda sus servicios por lo que ésta debe estar diseñada para funcionar en este contexto.

Es importante observar que la retroalimentación del usuario no sucede en forma sincronizada con el loop principal del motor, y que, en general, esta retroalimentación implica cambios a nivel del *grafo de escena* que procesa el motor (movimiento de un objeto, activación de una animación de caracteres, etc.).

Estas dos características llevaron a diseñar un sistema eventos que son llamados en momentos específicos del *loop* de generación de imágenes y que le permite al desarrollador de aplicaciones tener control sobre todos los elementos que forman parte de las escenas. El apéndice A6 detalla estos eventos así como el sistema de controladores que permiten expandir las posibilidades del motor implementado.

4

Grafo de Escena y Renderer

4.1 Introducción

Dentro de diseño general de un motor 3d, ya explicado en los capítulos 1 y 2, los módulos denominados *Grafo de escena y administrador de datos, Partición espacial y optimización Geométrica* y *Renderer* conforman un núcleo muy importante. Esto se debe al hecho de que son los módulos encargados de todas las tareas relacionadas con la administración y dibujo de datos y cuyo rendimiento es, en general, el cuello de botella en rendimiento general del motor 3D [ASHI06; SCHM07].

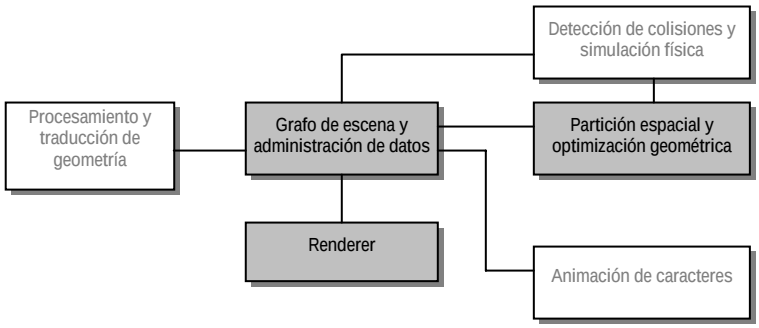


Figura 4.1 *Grafo de Escena, Renderer y Partición espacial ubicados dentro de la arquitectura general de un motor 3D.*

Sus funciones específicas se detallan en el capítulo anterior.

De estos módulos, el *renderer* merece especial atención debido a que su diseño está fuertemente condicionado por las características del hardware de soporte. Estas características son expuestas al usuario mediante las funciones que brindan las distintas API 3D, encargadas de servir de interfaz entre la aplicación y el *Frame Buffer*. El esquema general del funcionamiento de todos estos elementos conforman un *pipeline* como se ve en la siguiente figura:

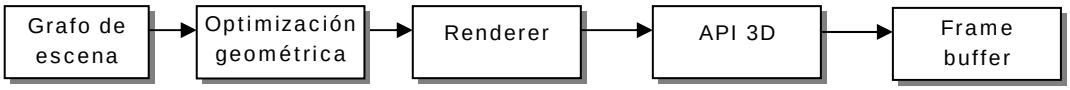


Figura 4.2 *Pipeline conformado entre el grafo de escena y el Frame Buffer*

En este capítulo se describen y justifican las distintas decisiones tomadas al diseñar el *grafo de escena* y el *renderer*. El módulo de optimización geométrica se detalla en el capítulo siguiente.

4.2 Algunas propiedades relevantes del hardware gráfico

Koji Ashida, en su disertación sobre optimización [ASHI06], analiza las diferentes características que tiene el diseño actual del hardware y realiza recomendaciones sobre medidas a tomar para maximizar su rendimiento.

De las técnicas descritas en ese artículo, las que se priorizaron se pueden agrupar en:

- Optimización en la transferencia de polígonos: Minimización de transferencia de vértices e indización de vértices.
- Uso del paralelismo en el procesamiento de píxeles: Minimización de cambios de estado.
- Minimización del uso del bus de datos de la CPU: técnica de *precaching*.

Es importante aclarar que el uso de estas técnicas es independiente del sistema de optimización geométrica que se utilice ya que éste se encarga de reducir el conjunto de datos, no de optimizar la estrategia utilizada para dibujarlos.

4.2.1 Minimización de transferencia de vértices

Los objetos geométricos de las escenas están compuestos por un conjunto de polígonos definidos a partir de vértices en el espacio. Cada vértice se define utilizando tres coordenadas: x,y,z por lo que debe ser proyectado para poder ser visualizado en una pantalla bidimensional. Las tarjetas de video actuales son capaces de realizar este trabajo, además de los cálculos simples de iluminación para cada vértice, en un proceso denominado *TnL* (Transformation and Lighting).

Actualmente todo el proceso de *TnL* se encuentra implementado por hardware en varias unidades de procesamiento que trabajan en paralelo, las cuales son capaces de transformar vértices a una velocidad muy superior a la lograda por las CPU estándar. La única condición que se debe cumplir es que los vértices se encuentren en la memoria de la tarjeta de video [ASHI06].

Como los datos de los vértices están originalmente en la memoria central, cuanto más rápida sea la transferencia hacia la memoria de video más rápido será luego su procesamiento.

Es así que una estrategia basada en la transferencia por vértice genera un *overhead* muy grande por lo que un programa de transferencia secuencial de vértices se considera inaceptable:

```
For i=1 to n
        glVertex x(i), y(i), z(i)
next i
```

Si los datos de los polígonos están almacenados en forma contigua se puede aplicar una estrategia de transferencia totalmente diferente. Las herramientas que las API actuales proveen permiten realizar transferencias de datos en bloque, por lo que con una sola llamada se pueden transferir miles de vértices a la memoria de video.

De esta manera se plantea una importante restricción de diseño:

Se deberá transferir a la memoria de video la mayor cantidad de polígonos en forma simultánea. Para ello, los vértices de los polígonos y se deben agrupar en buffers conteniendo la mayor cantidad de vértices posible.

4.2.2 Indización de vértices

Cuando un objeto es modelado utilizando polígonos es muy común que existan vértices compartidos por más de uno. La unidad geométrica básica de procesamiento de una tarjeta de video es el triángulo por lo que si queremos dibujar una malla como la mostrada en la figura 4.2 habrá que especificar cada uno de los 7 triángulos, definiendo para ello 3 vértices por triángulo.

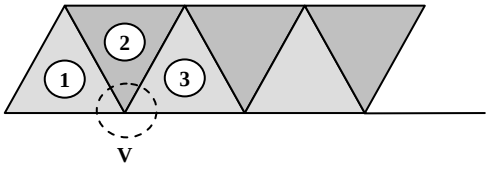


Figura 4.3 *Vértices compartidos: En esta malla el vértice V indicado es compartido por los triángulos 1, 2 y 3*

Según esta figura el vértice ***V*** deberá estar contenido en la definición de 3 triángulos, generando 9 coordenadas en total. Si cada una está codificada con un número flotante de 4 bytes entonces el total de memoria usada es de 36 bytes de datos.

Una forma de minimizar esta redundancia es numerar los vértices y construir los polígonos a partir de listas de referencias a ellos. De esta manera las 9 coordenadas se transforman en un vértice y 3 índices que serán las referencias de los 3 polígonos. Los índices se pueden codificar en 2 bytes, completando un total de 12+6 =18 bytes de datos.

Es importante aclarar que el hardware actual está optimizado para de procesar vértices agrupados de esta manera.

Este hecho plantea una "mejor manera" de almacenar los datos geométricos, conformando así una nueva restricción en el diseño del *renderer*:

Los polígonos deben almacenarse de manera de minimizar la cantidad de datos generados por la redundancia en las mallas. El modelo de indización de vértices se considera el más adecuado.

4.2.3 Minimización de cambios de estado

El diseño de las tarjetas de video actual aprovecha una propiedad muy importante de los datos gráficos: el contenido de un píxel no está relacionado con el de ningún otro. Este detalle hace las tarjetas actuales contengan una serie de *pipelines* que funcionan en paralelo procesando los píxeles. En ellos, la tolerancia a la latencia es muy superior que en las CPU, lo cual permite extender considerablemente sus etapas.

Si bien este diseño en paralelo permite aumentar la eficiencia de los pipelines existen situaciones en las que esta eficiencia trabaja en nuestra contra:

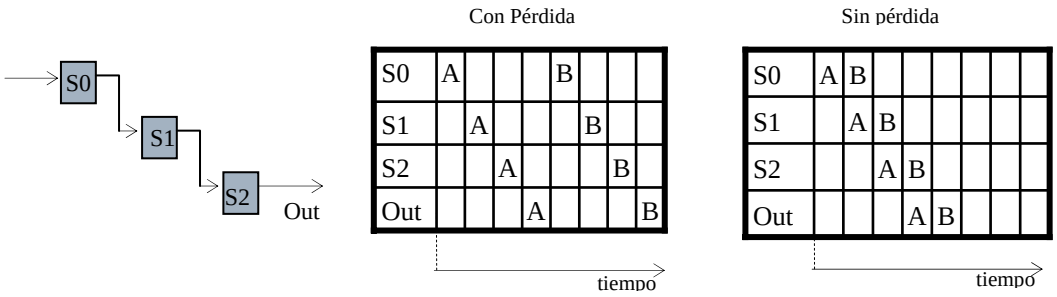


Figura 4.4 *Pipeline gráfico. Dos situaciones con pérdida y sin pérdida del paralelismo.*

Supongamos que el sistema debe dibujar 2 polígonos, A y B que se solapan.

- Si A y B son opacos entonces el sistema podrá procesarlos en paralelo ya que el aspecto de un píxel de la imagen final no dependerá de **ambos** polígonos. En este caso A y B entran a 2 *pipelines* diferentes y son procesados completamente en paralelo.
- Si A es opaco y B es transparente tendremos que el aspecto final del píxel dependerá de la superposición de ambos polígonos por lo que deberá procesarse primero A y luego B para aplicar la transparencia sobre A y obtener el resultado correcto.

Las API actuales funcionan como máquinas de estado en las que cada estado define un aspecto del aspecto final de un polígono. Es así que existen decenas de estados que se pueden definir: transparencia/opacidad, color/textura, cantidad de unidades de *multitextura*, *píxel shaders* , etc.

El estudio de la pérdida o ganancia de paralelismo en diferentes configuraciones de estados va más allá de nuestro trabajo pero el hecho que muchos de ellos no puedan aprovechar el paralelismo del hardware sugiere una nueva e importante restricción de diseño para el *renderer*:

Se debe minimizar la cantidad de cambios de estado agrupando toda la geometría posible que use un estado en particular.

4.2.4 Minimización del uso del bus de datos de la CPU.

Además de las reglas indicadas anteriormente existen ciertas optimizaciones que la experiencia ha ido indicando como fundamentales al programar aplicaciones que requieran alto rendimiento. Alguna de ellas son el tamaño de las estructuras, la alineación de los datos en la memoria y el evitar realizar llamadas al Sistema Operativo que involucren el uso del sistema de memoria virtual: básicamente evitar usar el *new()* en el código que se ejecuta dentro del loop principal del motor.

La minimización del uso del sistema de memoria virtual del SO plantea una restricción importante al programador de aplicaciones ya que, como veremos, el diseño del motor 3D exige que todos los objetos que serán visualizados estén previamente definidos y correctamente inicializados.

El proceso de precarga de las estructuras necesarias para que la aplicación funcione se denomina *precaching* y el motor 3D implementado provee un servicio especialmente diseñado para que el programador de aplicaciones realice estas tareas.

4.3 Arquitectura

Teniendo en cuenta todas las restricciones descritas en la sección anterior, se prestó especial atención en diseñar un *grafo de escena* que le permitiera al *renderer* seguirlas estrictamente, con el objetivo de minimizar su impacto en el rendimiento general del motor.

El siguiente diagrama muestra la arquitectura general de la solución propuesta.

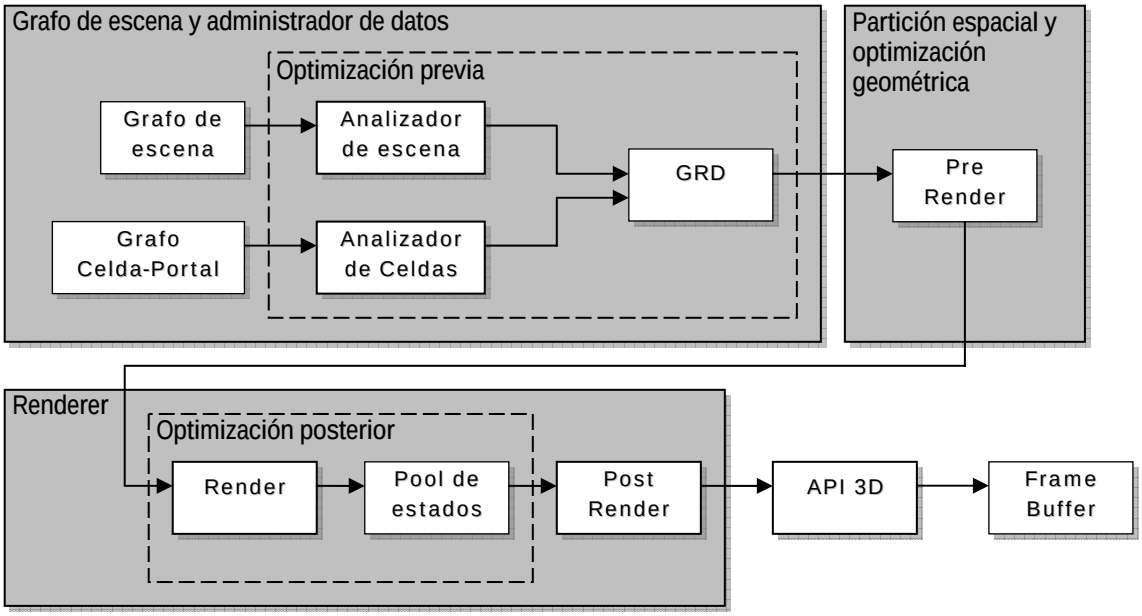


Figura 4.5 *Diagrama general de los módulos encargados de optimizar el funcionamiento del Renderer y del Grafo de escena.*

Como puede apreciarse, por comparación con la figura 3.2 del capítulo anterior, los módulos de Optimización previa y Optimización posterior fueron desarrollados especialmente. Esto se debió a la aplicación de *heurísticas* diseñadas especialmente para cumplir con las restricciones de diseño planteadas anteriormente.

Brevemente se puede describir la función de cada módulo, de la siguiente manera:

- **Grafo de escena:** Contiene los datos de los objetos geométricos que el *renderer* debe dibujar. Como veremos, estos datos no están en un formato óptimo por lo que deben ser procesados para poder ser dibujados por el *renderer*.
- **Grafo celda-portal**: Contiene la información sobre los objetos geométricos correspondientes al entorno arquitectónico de la escena a dibujar. Pese a que estos datos están agrupados en celdas y divididos por portales no pueden ser dibujados directamente por el *renderer*, por lo que deben también ser procesados para poder ser dibujados.
- **Analizador de escena:** Es el módulo que se encarga de procesar los datos del *grafo de escena* transformándolos a un formato eficiente para que el *renderer* pueda dibujarlos.
- **Analizador de celdas:** Cumple la misma función que el Analizador de escenas, pero con los datos del *grafo celda-portal*.

- **GlobalRenderData (GRD):** Es la estructura de datos optimizada que el *renderer* puede dibujar directamente. Se construye con los datos de salida de los dos analizadores.
- **Render:** Es el módulo encargado de dibujar directamente todo lo indicado en el GRD.
- **Pool de Estados:** es el módulo encargado de registrar todo el trabajo del módulo Render, agrupándolo de manera de optimizar su transferencia final.
- **PostRender:** Es el módulo que, utilizando los servicios de la API 3D, dibuja efectivamente los datos optimizados. Sirve de capa de abstracción para utilizar múltiples API.

4.4 Detalles del diseño

En esta sección se describen las principales decisiones que guiaron el diseño de cada uno de los módulos descritos anteriormente, haciendo especial énfasis en las propiedades que las justifican.

4.4.1 Estructura del grafo de escena

Dentro de los módulos involucrados en un motor 3D, el *grafo de escena* es el que dialoga más directamente con el programador de aplicaciones. Como su función es contener objetos geométricos de la escena, la definición de su contenido debe ser realizada externamente, previamente al proceso de dibujado. Esta definición puede realizarse con editores o mediante funciones que se exponen al programador de aplicaciones.

Una de las cotas de este proyecto consistió en no implementar editores de escenas, por lo que el desarrollo de una API propia resultó fundamental. En este punto se plantearon una serie de problemas muy significativos:

- ¿Es posible flexibilizar la funcionalidad de la API sin perder rendimiento final?
- ¿De que manera se puede describir una escena formada por objetos geométricos?
- ¿Cuáles son las funcionalidades que pueden resultar útiles a un programador de aplicaciones y cómo se pueden traducir a funciones de la API?

El estudio de las restricciones de diseño descritas al comienzo del capítulo, mostró que detalles específicos relacionados a la geometría, como ser el orden en que se dibujan los objetos, pueden tener un impacto significativo en el rendimiento general. Este tipo de restricciones son muy difíciles de controlar, porque generalmente se utilizan datos de varias fuentes diferentes: editores de escenas, scripts, etc. Es por esta razón que el programador de aplicaciones generalmente no tiene control sobre el orden en que son procesados los datos con los que está trabajando.

Esta observación sugirió separar las etapas de descripción de la escena y la de optimización de sus datos, de manera de poder flexibilizar el diseño de la API.

A partir de este punto se buscó diseñar un sistema flexible e intuitivo de descripción de escenas, para lo cual se estudiaron distintas alternativas extraídas de la observación de aplicaciones comerciales existentes y de la literatura disponible.

Los siguientes puntos fueron tomados en cuenta para decidir el diseño final:

- La memoria de video de las tarjetas 3D es uno de los factores limitantes más importantes por lo que los artistas y diseñadores trabajan con conjuntos reducidos de texturas, las cuales combinan de diversa manera para darle variedad a las escenas. Es, por lo tanto, muy común que varios objetos compartan recursos, como texturas o luces. Esto sugiere dos importantes aspectos del sistema:
- Conviene separar la descripción geométrica del resto de las características de los objetos como ser material, color, transparencia o textura, de manera de poder agrupar objetos que utilicen los mismos recursos.
- Es muy común que las posiciones de los objetos se definan a partir de matrices de *transformación* ya que pueden ser combinadas mediante la operación de producto para generar nuevas posiciones. Si utilizamos esta propiedad para definir una relación de dependencia entre las posiciones de los objetos, tendremos que éstas consistirán en una matriz de *transformación* local que especificará la posición relativa de un objeto respecto de su nodo padre. [EBER02c].
- Las agrupaciones posicionales son absolutamente independientes de las agrupaciones por recursos.

Estas observaciones llevaron a inferir la necesidad de trabajar con dos agrupaciones jerárquicas: una para definir las posiciones y otra para definir las características de los objetos. Se construyeron, entonces, dos árboles n-arios: un árbol de descripción y un árbol de posición.

4.4.2 Árboles de descripción y árboles de posición

Siguiendo la idea de Eberly [EBER02c], definimos los árboles de descripción a partir de dos tipos de nodos: Técnicas y Geométricos. Las Técnicas son las encargadas de describir características de los objetos como su material, luces que los afectan o niveles de transparencia y se ubican en los nodos internos del árbol. Los nodos geométricos son los encargados de contener la información geométrica de los objetos y están en las hojas del árbol.

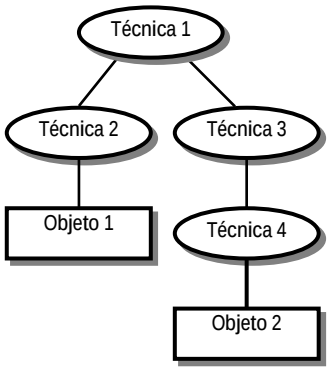


Figura 4.6 *Ejemplo de un árbol de descripción: el Objeto2 es afectado por las técnicas 1, 3 y 4, mientras que el Objeto1 solo por las técnicas 1 y 2.*

Por otra parte los denominados árboles de posición están compuestos por nodos Posición que contienen las transformaciones locales a aplicar.

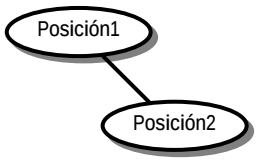


Figura 4.7 *Árbol de posición que define una relación de dependencia espacial entre la Posición2 y la Posición1. La Posición2 es relativa a la Posición1.*

Ambos árboles se vinculan mediante una asociación 1 a n entre nodos posición y nodos geométricos como pude verse en la figura 4.7. El hecho que la relación sea 1 a n permite asociar un mismo nodo posición a más de un nodo geométrico.

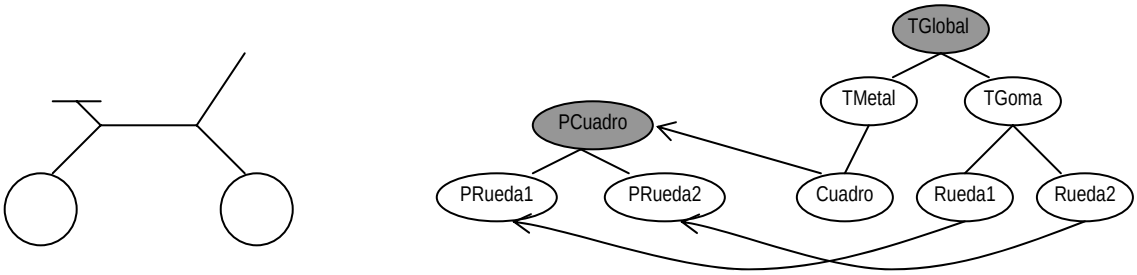


Figura 4.8 *Grafo de escena compuesto por dos árboles relacionados: el árbol de descripción y el árbol de posición.*

Esta forma de representación de escenas resultó ser muy cómoda a la hora de desarrollar aplicaciones debido a la naturaleza intuitiva de la realidad a modelar: los objetos se agrupan en función de su dependencia a técnicas y de su dependencia espacial en forma directa..

4.4.3 API

A partir del diseño del *grafo de escena*, y conjuntamente con su implementación, se fue desarrollando un conjunto de funciones básicas que permitiera probar su funcionamiento en aplicaciones reales. Este conjunto de funciones continuó evolucionando con el desarrollo de las distintas partes del proyecto, llegando a conformar una API básica que cubre las principales necesidades. En esta etapa se definieron funciones básicas (altas, bajas, modificaciones) sobre cada uno de los árboles, así como funciones de carga archivos con contenido geométrico en varios formatos.

4.4.4 Grafo celda-portal

La función del *grafo celda-portal*, como se dijo anteriormente, es la de contener la información sobre los objetos geométricos correspondientes al entorno arquitectónico de la escena a dibujar. Su construcción generalmente se asocia a la existencia de un grafo *BSP* que es utilizado en el proceso de división geométrica. En el capítulo 6 se detalla todo este proceso.

Desde el punto de vista del *renderer* y *grafo de escena*, lo importante es que en nuestro proyecto, esta información se generó utilizando un traductor de mapas del editor de escenas GTK Radiant. El formato en que se presentan los datos traducidos no respeta ninguna de las restricciones de diseño antes mencionada, fundamentalmente por tratarse de un trabajo que normalmente se realiza *off-line*.

Detalles del formato de salida del traductor pueden encontrarse en los apéndices C3 y C4.

4.4.5 Optimización Previa: analizadores y GRD

Si bien los árboles n-arios resultan muy adecuados para describir las escenas, su diseño se basó en consideraciones que, en general, no son compatibles con las reglas de diseño indicadas al comienzo de este capítulo. Dentro de estas reglas, la que indica que se debe dibujar la mayor cantidad de objetos geométricos que compartan estados de la API resulta una de las más significativas [ASHI06].

Observamos varios puntos de incompatibilidad de los grafos de escena con estas reglas. Estos son:

- La naturaleza jerárquica de los datos contenidos en el *grafo de escena* no permite dibujar los objetos minimizando los cambios de estado ya que las recorridas en los árboles generalmente no cumplirán con esta propiedad.
- Los objetos geométricos se encuentran en las hojas de los árboles por lo que una recorrida en *pre* o *post* orden generará una cantidad importante de llamadas recursivas y uso de stack que no es recomendable si se quiere maximizar la velocidad de procesamiento.
- Las *luces* activas, definidas a partir de técnicas, generan un trabajo de procesamiento significativo por lo que el sistema debe racionalizar al máximo su uso en función de la dependencia de los objetos en el árbol de descripción, Ver apéndice A.4. Se requiere, entonces, un pre-procesamiento de las luces y objetos involucrados en el sistema para minimizar los cálculos necesarios.
- Por último, la filosofía de *Precaching* aplicada al diseño del Engine implica que el árbol no cambiará de estructura en ningún momento por lo que muchas tareas de optimización pueden realizarse una sola vez y ser utilizadas durante todo el proceso de dibujado en tiempo real.

Estas observaciones justificaron la creación de dos módulos encargados de optimizar la información geométrica de las escenas: el **Analizador de escena**, encargado de optimizar los datos del *grafo de escena* y el **Analizador de celdas** cuya tarea es convertir los datos del *grafo celda-portal*. La arquitectura de la solución propuesta es la siguiente:

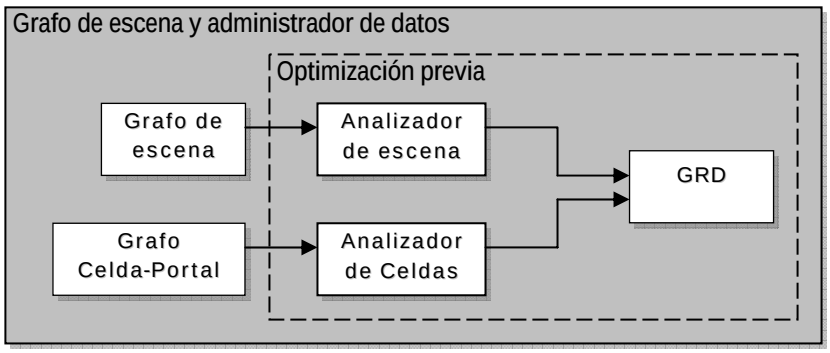


Figura 4.9 *Analizadores dentro de la arquitectura del grafo de escena. Forman parte del módulo que en la arquitectura general se denomina Optimización previa. Ver fig. 3.2*

4.4.6 El GRD

Partiendo de la base que la escena estará definida a partir de un *grafo de escena* con las características descritas anteriormente, se plantea la necesidad de transformar esa información a un formato que el *renderer* pueda dibujar de manera eficiente.

Para que esto pueda ser realizado se debe definir una estructura de datos que permita contener la misma información que el grafo, pero que se encuentre organizada de manera de permitirle al *renderer* dibujarla eficientemente.

En el diseño de esta estructura se tuvieron en cuenta las restricciones impuestas por el hardware. Dentro de ellas, la que indicaba la necesidad de minimizar los cambios de estado resultó clave. La solución encontrada consistió en generar listas de objetos geométricos que compartieran los mismos estados: texturas, nivel de opacidad, *shaders*, etc.

El GRD, cuyos detalles de diseño pueden leerse en el A1, es una estructura que cumple con las principales restricciones de diseño planteadas en [ASHI06]. Es utilizada por ambos analizadores para optimizar los datos de la escena.

4.4.7 Analizador de escenas

La función del analizador de escenas es, como se dijo anteriormente, procesar el *grafo de escena* y cargar la información en un GRD. Dado que esta estructura permite agrupar objetos geométricos, la tarea del analizador es recorrer el *grafo de escena* buscando repeticiones de estados.

Los detalles las características específicas de los datos que el analizador debe optimizar, así como una breve descripción del algoritmo implementado pueden leerse en A1.

4.4.8 Analizador de celdas

La información del *grafo celda-portal*, generada por el traductor a partir de los datos del GTK-Radiant, no viene agrupada según las restricciones de diseño planteadas inicialmente.

Este grafo define zonas cerradas denominadas **Celdas**, cuyo contenido es chequeado en tiempo real para determinar si el sistema debe o no procesarlas.

La función del analizador de escenas es procesar estos datos para que puedan ser eficientemente dibujados.

El problema adicional que se plantea respecto al analizador de escenas, es que el dibujo de las celdas depende del resultado de los chequeos realizados en la etapa de optimización geométrica, por lo que una optimización a nivel de todo el grafo no es posible.

La solución planteada, propone la creación de listas de datos optimizadas para cada celda en forma independiente utilizando un GRD en cada caso.

Los detalles de implementación así como una discusión más profunda sobre sus pro y contra pueden leerse en el A1.

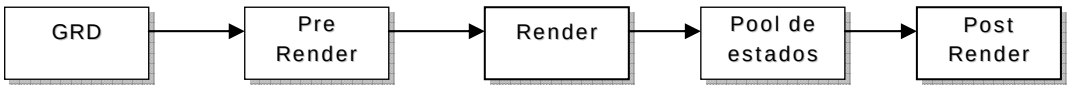
4.4.9 Render y pool de estados.

EL proceso mediante el cual la escena es finalmente dibujada parte de las listas de datos optimizados contenidas en el GRD y, previo pasaje por el proceso de optimización geométrica, se transforma en instrucciones de dibujo. Este es el trabajo que realiza el módulo Render.

Dado que los datos de las celdas se optimizan localmente (hay un GRD por cada celda), los diferentes conjuntos de celdas visibles pueden definir conjuntos de datos no óptimos.

La función del módulo denominado Pool de estados consiste en intentar optimizar los cambios de estado de un conjunto no constante de datos geométricos agrupándolos, en tiempo real, en función de los estados asociados a cada objeto.

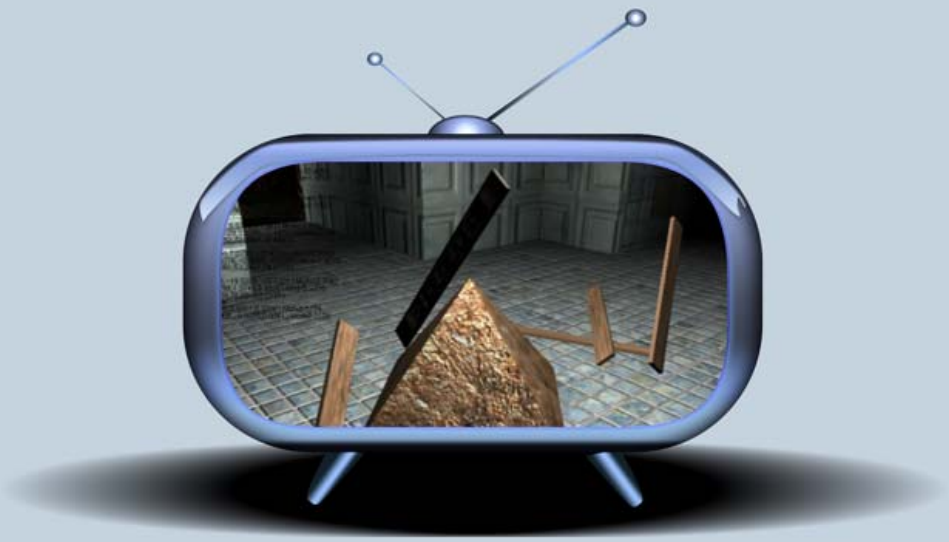
El pipeline final que se conforma es el siguiente:



Los detalles de la estrategia implementada por el pool de estados se pueden leer en el *A1: Optimización de datos geométricos.*

5

Colisiones y Respuesta



5.1 Introducción
5.2 Análisis de requerimientos
5.3 Arquitectura
5.4 Diseño general
5.5 Cercanía y detección en dos fases
5.6 Determinación de celdas de pertenencia
    5.6.1 Volúmenes barridos
5.7 Obtención de colisionantes potenciales
    5.7.1 Mapa de colisiones
5.8 Detección fina
5.9 Respuesta física utilizando impulsos
5.10 Otras consideraciones

5.1 Introducción

El sistema de colisiones y respuesta incluido en el motor, constituyó uno de los elementos que reconocimos como fundamentales para poder otorgar, a las aplicaciones tridimensionales construidas con él, algún tipo de interacción entre los objetos componentes de una escena.

Para lograr este objetivo se realizó, en primera instancia, un estudio de los métodos existentes, lo que nos permitió luego utilizar diversas técnicas. La premisa principal fue siempre tratar de utilizar algoritmos que resultaran poco sensibles a la cantidad de objetos incluidos dentro del sistema de colisiones. Esto nos llevó, en muchos casos, a desechar caminos antes elegidos. Finalmente encontramos una serie de algoritmos que encajaban bien con el sistema de división espacial y que podían acoplarse correctamente con el sistema de visualización.

El trabajo de Brian Mirtich en esta área resultó una guía desde el comienzo, donde no se consideró la implementación de un sistema físico pero sí la implementación de un sistema de colisiones continuo. Luego, en una segunda etapa, cuando se visualizó como algo realista la implementación de física de sólidos entre objetos, la lectura de su paper germinal [MIRT96] resultó fundamental, así como también los trabajos de David Baraff [BARA89; BARA92; BARA93].

El proceso de construcción de este sistema evolucionó en las etapas clásicas de análisis de requerimientos, diseño e implementación, según se describe en las siguientes secciones.

5.2 Análisis de requerimientos

Las observaciones que realizamos de aplicaciones 3d en tiempo real, mayoritariamente juegos, nos mostraron que una característica importante de ellos era la simulación de colisiones y respuesta entre objetos. En aplicaciones en que se simulan personajes reales inmersos en un entorno natural, estas técnicas ayudan a fortalecer la sensación de realismo del personaje, al evitar la interpenetración de este con paredes, puertas, etc., y al lograr una reacción física que se corresponde, en la mayoría de los casos, con la intuición que todos tenemos.

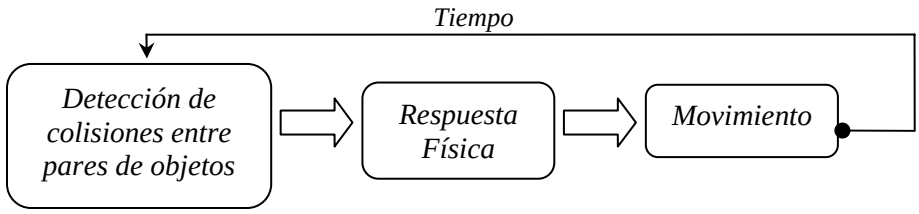
En este sentido se puede observar que las soluciones encontradas para obtener estos resultados no siempre parecen convincentes o intuitivamente realistas para un usuario experimentado. Por ejemplo, en el juego Half Life [VALV03] es posible manipular unas cajas como forma de alcanzar ciertos objetivos del juego, pero los movimientos que se logran durante esta interacción son bastante toscos y poco intuitivos. En cambio, en la siguiente versión del juego, Half Life 2 [HAL204a], esta interacción fue mejorada notablemente debido a la inclusión de un sistema de respuesta física que modela la dinámica de sólidos [HAL204b].

La simulación de dinámica de sólidos fue uno de los principales objetivos del proyecto. La investigación inicial mostró un número importante de metodologías diferentes y una serie de algoritmos intermedios complejos, que requerían además, una pulida optimización con el fin de obtener tiempos de respuesta interactivos [ERLE05b]. Asimismo se evidenció que para implementar estas técnicas primero debía solucionarse el problema de la *detección de colisiones* entre los objetos componentes de un sistema. Luego de solucionado este problema, la simulación física no sería más que una forma de responder frente a la colisión detectada. Sin embargo, para implementarla, se reconoció la necesidad de obtener información muy concreta sobre cada colisión como son los puntos de contacto entre los pares colisionantes, las normales a las superficies de contacto, y el tiempo en que la colisión ocurrió. Con esta información, más ciertas características de los objetos como sus tensores de inercia y masa, es posible modelar una plausible respuesta física aplicando ecuaciones de impulso que definan el movimiento posterior a la colisión [BARA92].

El desarrollo del sistema de colisión y respuesta se divide entonces en dos partes: la implementación de algoritmos encargados de la *detección de colisiones* entre objetos y la implementación de algoritmos de respuesta, que permitan proveerles un comportamiento físico realista. De estas dos partes, el problema de detectar correctamente la colisión entre objetos resultó el más dificultoso de resolver debido a la necesidad de optimizar este proceso en función de la elevada cantidad de objetos que pudieran componer una escena. La solución "simple" que consiste en verificar la colisión entre cada par de objetos se desechó desde el comienzo, por tener un orden cuadrático en el número de objetos. La elección de estrategias alternativas formó parte de un proceso de investigación que terminó en el desarrollo de las técnicas que se describen en los párrafos siguientes. La respuesta física tuvo un costo menor de implementación, aunque el camino seguido fue menos ambicioso. Las consecuencias de esta elección se reflejan en un comportamiento poco eficiente en determinadas situaciones. En el capítulo 7: Trabajos Futuros, se describen procedimientos alternativos que permitirán eliminar estos comportamientos no deseados.

5.3 Arquitectura

La relación entre los algoritmos de *detección de colisiones* y los algoritmos encargados de responder a ellas en forma realista, se expresa en el siguiente diagrama:



donde el modulo "Detección de colisiones entre pares de objetos" es el encargado de reportar cada par de objetos que estén en situación de colisión en el instante actual de tiempo y el modulo "Respuesta Física" es el encargado de recoger esa información y aplicar algoritmos de impulso entre cada par colisionante. Luego, el sistema es libre de moverse hasta que se detecte alguna nueva colisión.

El módulo de *detección de colisiones* debe ser capaz de devolver el conjunto de pares de objetos que se encuentran en esa situación considerando todos los objetos que componen una escena. Si el sistema considera únicamente objetos rígidos, la principal categorización dinámica que se puede establecer entre ellos es su movilidad: los objetos móviles serán los únicos con capacidad de moverse y por lo tanto colisionar con otros objetos (móviles o fijos). Esta categorización permite reducir la cantidad de pares de objetos a considerar, dado que en cada posible colisión debe haber al menos un objeto móvil. La relación de colisión entre objetos fijos, entonces, no se considera ni resuelve.

Para representar el movimiento se optó por asociar dos vectores a los objetos móviles: un vector velocidad lineal, que rige su movimiento de traslación y un vector velocidad angular que gobierna la forma en que el objeto rota alrededor de su centro de coordenadas local. Ambos vectores están expresados en un sistema global de coordenadas o, como se denomina: sistema de coordenadas del mundo ("World Coordinates" en inglés).

El modulo de respuesta física tendrá como objetivo el evitar que los objetos en colisión terminen intersecándose. Para lograrlo será suficiente un cambio adecuado de sus vectores velocidad pero nada más, es decir, no hará ningún tipo de reposicionamiento de los objetos en conflicto. Esta idea se sustenta en poder detectar colisiones antes de que se produzcan intersecciones entre los objetos.

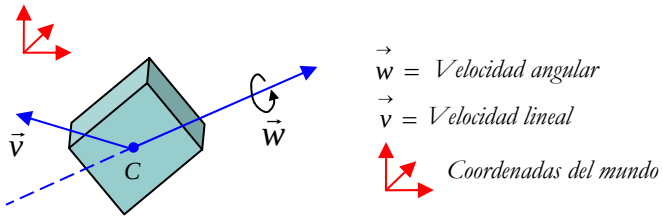


Figura 5.1 *Vectores que rigen el movimiento de los objetos móviles expresados en un sistema global de coordenadas.*

El problema de programar técnicas para simular la interacción física puede ahora expresarse de la siguiente manera:

> "Partiendo de una situación en la que no existe intersección entre los objetos de un sistema, determinar los pares de objetos que se encuentren en situación de colisión luego de haberlo hecho evolucionar durante un cierto lapso de tiempo. Para cada uno de ellos aplicar funciones de respuesta realistas determinando sus nuevos vectores de movimiento".

El concepto de intersección se aplica, en este contexto, únicamente a la relación entre objetos móviles entre sí y a objetos móviles con objetos fijos. Por lapso de tiempo nos referimos aquí al tiempo establecido por el sistema para realizar el movimiento de los objetos. Su valor concreto depende fuertemente de la técnica de *detección de colisiones* elegida. En nuestro caso, en el que se implementó un sistema de colisiones continuo, ese tiempo es determinado por el propio sistema en función de la posición y vectores de movimiento actuales de los objetos (o sea antes de realizar el movimiento). En lo que sigue se denomina tiempo de evolución del sistema a este lapso temporal.

Cabe destacar además, que lo que denominamos movimiento de los objetos, se refiere a la aplicación instantánea de los vectores velocidad durante el tiempo de evolución establecido. Nada puede ser detectado durante ese tiempo. El sistema se dice, así, de paso discreto, aunque el tiempo de evolución sea diferente cada vez. Como se verá, el sistema evoluciona de a pasos hasta la ocurrencia de eventos de colisión en los que debe actuar para cambiar adecuadamente los vectores velocidad de aquellos objetos que colisionen.

5.4 Diseño general

La solución implementada permite adaptar el tiempo de evolución del sistema, con el fin de poder detectar cualquier choque posible (ver figura 5.2). El sistema utilizado se basa en el cálculo aproximado del *tiempo de impacto* entre cualquier par de objetos que se encuentren "suficientemente" cerca como para estar en peligro de colisión [MIRT96; REDO04a]. Los pares de objetos cercanos son ordenados en una *cola de prioridad*, que llamamos cola *TOI*[2], cuya clave es ese *tiempo de impacto* estimado. Esta estructura permite obtener rápidamente el elemento con menor *tiempo de impacto*, que se ubicará siempre en el tope de la cola y estará listo para ser utilizado[3]. Debido a esto, el sistema podrá evolucionar (o sea mover a todos sus objetos) hasta ese tiempo *TOI*, sin riesgo de no detectar la ocurrencia de algún choque. Ese menor *TOI* es el que se asocia, en cada paso, al tiempo de evolución de todo el sistema.

[2] Sigla de "*time of impact*" de acuerdo con el paper original de Brian Mirtich [MIRT96]

[3] La implementación utilizada de colas de prioridad r [DRDO01] tiene operaciones de inserción y borrado de orden uno en la cantidad de elementos.

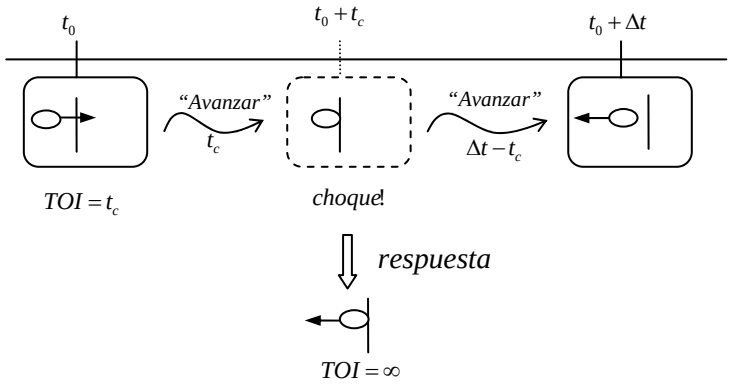


Figura 5.2 *La solución encontrada implica hacer avanzar el sistema hasta el menor tiempo de impacto calculado para cada par de objetos que se encuentren cerca. Esta estrategia garantiza que no se pierda ningún choque y hace robusto al sistema. En la figura, evolucionar el sistema un tiempo* Δt *se hace en 2 pasos: primero hasta el primer tiempo de impacto* t_c *y luego hasta el resto del tiempo* $\Delta t - t_c$ *una vez aplicada una función de respuesta a la colisión detectada.*

El diagrama de bloques de la figura 5.3 muestra el funcionamiento general del sistema de colisiones y respuesta.

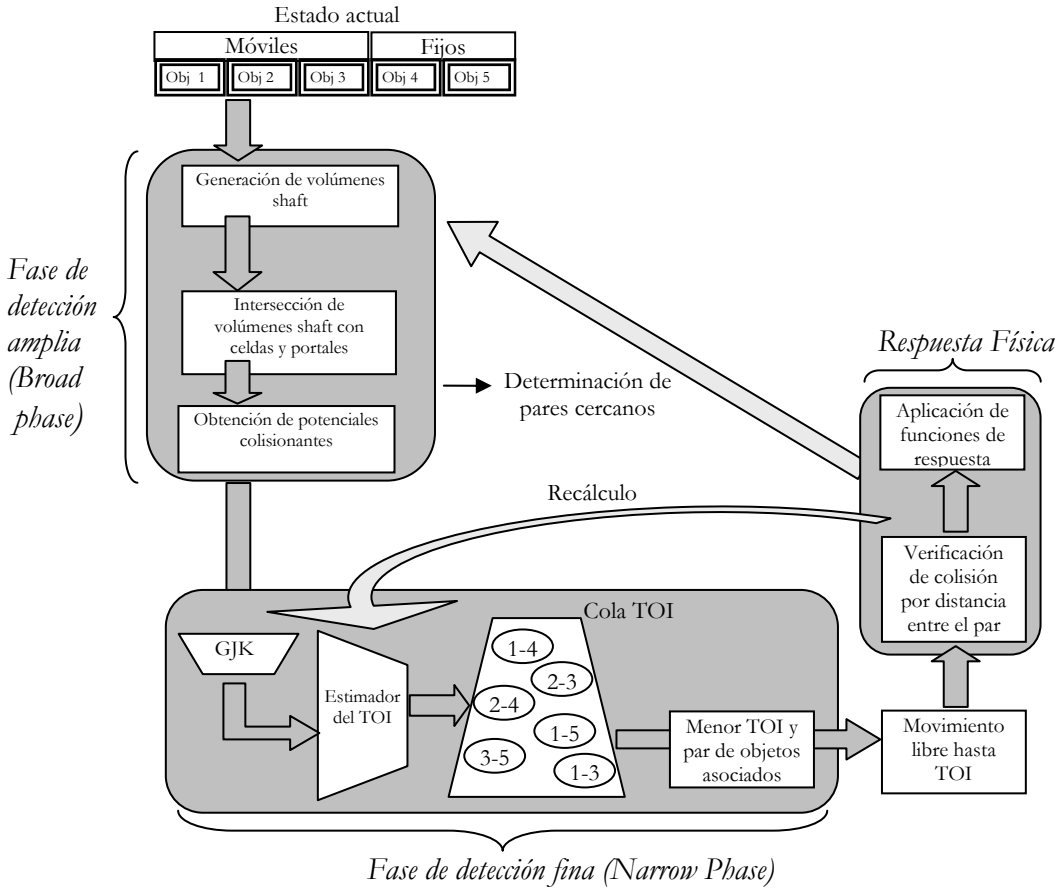


Figura 5.3 *Diagrama de bloques del sistema de colisiones. Las flechas representan flujo de datos. E bloque "Determinación de pares cercanos" se corresponde con la fase de descarte o Fase de detección amplia del sistema, y los bloques "Estimador de TOI", "Cola TOI" y "Menor TOI y par de objetos asociados" se corresponden con la Fase de detección fina o segunda fase del sistema de detección de colisiones. El resto de los bloques se relacionan con el sistema de respuesta.*

Partiendo de un estado inicial de posición y orientación de los objetos móviles, el sistema descarta, en primera instancia, todos aquellos pares de objetos que se encuentren lejos de colisionar. Este trabajo es realizado por el conjunto de algoritmos que conforman la *Fase de detección amplia* (o *Broad Phase*) los cuales son específicos a la técnica de detección continua de colisiones [REDO04a]. Como resultado se obtiene el conjunto de pares de objetos cercanos para los cuales se estima el *tiempo de impacto* con el que se alimenta la cola *TOI*. Como se dijo, la cola *TOI* mantiene, en su tope, el par de objetos que colisionarán antes que cualquier otro junto con el tiempo estimado de su impacto. El sistema entonces avanza hasta ese tiempo. Luego, sin embargo, será necesario volver a verificar por la existencia de impacto entre los objetos que determinaron ese tiempo ya que, debido al carácter aproximativo del cálculo del *tiempo de impacto*, es posible que los objetos no terminen en esa situación. Si efectivamente se verifica la existencia de la colisión, se aplican las funciones de respuesta antes mencionadas que cambian sus vectores velocidad. Como consecuencia de este cambio será necesario recalcular el *tiempo de impacto* de todo par de objetos de la cola *TOI* que contenga alguno de los objetos que colisionaron. Esto se debe a que el método de cálculo del *tiempo de impacto* entre objetos es dependiente de los vectores velocidad (ver apéndice B1). Los objetos móviles se encuentran luego, en un nuevo estado de posición y orientación por lo que el proceso puede volver a repetirse.

El algoritmo principal del sistema entonces, es el que permite hacer la estimación del *tiempo de impacto*. La información resultante permite, mediante la cola *TOI* la determinación del tiempo de evolución del sistema así como también determinar cuando aplicar las funciones de respuesta física. Su aplicación exhaustiva para todo par de objetos resulta, sin embargo, demasiado costosa. En la sección siguiente se define este problema y se justifican y explican los algoritmos que conforman la *fase de detección amplia* que permite eliminar la aplicación exhaustiva del estimador para luego explicar detalles concretos del proceso de estimación y su influencia en el funcionamiento general del sistema.

5.5 Cercanía y detección en dos fases.

Para calcular el *tiempo de impacto* entre pares de objetos se utiliza una técnica que necesita conocer la distancia que los separa y sus puntos más cercanos (Ver apéndice B1). Esta información es proporcionada por el algoritmo *GJK* [GILBE88; CAME97; BERG04]. Este algoritmo es capaz de obtener esta distancia en forma eficiente[4], con la condición de que los objetos sean convexos y que su descripción geométrica se haga en forma adecuada (grafo de adyacencia en los vértices).

Este cálculo es entonces razonablemente eficiente debido a la aplicación del algoritmo GJK. Sin embargo, esta eficiencia no es suficiente para garantizar un funcionamiento global eficiente de todo el sistema de colisiones, debido a que su aplicación a todos los pares de objetos de una escena resulta en un tiempo cuadrático en la cantidad de objetos. Se puede argumentar que entre los objetos fijos no es necesario responder frente a una colisión y que por su carácter estático nunca habría que aplicar el algoritmo entre ellos. Este hecho, si bien reduce la cantidad de cálculos, resulta insuficiente: para M objetos fijos y N objetos móviles, la cantidad de comparaciones es igualmente considerable:

$$\begin{array}{l} \textit{Objetos móviles contra objetos fijos: } M*N \textit{ comparaciones} \\ \textit{Objetos móviles entre sí: } N(N+1)/2 \textit{ comparaciones} \end{array} \qquad N<<M.$$

Insertar un nuevo objeto móvil en el sistema requeriría M nuevas comparaciones contra los objetos fijos y N nuevas comparaciones contra los móviles (Siendo N+1 el total de

[4] Orden uno en función de la cantidad de vértices de ambos objetos

nuevos objetos móviles), lo cual resulta excesivo cuando se manejan cientos de objetos. Se concluye así que es necesaria alguna técnica que permita reducir la cantidad de mediciones de distancia necesarias para resolver el problema del cálculo del *tiempo de impacto*.

En la mayoría de las aplicaciones que observamos y que utilizan un sistema de colisiones, pudimos apreciar que no todos los pares de objetos que componen el sistema se encuentran siempre en peligro de colisión. En una situación común, los objetos móviles estarán más bien dispersos en una escena, colisionando, en general, con un subconjunto muy menor de otros objetos del sistema. Una heurística interesante sería aquella que nos permita determinar rápidamente si dos objetos cualesquiera están en potencial situación de colisión, es decir, si se encuentran peligrosamente cerca. Solo para ellos sería entonces provechosa la aplicación de un algoritmo (más costoso) de determinación de distancia y cálculo de *tiempo de impacto*. Para los demás pares de objetos, la lógica de la heurística debería garantizar la imposibilidad del choque, por lo menos en un tiempo cercano.

Un sistema de colisiones basado en este principio se denomina *sistema de detección en dos fases*. Básicamente se aplica al sistema una primera fase de descarte (o *Fase de detección amplia*) en donde únicamente los pares de objetos en potencial colisión son determinados. Para cada par se aplica, luego, la segunda fase (o *Fase de detección fina*) en donde se determina su tiempo estimado de impacto, dato que se utiliza para alimentar la cola *TOI*. El resultado de esta fase es la cola *TOI* debidamente actualizada en cuyo tope se encuentra el par de objetos que colisionará antes que cualquier otro.

El sistema construido (ver figura 5.4) es un sistema de dos fases en donde la primera fase está a su vez dividida en otras dos etapas. En la primera etapa, a cada objeto móvil se aplica un algoritmo que determina el conjunto de celdas del *grafo celda–portal* que el objeto podría ocupar en función de la trayectoria del movimiento definido por sus vectores velocidad, aplicadas durante el intervalo de tiempo establecido, y partiendo de su posición actual. Luego, en una segunda etapa, se aplica para cada celda encontrada, un algoritmo (*Sweep & Prun*e, explicado más adelante) que determina cuales son los objetos internos a ellas que podrían ser próximos al objeto móvil durante su trayectoria de movimiento. Esos objetos se declaran como potencialmente colisionantes.

En la segunda fase que llamamos *fase de detección fina* se aplica el algoritmo de determinación de distancia entre el objeto y todos sus potenciales colisionantes declarados, obteniendo el tiempo estimado de colisión con cada uno de ellos. La información obtenida es insertada en la *cola de prioridad TOI* para la posterior determinación del tiempo mínimo de evolución global del sistema el cual surgirá naturalmente en el tope de la cola luego de aplicar estos procedimientos a cada objeto móvil definido. Durante ese tiempo mínimo queda garantizado entonces que no ocurrirá ninguna colisión. Finalmente el sistema mueve a todos los objetos durante ese lapso temporal, en forma discreta.

En lo que sigue se hace una descripción detallada del diseño e implementación de cada una de las fases antes mencionadas, para explicar luego como se aplican concretamente estas ideas en el sistema de implementado.

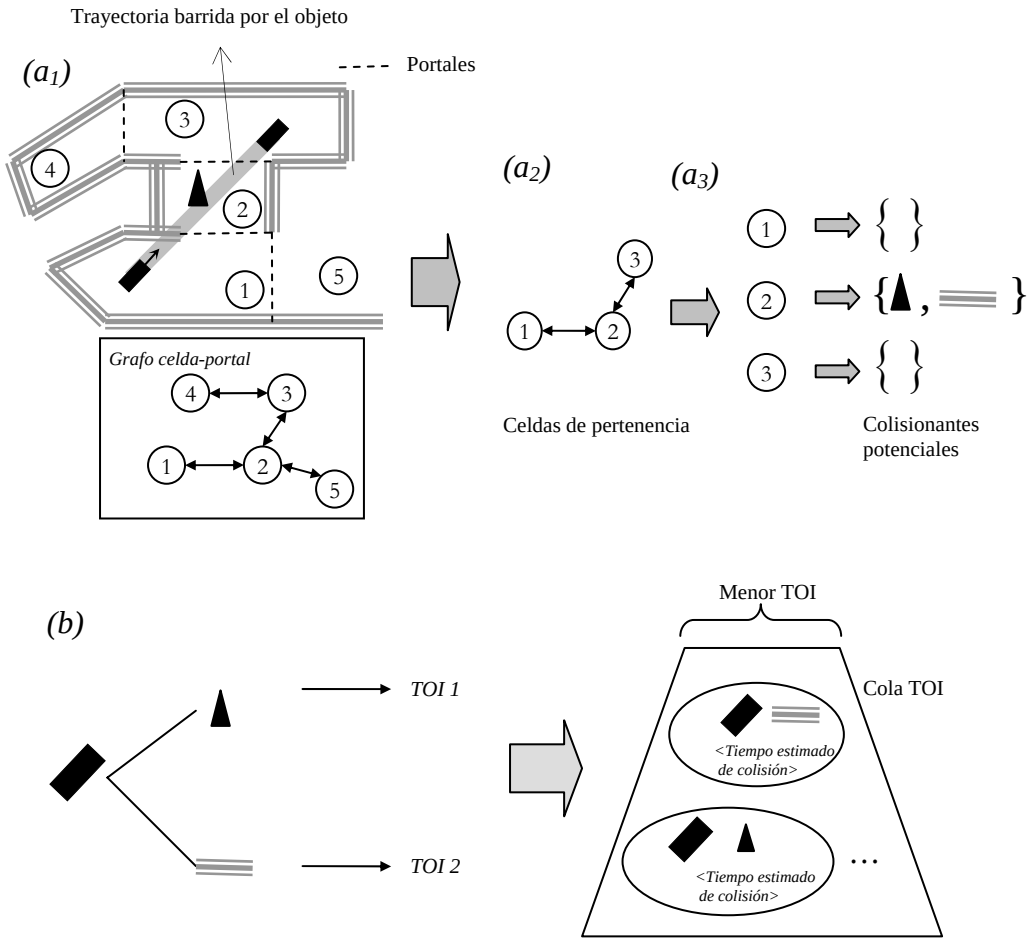


Figura 5.4 *Descripción del sistema de colisiones aplicado a un solo objeto móvil. En (a_1) se muestra el objeto (rectángulo) en su posición inicial, y la trayectoria barrida por su movimiento. En (a_2) se muestran todas las celdas atravesadas por la trayectoria denominadas celdas de pertenencia del objeto. Para cada una de ellas se muestra, en (a_3), los objetos atravesados por la trayectoria. Esos son los objetos potenciales colisionantes. En (b) se muestra el proceso de la fase de detección fina. Para cada potencial colisionante se calcula el tiempo de impacto que alimenta la cola TOI (global para todo el sistema) cuyo par del tope indica el tiempo de impacto y los objetos que podrían colisionar antes que cualquier otro.*

5.6 Determinación de celdas de pertenencia.

La lógica utilizada en esta fase se basa en la utilización del *grafo celda-portal*. Como cada celda del grafo define una región cerrada del espacio[5], definida por los objetos fijos que definen su arquitectura y por sus portales, podemos decir que en cada instante de tiempo los objetos móviles se encontrarán perteneciendo (siendo total o parcialmente interiores) a un conjunto limitado de celdas del grafo (Ver figura 5.5). Mientras estas relaciones de pertenencia se mantengan, un objeto móvil podrá colisionar únicamente con los objetos que se encuentren en las mismas celdas a las que él pertenece, que llamamos celdas de pertenencia.

[5] Y convexa

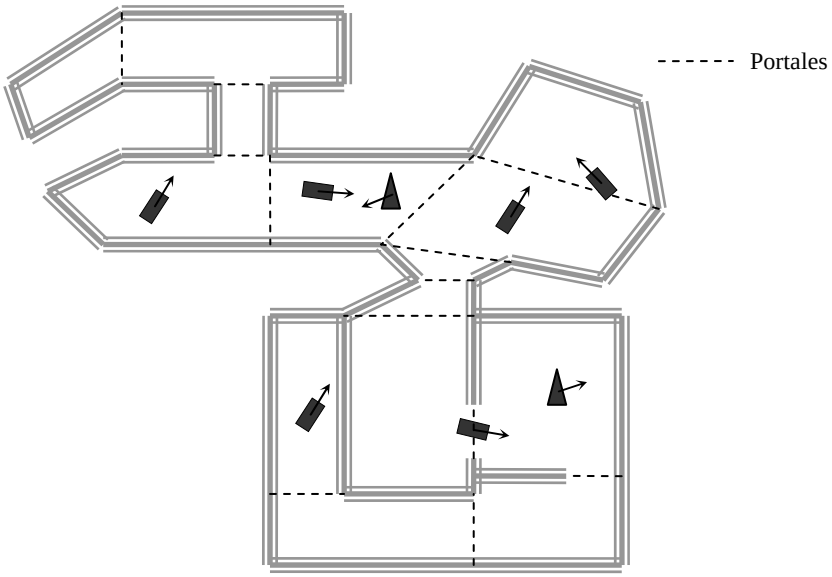


Figura 5.5 *Un modelo arquitectónico particionado en celdas (delimitadas por paredes y las líneas punteadas) y un conjunto de objetos móviles ubicados en él. Solamente habrá riesgo inminente de colisión entre objetos que se encuentren en el mismo conjunto de celdas.*

La pertenencia a celdas por parte de los objetos queda establecida en dos situaciones bien diferentes: un objeto pertenece a una celda debido a que fue ubicado inicialmente allí, o debido a que ha ingresado a ella a través de alguno de sus portales (única vía de comunicación entre celdas). Durante la simulación, los objetos se moverán entre las celdas del grafo, atravesando los portales comunicantes. Toda otra interacción entre objetos es considerada una colisión por el sistema y será resuelta adecuadamente por aplicación de funciones de respuesta física. Resulta esencial entonces, mantener correcta la información de cuáles son las celdas de pertenencia que va ocupando cada objeto durante el transcurso de la simulación y esto puede hacerse verificando cuales son los portales que el objeto va atravesando durante su movimiento. Será entonces necesario definir concretamente qué se entiende por el concepto de "atravesar un portal", tomando en consideración la forma en que se mueven los objetos.

5.6.1 Volúmenes barridos

El movimiento de cualquier objeto durante un cierto intervalo de tiempo genera, en el espacio, un volumen barrido. Este puede verse como el "espacio utilizado" por la trayectoria del objeto en su movimiento. Para garantizar que en un sistema no exista ninguna colisión durante todo ese tiempo, es suficiente verificar que no exista intersección entre los volúmenes barridos por sus objetos, ni entre estos volúmenes con los objetos fijos del sistema. Si alguna de estas condiciones no se cumple, entonces es probable que entre los objetos asociados a los volúmenes intersecados exista alguna colisión en el período de tiempo considerado.

El volumen real barrido por un objeto en movimiento, puede llegar a ser muy complejo (Ver figura 5.6). Uno de los determinantes de esta complejidad es la existencia o no de velocidad angular en el objeto[6]. El sistema implementado utiliza una versión simplificada del volumen barrido llamado *volumen Shaft* [HAIN94], formado por las cajas acotantes del objeto en su posición inicial y final. Este tipo de volumen proporciona una aproximación aceptable al volumen real y puede ser calculado muy rápidamente. En el apéndice B2 se explican los

[6] Otro determinante es la geometría del objeto. Se consideran aquí solamente objetos convexos que generan volúmenes más simples que serán también convexos si el objeto no tiene velocidad angular.

algoritmos asociados a estos volúmenes así como también la forma de construirlos y sus propiedades.

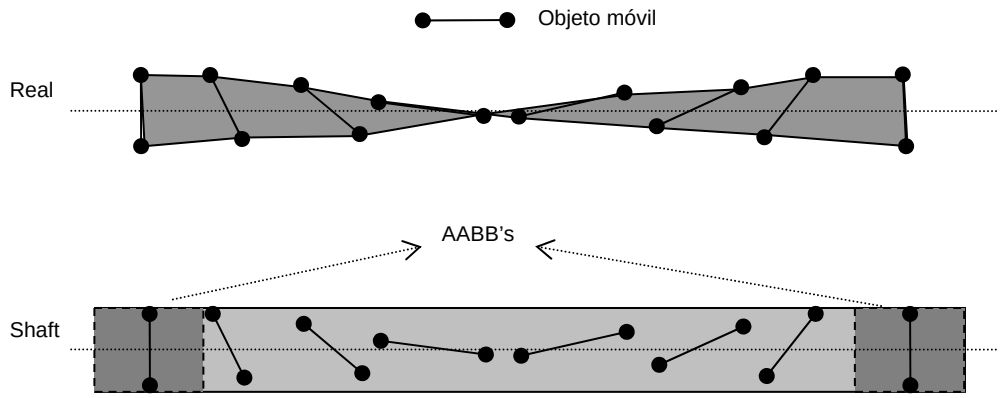


Figura 5.6 *Volumen real barrido por un objeto en movimiento y su correspondiente aproximación como volumen shaft. Se trata del volumen generado por las cajas acotantes AABB del objeto en su posición inicial y final. Su construcción es rápida así como la verificación de intersecciones con otros objetos. Ver Apéndice B2.*

Utilizando los volúmenes Shaft podemos entonces detectar claramente cuándo un objeto cambia de celda durante su trayectoria de movimiento: si el *volumen shaft* interseca algún portal de sus celdas de pertenencia, entonces la celda que conecta podría ser alcanzada durante ese movimiento. Esa nueva celda se declara como perteneciente y se vuelve a aplicar el procedimiento de verificación sobre ella en forma recursiva.

Como en general la verificación de intersección de volúmenes shaft contra portales es más costosa que la verificación de intersección de estos con cajas acotantes, el sistema verifica primero que el *volumen shaft* efectivamente interseque la caja acotante de las celdas antes de verificar si sus portales también son intersecados. En la figura 5.7 se muestra la trayectoria de un objeto perteneciente a la celda A y la verificación que se hace sobre los portales que conectan hacia las celdas B y C. Como la caja acotante de la celda B no interseca el *volumen shaft* del objeto, no es necesario verificar por la intersección de este con el portal A-B. Sin embargo, como hay intersección entre el volumen y la caja acotante de la celda C, se realiza la verificación contra el portal A-C, dando negativo en el caso (a) y positivo en el (b). Solamente en el caso positivo se considera a C como nueva celda de pertenencia del objeto.

Cabe destacar que el algoritmo de intersección de volúmenes shaft contra portales es sensible a la geometría del portal. Para acelerar los cálculos, antes del comienzo de la simulación, el sistema simplifica dicha geometría transformándolos en uno o dos triángulos. Esta simplificación permite utilizar un algoritmo eficiente de verificación de intersección entre volúmenes shaft y triángulos para determinar si un *volumen shaft* interseca o no a un portal complejo (Ver apéndice B2).

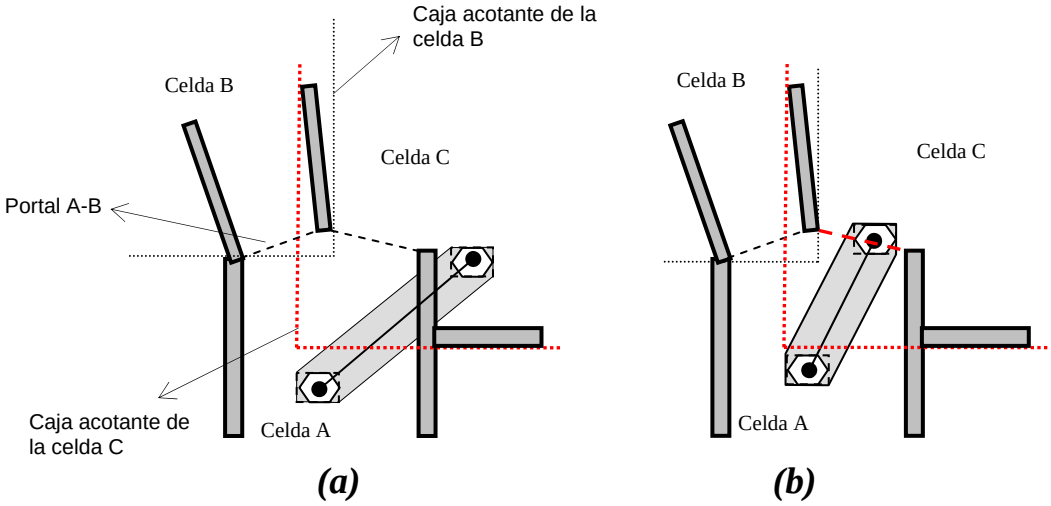


Figura 5.7 *Uso de portales. La celda C será incluida como celda de pertenencia del objeto solamente si su portal es atravesado por el volumen shaf barrido por el objeto. En el caso (a) el volumen shaft no atraviesa el portal A-C y por lo tanto la celda C no es incluida en las celdas de pertenencia del objeto. En el caso (b) esto si sucede y la celda C se incluye. El portal A-B no es considerado en ninguno de los dos casos ya que el volumen shaft no interseca la caja acotante de la celda B.*

5.7 Obtención de colisionantes potenciales

Una vez determinadas las celdas intersecadas por el *volumen shaft* de un objeto móvil, será necesario determinar qué objetos interiores a estas celdas intersecan también con ese volumen. Que un objeto interseque el *volumen shaft* de otro no nos permite asegurar que ambos objetos efectivamente colisionarán, pero sí nos permite afirmar que habrán estado peligrosamente cerca durante el movimiento[7]. Debido a ello los llamamos objetos potencialmente colisionantes. Del resto de los objetos, es decir, de aquellos que no colisionan con su *volumen shaft*, podremos afirmar que no participarán de ninguna colisión durante el lapso de tiempo establecido.

La metodología seguida para la obtención de colisionantes potenciales, se basa en la aplicación de un algoritmo que permite detectar la situación de intersección entre volúmenes acotantes sin recurrir a una verificación exhaustiva entre cada uno de ellos. Como premisa, el algoritmo supone que el movimiento de los objetos verifique la propiedad de *coherencia temporal* que es un concepto que surge de considerar que el movimiento de los objetos es suave y, por lo tanto, no es de esperar que cambien abruptamente de posición en cada paso de la simulación. Mayoritariamente se supone que un objeto móvil permanecerá dentro del conjunto de celdas al que pertenece actualmente y durante un tiempo relativamente grande (al menos un orden de magnitud más que el tiempo promedio de evolución del sistema).

El algoritmo, llamado Sweep and Prune [BARA92], detecta de forma simple la intersección entre las cajas acotantes *AABB* asociadas a cada objeto móvil o fijo, identificando cambios de orden entre los intervalos que las definen para cada eje coordenado. En la figura 5.8 se muestra un ejemplo para un solo eje. Cada caja proyecta en el eje un intervalo [i,f] con i menor o igual a f. Debido al movimiento de los objetos, sus valores cambiarán en cada paso de la simulación. El ordenamiento de los valores en el eje está directamente relacionado con la relación de solapamiento de las cajas. Los intercambios entre inicios y fin de intervalos indican un cambio de la relación de solapamiento: Si las cajas estaban solapándose, dejarán de estarlo y si no lo estaban, comenzarán a hacerlo. En las hipótesis de movimiento suave, esos intercambios siempre podrán ser detectados en forma eficiente con un algoritmo de ordenación simple (bubble sort) y, por lo tanto, también la

[7] Esto puede afirmarse debido a que las trayectorias son establecidas para un mismo lapso temporal para cada objeto, y a que las velocidades lineal y angular de ellos permanecen constantes durante ese tiempo.

relación de solapamiento entre las cajas. La extensión tridimensional del ejemplo implica detectar estos cambios de orden en los tres ejes de coordenadas. Una descripción de este algoritmo y las estructuras asociadas se encuentra en el apéndice B3.

La aplicación de Sweep & Prune nos permite determinar rápidamente aquellos objetos de la celda que estén solapándose con un objeto móvil. Pero para aplicarlo más eficientemente se construye, en una etapa de precálculo, una estructura ordenada de intervalos de todos los objetos fijos de cada celda, que constituye el llamado mapa de colisiones.

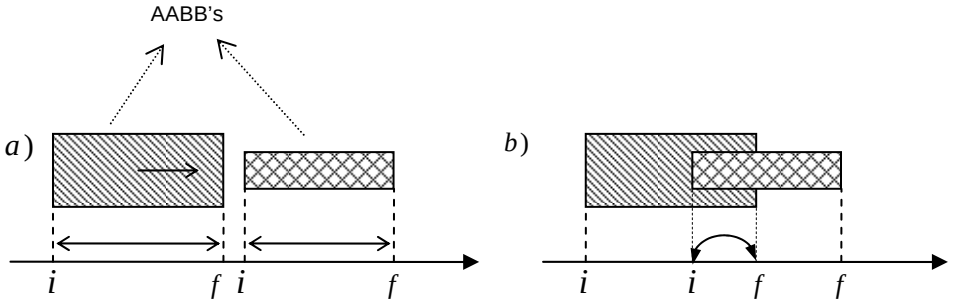


Figura 5.8 *Sweep & Prune en una dimensión: En (a), dos objetos representados por sus AABB y los intervalos que definen en un eje de coordenadas. En (b), el movimiento de un objeto se traduce en un desorden de los intervalos en el eje. El reordenamiento necesariamente deberá intercambiar un extremo de intervalo i por uno f lo cual indica directamente, en este caso, que los objetos están intersecándose. Para un espacio y objetos tridimensionales se deben aplicar estas ideas a cada eje de coordenadas.*

5.7.1 Mapa de colisiones

La eficiencia de Sweep & Prune se debe a que utiliza cajas alineadas con los ejes (o *AABB*) como volúmenes acotantes de los objetos. En el caso de nuestra división en celdas, los objetos involucrados en las colisiones serán: los objetos fijos que definen la arquitectura de las celdas, y los objetos móviles. El algoritmo reportará un potencial colisionante cuando detecte el solapamiento de alguna de las cajas *AABB* que representan esos objetos pero, debido a la naturaleza estática de los objetos fijos, la relación de intersección entre ellos permanecerá constante a lo largo de toda la simulación y no será reportada como colisión a resolver por el sistema. Esto permite crear, para cada celda, el llamado mapa de colisiones, que almacena los intervalos ordenados de la proyección de las cajas *AABB* de cada objeto fijo, en cada eje de coordenadas (ver Figura 5.9). Los objetos móviles también estarán representados por intervalos en los mapas de colisiones, pero serán intervalos móviles que el sistema actualiza al realizar su movimiento.

Cuando un objeto móvil ingresa a una celda, el sistema procede a insertar la caja *AABB* que representa a su trayectoria, dentro del mapa de colisiones de la celda[8]. Esta caja acotante englobará al objeto en su posición inicial y final de movimiento y a todo el espacio recorrido (representado por el *volumen shaft*). La justificación de insertar este volumen exagerado surge por el hecho que nuestro objetivo en esta fase es la determinación de los objetos potencialmente colisionantes con el objeto móvil entrante. Considerar su caja acotante de toda la trayectoria permite, vía el algoritmo Sweep & Prune, determinar rápidamente los objetos que potencialmente la intersequen durante todo ese movimiento (ver Figura 5.9). Puede verse entonces como una primera fase, rápida, de descarte de objetos colisionantes fijos.

[8] La inserción de un nuevo intervalo en el mapa de colisiones de una celda se hace utilizando *árboles de búsqueda por prioridad* (o *PST* trees) calculados en una etapa de precálculo. Ver apéndice B4.

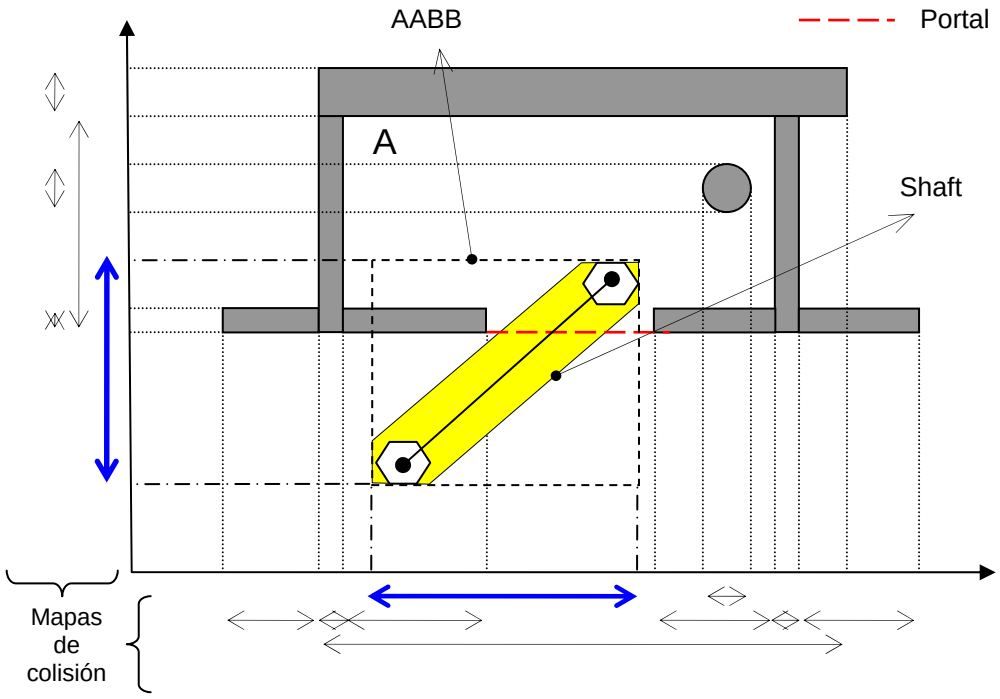


Figura 5.9 *Mapa de colisiones en 2 dimensiones: Las proyecciones de las cajas acotantes de los objetos fijos, representados en gris, generan intervalos en cada eje coordenado cuya ordenación es el mapa de colisiones de la celda A.Esta es una estructura que se calcula en una etapa de precálculo, antes del comienzo de la simulación. En amarillo se muestra la trayectoria de un objeto móvil entrando a la celda y la correspondiente caja acotante generada (AABB). Sobre los ejes se muestran los intervalos estáticos asociados a cada objeto fijo y los dos intervalos dinámicos (en azul) que representan la AABB de la trayectoria del objeto móvil.*

Debido a la naturaleza las cajas *AABB*, un solapamiento podrá o no representar una real colisión, situación que será finalmente determinada en una etapa posterior en la que cada objeto reportado se verifica por intersección contra el *volumen shaft*. Aquellos que lo intersequen se reportan como potencialmente colisionantes. Nuevamente, una colisión no queda garantizada por esta intersección por lo que será necesario verificarla en la llamada *fase de detección fina*.

5.8 Detección fina

Una vez que se han determinado todos los pares de objetos potencialmente colisionantes del sistema, se aplica una serie de algoritmos encargados de hacer una aproximación más fina sobre esos resultados. Su objetivo es estimar el par que colisionará antes que ningún otro, determinando así el nuevo tiempo de evolución. Para ello se aplica, a cada par, un algoritmo que aproxima el *tiempo de impacto* entendiéndolo como el tiempo que debe transcurrir para hacer nula la distancia entre ellos. El algoritmo de cálculo hace una estimación de ese valor haciendo operaciones vectoriales con sus vectores velocidad. Una explicación se describe en el apéndice B1.

El tiempo así estimado alimenta la cola *TOI*, cuya clave es justamente ese valor temporal (ver figura 5.10). Luego de hacer el cálculo para cada par de potenciales colisionantes, la cola *TOI* es capaz de ofrecer, en su nodo raíz, el par de objetos que colisionará antes que cualquier otro, así como también un estimativo del tiempo en que esa colisión ocurrirá. Ese tiempo será tomado por el módulo de animación para hacer evolucionar (mover) efectivamente todo el sistema hasta ese futuro marcado.

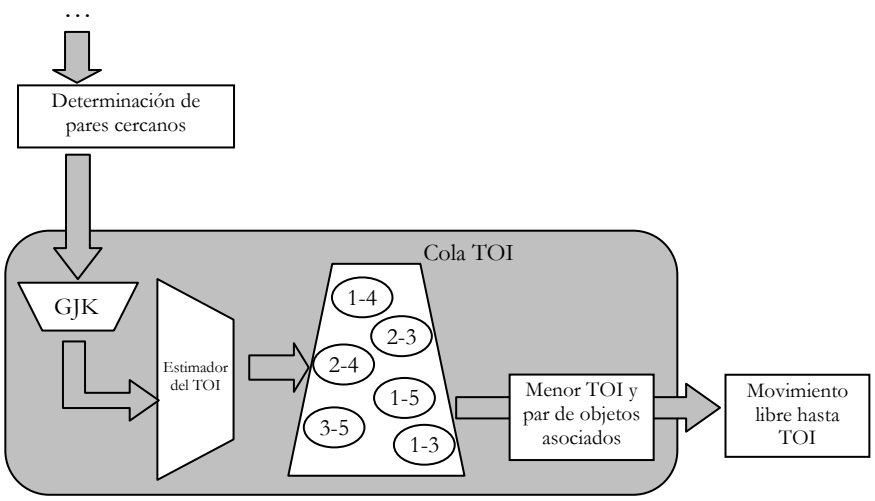


Figura 5.10 *Fase de detección fina: tomando como entrada cada par de objetos cercanos (o potencialmente colisionantes) esta fase se encarga de calcular un estimativo del tiempo de impacto de cada uno. El menor tiempo de impacto determinará el tiempo de evolución de todo el sistema. El trabajo se hace utilizando el algoritmo GJK, un estimador que lo utiliza, y una cola de prioridad para almacenar los valores estimados de cada par.*

El algoritmo principal de esta fase es el que permite calcular la distancia y puntos más cercanos entre 2 objetos convexos. Estos datos son los que exclusivamente toma el estimador del *TOI* para hacer sus cálculos. Como se dijo anteriormente y se justifica en el apéndice B.6, se decidió utilizar el algoritmo GJK para obtener estos valores. Su implementación no forma parte del proyecto. Se utilizo la librería SOLID que provee una implementación robusta y portable de este algoritmo. Una descripción de su funcionamiento general se describe en el apéndice B5.

Para la implementación de la cola *TOI* se utilizó la implementación de *heaps* de Fibonacci aparecida en [DRDO01]. Cabe destacar que la implementación que provee la Standard Template Library de C++ era insuficiente para el tipo de trabajo que fue necesario realizar con la *cola de prioridad* (borrado de elementos que no fueran la raíz provocando un recálculo de los elementos de forma eficiente).

Un punto importante de esta fase es la necesidad de utilizar una *cola de prioridad* para almacenar los cálculos de *tiempo de impacto* entre los objetos. Si bien hasta este punto se ha descrito el sistema como una serie de pasos continuos, lo que sucede en realidad es una realimentación de las estructuras en cada paso de la simulación. Así como los objetos móviles actualizan su posición en la medida en que el sistema va evolucionando, los tiempos de impacto, y por lo tanto los valores de la *cola de prioridad*, irán actualizándose en la medida en que el tiempo pase. Un par de objetos cuyo *tiempo de impacto* esté estimado para dentro de 10 minutos a partir de ahora, difícilmente imponga un tiempo de evolución ahora, pero sí probablemente dentro de 9:59:59 minutos. Debido a esta naturaleza, se reconoció la necesidad de utilizar una estructura que permita almacenar la información de las colisiones ya detectadas, obtener rápidamente el menor *tiempo de impacto* almacenado y permitir recalcular, insertar y borrar nodos de la estructura en forma eficiente. La *cola de prioridad* surgió entonces como la estructura natural que permite alcanzar estos resultados.

5.9 Respuesta física utilizando impulsos

El resultado de la *fase de detección fina* es la determinación del mayor tiempo en que todo el sistema puede evolucionar sin que ocurra alguna colisión. Obtenido ese valor temporal, el sistema entonces procede a mover cada objeto hacia su nueva posición. Sin embargo, una vez avanzado hasta ese momento, no es posible garantizar que el par de objetos que lo determinaron se encuentren en una real situación de impacto. En una situación perfecta, el tiempo de evolución debería determinar el momento exacto en que los objetos del tope de la cola *TOI* entran en colisión. No sucede así debido al carácter aproximado del cálculo utilizado y que se describe en el apéndice B1 (ver también [MIRT96]). Únicamente en condiciones especiales (cuando los objetos carecen de velocidad angular), este cálculo determina exactamente el momento en que los objetos entran en contacto, pero en el caso general (ver figura 5.11), el tiempo de evolución solamente acercará un poco a los objetos involucrados. La determinación de si los objetos entran o no en contacto se realiza midiendo la distancia entre ellos luego de haber hecho evolucionar el sistema. Solamente será necesario verificar la distancia entre un solo par de objetos que son justamente los que determinaron el tiempo de evolución. El resto se encontrará, por lo menos, a una distancia igual que ellos ya que de no ser así, hubiese sido otro par el que determinara el tiempo de evolución del sistema.

Si la distancia entre los objetos del tope de la cola es mayor que un cierto umbral epsilon, los objetos no se consideran en situación de colisión. El trabajo que debe hacerse entonces es volver a calcular el *tiempo de impacto* entre ellos. Debido a que los objetos se encuentran más cerca, el nuevo *tiempo de impacto* será una aproximación mejor de su real *tiempo de impacto*. El sistema entonces vuelve a aplicar la *fase de detección fina* determinando un nuevo tiempo de evolución repitiendo así esa parte del proceso.

Eventualmente se llegará a una situación en la que el par de objetos del tope de la cola *TOI* se encuentre a unan distancia menor que el umbral epsilon. La cantidad de iteraciones hasta llegar a este punto depende de la relación entre las velocidades angulares de los objetos que sucesivamente van determinando el tiempo de evolución [MIRT96].

Llegados al punto en que la distancia entre los objetos del tope de la cola es menor que el umbral epsilon, los objetos se declaran en colisión y se aplica sobre ellos las funciones de respuesta física. En forma general se puede decir que estas funciones cambian los vectores de movimiento de al menos uno de los objetos con la intención de sacarlos de la situación de colisión[9]. En el caso más general, será necesario realizar este cambio en los dos objetos.

El sistema impulsivo implementado, basado en las formulaciones de Baraff y Mirtich [BARA92,MIRT96], resuelve cualquier interacción entre pares de objetos únicamente mediante la aplicación de fuerzas impulsivas. Ya sea un impacto de alta velocidad como una situación de contacto permanente, la metodología aplica los mismos algoritmos sobre los objetos. En particular, el contacto permanente es simulado mediante la aplicación de pequeños impulsos a una frecuencia elevada. Un bloque que descansa sobre una tabla experimentará una rápida secuencia de pequeños impulsos con la tabla, cada uno de los cuales es resuelto utilizando únicamente información local del impacto de sus puntos cercanos.

[9] Si la colisión sucede entre un objeto móvil y otro fijo, se cambiarán únicamente los vectores velocidad del objeto móvil.

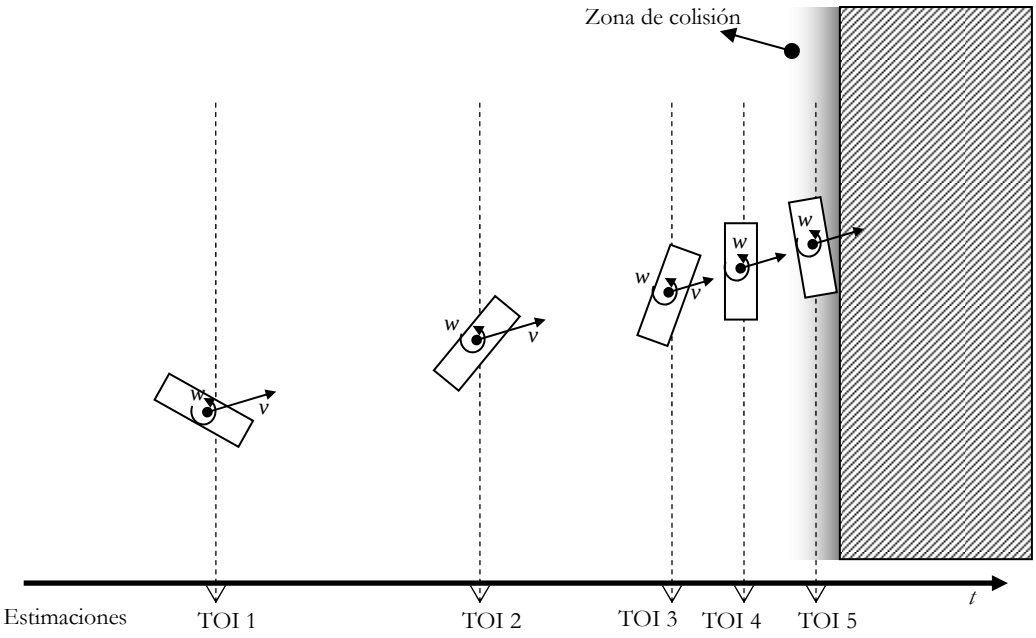


Figura 5.11 *Carácter aproximativo del tiempo de impacto. Cuando la velocidad angular relativa entre dos objetos es mayor que cero, el sistema solo puede estimar conservativamente el tiempo de colisión entre ellos. Una vez llegado a cada estimación, el sistema vuelve a estimar ese tiempo llegando, eventualmente, a dejar a los objetos a una distancia de colisión. El método nunca hace evolucionar el sistema hacia situaciones de interpenetración. Ver apéndice B1.*

El método basado en impulsos se apoya en un conjunto de premisas sobre el impacto entre objetos, las cuales con comúnmente utilizadas por otros métodos. Ellas permiten simplificar el modelo de colisión haciéndolo tratable y eficiente, sin comprometer el resultado global de la simulación. Siguiendo a [ERLE05b], ellas son:

- **Tiempo infinitesimal de colisión:** Implica que la posición de los objetos puede tomarse como constante durante todo el transcurso de una colisión entre ellos. De esta forma la resolución de una colisión puede hacerse mediante un cambio instantáneo de sus vectores velocidad.
- **Hipótesis de Poisson:** Es una aproximación de la perdida de energía y la deformación que sufren los objetos cuando colisionan. Se trata de una hipótesis empírica que captura el comportamiento básico de los objetos durante la colisión y permite establecer una relación simple sobre el intercambio de energía producido. En choques perfectamente elásticos la energía se transfiere a las nuevas velocidades resultantes, mientras que en choques perfectamente plásticos la emergía se pierde completamente y los objetos terminan con velocidad cero. En situaciones intermedias, el intercambio se realiza parcialmente. El llamado *coeficiente de restitución* es una constante asociada a los materiales de los objetos en colisión que gobierna el porcentaje de energía perdida en un choque.
- **Ley de fricción de Coulomb:** En una aproximación al concepto de fricción entre objetos. Como en la hipótesis de Poisson, el protagonismo de este efecto está gobernado a través del llamado *coeficiente de fricción dinámico* entre los objetos en colisión.

Estas hipótesis permiten expresar los impulsos a aplicar, como una expresión algebraica en la que se involucran los vectores velocidad y la distribución de su masa (*tensor de inercia*) de los objetos en colisión. La modificación de los vectores velocidad con esta fórmula garantiza

la no interpenetración en instantes posteriores, además de otorgar un comportamiento realista al movimiento. La derivación de esta fórmula se describe en el apéndice B7.

Cualquier cambio en los vectores velocidad de un objeto tendrá repercusión en el *tiempo de impacto* de él con todos sus objetos cercanos. Es así que, en general, la aplicación de funciones de respuesta tendrá como consecuencia forzar el recálculo del *tiempo de impacto* de todos los pares de objetos de la cola *TOI* en que aparezcan involucrados los objetos que cambiaron sus vectores velocidad. Luego de esto, el sistema podrá volver a determinar el nuevo tiempo de evolución. Es así que el proceso de detección y respuesta termina con el movimiento efectivo de los objetos en forma instantánea. Luego todo el proceso se repite para ese nuevo tiempo de evolución.

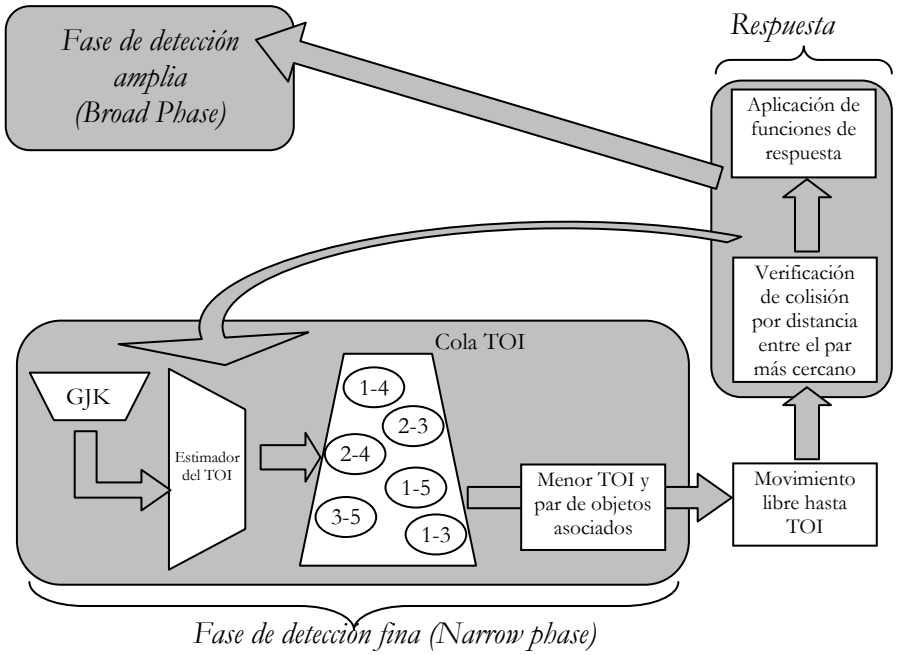


Figura 5.12 *Respuesta a colisiones y su relación con la otras fases el sistema de colisiones: Cualquier modificación en los vectores velocidad de un objeto móvil repercutirá en un recálculo del tiempo de impacto de todos los pares en los que participe dentro de la cola de prioridad. Si los pares de objetos que determinan el tiempo de evolución están suficientemente lejos el, sistema podrá volver a la Fase de detección amplia.*

5.10 Otras consideraciones

Hasta ahora hemos considerado que lo que determina el tiempo de evolución de todo el sistema es la f*ase de detección fina*. Ese tiempo, en definitiva, coincide con *tiempo de impacto* más cercano entre cualquier par de objetos. En una situación en la que el movimiento de los objetos sea lento, este tiempo puede ser considerablemente grande (desde cientos de milisegundos a horas). No se puede esperar hasta ese tiempo para efectivamente mover los objetos y mostrarlos en pantalla, debido a que los usuarios deben percibir, en este tipo de sistemas, que el tiempo evoluciona al unísono que el suyo y por lo tanto necesitan percibir las posiciones intermedias entre las colisiones. Será necesario generar entonces, imágenes con posiciones intermedias a una cadencia superior a 30 *cuadros por segundo* para crear la sensación de animación. Estas imágenes estarán asociadas a resultados físicos también intermedios respecto a cada cambio sustancial en la dinámica, impuesto por el sistema de colisiones o por la interacción del usuario. Esto impone una capa externa al sistema de colisiones que gobierne estos aspectos. Los algoritmos implicados se detallan en el apéndice B6.

6

Diseño y Generación de Escenas

6.1 Introducción

Los motores 3D con las características del desarrollado en nuestro proyecto, representan en pantalla entornos que, generalmente, son diseñados usando un editor que permite modelar en forma interactiva y visual los elementos que componen dicho entorno: luces con sus intensidades y colores, texturas, figuras del decorado, personajes, etc [GTKR03; UNRE03; VALV03].

Desde el comienzo del desarrollo del proyecto se tuvo claro que no era un objetivo diseñar e implementar este tipo de herramientas, por lo que surgió la necesidad de seleccionar uno de los ya existentes como herramienta base de diseño. Se reconoció también la necesidad de entender el formato de almacenamiento de la información generada para poder procesarla y adaptarla a nuestras necesidades

6.2 Análisis de requerimientos

El primer requerimiento de esta etapa fue determinar el editor de niveles que se ajuste más a nuestros requerimientos.

Las principales necesidades que debería cubrir el editor elegido son:

- Conocimiento del tipo de aplicaciones creadas con él para tenerlas como referencia.
- Conocimiento del funcionamiento del editor para facilitar la tarea del diseño de los niveles.
- Cantidad y calidad de la documentación disponible.
- Cantidad y variedad de ejemplos disponibles.
- Costo y forma de licenciamiento.
- Que esté técnicamente actualizado.

Los editores que consideramos fueron: Hammer editor [HAM03], UnrealED [UNED03] y GtkRadiant 1.4 [GTKR03]. Analizando estos puntos llegamos a la decisión de considerar al editor GtkRadiant 1.4 del juego Quake 3 Arena como editor base de nuestro proyecto. Los otros editores no disponían de suficiente información especialmente aquella referida al formato de almacenamiento de las escenas generadas con ellos.

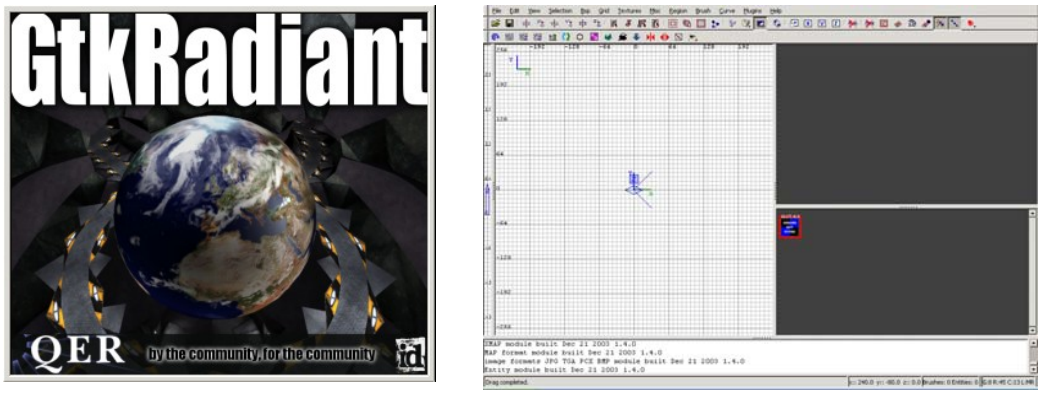


Figura 6.1 *El editor de niveles GTKRadiant 1.4*

Este editor cubre con mayor amplitud nuestras necesidades debido a las siguientes características:

- El Quake 3Arena es un juego que pertenece a una serie conocida y con el que todo el equipo del proyecto estaba familiarizado [IDSO03].
- En Internet se encuentra una comunidad de muchos usuarios del juego que proveen de un importante volumen de información, aplicaciones complementarias, código, mapas, personajes, etc.
- El código fuente del juego fue liberado bajo licencia GPL [GPLI03] por parte de *Id Software*, la empresa responsable del mismo [IDSO03].

Una vez elegido el editor de niveles a utilizar, procedimos a analizar las herramientas que éste brinda para el desarrollo de los niveles, así como también el formato de almacenamiento de la información que genera.

6.2.1 Diseño de niveles en GtkRadiant

Con la aparición del juego Quake III Arena, Id Software hizo público el código fuente de un editor que permite editar escenas (o niveles, como ellos los denominaron) ya existentes en el juego así como también crear las propias. Ese editor, llamado QERadiant [QERA03] fue luego ampliado por la comunidad de usuarios, y portado a diversas plataformas, utilizando para ello la librería gráfica GTK [GTKL03]. Como resultado surgió un editor compatible con QERadiant que se llamó GtkRadiant, cuya versión 1.4 es la que se utiliza como editor de escenas de nuestro motor.

El editor de niveles GtkRadiant tiene como bloque constructivo principal a los *brushes*. Un *brush* es básicamente un volumen formado por polígonos. Se pueden definir volúmenes con un número arbitrario de lados, con bases triangulares, hexagonales, etc., a partir de volúmenes básicos predefinidos como son: conos, cilindros, parches y esferas, todos aproximados por una cierta cantidad de planos (Ver figura 6.2).

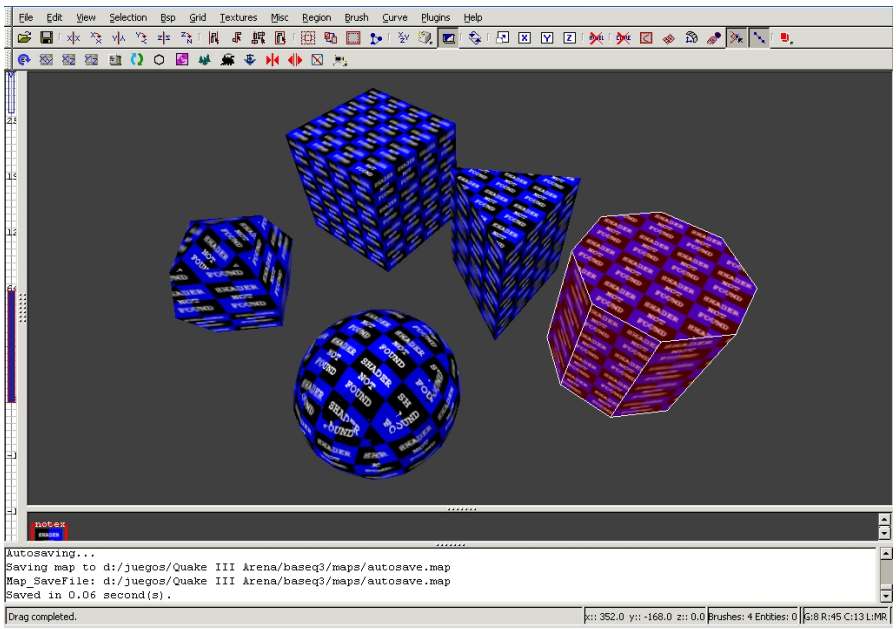


Figura 6.2 *Ejemplos de Brushes en el GTKRadiant 1.4*

Para flexibilizar el proceso de diseño, GtkRadiant posee además herramientas que permiten realizar operaciones *CSG* (*Constructive Solid Geometry*) como:

- Ahuecar
- Sustraer
- Unir

También tiene la posibilidad de incluir, entre otras cosas:

- Luces, especificando la intensidad y el color.
- Atajos a modo de teletransportadores.
- Inclusión de modelos animados.
- Scripts para el manejo de eventos y para el control de personajes.
- Especificación y manipulación de texturas tanto a nivel de objetos como a nivel de polígonos.

De todos los elementos que se pueden especificar en el editor, consideramos para el desarrollo del proyecto sólo a los siguientes elementos:

- Los volúmenes con sus texturas.
- Las luces con su intensidad y sus colores.

Muchos de los elementos que se pueden definir en el editor GtkRadiant son para el control de eventos y manejo de enemigos y personajes secundarios del juego Quake 3 Arena, por lo que no fueron tomados en cuenta por estar pensados para diseñar el juego en sí y no el motor.

La información del editor GtkRadiant se almacena en archivos con extensión .map, el archivo es de formato texto y la descripción de los elementos esta estructurada. Una descripción detallada de este formato se encuentra en el apéndice C1.

6.2.3 Optimización de la visualización

El último paso de esta etapa consistió en procesar la información de la escena (almacenada en los archivos .map), y generar el ordenamiento de la información geométrica de forma tal que se adecue a nuestras necesidades. El hecho de tomarla directamente del archivo .map en el motor 3d implica la necesidad de realizar una serie de cálculos costosos para determinar la geometría e información de las texturas y luces. Esto es un proceso poco eficiente si queremos obtener una animación interactiva.

Los juegos como el *Quake 3 Arena* se basan en simular entornos arquitectónicos cerrados cuya característica principal es la de contar con elementos de gran oclusión (un *oclusor* en nuestro contexto es un elemento que no permite ver a través de él). Esta característica permite implementar una optimización basada en celdas y portales. Una *celda* es una porción de espacio 3d delimitado por geometría que representa a un *oclusor* o por portales. Los portales son las secciones resultantes de un plano al que se le restaron las zonas coincidentes con polígonos de *oclusores*. Muchas veces se intenta que los portales coincidan con puertas y ventanas, aunque esto no es obligatorio.

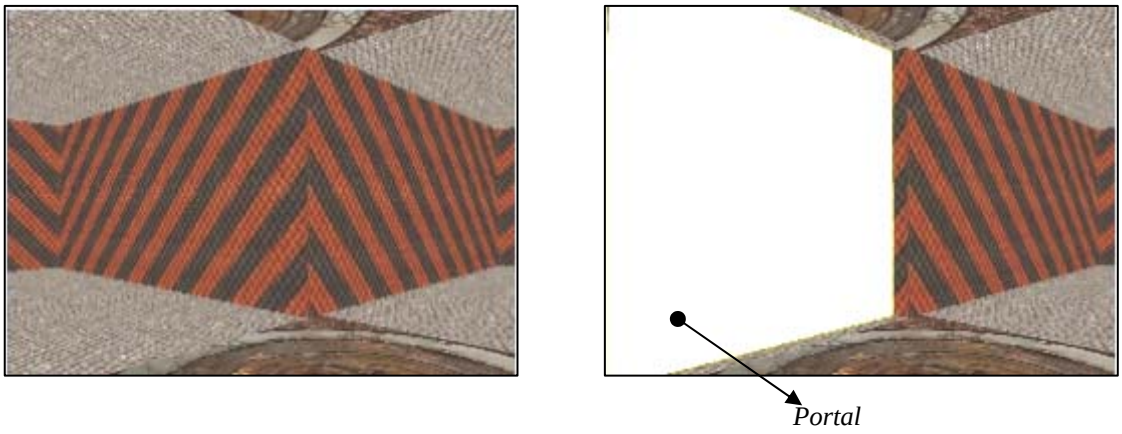


Figura 6.3 *En la imagen de la izquierda se aprecia una parte del modelo de una habitación cerrada en forma de cruz. En la imagen de la derecha se aprecia como, en la intersección del pasillo, se genero una división. Las partes del plano divisor que no estaban superpuestas con oclusores son los portales, representados aquí por la sección blanca. La única forma de acceder a la otra parte del modelo es atravesando el portal (asumimos que en la simulación no se permite atravesar oclusores)*

La optimización en base a celdas y portales se basa en que en entornos con grandes *oclusores*, al estar el observador en una habitación o estancia determinada, la cantidad de espacio visible es reducida en comparación al total del entorno. Si tomamos como ejemplo un edificio, veremos que un observador situado, por ejemplo, en un quinto piso en alguna habitación de un apartamento será capaz de ver e interactuar con una reducida porción de todo el edificio, siéndole imposible ver lo que esta pasando habitaciones de otros apartamentos y mucho menos lo que sucede en otros pisos.

En las simulaciones de entornos tridimensionales hay que realizar varios tipos de tareas y cálculos, algunos exigentes en lo referido al tiempo de cómputo. Si, como hemos visto, la porción visible del conjunto a representar es relativamente más pequeña en comparación con el total, sería interesante optimizar el proceso de visualización de modo de no gastar recursos computacionales escasos.

Para poder realizar esta optimización existe una técnica para la cual es necesario organizar la información geométrica del entorno a visualizar de manera de descomponerla en un conjunto de *celdas* comunicadas por un conjunto de portales. En esta técnica la única forma que se tiene para pasar de una *celda* a otra es a través de un portal, o sea que los portales comunican a celdas adyacentes. Para cada celda se calcula su *PVS* ó conjunto potencialmente visible, que determina el conjunto de celdas que serian visibles al menos en forma potencial por un observador situado en una celda determinada. Por tanto, para asegurar una visualización coherente, alcanza con representar en pantalla solamente al *PVS* de la celda en la que se encuentra el observador.

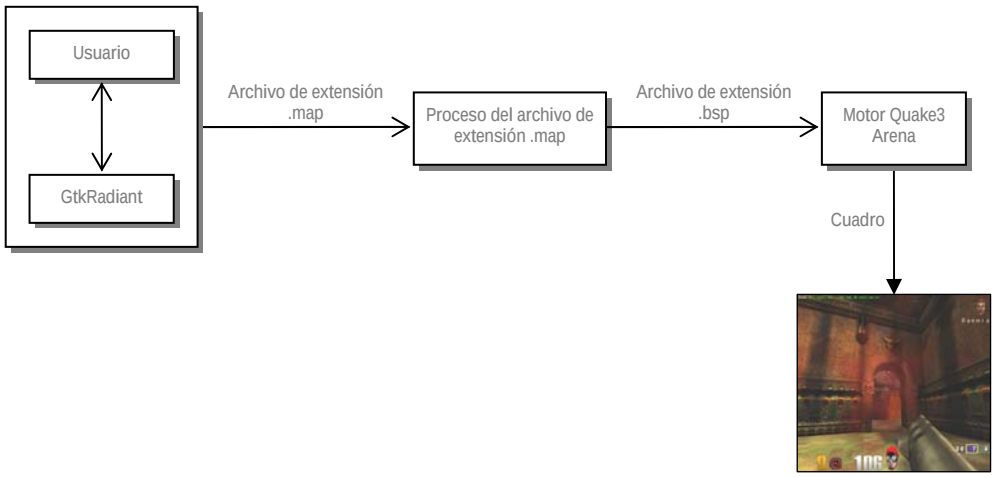


Figura 6.4 *Ciclo de creación de un nivel para el juego Quake 3 Arena.*

En el diagrama de la figura 6.4 se detalla el ciclo requerido para crear un nivel y visualizarlo en el motor del Quake 3 Arena. En primer lugar el diseñador crea un nivel en el editor GtkRadiant. Luego, el editor lo procesa y genera un archivo .bsp. Ese archivo .bsp es el que toma el motor para representar la escena en el juego.

El punto clave en el ciclo anterior es el procesamiento del archivo map para obtener el archivo .bsp. En ese procesamiento se realizan varias tareas entre las cuales destacamos el cálculo del aporte de las luces a la escena, la generación de los *lightmaps*, la descomposición del nivel en celdas y portales y el cálculo de los *PVS* de cada celda.

Los *lightmaps* son una forma de representar el efecto de las diferentes luces estáticas sobre el entorno. Si bien producen un buen efecto no se compara con la sensación que puede producir el tratamiento de las luces en tiempo real [MOLL02l].

Para la descomposición del nivel se construye un *árbol BSP* (partición binaria del espacio, por su sigla en ingles) con los polígonos que componen la geometría del entorno y luego se lo procesa para determinar las celdas y los portales. Como resultado se obtiene un conjunto de celdas comunicadas por portales en una estructura que llamamos *grafo celda-portal*.

En el *grafo celda-portal* los nodos son las celdas, hay una arista entre dos nodos del grafo si existe un portal que hace posible la transición entre las celdas.

Una vez determinada esta descomposición, se realiza el cálculo de *PVS* para cada celda, el cálculo se basa en las técnicas descritas en [TELL92].

Las técnicas de generación de *lightmaps* y de cálculo de los *PVS* requieren de una gran cantidad de cálculos que crecen en gran medida con la complejidad de la escena a procesar. Esto provoca que en niveles grandes y complejos se requiera un tiempo considerable entre que se termina con el nivel y esta disponible para su visualización. También es de destacar que las técnicas de iluminación y sombras soportada por el motor original de Quake 3, no generan resultados comparables a un sistema de iluminación en tiempo real debido a la apariencia estática que proveen.

En nuestra propuesta, el ciclo requerido para crear un nivel y visualizarlo en el motor del proyecto es muy similar al ciclo original sugerido por Id Software. Las grandes diferencias se incorporan en dos puntos claves: la etapa de procesamiento del archivo map y en el propio motor.

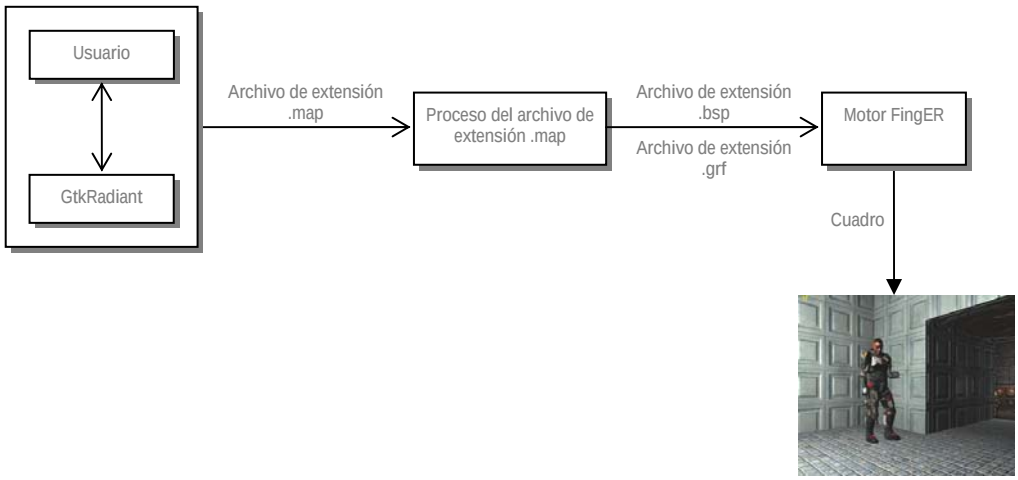


Figura 6.5 *Ciclo de creación de un nivel para nuestro proyecto*

En la etapa de procesamiento se omite el cálculo del aporte de las luces a los elementos del nivel así como la generación de *lightmaps*. También se omite el cálculo de los conjuntos *PVS*. Lo que sí se mantiene es la descomposición del nivel en celdas y portales utilizando la técnica de *BSP*.

Las etapas omitidas se trasladan al propio motor. No es necesario calcular el aporte de las luces y generar los *lightmaps* porque la apuesta es a tener un sistema de iluminación dinámico.

El cálculo de los conjuntos *PVS* tampoco es necesaria ya que se desarrolló e implementó una nueva aproximación a la determinación de las celdas visibles, descrito en [LUEB95]. La nueva aproximación se basa en realizar la selección de celdas a visualizar en tiempo de ejecución. Para poder aplicar esta técnica es fundamental contar con la descomposición del nivel en celdas y portales

En el archivo de extensión .bsp se almacena la definición de las particiones del espacio representadas por planos de corte, información necesaria para posicionar objetos en el nivel y determinar a que celda pertenece cada elemento. En el archivo de extensión grf se almacena la definición de las celdas con su geometría, sus luces, sus modelos y la "relación" con otras celdas a través de los portales, que, como mencionamos, son los que permiten poder "pasar" de una celda a otra.

6.3 Arquitectura e implementación

Para poder cumplir con la etapa de procesamiento, se debió diseñar un proceso que a partir de un archivo de extensión map creado con el editor GtkRadiant generara dos archivos: uno con la información de la partición espacial y el otro con el detalle de la geometría de las celdas y su interrelación por medio de portales.

Los principales elementos que dan soporte al proceso son:

- **Parser de archivos con extensión map:** modulo que valida e interpreta el contenido del archivo de entrada de acuerdo al formato de archivos de extensión map.

- **Generador de volúmenes (brushes):** modulo que determina la geometría de un volumen (las coordenadas de todos sus vértices) dado el conjunto de planos en donde están contenidas sus caras.
- **Generador de coordenadas de textura:** modulo que determina las coordenadas de textura de una cara de un volumen dado el camino a la textura parámetros de control de la textura.
- **Generador de BSP:** modulo que descompone la escena en un árbol binario y que permite, en base a esa descomposición, determinar las celdas, los portales y la relación entre celdas.
- **Escritor de archivos:** modulo que escribe los archivos resultantes del proceso a partir de la descomposición de la escena en celdas y portales.

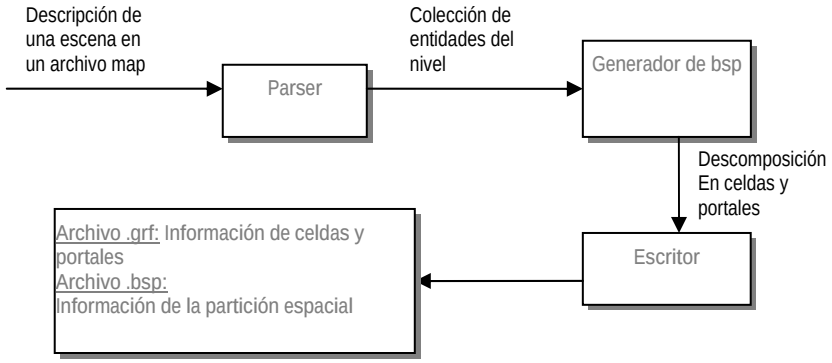


Figura 6.6 *Arquitectura del traductor de archivos .map a .grf y .bsp*

La figura 6.6 representa el flujo principal del proceso con la identificación de las etapas más significativas.

A continuación se detalla cada una de las etapas presentadas junto con sus subprocesos más importantes.

6.3.1 Parser de archivos .map

La parte del proceso que obtiene la información del nivel del archivo .map se puede resumir en el siguiente diagrama:

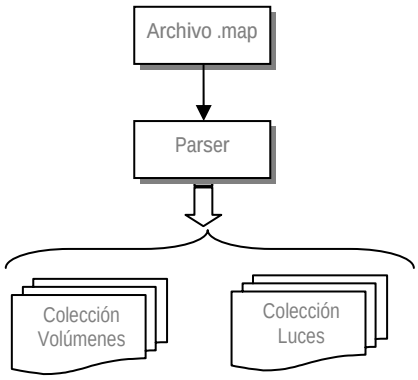


Figura 6.7 *Clasificación de elementos realizada por el parser*

La fuente de datos del proceso es el archivo con la definición de la escena (archivo .map) y la salida es un conjunto de colecciones de los elementos que procesara el motor. Estos son la colección de volúmenes que conforman la "arquitectura" estática de la escena y la colección de luces utilizadas para iluminar la escena.

El modulo de *parser* interactúa con otros módulos para ir generando información, a medida que va reconociendo los distintos componentes definidos en la gramática que define el formato .map.

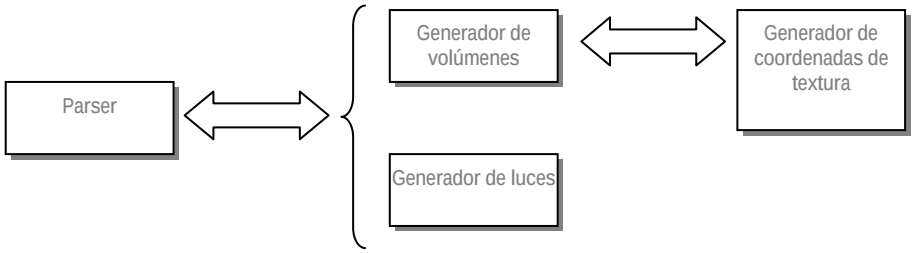


Figura 6.8 *Parser y generadores*

La generación de información se realiza por los módulos Generador de Volúmenes y Generador de Luces.

6.3.2 Generador de luces

Lo que definimos en el diagrama anterior como Generador de Luces es simplemente la identificación de los componentes que las definen para luego almacenar la información. Los datos obtenidos serán procesados en la etapa de generación de la descomposición *BSP*.

6.3.3 Generador de volúmenes

Es en la generación de los volúmenes es en donde está el mayor trabajo de esta etapa del proceso. Los volúmenes se describen en el archivo de extensión map, en base a una línea por cada cara. En esa línea se especifica el plano que contiene la cara, el archivo de textura y los parámetros de control de la aplicación de la textura. Estos parámetros permiten especificar exactamente la apariencia final de ella en la cara.

Luego de que el *parser* reconoce la aparición de una línea se crea, internamente, un polígono conteniendo la información definida por la cara y la textura. Al terminar de reducir toda la información que definen las caras, el *parser* reduce la expresión que define a todo el volumen iniciando el proceso de cálculo de su geometría.

El proceso de cálculo de la geometría se describe en el apéndice C4 mediante seudo-código.

Con estos cálculos se completa la interpretación de la información que define a la geometría de la escena. Resumiendo, se parte de la información de los planos de las caras y de parámetros de control de las texturas y obtenemos volúmenes completamente definidos en base a la definición de todos sus vértices y del cálculo específico de las coordenadas de textura que le corresponde a cada uno.

6.4 Generador de BSP

Como resultado del proceso de *parser*, se obtiene un conjunto de luces y volúmenes que definen la escena. El generador de *BSP* es el encargado de procesar esta información para crear el *grafo celda-portal*.

La base para obtener estas estructuras es aplicar el algoritmo de división espacial *BSP*. Los procesos principales con los que interactúa el modulo de Generación del *BSP* son:

- selección del plano divisor
- clasificación de elementos del nivel
- divisor de volúmenes
- cálculo de portales
- resta de polígonos
- procesamiento de portales
- posicionar portales y generar conexiones

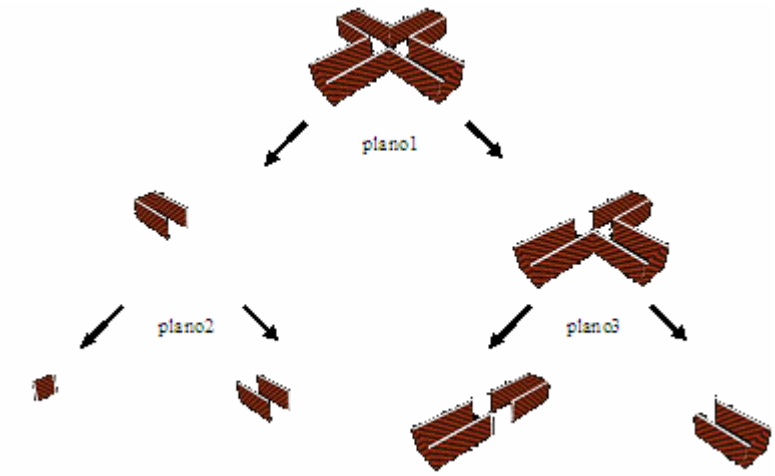
A continuación se presenta una visión esquemática del algoritmo de división espacial *BSP*.

1. Se parte de una representación tridimensional a la que se quiere dividir en particiones:

2. Se selecciona un plano por el cual dividir la representación y se generan dos subsecciones disjuntas:

3. Se repite la operación con cada subsección por separado:

El proceso se repite recursivamente hasta que se llega a la condición de parada, la cual puede ser el llegar hasta una determinada cantidad de polígonos por sección o hasta que se utilicen como planos de división todos los polígonos. La elección dependerá del objetivo por el que se hace la división espacial, generándose la siguiente clasificación:



A

continuación se describen los algoritmos de los principales procesos que se han detallado y luego se describen los algoritmos de generación del *árbol BSP* y de descomposición en celdas y portales.

6.4.1 Selección del plano divisor

Para la selección del plano divisor no existe una fórmula única. Como referencia se sugiere consultar los papers de Airey y Teller en donde se encuentran diferentes *heurísticas* y su combinación [AIRE90; TELL92].

En el proyecto se implementaron dos *heurísticas* para la selección del plano divisor. La primera se basó en la minimización de los cortes de polígonos y la segunda en lograr una cantidad balanceada de elementos en las subsecciones. Si bien no se realizaron comparaciones para determinar la conveniencia de una sobre la otra, ya quedaron implementadas para poder realizarlas en el futuro.

La heurística de minimización de cortes busca seleccionar el plano que corte la menor cantidad posible de volúmenes. De esta forma se minimiza la necesidad de calcular la división de volúmenes, ahorrando tiempo de procesamiento y memoria.

La heurística de balanceo busca seleccionar el plano que genere la división más balanceada posible en términos de los volúmenes que contiene cada una. Se obtiene una división que mantiene la carga de elementos a manejar en cada subsección lo más balanceado posible, repartiendo lo más equitativamente posible el "esfuerzo" de la representación, el manejo de las subsecciones en los cálculos y posteriormente en tiempo de ejecución en el dibujado.

Un punto a considerar es que en la elección del plano divisor, no se han tomado en cuenta las luces que componen la escena.

Un seudo-código de estas *heurísticas* se describe en el apéndice C5.

6.5 Clasificación de elementos de escenas

Una vez que se ha seleccionado el plano por el cual se realizará la división del espacio, se procede a la clasificación de los elementos la escena con respecto a su relación con el plano divisor. En este punto del proceso, los elementos que mayor atención requieren son los elementos que conforman la geometría de la escena, ya que son los que se verán realmente afectados por la división espacial.

En lo que respecta a las luces, se las trata como elementos puntuales y por tanto no requieren mayor trabajo que el determinar la región del espacio a la cual pertenecen. En futuros trabajos se podrá contemplar la complejidad que aportan a cada subregión e incorporarlos en los diferentes pasos del proceso.

Los elementos que conforman la geometría del nivel son los más afectados por la división espacial ya que no sólo se clasifican para determinar a que subregión pertenecen sino que de detectarse que un volumen es cortado por un plano divisor, se realiza la generación de dos nuevos volúmenes, cada uno perteneciente a una de las subregiones. De este proceso de división de los volúmenes también se obtiene el polígono intersección entre el volumen y el plano divisor. Estos polígonos son fundamentales para el posterior proceso de cálculo de los portales. Un caso particular dentro de lo mencionado es cuando el plano divisor no corta al volumen sino que tan solo contiene a una de las caras del volumen. En este caso no se divide el volumen pero si se registra la cara en cuestión para el cálculo de los portales.

6.6 Divisor de volúmenes

El proceso de división de volúmenes esta basado en [LIDLA86]. Si bien el algoritmo aplicado no alcanza la complejidad del que se describe en el paper, sí se utilizó como fundamento y referencia.

La idea del proceso de división se basa en que un plano divide el espacio en dos subregiones, una en la que los puntos del espacio guardan una distancia positiva o nula con respecto al plano y otra en la que los puntos del espacio guardan una distancia negativa con respecto al plano. El volumen se procesa con respecto al plano obteniéndose una de las siguientes situaciones:

- Todos los puntos del volumen se encuentran en el semiespacio positivo, por tanto se marca al volumen como volumen positivo y el polígono intersección queda vacío.
- Todos los puntos del volumen se encuentran en el semiespacio negativo, por tanto se marca al volumen como volumen negativo y el polígono intersección queda vacío.

  Un caso particular de ambas situaciones se da cuando todos lo puntos pertenecen a una de las subregiones excepto los puntos de una de las caras, los cuales coinciden con el plano divisor. Ante esta situación el resultado es tener un volumen positivo o negativo y un polígono intersección que coincide con la cara que esta sobre el plano divisor.

- Ambos subespacios contienen al menos un punto del volumen, por tanto luego de la división del volumen original se tendrá un volumen positivo, un volumen negativo y un polígono intersección

Este algoritmo tiene como resultado un volumen positivo, un volumen negativo y un polígono intersección con respecto al plano divisor. Alguno de estos datos puede contener un valor neutro si corresponde según la relación volumen / plano.

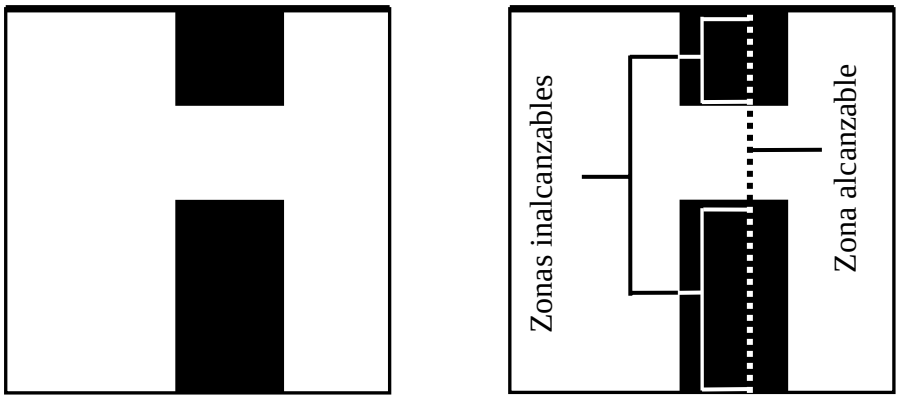
En el apéndice C7 se encuentra el seudo código del algoritmo utilizado.

6.7 Cálculo de portales

En este punto se tiene una representación de una escena que ha sido divida y clasificada en base a un plano divisor. Hay geometría, luces y objetos pertenecientes al subespacio positivo, y geometría, luces y objetos pertenecientes al subespacio negativo. También se tiene una serie de polígonos que surgieron de la intersección entre la geometría de la escena y el plano divisor.

El nexo entre estas dos subregiones de la escena es el plano divisor. Pero si consideramos que los volúmenes que definen la geometría representan elementos sólidos (por ejemplo paredes), no cualquier parte del plano divisor permitiría "conectar" las subregiones del nivel. En particular no seria posible "pasar" de una región a otra a través de los polígonos intersección volúmenes / plano. Por tanto para establecer las secciones del plano divisor que comunican las subregiones de la escena se requiere el cálculo de los portales.

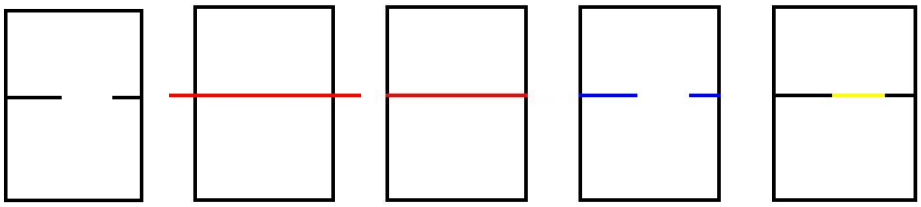
Como ejemplo de lo anterior considerar las siguientes figuras:



En la figura de la izquierda se aprecia una habitación vista desde arriba que tiene un par de volúmenes (rectángulos negros) que representan muros y por tanto se considera que no es posible atravesarlos. En la figura de la derecha se aprecia como una línea punteada al plano divisor el cual divide a ambos volúmenes en dos secciones, una a la izquierda y otra a la derecha del plano. La intersección de ambos volúmenes con el plano genera dos polígonos los que están representados en la figura como las zonas inalcanzables y en la descripción del párrafo anterior son identificados como los "polígonos intersección volúmenes / plano". Se los considera inalcanzables porque están contenidos en volúmenes que se consideran como sólidos a efectos de la simulación. Teniendo esto en cuenta la única parte que queda para "pasar" de una sección a otra de la escena dividida es la zona señalada como alcanzable que se denomina portal.

Para el cálculo de los portales se limita el plano divisor a un polígono calculado a partir de la intersección del plano divisor con el *volumen acotante* de toda la escena (en nuestro caso el *volumen acotante* se calcula como un *AABB*), a este polígono lo llamaremos polígonoNivel. En definitiva lo que llamamos portales es el conjunto de polígonos resultantes de restarle al polígonoNivel el conjunto de polígonos intersección volúmenes / plano.

Un ejemplo bidimensional del algoritmo para el cálculo de portales es el siguiente:



(1) (2) (3) (4) (5)

1. Se considera la vista desde arriba de una habitación.
2. Se selecciona el plano divisor representado por la línea roja
3. Se crea el polígonoNivel como la intersección del *volumen acotante* del nivel y el plano divisor
4. Se calculan los polígonos intersección plano divisor / volúmenes del nivel
5. Se le restan al polígonoNivel los polígonos intersección quedando el conjunto de polígonos portales, que en este caso es un solo polígono. Ese es el portal (amarillo en el diagrama)

6.7.1 Resta de polígonos

Para la realización de la resta de polígonos, necesaria para calcular los portales a partir del polígonoNivel y del conjunto de polígonos intersección plano divisor / volúmenes, se intento el desarrollar nuestras propias funciones. Si bien se logro una aproximación a la solución, se detectaron problemas en casos complejos que impedían la utilización de ese código en el proyecto. Dadas esas dificultades se opto por utilizar una biblioteca de terceros que realizara las resta de polígonos.

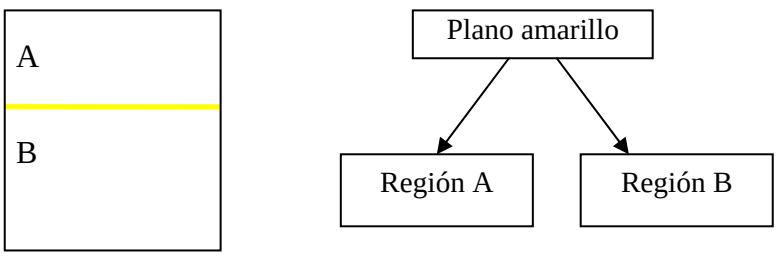
La biblioteca seleccionada pertenece al proyecto "Generic Polygon Clipper", desarrollado por Alan Murta de la Universidad de Manchester [MURT03].

Esta biblioteca solucionó muchos de los problemas encontrados en nuestro algoritmo pero aún se detectaron inconvenientes en ciertas situaciones límites que solucionamos realizando un análisis al resultado de la operación.

6.7.2 Procesamiento de portales

Una vez que se ha completado el cálculo de las divisiones espaciales y esta completo el *árbol BSP*, se debe realizar un procesamiento de los portales para garantizar la correcta vinculación entre celdas a través de los mismos. Dado que una vez calculado el conjunto de polígonos que representan a los portales, cada una de las subregiones continuó el proceso de división, hay que procesar los portales calculados según las divisiones que se produjeron en la continuación del proceso.

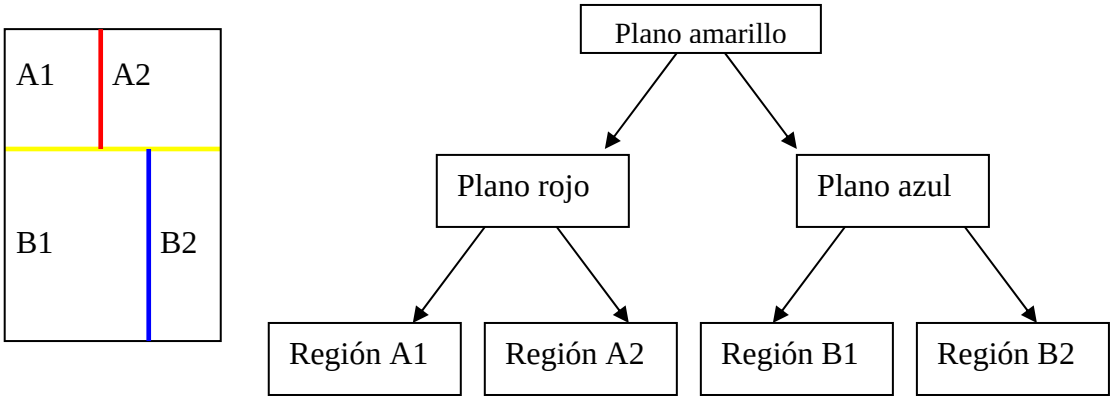
Como ejemplo presentamos la siguiente situación:



En el primer paso de la generación del *árbol BSP* se obtiene la siguiente división en base al plano representado por la línea amarilla.

Como no hay polígonos coincidentes con el plano el portal que comunica, las dos regiones es todo el plano amarillo.

Cada una de las subregiones continúan con el proceso generando la siguiente división del espacio:



En este caso los portales se corresponden con los planos divisores. Esta claro, por la descomposición de la escena, que el portal rojo conecta a la región A1 con la región A2 y que el portal azul hace lo propio con las regiones B1 y B2. Lo que no es posible determinar de forma directa es la forma en que el portal amarillo conecta a las regiones Ai con las regiones Bi. Esta situación es la que hace necesario el procesamiento de los portales pertenecientes a los nodos que no son hojas del *árbol BSP*.

El procesamiento de los portales consiste en tomar los portales de un nodo que no es hoja y hacerlos "bajar" por el subárbol derecho y luego por el subárbol izquierdo dividiéndolos con respecto al plano divisor de cada nodo de los subárboles.

En nuestro ejemplo tomaríamos el portal amarillo y lo dividiríamos respecto al portal rojo. Esto genera dos portales amarillos uno como frontera de A1 y otro como frontera de A2. Se toman estos dos nuevos portales y los dividimos con respecto al portal azul. Esto genera que el portal frontera con A1 quede igual y que el portal frontera con A2 sea dividido en dos portales uno formando parte de la frontera con B1 y el otro como frontera con B2. Con este proceso el portal amarillo original fue dividido en tres partes y ahora es clara la comunicación entre las celdas.

La celda A1 se comunica con la celda A2 a través del portal rojo y con la celda B1 a través de la primer sección del portal amarillo. La celda A2 se comunica con la celda A1 a través del portal rojo, con la celda B1 a través de la segunda sección del portal amarillo y con la celda B2 a través de la tercer sección del portal amarillo. La celda B1 se comunica con las celdas A1, A2 y B2 y por último la celda B2 se comunica sólo con las celdas B1 y A2.

En esta sección simplemente se realiza la división de los portales de los nodos que no son hojas, la nueva colección de portales es almacenada en el nodo original. Aún no se realiza la vinculación entre las celdas, esa operación es realizada en la siguiente sección.

6.7.3 Posicionar portales y generar conexiones

Una vez que se tienen todos los conjuntos de portales divididos con respecto a los planos divisores de ambos subárboles se procede a posicionar los portales en el espacio generando las conexiones resultantes entre celdas.

Continuando con el ejemplo anterior, se tienen tres portales amarillos que conectan de alguna manera a las celdas Ai y Bi. Se toma el primer portal amarillo y se lo hace descender a través del subárbol positivo y del subárbol negativo. El descenso se hace considerando la región a la que pertenece cada portal con respecto a la sucesión de planos divisores de los subárboles. El primer portal amarillo cuando desciende por el subárbol izquierdo se compara contra el portal rojo determinándose que pertenece a la sección izquierda y que por tanto sirve como conexión de la celda A1. Cuando desciende por el subárbol derecho se compra contra el portal azul determinándose que pertenece a la sección izquierda y que por tanto sirve como conexión de la celda B1. De este proceso se determina que la celda A1 esta conectada a la celda B1 y que el nexo es el primer portal amarillo. Se agrega a cada celda la información de con que celda esta conectada y la información del portal que permite dicha conexión. Esto se realiza con cada portal de cada nodo del *árbol BSP* completando así la información que vincula a las celdas entre si.

6.8 Algoritmo de generación del árbol BSP

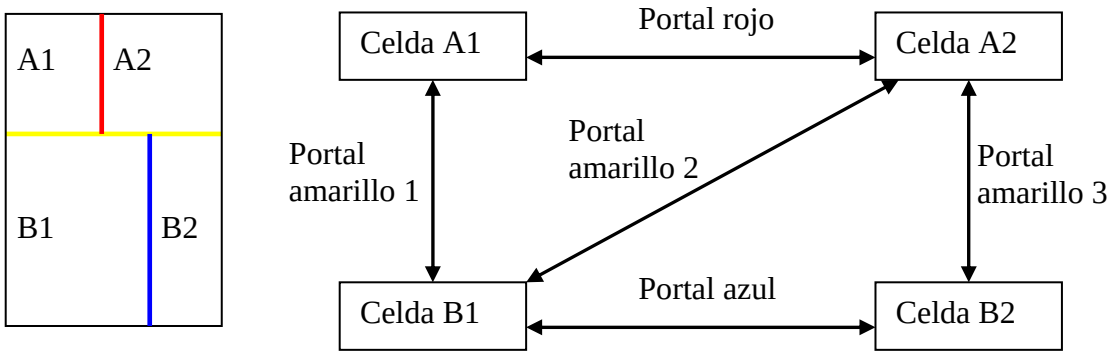
Ya se han detallado las diferentes etapas que conforman el algoritmo para la generación del *árbol BSP*. A continuación se muestra aproximadamente cómo se unen estos componentes en el algoritmo de generación del *BSP*.

Del procesamiento del archivo .map se obtuvo una colección de volúmenes y una colección de luces. Todos estos elementos se agrupan en un único conjunto que representa la escena completa. Como nuestro fin es conseguir su descomposición en celdas y portales, se realiza el proceso que se detalla a continuación:

1. Se selecciona el plano divisor en base a la información contenida en la escena o sección de la escena que se esta procesando.
2. Se clasifican las luces con respecto al plano seleccionado generando dos conjuntos disjuntos.
3. Se clasifican los volúmenes con respecto al plano seleccionado generando dos conjuntos de volúmenes y además un conjunto de polígonos generados por la intersección entre los volúmenes y el plano divisor.
4. Se calculan los portales a partir del plano divisor y los polígonos intersección volúmenes / plano.
5. Con la información generada para cada semiespacio con respecto al plano se repiten los cuatro pasos anteriores
6. Se procesan los portales con respecto a los nodos descendientes del nodo que se esta procesando
7. Se generan las conexiones entre las celdas

Al finalizar el proceso de generación del *árbol BSP* obtenemos la organización de la información de la escena que, entre otras cosas, hace posible optimizar el dibujado en pantalla. Esa organización, que se denominó g*rafo celda–portal*, es el resultado directo de la descomposición *BSP*. Es un grafo dirigido en el que los nodos son las celdas y en donde existe una arista que va de la celda A a la celda B si existe un portal que los conecte.

Tomando nuevamente como referencia el ejemplo que veníamos considerando ante la descomposición de la figura de la izquierda se obtiene el siguiente *grafo celda–portal*:



6.9 Escritor de archivos

Como resultado del procesamiento de los archivos de extensión map obtuvimos una descomposición espacial, el *árbol BSP*, y una clasificación de la información del nivel, el *grafo celda–portal*. Esta información se almacena para que pueda ser utilizada por el motor. La forma que se eligió para almacenar esta información es crear un archivo XML para cada tipo de resultado por lo que se generan dos archivos, uno con la información de la partición espacial y otro con el *grafo celda-portal*.

La información de la partición espacial se puede utilizar para ubicar puntos del espacio en una determinada celda mientras que la información del *grafo celda–portal* sirve para implementar optimizaciones en el momento de dibujar la representación así como también para determinar el pasaje de una celda a otra por parte de objetos. Se da la posibilidad de utilizar la información que se considere necesaria sin tener que cargarla en su totalidad.

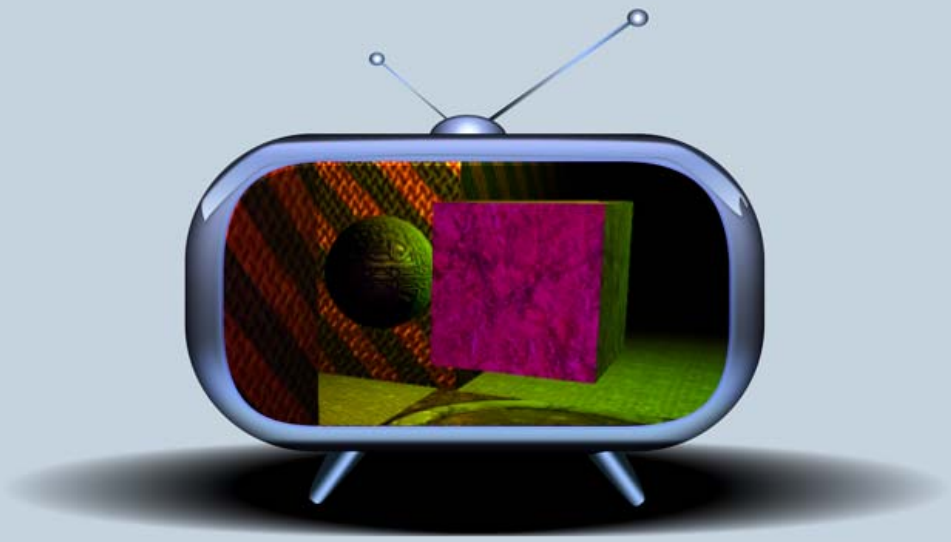
Se decidió grabar toda información en formato XML debido a que este es un formato fácil de leer e interpretar. El formato XML, al ser un formato auto contenido, mejora la comprensión e interpretación de la información además de proveer la facilidad de transformar esa información de muchas maneras que ya están predefinidas por estándares internacionales. Lo malo de utilizar este formato es que se obtienen archivos un tanto grandes en tamaño.

La información de la partición espacial se almacena en archivos con la extensión bsp y los archivos con la información de la clasificación celda–portal se almacena en archivos con extensión grf.

Por una información más detallada de la estructura de los archivos de extensión bsp y grf consultar el apéndice C2 y C3 respectivamente.

7

Resultados Obtenidos, Conclusiones y Trabajos Futuros



7.1 Resultados obtenidos

El principal resultado que se esperaba obtener con este proyecto era la creación de una herramienta flexible y extensible con las características descritas en el capítulo dos, que facilitara la creación de aplicaciones gráficas interactivas en tiempo real y que fuera lo suficientemente abierta como para poder ampliar su capacidad sin necesidad de conocer a fondo su estructura interna. Se presentaría entonces una herramienta con funcionalidades básicas que sirviera de base para continuar agregando funcionalidad haciendo uso de las propiedades del paradigma de programación orientada a objetos. En forma global podemos decir que el resultado obtenido es un motor 3d con las características esperadas. Llegar a ello fue solamente posible luego de alcanzar el siguiente conjunto de resultados.

7.1.1 Grafo de escena y renderer

El objetivo inicialmente planteado fue el de brindar al desarrollador un sistema de descripción de escenas que fuera flexible a la hora de definir sus elementos constitutivos y eficiente a la hora de ser dibujadas. Adicionalmente se agregó, al requerimiento de rendimiento, la posibilidad de definir diversos tipos de efectos visuales, en particular aquellos que soporten iluminación por píxel.

Para lograr estos objetivos se implementó lo siguiente:

- Un grafo de escena basado en dos árboles n-arios, uno de los cuales permite describir las características visuales de los objetos y el otro sus posiciones en el espacio.
- Un sistema de optimización previo *off-line* que analiza los datos del grafo de escena y los transforma en estructuras compatibles con el diseño del hardware de soporte.
- Un sistema de optimización en tiempo real que aplica la misma heurística que el sistema de optimización previo para optimizar el proceso de visualización.
- Un sistema de definición de efectos visuales que permite trabajar tanto con los estados de la API como con programas escritos en lenguajes de sombreado.
- Un sistema de administración de luces que optimiza su aplicación en función de su posición y la de los objetos que ilumina.
- Una arquitectura que le permite al desarrollador inicializar, procesar en tiempo real y dar de baja a todos los elementos que constituyen las escenas tridimensionales. Adicionalmente se implementó un sistema de controladores que permite extender las capacidades del motor sin necesidad de programar aplicaciones específicas.

Se implementaron también técnicas visuales específicas y controladores que permitieron verificar la funcionalidad del diseño y medir su rendimiento:

- Técnicas básicas de texturas y efectos de transparencias.
- Técnicas de iluminación por píxel incluyendo efectos de bump mapping y reflexión especular.
- Controladores de cambio de posición, rotación, animación de texturas y animación de caracteres.

7.1.2 Colisiones y respuesta

- Modelado del movimiento de objetos convexos mediante vectores de velocidad lineal y angular.
- Un sistema eficiente de cálculo del tiempo de impacto entre objetos convexos utilizando el algoritmo *GJK*[CAME97].

- Un sistema de respuesta a colisiones entre pares de objetos utilizando ecuaciones de impulso en el punto de impacto.
- Un sistema de optimización global de verificaciones de contacto utilizando una división espacial basada en *celdas* y *portales*.
- Un sistema de optimización local de verificaciones de contacto basado en mapas de colisiónes y el algoritmo *Sweep & Prune*[BARA92].
- Un sistema de verificación de pasaje por portales basado en la creación de volúmenes shaft para el movimiento inmediato de los objetos móviles.
- Un sistema iterativo de acercamiento al primer contacto basado en el refinamiento del cálculo del tiempo de impacto y la utilización una *cola de prioridad* de Fibonacci.
- Un sistema de optimización de entrada de objetos en celdas basado en el precálculo de árboles de búsqueda por prioridad *PST* [CREI85] para cada celda del entorno.

7.1.3 Diseño y generación de escenas

- Un parser de interpretación de archivos de formato .map generados con el editor libre GTKRadiant.
- Un sistema de creación de brushes basado en la descomposición *BSP* de los entornos interpretados.
- Un algoritmo de creación de objetos poliédricos convexos en función de las ecuaciones de los planos de sus caras.
- Un algoritmo de seccionamiento de objetos poliédricos por planos.
- Un sistema de determinación de geometría sólida para la determinación de portales.
- Un sistema de creación de un grafo celda-portal utilizando la información previamente procesada.

7.2 Conclusiones

Creemos que la principal conclusión que se puede sacar es que este tipo de software es realizable con las estrategias elegidas. Sin embargo, cada una de las tres líneas de trabajo ofrecen conclusiones concretas que pueden ser útiles para futuros trabajos.

7.2.1 Grafo de escena y renderer

- El grafo de escena diseñado brinda un adecuado nivel de control sobre la escena, permitiendo controlar variables como: cantidad de elementos, posición, orientación, visibilidad y aspecto.
- El diseño del renderer conjuntamente con la clase CTecnica permiten crear efectos visuales que, al ser compatibles con programas escritos en lenguajes de *sombreado*, le dan al desarrollador flexibilidad suficiente como para crear efectos de iluminación por píxel. El sistema diseñado también permite crear, entre otros, efectos de texturas y transparencias.
- La optimización en dos etapas del proceso de dibujo brinda un aumento significativo en el rendimiento del motor. La siguiente tabla muestra la mejora obtenida por la aplicación del sistema de optimización previa o *precaching*:

Resolución	Polígonos	Mesh	Luces	Recorrida	FPS
1280 x 1024	2677	307	1	Árbol	88
1280 x 1024	2677	307	1	Precaching	101
1024 x 768	2677	307	1	Árbol	143
1024 x 768	2677	307	1	Precaching	184
800 x 600	2677	307	1	Árbol	206
800 x 600	2677	307	1	Precaching	248

La etapa de optimización en tiempo real, denominada PoolRS, proporciona una mejora adicional de un 5% en promedio:

Resolución	Polígonos	Mesh	Luces	PoolRS	FPS
1280 x 1024	2677	307	1	NO	101
1280 x 1024	2677	307	1	SI	104
1024 x 768	2677	307	1	NO	184
1024 x 768	2677	307	1	SI	195
800 x 600	2677	307	1	NO	248
800 x 600	2677	307	1	SI	258

La aplicación conjunta de las técnicas de optimización da una mejora entre un 20 y un 25% al sistema de visualización.

- La implementación de efectos de luz por píxel que incluyan reflexión especular puede realizarse directamente utilizando un lenguaje de sombreado basado en *Register Combiners* V1.5. Las pruebas realizadas demuestran que si bien la implementación resulta eficiente[10], las limitaciones propias del lenguaje impiden realizar optimizaciones en función de la cantidad de luces involucradas. Las versiones posteriores del lenguaje de sombreado V2.0, 3.0 y 4.0 admiten programas de mayor longitud y brindan posibilidades que pueden ser aplicadas al motor para mejorar el rendimiento de estos efectos.
- El sistema de controladores demostró ser una poderosa herramienta para desarrollar aplicaciones con el motor. Al permitir encapsular funciones que se ejecutan en cada *frame* generado, son el formato más indicado para programar trayectorias, efectos de transición, variaciones de las variables de estado de las aplicaciones, visibilidad, posición de la cámara e incluso sistemas de animación de caracteres (Ver Apéndice A7).

7.2.2 Colisiones y respuesta

- La detección de colisiones puede ser realizada utilizando una partición del entorno en objetos poliédricos convexos. La eficiencia del algoritmo Sweep & Prune junto con el algoritmo GJK de medición de distancias son suficientes para permitir darle esta funcionalidad al motor.
- La información aportada por el algoritmo GJK, unida al modelado de movimiento mediante vectores velocidad, es suficiente para permitir aplicar a los objetos en colisión, ecuaciones de respuesta simple basada en física de impulsos.
- El sistema de avance conservativo en función de eventos (colisiones principalmente) no sobrecarga al sistema cuando hay un número menor a diez objetos en escena a los que se

[10] Probado en una tarjeta de video NVidia GeForce 4 Ti 4200 a una resolución de 800 x 600 x 32

aplica respuesta física. Más de esa cantidad llevan al sistema a tener que hacer demasiadas evaluaciones haciendo ineficiente este método de funcionamiento.

- La incorporación de estrategias de volúmenes shaft para la determinación de pasaje por celdas y como guía para el cálculo de tiempo de impacto entre pares objetos móviles ha resultado efectiva. No hemos detectado falta de coherencia en los movimientos aunque pensamos que sería adecuado realizar mas pruebas en entornos de diferente topología.
- La utilización de árboles PST proveen una adecuada independencia a la complejidad en la obtención de intervalos solapados a un objeto entrante a una celda. Sin embargo podrían darse situaciones en las que la cantidad de intervalos obtenidos sea exageradamente grande. En las pruebas realizadas no se ha detectado este caso ni que los árboles PST constituyan un cuello de botella para el sistema. Comparados con otras técnicas otorgan una mucho mejor eficiencia en promedio (Ver apéndice B4).
- Si bien la utilización de cajas AABB como volúmenes acotantes podría parecer exagerada, especialmente cuando se incluyen objetos alargados en el sistema, no se ha detectado una degradación importante en la performance debido a su utilización. De todas formas sería conveniente verificar el comportamiento con volúmenes mas estrechos.
- La implementación de todos los algoritmos de detección y respuesta podrían incluirse en muchas otras arquitecturas generales ya que la información geométrica involucrada es calculada y utilizada en forma autónoma. Resulta necesaria, sin embargo, una estructura de grafo que represente las celdas y portales con las operaciones básicas.

7.2.3 Diseño y generación de escenas

- La traducción de datos de escenas almacenados en el formato del editor libre GTK Radiant resulta adecuado para ser utilizado dentro del motor aunque la eficiencia y flexibilidad logradas en este proceso pueden ser mejoradas.
- El algoritmo de optimización de la visualización descrito en [LUEB95] resultó importante en la disminución del trabajo enviado al modulo de visualización como era de esperar de acuerdo a los resultados obtenidos con el prototipo bidimensional.

7.3 Trabajos Futuros

Debido a los altos requerimientos de performance que se requieren y al proceso de mejora continua que necesita realizarse en este aspecto, no consideramos que se hayan alcanzado plenamente los objetivos de lograr un núcleo totalmente pulido y optimizado. En lo que sigue, se detallan algunos de los puntos en los que se necesitaría trabajar para lograr este objetivo, así como también una descripción de las extensiones que deberían realizarse para obtener un producto más maduro. Como en el resto del documento, dividimos estas conclusiones en tres partes: renderer, colisiones y generación de niveles.

7.3.1 Grafo de escena y renderer

Mejora en el sistema de cambios de estado. Dentro de los objetivos planteados al implementar el módulo del *grafo de escena* y *renderer* estaba el de ofrecer al programador de aplicaciones una manera flexible de describir escenas. Dentro de estas funcionalidades, la optimización de cambios de estado es una de las prioridades para lograr eficiencia ya que según [ASHI06] este es uno de los cuellos de botella que se deben evitar. La etapa de optimización previa del motor intenta optimizar los cambios de estado agrupando objetos a técnicas, pero no tiene en cuenta cuales son los estados que cada una utiliza. Una mejora en este aspecto consistirá en

implementar un sistema que tome no a las técnicas sino a los estados que utilizan para optimizar sus cambios.

Rediseño del LightManager. En cuanto a los efectos visuales, el sistema desarrollado permite crear nuevos efectos con suficiente flexibilidad como para utilizar programas de *sombreado (shaders)* de generaciones posteriores al utilizado en los efectos básicos implementados. En particular, las nuevas generaciones de *shaders* permiten realizar programas mucho más largos, pudiendo iluminar objetos con muchas luces en una sola pasada. Un cambio de este tipo, que implicaría un aumento significativo de la performance, requiere rediseñar la etapa de optimización de luces del *LightManager*.

Manejo correcto de transparencias. Los efectos visuales que utilizan transparencias requieren un tratamiento especial debido a que su aplicación se realiza combinando los datos con el contenido previamente existente en el *frame buffer*. El motor considera este aspecto generando un *rendereo* posterior de todos los objetos cuyas técnicas asociadas sean de este tipo, pero no ordena especialmente los polígonos, cosa que puede generar errores en la etapa de dibujo. Este problema se resuelve construyendo un *árbol bsp* con el conjunto de polígonos transparentes, proceso que debe ser hecho en tiempo real.

Adición de sombras. La adición de sombras es una de las faltas más significativas del motor. Su implementación implica trabajar a nivel global con los objetos: no es un efecto que se pueda resolver a nivel local debido a que la sombra de un objeto puede afectar a muchos otros. Dentro de las técnicas más comunes de implementación de sombras se encuentran los shadowMaps [CATM74]. Esta técnica fue probada con el motor pero no dio resultados satisfactorios debido a problemas con el *z-buffer*.

Inclusión de técnicas de nivel de detalle. Otra de las técnicas de visualización y optimización muy usadas en motores comerciales es la denominada *Level of Detail*, descrita en el capítulo 2. Este motor no implementa ninguna de sus variaciones por lo que no está especialmente recomendado para trabajar con escenas que contengan muchos objetos geométricos que se encuentren lejos del observador.

Creación de un editor. La unión implementada entre el editor GTK-Radiant y el *grafo de escena* se realiza únicamente a nivel de geometría y no tiene en cuenta luces ni otro tipo de objetos geométricos. Tampoco permite asociar técnicas a objetos por lo que la construcción de escenas debe realizarse, en gran medida, en forma manual. Uno de los trabajos futuros que se plantean es mejorar esta relación o implementar directamente un editor propio que permita trabajar con todos los elementos componentes de una escena.

Expansión de la API. La API que el desarrollador utiliza para crear aplicaciones se fue completando a medida que se creaban las aplicaciones de prueba y las demos del proyecto por lo que es una de las áreas que se consideró de mayor posibilidad de ampliación y que evolucionará como consecuencia natural de la utilización del motor en la creación de nuevas aplicaciones.

7.3.2 Colisiones y respuesta

Rediseño del método de calculo de tiempos de impacto. El sistema de cálculo de *tiempo de impacto* parece funcionar bien en el sentido de dar robustez al sistema. No se detectaron situaciones erróneas en este aspecto y, en condiciones favorables, el sistema es suficientemente rápido como para lograr animaciones interactivas. Sin embargo, el carácter aproximativo de las formulaciones de Mirtich, no parecen adecuarse a situaciones de simulación física en la que más de tres objetos se encuentran uno sobre otro. La escasa distancia entre ellos hace que el tiempo estimado de impacto sea extremadamente pequeño, forzando al sistema a realizar una gran cantidad de iteraciones en su camino hacia el paso de tiempo establecido. La formulación de Redon [REDO04a] podría solucionar parte de este problema al permitir calcular con mayor exactitud el *tiempo de impacto* entre dos objetos.

Modelación de contactos permanentes. La modelación de contactos permanentes en el sistema se hace exactamente igual que la modelación de una colisión de alta velocidad. Las

consecuencias de esto están relacionadas con lo que se decía en el punto anterior y también con la estabilidad del sistema. La resolución de colisiones implica un cambio en las velocidades de los objetos que los hace estar siempre en movimiento, aunque sea imperceptible para el observador. En situaciones de apilamiento de objetos esto se traduce en movimientos vibratorios y falta de estatismo aunque los coeficientes de fricción sean elevados. Guendelman [GUEN03] aborda este problema y provee un cambio en la forma estructural de realizar las verificaciones de colisiones con el fin de solucionar estos problemas. Requiere una modelación concreta de situaciones de contactos permanentes utilizando grafos de contacto.

Modelación de movimientos prefijados. El sistema implementado no permite definir objetos con movimiento fijo, que interactúe además con los objetos físicos.. La implementación de un sistema así permitiría, entre otras cosas, implementar colisiones realistas entre estos objetos y sistemas de caracteres md5. Un camino de estudio seria el indicado por Redon [REDO04a] y Erleben [ERLE05b].

Implementación de restricciones de movimiento. La implementación de un sistema de colisiones física basado en otras metodologías, por ejemplo métodos de penalización o restricciones, permitiría extender sus funcionalidades para generar objetos encadenados y ragdolls. La investigación de la flexibilidad y eficiencia de estos métodos, comparados con el implementado, seria deseable.

Optimización en la ubicación inicial de objetos móviles. La ubicación inicial de los objetos móviles en el mapa se hace mediante una verificación exhaustiva de todas sus celdas. Una opción mejor seria utilizar el *árbol BSP* para ubicar inicialmente estos objetos. Quizás sea una necesidad si la creación de objetos móviles en forma dinámica sucede con demasiada frecuencia.

Utilización de cajas OBB como volúmenes acotantes. Como las cajas acotantes de las celdas son precalculadas, quizás se pueda obtener un mejor descarte en el sistema de colisiones si estas son *OBB* en vez de orientadas con los ejes. El cálculo de intersección entre una *AABB* y un *OBB* es lo suficientemente rápido como para no descartar esta opción.

Fusión de celdas para el sistema de colisiones. La asignación de nuevas celdas de pertenencia a objetos puede ser costoso si el movimiento del objeto es rápido y las celdas generadas son pequeñas. Al haber llevado el *BSP* del mapa hasta su casi máximo nivel, la cantidad potencial de celdas de un mapa puede ser muy grande y una trayectoria podría intersecar muchas de ellas, provocando una asignación de celdas costosa. Una solución podría ser fusionar celdas, desde el punto de vista del sistema de colisiones, creando celdas más extensas. Como en realidad lo que determina la eficiencia de del algoritmo de asignación es la cantidad de intervalos que la trayectoria atraviesa (dentro de una celda), podría usarse ese dato como determinante de hasta donde fusionar.

Optimización en la formación de volúmenes shaft. La creación de volúmenes shaft se hace insertando de a uno los planos formantes. Como a lo sumo podría haber hasta seis planos formantes, este espacio podría previamente reservarse, en vez de ser pedido dinámicamente con new().

Implementación de un manejador de memoria. La implementación de un manejador de memoria permitiría flexibilidad, eficiencia y claridad en el código. Pruebas iniciales nos muestra que la rutina de new() seria 100 veces más lenta que un manejador hecho a mano. También permitiría controlar mejor la memoria utilizada por todo el sistema.

7.3.3 Diseño y generación de escenas

Inclusión de luces en el proceso de división espacial. En la fase de generación de celdas y portales se podría considerar el aporte de las luces en los objetos de la escena para que al momento de generar la división espacial se intente minimizar la cantidad de objetos que son afectados por una fuente de *luz* que pertenece a una región diferente a la del objeto. Este tipo de consideración podría llevar a una simplificación en el árbol de escena y por tanto facilitar el dibujado en pantalla. Con respecto a los objetos fijos, se podría tener en cuenta su complejidad al momento del cálculo del balance entre las regiones.

Heurísticas en la generación de celdas y portales. El algoritmo utilizado para generar las celdas y portales llega, arbitrariamente, hasta el último nivel del árbol BSP. Una mejor decisión seria aplicar las heurísticas propuestas en [LEFE03].

Glosario

AABB: Ver *Caja alineada con los ejes.*

Aliasing: Efecto visual generado al dibujar sobre un dispositivo discreto (asociado a una malla de puntos rectangulares). El efecto se puede apreciar más claramente al dibujar líneas las cuales aparecen como escalones o "dentadas". Antialiasing es el proceso de dibujar objetos con un aspecto borroso (blurr) con el objetivo de ocultar este efecto.

Alpha Blending: Ver *Mezclado alfa.*

Alpha Channel: Ver *Canal alfa.*

Alpha Test: Ver *Test alfa.*

Animación: El concepto utilizado es el de que una cierta cantidad puede variar en el transcurso del tiempo. En un concepto más clásico se refiere a valores geométricos que varían en función del tiempo. La animación por *Key frames* se refiere a cambios completos de la geometría (los *keys*) en función del tiempo. *Morphing* se refiere al cambio en cada uno de los puntos que definen un modelo, en función del tiempo.

Árbol: Estructura de datos que representa una relación jerárquica entre nodos. Cada nodo tiene , como máximo, un predecesor(nodo padre) y cualquier cantidad de sucesores (nodos hijo). Un árbol binario es un árbol cuyos nodos tiene, a lo sumo, dos hijos.

Árbol BSP: Es un árbol binario utilizado para partir el espacio usando planos. El acrónimo *BSP* significa *binary space partition* o partición binaria espacial. La raíz del árbol representa a todo el espacio y tiene asignado un plano relevante a la aplicación. Los dos hijos de la raíz representan a los dos semi-espacios divididos por el plano del padre. Cada hijo puede dividir el espacio que representa usando otro plano significativo y así sucesivamente.

Árbol de búsqueda por prioridad: Es una estructura que mezcla las propiedades de los árboles binarios de búsqueda y las colas de prioridad. En este contexto se usan para determinar eficientemente la relación de solapamiento entre intervalos.

Back buffer: Ver *Buffer de atrás.*

Backface culling: Ver *Culling.*

Broad phase: Ver *Fase de detección amplia.*

BSP: Ver *Árbol BSP.*

Buffer de atrás: Un bloque de memoria en donde la aplicación escribe los píxeles del *cuadro* actual de la escena, mientras el sistema muestra el contenido del *cuadro* anterior dibujado previamente en otro buffer (FRONT BUFFER). Esta técnica se llama *double buffering.*

Buffer de profundidad: Bloque de memoria que almacena la profundidad de los puntos que se dibujan en pantalla. Es usado para ordenar el dibujado de los píxeles.

Bump mapping: Efecto especial que permite apreciar la apariencia de geometría de baja escala que poseen algunos objetos. Se basa en la utilización de una textura mapeada, que representa esa variación geométrica.

Caja alineada con los ejes: Es una caja cuyos ejes son paralelos a las coordenadas, cartesianas, del mundo. En inglés se denomina *AABB.*

Caja orientada: Caja cuyos ejes pueden tener cualquier orientación.

Cámara: Es un sistema compuesto por un punto de vista (ubicación de la cámara), un conjunto de ejes coordenados y un *volumen de vista.* En proyecciones en perspectivas, el *volumen de vista* es una

pirámide truncada, de base rectangular, llamada *frustum*, delimitada por seis planos llamados: el *plano near*, el plano *far*, el plano *left*, el *right*, el *bottom* y el *top*. El *plano near* se utiliza como plano de vista, donde se proyecta todo el contenido del volumen. Dentro de este plano, la zona rectangular que contiene todos los puntos proyectados define el *view port*.

Celda: En este contexto, *poliedro* convexo del espacio definida por los planos de corte creados en la división por *BSP*. Las superficies del *poliedro* no formadas por polígonos determinan los *portales*.

Canal alfa: Es el valor de opacidad asignado a un color. Un valor alfa de uno se corresponde con un color totalmente opaco. Un valor de vero se corresponde con un color totalmente transparente.. Este valor generalmente se asocia a una cuarta componente de color formando el conjunto (rojo, verde, azul, alfa).

Cinemática: Estudio del movimiento de un objeto sin la consideración de su masa ni fuerzas que actúan sobre el. La cinemática directa (forward) se refiere al proceso de especificación de transformaciones a aplicar a un objeto compuesto y la obtención del resultado de esas transformaciones, calculando cada una de las orientaciones y posiciones de sus subobjetos componentes. La cinemática inversa se refiere a la especificación de una posición y orientación final de uno de los subobjetos y la determinación de las transformaciones necesarias a aplicar al resto de los subobjetos para obtener esa posición y orientación deseadas.

Clipping: Ver *Recorte*.

CLOD: Ver *Nivel de detalle continuo*.

Coeficiente de fricción: Coeficiente adimensional que expresa la oposición que ofrecen las superficies de contacto entre dos cuerpos al deslizar uno respecto del otro. Se distinguen dos valores: el coeficiente de fricción estático que se mide cuando ambas superficies están en reposo y el coeficiente de fricción dinámico que se mide cuando ambas superficies están en movimiento relativo el uno respecto del otro.

Coeficiente de restitución: Medida del grado de conservación de la energía cinética en un choque entre partículas clásicas.

Coherencia: Medida del cambio entre las imágenes generadas durante una secuencia animada. Muchos algoritmos toman ventaja del hecho de que, usualmente, los cambios entre dos *cuadros* consecutivos, es pequeño.

Coherencia temporal: En un sistema de detección de colisiones se refiere a tomar como válida la premisa de suponer que el movimiento de los objetos es lo suficientemente lento como para que no hayan grandes cambios posicionales en cada paso de la simulación.

Cola de prioridad: Tipo abstracto de datos que soporta las operaciones de: agregar un elemento con una prioridad asociada, remover y devolver el elemento de mayor prioridad y obtener el elemento de mayor prioridad sin removerlo de la cola.

Complejidad de profundidad: Se refiere a cuantas veces es dibujado un píxel durante el dibujado de un solo *cuadro* de una escena. Este valor es conveniente que sea cercano a uno ya que al aumentar este numero, disminuyen los *cuadros por segundo* generados por la aplicación.

Conjunto convexo: Un conjunto es convexo si, dados dos puntos del conjunto, el segmento de recta que los conecta también se encuentra en el conjunto. La envolvente convexa de un conjunto de punto es el menor conjunto convexo que los contiene.

Coordenadas homogéneas: Son coordenadas de la forma (x,y,z,w) usadas por el modelo de *cámara* en proyecciones perspectivas. Cualquier punto en coordenadas homogéneas es equivalente al punto (x/w,y/w,z/w,1) y esta naturalmente relacionado con una proyección en perspectiva. Transformaciones homogéneas son operaciones que se aplican a coordenadas homogéneas para obtener otras coordenadas homogéneas.

Coordenadas: Es la abstracción algebraica para localizar puntos en el espacio como tuplas de números. Los números son medidas relativas a un conjunto de ejes llamados ejes de coordenadas.

Estos ejes pueden formar un sistema de mano derecha o un sistema de mano izquierda. La conversión de uno a otro involucra una *transformación* simple de dos componentes de una tupla dada. Coordenadas del modelo se refiere a las ternas usadas en la construcción de un objeto, generalmente hecha con una aplicación específica. Coordenadas del mundo se refiere a las ternas que representan al objeto en un sistema global de coordenadas. Coordenadas de vista se refiere a las coordenadas de un punto referido al sistema de coordenadas de la *cámara*, en donde el origen del sistema se ubica en la posición del observador. Las coordenadas de pantalla se refieren a las tuplas que representan al objeto luego de haber pasado por el ducto geométrico (*geometric pipeline*) del *renderer*.

CSG: Es una técnica que permite modelar objetos sólidos complejos utilizando operadores boléanos sobre objetos más simples, partiendo de un conjunto predefinido llamados primitivos.

Cuadro: Es una imagen completa en una secuencia animada, Cada cuadro representa un momento en el tiempo de la animación.

Cuadros por segundo: Es la cantidad de imágenes completas o cuadros por segundo generados por una aplicación grafica interactiva. Se mide en hertz(Hz). Su inversa es el tiempo en que demora el dibujado de un cuadro y se mide en milisegundos. Un acrónimo de este concepto es *FPS*.

Culling: Ver *Descarte*.

Descarte: Proceso por el cual se determina si un objeto no será visible permitiendo al sistema, no dibujarlo. Los métodos estándar comparan los volúmenes acotantes asociados a los objetos contra el *volumen de vista*. Otros métodos, como el de oclusión por portales permiten verificaciones de oclusión basadas en el punto de vista del observador y en planos de corte. *Backface culling* permite descartar el dibujo de triángulos de un objeto al determinar que no son visibles por el observador.

Descarte por oclusión: Clase de algoritmos encargados de descartar rápidamente geometría que no puede ser visible debido a la interposición de otros objetos. En escenas complejas, como por ejemplo ambientes arquitectónicos, una gran porción de la escena no será visible en cada momento y puede ser descartada usando estas técnicas. Esto permite acelerar el *FPS* de una aplicación.

Detección de colisiones: Proceso que determina si dos objetos estacionarios están intersecándose es llamado detección estática de colisiones. Si alguno de ellos, o los dos, se mueven, el termino detección dinámica se refiere al proceso de determinar cuando y donde los objetos comenzarán a intersecarse.

Determinante: Cantidad escalar derivada de una matriz cuadrada M. Si M es una matriz 2 x 2, M se refiere a la *transformación* de un cubo unidad en un paralelogramo o un segmento de línea. El determinante de M es el área, con signo, de ese paralelogramo y cero en el caso de que la *transformación* sea un segmento de línea. Esta idea de que un determinante representa un área se generaliza a dimensiones mayores.

Doble buffering: Es el uso de dos buffers, llamados *front* y *back* para representar la imagen a dibujar. El buffer *front* contiene la imagen que se muestra actualmente, mientras la aplicación dibuja la siguiente imagen en el buffer *back*. Cuando el nuevo *cuadro* esta listo para ser mostrado, los buffers se intercambian.

Ejes separadores: Línea tal que la proyección de dos objetos en ella genera intervalos disjuntos. En el caso en que dos objetos no estén intersecándose, siempre existirá una línea con estas características al existir un plano que puede separarlos.

Environment mapping: Ver *Mapeo del ambiente*.

Fase de detección amplia: En el contexto de un sistema de *detección de colisiones* se refiere al conjunto de algoritmos encargados de determinar rápidamente el conjunto de objetos que se encuentran cerca y, por lo tanto, en peligro de colisión. Esto alivia mucho el trabajo posterior del sistema de colisiones al permitirle focalizarse únicamente en esos objetos cercanos.

Fase de detección fina: En el contexto de un sistema de *detección de colisiones*, se refiere al conjunto de algoritmos que determina si un par de objetos están o no en situación de colisión. Generalmente

se aplican estos algoritmos con el subconjunto de objetos de un sistema determinado por la *fase de detección amplia*.

Frame: Ver *Cuadro*.

FPS: Ver *Cuadros por segundo*.

Frame buffer: Bloque de memoria que almacena el color actual de cada píxel de una imagen. Comúnmente es un área de memoria de video. En este bloque se encuentran, generalmente, los buffers *front* y *back*.

Gouraud: Ver *Sombreado de Gouraud*.

GJK: Algoritmo que determina la minima distancia entre dos conjuntos convexos y que debe su nombre a los creadores Gilbert, Johnson and Keerthi. A diferencia de otros algoritmos, no requiere que los datos geométricos estén almacenados en un formato especial sino que requiere solamente una función de soporte que permita generar símplices mas cercanos a la respuesta correcta final usando la suma de Minkowsky de los dos conjuntos convexos.

Grafo celda-portal: Grafo dirigido en el que cada nodo representa una celda y cada arco un portal. Geométricamente representa un conjunto de volúmenes convexos (celdas) comunicados por polígonos convexos (portales) que forman parte de alguna de sus caras.

Grafo de escena: Estructura de datos que representa todos los datos de una escena. Las relaciones geométricas son representadas como un árbol mientras que los nodos hoja contienen datos que pueden ser dibujados. El compartir datos lleva a uniones entre datos que determina una estructura de grafo. El proceso de mantener el grafo de escena incluye varios sistemas como ser descarte jerárquico, transformaciones, acumulación y actualización de estados del *renderer*, selección de niveles de detalle, etc.

Heap: Ver *Cola de prioridad*.

Heurística: Técnica diseñada a resolver un problema que ignora si la solución encontrada puede ser probada en correctitud pero que usualmente produce una buena aproximación o soluciona un problema mas simple relacionado con la solución del problema original.

Iluminación: Proceso de computar el color final de cada vértice de una escena. Los cálculos dependen de las fuentes de *luz* y su contribución es clasificada en: ambiente, difusa, especular o de emisión. La *luz* puede atenuarse para que su contribución final dependa de la distancia desde la fuente hasta el objeto. Por iluminación estática se entiende el proceso de iluminar objetos estáticos con luces fijas. Estos cálculos pueden hacerse off-line almacenando la información en mapas de *luz*. Por iluminación dinámica se entiende hacer los cálculos de *luz* en tiempo real, permitiendo mover objetos y luces. En este caso, las normales en los vértices deben ser almacenadas junto a los datos geométricos del objeto.

Key frame: Ver *Animación*.

Lightmap: Ver *Mapa de luz*.

LOD: Ver *Nivel de detalle*.

Luz: Representación, en un sistema gráfico, de una luz real. Por razones de eficiencia, generalmente se modelan luces de ambiente, direccionales, de punto y spots. La luz ambiente afecta a los objetos en la misma forma en el sentido que todos sus vértices reciben la misma contribución de luz. La luz direccional se asume localizada en el infinito pero con sus rayos teniendo la misma dirección. Su contribución a los vértices depende del vector normal definido en el vértice. Una luz de punto tiene una ubicación determinada y emite en todas direcciones. Su contribución a los vértices depende de la normal en ellos. Una luz spot tiene una posición determinada pero su efecto se reduce a un cono. La contribución a los vértices depende de la normal en ellos.

Luz proyectada: Es un ejemplo de *multitexturizado* dinámico en donde se asocia una textura a una determinada *luz*. Aquellos triángulos iluminados (afectados por la textura) tendrán que calcular sus

coordenadas de textura en tiempo real debido a que la posición y orientación de la *luz* pueden cambiar.

Malla de triángulos: Colección de triángulos que comparten un conjunto de vértices pero cuya conectividad debe ser especificada. Una malla de triángulos *manifold* es aquella en la que una arista puede ser compartida por dos triángulos como máximo.

Mapa de luz: En el contexto de motores 3D, un *lightmap* es una estructura de datos que contiene información del brillo de las superficies. Son estructuras que se precalculan y sirven para ser usadas en objetos estáticos. Para su creación puede utilizarse cualquier *modelo de luz*, aunque generalmente se utiliza radiosidad.

Mapa de normales: Es una imagen análoga al mapa de texturas excepto que los píxeles representan normales a la superficie en vez de colores. Las componentes roja, verde y azul se usan para codificar las coordenadas x, y, z de la normal. Es posible luego sombrear el triangulo usando esa información de normales en vez de interpolaciones hechas con las normales en los vértices.

Mapeo del ambiente: Efecto especial que permite dibujar superficies junto con la reflexión del ambiente en donde están inmersas. Es una técnica utilizada para modelar efectos de espejado en los objetos.

Material: Atributo usado conjuntamente con información lumínica para obtener una apariencia más realista de un objeto.

Matriz: Es una tabla de R x C valores con representando una cantidad R de filas y C de columnas. En general, en computación gráfica, se utilizan matrices de 3 x 3 o 4 x 4. Varias operaciones pueden aplicarse a estas matrices incluyendo su transpuesta, inversa, adjunto y el cálculo de su *determinante*.

Mezclado alfa: Es el proceso de mezclar los colores ya existentes en el buffer de dibujo con un nuevo conjunto de colores. Usado para hacer efectos de transparencia.

Modelo de luz: *Ver Luz.*

Multi textura: Ver *Multitexturizado*

Multi texturizado: Muchos efectos pueden implementarse aplicando dos o más texturas a un objeto. *Multitexturizado* estático se refiere a que dos o más texturas y sus coordenadas, son conocidas antes de la ejecución de la aplicación. La interpolación de estas texturas es hecha por el *rasterizador* y los colores combinados con funciones de composición. *Multitexturizado* dinámico se refiere a que una o más texturas y sus coordenadas son calculadas en tiempo real. Por ejemplo, la técnica de sombras proyectadas es un ejemplo de *multitexturizado* dinámico.

Narrow phase: Ver *Fase de detección fina.*

Near: Ver *Plano near.*

Nivel de detalle: Es el concepto de enviar al *rasterizador*, la representación de un objeto con una complejidad dependiente de varias parámetros como son su geometría real y el modelo de *cámara*. La idea esencial es dibujar una representación de alta resolución cuando el objeto se encuentra cerca de la *cámara* y una representación de baja resolución cuando el objeto se encuentra lejos del observador. Un acrónimo de este concepto es *LOD* (por Levels of Detail). Dos técnicas básicas son el *nivel de detalle continuo* (*CLOD*) y el discreto.

Nivel de detalle continuo: Un sistema de *nivel de detalle* en el que la superficie varia continuamente permitiendo al objeto ser representado con muchos niveles de detalle.

Nivel de detalle discreto: Es un sistema de *nivel de detalle* donde una cantidad de versiones de un objeto son previamente almacenadas, cada una con *nivel de detalle* menor. El sistema luego selecciona el *nivel de detalle* más apropiado de acuerdo al espacio que el objeto ocupará en pantalla.

Normal map: Ver *Mapa de normales.*

OBB: Ver *Caja Orientada.*

Observador: Ubicación de las coordenadas de origen de una *cámara* virtual.

Oclusor: En este contexto, toda geometría poligonal que se interpone a la línea de vista impidiendo la *visibilidad* de otros objetos poligonales. Su proyección sobre el plano *near* del *volumen de vista* determina una zona de oclusión bidimensional.

Occlusion culling: Ver *Descarte por oclusión*.

Octree: Un esquema de división espacian en donde la caja acotante de un objeto es recursivamente subdividida en ocho octantes de igual tamaño los cuales, a su vez, pueden ser nuevamente. Puede almacenarse en una estructura de árbol en el que cada nodo puede tener hasta ocho hijos. Es un caso especial de un *árbol BSP*.

Optimización: Proceso de ajuste de variables para alcanzar un estado que minimiza o maximiza una determinada función objetivo. En computación grafica generalmente se refiere a cambios en la aplicación orientados a maximizar los *FPS*.

Parser: Proceso de análisis de una secuencia de *tokens* para determinar su estructura gramatical con respecto a una gramática formal. El *parser* es la parte de un compilador que realiza este trabajo. Para ello transforma una entrada de texto en una estructura de datos, usualmente un árbol, el cual es mas idóneo para hacer un procesamiento posterior de los datos y que captura la jerarquía implícita del texto de entrada.

Phong: Ver *Sombreado*

Píxel shader: Se refiere al proceso aplicado a un píxel antes de ser dibujado en el *frame buffer*. Un píxel *shader* generalmente involucra uno o más accesos a texturas, interpolación de propiedades en los vértices y posiblemente una lectura y modificación de valores ya existentes en el *frame buffer*. Se trata de componentes dependientes del hardware y se especifican como cambios de estado o por seudo código.

Plano near: Plano en que se proyectan los objetos tridimensionales visibles en el proceso de creación de una imagen. Al resultado de la proyección se le aplican transformaciones de ajuste de escala, rotación y traslación antes de dibujarla en pantalla. Este plano también determina una de las cotas del *volumen de vista*.

Poliedro: En este contexto, objeto tridimensional formado por caras planas.

Portal: La determinación de lo que se puede ver en un ambiente interior puede realizarse usando portales. El ejemplo típico aparece cuando un *observador* esta fuera de una habitación y está mirando hacia su interior a través de la apertura de una puerta (representada por un portal). El *observador* puede ver el interior de la habitación únicamente a través de la abertura del portal. Las paredes ocluyen todo lo demás. Un sistema de oclusión con portales construirá, generalmente, planos de corte formados desde el *observador* hacia cada borde del portal, asumiendo que se trata de un polígono convexo. Los portales surgen naturalmente durante la aplicación de algoritmos de división espacial, como por ejemplo en la construcción de *árboles BSP*.

PST: Ver *Árbol de búsqueda por prioridad*.

PVS: Es una técnica de oclusión que permite establecer un conjunto conservativo de polígonos visibles para cualquier punto de vista de una escena. Se apoya fuertemente en algún tipo de división espacial, por ejemplo, *BSP*.

Quadtree: Estructura de datos que representa a un nodo en un árbol que tiene cuatro hijos. Es una estructura generalmente utilizada para particionar un plano. Es un caso especial de un *árbol BSP*.

Quaternión: Conjunto de cuatro escalares que forman un sistema algebraico en el cual la multiplicación es una operación no conservativa. Los quaterniones de modulo uno son usados como una forma compacta de representar rotaciones. Esta representación es también conveniente para el uso en sistemas de animación de rotaciones.

Rasterizador: En este contexto, se refiere al componente de un sistema grafico que tiene la responsabilidad de calcular y dibujar los píxeles que se corresponden con cada triangulo procesado por el *renderer*.

Recorte: Proceso utilizado al realizar la proyección de los objetos en el plano de vista en el cual los objetos que no están en el *volumen de vista* no son proyectados. Los que están completamente contenidos si lo son y los que están parcialmente contenidos son proyectados parcialmente (solo la parte que es interna al *volumen de vista*).

Renderer: En este contexto, el componente de un sistema grafico que tiene la responsabilidad de dibujar los triángulos de un modelo usando el estado actual del sistema, llamado estado del *renderer*. Un *renderer* por software es una implementación que usa únicamente la CPU para realizar su trabajo. Un *renderer* por hardware es una implementación que se apoya en un procesador grafico para realizar su trabajo.

Respuesta a colisiones: Dados dos objetos que van a intersecarse, este concepto se refiere a como van a reaccionar luego de que la intersección ocurra. Este comportamiento es determinado por la aplicación y finalmente definido por el componente denominado motor de física.

Shader: Ver *Sombreado*.

Sistema de detección en dos fases: En este contexto, sistema de colisiones que utiliza *heurísticas* para descartar rápidamente la detección de objetos lejanos (*Broad Phase*). Luego de la fase de descarte se determina la situación de colisión con un numero generalmente muy reducido de objetos del sistema (*Narrow Phase*).

Sombra proyectada: Es un ejemplo de *multitexturizado* dinámico donde se asocia una textura a una fuente de *luz*, que representa la sombra generada por la *luz*. Aquellos triángulos sombreados por la *luz* tendrán que calcular sus coordenadas de textura en tiempo real debido a que la posición o orientación de la *luz* pueden cambiar.

Sombreado: Proceso de calcular el color de los píxeles. El sombreado plano usa el mismo color para cada píxel de un triangulo dibujado. El *sombreado de Gouraud* aplica las ecuaciones de *luz* sobre los vértices de un triangulo y luego interpola sus valores en el interior. El sombreado de Pong interpola las normales en los vértices de los triángulos y aplica luego las ecuaciones de *luz* en cada píxel. Es un método costoso que se hace, en general, por algún procesador gráfico.

Sombreado de Gouraud: Sombreado que aplica las ecuaciones de *luz* en los vértices de un triangulo y luego interpola estos valores para obtener el *sombreado* del triángulo completo.

Sweep & Prue: Técnica para determinar la situación de solapamiento entre intervalos que utiliza el concepto de coherencia temporal para hacerlo en un tiempo promedio de orden constante en la cantidad de intervalos. Esta técnica se extiende a tres dimensiones para obtener la relación de solapamiento de cajas AABB también en un tiempo constante en promedio.

Tensor de inercia: El tensor de inercia es un tensor simétrico de segundo orden que caracteriza la inercia rotacional de un sólido rígido. Expresado en una base ortonormal, dicho tensor se forma a partir de los momentos de inercia según tres ejes perpendiculares y tres productos de inercia. Ver [GOLD02].

Test alfa: Una etapa del pipeline de OpenGL que descarta fragmentos dependiendo del resultado de la comparación del valor alfa actual del fragmento y una constante de referencia.

Texel: Ver *Textura*.

Textura: Imagen aplicada a los modelos para otorgarles realismo. Las texturas pueden ser generadas por un artista o ser obtenidas de fotos digitales. Cada píxel de una textura junto con su valor de color es llamado *texel*. Para que el *rasterizador* pueda dibujar un modelo con texturas, los vértices de sus triángulos necesitan contener coordenadas de textura que representan a un determinado píxel en ella. Durante la interpolación, la ubicación de un píxel en pantalla puede corresponderse a más de una posición en de un *texel*, o a menos de un *texel*, resultando en imágenes con *aliasing*. Para minimizar estos efectos se utilizan diversos filtros como por ejemplo el filtro bilineal y trilineal.

Tiempo de impacto: En este contexto es el tiempo estimado de colisión de dos objetos. La exactitud de la estimación depende, según la formulación de Brian Mirtich [MIRT96], del módulo de la velocidad angular relativa entre ellos.

TnL: Generación de procesadores gráficos capaces de manjar completamente el ducto geométrico (*geometric pipeline*). Se incluyen las etapas de *transformación*, *iluminación*, *recorte*, oclusión y rasterización. La aplicación únicamente provee los datos geométricos, la *cámara* y las transformaciones modelo-mundo.

TOI: Ver *Tiempo de impacto.*

Token: palabra o elemento atómico dentro de un string.

Topología de una malla: En computación grafica, se refiere a las características de una malla referida a sus bordes, a la relación de conexión entre sus puntos y a cuantos hoyos tiene.

Transform and lithing: Ver *TnL.*

Transformación: Operación que convierte puntos en un sistema de coordenadas en puntos en otro sistema de coordenadas. En un sistema jerárquico, las transformaciones locales son usadas para representar el posicionamiento de los objetos referido al sistema de coordenadas padre. Las transformaciones del mundo son usadas para representar la posición de los objetos en un sistema de coordenadas global.

Vector: Tabla de R x 1 valores que representan una cantidad R de filas y una sola columna. En computación gráfica se utilizan vectores de 3 x 1 o 4 x 1. Varias operaciones pueden ser aplicadas a los vectores incluyendo el producto escalar, el vectorial y la normalización.

Velocidad de refresco: Velocidad en que se refresca el contenido de la pantalla, medida en hertz. En gran parte de sistemas gráficos, es un proceso realizado por el hardware.

Vertex shader: Se refiere al tipo de proceso aplicado a un vértice para transformarlo, recortarlo, iluminarlo o prepararlo para ser procesado por un *píxel shader*. Dependiendo de la plataforma, los *vertex shaders* pueden ser representados explícitamente en Assembler o como un conjunto de cambios de estado en el procesador grafico.

Vértice: En este contexto es un vector que esta asociado a una *malla de triángulos* que representan a un objeto. Un vértice puede tener varios atributos, llamados atributos del vértice, que incluyen: coordenadas de textura, color y un vector normal. Estos atributos son usados para calcular el color final del vértice que es usado por el *rasterizador* durante la interpolación al dibujar el triangulo. Los valores interpolados son llamados atributos de superficie.

Vertical retrace: Momento en que se produce el reposicionamiento del haz de electrones que dibuja la imagen en un tubo de rayos catódicos, en el proceso que típicamente sucede en los monitores con esa tecnología.

Viewport: Ver *Cámara.*

Visibilidad: Se refiere a qué objetos son visibles desde un punto de vista. Saber que objetos son visibles es útil debido a que solo ellos deben ser procesados por el *renderer*.

Volumen acotante: Es un objeto regular que engloba una región del espacio con el propósito de acelerar la *detección de colisiones* y el proceso de *culling*. Objetos típicos son cajas, capsulas, cilindros, elipsoides, lozengues y esferas.

Volumen de vista: Es la porción del espacio que es considerado visible desde el punto de vista del *observador*. Es una pirámide truncada formada por seis planos llamados *near, far, left, right, top* y *bottom*.

Volumen shaft: Es una simplificación del volumen barrido por un objeto durante su trayectoria en un lapso de tiempo. Se construye considerando las cajas acotantes *AABB* del objeto en su posición inicial y final y armando el volumen resultante barrido por las cajas. Si estas son cúbicas el volumen

puede formarse con, a lo sumo, seis planos. En el caso general se necesitarán como máximo ocho planos [HAIN94; HAIN00].

Z-buffer: Ver *Buffer de profundidad*

Referencias

3DEN03 3D Game engines list. http://www.devmaster.net/engines.2003.

AGE02 Ageia PhysX. http://www.ageia.com/physx/index.html.2002.

AIRE90 Airey, J. M., Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations. PhD thesis, Dep. of CS, U. of North Carolina, July 1990.

AKEN01 Akenine Moller, T., Fast 3d Triangle Box Overlap Testing. Journal of graphics tools, vol. 6, no. 1. pp.29-33. 2001.

ANIT96 Anitescu, M. and F. Potra, Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementary problems. Reports on Computational Mathematics, No.93/1996. Department of Mathematics, The University of Iowa. 1996

APOD99 Apodaca, A. y L. G., Advanced RenderMan: Creating CGI for Motion Pictures, Morgan Kaufman Publishers, Inc., San Francisco, 1999.

ARCO03 Arconovum Software. http://arconovum.com. 2003.

ASHI06 Ashida, K., Optimising the Graphics Pipeline. NVIDIA Corporation, 2006

BARA89 Baraff, D., Analytical methods for dynamic simulation of non penetrating rigid bodies. *Computer Graphics,* 23(3):223-232, July, 1989.

BARA91 Baraff, D., Copping with friction for non penetrating rigid body simulation. *Computer Graphics*, 25(4):31-40. August, 1991.

BARA92 Baraff, D., Dynamic Simulation of Non Penetrating Rigid Bodies. PhD thesis. Department of Computer Science Cornell University. March 1992.

BARA93 Baraff, D., Issues in computing contact forces for non-penetrating rigid bodies. *Algorithmica*, 10:292-352. 1993

BARA94 Baraff, D., Fast contact force computation for non penetrating rigid bodies. In *SIGGRAPH Conference Proceedings*. ACM Press, 1994.

BARA96 Baraff, D., Linear time dynamics using Lagrange multipliers. In *SIGGRAPH Conference Proceedings*. ACM Press, 1996.

BARZ96 Barzel, B., and A. Barr, A Modelling System based on dynamic constraints. In Computer Graphics, volume 22, pp. 179-187. 1996

BEND05 Bender, J., D. Finkenzeller, and A. Schmitt, An impulse-based dynamic simulation system for VR applications. Proceedings of Virtual Concept, Biarritz, France, November 2005.

BEND06 Bender, J., and A. Schmitt, Fast Dynamic Simulation of Multi-Body Systems Using Impulses. Institut fur Betriebs. Und Dialogsysteme, Universitat Karlsruhe, Germany. 2006

BERG00a de Berg, M., M. van Kreveld, M. Overmars, O. Schwarzkopf, Interval trees. Ch 10.1. Computational Geometry: Algorithms and Applications. Springer Verlag. ISBN: 3-540-65620-0. 2000.

BERG00b de Berg, M., M. van Kreveld, M. Overmars, O. Schwarzkopf, General Set of Points. Ch 5.5. Computational Geometry: Algorithms and Applications. Springer Verlag. ISBN: 3-540-65620-0. 2000.

BERG01 van den Bergen, G., Proximity Queries and Penetration Depth Computation on 3D Game Objects. Game Developers Conference, pp. 821-837, March 2001.

BERG04 van den Bergen, G., Collision Detection in Interactive 3D Environments. Morgan Kaufmann publishers. Elsevier Inc. ISBN: 1-55860-801-X. 2004.

BERG97 van den Bergen, G., Efficient Collision Detection of Complex Deformable Models Using AABB Trees. Journal of graphics tools, vol. 2, no. 4, pp. 1-13, 1997.

BERG99a van den Bergen, G., A Fast and Robust GJK Implementation for Collision Detection of Convex Objects. Journal of graphics tools, vol. 4, no. 2, pp. 7-25, 1999.

BERG99b van den Bergen, G., Collision Detection in Interactive 3D Computer Animation. PhD thesis, Eindhoven University of Technology, 1999.

BLOW04 Blow, J., Game Development: Harder Than You Think. ACM Queue vol. 1, no. 10 - February 2004

CAME97 Cameron, S., Enhancing GJK: Computing Minimum and Penetration Distance Between Convex Polyhedra. International Conference on Robotics and Automation, pp. 3112-3117, 1997.

CATM74 E. Catmull A Subdivision Algorithm for Computer Display of Curved Surfaces. University of Utah, 1974.

CATT05 Catto, E., Iterative Dynamics with Temporal Coherence. Crystal Dynamics. Menlo Park, California. June 2005.

COTT92 Cottle, R. J.S. Pang, and R. E. Stone, The Linear Complementarity Problem. Academic Press, 1992.

COWM03 Cowmans, E., Bullet Physics Library. http://www.continuousphysics.com/. 2003

CREI85 McCreight, E., Priority Search Trees. SIAM Journal on Computing, vol. 14, no.2. May 1985

CROM01 Cormen, T. C. E. Leiserson, R. L. Rivest and C.Stein, Red-Black Trees. Ch. 13, pp.273-301.Introducion to algorithms, Second Edition. MIT Press and Mc Graw-Hill, ISBN 0-262-03293-7. 2001

DFCI03 DFC: Game Industry Research. http://www.dfcint.com/.2003.

DRDO01 Dr. Dobbs Portal, The Fibonacci Heap. http://www.ddj.com/184410119 2001

EBER02 Eberly, D., Polyhedral Mass Properties (Revisited). Magic Software inc. http::://www.magic-software.com. December 2002.

EBER03 Eberly, D., Minimum Area Rectangle Containing a Convex Polygon. Geometric Tools, Inc. December 8, 2003. http://www.geometrictools.com

EBER02e Eberly, D., Preface, pp XXVII. 3D Game Engine design. Morgan Kaufman Publishers. 2002.

EBER04b Eberly, D., Game Physics. Morgan Kaufmann Publishers. Elsevier. ISBN: 1-55860-740-4. 2004.

EBER02d Eberly, D., A brief Motivation, Ch 1.1. pp 3. 3D Game Engine design. Morgan Kaufman Publishers. 2002

EBER02c Eberly, D., Hierarchical Scene Representations , Ch 4. pp 141. 3D Game Engine design. Morgan Kaufman Publishers. 2002.

EBER02b Eberly, D., Character Animation , Ch 9. pp 341. 3D Game Engine design. Morgan Kaufman Publishers. 2002.

ECCL00 Eccles, Allen, The Diamond Monster 3Dfx Voodoo 1, Gamespy Hall of Fame, 2000. http://gamespy.com/halloffame/october00/voodoo1/

ERLE05a Erleben, K., Stable, Robust, and Versatile Multibody Dynamics Animation. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), Universitetsparken 1, DK-2100 Copenhagen, Denmark. 2005.

ERLE05b Erleben, K., J. Sporring, K. Henriksen, and H. Dohlmann, Physics-Based Animation. Published by Charles River Media INC, Hingham, Massachusetts, 2005.

FOLE90 Foley, J., A. van Dam, S. Feiner, J. Hughes, The quest for visual realism. Ch. 14, pp 605. Computer Graphics, Principles and Practice, Second Edition. Addison Wesley, ISBN 0-201-12110-7.1990.

FOLE90b Foley, J., A. van Dam, S. Feiner, J. Hughes, Computer Graphics, Principles and Practice, Second Edition. Addison Wesley, ISBN 0-201-12110-7.1990.

GBRY03 GameBryo Game Engine. http://www.emergent.net. 2003

GILBE88 Gilbert, E., D. Johnson, and S. Keerthi, A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional-Space. IEEE Journal of Robotics and Automation, vol. 4, no. 2, pp. 193-203, April 1988.

GOLD02 Goldstein, H., C.P. Poole, J. L. Safko, Classical Mechanics. Addison Wesley; 3 edition (January 15, 2002). ISBN-10: 0201657023

GOTT96 Gottschalk, S., M. C. Lin, and D. Manocha, OBB Tree: A Hierarchical Structure for Rapid Interference Detection. Computer Graphics, SIGGRAPH 96 Proceedings, pp. 171-180, August 1996.

GPLI03 General Public License, http://en.wikipedia.org/wiki/GPL. 2003

GREG00 Gregory, A., A. Mascarenhas, S. Ehmann, M. Lin and D. Manocha, Six degree of freedom haptic display of polygonal models. In Proceedings IEEE Visualization, 2000.

GUEN03 Guendelman, E., R. Bridson, and R. Fedkiw, Nonconvex Rigid Bodies with Stacking. ACM Transactions on Graphics, Proceedings of ACM SIGGRAPH, 2003.

GTKL03 GTK+ library, http://en.wikipedia.org/wiki/Gtk. 2003.

GTKR00 GtkRadiant editor. http://en.wikipedia.org/wiki/GtkRadiant. 2000.

HAIN00 Haines, E., A shaft Culling Tool. Journal of graphics tools, vol.5, no. 1, pp. 23-26, 2000.

HAIN94 Haines, E., and J. Wallace. Shaft Culling for Efficient Ray-Traced Radiosity, In P. Brunet and F.W. Jansen, eds., Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering), Springer-Verlag, pp. 122-138, 1994.

HAL204a Half Life 2 Review. http://www.guru3d.com/article/gamereviews/163/3/y. 2004.

HAL204b Half Life 2 Review. http://www.anandtech.com/printarticle.aspx?i=2281. 2004.

HAM03 Hammer Editor, Valve Software. http://www.valvesoftware.com. 2003

HAZE03 Hazegawa, S., N. Fujii, Y. Koike, and M. Sato, Real Time rigid body simulation based on volumetric penalty method. In Proceedings of the 11th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (HAPTICS'03), page 326. IEEE Computer Society. 2003.

HENR05 Henry, D. MD5Mesh and MD5Anim files formats.http://tfc.duke.free.fr, august 2005.

IDSO03 IdSoftware. http://www.idsoftware.com. 2003.

IRRL03 Irrlicht Engine. http://irrlicht.sourceforge.net/. 2003.

KARO96 Karon, D., Lectures on Priority Search Trees. Brown University. http://www.cs.brown.edu/courses/cs252/misc/proj/src/Spr96-97/mjr/doc/09.ps. 1996.

KILG00 Kilgard, M., A practical and robust bump-mapping technique for today's gpus.GDC 2000: Advanced OpenGL Game Development, July 2000.

KLOS98a Klosowski, J. T., M. Held, J.S.B Mitchell, H. Sowizral, and K. Zikan, Efficient Collision Detection Using Bounding Volume Hierarchies of K-DOPs. IEEE Transactions on Visualization and Computer Graphics, vol. 4, no. 1, 1998.

KLOS98b Klosowski, James T., Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments, Ph. D. Thesis, State University of New York at Stony Brook, May 1998.

KOCH98 Kochy, K., M.K., Interactive Manipulation of Real-time Visualization from Medical Volume Data by using 2-Handed VR-Techniques. UKBF, FU Berlin, Germany. EuroPACS 1998.

LEFE03 Lefebre, S., S. Hornos, Automatic Cell-and-portal Decomposition. INRIA Report de Recherche N°4898, Juliet 2003

LIDLA86 Lidlaw, David H., W. Benjamin Trumbore, J. F. Hughes, Constructive Solid Geometry for Polyhedral Objects, Brown University, 1986.

LUEB95 Luebke, David P. and C. Georges, Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets, Proceedings 1995 Symposium on Interactive 3D Graphics, pp. 105-106, April 1995.

MERC03 Mercury computer Systems. http://www.tgs.com. 2003.

MIRT95 Mirtich, B., Hybrid Simulation: Combining constraints and impulses. Proceedings of First Workshop on Simulation and Interaction in Virtual Environments, 1995.

MIRT96 Mirtich, B., Impulse-Based Dynamic Simulation of Rigid Body Systems. PhD thesis, Universidad de California, Berkeley, 1996.

MIRT98a Mirtich, B., Rigid Body Contact: Collision detection to force computation. Technical Report TR-98-01,MERL, 1998.

MIRT98b Mirtich, B., V-Clip: Fast and robust polyhedral collision detection. ACM Trasactions on Graphics, 17(3):117-208, 1998.

MOHR03 Mohr, A., L. T. y M. G. Direct Manipulation of Interactive Character Skins. University of Wisconsin, Madison, 2003.

MOLL02 Moller, T., E. Haines, Separating Axes Theorem. Ch. 13.2, pp. 563-564. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02b Moller, T., E. Haines, A Simple Display System. Ch. 15.1.1, pp. 669-671. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02c Moller, T., E. Haines, Spatial Data Structures. Ch. 9.1.2, pp. 349-353. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02d Moller, T., E. Haines, Bounding Volume Hierarchies. Ch. 9.1.1, pp. 347-349. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02e Moller, T., E. Haines, Occlusion Horizons. Ch. 9.7.1, pp. 373-377. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02f Moller, T., E. Haines, Level of Detail. Ch. 9.8, pp. 389-401. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02g Moller, T., E. Haines, Radiometry and Photometry. Colorimetry. Ch. 6.1-6.2, pp. 182-194. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02h Moller, T., E. Haines, Introduction. Ch. 1, pp. 1. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02i Moller, T., E. Haines, Performance Measurements. Ch. 10.2, pp. 409. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02j Moller, T., E. Haines, Spatial Data Structures Ch. 9.1.1, pp. 347-349. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02k Moller, T., E. Haines, Level of Detail Ch. 9.8, pp. 389-401. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOLL02l Moller, T., E. Haines, Visual Appearance Ch. 4, pp. 67. Real Time Rendering. A. K. Peters. ISBN: 1568811829. Nov 2002.

MOOR88 Moore, M., J. Wilhelms, Collision Detection and Response for Computer Animation. Computer Graphics, Volume 22, Number 4. August 1988

MURT97 Murty, K.G., F.T. Yu., Linear Complementarity. Linear and Nonlinear Programming. Dept. of Industrial and Operations Engineering. The University of Michigan. 1997

MURT03 Generic Polygon Clipper. http://www.cs.man.ac.uk/~toby/alan/software/ 2003.

NVID02 Mark, B, NVIDIA Corporation, Preface, pp. 1, The Cg Tutorial. 2002.

NVID04 NVIDIA Corporation, Introduction to the Cg Languaje, pp. 1. Cg toolkit. Rev 1.2 Jun. 2004.

NVID04b NVIDIA Corporation, The Cg profiles, pp. 3. Cg toolkit. Rev 1.2 Jun. 2004.

OGRE03 Ogre Engine. http://ogre3d.org/. 2003.

QERA03 QERadiant. http://www.qeradiant.com. 2003.

Q3WO07 Quake 3 World. http://www.quake3world.com.2007.

REDO02 Redon, S., A. Kedar, and S. Coquillart, Fast continuous collision detection between rigid bodies. Computer Graphics Forum (Eurographics 2002 Proceedings), 21(3).

REDO04a Redon, S., Continuous collision detection for rigid and articulated bodies. ACM SIGGRAPH Course Notes, 2004.

REDO04b Redon, S., Y. J. Kim, M. C. Lin and D. Manocha, Fast continuous collision detection for articulated models. Proceedings of ACM Symposium on Solid Modeling and Applications, 2004.

SCHM04 Schmidl, H., V. Milenkovic., A Fast Impulsive Contact Suite for Rigid Body Simulation. IEEE Transactions on Visualization and Computer Graphics, Vol. 10, No. 2, pp. 189-197, March 2004.

SCHM07 Schmidth, M., Advanced Topics In Computer Graphics: Pipeline Optimization. CS 563. 2007.

SNOO03 Snook, G., Real-Time 3D Terrain Engines Using C++ and DirectX 9, 1st edition (July 2003). ISBN 1-58450-204-5

STEW97 Stewart, D., J.C. Trinkle, An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction. Dept. of Mathematics, University of Iowa and Sandia National Labs. Submitted to *International Journal for Numerical Methods in Engineering*. 1997.

TASD03 Tasdizen, T., R. W., P. B. y S. O. Geometric Surface Processing via Normal Maps. ACM Transactions on Graphics, 2003

TELL91 Teller, Seth J., y Carlo H. Séquin, Visibility Preprocessing For Interactive Walkthroughs, Computer Graphics (SIGGRAPH '91 Proceedings), pp. 61-69, July 1991.

TELL92 Teller, Seth J., Visibility Computations in Densely Occluded Polyhedral Environments, Ph.D. Thesis, Department of Computer Science, University of Berkeley, 1992.

UNED03 Unreal Editor. http://en.wikipedia.org/wiki/Unrealed. 2003.

UNRE03 Unreal Technology. http://www.unrealtechnology.com/. 2003.

VALV03 Valve Software. http://www.valvesoftware.com. 2003.

WON00 Wonka, P., M. W. y D. S., Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. 11th Euro-graphics Workshop on Rendering, pp.71-82, 2000.

ZERB04 Zerbst, S. y O. D., 3D Game Engine Programming. Course Technology PTR, first edition, June 2004.