Tema VI
Técnicas de aceleración de ray tracing

Ricardo Ramos

Colaboradores: Asunción Fernández Urdangaray, José Javier Cortés Gutiérrez

El problema principal del algoritmo trazador de rayos simple es su lentitud, ya que para generar una imagen puede emplear una enorme cantidad de tiempo. Básicamente, esto se debe a que para cada rayo trazado se ha de probar si interseca con cada objeto presente en el escenario, tarea que puede acaparar hasta el 95% del tiempo de renderizado. No es de extrañar que las principales técnicas de aceleración traten de minimizar el tiempo empleado en la búsqueda de puntos de intersección, aunque este no es el único criterio de aceleración utilizado, como ahora veremos.

1.1 Clasificación de las técnicas de aceleración de ray tracing

Son tres las principalmente estrategias de aceleración:

1) Aumentar la velocidad de búsqueda de las intersecciones

Esta estrategia engloba a otras dos:

A) Procurar que el *cálculo de las intersecciones sea lo más rápido posible*. Este planteamiento admite a su vez dos posibilidades:

   a. *Optimizar los algoritmos de intersección*, disminuyendo en lo posible el total de operaciones aritméticas a realizar.

   b. Hacer *que las intersecciones se realicen con figuras simples*, con lo que en muchos casos se simplifica la naturaleza de los cálculos aritméticos. Dentro de este subcampo estudiaremos los **volúmenes envolventes** (**VVEE**).

B) Utilizar *algoritmos que minimicen el número de intersecciones rayo-objeto*. Entre las técnicas que utilizan esta estrategia podemos encontrar la *jerarquía de volúmenes envolventes*, la *subdivisión espacial*, el *buffer de elementos*, y las *técnicas direccionales*.

2) Reducir el número de rayos trazados

Como hemos visto al estudiar ray tracing, además de los rayos *primarios*, concurren en el escenario otros rayos generados mediante refracciones,

reflexiones y sombras. Entre las técnicas más eficientes para disminuir el número de rayos trazados está la del *control adaptativo de la profundidad del árbol de rayos*. Con ella, el árbol de rayos generado a partir de un rayo *primario* dado, no finalizará en una profundidad determinada (constante), sino cuando se determine que el trazado de nuevos rayos apenas aportarán color al píxel.

Otra estrategia a seguir dentro de este grupo consiste en la *optimización estadística* en los planteamientos *antialiasing*. Así, la detección de situaciones en las que un número relativamente pequeño de muestras produce resultados estadísticamente fiables de una región de la imagen, permite reducir el número de rayos primarios Este tipo de técnicas son, principalmente, optimizaciones en tiempo de ejecución.

3) Generalizar el concepto de rayo

El objetivo de esta estrategia es el de remplazar el concepto de rayo (como fotón), por entidades más generales, como pueden ser haces finos, haces de conos, etc.

Esta clasificación de las estrategias y técnicas de aceleración podemos resumirla en el siguiente esquema:

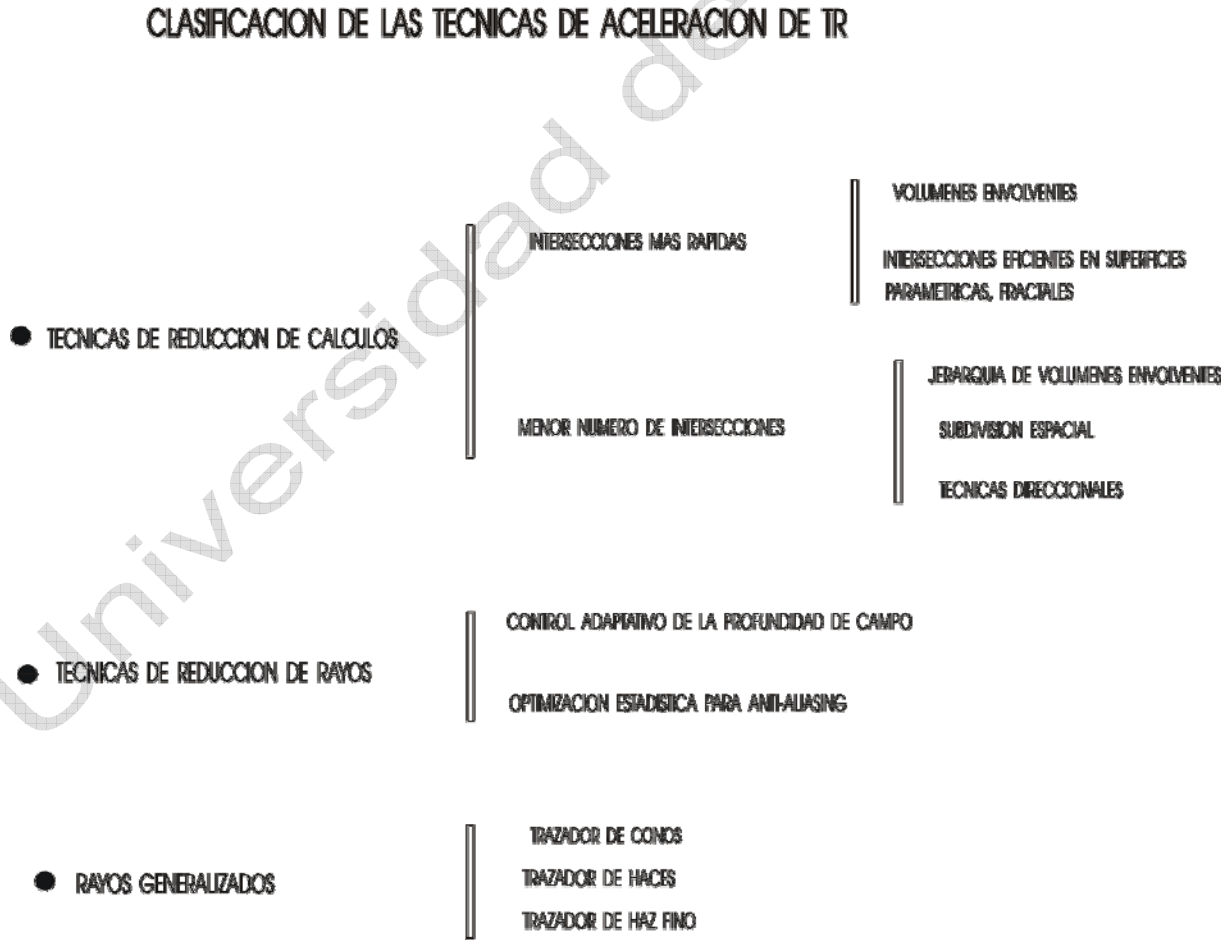

figura 1: clasificación de las técnicas de aceleración de ray tracing

1.2 Volúmenes envolventes

Una de las herramientas fundamentales para la aceleración de ray tracing es la de los volúmenes envolventes. Esta técnica consiste en encerrar los objetos a visualizar, dentro de volúmenes de geometría simple (esferas, elipsoides, prismas rectangulares, etc.).

Puesto que sólo cabe la posibilidad de que el rayo interseque con el objeto después de haber intersecado con el volumen que lo envuelve, en muchas ocasiones un cálculo sencillo de intersección (con el VE) es suficiente para determinar si el rayo interseca o no con el objeto.

1.2.1 Eficiencia de los volúmenes envolventes

Para que esta técnica sea eficiente se ha de elegir un volumen envolvente (**VE**) adecuado. La idoneidad de un VE depende de *cómo se ajusta al objeto que encierra*, y del *coste del cálculo de la intersección rayo-VE*. Este último parámetro, o sea, la simplicidad del test de intersección con el VE, es un factor importante a la hora de elegir el VE.

Sin embargo, Weghorst, Hooper y Greenberg [WEGH84] puntualizaron que la simplicidad del test de intersección no debe ser el único criterio en la selección del volumen adecuado. Definieron el *área nula* de un volumen, como la diferencia de área entre las proyecciones ortogonales del objeto y del volumen, en un plano perpendicular al plano que pasa por el origen del rayo.

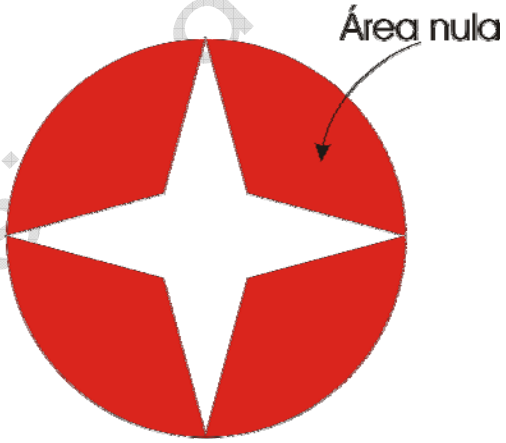

figura 2: área nula con distintos volúmenes envolventes

Demostraron que el *área nula* es una función del objeto, del volumen envolvente y de la dirección del rayo. Basándose en esto, llegaron a la conclusión de que *el mejor volumen envolvente es aquel que minimiza el coste de la búsqueda de los puntos de intersección con el objeto*. Este coste lo expresaron como

$$T = nB + mI$$

donde

n: número de veces que el VE es examinado como candidato para una intersección.

B: coste del cálculo de la intersección con el volumen.

m: número de veces que el objeto del interior es examinado buscando una intersección (número de veces que el volumen presenta una intersección)

I: coste del cálculo de la intersección con el objeto.

Como el objeto sólo se prueba cuando hay una intersección con el volumen envolvente, necesariamente $m \leq n$. Puesto que I y n pueden ser consideradas como constantes para un objeto particular, se minimizará el coste total (T) haciendo que B y m sean lo menor posible. Un volumen que se ajuste bien al objeto minimizará m y, en general, aumentará el valor de B, por lo que es preciso buscar un equilibrio coste/ajuste.

Por otro lado, la efectividad de un determinado volumen envolvente, dependerá también de la orientación del objeto dentro del volumen.

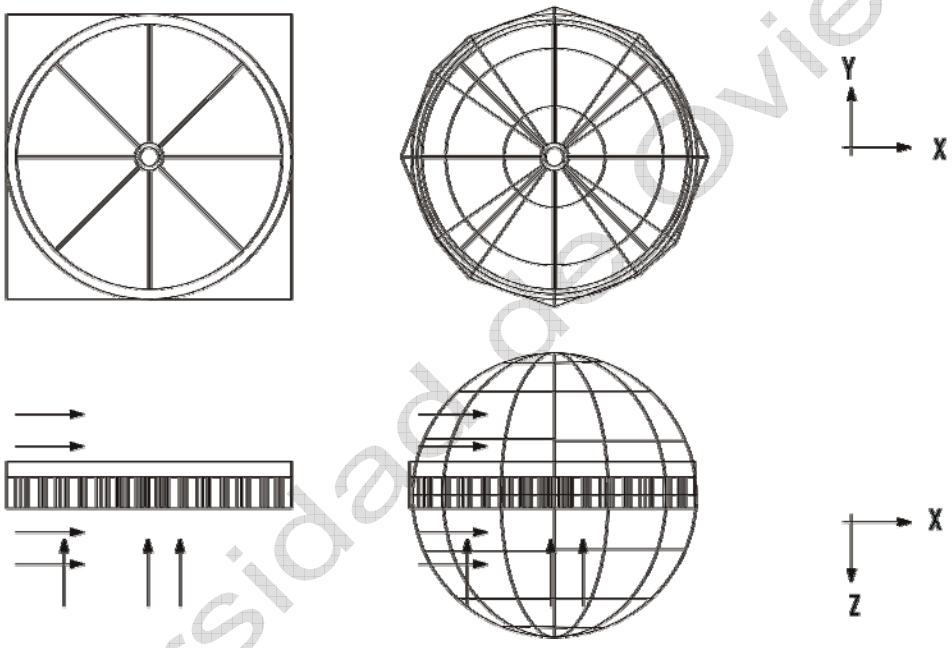

figura 3: orientación de los objetos en los VVEE

En la figura anterior se muestra el mejor volumen envolvente de una rueda según las circunstancias. Si el objeto va a ser intersectado mediante proyecciones perpendiculares al plano (*x, y*) entonces el mejor volumen envolvente será una esfera. Si las proyecciones son, por el contrario, perpendiculares al plano (*x, z*), el mejor volumen envolvente será un prisma rectangular.

Para averiguar a priori la efectividad de un VE es necesario hallar el *área nula*, ya que permite calcular la probabilidad de que un rayo interseque el VE sin tocar el objeto encerrado. Dado que *el área nula depende de la dirección del rayo*, se ha de calcular la *media de ajuste*, como la media de las áreas nulas proyectadas en todas las direcciones, suponiendo que las trayectorias de los rayos son aleatorias.

El ceñido de los VVEE a los objetos para minimizar el área nula, normalmente conlleva un coste mayor en el cálculo de los puntos de intersección rayo-VE. Por ejemplo, el VE de la figura 4-*c* se ajusta mejor que el caso

b, pero requiere una transformación lineal para cada intersección. Ver que *b* y *c* son dos tipos diferentes de VVEE, dado que presentan diferentes relaciones de coste/ajuste.

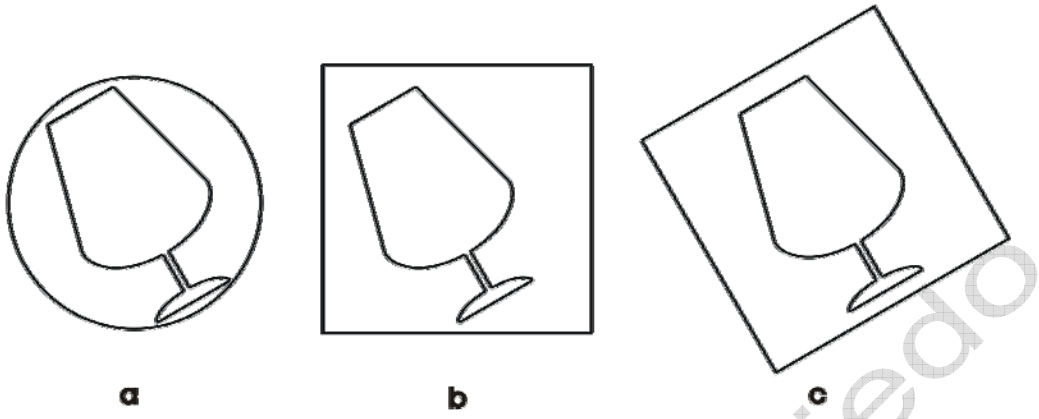

figura 4: ejemplos de VVEE

1.2.2 Planteamientos compuestos de volúmenes envolventes

Es posible realizar la composición de VVEE para lograr un mayor ajuste, como se muestra en la figura 5. Los planteamientos compuestos tienen sus ventajas e inconvenientes.

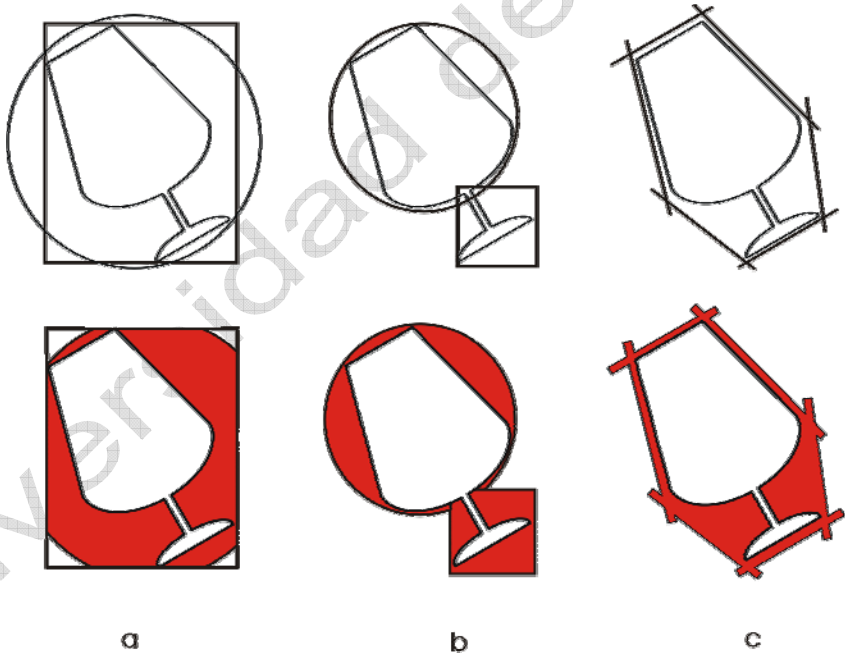

figura 5: composición de volúmenes envolventes

a) **Intersección de VVEE**: cada rayo ha de intersecar todos los volúmenes que forman el VE compuesto, antes de tener que probar con el objeto. Su coste es el acumulado de los costes de los volúmenes individuales, si el rayo interseca todos los VVEE, o bien el coste del primer VE, en caso de fallo (figura 5-a).

b) **Unión de VVEE**: El objeto ha de ser probado, si el rayo interseca con cualquiera de los VVEE. Si se produce un fallo (el rayo no interseca con el objeto), el coste nor-

malmente será mayor que en caso de la intersección de volúmenes; de lo contrario el coste total será menor.

1.2.3 Volúmenes envolventes de poliedros convexos

Kay y Kajiya [KAY86] introdujeron un nuevo tipo de volúmenes envolventes, basado en el criterio de ajustar las superficies convexas de los objetos. Para ello, los objetos se encierran en volúmenes envolventes poliédricos, formados mediante la intersección de un conjunto de *rodajas infinitas* (slabs), cada una de las cuales queda definida por un par de planos paralelos, que limitan al objeto. La posición de estos planos se determina a partir de las coordenadas máximas y mínimas del objeto, en un plano de proyección perpendicular al par de planos considerados.

figura 6: poliedro convexo formado a base de pares de planos paralelos

En 3D se requiere un mínimo de 3 pares de planos. A mayor número de rodajas, mayor será el ajuste, aunque por el contrario el coste de la intersección por lo común resultará también mayor.

A) Definición de los pares de planos

Cada par de planos puede quedar representado por la ecuación del plano

$$Ax + By + Cz + D = 0$$

siendo $D = D_{mín}$ o $D_{máx}$. También pueden quedar definidos mediante el vector unidad normal a los planos.

B) Intersección de los rayos con las rodajas

La intersección de los rayos con el volumen envolvente poliédrico se realiza con cada par de planos. Para ello se puede utilizar la ecuación que vimos en su momento,

$$t = -(ax_0 + by_0 + cz_0 + d)/(ai + bj + ck)$$

Conocidos los puntos de intersección de cada par de planos, si la intersección de los intervalos ($t_{mín}$, $t_{máx}$) de todas las rodajas es el vacío, entonces el rayo no interseca con el VE. Un caso así podemos verlo la figura 7, donde los intervalos formados en las rodajas A y B por el Rayo 1 se encuentran vacíos.

Por el contrario, si la intersección de los intervalos no está vacía, entonces el **máx**($t_{mín}$), es decir, el máximo de los puntos de intersección mínimos (de los dos puntos de intersección en la rodaja, el más cercano al observador) y el **mín**($t_{máx}$), o sea, el mínimo de los puntos de intersección máximos, son los puntos de intersección con el VE. En la figura 7, el Rayo 2 se encuentra en esta situación. Por lo tanto [máx($t_{mín}$), mín($t_{máx}$)] = [B1, A2].

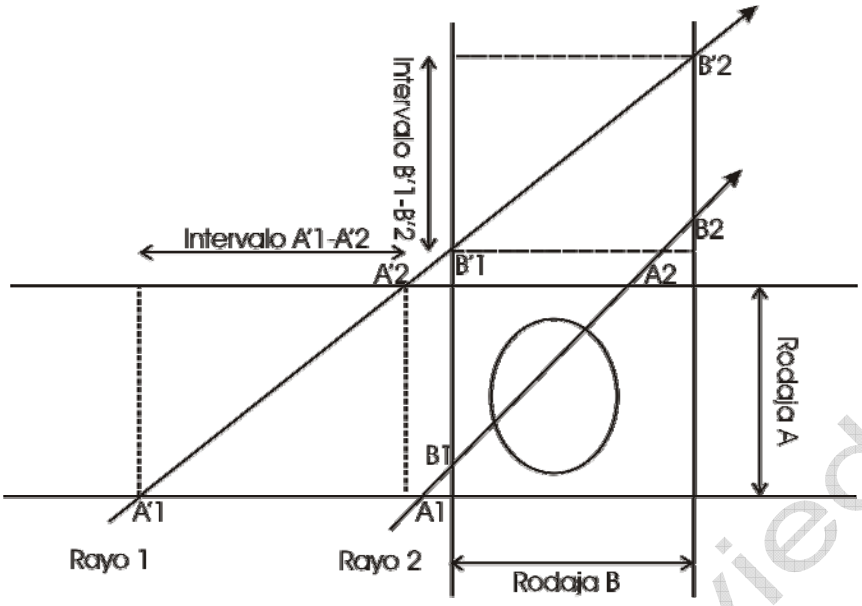

figura 7: intersección con el volumen poliédrico

C) Movimiento de los objetos

Utilizando volúmenes envolventes poliédricos, la traslación en el espacio de los objetos no afecta a estos VVEE. En cambio, el giro afecta al ajuste, lo que en principio sugiere que se debería corregir la orientación de cada par de planos, después de cada giro.

Sin embargo, existen grandes ventajas si se utiliza el mismo conjunto de rodajas para cada objeto, independientemente de que varíe su orientación. La más significativa radica en una importante reducción del coste de la intersección, cuando en el escenario aparecen varios o muchos volúmenes. Si escribimos la ecuación del cálculo de t como

$$t = (S + D) \cdot T$$

donde $S = (ax_0 + by_0 + cz_0 + d)$ y $T = -1/(ai + bj + ck)$, entonces S y T *solamente será preciso calcularlos una vez, para cada rayo y rodaja.*

1.2.4 Ventajas e inconvenientes de los volúmenes envolventes

Entre las objeciones que se plantean contra los VVEE, una de las principales es que la eficiencia de un VE depende de lo bien que se ajuste al objeto que contiene. Ver que esta técnica, incrementa el cálculo en aquellos casos en los que el rayo atraviese el volumen envolvente sin tocar al objeto. No obstante, en situaciones reales se consigue en general gran eficiencia, porque la mayor parte de los rayos que atraviesan los volúmenes llegan al objeto.

Además, aunque llegue a incrementar la eficiencia en la búsqueda de los puntos de intersección rayo-objeto, no se puede reducir su dependencia del número de objetos en el escenario. Así, por cada objeto en el escenario debe lanzarse un rayo contra su correspondiente VE, por lo que el tiempo de búsqueda viene dado en función de la complejidad del escenario.

Resumiendo, los volúmenes envolventes permiten realizar intersecciones simples en situaciones que serían bastante más costosas (en tiempo

de cálculo), pero no disminuyen el número de intersecciones. Desde un punto de vista teórico, para evaluar la eficacia global de esta técnica, los cálculos de intersección con los VVEE pueden quedar indicados por un factor constante, lo que hace que en un ray tracing exhaustivo el *tiempo teórico* varíe de forma lineal, en función de la complejidad del escenario.

1.3 Jerarquía de volúmenes envolventes

Rubin y Whitted [RUBI80] introdujeron el concepto de *jerarquía de volúmenes envolventes*, para lograr un *tiempo teórico* que varíe según una función logarítmica, dependiente del número de objetos presentes en el escenario.

Si se agrupa cierto número de volúmenes envolventes en un volumen mayor, es posible, con una única prueba de intersección, eliminar muchos objetos. Si el rayo no intersecta con el volumen envolvente raíz, no es necesario que se estudie su intersección con los objetos o volúmenes que están dentro de él.

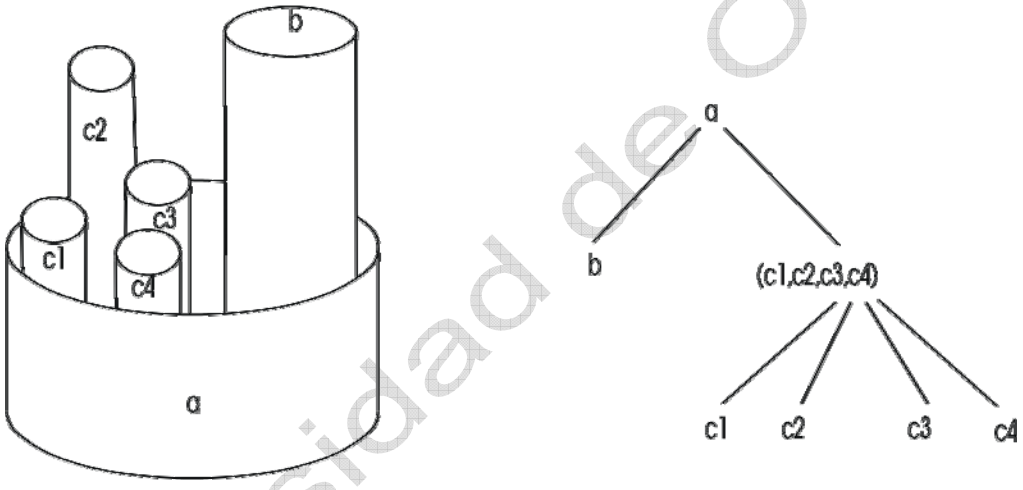

figura 8: jerarquía de volúmenes envolventes

La jerarquía de volúmenes se basa en la formación de una estructura en árbol, en la que el nodo raíz se corresponde con el volumen que envuelve a todos los objetos que hay en el escenario. Cada nodo hoja del árbol es un único objeto, y cada nodo interior está formado por un volumen envolvente y una lista de punteros a otros nodos del árbol.

Los autores citados desarrollaron una jerarquía en la que los volúmenes envolventes son paralelepípedos rectangulares (*cajas envolventes*). Estas cajas pretenden minimizar el tamaño del volumen y encerrar al objeto de la manera más ajustada posible.

1.3.1 Búsqueda de puntos de intersección en una jerarquía

El acceso a la jerarquía se efectúa lanzado el rayo sobre el nodo raíz. Si lo interseca, se accede a la lista de objetos asociada, averiguando a continuación si el rayo interseca con los volúmenes u objetos que contenga. El proceso se ha de continuar hasta que se haya completado el recorrido del árbol, momento en cual se devuelven las coordenadas del punto de intersección más próximo al observador.

```
BVH_intersection(rayo, nodo)
{
    if(nodo_terminal)
       intersecar_objeto(rayo, nodo.objeto);
    else
       if(intersecar_VE(rayo, nodo.VE))
          while(nodo.hijo) //para cada nodo hijo
             BVH_intersection(rayo, nodo.hijo)
}//Fin BVH_intersection()
```

Este es el pseudocódigo del procedimiento BVH_intersection dado por Glassner [GLAS89], en el que se intersecta el rayo con una colección de objetos organizados en una jerarquía de volúmenes envolventes.

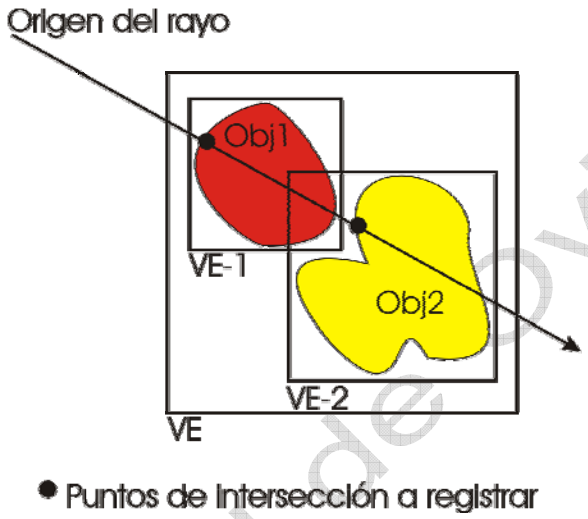

figura 9: búsqueda en la jerarquía de objetos

A) Generalidades y funcionamiento de este procedimiento

- El procedimiento "intersecar_objeto()" es el responsable de llamar al algoritmo de intersección rayo-objeto más apropiado (en función del argumento "nodo.objeto"), según sea el objeto (primitiva), con el que ha de intersecar el rayo.
- El argumento "rayo" ha de indicar las coordenadas 3D del origen del rayo, su vector de dirección, y la *distancia mínima*. Los puntos de intersección que caigan fuera de la distancia mínima serán ignorados.
- Cuando un punto de intersección rayo-objeto (**no** rayo-VE) se encuentra más cerca del origen del rayo que los puntos intersecados hasta el momento, "intersecar_objeto()" guardará en "distancia_minima" las coordenadas del nuevo punto (figura 9), por lo que este campo deberá ser inicializado con un valor máximo. Además, también salvará toda aquella información que pueda ser necesaria para los modelos de intensidad.
- La función "intersecar_VE()" es similar al procedimiento "intersecar_obje-to()", excepto que retorna un valor booleano que indica si se ha producido una intersección. Además, *no altera el valor almacenado en "distancia_mini-ma"*. El reajuste de la distancia mínima de los puntos de intersección (los que son los visibles por el observador), sólo debe realizarse en los casos de intersección con los objetos. Esta función se utiliza exclusivamente para determinar si un

volumen envolvente es intersecado por un rayo.

- El proceso comienza en el nodo raíz del árbol (que representa un volumen envolvente que rodea a todos los objetos del escenario) y un rayo infinito. *"distancia_minima"* se inicializa con el valor máximo.
- Cada llamada recursiva al procedimiento "BVH_intersection()", desciende otro nivel en la jerarquía. La recursión termina con un test de intersección rayo-objeto en las hojas del árbol. En cada nivel se analiza si el rayo interseca con algún volumen envolvente, y solamente se desciende en aquellos casos en los que esto ocurra. De este modo, muchos objetos quedan automáticamente descartados como candidatos a intersecar (poda del árbol), lo que puede suponer, dependiendo de la complejidad del escenario, un considerable ahorro de tiempo.

✓ El ajustar la distancia mínima de intersección proporciona gran optimización. Conociendo el punto de intersección más cercano al observador, *todos los objetos o volúmenes envolventes* que intersecten con el rayo a una distancia superior pueden ignorarse. Esto proporciona un segundo mecanismo para descartar intersecciones inútiles.

1.3.2 Predicción de la efectividad de una jerarquía

A la hora de predecir la efectividad de las jerarquías de volúmenes, obviamente tienen mucho que decir factores como el tipo de volumen utilizado, su capacidad para ceñirse a los objetos, la distribución de los objetos en la jerarquía, o el número de niveles en la jerarquía.

Como vemos, son muchos los parámetros que intervienen en la evaluación del comportamiento de una jerarquía de VVEE, por lo que no es sencillo establecer patrones de eficacia; no obstante, algunas técnicas de evaluación de las jerarquías han sido desarrolladas. Veamos uno de los métodos más populares.

A) Método Goldsmith y Salmon

Las jerarquías creadas durante el proceso de modelado tienden a ser claramente superficiales, con una estructura orientada más al control de los objetos, que a minimizar el número de intersecciones del rayo con los objetos.

Goldsmith y Salmon [GOLD87] han desarrollado un método para determinar la calidad de una jerarquía, basado en la estimación del coste de la intersección de un rayo con ella. En este método se supone que *todos los volúmenes envolventes tienen el mismo coste*, es decir, el coste de un VE será directamente proporcional al número de rayos que inciden sobre él. Recordemos que la probabilidad de que un volumen sea atravesado por un rayo (sin intersecar con el objeto), es proporcional al área del volumen proyectada en el plano de visión (área nula del rayo).

Como cada volumen está incluido dentro de otro volumen padre, la probabilidad de que un rayo intersecte con un determinado volumen (V_i), se puede aproximar por A_i / A_r, donde A_i es la superficie del volumen V_i, y A_r es la superficie del volumen padre.

Si un rayo intersecta con un volumen, debemos calcular la intersección con cada uno de sus k hijos. Entonces, el coste total de ese volumen en número de intersecciones es $k \cdot A_i / A_r$.

El coste del nodo padre, como $A_i = A_r$, es k. El coste de un nodo hoja es 0, ya que $k = 0$.

Para estimar el coste de la jerarquía sumaremos los costes de cada uno de los volúmenes envolventes que la componen.

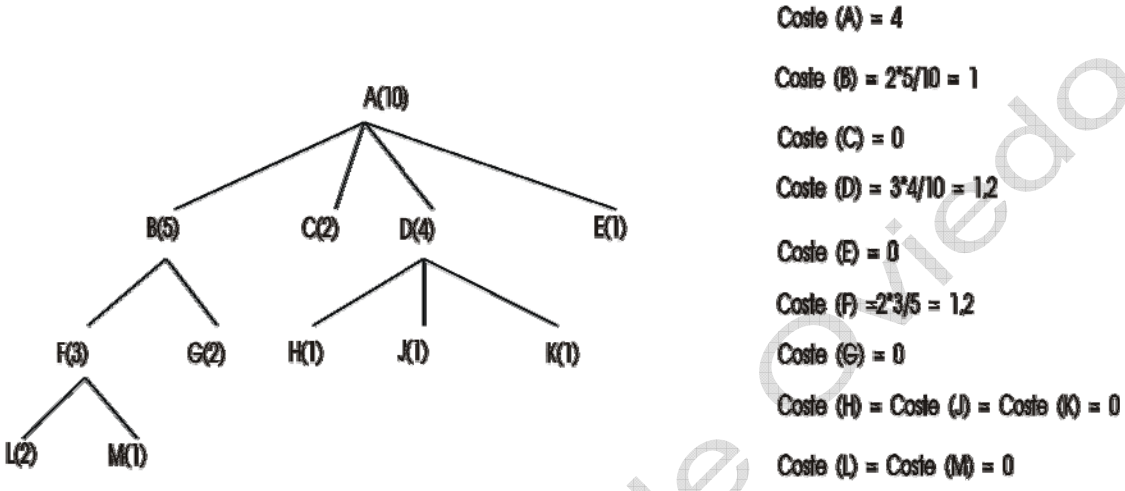

figura 10: ejemplo del coste de una jerarquía

1.3.3 Construcción de jerarquías de volúmenes envolventes

Construir una jerarquía implica dos tipos de decisiones:

I) Elegir el grupo de objetos que vamos a encerrar en un mismo volumen.

II) Elegir el tipo de volumen que vamos a utilizar para encerrar ese grupo de objetos.

Ya hemos estudiado que el mejor volumen envolvente es aquel que minimiza el coste de búsqueda de puntos de intersección. También hemos tratado el método de Goldsmith y Salmon para predecir la eficiencia de una jerarquía.

Ahora nuestro problema estriba en construir una jerarquía, seleccionando los volúmenes envolventes y los grupos de objetos. Se trata de un gran desafío, porque el número de posibles jerarquías crece de forma exponencial según sea el número total de objetos en el escenario, lo que hace totalmente impracticable una búsqueda exhaustiva.

A) Modelo de creación de una jerarquía

Goldsmith y Salmon [GOLD87] desarrollaron un método de construcción incremental de jerarquías, en la que los nodos se van añadiendo de uno en uno, procurando *reducir al mínimo el aumento del área de exposición* a los rayos. Cada nodo que se incorpora en la jerarquía, o se hace hijo de un nodo ya existente, o bien se incorpora como padre sustituyendo a otro existente, mediante un nodo que contenga al anterior, y que por tanto pasa a ser hijo del recién incorporado. En cada caso, en vez de evaluar el coste del nuevo

árbol, lo que se hace es determinar el incremento que supone la adición del nuevo nodo.

Por ejemplo, si el nodo se añade como hijo de un nodo ya existente, supondrá un incremento en el área de dicho nodo y en su número de hijos. Así, el incremento vendrá dado por

$$I = k \cdot (A_{nuevo} - A_{viejo})$$

donde A_{nuevo} y A_{viejo} son las superficies nueva y anterior del nodo padre, y k el número de hijos que tenía inicialmente.

Un criterio de optimización simple consiste en evaluar el incremento que supondría añadir un nuevo nodo en cada posición posible del árbol, y elegir aquella que represente un menor incremento, es decir, siguiendo el camino (rama) del mínimo esfuerzo (coste).

Goldsmith y Salmon, crearon un método heurístico de búsqueda que comienza en la raíz, evaluando el coste que supondría añadir un nodo hijo, dentro de otro dado. La búsqueda se continúa en la subrama en la que suponga menor coste, hasta alcanzar los nodos hoja, los cuales quedarán incorporados en la jerarquía de modo conveniente, basándose en las evaluaciones realizadas.

Determinar de esta manera el punto de inserción de un solo nodo, requiere una búsqueda cuyo coste es función de log n, siendo n el número de nodos de la jerarquía. El coste de la búsqueda de los puntos de inserción de la jerarquía completa será función de $n \cdot \log n$.

Este tipo de procesos de búsqueda y evaluación basados en métodos heurísticos, generan una jerarquía que normalmente no será la óptima. Sin embargo estas técnicas pueden crear jerarquías que proporcionan ahorros significativos en costes de intersección.

B) Editores de estructuras

Existen otros mecanismos para la construcción de jerarquías de VVEE, como los *editores de estructuras*, que requieren la intervención del usuario, pero que en según qué casos pueden ser más eficientes que los generadores automáticos del estilo del anterior.

Los editores permiten crear jerarquías de abajo arriba (down-top), agrupando los objetos según el criterio del diseñador, y por tanto tienen la desventaja de que se requiere la intervención humana.

1.3.4 Inconvenientes de las jerarquías de volúmenes envolventes

Usando una jerarquía de volúmenes envolventes se consigue un ahorro significativo del tiempo de búsqueda de puntos de intersección, pero se requiere un preproceso costoso para determinar y construir el tipo de jerarquía más apropiado.

Los nodos, o más estrictamente las ramas del árbol, sólo pueden ser creados si los objetos están lo suficientemente cerca unos de otros. La can-

tidad de ramas y la profundidad del árbol dependerán de la naturaleza y disposición de los objetos en el escenario.

1.4 Técnicas de subdivisión espacial

Buscando los mismos objetivos que la técnica de las jerarquías que acabamos de ver (o sea, minimizar el número de intersecciones rayo-objeto), la subdivisión espacial 3D también se basa en la construcción de VVEE, aunque aplicando una filosofía diferente.

En el método de la subdivisión espacial el espacio del escenario se divide en regiones. A cada región se le asigna una lista con todos los objetos que contiene, total o parcialmente; las listas se completan asignando a cada objeto la celda o celdas que lo contienen.

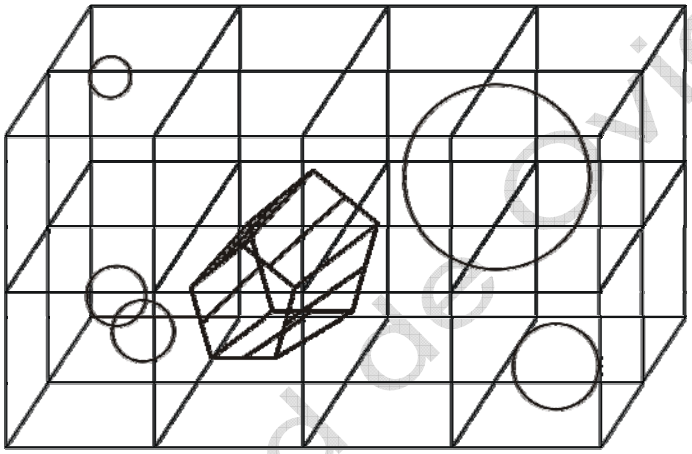

figura 11: división del escenario en subvolúmenes

La subdivisión espacial requiere un preproceso para crear la estructura de datos donde quedará registrada la información relativa al espacio que ocupan los objetos en el escenario.

La diferencia fundamental entre las técnicas de jerarquías de volúmenes envolventes y de subdivisión espacial, estriba en que las primeras crean los VVEE basándose en la distribución espacial de un conjunto dado de objetos (Bottom-Up); en cambio, las de subdivisión espacial forman grupos de objetos en función de unos volúmenes envolventes dados (Top-Down).

1.4.1 Conceptos básicos de la subdivisión espacial

El preproceso que exigen las técnicas de subdivisión espacial, consiste en dividir el volumen total del escenario en pequeños volúmenes o *vóxeles* (el término vóxel es la extensión tridimensional de su homónimo bidimensional píxel). La forma de definir estos vóxeles es lo que marca la diferencia entre las técnicas de subdivisión espacial. Una vez definidos, los vóxeles juegan el mismo papel en todas las técnicas.

Atendiendo a las características de los vóxeles, las técnicas de subdivisión espacial se clasifican en dos grandes grupos:

a) Técnicas de **subdivisión espacial no uniforme** o **adaptativa** (dividen el escenario en particiones de distinto tamaño).

b) Técnicas de **subdivisión espacial uniforme** (dividen el escenario en particiones del mismo tamaño).

La decisión de utilizar una técnica uniforme o una técnica adaptativa, depende de varios factores, como el número de objetos presentes en el escenario, su distribución, y el coste de su intersección.

La gran ventaja de las técnicas de subdivisión es que sólo los objetos asignados a los vóxeles atravesados por los rayos deben ser probados para una posible intersección.

Una observación importante es que los vóxeles se procesan en el mismo orden en que son encontrados por el rayo, lo que garantiza que cualquier vóxel intersecado por un rayo estará más cerca del origen que los restantes (en la dirección del rayo). Por tanto, una vez encontrado un punto de intersección, normalmente no será preciso considerar el contenido de los restantes vóxeles. Esto reduce considerablemente el número de objetos que se han de probar para intersección.

Cuando un rayo intersecta con un vóxel, se ha de averiguar si hay intersección con cada uno de los objetos contenidos en él escogiendo, como siempre, la intersección que se encuentre más cerca del origen del rayo. Por esta razón interesa que los vóxeles contengan el menor número posible de objetos, ya que así en cada vóxel el total de intersecciones rayo-objeto será mínimo.

1.4.2 Subdivisión espacial no uniforme o adaptativa

Como acabamos de ver, las técnicas de subdivisión espacial no uniforme dividen el espacio en regiones de distinto tamaño, en función de la distribución de los objetos en el escenario. La división en celdas (vóxeles) se realiza *sin solapamientos*, es decir, el "pegado"de todas las celdas permitiría recomponer el espacio original del escenario. La no uniformidad de las celdas permite hacer más subdivisiones en aquellas zonas que tengan mayor densidad de objetos, y menos subdivisiones en las regiones del escenario que estén prácticamente vacías, de ahí el apelativo de adaptativas.

Puesto que las celdas son de tamaños diferentes, estas técnicas requieren la presencia de estructuras de datos que registren el tamaño, la disposición, y los objetos contenidos en las celdas. Las estructuras de datos más usadas para representar el escenario son los **árboles octales** (*Octrees*) y los *árboles binarios* o **BSP** (*Binary Spatial Partition*).

.

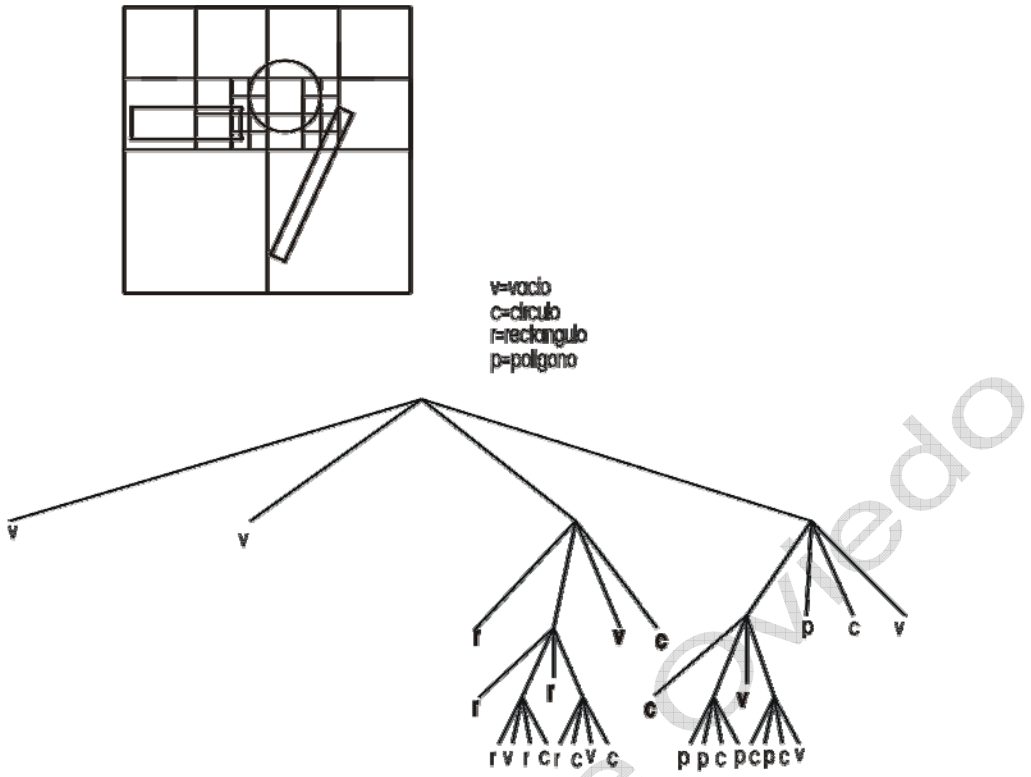

figura 12: representación de un quadtree en 2D que es similar al octree en 3D

A) Árboles octales

Un octree es una estructura de datos que, en último término, describe cómo se distribuyen los objetos en el espacio tridimensional del escenario. Se trata de un árbol (estructura jerárquica) de orden 8, por lo que cada nodo, en un nivel dado, tiene 8 descendientes. Cada nodo especifica (tiene asignada) una región cúbica del espacio (vóxel), cuyo tamaño depende del nivel en que se encuentre definido el nodo; la raíz del octree está asociada a todo el espacio del escenario.

Los octrees se generan mediante la subdivisión recursiva del espacio en ocho partes (octantes). Así, una subregión que esté ocupada por objetos, se irá subdividiendo según este criterio, hasta obtener unos vóxeles de tamaño apropiado, en función de la resolución requerida o de la memoria disponible.

En el octree hay dos tipos de nodos terminales:

- Aquellos que corresponden a subregiones vacías, y
- Aquellos que corresponden a vóxeles de tamaño mínimo, que cumplen el *criterio de hoja* o terminal. Normalmente este criterio especifica que en cada vóxel haya un solo objeto, o parte del objeto (figura 12).

Nótese que, al estar definido en tres dimensiones, si decimos que un objeto está dentro de un vóxel, implicará que la superficie del objeto está completamente contenida por el vóxel. Durante el preproceso de construcción del árbol octal, se averigua si los objetos están incluidos en un vóxel, evaluando la ubicación de las superficies en relación con las seis caras del

vóxel. Si la superficie intersecta a una de las caras, el objeto se añade a la lista de candidatos asociados a ese vóxel; por supuesto, si la superficie no interseca con las caras del vóxel, pero se encuentra en su interior, también será incluido en la lista.

Actualmente existen dos métodos para representar un escenario mediante un octree:

I) Representar los objetos por el conjunto de "vóxeles compactos" que ocupa, a modo de mosaico 3D. Esta técnica es la utilizada con la información volumétrica. Por lo común requiere gran cantidad de espacio de almacenamiento, pero hoy día esto ya no es un problema.

II) Cuando los objetos está definidos vectorialmente (el caso más común), el esquema de vóxel compacto no sirve, a no ser que se haga una voxelización (discretización) del modelo. Con modelos vectoriales se ha de utilizar un octree de celdas o vóxeles huecos, como el que acabamos de ver arriba, donde cada vóxel terminal tiene una lista de punteros asociada, indicando los objetos que contiene, total o parcialmente.

B) Uso de los árboles octales en ray tracing

La utilización de los octrees con ray tracing, en lo básico es similar en los dos planteamientos anteriores, por lo que nos limitaremos a ver cómo se utilizan de los árboles octales que se generan cuando los modelos son vectoriales.

Una vez que el octree está construido, en vez de hacer cálculos de intersección entre el rayo lanzado y los objetos del escenario, lo que haremos será trazar el rayo de vóxel en vóxel, según esté organizado el escenario, siguiendo el orden establecido por el rayo. En cada vóxel por donde pasa el rayo habrá un número pequeño de objetos (1 ó 2), con los que puede intersectar. Dado que ahora podemos encontrar rápidamente el nodo del octree correspondiente a la subregión activa (por la que está pasando el rayo), tenemos acceso inmediato a los objetos que están en o cerca de la trayectoria del rayo. Los cálculos de intersección, sólo serán necesarios para estos objetos.

Si el espacio total se ha subdividido hasta llegar a un nivel en el que cada vóxel contenga sólo 1 ó 2 objetos, el número de test de intersección en una región es pequeño, y no tiende a crecer si aumenta la complejidad del escenario.

El algoritmo básico de búsqueda de los objetos en el octree queda como sigue:

```
BuscarObjetosEnOctree(rayo)
{
  punto_en_voxel = rayo.origen;

  do
  {
     /* Se localiza el vóxel que contiene el origen del rayo,
     indicado por "punto_en_voxel" */
```

```
        localizar_voxel(punto_en_voxel);

        /*Se interseca el rayo con los objetos asociados a cada
        vóxel. Devuelve el más cercano al origen */
        for(cada objeto asociado al vóxel)
           intersecar_objeto(rayo, objeto);

        /* Si no hay intersección se ha de localizar un nuevo pun-
        to en el próximo vóxel cruzado por el rayo */
        if(no intersección)
           punto_en_voxel = nuevo_voxel();
     }while(intersección o "punto_en_voxel" pertenezcan al espa-
     cio);
  }//Fin BuscarObjetosEnOctree()
```

Para determinar los objetos cercanos a un rayo, se han de encontrar los vóxeles del espacio que son atravesadas por el rayo. Esto implica que, en su trayectoria, se ha de probar el rayo (test de intersección con los vóxeles), dentro y fuera de cada vóxel.

El proceso de avance del rayo comienza buscando en el octree la información del vóxel en que se encuentra el origen del rayo. Si no hay intersecciones rayo-objeto en este vóxel inicial, el rayo deberá intersecarse con el siguiente vóxel que atraviese. Esto se consigue calculando la intersección del rayo con los límites del vóxel activo, averiguando de este modo el punto por donde el rayo abandona el vóxel. Para encontrar el nodo en el octree correspondiente al siguiente vóxel, se utiliza un punto del rayo situado a una corta distancia del punto de salida, después de asegurarse que pertenecer al vóxel vecino[1]. Todos los objetos asociados al nuevo vóxel serán utilizados para encontrar posibles intersecciones con el rayo.

El proceso se repite avanzando el rayo de vóxel en vóxel, hasta que se encuentre una intersección con un objeto, o bien, hasta que el rayo se salga del espacio total representado.

Vemos que una de las operaciones más frecuente será hallar el nodo en el octree que contenga un punto determinado (*x, y, z*). Hay dos formas de realizar esta operación:

1) El método más simple para hallar el nodo del octree que contiene el punto (*x, y, z*), es aquél que aprovecha la regularidad y uniformidad (en cada nivel) de la división espacial, y del octree que la representa.

Una comparación de coordenadas determinará qué nodo hijo representa al vóxel que contiene el punto (*x, y, z*). A su vez, otra comparación de las coordenadas del vóxel correspondiente a dicho nodo determinará cuál de sus hijos representará el subvóxel que contiene el punto (*x, y, z*). La búsqueda irá descendiendo por el árbol hasta llegar a un nodo terminal.

Si las coordenadas (*x, y, z*) fuesen enteras, se puede mejorar el método anterior, transformado las coordenadas (*x, y, z*) en un *código octal*. Así, cada cifra de este código proporcionará directamente el nodo donde se encuentra

[1] Conociendo el tamaño mínimo de los vóxeles y las coordenadas de entrada en ellos, no hay problema para prevenir que el avance en la dirección del rayo sea lo suficientemente grande como para saltarse uno o más vóxeles.

el punto (*x, y, z*) en un nivel dado del árbol. Por tanto, solamente resta ir descendiendo por el árbol siguiendo la rama indicada por el código octal, hasta encontrar un nodo terminal.

El máximo número de nodos sometidos a test durante la búsqueda, necesariamente ha de ser igual a la profundidad máxima del árbol.

2) Otro método posible es el propuesto por Glassner [GLAS88]. En él, se hace corresponder un nombre único con cada nodo del octree. El convenio utilizado para la generación de estos nombres es etiquetar cada una de las ocho partes de un vóxel, con un número del 1 al 8. Al subdividir un nodo, el nombre de sus hijos se obtiene multiplicando por 10 el nombre del padre, y sumando el nombre correspondientes del hijo[2].

Para acceder a los datos asociados al nombre de un nodo se usa el nombre (numérico) del nodo, para localizar un puntero a una tabla hash, mediante una función hash determinada. Por ejemplo, puntero_lista = nombre_nodo *mod* (tamaño_tabla_hash) puede ser una función apropiada, siendo "puntero_lista" el puntero a la lista de objetos candidatos, asociada al nodo terminal.

Con este planteamiento, la búsqueda comienza en el nodo raíz, determinando en cuál de los ocho octantes está el punto, construyendo el nombre correspondiente a ese octante, y consultando en la tabla hash el estado de dicho octante. El proceso se repite hasta que se encuentra un nodo hoja, o hasta que se genera el nombre de un nodo inexistente.

C) Árboles BSP

Kaplan [KAPL85] introdujo un método alternativo para la subdivisión del espacio en vóxeles, basado en una **p**artición **b**inaria del e**s**pacio, cuyo resultado queda reflejado en los *árboles BSP*.

La estructura auxiliar usada para representar la subdivisión espacial mediante un árbol BSP es esencialmente un árbol binario, en el que cada nodo, como su nombre indica, tiene solamente dos subnodos.

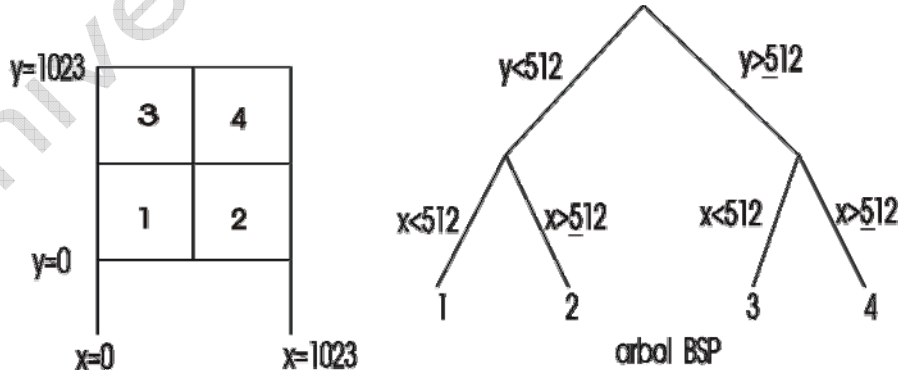

figura 13: árbol BSP de un espacio en 2D

[2] Los nombres numéricos generados de este modo son muy parecidos a los códigos octales, solo que en éstos las cifras van de 0 a 7.

En la figura anterior se da una idea de cómo se generaría un árbol BSP sobre una escena de dos dimensiones. En 3-D, cada nodo no terminal del árbol BSP representa un plano de partición que divide al espacio en dos mitades. Un nodo terminal representa una región que no puede subdividirse más y, como en los octrees, tiene asociados punteros a las estructuras de datos que representan los objetos con los que intersecta, o contiene, su correspondiente vóxel.

Para buscar la información en un BSP podrían utilizarse las técnicas de nombre y hashing propuestas por Glassner para los octrees, aunque lo más común es utilizar punteros explícitos a los nodos hijos.

La utilización de los BSP para localizar los vóxeles es similar a la descrita anteriormente, aunque Jansen [JANS86] ha planteado un algoritmo de rastreo alternativo.

En él, para seguirle la pista a un rayo, lo que se hace es ir descendiendo recursivamente por las ramas del árbol BSP con nodos asociados a los vóxeles atravesados por el rayo, usando el nodo de cada partición solamente una vez por rayo. A esta forma de búsqueda Jansen la llama "muestreo recursivo". El pseudocódigo del algoritmo queda como sigue:

```
BSP_intersection(rayo, nodo)
{
    if(rayo.intervalo está vacío O nodo es nulo)
       return;
    if(nodo_terminal)
       for(cada objeto asociado al nodo)
          intersecar_objeto(rayo, objeto);
    else
    {
       rayo_cercano = primera mitad del rayo original;
       nodo_cercano = primera mitad del escenario;
       BSP_itersection(rayo_cercano, nodo_cercano)

       if(no hay interseccion)
       {
          rayo_lejano = segunda mitad del rayo original;
          nodo_lejano = segunda mitad del escenario;
          BSP_itersection(rayo_lejano, nodo_lejano)

       }
    }
}//Fin BSP_intersection()
```

En este procedimiento la distancia de intervalo del rayo se divide en dos subintervalos, que se corresponden con los segmentos del rayo en cada cara del plano. Si en el segmento más cercano al origen del rayo se encuentra una posible intersección con un objeto, se continúa el estudio recursivo en dicho segmento, descartando el otro. En caso contrario la búsqueda se continúa en el segmento más lejano.

Si en un segmento no hay nada, el procedimiento termina y se continúa la búsqueda en la instrucción siguiente del procedimiento que realizó la llamada recursiva.

D) Subdivisión adaptativa con BSP

Cuando un árbol BSP se utiliza para representar una subdivisión del espacio en celdas cúbicas, no se aporta ninguna ventaja sobre los octrees (figura

14). La idea de los árboles BSP ha sido desarrollada pensando en una subdivisión del escenario en celdas de cualquier tamaño.

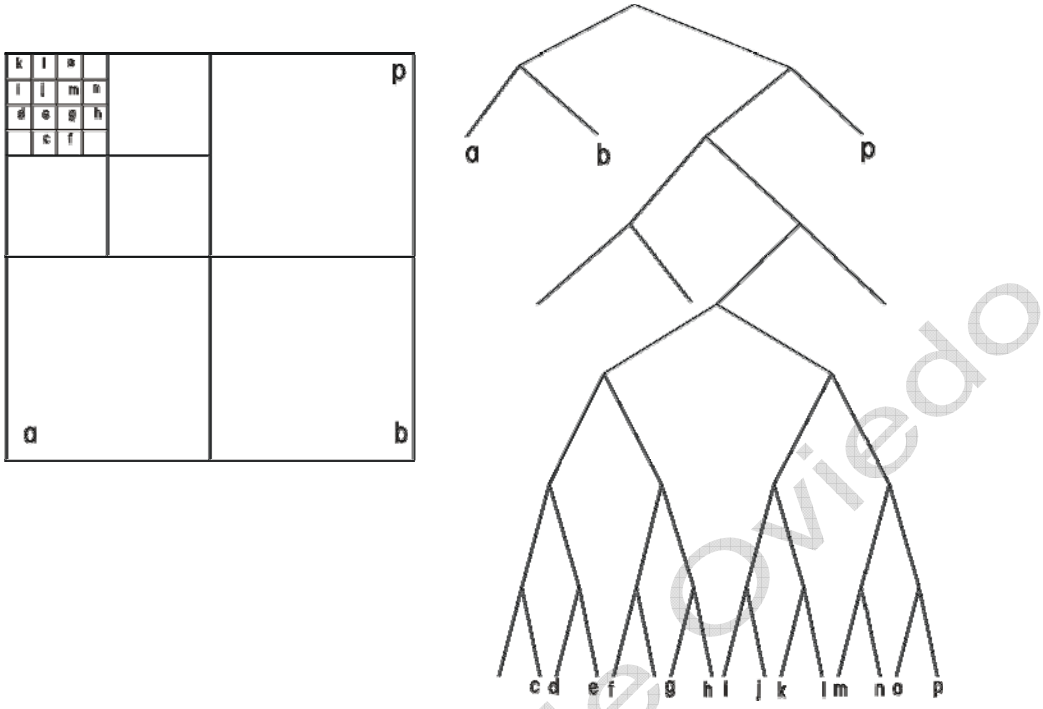

figura 14: subdivisión un espacio 2D utilizando celdas cuadradas

Un esquema donde la posición de los planos de partición dependa de la distribución de los objetos, presenta ciertas ventajas. Así, si las líneas de partición se escogen de tal forma que las regiones conseguidas tengan igual número de objetos a uno y otro lado, se conseguirá un árbol más equilibrado (figura 15).

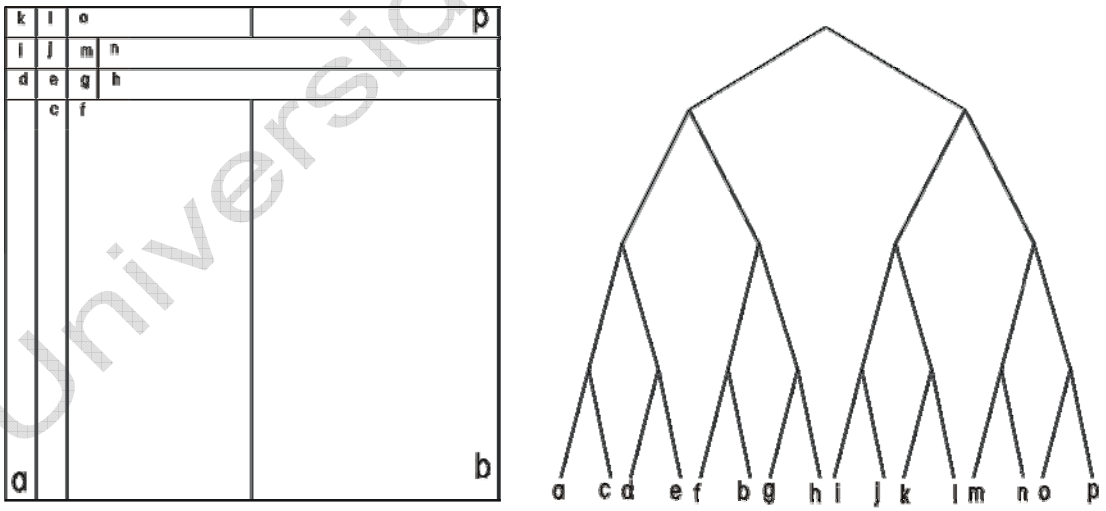

figura 15: BSP adaptativo de un espacio 2D, con objetos desigualmente distribuidos

1.4.3 Subdivisión espacial uniforme

Fujimoto [FUJI86] introdujo un método diferente de subdivisión espacial en el que vóxeles del mismo tamaño se organizan en matrices regulares tridimensionales. Esta organización recibe el nombre de SEADS (Spatially Enumerated Auxiliary Data Structure).

La idea general de esta organización es bastante parecida a la de las técnicas de subdivisión adaptativas. Las listas de candidatos que serán consultadas para detectar las posibles intersecciones dependerán de los vóxeles atravesados por los rayos, según el orden establecido por éstos.

Se introducen sin embargo dos diferencias, que son consecuencia directa de la regularidad de los vóxeles:

1) La subdivisión es totalmente independiente de la estructura del escenario.

2) El acceso a los vóxeles puede realizarse eficientemente mediante algoritmos incrementales.

La primera diferencia es una desventaja, pues representa una carencia de adaptatividad, que se ve compensada con las ventajas que proporciona el segundo punto.

A) Rastreo de los vóxeles en una SEADS: algoritmo 3DDDA

La clave para el acceso eficiente a los vóxeles está en que, para localizarlos a lo largo de la trayectoria del rayo en una matriz rectangular 3D, se procede de igual forma que cuando se traza una línea en una matriz de píxeles.

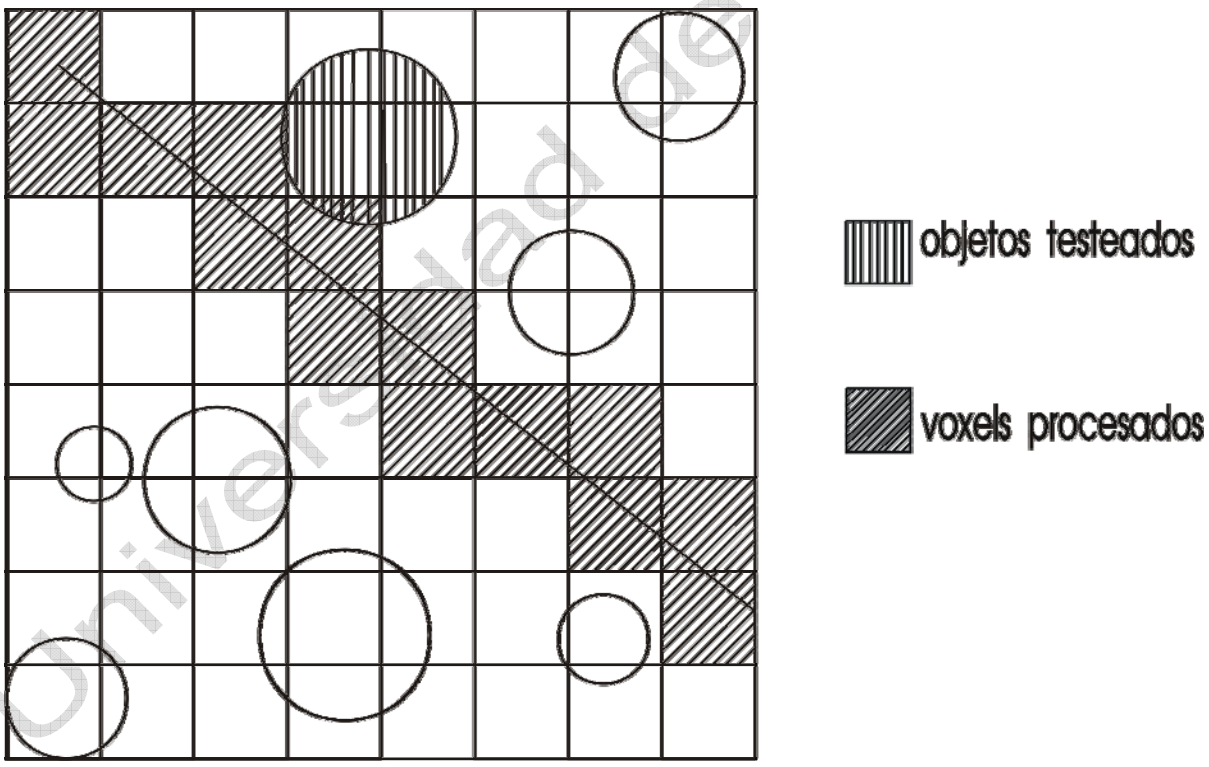

Figura 16: subdivisión espacial uniforme en 2D

Para explotar este hecho, Fujimoto desarrolló un algoritmo al que llamó analizador tridimensional de diferencias digitales (3DDDA), que es la versión 3D del generador de líneas DDA, utilizado en 2D. El 3DDDA permite calcular los índices de los sucesivos vóxeles por los que pasa el rayo de forma muy eficiente. Para que trabaje adecuadamente, antes de comenzar el avance, el origen de los rayos ha de quedar ubicado en el centro de los vóxeles.

En cuanto a la información asociada a los vóxeles (punteros a los objetos que contienen o intersecan) es conveniente organizarla en una matriz 3D, ya que el 3DDDA genera índices enteros (*i, j, k*), por lo que resulta muy sencillo acceder a la información, si en cada elemento de la matriz 3D se encuentra un puntero a la lista de objetos asociados.

El algoritmo de búsqueda dado por Fujimoto es el siguiente:

```
SEADS_intersection(rayo, nodo)
{
   /* Se calcula el (I, j, k) del vóxel que contiene el origen
   del rayo */
   calcular_ijk(rayo.origen);

   /* Se inicializa el 3DDDA en el centro del vóxel */
   centrar_rayo(dir_rayo, rayo.origen);

   /* Comienza el bucle de búsqueda */
   do
   {
      for(cada objeto asociado con el vóxel (i,j,k))
         intersecar_objeto(rayo, objeto);
      if(no intersección)
         nuevo_ijk = 3DDDA();
   }while(intersección o (i, j, k) estén en los límites);
}//Fin SEADS_intersection()
```

B) Ventajas e inconvenientes del algoritmo 3DDDA

- Acceso rápido a los vóxeles que contienen los objetos
- Falta de adaptatividad

Si se aumenta la precisión, necesariamente ha de disminuir el tamaño de los vóxeles, por lo que:

a) Se hace más costoso cruzar los rayos por los espacios vacíos.

b) Aumenta mucho las necesidades de memoria para almacenar la matriz tridimensional con la información asociada.

C) 3DDDA aplicado a los árboles octales

El algoritmo 3DDDA puede ser modificado para acceder a los nodos de un octree. Como 3DDDA sólo se puede aplicar en subdivisiones espaciales uniformes, su uso está restringido al paso horizontal entre vóxeles "hermanos" del octree. Cada grupo de ocho hermanos puede interpretarse como una pequeña matriz uniforme; 3DDDA proporciona un medio eficiente para pasar el rayo a través de ellos. Después de avanzar cuatro hermanos, como mucho, debe desarrollarse otra vez un muestreo vertical para localizar el siguiente bloque de ocho hermanos.

1.5 Técnica del buffer de elementos

Otra de las técnicas pertenecientes al grupo que trata de minimizar el número de intersecciones rayo-objeto, se conoce como **buffer de elementos**.

Weghorst, Hooper y Greenberg [WEGH84], sugieren la utilización de un trazador de rayos híbrido, donde la intersección con el rayo primario se evalúa durante una fase de preproceso, utilizando un algoritmo de superfi-

cies ocultas, similar al *Z-buffer*, donde se asigna un registro (de tamaño apropiado) a cada píxel del plano visual. Este método es más eficiente encontrando los puntos de intersección correspondientes a los rayos primarios, que efectuando el trazado de los propios rayos. En definitiva, se trata de encontrar los objetos más cercanos al observador utilizando un buffer de elementos, antes de aplicar ray tracing, pues resulta más rápido.

Así, lo que esta técnica propone es grabar en el registro asociado a cada píxel, la identidad del objeto más cercano. Más tarde, sólo este objeto necesita ser procesado por el trazador de rayos, para determinar el punto exacto de intersección del rayo primario que es trazado por ese píxel. De esta manera se consigue eliminar el coste de intersección asociado a este primer impacto del rayo. Los restantes niveles del árbol de rayos han de ser procesados aplicando procedimientos habituales.

Estudios realizados muestran que un trazador de rayos híbrido que incluya los métodos de jerarquía de volúmenes envolventes, buffer de elementos y control adaptativo de la profundidad del árbol de rayos (lo veremos más adelante), proporciona una gran mejora. Para un escenario con un nivel normal de complejidad, el tiempo de cálculo empleado es aproximadamente la mitad que el utilizado con un trazador de rayos sencillo, que aplique la técnica de volúmenes envolventes esféricos.

1.6 Técnicas direccionales

Otro grupo de técnicas de aceleración que también busca minimizar el número de intersecciones rayo-objeto, se conoce como **técnicas direccionales**. Estas técnicas ya no consideran el rayo como algo individual, sino que lo sitúan dentro de un escenario tridimensional. Para ello, a la hora de realizar la subdivisión del espacio incorporan en las estructuras de datos información explícita relativa a la dirección del rayo. Así, las operaciones que se aplicaban a un solo rayo, pueden aplicarse ahora sobre un conjunto de rayos, mediante la adición de un bucle interno.

La desventaja que plantean es que necesitan mayor capacidad de almacenamiento. Dentro de las técnicas direccionales podemos encontrar la del *Buffer de luz*, la de la *Coherencia de rayos*, o la de *Clasificación de rayos*. De éstas, solamente analizaremos la primera.

1.6.1 Buffer de luz

Esta técnica de aceleración esta basada en un concepto importante conocido como **cubo de dirección**, utilizado para *discretizar todas las direcciones posibles del escenario*, en un número finito de celdas de dirección.

El cubo de dirección tiene las siguientes características:

- Está alineado con el sistema de universal de referencia (SUR).
- Las caras del cubo están etiquetas, normalmente con +X, –X, +Y, –Y, +Z, –Z
- Las superficies de las 6 caras del cubo están divididas en **celdas de dirección**, definidas por las coordenadas (u, v), pertenecientes al

rango [–1, 1]. A la hora de crear las *celdas de dirección*, se pueden realizar divisiones uniformes o adaptativas.

- Cada celda de dirección tiene asociada una **pirámide de dirección**, con el vértice en el centro del cubo, y las aristas pasando por los vértices de las celdas.

Utilizando un cubo de dirección, Haines y Greenberg [HAIN86] introdujeron la técnica conocida como *buffer de luz*[3], para incrementar la velocidad de procesamiento de los rayos de sombra; se trata pues de un proceso de *propósito restringido.*

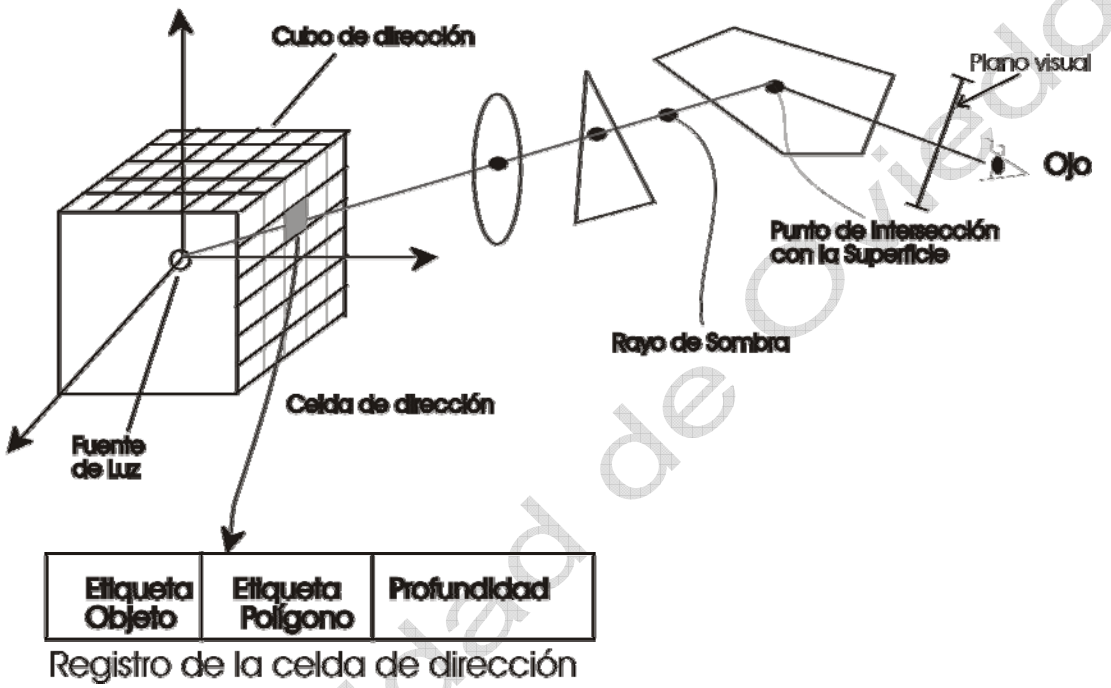

figura 17: el *buffer de luz* utiliza un *cubo de dirección*

La técnica del *buffer de luz* utiliza un cubo de dirección por cada fuente en el escenario, ubicando ésta en el origen del cubo. Las *celdas de dirección* son cuadros de igual tamaño (uniformes). Veamos con más detalle la construcción de la estructura.

1) Como acabamos de comentar, cada fuente de luz tiene asociado un *cubo de dirección*, con las caras divididas en *celdas de dirección* uniformes.

2) Cada *celda de dirección* tiene asociada una lista, con los objetos que "caen", total o parcialmente, dentro de la *pirámide de dirección*.

3) Las listas de candidatos se construyen proyectando cada objeto del escenario sobre las 6 caras del cubo de dirección, tomando como centro de proyección la fuente de luz. A la lista de una *celda de dirección* se añaden los objetos que cubren total o parcialmente la superficie de la celda.

4) Una vez que las listas han sido creadas, se clasifican los candidatos según la distancia a la fuente de luz, de menor a mayor.

---

[3] Para algunos autores hablar de *cubo de dirección* o de *buffer de luz* es lo mismo.

5) A continuación se simplifican las listas de candidatos. Los criterios de simplificación utilizados son:

- Si el modelo es poliédrico (que es lo más frecuente), todos los polígonos orientados en dirección opuesta a la fuente de luz, y que forman parte de un objeto opaco, pueden ser eliminados (culling).
- Si una lista posee sólo un polígono puede eliminarse, pues un solo polígono no puede darse sombra a sí mismo.
- Si la proyección de un candidato cubre por completo una celda de dirección, todos los candidatos posteriores a él (según la ordenación) pueden ser eliminados de la lista, finalizando ésta en un **registro de oclusión completa**.

A) Utilización del buffer de luz

Para averiguar si un punto se encuentra en sombra, se procede como sigue:

a) Se comprueba la orientación de la superficie con respecto a la fuente. Si mira hacia fuera, el punto está en sombra, referente a esa fuente.

b) En caso contrario, utilizando el vector director del rayo, en el cubo de dirección se busca la lista de objetos oclusores.

c) Se prueban los objetos de la lista, en orden creciente, hasta que se encuentra una oclusión, en cuyo caso el punto de intersección rayo-objeto está en sombra, respecto a la fuente; si no hay objetos oclusores, necesariamente se ha de encontrar en la lista el objeto que estamos considerando, y por tanto se encuentra iluminado por la fuente ubicada en el cubo de dirección.

Según el punto c), si se encuentra un *registro de oclusión completa* y el objeto considerado se encuentra a mayor profundidad, necesariamente ha de estar en sombra.

1.7 Control de profundidad adaptativo

Cambiamos de estrategia de aceleración, y la técnica que veremos a continuación pertenece al grupo de las que intentan acelerar ray tracing *trazando menos rayos*, sin descuidar la calidad de la imagen.

De entre todas las técnicas de aceleración, el **control de profundidad adaptativo** es la de menor coste, su implantación es sencilla, y se puede aplicar en todos los casos.

En un trazador simple es necesario fijar de antemano la profundidad máxima con la que los rayos serán trazados. El proceso recursivo finaliza cuando se alcanza esa profundidad.

Para un escenario particular, la profundidad máxima tendrá un valor que dependerá de la naturaleza de los objetos, de forma que si en el escenario aparecen superficies altamente reflectantes y objetos transparentes, la profundidad máxima será mayor que si aparecen solamente objetos opacos

con superficies poco reflectantes.

Hall y Greenberg [HALL83] puntualizaron que el porcentaje de escenas que contienen objetos reflexivos, es en general bastante pequeño, y hacen ineficaz el trazar cada rayo hasta su profundidad máxima. Dichos autores sugirieron el uso de un control adaptativo de la profundidad, que dependerá de las propiedades de los materiales con los que interactúa el rayo.

Un rayo puede ser atenuado de varias formas:

- Cuando un rayo se refleja en una superficie, es atenuado según el coeficiente de reflexión de dicha superficie.

- Cuando un rayo se refracta en una superficie, sufre una atenuación determinada por el coeficiente de transmisión de la superficie.

Un rayo, según va avanzando por el escenario, va siendo atenuado por varios de estos coeficientes. Si en un momento determinado, la contribución de ese rayo al píxel es menor que un cierto valor límite establecido de antemano, se termina su avance.

Un algoritmo propuesto por Watt [WATT89], mostrado abajo, implementa este control adaptativo de profundidad de forma recursiva. Cada vez que se activa el procedimiento, se comprueba si el coeficiente acumulado hasta ese momento (pasado como parámetro) es menor que un coeficiente límite. Si no es así, el coeficiente que resulta de esta nueva activación del procedimiento, se calcula multiplicando el coeficiente que tenía hasta ese momento, por el coeficiente de transmisión o de reflexión de la superficie que genera el nuevo rayo.

```
PROCEDURE Trazador_de_rayos (comienzo, direccion : vectors; profundidad :
integer; VAR color : colour; peso_acumulado: real);

 VAR punto_interseccion, direccion_reflejada,
       direccion_transmitida : vectors;  color_local, color_reflejado,
       color_transmitido : colour;

 BEGIN
   IF peso_acumulado < peso_umbral THEN
        color := negro
   ELSE
     IF profundidad > maxima_profundidad THEN
         color := negro
     ELSE
       BEGIN
       { Intersecar el rayo con todos los objetos y encontrar el punto
         de intersección (si lo hay) que se concluye en el comienzo del
         rayo }
       IF { no interseccion } THEN
           color := color_de_fondo
       ELSE
         BEGIN
              color_local := { contribución del modelo del color local al
                                punto_intersección }
             {Cálculo de la dirección del rayo reflejado}
             Trazador_de_rayos(punto_interseccion, punto_reflexion,
                                profundidad+1, color_reflejado,
                                peso_acumulativo*
                                peso_reflejado_superficie );
             { Cálculo de la dirección del rayo transmitido }

             Trazador_de_rayos(punto_interseccion, direccion_transmitida,
                                profundidad+1, color_transmitido,
                                peso_acumulado_transmitido-
                                peso_superficie );
              Combinar(color, color_local, peso_local_superficie,
                           color_reflejado, peso_transmitido_superficie,
                           color_transmitido, peso_transmitido_superficie )
         END
     END
 END  {Trazador_de_rayos};
```

En la figura 18 se puede apreciar cómo el rayo que está siendo trazado hacia atrás es atenuado después de varias intersecciones. Cualquier contribución a la intensidad es atenuada desde el "rayo 4", en el nivel superior del rayo, al realizar el producto de los coeficientes globales k_{t1} k_{t2} k_{t3} k_{t4}. Si este valor está por debajo del umbral establecido, no habrá trazado hacia atrás más allá del rayo 4.

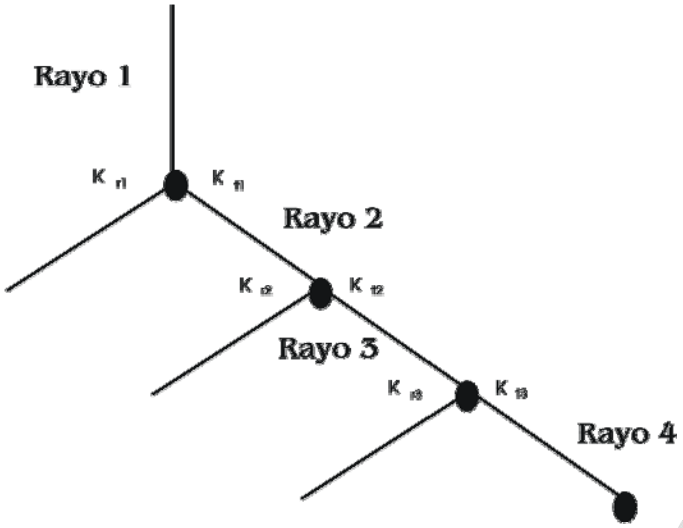

figura 18: atenuación del rayo

Aunque para simplificar se supone que en cada intersección hay un sólo coeficiente, en la realidad tendremos tres contribuciones (RGB) para cada rayo, y tres componentes por cada coeficiente de atenuación.

Hall y Gremberg, estudiando escenas con gran número de reflexiones, y empleando un árbol de profundidad 15, constataron que el método funciona con una profundidad media de 1,71, produciendo un considerable ahorro en el tiempo de generación de la imagen. Dicho ahorro dependerá de la naturaleza y distribución de los objetos en el escenario.

Sin embargo, es fácil diseñar casos en los que este método falla, ya que sobre un píxel puede haber gran cantidad de rayos que provoquen efectos imperceptibles, que individualmente son desestimados, pero la suma de estos efectos podría suponer una cantidad de luz significativa.

REFERENCIAS BIBLIOGRAFICAS

[AMAN84] Amanatides, J.,"Ray Tracing with Cones". *Computer Graphics*, Vol. 18, No. 3, Julio 1984, pp. 129-135.

[ARVO87] Arvo, J., D. Kirk, "Fast Ray Tracing by Ray Classification", *Computer Graphics*, Vol. 21, No. 4, Julio 1987, pp. 55-64.

[DADO85] Dadoun, N., D.G. Kirkpatrick, "The Geometry of Beam Tracing", *Proc. of the Symposium on Computational Geometry*, Junio 1985, pp. 55-61.

[FUJI86] Fujimoto, A., T. Tanaka, K. Iwata, "ARTS: Accelerated Ray-Tracing System", *IEEE Computer Graphics Applications*, Vol. 6, No. 4, Abril 1986, pp. 16-26.

[GLAS88] Glassner, A.S., "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics Applications*, Vol. 4, No. 10, Octubre 1984, pp. 15-22.

[GLAS89] Glassner, A.S., *An Introduction to Ray Tracing*, Academic Press, 1989.

[GOLD87] Goldsmith, J., J. Salmon, "Automatic creation of Object Hierarchies for Ray Tracing", *IEEE Computer Graphics Applications*, Vol. 7, No. 5, Mayo 1987, pp.14-20.

[HAIN86] Haines, E.A., D.P. Greenberg, "The Light Buffer: a Shadow Testing Accelerator", *IEEE Computer Graphics Applications*, Vol. 6, No. 9, Setiembre 1986, pp. 6-16.

[HALL83] Hall, R.A., D.P. Greenberg, "A Testbed for Realistic Image Synthesis", *IEEE Computer Graphics Applications*, Vol. 3, No. 10, Noviembre 1983, pp. 10-20.

[HECK84] Heckbert, P.S., P. Hanrahan, "Beam Tracing Polygonal Objects", *Computer Graphics*, Vol. 18, No. 3, Julio 1984, pp. 119-127.

[JANS86] Jansen, F.W., "Data Structures for Ray Tracing", *Data Structures for Raster graphics, Proceedings Workshop*, 1986, pp. 57-73.

[KAPL85] Kaplan, M.R., "Space Tracing a Constant Time Ray Tracer", *Siggraph '85 Course Notes*, Vol. 11, Julio 1985.

[KAY86] Kay, T.L., J. Kajiya, "Ray Tracing Complex Scenes", *Computer Graphics*, Vol. 20, No. 4, Agosto 1986, pp. 269-278.

[KIRK87] Kirk, D.B., "The Simulation of Natural Features using Cone Tracing", *The Visual Computer*, Vol. 3, No. 2, 1987, pp. 63-71.

[OHTA87] Ohta, M., M. Maekawa, "Ray Coherence Theorem and Constant Time Ray Tracing Algorithm", *Computer Graphics (Proc. of CG International '87)*, 1987, pp. 303-314.

[RUBI80] Rubin, S., T. Whitted, "A Three-Dimensional Representation for

Fast Rendering of Complex Scenes", *Computer Graphics*, Vol. 14, No. 3, Julio 1980, pp. 110-116.

[SHIN87] Shinya, M., T. Takahashi, S. Naito, "Principles and Applications of Pencil Tracing", *Computer Graphics*, Vol. 21, No. 4, Julio 1987, pp. 45-54.

[WATT89] Watt, A., *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley, 1989.

[WEGH84] Weghorst, H., G. Hooper, D. Greenberg, "Improved Computational Methods for Ray Tracing", ACM Trans. Graphics, Vol. 3, No. 1, Enero 1984, pp. 52-69.

BIBLIOGRAFIA RECOMENDADA

Se dan aquí una serie de libros y artículos que también tratan el tema de las Técnicas de aceleración de Ray tracing.

Para la técnica de Volúmenes Envolventes:

[BOUV85] Bouville, C., "Bounding Ellipsoids for Ray-Fractal Intersections", *Computer Graphics (Siggraph '85 Proceedings)*, Vol. 19, No. 3, Julio 1985, pp. 45-52.

Para las técnicas de Subdivisión Espacial:

[DEVI88] Devillers, O., C. Puech, F. Sillion, *Efficiency of Space Subdivision Structures for Ray Tracing*, Technical Report 88-2, Marzo 1988.

[DEVI89a] Devillers, O., "The Macro-Regions: an Efficient Space Subdivision Structure for Ray Tracing", *Eurographics '89*, Setiembre 1989, pp. 27-38, 541.

[DEVI89b] Devillers, O., "Tools to Study the Efficiency of Space Subdivision Structures for Ray Tracing", *Proc. of the PIXIM '89 Conference*, Setiembre 1989, pp. 467-481.

[GIGA90] Gigante, M., "Accelerated Ray Tracing using Non-Uniform Grids", *Proc. of Ausgraph '90*, Setiembre 1990, pp. 157-163.

[JEVA89] Jevans, D., B. Wyvill, "Adaptative Voxel Subdivision for Ray Tracing", *Proc. of Graphics Interface '89*, Junio 1989, pp. 164-172.

[JIHM87] Jih-Ming Lin, J., *Fast Ray Tracing on BSP-Trees*, MS Thesis, Dep. of Computer Science, Univ. of Central Florida, 1987.

[MACD90] MacDonald, J.D., K.S. Booth, "Heuristics for Ray Tracing using Space Subdivision", *The Visual Computer*, Vol. 6, No. 3, Junio 1990, pp. 153-166.

[MURA83] Murakami, K., H. Matsumoto, "Ray Tracing with Octree Data Structure", *Proc. 28th Information Processing Conference*, 1983.

[PENG87] Peng, Q., Y. Zhu, Y. Liang, "A fast Ray Tracing Algorithm using Space Indexing Techniques", Eurographics '87, Agosto 1987.