Trazado de rayos (parte 2)

Autor: Ramix (Ramiro A. Gómez)
Fecha: 4 de noviembre, 2007
Sitio web: www.peiper.com.ar

Esta es la segunda parte de trazado de rayos. Asumimos que si vas a leer este white paper ya leíste la primera parte, en la que te damos una pequeña pero concentrada introducción a este fascinante mundo del 3D. En este white paper vamos a ver como es el algoritmo en pseudo código, y vamos a explicar como se logran las técnicas más interesantes y que nos otorgan la mejor calidad a nuestras imágenes: el antialiasing, el desenfoque en movimiento y la profundidad de campo.
Como ya mencionamos en la parte 1, el algoritmo de trazado de rayos propiamente dicho es costoso en su ejecución, y tiene una eficiencia $O(n^2)$. En este punto podemos notar un gran problema: si queremos implementar videojuegos o realidades virtuales con este técnica vamos a necesitar una capacidad de procesamiento que pocas computadoras personales tienen hoy en día (fines del 2007). Ya calculamos que para una escena simple con un resolución nativa tenemos alrededor de 78 millones de rayos por cuadro (frame), y no consideramos rayos recursivos. Para lograr los tan preciados 24 cuadros por segundo nos van a hacer falta varias cosas, que las enumero en:

- un algoritmo optimizado (parte lógica)
- hardware potente
- implementación nativa en placas de video

Quizás el recurso que más nos beneficie es el de un algoritmo optimizado. Optimizar un algoritmo puede requerir varios pasos y un toque fundamental en nuestra manga, que es la creatividad. Las optimizaciones enunciadas en general para el trazado de rayos son de tipo espacial.
El algoritmo de trazado de rayos es muy elegante, dada su simpleza y recusividad.
¿Por qué es un algoritmo recursivo? Porque en su definición se autoinvoca una cantidad determinada de veces, lo que nos abstrae de la enorme complejidad de armar un árbol manualmente.
Supongamos que “tiramos” un rayo por pixel. Cuando ese rayo choque contra el objeto más cercano a la cámara, desde ese punto de choque “rebotarán” muchos más rayos, y cada uno de estos también rebotará. Si lo pensamos detenidamente, se forma una estructura de árbol, donde cada nodo es un rayo, y el nodo raíz es el primer rayo que tiramos por el pixel. Los hijos de cada nodo van a ser los rayos que se forman a partir del choque de este con un objeto.
Ahora pasemos al pseudo código:

```
trazarImagen () {
        si p > limite de profundidad {      retornar;      }
        para todos los pixeles {
                rayo = rayo por los puntos (Ox,Oy) y (pixX,pixY)
                prof = 0
                trazarRayo(rayo, prof)
        }
}
trazarRayo (Rayo r, profundidad p): Color {
        obj = objeto más cercano que intersecta el rayo
        rayos[] = rayos reflejados en obj
        c = (color(obj) + suma(colores_luces, intensidad_luces)) * peso(p)
        para todos los rayos[i] {
                c =  c + trazarImagen(rayo[i], p+1)*peso(p+1)
        }
}
```

La función peso calcula un factor de peso para el color con respecto a la profundidad del rayo. Cuanto más profundo sea el rayo, más pequeño va a

ser este factor. Es decir que peso(p) nos devuelve un valor numérico flotante entre 0.0 y 1.0, que constituye una ponderación para el color adquirido.
Como ven, el algoritmo es muy simpático y cuesta creer que un código tan escueto tenga la potencia que tiene. Obviamente que nos estamos olvidando de las funciones para describir los objetos, las texturas, las luces, etc. Pero la idea está en el algoritmo.

Objetos tridimensionales

Un detalle que no podemos olvidar mencionar en este artículo es el aspecto formal matemático que hay detrás de esto. Para representar los rayos, los objetos, los colores, los efectos, etc. necesitamos una herramienta que nos permita hacerlo usando números. Así, un rayo va a ser un conjunto de números, los objetos también van a ser un conjunto de números, y así podemos seguir. Lo importante del uso de las matemáticas en este sentido es que hay una rama de ellas que se ocupa de describir objetos geométricos: la GEOMETRIA ANALITICA.
Aunque el nombre nos asuste un poco, con su aplicación vamos a poder hacer todo lo que comentamos antes, hacer intersecciones, representar objetos tridimensionales, luces, humo, niebla, fuego, partículas, resplandores, etc.
Los objetos más comunes que se pueden encontrar en un paquete de raytracing son:

- las esferas
- los planos infinitos
- los triángulos
- los conos
- los cilindros
- las elipsoides

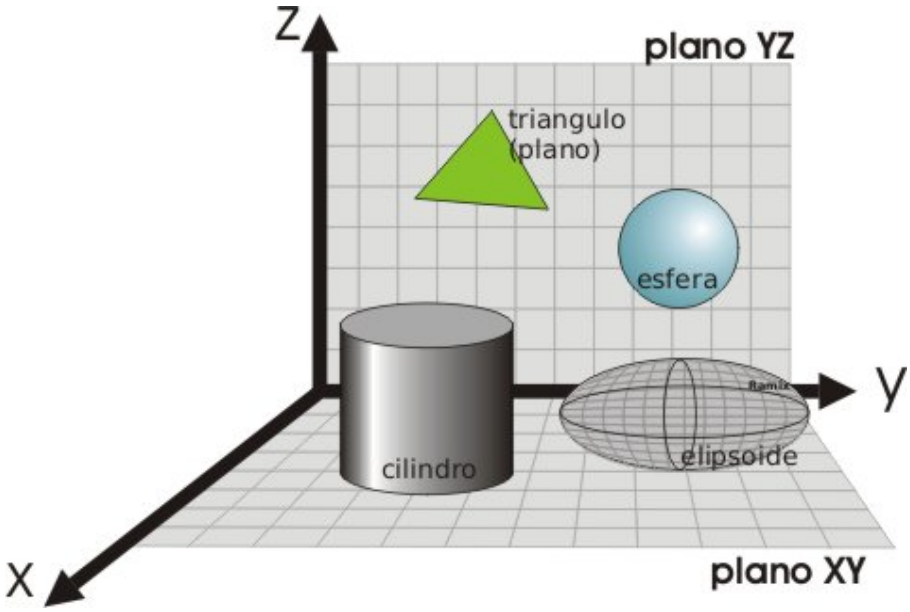
Cada una de estas figuras tiene una función que la representa. Por ejemplo, la ecuación de una esfera es:

$$(X-xc)^2+(Y-yc)^2+(Z-zc)^2=radio^2.$$

También parece complicada, pero créanme que en un tiempito se nos hace familiar y hasta le empezamos a poner nombres(¡¡??). Bueno, no es para tanto. Nadie quedó internado por programar un trazador de rayos en varios meses. Pero si lo hacemos en unos pocos días la historia ya es distinta. No se aceptan quejas de ningún tipo, nada de reclamos si tía elipsoide los persigue, eh!!
OK, volvamos a Tierra. X, Y, Z son variables, mientras que xc,yc, zc son el centro de nuestra esfera. radio lo dejo para que lo adivinen.
La ecuación de un plano es X:

$$Z = a*Y + b*X + c$$

Las letras en mayúscula son las variables, mientras que las minúsculas a, b, c, etc. son los números que nos van a determinar la inclinación y posición del plano.
No vamos a enunciar todas las ecuaciones, estas son algunas para dar una idea de la forma que tiene la descripción de cada figura 3D. Pasemos al rayo.
El rayo es un caso especial de geometría, ya que no presenta superficie ni volúmen, es unidimensional. Pero sin embargo, sí atraviesa un espacio tridimensional.



Podemos considerar que un rayo es un vector (matemático), con un punto de origen y otro punto de fin. La salvedad que tenemos que hacer es que el punto de fin en un rayo no determina realmente su fin, sino que nos determina el sentido y dirección del rayo. El sentido se refiere a si está dirigido hacia atrás o hacia adelante, la dirección especifica la pendiente de la recta que lo describe.
Para definir un rayo nos tenemos que valer de dos funciones de recta, una para el plano XY y otra para el plano YZ. Los dos juntas nos determinan un rayo que atraviesa un espacio 3D.
El rayo sería:

$$Y = m1*X + b1$$
$$Z = m2*Y + b2$$

Las letras m1 y m2 son las pendientes que tiene la recta. Por ejemplo, si m1 es 1, la recta en el plano XY va a estar inclinada 45°. Estas variables pueden ser positivas y negativas. En el caso de que m sea positiva, cuanto más grande es más rápido crece. En el hipotético caso de que m=infinito, la recta va a tener 90°, va a ser vertical. Este es un punto a destacar, porque si implementamos un trazador de rayos vamos a tener que considerar el caso especial de que la recta sea vertical.
Las letras b1 y b2 son el punto que va a tener cada recta cuando su variable vale 0. Dicho en un amistoso lenguaje criollo, el desplazamiento hacia arriba o hacia abajo que va a tener la recta (manteniendo la pendiente igual).

Las intersecciones

Para intersectar dos figuras geométricas vamos a tener que encontrar puntos X, Y, Z que satisfagan al mismo tiempo las ecuaciones de cada figura. ¿Cómo hacemos esto? Con un sistema de ecuaciones.

Sistema de ecuaciones para intersectar un rayo y una esfera

sistema de ecuaciones

$$\begin{cases} (X-xc)^2 + (Y-yc)^2 + (Z-zc)^2 = r^2 \\ Y = m1*X + b1 \\ Z = m2*Y + b2 \end{cases}$$

Resolver un sistema de ecuaciones consta de reemplazar en alguna de

estas las variables por su definición. Para que quede más claro, podemos tomar la fórmula de la esfera y reemplazar Y por la definición de Y (la segunda ecuación), y Z por la definición de Z. Como Z está en función de Y, la reemplazamos nuevamente. Con esto, la ecuación de la esfera nos va a quedar representada sólo en función de X. Despejamos esta variable y obtenemos 2 soluciones X, que son las coordenadas en el eje x de las 2 intersecciones que tiene el rayo con la esfera.
Además nos tenemos que asegurar que xc, yc, zc, radio, m1, m2, b1, b2 sean valores numéricos, no variables.
Aplicando esta solución y el algoritmo presentado arriba en pseudo código, ya tenemos bastante desarrollado nuestro trazador de rayos.

Sobre los triángulos

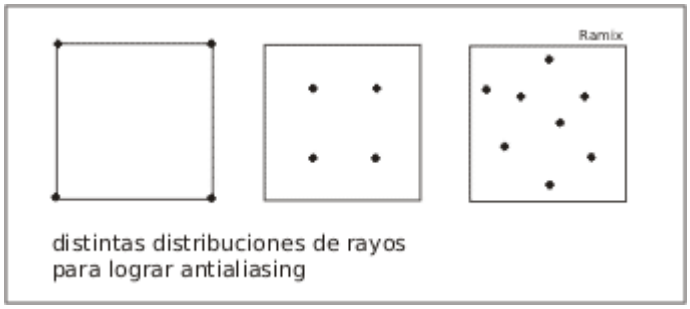
Como habrán notado en la mayoría de los juegos actuales, los objetos tridimensionales vienen poligonizados. ¿Qué quiere decir esto? Que se representan utilizando un montón de triángulos uno al lado del otro. Esto tiene sus ventajas, en particular si consideramos que la ecuación de un triángulo es la del plano con algunas restricciones, y resolver un sistema rayo-plano es más rápido que resolverlo con otra figura geométrica (con cuádricas, por ej.). Esta es una de las posibles razones por las que las placas de video vienen especialmente optimizadas para el procesamiento de triángulos, a tal punto que muchas veces escuchamos hablar de su potencia en base a la cantidad de millones de triángulos por segundo que procesa.

Efectos gráficos para trazado de rayos

Ya nombramos algunos de los más comúnes, el antialising, la profundidad de campo y el desenfoque en movimiento (motion blur). Hablemos primero del antialising.

Antialiasing

La idea es la misma que en los paquetes gráficos actuales más usados, como OpenGL y DirectX.
Cuando tenemos variaciones de intesidad de color muy bruscas entre pixeles adyacentes, se puede apreciar un efecto de “serrucho” cuando vemos la imágen. Esta característica no es para nada deseable, en especial si lo que deseamos es realismo gráfico.
Por eso, aplicando algunas técnicas conseguimos suavizar la imágen, de manera que entre los pixeles haya una especie de transición en degradado que mejore la calidad gráfica.
Una de las técnicas más usuales es el supermuestreo. Consiste en tomar un pixel como si fuera un área, y por cada subárea de éste calculamos su color. Como la mínima unidad que pueden representar los monitores y pantallas es un pixel, tenemos que promediar los colores de cada subárea del pixel. O sea, por cada color (rojo, verde y azul) los sumamos y luego dividimos la suma por la cantidad de subáreas del pixel.
En el trazado de rayos podemos implementar esta técnica de una manera harto sencilla: tirando más rayos por cada pixel. La calidad que obtengamos va a ser mejor, pero la performance va a caer estrepitosamente, 4 veces o más. Esto es porque al ser el pixel cuadrado, si ponemos 2 rayos por pixel, el total va a ser 2*2, con 3 rayos por lado sería 3*3, etc.



distintas distribuciones de rayos
para lograr antialiasing

Podemos apreciar en la imágen las distintas maneras de distribuir los rayos dentro del pixel. Una buena alternativa es hacerlo aleatorio, teniendo en cuenta una restricción. Como corremos el riesgo de que los rayos aleatorios nos queden todos juntos en una pequeña parte del pixel, podemos dividir el pixel en subáreas, y dentro de cada una tirar un rayo aleatorio que tome como límites para su posición los extremos de ésta. Si no usamos la distribución aleatoria, yo recomiendo usar la segunda presentada en el diagrama porque si usamos la primera vamos a tener algunas complicaciones a nivel de programación cuando pasemos al pixel horizontal contiguo y también cuando pasemos al pixel vertical contiguo.

Profundidad de campo

Las cámaras fotográficas tienen la opción para regular la nitidez con la que va a salir la fotografía. En cualquier fotografía que se saque a un objeto cercano se puede apreciar un fondo borroso o difuso, que no nos permite apreciar sus detalles. Esto es la profundidad de campo, que se ajusta de acuerdo a la cercanía del objeto al plano focal de nuestra cámara.

En la imágen podemos ver una demostración del efecto logrado con esta técnica aplicada al trazado de rayos. La imágen es propiedad de su correspondiente autor y sólo la exponemos a modo de ejemplo, sin ánimo de lucro.

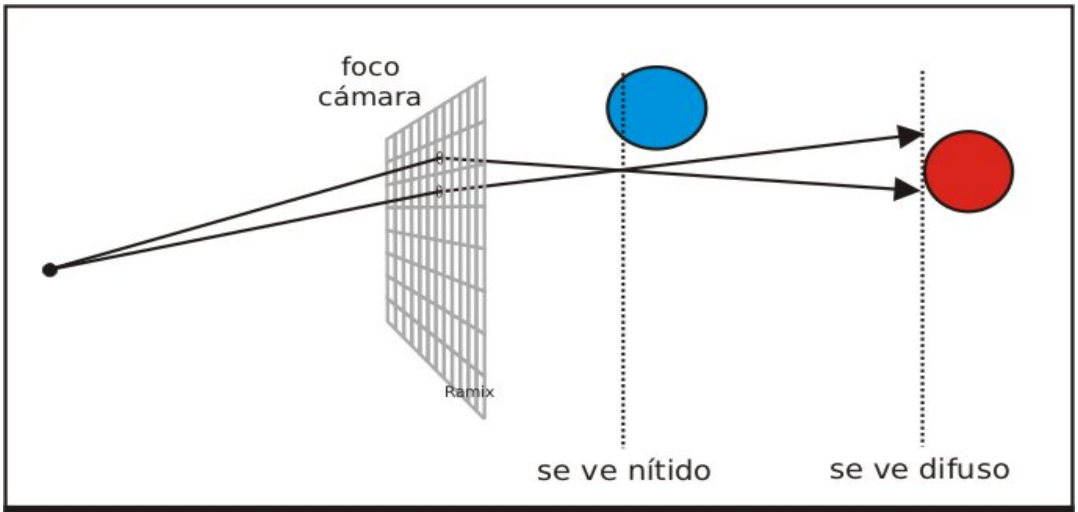
El realismo gráfico casi no tiene precedentes. Vemos que es una imágen con antialiasing, con profundidad de campo, con objetos traslúcidos que refractan la luz de la escena.

La imágen fue trazada con Yafray, usando el excelente modelador Blender (además gratuito).

Si miran bien, las sombras de las piezas en la caja son muy reales, y el "efecto de lupa" de las piezas que se consigue usando refracción nos termina de convencer de que la escena "existe" y es real.

La profundidad de campo se consigue "doblando" los rayos cuando salen por el foco de la cámara. Desde el origen hasta el foco tienen la misma pendiente que siempre, pero cuando llegan al foco se "doblan", de manera que dos rayos contiguos se intersectan en el punto donde está el plano de visión (donde la nitidez es máxima). Mirando el esquema no hace falta siquiera leer la explicación.

Entonces, los colores de dos pixeles contiguos cuyos rayos intersecten objetos más allá del plano focal van a ser distintos que los que serían si el rayo no se quebraría.



Profundidad de campo en trazado de rayos.

Desenfoque en movimiento

Vamos a explicar brevemente como se logra este efecto, que nos va a dar la sensación de velocidad, de que los objetos se mueven rápido. En las cámaras fotográficas tradicionales esto se puede lograr aumentando el tiempo de exposición. Si el diafragma se abre y cierra demasiado rápido, vamos a ver los objetos totalmente estáticos, aunque se muevan con velocidad. Pero si lo mantenemos abierto medio segundo o más, la luz que va a entrar en la cámara va a ser la de cada instante de tiempo, o sea que se va a agregar una imágen a la otra durante este intervalo.

Esta idea es la que intenta imitar el efecto de desenfoque en movimiento (motion blur). Para empezar, tenemos un intervalo de tiempo, que es el tiempo durante el que se va a capturar cada cuadro (en el caso de video) o la imágen. Por cada instante se tiran rayos, luego se procesa un cuadro más haciendo que los objetos se muevan, se tiran más rayos, y así sucecivamente, hasta completar el intervalo.

Ahora tenemos por cada rayo principal (por ej. por cada pixel) un montón de colores, que son los que se obtuvieron de trazar la escena durante el intervalo de tiempo. Ahora podemos hacer varias cosas:

- sumar los colores
- promediar
- hacer un promedio ponderado

En el caso de la suma de colores, sería lo más lógico y real hacer esto, pero vean que la imágen se va a sobreexponer muy rápido y como resultado podemos obtener una imágen totalmente blanca, producto de la cantidad de luz que entró en la cámara.

Si queremos evitar esto, podemos promediar los colores. Hacemos su suma y después dividimos el valor numérico del color por la cantidad de cuadros capturados.

Otro enfoque es el de hacer un promedio ponderado. El promedio ponderado es la misma idea que el promedio, pero le dá más importancia a algunos cuadros que a otros. En otras palabras, los colores de algunos cuadros van a influir más en la imágen final que los colores de otros. Para esto tenemos un conjunto de números decimales que van de 0.0 a 1.0, tal que la suma total de ellos sea = 1. Cuanto más chico es el número, menos importancia se le dá al color por el que está multiplicado.

Para dar un ejemplo, tenemos 5 cuadros que se tomaron en 2.5 segundos. Un cuadro cada medio segundo. Los colores en escala de grises del rayo R1 en los distintos tiempos son 0, 20, 50, 100, 50.

Ahora tenemos los pesos, que son 0.1, 0.15, 0.25, 0.35, 0.15. El cuadro que va a tener más importancia en el cálculo del color va a ser el 4to, porque su factor es 0.35 o 35%.

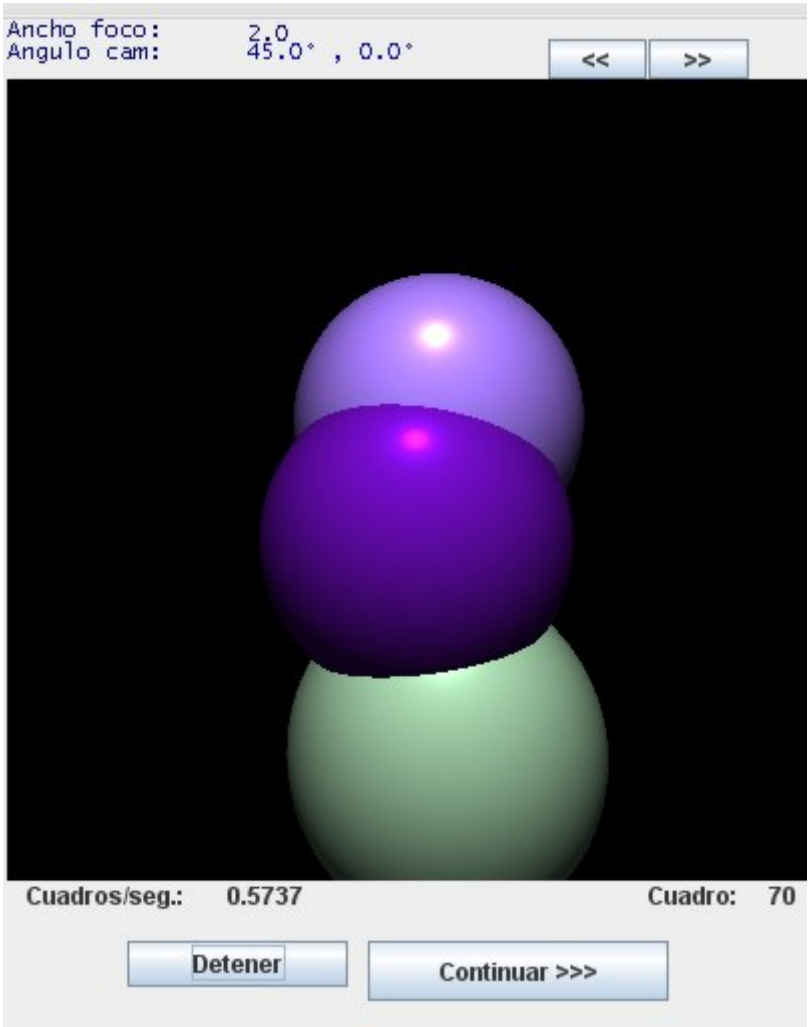
El color final nos queda:

C = 0*0.1 + 20*0.15 + 50*0.25 + 100*0.35 + 50*0.15

Si lo queremos hacer para una paleta RGB (rojo, verde y azul), multiplicamos la intensidad de cada color por el peso. Aclaramos que los colores en RGB (red,green,blue) se componen de tres números que van de 0 a 255, el primero representa la cantidad de rojo, el segundo la cantidad de verde, y el tercero la de azul.

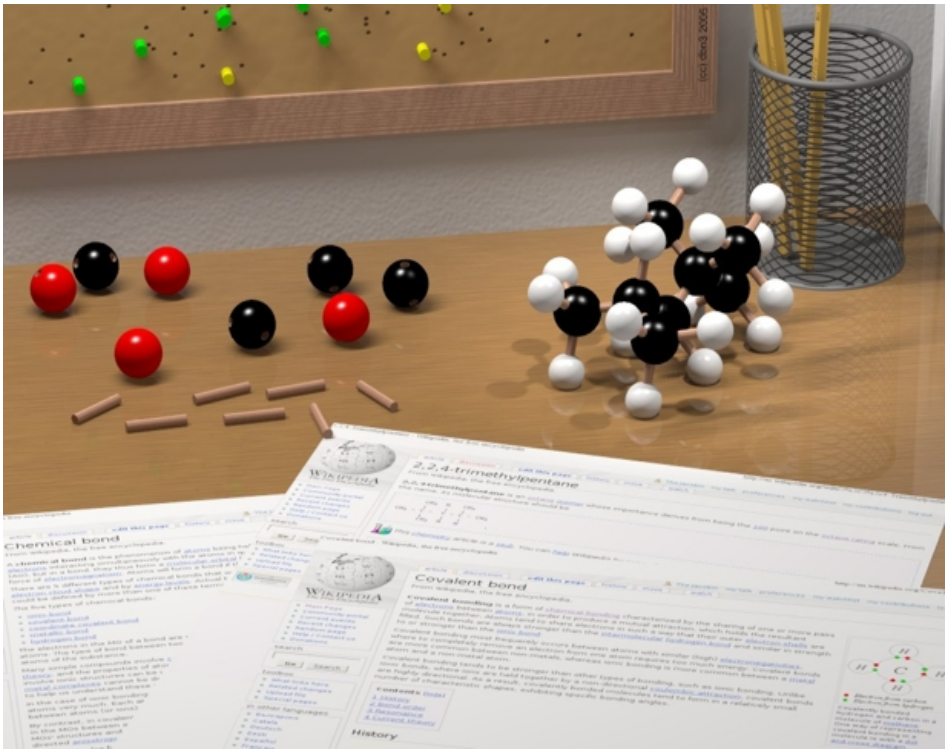
Por hoy no nos queda más que despedirnos, pero prometemos que la próxima entrega -parte 3- vamos a comentar las técnicas que se usan para optimizar los cálculos de las intersecciones de los rayos, ya que son las operaciones que más tiempo y procesador consumen.
Entre lás técnicas más comúnes podemos encontrar la subdivisión espacial y la agrupación con objetos de contorno. Usando alguna de ellas nuestro trazador de rayos puede empezar a tener una eficiencia lineal, cuasi – lineal y hasta logaritmica O(log(n)).
Como ejemplo les dejo una imágen que obtuve de mi propio trazador de rayos. Mis planes son usar trazado de rayos en tiempo real. Logré una eficiencia logaritmica usando la técnica de “agrupación con objetos de contorno” y una propia que yo llamo “aproximación PV” (que NADA tiene que ver con las siglas PV que se pueden encontrar en algunos papers de IEEE; son más bien las siglas de la estructura que uso).
Para tener una idea, con 5 esferas se calcula 1.5 cuadros/segundo con definición mínima. Con 1000 esferas toma 0.3 cuadros/segundo con la misma definición. Exquisitamente logarítmico.

Autor: Ramix (Ramiro Andrés Gómez)



Ejemplo 1. Fuente: www.blender.org

Ejemplo 2. Fuente: www.blender.org



Referencias:

- Gráficos por computadora con OpenGL (Donald Hearn & Pauline Baker). PEARSON – Prentice Hall. ISBN-10: 84-205-3980-5. ISBN-13: 978- 84-205-3980-5
- Wikipedia (www.wikipedia.com)
- Diversos tutoriales de la web

Recursos gráficos

Los esquemas son diseños propios.
Las imágenes fueron tomadas de www.blender.org. Se presentan sólo para fines demostrativos y pertenecen a sus respectivos autores. Prohibida su copia y/o utilización comercial sin autorización del autor.