DEPARTAMENT D'INFORMÀTICA
UNIVERSITAT JAUME I

E80
PROYECTOS INFORMÁTICOS

INGENIERÍA INFORMÁTICA
Curso 2002--2003

Memoria Técnica del Proyecto

Métodos de iluminación global.

Proyecto presentado por el Alumno

Sergio Sancho Chust

Dirigido por ***Miguel Chover Sellés***

Castellón, a 15 de Septiembre de 2003

Resumen

La utilización de programas informáticos capaces de sintetizar imágenes realistas ha sufrido un gran aumento en la última década. Tal es así que en muchos ámbitos donde no se utilizaba ahora se utiliza y en otros donde ya se utilizaba lo hace con más importancia. Esta síntesis de imágenes realista podemos observarla en los efectos visuales de las últimas películas de cine o en videojuegos de todos los tipos y en todas las plataformas.

La búsqueda de métodos que resultaran fáciles de manejar y correctos en su representación del render realista ha pasado por multitud de estados como son el scanline o el ray tracing. De entre los más sencillos, potentes y flexibles se desmarcó al instante el método del ray tracing.

Paralelamente, se ha ido mejorando en la simulación informática de la iluminación real hasta llegar a métodos de iluminación avanzados como la radiosidad o el photon mapping. Estos métodos nos definirán cuál será la iluminación indirecta en toda la escena. Iluminación indirecta que por otro lado no podríamos calcular con la misma efectividad ni rapidez con la que lo hacen estos métodos.

Por tanto, el presente documento hace referencia al estudio, análisis, diseño e implementación de un trazador de rayos basado en ray tracing y con utilización de iluminación global. Este trazador de rayos será lo más flexible posible a fin de poderse utilizar en cualquier otra aplicación. Con el fin de demostrarlo se desarrollaran también junto con el trazador de rayos dos aplicaciones que lo utilizan: una aplicación independiente que interprete la escena mediante un fichero xml y dos plugins de materiales y render del 3d Studio Max 5.

Con el contenido de la presente memoria abarcaremos todo lo necesario tanto para comprender la teoría utilizada como para desarrollar el proyecto enteramente.

Palabras clave

Trazador de rayos, ray tracing, rayo, radiosidad, photon mapping, iluminación global, CSG, material, Monte Carlo, KdTree, síntesis de imágenes realista, plugin, 3d Studio Max.

Índice

Capítulo 1: Introducción

Hace ya varios años que se comenta en todas partes la afirmación de que algo computerizado se asemeja o equivale a lo real. Y no sólo en términos matemáticos de parametrización de sucesos científicos sino también en otros ámbitos mucho más ociosos como el cine o los videojuegos. No cabe duda que este afán creador de realidad ya viene de lejos, de otras épocas donde la literatura, la escultura y la pintura quisieron plasmar o crear obras que parecían reales. En la época actual existen muchas herramientas para seguir en esta línea y sin duda alguna la informática ha presentado al hombre el nuevo pincel para plasmar aquello que sentimos real y verdadero.

1.1 Estructura del documento

Con el fin de cubrir el proyecto de la asignatura Proyectos Informáticos E80 de la universidad Jaume I y de saciar la necesidad que para el autor es toda esta investigación, nace pues este documento durante el curso 2002/2003. El cual se dedicará a describir una investigación realizada sobre los métodos de síntesis de imágenes por computador con métodos de iluminación global. Paso a paso se describirá con la mayor profundidad y sencillez de la que el autor sea posible todo lo relacionado con el tema. Todo como base de la descripción del proyecto software que se acompaña con la memoria: un trazador de rayos basado en la programación orientada a objetos y que implementa todo aquello que se describe en el presente documento.

En éste primer capitulo, numerado como 1, nos dedicaremos a describir por tanto cual es el estado del arte o ambiente que rodea ahora mismo al tema de los métodos de generación realista de imágenes y de la informática dedicada a tal efecto. Del mismo modo dejaré en el apartado sobre los objetivos del proyecto bien claro los objetivos que se persiguen con la realización del proyecto.

A partir del capítulo 2 y hasta el 6, ambos inclusive, pasaré ya a definir la base de conocimientos previos y necesarios para la realización del proyecto. Explicando en el capítulo 2 los tipos de modelado de objetos utilizados. Pasando luego a explicar qué tipo de material se utilizará para ellos en el capítulo 3. El capítulo 4 se dedicará a los métodos de iluminación directa, necesarios para la creación de la parte de *ray tracing* del proyecto, que se explicará en el capítulo 5. Por último, el capítulo 6 nos mostrará la base teórica y práctica para la inclusión de la iluminación global en nuestro proyecto. Nótese que dado que he creído conveniente explicarlos con detenimiento, la proporción de los capítulos en referencia a la memoria total tiene un gran peso.

Con ello habremos terminado con los conocimientos que hay que tener para por fin implementar lo que en el capítulo 7 se describirá. Este capítulo se dedicará a la descripción del proyecto en si.

En el capítulo 8 se verán las conclusiones que se extraen del estudio completo del proyecto, tanto teóricamente al describir la base teórica como prácticamente al implementar el proyecto en si.

El capítulo 9 y último, estará dedicado a un pequeño estudio sobre que posibles extensiones se podrían realizar sobre el proyecto realizado hasta la fecha de presentación de la presente memoria.

Al final de la memoria también se presenta una lista de referencias utilizadas en la creación del presente documento a fin de que puedan servir de ayuda para entender el tema con mayor profundidad.

1.2 La informática

Desde el principio de nuestras culturas, se busco gracias a la filosofía la explicación del funcionamiento de la razón humana. Así se proclamaron diferentes leyes del razonamiento que luego se matematizaron en el siglo XIX bajo nombres de distinguidos matemáticos como Boole o Morgan. Más tarde, fueron estas leyes sobre la razón las que iniciaron en el campo tecnológico del siglo XX lo que ahora llamamos como informática. Y no cabe duda que esto se deba también a multitud de otros nombres ilustres como Turing, Gödel, IBM, Eniac, Fortran, etc...

Sin embargo no fue antes de la década de los 80 cuando la informática se hizo de dominio público a través de computadoras personales capaces de aliviarnos de trabajos diversos, normalmente repetitivos o tediosos.

Ahora en cambio, los ordenadores personales han dado un vuelco a nuestra vida. ¿Quién definiría lo que es ahora mismo la humanidad sin asegurar que la informática es parte principal de todo el ambiente que nos rodea?. Es cierto, ahora los ordenadores son necesarios, y no sólo eso, sino que han cambiado nuestra forma de ver. Ahora lo que vemos en nuestros libros es fruto de la informática, lo que vemos en las propagandas también, o lo que vemos en el cine o la televisión. Mi proyecto se basa también en la informática y más específicamente en la síntesis de imágenes por ordenador. La misma síntesis de imágenes que es la que nos sirve ahora para plasmar la realidad.

1.3 Síntesis de imágenes

Los computadores por tanto han demostrado en su corta vida ser una importante herramienta de realización de imágenes sintéticas. Tanto para usos científicos, médicos, etc... o para usos artísticos.

Seguramente se pueden nombrar un sinfín de programas capaces de realizar imágenes a partir de una descripción más o menos ajustada a la realidad y obtener así representaciones más o menos realistas. Estos programas pueden o no ir en tiempo real como así lo hacen librerías gráficas aceleradas como OpenGL, DirectX o anteriormente Glide. Sin embargo los programas que al final consiguen de momento un mejor realismo a costa de un gran coste temporal son programas que utilizan métodos de ray tracing, scanline o iluminación global. Quizá todos conocemos programas profesionales como 3d Studio Max, maya o programas de código abierto como PovRay, además de muchos plugins hechos para ellos y que producen mayor realismo como pueden ser DirtyReyes, MentalRay o últimamente, Arnold.

Se puede asegurar que la síntesis fotorrealista de imágenes está ahora mismo muy adelantada y que con este método se consiguen imágenes casi idénticas a lo real. Y es entonces cuando en el presente nace otra dirección totalmente opuesta a la síntesis fotorrealista. Todo nace de la pregunta que se puede realizar cualquiera: ¿para qué hacer ese rendering cuando uno puede hacer una foto a la realidad y así ahorrarse todo el tiempo de cálculo necesario?. Es así como ha surgido en estos últimos tiempos la necesidad de hacer renderings que aumenten la parte artística de la imagen aunque reduzcan así dicho fotorrealismo. Así una vez alcanzado el fin que se propuso hace tiempo sobre la síntesis fotorrealista de imágenes, el conseguirlo ha llevado a otros fines como han sido el que se generen imágenes que parezcan dibujadas a lápiz o pintadas a óleo o acuarela. Esto es llamado rendering no fotorrealístico.

De todos modos hasta ahora se han realizado grandes avances tanto por una u otra vertiente y es dudoso que el rendering no fotorrealista haga inútil al rendering fotorrea-

lista. Simplemente los dos serán herramientas informáticas para el hombre del siglo XXI y quién sabe cuáles serán sus límites y hasta dónde se llegará en la síntesis de imágenes por computador.

Ilustración 1 - Render fotorrealista.

Ilustración 2 - Render no fotorrealista.

1.4 Objetivos del proyecto

Una vez explicada toda la base necesaria para la realización del proyecto, nuestro objetivo será pues realizar partiendo de esta, un trazador de rayos con tratamiento de la iluminación global. Este trazador de rayos manejará todos los tipos de objetos propuestos en el capítulo 2, y además estos poseerán un material con los mismos parámetros y posibilidades del material definido en el capítulo 3. El trazador de rayos estará basado en la teoría de los trazadores de rayos explicada en el capítulo 5 y manejará tanto la iluminación directa vista en el capítulo 4 como la iluminación indirecta vista en el capítulo 6.

El objetivo de este proyecto será pues el análisis, diseño e implementación de ese trazador de rayos fotorrealístico, al que llamaré *PiscisRT* (Piscis Ray Tracer). El trazador estará implementado como una librería dinámica y contendrá solamente como base una función que lanza rayos y devuelve los colores que ven esos rayos. Esto se hace así para darle al trazador la mayor utilidad posible siendo posible utilizarlo en cualquier programa de una manera muy sencilla. Por eso además del trazador *PiscisRT* implementaré una aplicación independiente que utilice el trazador para renderizar imágenes llamada *elements* y un plugin de render para el 3d Studio Max que también lo utilice llamado *PRTMaxPlugin*. Nótese que para que el plugin *PRTMaxPlugin* funcionara optimamente, se ha creado otro plugin *PRTMaxMaterial* que gestionara materiales del tipo del trazador dentro del 3d Studio Max.

El funcionamiento del trazador *PiscisRT* quiero que sea lo más sencillo posible, por eso para utilizarlo valdrá primero con importarlo en el proyecto. Después se creará un objeto de la clase principal del trazador y será a partir de este objeto desde donde se manejará todo el tratamiento de los rayos. Desde el objeto de la clase se añadirán objetos y luces a la escena, se activarán o desactivarán opciones del trazador, y por fin, mediante una función que lance un rayo y devuelva el color que este rayo ve, obtendremos el fruto final del trazador.

El funcionamiento de la aplicación *elements* consistirá en a partir de un fichero de entrada con la descripción de la escena a renderizar, parsearlo añadiendo todos los componentes necesarios a la escena, activar o desactivar las opciones que se quieran, y por fin crear la imagen que se vería desde la posición de la cámara, con la dirección de la misma, la amplitud del fov y los tamaños requeridos para el render.

Por su parte los plugins para 3d Studio Max *PRTMaxPlugin* y *PRTMaxMaterial* servirán para incluir el PiscisRT dentro de un entorno de diseño 3d. De tal manera que nos sea fácil tratar todo tipo de geometría, visualizarla en tiempo real, y por fin renderizarla con el PiscisRT.

Con la creación de la librería *PiscisRT*, la aplicación *elements* y los plugins para 3d Studio Max *PRTMaxPlugin* y *PRTMaxMaterial*, el presente proyecto se dará por terminado.

1.5 Entorno de desarrollo

El trazador de rayos *PiscisRT* en forma de librería dinámica será implementada tanto para plataforma Windows como para cualquier plataforma basada en GCC: Linux, Unix, etc. El trazador además necesitará para compilar la librería de tratamiento de texturas jpg, jpeg lib versión 6b. Para Windows se dispondrá de un proyecto tipo visual studio 6 y para linux de un archivo tipo Makefile.

En cuanto a la aplicación *elements*, está también será multiplataforma y se basará en las librerías *PiscisRT*, *xmlparse* y *glut32*. Esto es porque como hemos visto la aplicación funcionará parseando un archivo de entrada de tipo *.prt*, y gracias al *PiscisRT* y a la *glut32* mostraremos por pantalla aquel render que se demanda mediante la especificación del archivo de entrada.

La única aplicación que sólo funcionará en Windows será el plugin para 3d Studio Max *PRTMaxPlugin* el cual dependerá del 3d Studio Max SDK, del trazador *PiscisRT* y del plugin *PRTMaxMaterial* para tratar materiales nativos del PiscisRT. Este plugin renderizará la escena que creemos en 3d Studio Max, pero utilizando el *PiscisRT* en lugar del motor de render propio del programa.

Todos los códigos fuente que genere estarán documentados mediante la especificación del Doxygen para poder generar luego la documentación automática del mismo.

Además, para la correcta optimización del código generado, utilizaré el programa Compuware Devpartner Studio que nos servirá para predecir fallos en el código y para hacer un estudio sobre las partes con más coste de cómputo. Gracias a esto conseguiremos un proyecto altamente optimizado.

Capítulo 2: Modelando los objetos

El computador generará una imagen realista o no pero habrá de hacerlo en base a alguna representación de la realidad enteramente informática. Veamos entonces cuales pueden ser estas representaciones de la realidad que se utilizan actualmente en todos los programas de síntesis de imágenes.

2.1 Representación basada en polígonos

Posiblemente la representación en 3 dimensiones más extendida sea la representación poligonal de objetos. Esta representación consiste en convertir objetos que en realidad son infinitamente más complejos en otros hechos a base de un número a voluntad de polígonos simples. Cuanto más grande sea ese número de polígonos más cercana estará la representación poligonal a la realidad.

Así por ejemplo la conversión de una forma cerrada cilíndrica en forma de polígonos se realizaría dividiendo el cilindro en partes independientes y luego éstas convertirlas en polígonos de forma que al colocarlos juntos formen una representación correcta del cilindro original.

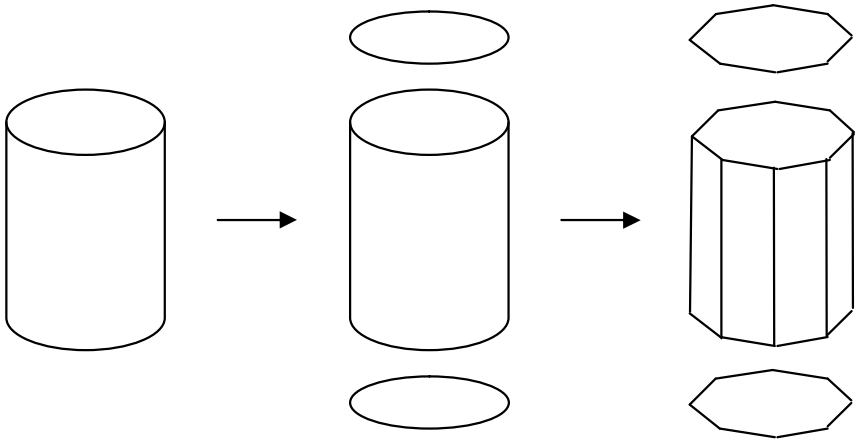

Ilustración 3 - Representación poligonal de un cilindro.

Nos asalta ahora la duda de cómo guardar estos polígonos de forma eficiente y con toda la información del objeto necesaria.

Primero de todo hay que definir qué cara del polígono es requerida ya que un polígono en 3 dimensiones forma a su vez dos polígonos opuestos o lo que es lo mismo, dos caras. Esta elección se podrá realizar por lo que se define como normal del polígono, que no es ni más ni menos que un vector unitario perpendicular a la superficie y apuntando en la dirección de la cara visible. Esto se podrá perfeccionar si la normal de la superficie es interpolada entre las normales de sus vértices. Además hay que tener en cuenta que en muchos casos la normal viene definida en el sentido horario de definición de vértices del polígono.

En la ilustración 4 se puede observar como sería la orientación de un polígono P_0 con cuatro vértices V_0, V_1, V_2 y V_3, cada uno de ellos con su propia normal. De este modo vemos como la normal del polígono coincidiría con la interpolación de las normales de sus vértices y con el sentido horario de sus vértices.

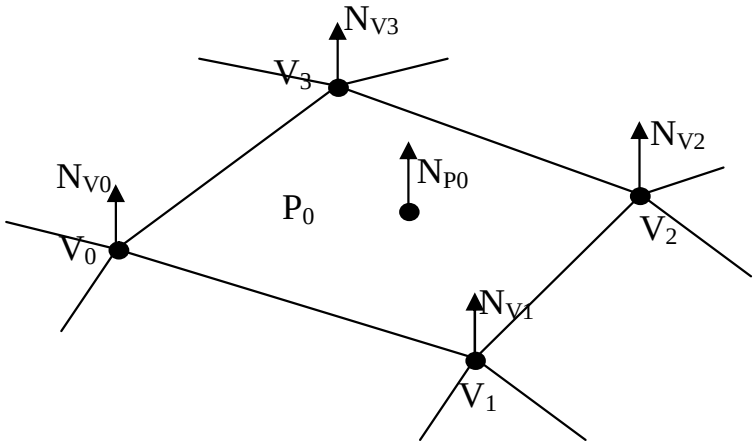

Ilustración 4 - Orientación un polígono en 3d.

Hemos comentado que los polígonos deberán estar definidos mediante sus vértices y que estos poseerán normal propia. Ahora hemos de discernir sobre cómo guardar esta información de la forma más óptima, para ello se propone que un polígono se defina como una lista de posiciones sobre una lista de vértices, así los vértices serán almacenados en una lista independientemente de que polígono los utilice y los polígonos se referirán a esos vértices como posiciones dentro de la lista de vértices.

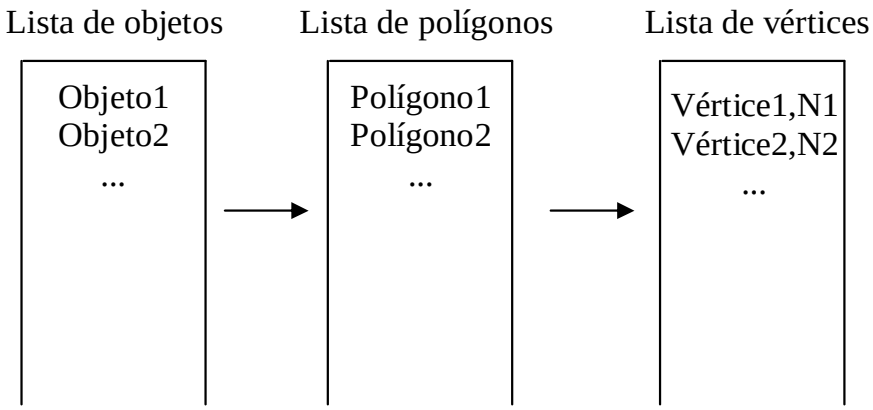

Ilustración 5 - Estructura jerárquica de la representación poligonal.

Podemos darnos cuanta enseguida que ya que los polígonos son tratados como independientes no estamos tratando situaciones donde dos aristas iguales entre dos vértices iguales pero en distintos polígonos no son tratadas como la misma sino que se crean dos aristas iguales. Para solucionar este punto se define una segunda representación de objetos en 3 dimensionas llamada representación basada en aristas.

2.2 Representación basada en aristas

Una alternativa a la anterior y que añade ciertas ventajas es pues la representación basada en aristas, descrita en [1]. Esta representación utiliza otra lista intermedia que será la lista de aristas. De este modo un objeto tendrá índices a polígonos, que a su vez tendrán índices a aristas, que a su vez tendrán índices a vértices. Veamos esta jerarquía de forma gráfica:

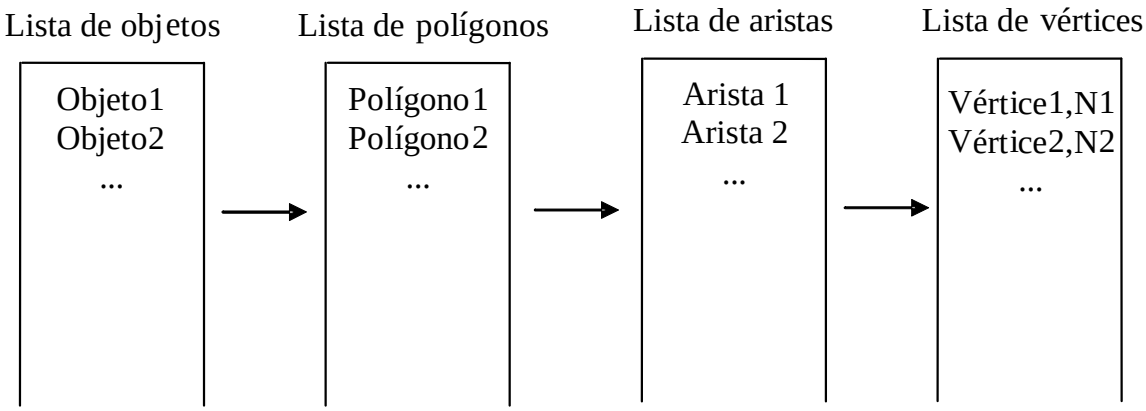

Ilustración 6 - Estructura jerárquica de la representación basada en aristas.

Así se solucionan situaciones donde se comparten aristas aunque también añaden otras complicaciones dependientes del modelo de render ya que podríamos decir que este modelo implica un modelo de render de scanline. De todas formas consideremos esta representación como una posibilidad de jerarquía de objetos 3d poligonales muy optimizada para ambientes donde se requiera poco espacio de almacenamiento o el uso de un método scanline.

2.3 Representación de objetos paramétricos

Frente a la representación poligonal nos encontramos con la posibilidad de definir paramétricamente cada objeto según su forma. Mientras que antes cualquier objeto 3d se reducía a un número grande o pequeño de polígonos, ahora cada objeto 3d se definirá según su forma por un número conocido de parámetros.

Así, para una esfera es mucho más sencillo guardar su centro (p) y radio (r) que un número grande de polígonos, y al final es la representación paramétrica la que no añade ningún error en la similitud con el objeto inicial.

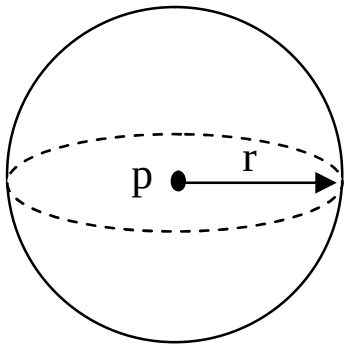

Ilustración 7 - Esfera paramétrica.

Visto cómo representaríamos la esfera podemos pasar a definir como representaremos paramétricamente otras formas básicas. El plano se representará como un punto que reside en él (p) y una normal que define su orientación (n). Un polígono será así un plano limitado por sus vértices (V0...VN). Y también así un círculo se podrá definir como un plano limitado por el radio el centro del círculo.

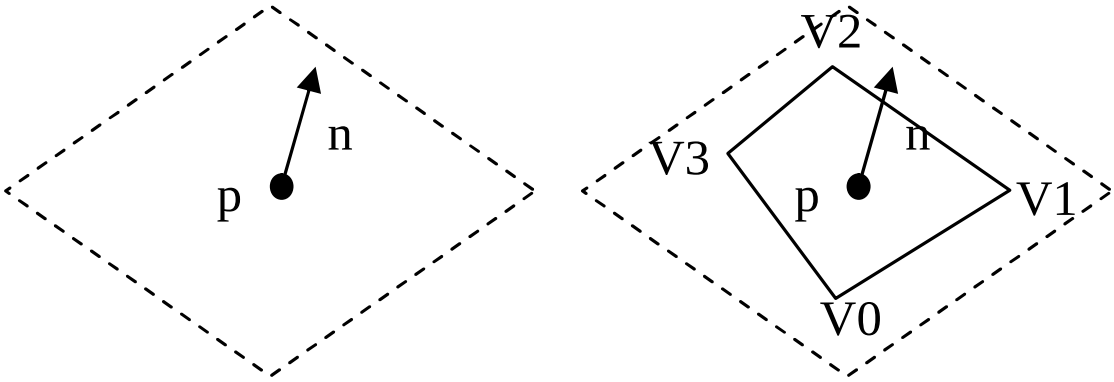

Ilustración 8 - Plano y polígono paramétrico.

Para otras formas como cilindros o conos habremos de primero estudiar lo que son las cuadricas y para formas más complejas como toroides tendremos que estudiar las cuarticas.

2.3.1 Cuádricas

Las cuádricas se definen como aquellas formas que se obtienen de la ecuación implícita:

$$Ax^2 + Ey^2 + Hz^2 + 2Bxy + 2Fyz + 2Cxz + 2Dx + 2Gy + 2Iz + J = 0 \qquad (2.1)$$

que es equivalente en forma matricial a:

$$\begin{vmatrix} x & y & z & 1 \end{vmatrix} \begin{vmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} = 0 \qquad (2.2)$$

Esta será la ecuación original de otras como:

Esfera: $$(x-l)^2 + (y-m)^2 + (z-n)^2 = r^2 \qquad (2.3)$$

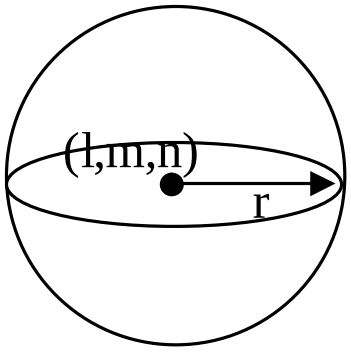

Ilustración 9 - Cuádrica esfera.

Cilindro infinito: $(x-l)^2+(y-m)^2=r^2$ (2.4)

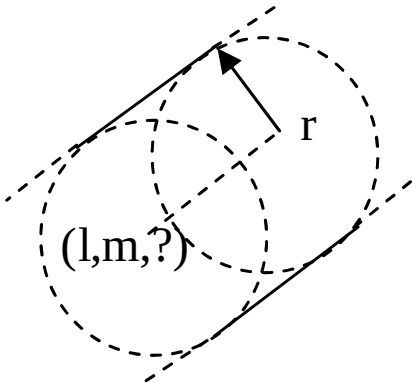

Ilustración 10 - Cuádrica cilindro infinito.

Cono infinito: $(x-l)^2+(y-m)^2-(z-n)^2=0$ (2.5)

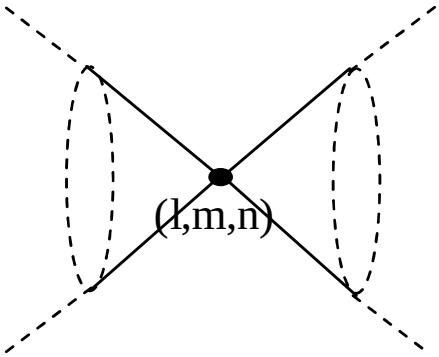

Ilustración 11 - Cuádrica cono infinito.

Elipsoide: $\frac{(x-l)^2}{a^2}+\frac{(y-m)^2}{b^2}+\frac{(z-n)^2}{t^2}-1=0$ (2.6)

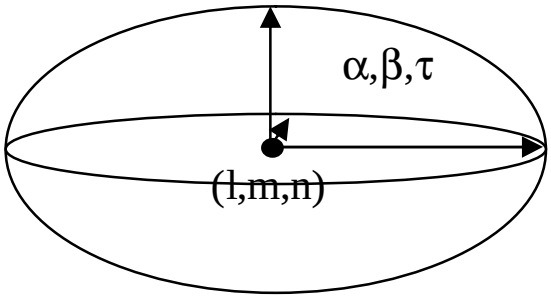

Ilustración 12 - Cuádrica elipsoide.

Paraboloide: $$\frac{(x-l)^2}{a^2}+\frac{(y-m)^2}{b^2}-z+n=0 \qquad (2.7)$$

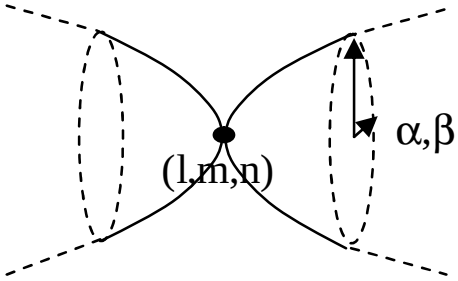

Ilustración 13 - Cuádrica paraboloide.

Hiperboloide: $$\frac{(x-l)^2}{a^2}+\frac{(y-m)^2}{b^2}-\frac{(z-n)^2}{t^2}-1=0 \qquad (2.8)$$

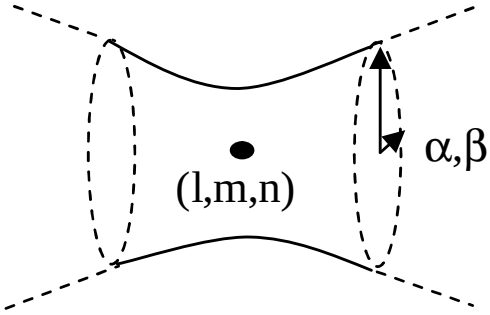

Ilustración 14 - Cuádrica hiperboloide.

siendo (l,m,n) el centro del objeto en cuestión y (α,β,τ) los semi-ejes.

Fijémonos que la esfera también pertenece al grupo de las cuádricas, entonces por qué la hemos parametrizado de una forma ajena a ellas, simplemente porque hay formas que resultan mucho más simples si las consideramos como ajenas a un grupo como será el caso de la esfera, y del cono o cilindro finito. Si estas formas las parametrizamos independientemente queda demostrado que la ecuación se simplifica y como veremos en capítulos posteriores la intersección con el rayo también se simplificará.

2.3.2 Cuárticas

Así como en la formula de las cuádricas se obtienen 2 resultados por ser de grado 2, en las fórmulas de las cuárticas se obtienen 4 resultados o raíces por ser de grado 4.

Un ejemplo sería el toroide:

$$(x^2+y^2+z^2-(a^2+b^2))^2=4*a^2*(b^2-z^2) \qquad (2.9)$$

que se define por un círculo de radio b que gira sobre el eje z distando de él una distancia a.

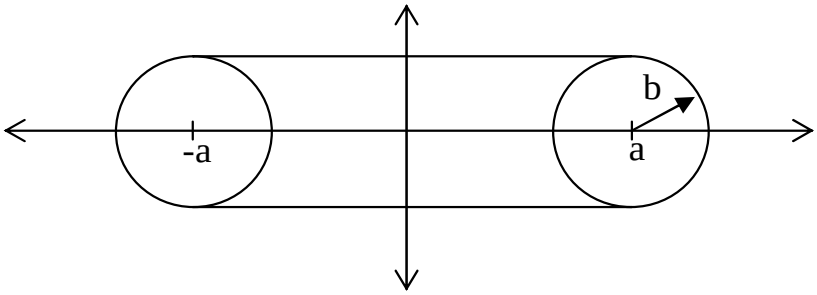

Ilustración 15 - Sección transversal de una cuártica toroide.

2.4 Representación de objetos fractales

Se ha escrito mucho sobre fractales y no creo que ahora vaya a explicar nada nuevo, sin embargo definiré los fractales como una operación matemática que define en 1, 2, 3 o 4 dimensiones un objeto característico. Tal es el caso de fractales archiconocidos tipo Julia [2] o Mandelbrot [3].

Un caso especial dentro de los fractales determinísticos en 3 dimensiones son los fractales por medio de quaterniones Julia [4]. Los cuales definen su forma en 3 dimensiones por medio del recorrido que genera un quaternion dado sobre el centro de coordenadas.

Ilustración 16 - Fractal basado en los quaterniones Julia.

Para saber si un punto pues pertenece o no a dicho fractal habremos de utilizar un algoritmo específico que en el caso de los quaterniones Julia se define así: un punto será perteneciente al fractal si después de un número de iteraciones elevado sobre el recorrido del quaternion, éste no tiende a una distancia del centro de coordenadas infinita. De este modo, si un punto se hace mover según el quaternion un número elevado de veces y después de ello vemos que sigue estando cerca del origen, ese punto pertenece al quaternion Julia.

Se definen así los fractales de una forma muy simple dejando al lector la posibilidad de encaminarse a las referencias propuestas o al mismo código que implementa los quaterniones Julia en el PiscisRT.

2.5 Representación booleana o CSG

La representación booleana o CSG se basa en las operaciones booleanas sobre diferentes objetos. Así se puede aplicar una unión, diferencia o intersección entre objetos a fin de conseguir una geometría sólida constructiva (constructive solid geometry CSG) nueva.

Un objeto complicado podrá ser representado mediante un arbol binario donde cada nodo representará una operación booleana y dos nodos hijos con un objeto ya definido en cada uno. La mejor forma de verlo será mediante un ejemplo:

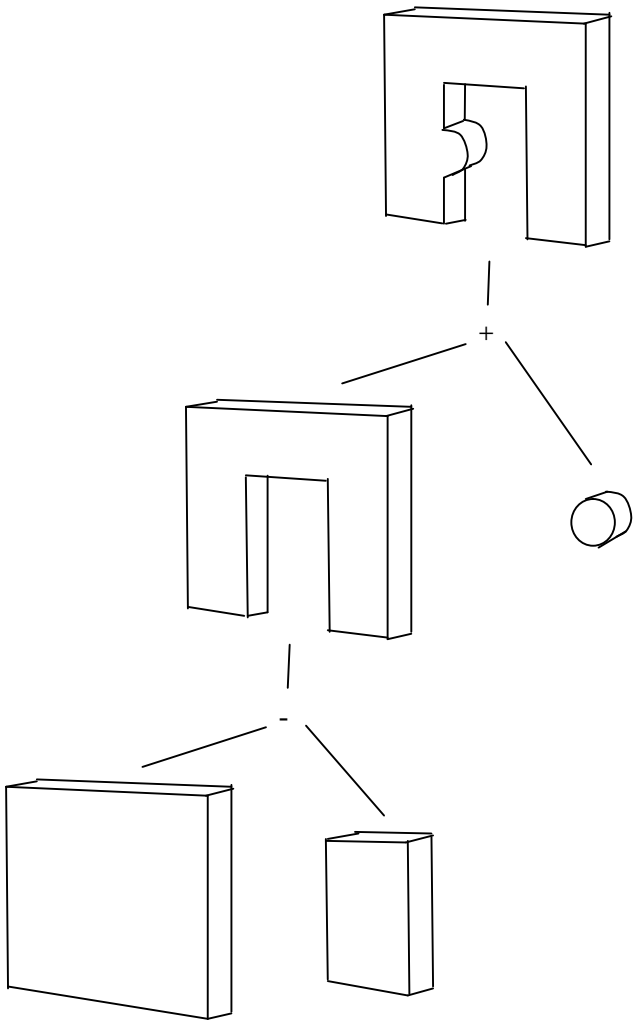

Ilustración 17 - Ejemplo de gemoetría sólida creada a base de operaciones CSG.

De esta forma tenemos la herramienta para crear formas sólidas muy complejas a partir de otras formas básicas, primitivas o reprsentaciones CSG.

Esta representación ha sido utilizada en casi todos los programas de modelado geométrico o de síntesis realista, tal es el caso del PovRay o 3d Studio Max.

Capítulo 3: Modelando los materiales

Una vez definidos todos los objetos en 3 dimensiones que podemos utilizar en un ámbito informático pasemos a darles el realismo que necesitan.

En primero lugar hemos definido un objeto como una superficie de puntos en 3 dimensiones pero en la realidad no vemos una superficie sino materiales con la forma del objeto. Por ejemplo si un objeto es de color rojo o si tiene aspecto rugoso o reflectante, todo eso ha de ser definido también en un modelo informático a fin de conseguir luego un render realista.

Los materiales son una de las principales partes de un buen programa de síntesis realista de imágenes y cuanto más se acomode el render del material a la realidad mucho mejor.

3.1 Color

En primero lugar es esencial el color del material que en el caso más simple de realismo puede definir completamente al material. Imaginemos un render de objetos con sólo el color como definitorio del material. Veremos así por ejemplo un par de esferas, una roja y otra verde.

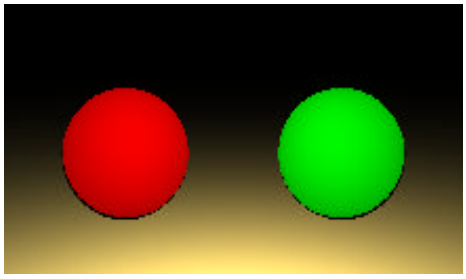

Ilustración 18 - Materiales con color.

El hecho de sólo utilizar el color puede parecer una buena solución pero fijémonos que en la realidad el color no es el único factor necesario.

3.2 Color ambiente

Imaginemos que queremos darle al material un factor de color ambiente, que sería el color por defecto en la escena. Este color ambiente se sumaría con el color del material.

La misma escena de antes con un color ambiente (0.2, 0.2, 0.2) en los dos materiales nos daría una imagen como la que sigue.

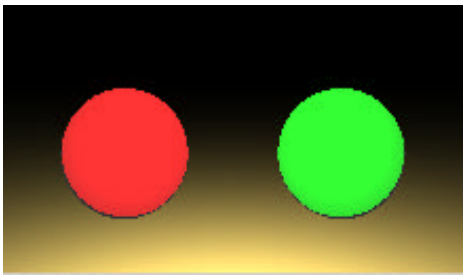

Ilustración 19 - Materiales con color ambiente.

El color ambiente es esencial si lo que se quiere es dar más claridad a los materiales de una escena que por defecto parece iluminada.

3.3 Factor de brillo especular

Vamos a añadirle ahora al material un factor de brillo o lustre que de al material una apariencia más metálica.

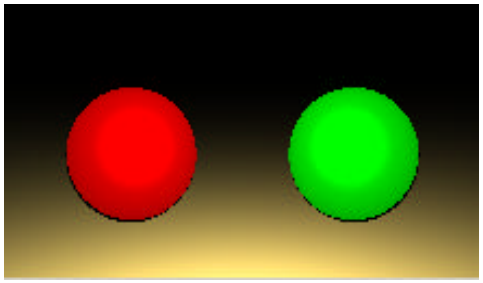

Ilustración 20 - Material con brillo especular.

Este factor de brillo especular influye en el color total de forma que los puntos del objeto que tienen una normal más directa hacia el observador y a la vez también hacia la luz, tendrán un color más cercano al blanco. Esto se hace para simular el brillo que se da en la realidad en tales situaciones.

Para simularlo pues necesitaremos conocer pues la dirección en el punto hacia la luz y hacia el observador y multiplicaremos escalarmente estas dos direcciones. De ser positivo el resultado elevaremos el resultado de la multiplicación al factor de shininess y luego el resultado de la potencia será multiplicado por el factor de brillo especular.

El código en el *PiscisRT* que realiza esta operación es:

```
PRTFloat specular=(-in)*((-out).Reflection(normal));
if(specular>0) specular=pow(specular,objeto->material->shininess)*objeto->material->specular;
else specular=0;
```

Ilustración 21 - Código PiscisRT para el cálculo del brillo especular.

Se explicará más sobre este brillo especular en el capítulo siguiente cuando expliquemos las técnicas de BRDF.

3.4 Textura

En todos los ámbitos de la síntesis realista de imágenes por ordenador se utiliza lo que se llama el mapeado de textura. Se basa en darle al material del objeto la apariencia óptima mediante un mapa de bits que reproduce su superficie real. Así por ejemplo si tenemos un objeto que ha de parecer madera, ¿qué mejor que aplicarle una textura al material con una imagen de madera?.

3.4.1 Coordenadas de textura

Hemos de definir un método para poder asignar a cada punto de la superficie del objeto un pixel definido dentro del mapa de bits. Esto se hace mediante el uso de coordenadas de textura.

Estas coordenadas de textura se moverán en dos ejes u y v dentro del mapa de bits, usualmente entre 0 y 1.

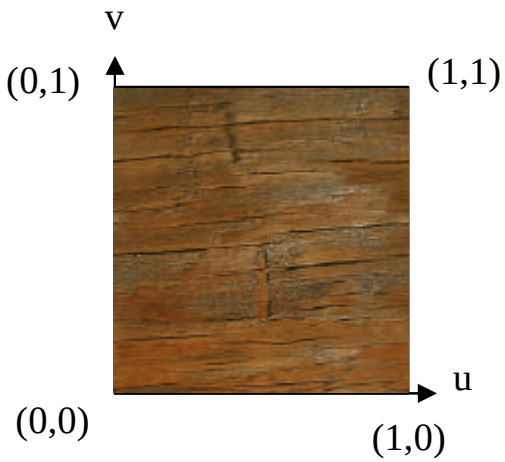

Ilustración 22 - Coordenadas de textura.

Estas coordenadas de textura nos dan además la clave para poder hacer repetir la textura tantas veces como queramos o de desplazarla a través de sus ejes.

3.4.1 Mapeado inverso

Nos asalta ahora otra cuestión muy importante y es que teniendo la textura en un mapa de bits n * m cómo vamos a aplicarlo a objetos diferentes y que en todos se aplique bien, puesto que en una esfera el mapeado habrá de ser esférico y en un cilindro habrá de ser cilíndrico. En cierto modo ya estoy dando la solución pues la respuesta es lo que llamamos mapeado inverso del objeto.

El mapeado inverso consiste en calcular que coordenadas de textura sobre un mapa de bits serán las que se han de aplicar sobre un punto en 3 dimensiones perteneciente a un objeto dado. Nos podemos referir a Paul Heckbert para desentrañar todo este problema y las soluciones para la mayoría de las formas básicas conocidas.

Con todo esto, aplicando sobre las esferas una textura de madera y utilizando mapeado inverso esférico, la escena quedaría:

Ilustración 23 - Material con textura.

Sin duda alguna cada vez el material parece más realista.

3.5 Mapa de elevaciones (Bump map)

Existe otra forma de aplicar un mapa de bits al material de forma que nos defina un material real. Este es el caso del bump mapping, que nos servirá para darle al material una rugosidad que a su vez se codifica dentro de un mapa de bits asignado.

La forma con la que esta rugosidad se crea es sencilla de comprender. Imaginemos que perturbamos la normal de cada punto del objeto según ese valor dentro del bump map. Al haber perturbado la normal de cada punto con valores distintos ahora la apariencia del objeto será mucho más complicada que antes, como si el objeto fuera mucho más complicado que a por ejemplo una esfera simple, sin dejar de ser una esfera con un bump map aplicado encima.

Para aplicar la perturbación a la normal de un punto, necesitamos dicha normal en el punto pero también la tangente y la binormal en ese punto de modo que quede definida la correspondencia entre (u,v) en la textura y (x,y,z) en la realidad. Calcularemos por tanto la variación que se da al pasar de un pixel al vecino más proximo y esta variación nos dará un vector de perturbación respecto a la normal del objeto.

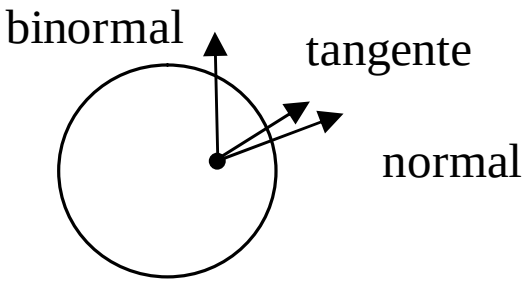

Ilustración 24 - Normal, tangente y binormal en un punto.

Esta perturbación de la normal del punto en la superficie del objeto influye en la iluminación de una forma tal que a nuestros ojos el objeto parece mucho más real que antes. La forma con que se aplica este cambio en la iluminación y el cómo se implementa el bump mapping en el presente proyecto serán asuntos a explicar en los siguientes capítulos.

Ilustración 25 - Material con bump map y su correspondiente mapa de bits.

3.6 Factor de reflexión

El factor de reflexión define que cantidad del color final pertenece a la influencia del rayo de luz de se refleja en el punto del cual queremos saber el color. Si este factor es 0 entonces el material no refleja nada, en cambio si se acerca a 1 refleja todo su ambiente como si fuera un espejo.

El principio físico de la reflexión de un vector incidente I respecto a una superficie con normal N nos da un vector reflectado R de la siguiente forma:

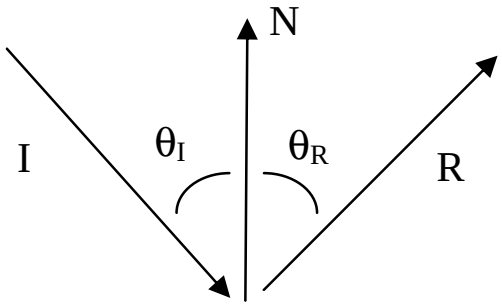

Ilustración 26 - La geometría de la reflexión.

Por tanto:

$$q_I = q_R \ , \ R = aI + bN \qquad (3.1)$$

de donde se podrá extraer en forma vectorial según [5]:

$$R = I - 2(N \cdot I)N \qquad (3.2)$$

Este vector R definirá la nueva dirección hacia algún punto que ha de intervenir según el factor de reflexión en el valor del color resultante (esto se verá más claramente cuando expliquemos el ray tracing).

Si aplicamos a la escena anterior materiales con reflexión 0.6 en ambos casos, podemos observar como una esfera refleja el suelo y parte de la otra esfera.

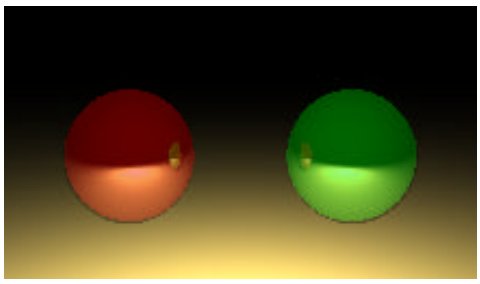

Ilustración 27 - Material con reflexión.

Digamos que las esferas parecen hechas de superficie metálica reflectante.

3.7 Factor de refracción

El factor de refracción funciona de forma idéntica al anterior pero con otro principio físico real, la refracción de los objetos. Según esta refracción en un material que refracte al estilo de un cristal la luz incidente sigue un camino específico introduciéndose en el objeto y luego saliendo o no del mismo. Es el típico experimento de introducir un lápiz dentro del agua y ver cómo su apariencia cambia según la inclinación con la normal del agua.

Hemos de darnos cuenta aquí que no todos los materiales refractan de forma exactamente igual pues cada material podemos decir que tiene un coeficiente de refracción distinto. Este coeficiente de refracción indica la intensidad con que el efecto de refracción se da en dicho material.

El principio físico de la refracción de un vector incidente I respecto a una superficie con normal N nos da un vector refractado o transmitido T según dos coeficientes de

refracción η_1 para el medio 1, η_2 para el medio 2 y η_{21} para el medio 2 con respecto al medio 1 de la siguiente forma:

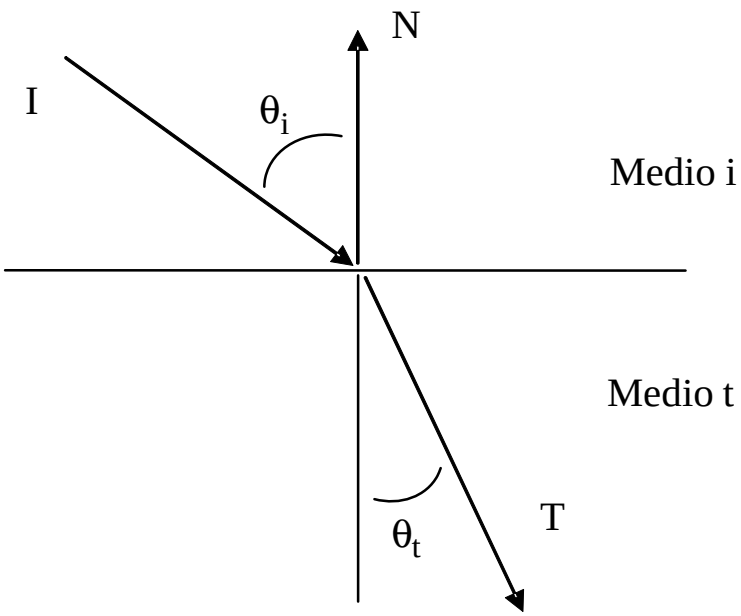

Ilustración 28 - La geometría de la refracción.

Por tanto:

$$\frac{\sin q_t}{\sin q_i} = h_{it} = \frac{h_2}{h_1} \ , \ T = aI + bN \qquad (3.3)$$

de donde se podrá extraer en forma vectorial según [5]:

$$T = h_{it} I + (h_{it} \cos q_i - \sqrt{1 + h_{it}^{\ 2} (\cos q_i^{\ 2} - 1)}) N \qquad (3.4)$$

Este nuevo vector T definirá ahora de forma parecida a la reflexión la nueva dirección hacia el valor de la refracción. Se aplicará según el factor de refracción siendo 0 el mínimo y 1 el máximo.

Es conveniente que aquí se comenten dos cosas, la primera es el valor del índice de refracción y la segunda una singularidad llamada reflexión total interna (total internal reflection) [5].

Tenemos que definir el efecto de refracción definiendo el índice de refracción apropiado a cada medio, por ejemplo un índice de aire cuando el medio esta fuera de las esferas y otro de cristal cuando el medio esta dentro de ellas. Observemos la siguiente tabla de ejemplo para cerciorarnos del factor apropiado:

Medio	Índice
Agua	1.33
Alcohol etílico	1.36
Aire (1 atm,20ºC)	1. 0003
Cuarzo fundido	1.46
Cristal	1.66

Ilustración 29 - Tabla de índices de refracción.

Como hemos anunciado antes existe un efecto especial llamado reflexión total interna, este fenómeno ocurre cuando la luz intenta pasar de un medio denso a otro menos denso mediante un ángulo demasiado bajo. De este modo en vez de producirse un vector de transmisión se produce uno de reflexión. Es el efecto por ejemplo de la fibra óptica en la cual el rayo de luz siempre va rebotando dentro de ella y nunca se transmite fuera de la fibra óptica.

En el ejemplo que sigue por debajo del ángulo crítico la luz tanto es transmitida a través de la superficie como reflectada desde la misma. A ángulos superiores al crítico sólo se da la reflexión.

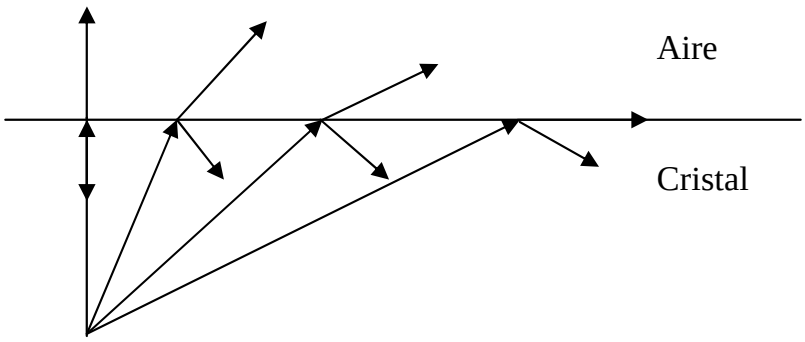

Ilustración 30 - El fenómeno de la reflexión total interna (TIR).

Explicado todo lo anterior pasemos a aplicara la escena que hemos estado utilizando un índice de refracción de 0.7 a ambos materiales y disponemos otra esfera detrás a fin de observar bien el efecto de la refracción la imagen resultante será:

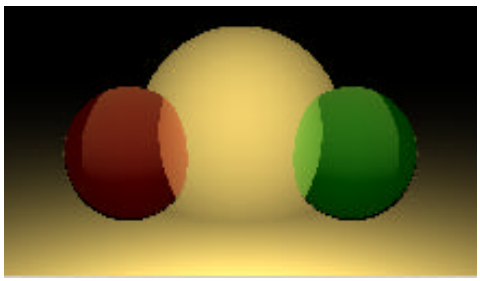

Ilustración 31 - Material con refracción.

Podemos observar como el interior de las esferas es resultado de la refracción de los objetos que hay detrás de ellas. Las esferas ahora se asemejan a esferas de cristal.

3.8 Factor de transparencia

Por último, el factor de transparencia indica que cantidad del color del punto recae sobre el color del punto situado inmediatamente detrás de él. Digamos que hacemos transparente al material en una cantidad igual al factor de transparencia. Así en un material con transparencia 0 no se observarán cambios con respecto a él mismo, con 0.5 se verá tanto el material del objeto como el del objeto que haya detrás de el y con 1 el objeto desaparecerá de la escena.

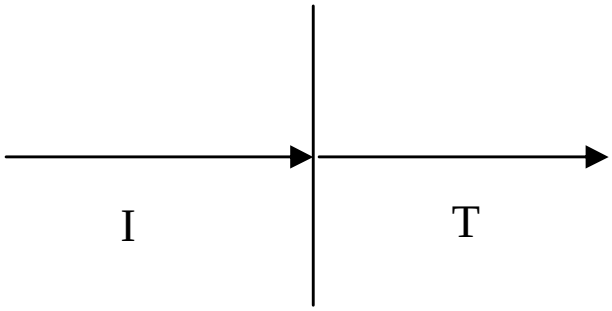

Ilustración 32 - La geometría de la transparencia.

Por tanto se ve simplemente que el vector incidente y el vector transparente son el mismo:

$$T = I \qquad (3.5)$$

Al aplicar materiales con transparencia 0.3 y con una esfera detrás, la escena que nos sirve de ejemplo se vería como sigue:

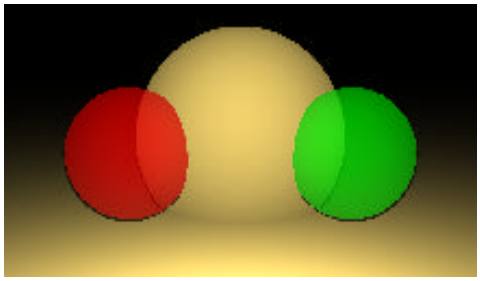

Ilustración 33 - Material con transparencia.

Por tanto las esferas se han vuelto semitransparentes perdiendo así la sensación de solidez. Esta transparencia nos servirá por ejemplo para simular cristales planos en los que no hay refracción visible.

Capítulo 4: Modelando la iluminación

Recapitulando, sabemos que existen objetos y que éstos tienen asignado un material específico. De todos modos esto no es suficiente, en la realidad hay algo más sin lo cual no se podría observar nada: la luz. En este capítulo vamos a explicar que tipos de luz y que modelos de iluminación se suelen utilizar en los sistemas informáticos capaces de renderizar y más explícitamente en los trazadores de rayos.

En primer lugar hemos de imaginarnos nuestra escena sin luz, en ella no habrá sensación de realismo pues todos los objetos se verán como si hubiera luz en todas partes de la escena. Esto puede resultar paradójico pero no lo es, cuando no se tienen en cuenta las luces, el material de los objetos se podrá considerar como iluminado al máximo por igual.

Así mismo, sin luces no habrá sombras y las profundidades de la escena serán mucho más difíciles de percibir para el ojo humano.

Con el fin de incluir luces en la escena lo primero que hemos de señalar es que la luz dispone de dos parámetros esenciales que son la intensidad de luz y el color de la misma. Esta intensidad se puede ver como la potencia lumínica de la luz y será uno de los factores más importantes que definen la apariencia de la escena final. Por otro lado el color de la luz se puede codificar como un vector (rojo, verde, azul) aunque la física nos dice que el color de la luz es provocado pro la longitud de onda provocada.

En segundo lugar habremos de representar la atenuación que provoca la luz sobre los objetos.

4.1 Atenuación

La atenuación es la propiedad física de la luz que nos dice que cuanto más alejado esté el objeto de la luz más se atenúa la luz que llega a este objeto. La intensidad de luz original sólo se dará en la distancia 0 de la luz al objeto y cuando esta tienda a 0 la intensidad también lo hará.

Así tenemos distintos tipos de atenuaciones respecto a la distancia entre el objeto y la luz.

La primera sería la atenuación lineal:

$$I_f = \frac{I_o}{dist} \qquad (4.1)$$

siendo I_f la intensidad al llegar al objeto, I_o la intensidad de la luz original y dist la distancia entre el objeto y la luz.

Directamente de la lineal se puede extraer una atenuación cuadrática:

$$I_f = \frac{I_o}{dist^2} \qquad (4.2)$$

Así la atenuación crecerá cuadráticamente con la distancia.

En general pues podemos decir que:

$$I_f = \frac{I_o}{a + b * dist + c * dist^2} \qquad (4.3)$$

donde a, b y c serán los coeficientes de la atenuación de la luz.

Una escena con atenuación tendrá el aspecto un poco más real pues los objetos alejados de la luz se verán más oscuros que los cercanos y al mover la luz cambiará toda la escena. Aún así esto no es suficiente, siguen sin haber sombras y no se tiene en cuenta el ángulo del objeto respecto a la luz. En el siguiente apartado vamos a estudiar como decidir qué intensidad recae sobre cada punto de un objeto teniendo en cuenta que la normal en ese punto decide si la luz recae directa o indirectamente.

4.2 BRDF

El BRDF (Bilinear Reflectance Distribution Function) de un material nos indicará como la luz es dispersada por su superficie. Aun siendo parte del material he preferido explicarlo aquí en la parte de iluminación porque es el único parámetro del material dependiente de la luz.

El BRDF del material será una función dependiente de la normal del punto, del vector de incidencia de la luz y de en ciertos casos el vector de salida hacia el observador. El último vector nombrado sólo se utilizará en casos donde se quiere tener en cuenta que la apariencia del objeto depende de desde donde se mira. Así:

$$I = f(u, n, v) \qquad (4.4)$$

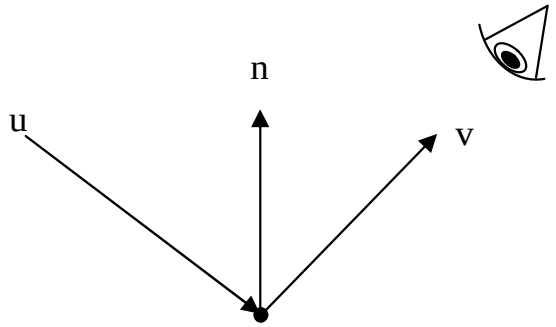

Ilustración 34 - Bilinear reflectance distribution function.

siendo u el vector de incidencia de la luz, n la normal en el punto y v el vector dirección hacia el observador. I será pues el valor buscado que nos dará el valor numérico que en ese punto del objeto representa la intensidad de luz dispersada. Este valor multiplicado por la intensidad de la luz y atenuado por la atenuación de la distancia nos dará la intensidad de luz que llega al punto requerido del objeto.

4.2.1 Lambertian diffuse

Quizá la iluminación más sencilla sea la difusa. Ésta es la que menos tiene de BRDF pues no depende de ninguno de sus factores, ni de u, ni de n, ni de v, pero igualmente se utiliza mucho y se puede representar como sigue:

$$I = \frac{1}{p} c1 \qquad (4.5)$$

siendo c1 el llamado factor de reflectividad total, que por defecto será 1.

4.2.2 Phong

La iluminación tipo Phong [6] o también llamada interpolación normal-vector se utiliza en casi todos los ambientes de render informáticos, incluso en librerías gráficas en tiempo real. La misma, se puede codificar en una función BRDF de la forma siguiente:

$$I = \cos \vec{u}\vec{n} \qquad (4.6)$$

siendo $\cos \vec{u}\vec{n}$ el coseno entre los vectores u y n. Como se puede observar la iluminación Phong no tiene en cuenta el vector hacia el observador y por eso el objeto será iluminado de forma igual para cualquier posición del observador.

Veamos pues ahora un ejemplo de BRDF que tenga en cuenta todos sus factores u, n y v.

4.2.3 Cosine lobe

El cosine lobe será uno de los modelos que nos puede servir de ejemplo para observar todo el potencial del BRDF. Usa todos los vectores definitorios del BRDF y su forma será como la que sigue:

$$I = (refl(-u,n) * v)^{c1} * \frac{c1+2}{2p} \qquad (4.7)$$

siendo refl(-u,n) el vector reflejo de –u respecto a n y c1 el llamado factor especular. Éste factor especular es propio del uso del vector v hacia el observador, así cuanto más cerca esté el observador de forma un ángulo recto con la luz en el punto del objeto, más fuerte será la iluminación en dicho punto.

Hay muchos más modelos de iluminación, cada uno quizás más apropiado a un caso específico que otros, y otros que intentan codificar el BRDF más realista posible. Además de los que se explican en esta memoria hay muchos otros implementados en el código del PiscisRT y en referencias como [7] donde uno habrá que remitirse para un mayor entendimiento de los modelos de iluminación BRDF.

4.3 Tipos de luces

Hemos hablado pues de qué características tiene una luz pero no hemos discernido sobre si hay más de un tipo y como se comportan según el tipo que sean. Esto es lo que vamos a ver en este apartado.

En efecto, hay varios tipos de luces y entre ellos se diferencian principalmente en su forma y cómo los rayos de luz son lanzados desde ella. Aunque existen muchísimos tipos de luces vamos a elegir tres de los más importantes y observar cómo se comportan.

4.3.1 Luces omnidireccionales

La luz omnidireccional es la más intuitiva de todas. Es la que provoca un punto de luz infinitesimal en todas las direcciones por igual. De ahí su nombre, ya que ilumina todas las direcciones.

Una luz omnidireccional se define por tanto por su posición en 3 dimensiones y los parámetros ya conocidos hasta ahora: atenuación, intensidad y color.

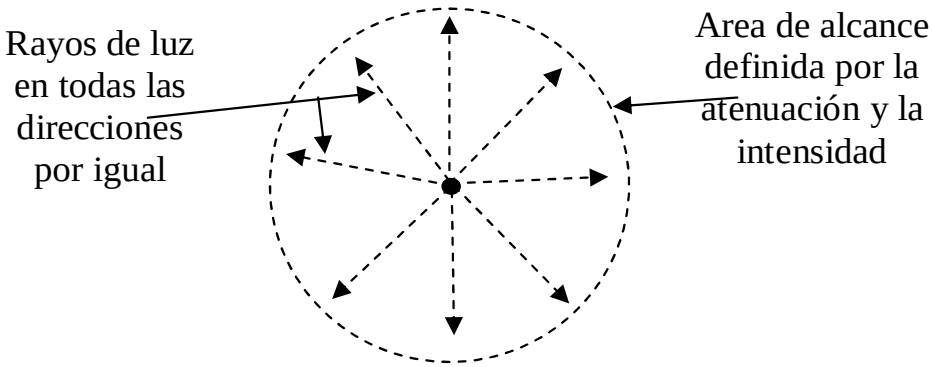

Ilustración 35 - Luz de tipo omnidireccional.

Una luz de este tipo nos servirá para simular bombillas o velas o incluso el sol ya que aunque no es un punto infinitesimal lo parece debido a su gran distancia hasta nosotros los observadores.

4.3.2 Sombras en luces omnidireccionales

Hemos podido observar que hasta ahora los renders producidos mediante lo expuesto no tendrían la capacidad de simular las sombras. También podemos comprender que este aspecto es fundamental si queremos representar bien la realidad. Por tanto veamos qué es la sombra y como se puede representar.

Una sombra se provocará cuando el punto del objeto que queramos representar tiene otro objeto interpuesto en el vector hacia la luz. Es decir, una sombra se produce cuando un objeto tapa a otro la luz.

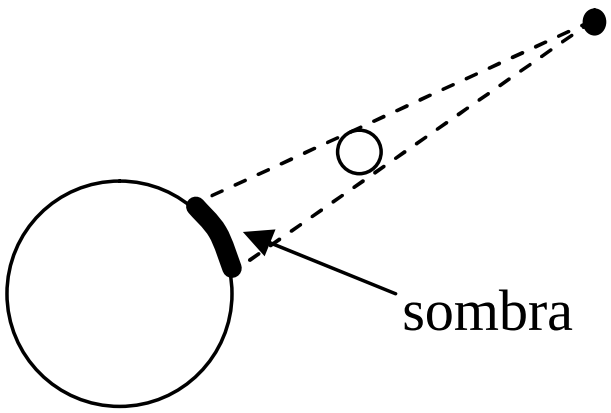

Ilustración 36 - Caso sencillo de sombra entre objetos con luces omni.

Estos casos son los más sencillos de sobra y en ellos diremos que el color de los puntos en sombra es del color de la sombra (en general negro (0, 0, 0)).

Sin embargo no siempre es tan sencillo deducir si un punto se encuentra en sombra o no.

Por ejemplo cuando hay más de una luz el punto puede estar en sombra de sólo una o más de una luz, y tendremos que calcular que luz o luces interferirán en el color del material resultante.

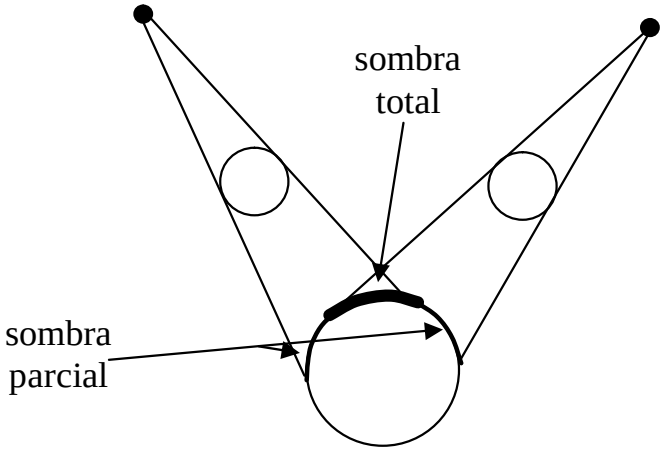

Ilustración 37 - Caso de sombra con multiples luces omni.

En otros casos, cuando haya objetos traslucidos a veces la luz pasará por ellos y las sombras no se provocarán por culpa de estos objetos. Y además puede haber objetos reflectantes o refractantes que cambien la dirección de la luz, estos casos podemos adelantar aquí que serán de ámbito global y se solucionarán con métodos de iluminación global, no como los otros más simples que se podrán solucionar con la teoría de los trazadores de rayos.

Veamos una escena de ejemplo:

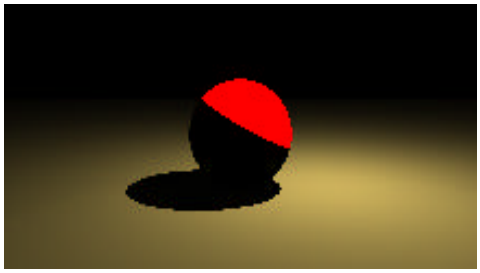

Ilustración 38 - Sombra con una luz omnidireccional.

Se puede observar claramente como se aprecia una luz de tipo omni en una posición arriba y a la derecha de la esfera y como produce sombras arrojadas en el suelo de la esfera. Estas sombras se habrán calculado basándose en lo expuesto anteriormente y al no haber más de una luz la misma sombra será negra.

Otra escena de ejemplo con dos luces omni nos servirá para observar el efecto de las sombras múltiples:

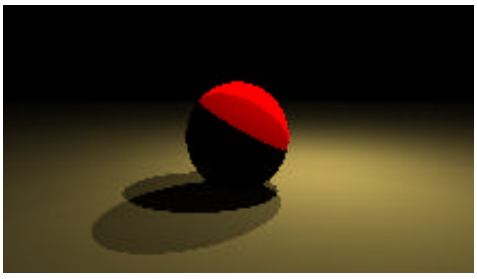

Ilustración 39 - Sombras con dos luces omnidireccionales.

Ahora se observa el efecto de las dos luces sobre la esfera, una en la posición de la escena anterior y otra más hacia el fondo de la escena. Se observa el efecto tanto en la

superficie de la esfera por lo mencionado en los BRDF como en las sombras arrojadas en el suelo, con sombra total en la intersección de las sombras individuales.

Una vez visto el caso de las luces omnidireccionales veamos los otros tipos y cómo se calculan sus sombras.

4.3.3 Luces direccionales

Como su propio nombre indica esta luz sólo lanza rayos de luz en una dirección predefinida que será su parámetro principal. Al contrario que las luces omnidireccionales que tenían una posición conocida, las direccionales no la tienen. Sólo son definidas por su dirección y por eso tampoco les afectará la atenuación producida por la distancia al punto a iluminar.

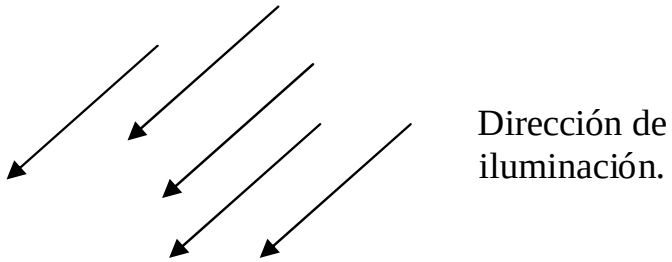

Ilustración 40 - Luz de tipo direccional.

Estas luces representan casos donde la luz se encuentra tan lejos y es tan potente que no vale la pena calcular la dirección que toma en cada punto ni la atenuación que produce. En todos los puntos de la escena la luz procederá de la dirección planteada y con la intensidad y color de la luz.

4.3.4 Sombras en luces direccionales

Siguiendo la misma deducción de lo que es una sombra que en las sombras para luces omni, en el caso de las luces direccionales las sombras tendrán el siguiente aspecto:

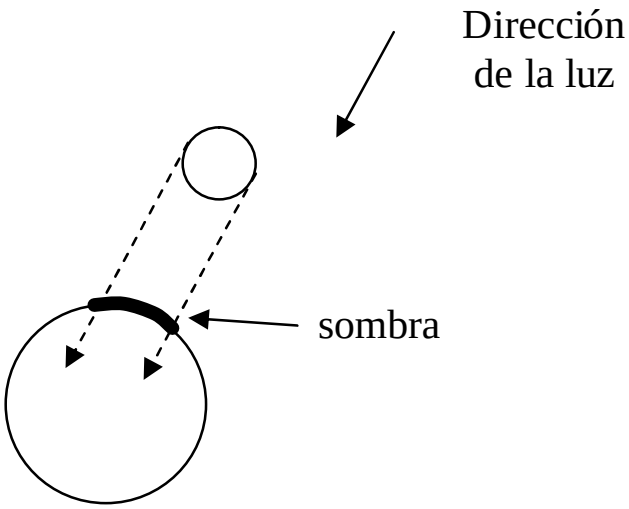

Ilustración 41 - Caso sencillo de sombra entre objetos con luces direccionales.

Las sombras aquí se podrán asimilar a proyecciones de la forma que crea la sombra sobre la que la recibe, en la dirección de la luz direccional.

En casos como los que vimos en las luces omni con más de una luz el efecto será similar y provocará sombras totales y parciales siguiendo la proyección de las formas.

Rendericemos las escenas de ejemplo anteriores pero esta vez con luces direccionales:

Ilustración 42 - Sombra con una luz direccional.

La luz de esta escena ha sido definida por la dirección (-1,-1,0), vemos así la principal diferencia con las omni: no hay atenuación.

En la otra escena se definían dos luces, esta vez una con dirección (-1,-1,0) y la otra con dirección (-1,-1,1).

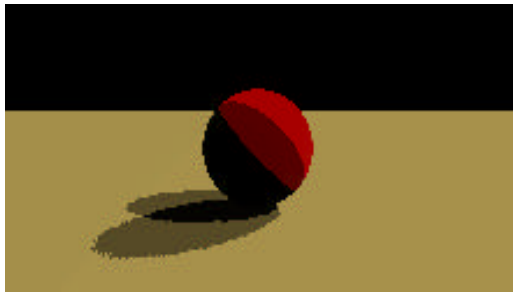

Ilustración 43 - Sombras con dos luces direccionales.

Similar al caso de las luces omni, se ven sombras totales y parciales en direcciones similares.

En resumen diremos que las luces direccionales son un tipo más simple de luces que aunque generan escenas menos definidas nos servirán para simular luces muy lejanas o de ámbito muy global que han de actuar sobre toda la escena por igual.

4.3.5 Luces de área

Vistas las dos luces anteriores cabe preguntarnos si las luces sólo tienen forma de punto o de dirección, y no es así, la luz puede tomar la forma que quiera incluso la de un objeto cualquiera. Así, las luces pueden tener un área definida en la que los rayos de luz serán lanzados en todas las direcciones por igual repartiendo su intensidad de luz.

Existen luces con forma de rectángulo simulando tal vez luces de techo fluorescentes o luces también con forma de esfera simulando la forma real de nuestro sol o de una bombilla. Estas luces emitirán rayos de luz en todas direcciones y por eso existe mayor complejidad que antes, ahora la luz que llega a un punto del objeto ya no sólo llega de un punto de la luz sino de todos a la vez.

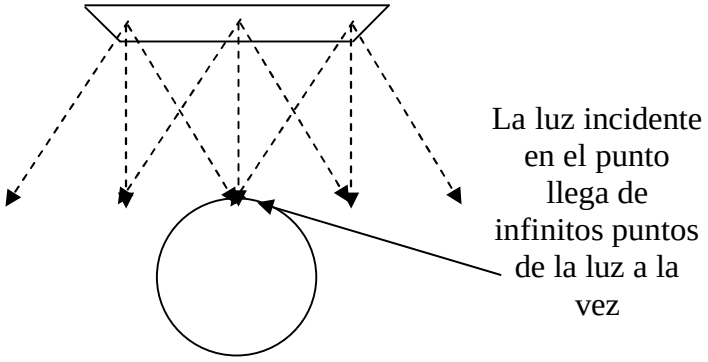

Ilustración 44 - Complejidad de las luces de área.

Debido a que no podemos calcular la influencia de todos los puntos de la luz a la vez y tampoco decidir que puntos tomamos y que otros no, la única solución será utilizar distribuciones de probabilidad. Estas distribuciones de probabilidad nos darán puntos aleatorios dentro del área de la luz de forma que la iluminación estará igualmente compensada aunque quizás no sea muy real hasta que no hagamos el promedio de iluminación tomando un gran número de posiciones aleatorias en el área de esa luz.

Así pues tomaremos posiciones aleatorias en las luces, pero hemos de estudiar antes cómo se calculan estas.

En el caso del rectángulo, la luz se podrá dividir en dos triángulos y para obtener una posición aleatoria en él bastará según [11] con:

```
PRTFloat a=PRTWRandom();
PRTFloat b=PRTWRandom();
if ((a+b)>1){          a=1-a;      b=1-b;}
PRTFloat c=1-a-b;
return p1*a+p2*b+p3*c;
```

Ilustración 45 - Cálculo de una posición aleatoria dentro de un triangulo.

siendo a, b, c números reales, PRTWRandom() un generador de número aleatorio real entre 0 y 1, y p1, p2, p3 tres puntos que definen al triángulo. Para mayor entendimiento mejor nos referiremos a el proyecto PiscisRT en si.

Del mismo modo para calcular un punto aleatorio dentro de una esfera bastará con:

```
PRTFloat a,b;
PRTVector ret;
a=PRTFloat((PRTWRandom()*PRTPI)-PRTFloat(1.57));
b=PRTFloat(PRTWRandom()*PRTFloat(6.28));
ret.x=p.x+(r*PRTCos(a)*PRTCos(b));
ret.y=p.y+(r*PRTCos(a)*PRTSin(b));
ret.z=p.z+(r*PRTSin(a));
return ret;
```

Ilustración 46 - Cálculo de una posición aleatoria en la superficie de la esfera.

Sabiendo cómo se calculan estas posiciones por tanto nos bastará para solucionar parcialmente la iluminación para un número grande de estas posiciones en la luz y luego promediar sobre el resultado. Veremos esto con más detenimiento en el capítulo referido al ray tracing.

4.3.6 Sombras en luces de área

Las sombras en este caso se solucionarán también promediando sobre las sombras que producen puntos aleatorios de luz. De este modo la sombra no quedará tan definida

como las de las luces anteriores pero a mayor número de muestras promediadas mayor calidad de la sombra.

Así una luz de área con forma de triangulo puede formar la sombra que sigue sin promediar:

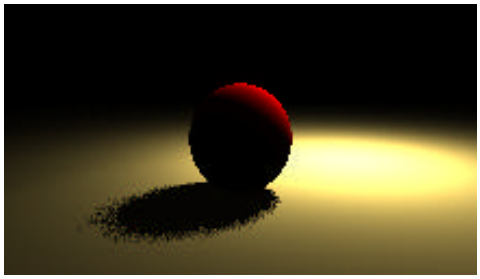

Ilustración 47 - Sombra con una luz de área triangular.

Vemos así como se produce granularidad en la sombra, que por otra parte es lógica debido a la utilización de distribuciones de probabilidad.

Del mismo modo que con las luces anteriores, si añadimos otra luz y promediamos sobre 5 en las sombras, la imagen resultante será:

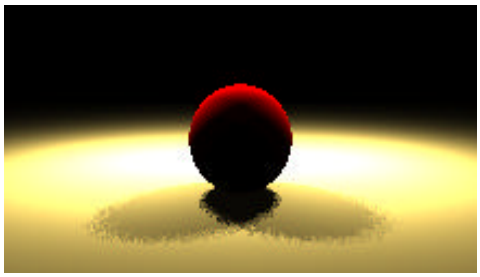

Ilustración 48 - Sombras con dos luces de área triangulares.

Por tanto las luces de área quedarán definidas y las implementaremos en el proyecto. Del mismo modo, todo lo referente a iluminación local se ha explicado en este capítulo y se ha dejado todo lo referente a iluminación global para el capítulo 6. Pero antes hemos de aprender el método que utilizaremos para renderizar la escena, el ray tracing, que nos ayudará a comprender mejor todo lo expuesto hasta ahora y nos dará paso libre para adentrarnos en la iluminación global o radiosidad.

Capítulo 5: Ray tracing

Hemos ido nombrando al ray tracing en los anteriores capítulos como base para realizar los renders utilizando lo expuesto hasta ahora. En este capítulo pues explicaremos como funciona el ray tracing y como es esencial que hayamos definido los objetos, materiales y modelos de iluminación para sacarle jugo al trazador de rayos.

Queda claro que hasta ahora se han introducido objetos definidos por diversas representaciones, materiales con multitud de parámetros de una apariencia real y modelos de iluminación y sombras que nos solucionan la iluminación local. Faltará por tanto explicar como se junta todo a fin de ir renderizando la imagen de forma correcta.

Para cualquier duda sobre el ray tracing aconsejo referirse a [5] y [8].

5.1 Introducción al ray tracing

El ray tracing es un método de síntesis de imágenes que crea una imagen de 2 dimensiones a partir de una escena en 3 dimensiones. Posiblemente porque todos los monitores son de forma plana o porque muchas veces las imágenes se imprimen sobre papel, el fin que se persigue es dar al observador la impresión de estar observando una fotografía sacada de un mundo en 3 dimensiones.

Por tanto lo primero que hemos de comprender y simular es cómo una cámara capta una imagen de un mundo en 3 dimensiones para luego nosotros hacer algo parecido en nuestro trazador de rayos.

5.1.1 Cámara pinhole

Se nos explica en multitud de materias que la cámara más simple de todas es la cámara pinhole. Una pieza plana de film fotográfico se dispone en el fondo de una caja cerrada ausente de luz. En la parte opuesta al film se perfora un pequeño agujero y se tapa para que no entre luz con algún material opaco.

Cuando se quiera tomar una fotografía, se quita este material opaco durante un lapso de tiempo pequeño. En ese tiempo la luz entrará por el agujero de la caja y producirá en el film fotográfico una serie de reacciones químicas que al fin producirán una fotografía. Cuando la foto se considera tomada se vuelve a tapar el agujero con el material opaco.

Aún su simpleza la cámara pinhole es muy práctica para tomar fotografías reales. Veamos la disposición de esta cámara:

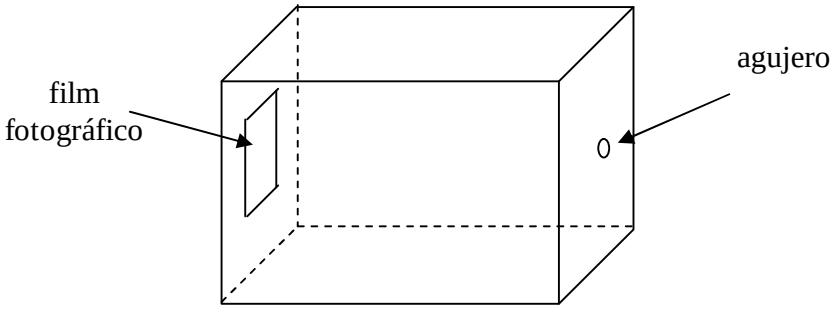

Ilustración 49 - La cámara pinhole.

Podríamos pensar que hacer entrar la luz por el agujero es una perdida de tiempo y que exponiendo el film directamente ganaríamos tiempo pero eso no es así. Si exponemos el film directamente la luz incidirá en él desde todas las direcciones y saturará en un instante toda la superficie. Obtendríamos así una imagen totalmente blanca. El aguje-

ro de la cámara pinhole soluciona este caso dejando pasar sólo un número pequeño de rayos de luz, es más, sólo podrá pasar el rayo de luz que coincida con ese trocito de film fotográfico y el agujero de la cámara. Variando el tamaño del agujero cada trozo de film recibirá más o menos rayos de luz y se volverá más o menos brillante.

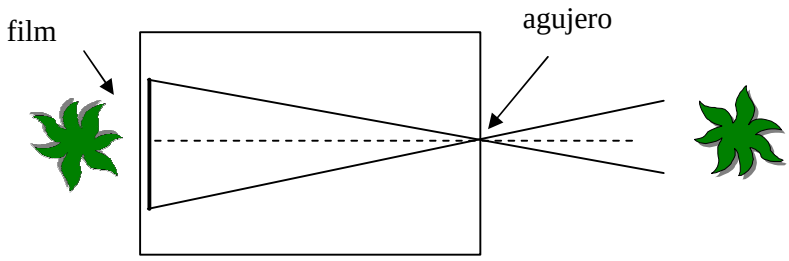

Ilustración 50 - Incidencia de rayos de luz en la cámara pinhole.

Aunque en la informática se han utilizado cámaras mucho más complicadas que esta, la cámara pinhole sigue siendo bastante popular debido a su simplicidad.

La versión clásica informática de la cámara pinhole desplaza el plano del film fuera de la caja y en frente del agujero y renombra el agujero como ojo. Si construyéramos una cámara real de este modo no funcionaría pero funciona bien en las simulaciones informáticas.

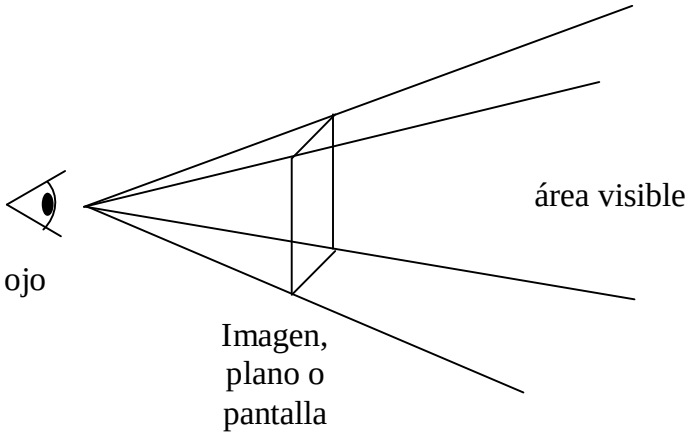

Ilustración 51 - La cámara pinhole modificada.

Las condiciones de la nueva cámara obligan a que el mundo visible desde la cámara esté dentro de la pirámide infinita sin base definida por el área visible y que todos los rayos de luz útiles pasen por el ojo o punto de visión.

5.1.2 Pixels y rayos

Un pixel es la componente más pequeña dentro de una imagen que normalmente codifica un color. Por tanto podemos deducir que para generar una imagen sólo hemos de determinar que color se ha de situar en cada pixel de la imagen. Si pensáramos en cada pixel como una ventanita pequeña hacia el mundo real, en ese pixel habría que codificar el color predominante en la escena que se ve desde la ventana.

El rayo en cambio se podrá definir como un camino recto con origen y dirección en 3 dimensiones. Por eso definiremos un rayo por un punto origen en la escena y definiremos su dirección para que su camino recorrido no tenga límite en esa dirección. Ya ve-

remos en secciones posteriores que podremos saber si este rayo colisiona con algún objeto y a qué distancia del origen del rayo lo hace, definiendo así un nuevo punto en 3 dimensiones.

Si ahora nos fijamos en la cámara pinhole podemos deducir que cada región del film se asocia con un pixel de la imagen. Además, como hemos dicho antes, a cada región del film se le asociarán un número finito de rayos de luz que inciden sobre ella. Por tanto para saber que color recae sobre cada pixel sólo hay que promediar sobre los colores que inciden con cada rayo de luz. Este promedio se podrá calcular por métodos distributivos o estadísticos con un número de rayos de luz a voluntad.

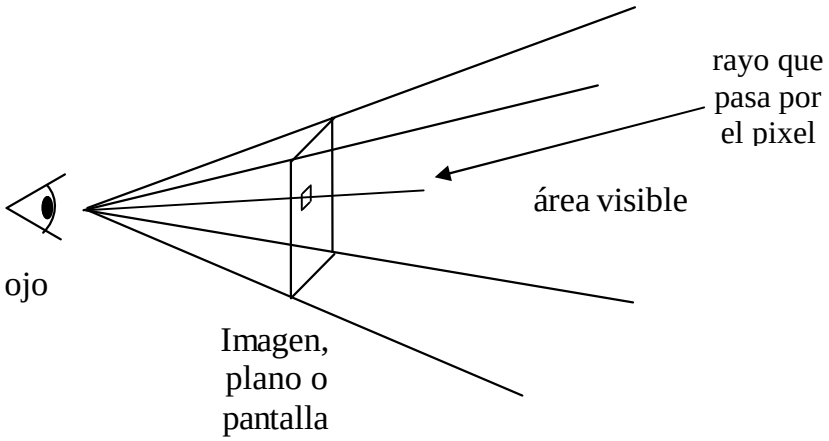

Ilustración 52 - Asociación entre pixel, región de film y rayo de luz.

El método del ray tracing nos dirá como manejar estos rayos de luz para saber el color que incide.

5.2 Trazando rayos

Hemos visto que hemos de calcular los rayos de luz que inciden en cada pixel y a partir de ellos obtener el color que resultan, pero de momento no sabemos cómo encontrar los rayos de luz ni saber que color representan.

El color de un rayo de luz no es del todo difícil de explicar. Podemos pensar en el rayo de luz cómo el camino que recorre una partícula de luz (llamada fotón) a través del aire u otro medio físico. En el mundo real un fotón transporta energía y cuando este fotón atraviesa nuestra retina la energía se traspasa a nuestro cerebro en forma de colores. Por tanto el color depende de la energía del fotón.

Si quisiéramos definir esta energía transportada por el fotón deberíamos referirnos a la energía de vibración de una onda. Y eso aun sabiendo que los fotones no vibran en la realidad pero que matemáticamente es útil e intuitivo asignarles una vibración para definir la energía que transportan. En este modelo de vibración del fotón, la velocidad de vibración conllevará diferentes energías y a su vez estas energías definirán diferentes colores. Por eso muchas veces definimos cada color por una frecuencia específica.

De todos modos hemos de observar que los colores o frecuencias de los rayos de luz se pueden mezclar entre ellos en un film fotográfico o en nuestro ojo. Así si un fotón rojo y otro verde llegan al mismo tiempo a nuestro ojo y en la misma posición de la retina, percibiremos la suma de ambos colores que en este caso será el amarillo.

Ahora consideremos un pixel de la imagen a renderizar. ¿Qué fotones de la escena en 3d contribuirán a su color?. Ya habíamos dicho que contribuirán aquellos que pasen

directamente por el pixel y el ojo del observador, pero no hemos de olvidar que los fotones se generan en las fuentes de luz en muchas direcciones diferentes y que la luz puede por ejemplo ir rebotando o retractándose a lo largo de la escena.

Observemos una escena de ejemplo:

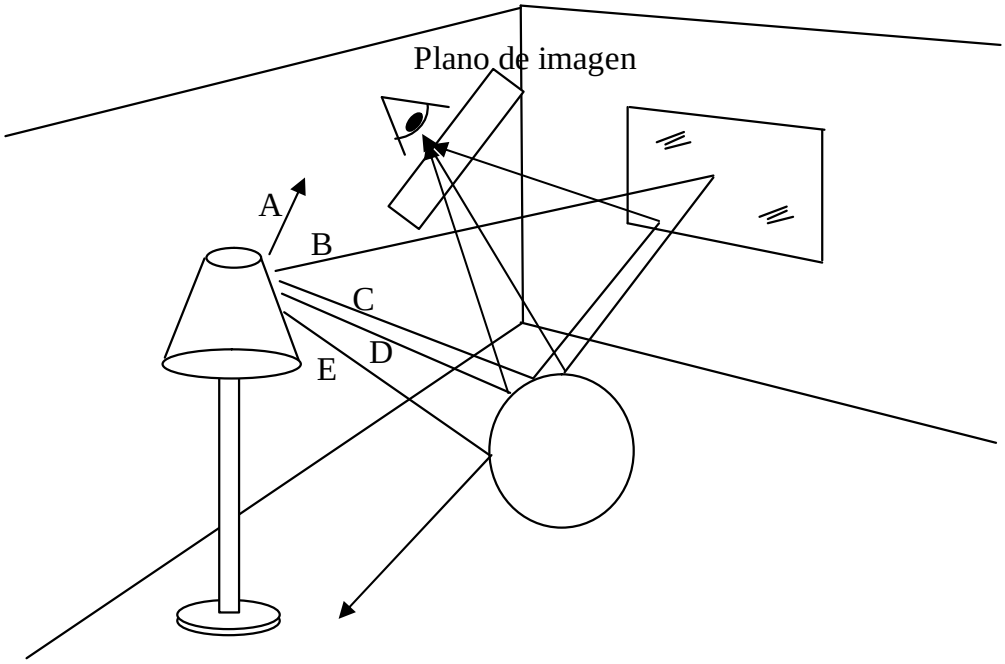

Ilustración 53 - El complicado camino de los fotones.

El fotón A conlleva un color azulado pero sale de la luz en dirección a la pared y esta no reflejará ninguna energía, por lo tanto el fotón no seguirá su camino y ni llegará a plano de la imagen.

El fotón B sale en dirección al espejo de la pared frontal, el cual hará rebotar el fotón sin perdida de energía hacia la esfera situada en medio de la escena. Esta esfera si hará perder un poco de energía al fotón por razones físicas de su material pero también lo dejará reflejarse en una dirección que pasará por el plano de imagen y llegará al ojo. El fotón B por tanto participará en el color del pixel por el que ha atravesado el plano de imagen.

El fotón C sale en dirección a la esfera, la cual lo hace rebotar con perdida de energía y su camino le llevará al espejo el cual a su vez lo llevará al ojo. El fotón C también participará en el render.

El fotón D sale en dirección a la esfera y esta lo rebota hacia el ojo directamente con perdida de energía pero también con participación en el render final.

Por último el fotón E sale en dirección a la esfera pero esta lo rebotará hacia el suelo, el cual por razones físicas de su material lo absorberá enteramente. El fotón E no participará en el render.

Por tanto, en general, los fotones se generan en las fuentes de luz y salen en líneas rectas desplazándose por la escena y luego interactuando con las propiedades de los materiales con reflexiones u otras propiedades físicas. Con cada interacción el fotón irá perdiendo energía y a lo largo de unas cuantas interacciones el fotón perderá tanta energía que no será visible. Así sólo los fotones que contengan energía y lleguen a pasar por el pixel y el ojo del observador contribuirán al render final.

Quizás pueda dar la impresión que todo este método explicado hasta ahora sea el ray tracing, pero sólo hemos definido lo que se llama forward ray tracing. Esto es cuando se

sigue el camino de los fotones desde su fuente de luz y se sigue su camino hacia adelante a lo largo de la escena. En el forward ray tracing se sigue el camino del fotón en la dirección que lo lleva hacia adelante.

5.3 Forward y backward ray tracing

La técnica del forward ray tracing describe una primera aproximación a cómo el mundo real funciona. Así mismo es fácil pensar que si simuláramos esto en el ordenador obtendríamos renders muy buenos de la escena, y no pensaríamos mal. Pero hay un problema que nos lleva a buscar otras soluciones y es que para simular el forward ray tracing se necesita un tiempo de cálculo exorbitante. Consideremos que cada luz genera posiblemente millones de fotones cada segundo, que cada fotón vibra a una frecuencia y en una dirección ligeramente diferente. Además cada fotón tendrá un recorrido por la escena diferente, con reflexiones diferentes y quizás podamos llegar a situaciones con un gran número de ellas antes que el fotón se apague o llegue al ojo. Por otro lado si simuláramos el forward ray tracing nos daríamos cuenta que al final, los fotones que llegan al ojo son muy poco comparados con todos los que hemos simulado en trayectorias que no servían para el render. Quizá esperáramos años para renderizar una imagen pequeñísima, eso si, la imagen sería físicamente correcta y muy similar a la realidad.

No es que el forward ray tracing sea malo sino que hemos de calcular muchos fotones que no conllevarán cambios en la escena final.

La solución al problema de la eficiencia computacional está en darle la vuelta al método y seguir los fotones no hacia adelante sino hacia atrás o lo que se llama también backward ray tracing.

El backward ray tracing intentará resolvernos la siguiente pregunta con más sencillez que antes: ¿que fotones contribuyen con certeza a la imagen final?. Por tanto consideremos un pixel dentro del plano de imagen y definamos que camino habrá de realizar un fotón para intervenir en ese pixel. La respuesta es obvia, aquellos que al final atraviesan el plano de imagen y llegan al ojo. Ahora sabemos que el camino tiene un fin en el ojo, que atraviesa el pixel y que pudo haber empezado en cualquier lugar en esa dirección. Esto equivale a la definición de un rayo, una línea que posee un punto de inicio.

Por eso, si algún fotón contribuye en la imagen en un pixel dado, el fotón tendrá que haber llegado en la dirección definida por el rayo con origen en el ojo y que pase por el pixel. Pero, ¿a partir de que objeto proviene el fotón si este ha sido transmitido por medio de reflexiones u otros efectos físicos?. Si seguimos el rayo hasta que interseccionemos con algún objeto habremos hallado el objeto del que procede el fotón.

Ahora seguimos los rayos hacia atrás y esto es muy importante porque restringimos nuestra atención a los rayos que sabemos que pueden ser útiles para nuestra imagen, aquellos que entrarán en nuestro ojo.

Resumiendo, hemos definido el rayo que ha de pasar por el pixel y hemos obtenido el objeto del que puede proceder el fotón que definirá el color del pixel. Hemos dicho puede porque no sabemos si algún fotón llegará al pixel y la respuesta a esta incógnita se resolverá en los apartados que siguen.

Desde ahora en adelante nos limitaremos a llamar por ray tracing al método del backward ray tracing, que es el que vamos a utilizar en la mayoria de casos.

5.4 Intersecciones con rayos

Hemos dado por supuesto que podemos calcular la intersección de un rayo con un objeto pero no hemos explicado cómo ni tampoco hemos relacionado esto con todo lo

explicado en el capítulo 2 sobre modelado de objetos. Esto será muy importante saberlo porque es el primer paso de nuestro proceso de render si utilizamos ray tracing. En este apartado vamos a explicar como se calcula la intersección de rayo con objeto en los tipos de objetos mencionados anteriormente.

El rayo se define como un punto origen y una dirección normalizada. Así la solución a la intersección de ese rayo con un objeto nos tendrá que venir dada por una distancia desde el punto origen del rayo hasta el objeto con el que se intersecciona. Sabiendo la distancia podremos calcular el punto de intersección de una forma directa.

El rayo pues será:

$$R_{origen} = R_o = (x_0, y_0, z_0) \qquad (5.1)$$

$$R_{dirección} = R_d = (x_d, y_d, z_d) \text{ con } x_d^2 + y_d^2 + z_d^2 = 1 \text{ (normalizado)} \quad (5.2)$$

lo cual define a un rayo como:

$$\text{conjunto de puntos en la linea } R(t) = R_0 + R_d * t \text{, donde t>0} \qquad (5.3)$$

con t la distancia desde el origen al objeto a interseccionar.

En el PiscisRT definiremos como veremos con posterioridad dos clases para controlar los rayos: PRTIntersectionPoint y PRTRay.

PRTIntersectPoint contendrá la información de la colisión siendo sus parámetros:

```
bool collision;      //!< There is a collision?.
PRTVector point;     //!< Point of collision.
PRTFloat distance;   //!< Distance from the origin of the ray to the point of collision.
PRTObject* object;   //!< Object in collision.
```

Ilustración 54 - Parámetros de la clase PRTIntersectionPoint.

Y PRTRay definirá un rayo siendo sus parámetros:

```
PRTVector orig;      //!< Origin of the ray.
PRTVector dir;       //!< Direction of the ray.
```

Ilustración 55 - Parámetros de la clase PRTRay.

Pasemos a ver los métodos que se utilizan para calcular las posibles intersecciones con los diferentes tipos de objetos utilizados en el PiscisRT.

5.4.1 Intersección rayo-triángulo

Como avanzamos en el capítulo 2 el triángulo es un polígono de tres vértices y se definirá por esos tres vértices y la normal de la cara. En principio la intersección del rayo con él no necesita de más parámetros aunque si hemos de decidir si se tendrá en cuenta la orientación del triángulo a fin de descartar colisiones cuando el rayo y la normal del triángulo sigan la misma orientación. Es decir, si la normal del triángulo no se contrapone a la dirección del rayo, éste no tendría que resolver ninguna intersección pues el triángulo estaría “de culo”. Matemáticamente, cuando el producto escalar de la dirección del rayo y la normal de triángulo sea mayor o igual a 0, el triángulo será visible al rayo.

Como se nos dice en [9] y está implementado en el PiscisRT, un algoritmo rápido y eficiente en c++ para resolver la intersección rayo-triangulo será:

```
if (!testcull || r.dir*facenormal<=0) //object visible
{
	PRTFloat EPSILON=PRTFloat(0.000001);
	PRTFloat DELTA=-PRTFloat(0.000001);
	PRTFloat LAMBDA=PRTFloat(0.000001);
	PRTVector edge1,edge2,edge3,tvec,pvec,qvec;
	PRTFloat det,inv_det;
	PRTFloat t,u,v;
	edge1=(p2-p1);
	edge2=(p3-p1);
	pvec=r.dir^edge2;
	det=edge1*pvec;
	if (testcull) //testcull active
	{
		if(det<EPSILON)
			return aux;
		tvec=r.orig-p1;
		u=tvec*pvec;
		PRTFloat liminf=(DELTA*det);
		PRTFloat limsup=(det-DELTA*det);
		if (u<liminf || u>limsup)
			return aux;
		qvec=tvec^edge1;
		v=r.dir*qvec;
		if (v<liminf || u+v>limsup)
			return aux;
		t=edge2*qvec;
		if (t<LAMBDA)
			return aux;
		inv_det=PRTFloat(1.0)/det;
		t*=inv_det;
	}
	else
	{
		if(det>-EPSILON && det<EPSILON)
			return aux;
		inv_det=PRTFloat(1.0)/det;
		tvec=r.orig-p1;
		u=(tvec*pvec)*inv_det;
		PRTFloat liminf=DELTA;
		PRTFloat limsup=PRTFloat(1.0)-DELTA;
		if(u<liminf || u>limsup)
			return aux;
		qvec=tvec^edge1;
		v=(r.dir*qvec)*inv_det;
		if(v<liminf||u+v>limsup)
			return aux;
		t=(edge2*qvec)*inv_det;
		if (t<LAMBDA)
			return aux;
	}

	aux.collision=true;
	aux.distance=t;
	aux.point=r.orig+r.dir*t;
}
```

Ilustración 56 - Algoritmo de intersección rayo-triángulo.

siendo r el rayo a estudiar; testcull un booleano que define si queremos testear la orientación de la normal o no; facenormal la normal de cara del triangulo; p1, p2 y p3 los vertices del triángulo; y aux un objeto de la clase PRTIntersectPoint donde se guardará la información de la posible colisión.

5.4.2 Intersección rayo-esfera

La esfera nos servirá como ejemplo para explicar la intersección de un rayo con un objeto paramétrico. No olvidemos que la esfera se define por su centro y su radio y que esos parámetros deben ser necesarios para calcular si hay o no intersección. Un caso

especial de ésta será cuando el rayo atraviese a la esfera pues entonces no habrá sólo una intersección sino dos, la esfera es un objeto cerrado y el rayo lo atravesará entrando por un lado de la esfera y saliendo por otro.

Se nos dice en [5] que la ecuación de la esfera en función del parámetro t y de la definición del rayo será:

$$A*t^2 + B*t + C = 0 \qquad (5.4)$$

siendo:

$$\text{centro de la esfera} = S_c = (x_c, y_c, z_c) \qquad (5.5)$$

$$\text{radio de la esfera} = S_r \qquad (5.6)$$

$$A = x_d^2 + y_d^2 + z_d^2 = 1 \qquad (5.7)$$

$$B = 2*(x_d*(x_o - x_c) + y_d*(y_o - y_c) + z_d*(z_o - z_c)) \qquad (5.8)$$

$$C = (x_o - x_c)^2 + (y_o - y_c)^2 + (z_o - z_c)^2 - S_r^2 \qquad (5.9)$$

A partir de esto, se obtienen dos soluciones para t que serán:

$$t_0 = \frac{-B - \sqrt{B^2 - 4*C}}{2} \qquad (5.10)$$

$$t_1 = \frac{-B + \sqrt{B^2 - 4*C}}{2} \qquad (5.11)$$

Cuando el discriminante o parte dentro de la raíz cuadrada dé negativo el rayo no colisionará con la esfera. Por otro lado si es positivo o 0 estudiaremos las raíces t_0 y t_1 dándose varios posibles casos:

- si una es negativa y la otra positiva, el rayo se inició desde dentro de la esfera y la colisión se produjo de frente con la raíz positiva.
- si las dos son negativas la esfera queda por detrás del inicio del rayo y no interseccionará.
- si las dos son positivas la esfera está por delante del inicio del rayo y colisiona con las dos raíces, en este caso tomaremos la menor raíz positiva entre las dos como punto de intersección más próximo.

Queda claro pues que las raíces t_0 y t_1 nos dan las distancias desde el inicio del rayo a la esfera cuando intersecciona. Una nos dará la distancia hasta la parte más próxima de la esfera y la otra hasta la parte más lejana, lo único que habremos de discernir es cual nos interesa de las dos.

Por tanto, el algoritmo en c++ del cálculo de la intersección será:

```
PRTFloat EPSILON=0.000001;
PRTFloat B=2*(r.dir.x*(r.orig.x-center.x)
                +r.dir.y*(r.orig.y-center.y)
                +r.dir.z*(r.orig.z-center.z));
```

```
PRTFloat C=(r.orig.x-center.x)*(r.orig.x-center.x)
                        +(r.orig.y-center.y)*(r.orig.y-center.y)
                        +(r.orig.z-center.z)*(r.orig.z-center.z)
                        -squareradius;
PRTFloat discr=B*B-4*C;
if (discr<EPSILON) //no collision
        return aux;
PRTFloat sqrtdiscr=PRTSqrt(discr);
PRTFloat t0=((-B)-sqrtdiscr)*0.5000000000000000;
PRTFloat t1=((-B)+sqrtdiscr)*0.5000000000000000;
if (t0<t1 && t0>EPSILON )
{
        aux.collision=true;
        aux.distance=t0;
        aux.point=r.orig+r.dir*t0;
        return aux;
}
else if (t1<t0 && t1>EPSILON)
{
        aux.collision=true;
        aux.distance=t1;
        aux.point=r.orig+r.dir*t1;
        return aux;
}

// we are inside the sphere
if (!testcull)
{
        if (t1>EPSILON)
        {
                aux.collision=true;
                aux.distance=t1;
                aux.point=r.orig+r.dir*t1;
                return aux;
        }
        else if (t0>EPSILON)
        {
                aux.collision=true;
                aux.distance=t0;
                aux.point=r.orig+r.dir*t0;
                return aux;
        }
}
```

Ilustración 57 - Algoritmo de intersección rayo-esfera.

siendo r el rayo a interseccionar; center el centro de la esfera; squareradius el radio de la esfera al cuadrado; y aux el punto de interseción del tipo PRTIntersectPoint.

5.4.3 Intersección rayo-cuádrica

Recordemos que las cuádricas son una definición general de objetos entre los que se encuentra también la esfera. Aún así con la definición de cuádrica podremos simular muchísimos más objetos como los mencionados en su apartado correspondiente dentro del capítulo 2.

Recordemos también que la cuádrica se definía por una matriz de coeficientes de 4 x 4 y que estos se nombraban de la A a la J. Estos coeficientes nos servirán a la hora de calcular la intersección.

Según [5], los coeficientes de la fórmula cuadrática después de resolver la formula para t serán:

$$A_q = A*x_d^2 + 2*B*x_d*y_d + 2*C*x_d*z_d + E*y_d^2 + 2*F*y_d*z_d + H*z_d^2 \quad (5.12)$$

$$\begin{aligned} B_q &= 2*(A*x_o*x_d + B*(x_o*y_d + x_d*y_o) + C*(x_o*z_d + x_d*z_o) \\ &+ D*x_d + E*y_o*y_d + F*(y_o*z_d + y_d*z_o) + G*y_d + H*z_o*z_d + I*z_d) \end{aligned} \quad (5.13)$$

$$C_q = A*x_o^2 + 2*B*x_o*y_o + 2*C*x_o*z_o + 2*D*x_o + E*y_o^2 + 2*F*y_o*z_o + 2*G*y_o + H*z_o^2 + 2*I*z_o + J \quad (5.14)$$

Y a partir de esto y de forma similar al caso de la esfera, se obtienen dos soluciones para t como las que siguen:

$$t_0 = \frac{-B_q - \sqrt{B_q{}^2 - 4*A_q*C_q}}{2*A_q} \quad (5.15)$$

$$t_1 = \frac{-B_q + \sqrt{B_q{}^2 - 4*A_q*C_q}}{2*A_q} \quad (5.16)$$

Si A_q es diferente de 0 entonces seguiremos el mismo racionamiento que la esfera, observando al discriminante y luego hallando la raíz positiva más pequeña.

Si por el contrario A_q es igual a 0, entonces la distancia t hasta la intersección quedará:

$$t = \frac{-C_q}{B_q} \quad (5.17)$$

El algoritmo en c++ quedará por tanto así:

```
PRTFloat AQ=A*r.dir.x*r.dir.x
        +2*B*r.dir.x*r.dir.y
        +2*C*r.dir.x*r.dir.z
        +E*r.dir.y*r.dir.y
        +2*F*r.dir.y*r.dir.z
        +H*r.dir.z*r.dir.z;

PRTFloat BQ=2*(A*r.orig.x*r.dir.x
        +B*(r.orig.x*r.dir.y+r.dir.x*r.orig.y)
        +C*(r.orig.x*r.dir.z+r.dir.x*r.orig.z)
        +D*r.dir.x
        +E*r.orig.y*r.dir.y
        +F*(r.orig.y*r.dir.z+r.dir.y*r.orig.z)
        +G*r.dir.y
        +H*r.orig.z*r.dir.z
        +I*r.dir.z);

PRTFloat CQ=A*r.orig.x*r.orig.x
        +2*B*r.orig.x*r.orig.y
        +2*C*r.orig.x*r.orig.z
        +2*D*r.orig.x
        +E*r.orig.y*r.orig.y
        +2*F*r.orig.y*r.orig.z
        +2*G*r.orig.y
        +H*r.orig.z*r.orig.z
        +2*I*r.orig.z
        +J;

PRTFloat t;
PRTFloat EPSILON=0.000001;
if (AQ>EPSILON || AQ<-EPSILON)
{
        PRTFloat discr=(BQ*BQ)-(4*AQ*CQ);
        if (discr<EPSILON)
                return aux;
        else
        {
                PRTFloat sqrtdiscr=PRTSqrt(discr);
                PRTFloat AQpor2=AQ*2;
```

```
                        PRTFloat t0=(-BQ-sqrtdiscr)/(AQpor2);
                        PRTFloat t1=(-BQ+sqrtdiscr)/(AQpor2);
                        if (t0<EPSILON && t1<EPSILON) //no colision
                                return aux;
                        else
                        {
                                if (t0>EPSILON && t1>EPSILON) //we are outside the quadric
                                {
                                        if (t0<t1)
                                                t=t0;
                                        else
                                                t=t1;
                                }
                                else
                                {
                                        if (t0>EPSILON) //we are inside the quadric
                                                t=t0;
                                        else
                                                t=t1;
                                }
                        }
                }
        }
        else
                t=-CQ/BQ;

        if (t>EPSILON)
        {
                aux.collision=true;
                aux.distance=t;
                aux.point=r.orig+r.dir*t;
                return aux;
        }
```

Ilustración 58 - Algoritmo de intersección rayo-cuádrica.

siendo r el rayo y de A a J los coeficientes de la cuádrica.

5.4.4 Intersección rayo-cuártica

Como ya vimos en el capítulo 2, las cuárticas tendrán grado 4 y nos darán como máximo 4 raízes. Si tomamos el ejemplo del toroide, partimos de su ecuación y substituimos en ella la ecuación del rayo, según [5], obtendremos los siguientes coeficientes de una ecuación de 4º grado:

$$a_4 = (x_d^2 + y_d^2 + z_d^2)^2 \tag{5.18}$$

$$a_3 = 4*(x_o * x_d + y_o * y_d + z_o * z_d)(x_d^2 + y_d^2 + z_d^2) \tag{5.19}$$

$$\begin{aligned} a_2 &= 2*(x_d^2 + y_d^2 + z_d^2)*((x_o^2 + y_o^2 + z_o^2) - (a^2 + b^2)) \\ &+ 4*(x_o * x_d + y_o * y_d + z_o * z_d)^2 + 4*a^2 * z_d^2 \end{aligned} \tag{5.20}$$

$$\begin{aligned} a_1 &= 4*(x_o * x_d + y_o * y_d + z_o * z_d) \\ &*((x_o^2 + y_o^2 + z_o^2) - (a^2 + b^2)) + 8*a^2 * z_o * z_d \end{aligned} \tag{5.21}$$

$$a_0 = ((x_o^2 + y_o^2 + z_o^2) - (a^2 + b^2))^2 - 4*a^2*(b^2 - z_o^2) \tag{5.22}$$

Obteniendo así la ecuación de grado 4:

$$a_0 * t^4 + a_1 * t^3 + a_2 * t^2 + a_3 * t + a_4 = 0 \tag{5.23}$$

Las raíces de la cuártica se podrán extraer de forma analítica. Si nos referimos a [10] obtendremos los métodos para hacerlo en forma de función SolveQuartic. Esta función nos dará el número de raíces encontradas y el valor de cada una, sólo quedará ver cual es la positiva con menor valor como hemos hecho hasta ahora y devolver ese punto como intersección.

La intersección del rayo cuártica quedará en el PiscisRT como sigue:

```
PRTVector despl=-center;
PRTVector origaux=r.orig+despl;
PRTFloat aux0=origaux*origaux;
PRTFloat aux1=r.dir*r.dir;
PRTFloat auxr=squarea+squareb;
PRTFloat auxa2=squarea;
PRTFloat auxb2=squareb;
PRTFloat auxx02=origaux.x*origaux.x;
PRTFloat auxy02=origaux.y*origaux.y;
PRTFloat auxz02=origaux.z*origaux.z;
PRTFloat auxx12=r.dir.x*r.dir.x;
PRTFloat auxy12=r.dir.y*r.dir.y;
PRTFloat auxz12=r.dir.z*r.dir.z;
PRTFloat auxod=origaux*r.dir;
PRTFloat a0=(aux0-auxr)*(aux0-auxr)-4*auxa2*(auxb2-auxz02);
PRTFloat a1=4*(auxod)*(aux0-auxr)+8*auxa2*origaux.z*r.dir.z;
PRTFloat a2=2*aux1*(aux0-auxr)+4*(auxod*auxod)+4*auxa2*auxz12;
PRTFloat a3=4*auxod*aux1;
PRTFloat a4=aux1*aux1;
PRTFloat c[5]={a0,a1,a2,a3,a4};
PRTFloat s[4];

int num=SolveQuartic(c,s);

PRTFloat EPSILON=PRTFloat(0.000001);
PRTFloat fint=PRTINFINITE;
for (int p=0;p<4;p++)
{
        if (num>p)
        {
                if(s[p]>EPSILON && s[p]<fint)
                        fint=s[p];
                else if (testcull && s[p]<-EPSILON) //we are inside
                        return aux;
        }
}
if (fint!=PRTINFINITE)
{
        aux.collision=true;
        aux.distance=fint;
        aux.point=r.orig+r.dir*fint;
        return aux;
}
```

Ilustración 59 - Algoritmo de intersección rayo-cuártica.

siendo center el centro del toroide, y squarea y squareb los cuadrados de a y b.

5.4.5 Intersección rayo-quaternion julia

Hasta ahora hemos visto métodos para calcular intersecciones que matemáticamente son precisos y directos. Llegamos ahora en cambio a un caso donde las intersecciones no se pueden extraer de forma directa sino iterando y aproximando.

En los quaternion julia, hemos dicho que lo único que se puede saber es si un punto pertenece a no al mismo. Del mismo modo según [12] y [13] se podrá obtener un estimador de distancia que nos defina en cada punto a cuanta distancia más o menos se encuentra el fractal.

Ya que no podremos saber exactamente donde colisiona trabajaremos de la siguiente forma:

- Comenzaremos por el inicio del rayo y miraremos el estimador de distancia, moviéndonos luego a esa posición.
- En la nueva posición miraremos si nos hemos metido ya en el fractal y de no ser así repetiremos la operación de movernos donde nos diga el estimador. Si ya estamos dentro de fractal tendremos que volver hacia atrás en el rayo la mitad de la distancia desplazada y volver a iterar.
- Cuando el estimador nos devuelva un valor suficientemente próximo y el punto que calculamos se encuentra cercano a la superficie del fractal, devolveremos ese punto como punto de intersección.

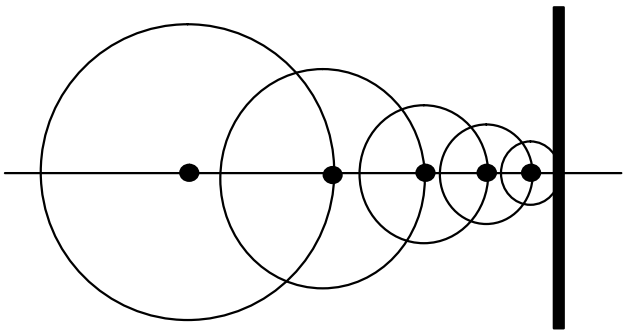

Ilustración 60 - Acercandonos al fractal.

5.4.6 Intersección rayo-csg

En el capítulo 2 hablamos de la técnica de los csg u operaciones booleanas entre objetos básicos. Si sabemos pues cómo interseccionar con estos objetos básicos, ¿cómo encontraremos la intersección con el conjunto global?.

En [5] se nos muestra que la mejor manera será representar un camino de intersecciones o diagrama de raíces. Éste diagrama será una línea entre -∞ y +∞ donde cada punto representa un valor del parámetro t del rayo. Cada punto del diagrama se encontrará bien dentro o fuera del csg y las transiciones entre dentro y fuera las definirá las intersecciones del rayo con las primitivas que forman el sólido csg. Si el csg sólo estuviera formado por un objeto, el diagrama nos definiría los puntos de dentro del objeto mediante aquellos que están comprendidos entre raíces. Si por el contrario el csg lo forman más de un objeto básico hemos de tratar primero el diagrama según lo siguiente:

Operación	Operando 1	Operando 2	Composición
Unión	IN	IN	IN
	IN	OUT	IN
	OUT	IN	IN
	OUT	OUT	OUT
Intersección	IN	IN	IN
	IN	OUT	OUT
	OUT	OUT	OUT
	OUT	IN	OUT
Diferencia	IN	IN	OUT
	IN	OUT	IN
	OUT	IN	OUT
	OUT	OUT	OUT

Ilustración 61 - Reglas de combinación CSG.

Con estas reglas calcularemos un nuevo diagrama de raíces con el que ya podremos intentar buscar la raíz más cercana al origen del rayo y devolverlo como intersección del rayo.

Veamos un ejemplo de diagrama de raíces:

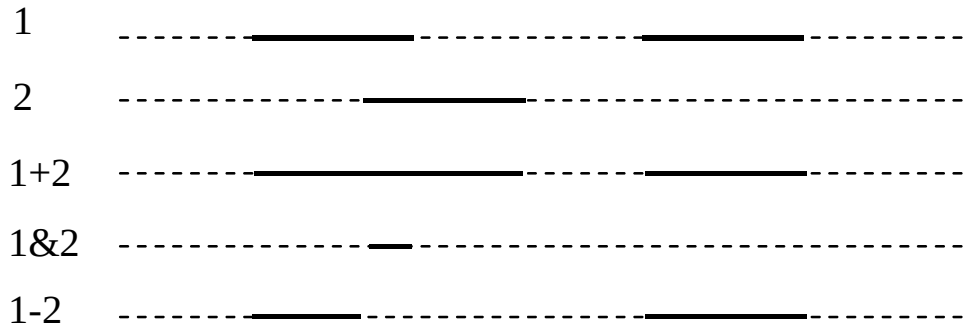

Ilustración 62 - Ejemplo de diagrama de raíces CSG.

A partir de la resolución de estos diagramas ya podríamos obtener la forma del CSG en ese rayo y más específicamente el punto de intersección más próximo al rayo.

5.4.7 Problemas de precisión

Quizá ahora al observar los modelos matemáticos de las intersecciones no nos demos cuanta de una cosa, el ordenador no tiene precisión absoluta. Esto nos llevará a puntos de intersección que no coinciden del todo con la realidad y que aunque sólo tengan un error de alrededor de 10^{-5} pueden llevarnos a singularidades erróneas en la imagen o el render.

Imaginemos que lanzamos un rayo contra una esfera o superficie curvada y nos da un punto de intersección. Ese punto de intersección debido al error producido en el cálculo de intersección puede estar un poco por delante del límite de la superficie o un poco por detrás. Si esta un poco por delante y queremos ahora calcular la refracción en la superficie, ¿no habrá un error al interseccionar otra vez con la misma superficie?. Y si se queda un poco por detrás, ¿no habrá un error al interseccionar también con la misma superficie al intentar calcular la reflexión en la superficie o el rayo hacia la luz?.

Estos posibles errores nos hacen buscar métodos para que el rayo sea lanzado desde una posición segura. Para ello se puede por ejemplo lanzar el rayo desde una posición un poco más adelantada a fin de corregir el posible error. O también se puede al lanzar el rayo otra vez descartar la posible intersección con el mismo objeto. Es decir, si hemos colisionado con un objeto y vamos a calcular el rayo de luz, de sombra, reflexión, refracción, etc..., no hace falta que incluyamos entre los posibles objetos de intersección el mismo que ya tenemos porque eso sería imposible.

En cualquier trazador de rayos se darán estos problemas de precisión igual que en cualquier programa informático también se dan. Pero lo que ocurre en nuestro caso es que un error pequeño de precisión al lanzar rayos puede destrozar todo un render que tarda varias horas por ejemplo en crearse. Un render que plantee alguno de los problemas que hemos definido no se creará bien y puede llegar tener una apariencia irreal. Por tanto tengamos mucho cuidado al tratar zonas del trazador tan delicadas como las intersecciones.

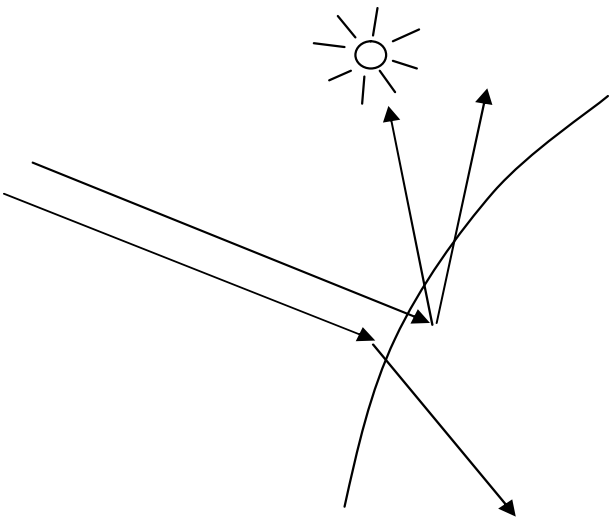

Ilustración 63 - Problemas en la intersección con superficies.

En la ilustración se pueden observar esos casos donde el rayo secundario a lanzar pueda dar resultados erróneos al volver a interseccionar con la misma superficie donde se supone que ya se encuentra el punto origen del rayo.

5.5 Calculando el color del píxel

Vamos ahora a explicar cómo se podrá calcular el color llamado color de cada uno de los píxeles del plano de imagen del render. Supongamos que se nos presenta la pregunta de cómo calcular el color de un píxel p en la posición (i,j) del plano de imagen (i para las ordenadas y j para las abscisas). A partir de ese píxel resolvemos que hemos de lanzar un rayo r en la dirección r.dir.

El primer paso será buscar la intersección más cercana que haya de ese rayo r con la totalidad de objetos en la escena.

Si ésta intersección no existe, podemos deducir que no habrá ningún objeto que haga llegar fotones al píxel p. Por eso, a ese píxel le asignaremos el color que hayamos designado para referirnos al fondo de la imagen. El color que normalmente se usará será el negro como expresión de aquello que no recibe fotones. Aquí terminará el trabajo cuando no hay intersección y se pasará al siguiente píxel a renderizar.

En cambio, si existe una intersección con algún objeto, por lejana que sea, tendremos que aplicar el proceso de coloración del píxel de la forma que sigue:

Llamaremos inter a la intersección calculada con inter.punto el punto de intersección e inter.objeto el objeto con el que se colisiona. Luego pasaremos a obtener el material que posee el objeto inter.objeto y lo designaremos con mat. Si este material nos informa que esta forzado a sólo mostrar el color sin dependencia de la luz, asignaremos tal color al píxel y terminaremos con el render. Si en cambio el material no es forzado necesitaremos aún otro parámetro necesario para seguir con el proceso, ese parámetro es la normal en el punto y se la pediremos al objeto asignándola a normal.

Hasta ahora disponemos pues del punto de colisión inter.punto, la normal en ese punto *normal*, el objeto inter.objeto sobre el que esta el punto y el material del objeto mat. Sin embargo hay uno de estos parámetros que es posible que hayamos de retocarlo. Como ya dijimos en sus momento el material puede disponer de bump map y que este habría de distorsionar la normal, por tanto si mat tiene bump map distorsionamos normal.

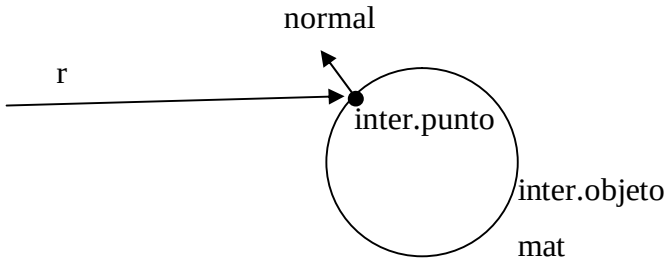

Ilustración 64 - Parámetros después de la intersección.

Llegados a este punto hemos de comprender que posiblemente el material nos informe que hay influencia de rayos propagados en el punto (reflejados, refractados...), de ser así habremos de actuar recursivamente lanzando rayos propagados y guardándonos el color que nos devuelvan para aplicarlos luego. Esta recursión podría llegar a situaciones donde no terminará nunca, por ejemplo cuando dos espejos están uno enfrente de otro y los reflejos no terminan. Por eso lo limitaremos con un número de profundidades llamado deep. Cuando este número sea 0 desecharemos los rayos propagados y aplicaremos en cambio la adición a color del color ambiente del material mat. El cual se explico en el capítulo 3.

Ahora pasaremos a tener en cuenta las luces de la escena y recorreremos todas ellas añadiendo la influencia que produzcan sobre el punto a color. Esta influencia vendrá definida por un nuevo rayo llamado rayo de luz. Cuando hayamos terminado con esto en color se podrá observar pues el color resultante por la actividad de las luces más el color ambiente del material.

Seguiremos por añadir la influencia de los rayos propagados calculados con anterioridad y por último tendremos que añadir, en caso de materiales con textura, el color el píxel de la textura correspondiente al punto inter.punto.

Así , por fin tendremos en color el color que le pertoca al píxel p y habremos de devolverlo para que se le sea asignado.

Veamos la función que hará todo esto dentro del PiscisRT para mayor entendimiento y veamos que realiza exactamente lo relatado hasta ahora:

```
PRTVector PRTRayTracing::RayTrace(PRTRay r,int deep)
{
	PRTVector color(0.0,0.0,0.0);
	PRTIntersectPoint colisiona=main->FindNearestIntersection(r);
	if (colisiona.collision)
	{
		PRTObject* objeto=colisiona.object;

		if (objeto->material->forced)
		{
			color=objeto->material->color;
		}
		else
		{

			PRTVector normal=objeto->ComputeNormal(colisiona.point);

			// si hay bumpmap distorsionamos la normal
			if (main->BBump && objeto->material->bumpmap)
			{
				PRTTexCoord mmm=objeto->ComputeTexCoord(colisiona.point);
				normal=normal+objeto->material->bumpmap->
				CalculateBump(mmm,
					objeto->ComputeTangent(colisiona.point),
					objeto->ComputeBinormal(colisiona.point))*1000;
				normal=normal.Normalize();
```

```
                }

                //si estamos en double sided y la normal esta al reves, invertirla
                if (main->BDoubleSided && r.dir*normal>0) //la normal esta en la misma direccion que el rayo
                        normal=-normal;

                PRTVector reflexion(0.0,0.0,0.0);
                PRTVector refraccion(0.0,0.0,0.0);
                PRTVector transparencia(0.0,0.0,0.0);

                if (deep>0)
                {
                        if (main->BReflections && objeto->material->reflection>0.0)
                                reflexion=main->ComputeReflection(r,colisiona,normal,deep,objeto);
                        if (main->BRefractions && objeto->material->refraction>0.0)
                                refraccion=main->ComputeRefraction(r,colisiona,normal,deep,objeto);
                        if (main->BAlpha && objeto->material->alpha>0.0)
                                transparencia=main->ComputeTransparence(r,colisiona,normal,deep,objeto);
                }
                else
                {
                        color+=objeto->material->ambientcolor;
                }

                PRTListMember *qluzesiterator=main->LightsList.first;
                PRTLight* qluzes;

                while (qluzesiterator!=NULL)
                {
                        qluzes=((PRTLight*)(qluzesiterator->object));

                        PRTVector coloraux(0.0,0.0,0.0);

                        coloraux=qluzes->ComputeLightRay(r,colisiona,normal,objeto,main);
                                //potencia que llega desde la luz al punto

                        coloraux=main->ComputeReflectedLight(
                        colisiona.point,
                        (colisiona.point-qluzes->ComputeWhatPointLight(colisiona.point)).Normalize(),
                        (r.orig-colisiona.point).Normalize(),
                        objeto,
                        normal,
                        coloraux);//cantidad de luz q se ve desde la camara en el punto

                        color+=coloraux;

                        qluzesiterator=qluzesiterator->next;
                }

                PRTFloat refl=0.0; if (main->BReflections) refl=objeto->material->reflection;
                PRTFloat refr=0.0; if (main->BRefractions) refr=objeto->material->refraction;
                PRTFloat alph=0.0; if (main->BAlpha) alph=objeto->material->alpha;

                if (deep>0)
                        color=((color)*(1-alph)*(1-refl)*(1-refr))+(transparencia*alph)
                                +(reflexion*refl)+(refraccion*refr);
                else
                        color=color;

                if (main->BTextures && objeto->material->texture!=NULL)
                {
                        PRTVector janemore;
                        janemore=objeto->material->texture->
                                CalculateColor(objeto->ComputeTexCoord(colisiona.point));
                        color.x=color.x*janemore.x;
                        color.y=color.y*janemore.y;
                        color.z=color.z*janemore.z;
                }

                if (color.x>1) color.x=1; if (color.y>1) color.y=1; if (color.z>1) color.z=1;
                if (color.x<0) color.x=0; if (color.y<0) color.y=0; if (color.z<0) color.z=0;
        }
}
main->numrayos++;
return color;
```

```
}
```

Ilustración 65 - Algoritmo RayTrace del PiscisRT.

Sin embargo no hemos explicado aún cómo funcionan los rayos de luz (ComputeLightRay, ComputeReflectedLight) ni los rayos propagados (ComputeReflection, ComputeRefraction, ComputeTransparence). Veámoslos ahora en los siguientes apartados.

5.5.1 Rayo de luz

Imagínate que estás en la superficie de un objeto y te preguntas si viene luz de algún sitio. La manera más sencilla de saberlo será observar directamente cada una de las luces. Si puedes observar la luz desde el punto en el que estás, o sea, si hay camino directo entre tu y la luz seguro que te llegarán fotones. En cambio si hay algún objeto opaco entre tu posición y la de la luz, no te llegarán fotones.

Ahora recordemos lo que dijimos en el capítulo 4 sobre los modelos de iluminación pues ahora nos servirán para calcular cuanta luz llega al punto a partir de cada una de las luces. En los modelos de iluminación hablamos de atenuación, del BRDF y de los diferentes tipos de luz con sus sombras específicas. Ahora que ya sabemos en qué punto estamos y dónde está la luz apliquemos lo ya conocido.

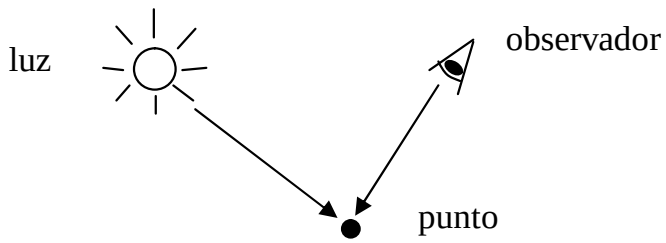

Ilustración 66 - Diagrama del rayo de luz.

La primera parte del trabajo será calcular qué luz llega de forma directa de la luz al punto, por eso utilizaremos la definición de atenuación de la luz y veremos si nos encontramos o no en sombra. Esto se llevará a cabo mediante la función ComputeLightRay de cada tipo de luz, la cual nos devolverá la cantidad de luz o color de la luz que nos llegará. En esta función primero calcularemos la distancia que nos separa de la luz, luego veremos si hay algún objeto que nos haga sombra y si no lo hay atenuará la luz de la forma que lo haría en la realidad para por fin devolverla como resultado.

Veamos el código de esta función en el caso típico de las luces omnidireccionales:

```
PRTVector PRTPointLight::ComputeLightRay(PRTRay &r,PRTIntersectPoint &collision, PRTVector &normal, const PRTObject*
object, PRTMain* main)
{
	PRTVector coloraux(0,0,0);

	PRTVector lightpos=(ComputeWhatPointLight(collision.point));

	PRTFloat collisiontolight=(collision.point-lightpos).Module();

	if ( (collision.point-lightpos).Normalize()*normal/*object->n1*/<=0) //the normal pointing to the light
	{
		// shadow
		PRTVector shadow(1.0,1.0,1.0); // ther isn't shadow, multiplier

		if (main->BShadows)
		{
			//ther is any object between?

			PRTRay rayodesombra(lightpos,(((collision.point)-lightpos)).Normalize());
			PRTListMember *o2=main->ObjectsList.first;
```

```
                        PRTFloat LAMBDA=PRTFloat(0.000001);
                        bool seguroqsi=false;

                        while (!seguroqsi && o2!=NULL)
                        {
                                PRTIntersectPoint sombrainter;//auxiliar collision
                                if (object!=((PRTObject*)(o2->object)))
                                {
                                        sombrainter=rayodesombra.Intersect((PRTObject*)o2->object,
                                                        !main->BDoubleSided);
                                        main->numintertest+=rayodesombra.numrayintertest;

                                        PRTFloat dissombra=(sombrainter.point-lightpos).Module();
                                        if (sombrainter.collision && dissombra<(collisiontolight-LAMBDA))
                                        {
                                                seguroqsi=true;
                                                shadow=PRTVector(0,0,0);
                                        }
                                }
                                o2=o2->next;
                        }
                }

                // ATTENUATION OF LIGHT
                PRTFloat dis=collisiontolight;
                PRTFloat attenuation=1/(attenuation0+attenuation1*dis+attenuation2*dis*dis);
                if (attenuation>1)
                        attenuation=1;

                coloraux=PRTVector(attenuation*shadow.x*color.x*intensity,
                                                attenuation*shadow.y*color.y*intensity,
                                                attenuation*shadow.z*color.z*intensity);

        }
        return coloraux;
}
```

Ilustración 67 - Algoritmo para el cálculo del rayo de luz.

Aún así, eso no es todo. En el modelado de la luz hablamos del efecto que tienen los BRDF de los materiales sobre la distribución de luz que llega al objeto, y esto también hemos de simularlo en nuestro trazador de rayos. Suponiendo que la función BRDF ya está implementada y que disponemos del color que llega a través del rayo de luz directo, veamos que color le pertocará al punto en cuestión dependiendo de donde se encuentre el observador.

```
PRTVector PRTMain::ComputeReflectedLight(PRTVector p,PRTVector in,PRTVector out,PRTObject* objeto,PRTVector nor-
mal,PRTVector power)
{
        PRTVector coloraux(0,0,0);

        coloraux=PRTVector(luzambiente.x*objeto->material->color.x,
                        luzambiente.y*objeto->material->color.y,
                        luzambiente.z*objeto->material->color.z);

        // COMPONENTE DIRECTA
        PRTFloat BRDF=objeto->material->ComputeBRDF(
                                                in,
                                                out,
                                                normal,
                                                objeto->material->BRDFp1,
                                                objeto->material->BRDFp2,
                                                objeto->material->BRDFp3,
                                                objeto->material->BRDFp4,
                                                objeto->material->BRDFp5,
                                                objeto->material->BRDFp6,
                                                objeto->material->BRDFp7);
        PRTFloat coseno=BRDF;

        // CALCULO DE COLOR DIRECTO
        PRTVector colormat;
```

```
	if (!BCInter || !(objeto->type==PRT_TRIANGLE))
		colormat=objeto->material->color;
	else
		colormat=objeto->material->color;
		//colormat=objeto->ComputeColor(p);

	if (BSpecular)
	{
		// COMPONENTE ESPECULAR ¿?
		PRTFloat specular=(-in)*((-out).Reflection(normal/*objeto->n1*/));//.Normalizado();
		if(specular>0) specular=pow(specular,objeto->material->shininess)*objeto->material->specular;
		else specular=0;

		coloraux.x+=((power.x*coseno/*+luzambiente.x*/+specular)*colormat.x);
		coloraux.y+=((power.y*coseno/*+luzambiente.y*/+specular)*colormat.y);
		coloraux.z+=((power.z*coseno/*+luzambiente.z*/+specular)*colormat.z);
	}
	else
	{
		coloraux.x+=((power.x*coseno/*+luzambiente.x*/)*colormat.x);
		coloraux.y+=((power.y*coseno/*+luzambiente.y*/)*colormat.y);
		coloraux.z+=((power.z*coseno/*+luzambiente.z*/)*colormat.z);
	}

	return coloraux;
}
```

Ilustración 68 - Algoritmo para el cálculo de la luz reflejada hacia el observador.

Por lo tanto, si primero calculamos el rayo de luz directo y luego, a partir de este calculamos la luz que se refleja en el punto hacia el observador, tendremos la luz que llega en la dirección correcta hacia el pixel *p*.

5.5.2 Rayos de luz propagados

Ya sólo nos faltaría explicar esas tres funciones capaces de calcular los rayos propagados desde el punto de intersección *inter.punto* y que luego se añadirán al color final bajo los factores definidos en el material para reflexión, refracción y transparencia.

Ya en el capítulo 3 explicamos los principios físicos de las reflexiones, las refracciones y la transparencia. Ahora esos principios físicos nos servirán para calcular la dirección que habrá de tomar el rayo propagado con origen en el punto de intersección *inter.punto*. Este rayo propagado nos devolverá el color que llega por esa dirección. Si los factores que en el material representan estas propagaciones nos revelan que hay que tenerlas en cuenta, las añadiremos según sus valores al color resultante total del píxel.

Así, creemos 3 funciones que sean capaces de lanzar esos rayos de forma correcta y añadamoslas al PiscisRT:

```
PRTVector PRTMain::ComputeReflection(PRTRay r,PRTIntersectPoint colisiona, PRTVector normal,int deep,PRTObject *o)
{
	PRTVector p=r.dir.Reflection(normal);
	if (BGlossy && o->material->glossyper>0.0)
	{
		p=p.Perturb((PRTWRandom()-0.5)*o->material->glossyper,(PRTWRandom()-0.5)*o->material->glossyper);
	}
	return RayTrace(PRTRay(colisiona.point,p),deep-1);
}

PRTVector PRTMain::ComputeRefraction(PRTRay r,PRTIntersectPoint colisiona, PRTVector normal,int deep,PRTObject *o)
{
	PRTVector ret(0,0,0);
	PRTVector normalaux2;
	PRTVector i=(colisiona.point-r.orig).Normalize();
	PRTFloat mu1=0,mu2=0;

	if (deep==2)
	{
```

```
                mu1=1.0;//rayo incidente aire
                mu2=2.0;//transmision cristal
                normalaux2=normal;
        }

        if (deep==1)
        {
                mu1=2.0;
                mu2=1.0;
                normalaux2=normal;
        }

        PRTFloat div=mu1/mu2;

        if (div!=1)
        {
                PRTFloat cosi=normalaux2*-i;
                PRTFloat discr=1+(div*div)*(cosi*cosi-1);
                PRTVector t=(i*div)+(normalaux2*((div*cosi)-PRTFSqrt(discr)));
                //cuando -normalaux2*t no es real -> total internal reflection

                if (discr>=0)
                        return RayTrace(PRTRay(colisiona.point+(t.Normalize()*0.01f),t.Normalize()),deep-1);
        }
        return PRTVector(0.0,0.0,0.0);
}

PRTVector PRTMain::ComputeTransparence(PRTRay r,PRTIntersectPoint colisiona, PRTVector normal,int deep, PRTObject*o)
{
        return RayTrace(PRTRay(colisiona.point+(r.dir*0.01f),r.dir),deep-1);
}
```

Ilustración 69 - Algoritmos para el cálculo de los rayos propagados.

Con esto pues habremos calculado por fin el color que influye en el píxel final por parte de cada una de las luces. Sólo faltará combinar todas las influencias por parte de todas las luces para calcular el color que le pertoca al píxel *p*. Aún así, ¿podemos garantizar que este proceso sea rápido?. En el siguiente apartado vamos a ver técnicas que mejoraran el rendimiento sin variar en nada el funcionamiento básico del ray tracing.

5.6 Métodos para acelerar el ray tracing

Lo primero que es necesario para calcular el color del pixel es calcular la intersección más próxima a algún objeto. Podríamos pensar por tanto que iterando sobre todos los objetos y mirando su intersección con el rayo para luego compararlas todas y quedarnos con la mejor, resolveríamos el problema, y no nos equivocaríamos. Pero hemos de considerar que nuestro trazador ha de ser rápido y que no siempre el número de objetos será reducido en nuestras escenas. Imaginemos que hay miles de objetos, cada vez que queramos calcular la intersección habremos de iterar sobre todos y esto se reflejará en un tiempo de cálculo enorme que a veces hará que nuestro trazador se convierta en algo inútil. Pero no nos bloqueemos aquí, hay métodos que acelerarán estos cálculos de forma abismal.

Además de los que explicaré yo existen muchos otros y siempre será bueno referirse a [5] y [8].

5.6.1 Convex hulls

El primero de los métodos es quizá el más utilizado en todas las aplicaciones de rendering y también en motores de juegos en tiempo real. Se trata de los convex hulls o también llamados bounding volumes. Éstos, se refieren a un objeto más simple que el que nosotros utilizamos, y que garantiza que lo envuelve en toda su totalidad. En la ma-

yoría de casos se utilizarán para esto, formas como cubos orientados en los ejes o esferas.

Consideremos entonces que queremos saber si un rayo colisiona con un objeto bastante complicado y que este objeto tiene asignado otro más simple que lo envuelve, ¿no será mejor mirar primero si colisiona con el objeto simple que lo envuelve y si no colisiona desechar la colisión con el objeto complicado?. Pues así es, primero calcularemos la colisión con el objeto que lo envuelve (convex hull) y sólo pasaremos a mirar la colisión de verdad con el objeto envuelto si el rayo colisiona antes con la envoltura.

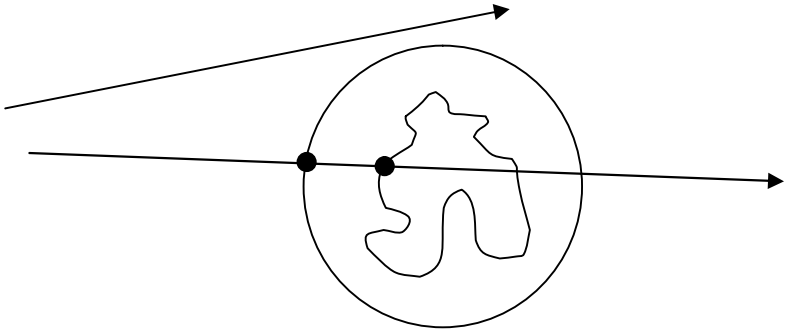

Ilustración 70 - Convex hulls.

Cabe señalar que cuanto más se acerque la envoltura al área del objeto que envuelve mejor, pues se limitarán las situaciones dónde el rayo si colisiona con el convex hull pero no con el objeto en si.

Otra forma de utilizar los convex hulls será la siguiente. Si sabemos que la mejor intersección hasta ahora está a una distancia t y el objeto con el que vamos a comprobar ahora la intersección tiene un convex hull distanciado en su parte más cercana al origen del rayo por una distancia superior a t, entonces no hará falta comprobar la intersección pues de darse siempre estará más alejada que la que ya disponemos.

5.6.2 Octree

Los octrees son un método de subdivisión espacial en 3 dimensiones que definirá la escena entera mediante un árbol en el que los objetos formarán parte de las hojas del mismo.

Así se creará un árbol donde en cada altura se partirá el espacio de la escena en 8 partes, sucesivamente hasta una profundidad. En las hojas de árbol se estudiará que objetos se encuentran dentro y se asignaran a él. En la primera iteración por tanto se partirá en 8 el espacio coincidiendo con los 8 cubos que forman los ejes en 3 dimensiones x, y, z. En la última iteración se habrán obtenido un número conocido de subdivisiones del espacio y sólo nos faltará asignarle a cada una de estas subdivisiones los objetos que se encuentran dentro de ella.

De este modo cuando queramos calcular la intersección más próxima lanzaremos el rayo contra el octree. Cuando se colisione con un nodo, se estudiará si colisiona con sus hijos, por el contrario cuando no se colisione contra un nodo, será imposible que pueda colisionar con los objetos de sus hijos así que se desecharan. Utilizando este método obtendremos una lista de objetos que son posibles de interseccionar por el rayo. El estudio de la intersección más cercana sólo se hará entre los objetos de esta lista.

Un ejemplo de la representación de un octree en 2 dimensiones (quadtree) sería:

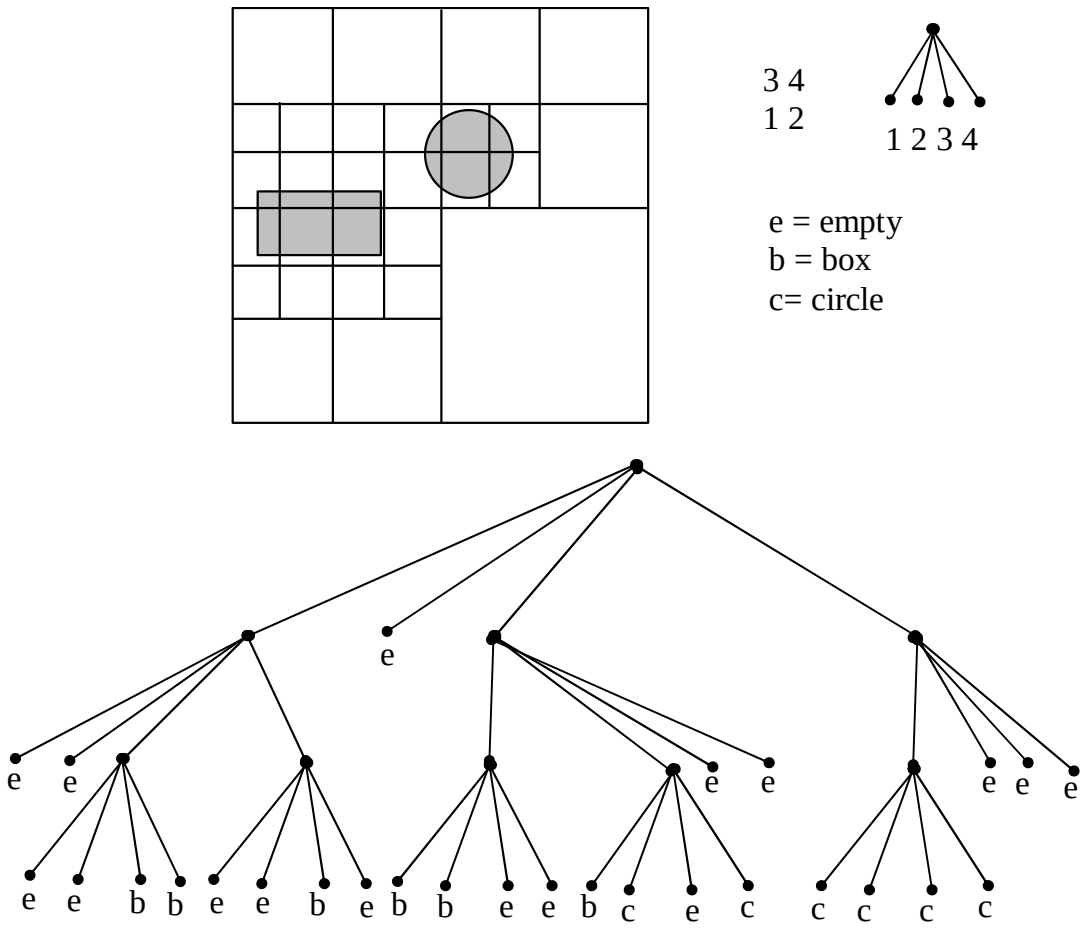

Ilustración 71 - Ejemplo de octree representado en 2 dimensiones (quadtree).

5.6.2 Encontrar la intersección más cercana

Con todos los métodos de aceleración de ray tracing explicados obtenemos pues un nuevo algoritmo para encontrar la intersección más cercana.

En este algoritmo el primer paso será obtener la lista de objetos posibles de intersecccionar ya estén activados los octrees o no. En el caso de no estar activados los octrees esta lista coincidirá con la lista de objetos entera.

El segundo paso será utilizar el convex hull cuando queramos calcular la intersección con el objeto en si.

Por tanto el algoritmo en el PiscisRT quedará de la siguiente forma:

```
PRTIntersectPoint PRTMain::FindNearestIntersection(PRTRay r,PRTObject *d)
{
	PRTFloat DELTA=PRTFloat(0.000001);
	PRTIntersectPoint aux; //colision temporal
	PRTIntersectPoint colisiona; //colision optima, final
	void* opt=NULL; //objeto optimo
	PRTFloat valoropt=PRTINFINITE; //distancia muy grande, para ir guardando la distancia mas corta

	PRTDinamicList* ejem;
	bool borra=false;
	if (!BOctrees || octreesdeep<=0 || Octrees==NULL)
		ejem=&ObjectsList;
	else // tengo en cuenta los octrees
	{
		borra=true;
		ejem=Octrees->ReturnObjects(r,d);
	}
```

```
	PRTListMember *o=ejem->first;
	bool ya=false;
	while (o!=NULL)
	{
		if ((((PRTObject*)(o->object))!=d)
			&& ( !ya
				|| !MejoraPorDistancia
				|| ((Module( ((PRTObject*)(o->object))->convexhull.chcen-r.orig )
					-(((PRTObject*)(o->object))->convexhull.chrad))<valoropt) ) )
		{
			aux=r.Intersect((PRTObject*)(o->object),!BDoubleSided);
			numintertest+=r.numrayintertest;
			if (aux.collision && aux.distance-DELTA<valoropt)
			{
				valoropt=aux.distance;
				opt=o->object;
				ya=true;
			}
		}
		o=o->next;
	}
	if (opt!=NULL)
	{
		colisiona.collision=true;
		colisiona.distance=valoropt;
		colisiona.point=r.orig+(r.dir*valoropt);
		colisiona.object=(PRTObject*)opt;
	}
	if (borra)
		delete ejem;
	return colisiona;
}
```

Ilustración 72 - Algoritmo final para calcular la intersección más cercana.

Capítulo 6: La iluminación global

Cuando vimos el ray tracing elegimos el backward ray tracing por su simplicidad, pero también apuntamos que aquellos fotones que tanto tiempo costaban en calcular debido a sus interacciones con el medio no iban a ser calculados. El no calcular estos fotones nos llevo a una imagen donde no había iluminación procedente de otros objetos y solamente la había por la acción de las luces. Ésta iluminación que nos llegará por fotones transmitidos a nosotros desde otros objetos se llamará iluminación indirecta. Esta iluminación indirecta junto con la que ya teníamos formará lo que se llama iluminación global.

Los modelos de iluminación global consideran por tanto no sólo la luz directa procedente de las luces sino también la luz indirecta. Estos modelos los que se utilizan con frecuencia cuando se quieren obtener imágenes realistas, porque consideran muchos niveles de transmisión de luz. La iluminación global se aproxima mucho más a la realidad que solamente la iluminación directa.

El problema de la iluminación global se resuelve informáticamente simulando la inter reflexión de la luz entre todas las superficies de la escena como se ve en [14]. En ésta tesis se aproxima la iluminación global por medio del método de la radiosidad. Aún así se han presentado muchos otros métodos, variando en velocidad y calidad. De momento se sigue buscando el método que nos sirva para simular la iluminación global de la forma más rápida y real posible.

6.1 La ecuación del rendering

En 1986, Kajiya [15] introdujo la ecuación del rendering. Ésta se creo para: "...crear un contexto único para ver los algoritmos de rendering como aproximaciones más o menos cercanas a la solución de una ecuación.". La idea es que ya que la iluminación local o global describe el mismo fenómeno físico, deben ser capaces de compararse con una ecuación matemática. La ecuación del rendering expresa la luz transportada en un entorno cerrado.

La ecuación del rendering es:

$$L(x,w) = L_e(x,w) + \int_S r(x,w,-w') * L(x',w') * G(x,x') * dA' \qquad (6.1)$$

donde:

- *x,x'* son puntos en superficies del entorno
- *w,w'* son direcciones de salida de la luz en *x,x'*
- *L(x,w),L(x',w')* es la radiación de salida total (reflejada + emitida) en los puntos *x,x'* con direcciones *w,w'* respectivamente
- $L_e(x,w)$ es la radiación emitida en el punto *x* y la dirección *w*
- *r(x,w,-w')* es el BRDF ya comentado en el capítulo 4
- *G(x,x')* es el término geométrico
- *dA'* es la diferencial de área en *x'*
- *S* es el conjunto de superficies que forman el entorno

El término geométrico *G(x,x')* es una de las partes cruciales en la ecuación del rendering porque define la visibilidad:

$$G(x,x') = \frac{V(x,x') * \cos q * \cos q'}{r^2} \qquad (6.2)$$

Donde *q* y *q'* son los ángulos entre las direcciones *w,w'* y las normales a la superficie en *x,x'* respectivamente; *r* es la distancia entre *x* y *x'* y *V(x,x')* es la función de visibilidad, igual a 1 cuando *x* y *x'* se ven mutuamente y igual a 0 en otro caso.

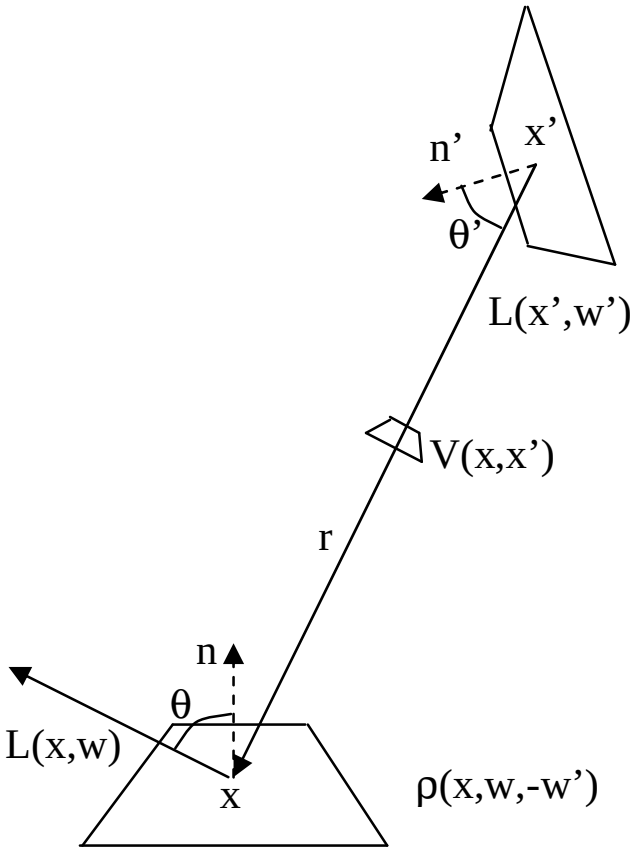

Ilustración 73 - La geometría de la ecuación del rendering.

Comentemos ahora todos aquellos otros términos que son nuevos para nosotros.

6.1.1 Parámetros radiométricos

La radiometría es la ciencia física que se encarga de medir la energía electromagnética [16]. En esta ciencia se utilizan varios parámetros que seguidamente definiremos.

6.1.1.1 Energía radiante

En la radiometría se llama energía radiante a la energía electromagnética o energía lumínica. Su símbolo será *Q* y su medida el Julio *J*.

6.1.1.2 Flujo radiante

El flujo radiante F es la energía radiante que fluye de o hacia una superficie en una unidad de tiempo. Su unidad es el vatio *W* y también es llamada potencia radiante.

$$\Phi = \frac{dQ}{dt} \qquad (6.3)$$

Normalmente sólo distribuciones de energía estática son considerados y por eso energía radiante y flujo radiante se usan como sinónimos.

6.1.1.3 Irradiación o radiosidad

La irradiación E es el flujo radiante incidente por unidad de área.

$$E = \frac{d\Phi}{dA} \qquad (6.4)$$

El contrapuesto *B* será el flujo radiante que sale de la superficie. También llamado radiosidad o radiación saliente.

$$B = \frac{d\Phi}{dA} \qquad (6.5)$$

Tanto la irradiación como la radiosidad son densidades de flujo y se miden mediante:

$$\frac{W}{m^2} \qquad (6.6)$$

6.1.1.4 Intensidad radiante

A veces es conveniente expresar el flujo dependiendo del ángulo sólido, por ejemplo en los casos donde el flujo sale de una luz puntual. Esto se llama intensidad randiante *I*.

$$I = \frac{d\Phi}{dw} \qquad (6.7)$$

La unidad de medida será:

$$\frac{W}{sr} \qquad (6.8)$$

Donde los radianes cuadrados (*sr*) no son una unidad física pero sirve para clarificar. En nuestro caso *dw* denotará un ángulo sólido diferencial sobre un vector director *w* como se verá en el apartado 6.1.1.5.

6.1.1.4.1 Ángulo sólido

La esfera unidad tiene radio 1 y posición en el origen de coordenadas. Una dirección en 3 dimensiones puede ser expresada mediante un punto en la esfera unidad. Por tanto una cierta área de la esfera unidad se traduce en un conjunto de direcciones. Ya que esa área se mide mediante radianes al cuadrado (steradians), la cantidad de estos será el ángulo sólido. Este ángulo sólido no tiene dimensiones. Así el conjunto de todas las direcciones tiene 4π radianes al cuadrado, por el hecho de ser el área de la esfera $4\pi r^2$.

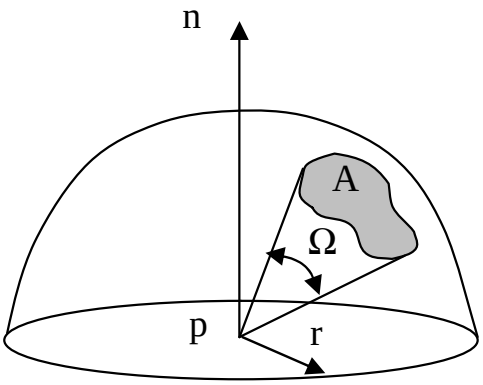

Ilustración 74 - Ángulo sólido.

$$\Omega = \frac{A}{r^2} \qquad (6.9)$$

6.1.1.5 Radiación

La radiación *L* es probablemente el parámetro radiométrico más importante dentro de la síntesis de imágenes. *L* describe el flujo que viaja en un hilo de luz infinitesimal. Se define como el flujo por unidad de área proyectada perpendicularmente al rayo por ángulo sólido en la dirección del rayo que llega o sale de una superficie. Así la radiación *L(x,w)* para un punto *x* en una superficie y dirección saliente *w* es:

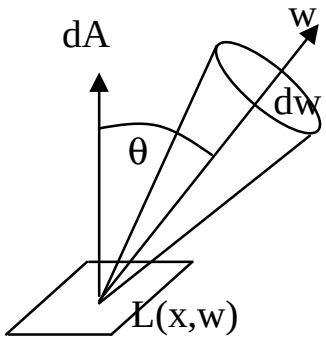

Ilustración 75 - Definición de radiación.

$$L(x,w) = \frac{d^2\Phi}{dw * dA * \cos q} \qquad (6.10)$$

Donde q es el ángulo entre el rayo y la normal de la superficie. Muchas veces *L* será partido en componentes L_i, L_o y L_r para la radiación entrante, saliente y reflejada, o L_e para la componente de radiación propia. La unidad de la radiación será:

$$\frac{W}{m^2 * sr} \qquad (6.11)$$

6.2 Métodos de Monte Carlo

Hemos visto que en la ecuación del rendering y en todas las que se derivan de ella aparecen integrales. El calcular integrales en un ordenador es una tarea bastante costosa y complicad. Tan costosa que de calcularlas exactamente nuestro render se volvería otra vez casi imposible por el tiempo de cálculo. Para eso se utilizan masivamente métodos de Monte Carlo.

En matemáticas, el término Monte Carlo se refiere a métodos que utilizan en la base de sus cómputos números aleatorios de igual forma que en el famoso casino también jugaban un importantísimo papel. Entre los métodos de Monte Carlo también hay soluciones a las integrales. La integración de Monte Carlo es una técnica basada en números aleatorios que nos da una aproximación a la solución de una integral.

Aquí sólo voy a tratar la base de la integración de Monte Carlo y su aplicación a los métodos de iluminación global, para más detalles aconsejo referirse a [17].

Supongamos que queremos calcular la integral *I* de la función *f(x)* sobre el intervalo [0,1]:

$$I = \int_0^1 f(x)dx \qquad (6.12)$$

Para obtener una aproximación sobre la integral tomaremos una muestra x_1 sobre la variable aleatoria *x* que está uniformemente distribuida sobre [0,1], siendo su función de densidad de probabilidad (PDF) $p(x)$=1 en el intervalo. Evaluando $f(x_1)$ obtendremos un primer estimador $<I>_{prim}$:

$$< I >_{prim} = f(x_1) \qquad (6.13)$$

Este valor esperado de este estimador será igual al valor actual de la integral *I*.

$$E(< I >_{prim}) = E(f(x_1)) = \int_0^1 f(x)dx = I \qquad (6.14)$$

Sin duda esta estimación no es muy buena. Para mejorarla, o en términos matemáticos reducir su varianza, podemos tomar más de una muestra. Tomemos *N* muestras independientes x_i sobre la variable aleatoria *x*. Así, ahora tenemos *N* estimadores primarios:

$$< I_i >_{prim} = f(x_i) \qquad (6.15)$$

Para combinarlos rescribiremos la ecuación de la integral original (6.12):

$$I = \int_0^1 f(x)dx = \frac{1}{N}\sum_{i=1}^{N}\int_0^1 f(x)dx = \frac{1}{N}\sum_{i=1}^{N} I_i \qquad (6.16)$$

6.3 Radiosidad

La radiosidad no es sólo una medida radiométrica, sino también el nombre de un algoritmo de iluminación global. Publicado en [14], la radiosidad considera solamente ambientes con un BRDF perfectamente difuso (Lambertian). Así en estos ambientes los BRDF dependen sólo del punto de reflexión *x* y no de las direcciones de luz incidente o saliente. El BRDF se convertirá en constante y nos servirá para simplificar la ecuación del rendering (6.1) en otra ecuación llamada ecuación de la radiosidad.

La nueva ecuación de la radiosidad expresar la radiosidad *B(x)* de cada punto en la escena en lugar de la radiación *L(x,w)*:

$$B(x) = E(x) + \frac{r(x)}{p} \int_S B(x') * G(x, x') * dA' \qquad (6.17)$$

donde:

- *B(x),B(x')* son las radiosidades de los puntos *x* y *x'* respectivamente.
- *E(x)* es la emisión del punto *x*.
- *r(x)* será la reflectáncia del punto *x* (es el nuevo BRDF).
- *G(x,x')* es el término geométrico.
- *dA'* es la diferencial de área en el punto *x'*.

Nótese que en esta ecuación la radiosidad de cada punto se expresa como la suma de la emisión del punto y la multiplicación entre el término que representa la radiosidad incidente (proveniente del resto de puntos de la escena) y la reflectáncia de ese mismo punto. Esto nos da una solución cerrada que sólo funciona en ambientes muy simples. Aún así sigue habiendo una integral a solucionar con métodos numéricos. Pasemos a ver ahora como podemos intentar solucionarla.

6.3.1 Patches

Un primer paso para solucionar la ecuación de la radiosidad (6.17) es considerar elementos finitos: todas las superficies de la escena serán subdivididas en trozos o patches planos y de pequeña extensión. Un caso particular y muy utilizado es que la radiosidad, la reflexión y la emisión serán constantes a lo largo del mismo patch. Esto nos lleva a una versión discreta de la ecuación de la radiosidad (6.17), como se publica en [14]:

$$B_i = E_i + r_i \sum_{j=1}^{N} F_{ij} * B_j \qquad (6.18)$$

donde:

- *Bi es la radiosidad de salida en el patch i.*
- *Ei es la emisión en el patch i.*
- r_i es la reflectáncia del patch *i*.
- *N* es el número de patches.
- F_{ij} es el factor de formación (form factor) del patch *i* al patch *j*.

Así el único parámetro desconocido será a la vez el más crítico de todos, F_{ij}. Hay que saber como calcular los factores de formación (form factors) ya que codifican las necesidades de visibilidad.

6.3.2 Form factors

El factor de formación F_{ij} desde el patch *i* al patch *j* es la fracción de energía que sale desde el patch *i* para ir directamente al patch *j*. F_{ij} se puede representar según [14] como la integral cuadrática:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos q_i * \cos q_j}{pr^2} * V(x_i, x_j) * dA_j * dA_i \qquad (6.19)$$

donde:

- *Ai y Aj son, respectivamente, las áreas de los patches i y j.*
- *qi y qj son los ángulos de la línea que une dAi y dAj en los puntos xi y xj* y sus respectivas normales.
- *$V(x_i,x_j)$* es la función de visibilidad binaria entre *dAi y dAj*.

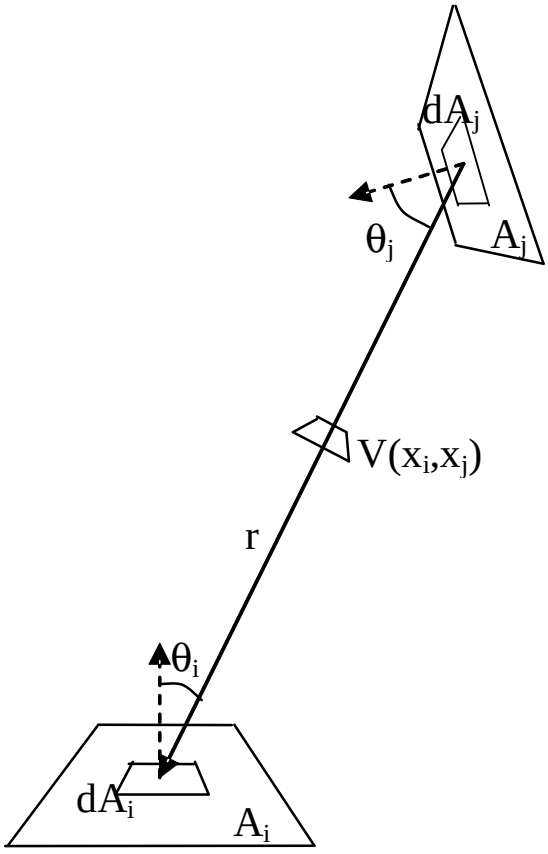

Ilustración 76 - La geometría de los form factors.

Nótese que los form factors dependen sólo de la geometría de la escena. Ahora para intentar simplificar la ecuación, ya que contiene 2 integrales dobles, consideraremos la integración sobre el hemisfereo en lugar de sobre el patch *j*. Como la diferencial del ángulo sólido es:

$$dw = \frac{\cos q_j}{r^2} dA_j \qquad (6.20)$$

tendremos el form factor que va desde el patch de *i* al hemisfereo de *j*:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{\Omega} \frac{\cos q_i}{p} * V_j(x_i, w) * dw * dA_i \qquad (6.21)$$

donde:

- $V_j(x_i,w)$ es la función de visibilidad que indica si el patch *j* es visible desde x_i en la dirección *w*.

Este último form factor nos será de utilidad de una sola vez todos los form factors del patch *i*. Además aún podemos simplificar la ecuación ya que *dw* puede expresarse en coordenadas polares como $sinq*dq*dy$:

$$F_{ij} = \frac{1}{pA_i} \int_{A_i} \int_{q} \int_{y} V(x_i, q, y) * \cos q * \sin q * dq * dy * dA_i \qquad (6.22)$$

donde:

- $V_j(x_i,q,y)$ es la función de visibilidad que indica si el patch *j* es visible desde x_i en la dirección (q,y).

A partir de aquí podemos sacar dos propiedades de los form factores que nos serán de utilidad. Existe una relación de reciprocidad (6.23) y una conservación del flujo de energía (6.24):

$$A_i F_{ij} = A_i F_{ji} \quad \forall (i, j) \qquad (6.23)$$

$$\sum_{j=1}^{N} Fij = 1 \qquad \forall i \qquad (6.24)$$

La relación de reciprocidad nos permite una nueva formulación de la ecuación de la radiosidad basada en potencia en vez de radiosidad. Esto dará la ecuación de la potencia que hace más evidente el sentido físico de los form factors. La ecuación de la potencia multiplica la potencia de salida del patch *j*, P_j por el form factor F_{ji}, dando así la solución a la fracción de potencia que sale del patch *j* y llega a *i*:

$$P_i = \Phi_i + r_i \sum_j F_{ji} * P_j \qquad (6.25)$$

siendo:

- P_i es la potencia de salida del patch *i*, $P_i=B_i*A_i$.
- F_i es la potencia de emisión del patch *i*, $F_i=E_i*A_i$

De todo esto concluimos que no hay una solución única para los form factors excepto para objetos muy simples, por eso se han desarrollado los métodos númericos deterministas. Por ejemplo, en [19] se presenta una aproximación muy utilizada, el método del hemicubo. En este método, el patch *i* es cubierto por un hemicubo que a su vez se divide en pixels. El patch *j* será proyectado a través de este hemicubo. Este método del hemicubo es el que utilizaré yo para implementar el método de radiosidad por patches.

6.3.3 Implementación mediante Hemicubos

Llego la hora de implementar un método que nos represente la iluminación global o más específicamente la radiosidad. Gracias a los hemicubos, desarrollados en [19] y basándonos en el algoritmo propuesto en [20] vamos a poder hacerlo.

Primero de todo hemos de tener algo muy en cuenta, y es que para calcular la radiosidad hemos de eliminar la distinción entre objetos y luces. Al contrario del ray tracing donde objetos y luces eran considerados elementos diferentes. Así, con los objetos emitiendo también luz, todo en la escena ha de ser considerado como una luz potencial. Todo lo que sea visible estará emitiendo o reflejando luz y por eso cuando consideremos la cantidad de luz que llega a alguna parte de la escena tendremos que tener en cuenta la luz que puede llegar desde cualquier componente de la misma. Por tanto:

1- No hay diferencia entre luces y objetos.
2- Cualquier superficie en la escena depende de todas las otras que sean visibles desde ella.

6.3.3.1 Observando el proceso de radiosidad

Ahora pasemos a observar como será el proceso de aplicar radiosidad a la escena. Primero partiremos de una escena simple, por ejemplo una habitación con 3 ventanas. Además añadiremos a la escena algunos pilares para que se puedan observar bien las sombras. Todo en principio será oscuro dentro de la habitación y sólo llegará luz de tipo solar desde el exterior de las ventanas.

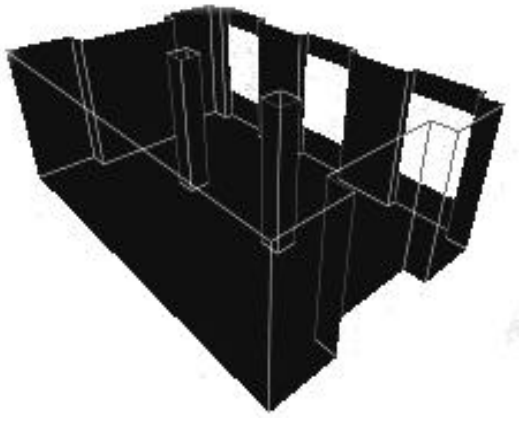

Ilustración 77 - Radiosidad mediante hemicubos, escena inicial.

Ahora observemos con detenimiento alguna de las superficies de la escena y consideremos la luz que le llega.

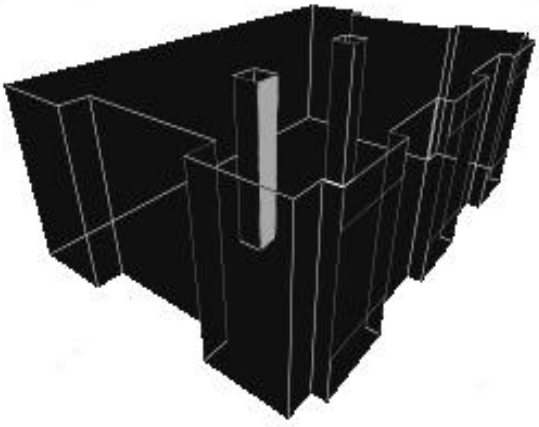

Ilustración 78 - Radiosidad mediante hemicubos, una superficie.

Ahora dividamos la superficie, sea el algoritmo que sea, en pequeños trozos o parches (patches) e intentemos ver el mundo desde el punto de vista de sólo uno de ellos.

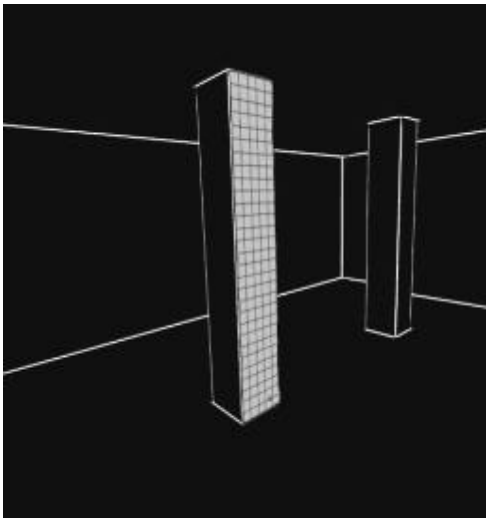

Ilustración 79 - Radiosidad mediante hemicubos, superficie parcheada.

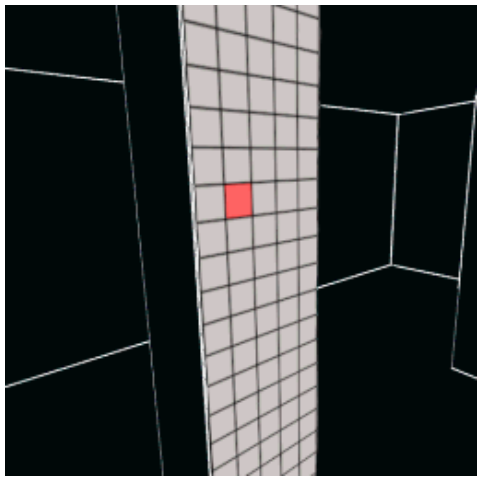

Ilustración 80 - Radiosidad mediante hemicubos, un patch.

Si imaginariamente disponemos nuestro ojo en el patch y miramos hacia el frente podremos ver lo que él ve. La escena es muy oscura porque no ha entrado luz alguna. Ya que no vemos luz alguna podemos afirmar que la luz incidente es nula y por eso el color del patch será nulo o negro.

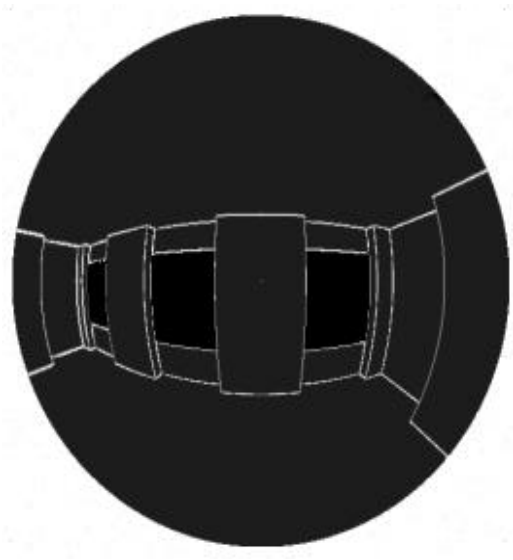

Ilustración 81 - Radiosidad mediante hemicubos, la visión del patch.

Ahora desplacémonos hacia algún patch inferior al anterior y colocada en la base del pilar, el cual si vea directamente al sol. Ya que ve el sol y este es muy brillante la luz incidente tendrá un valor diferente de nulo y por eso el patch tendrá un color más cercano al blanco que el anterior.

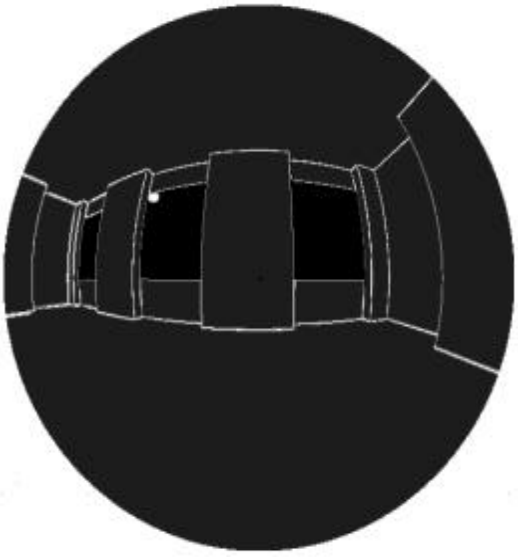

Ilustración 82 - Radiosidad mediante hemicubos, un patch iluminado.

Si seguimos el proceso con todos los patches de la superficie del pilar, podremos ver su iluminación en el primer paso. Los patches que se encuentran cercanos al techo no recibirán luz, mientras que los de la base serán iluminados directamente por el sol.

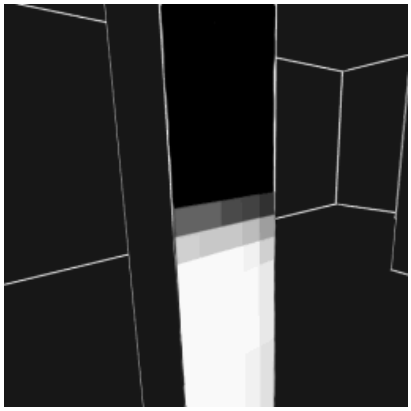

Ilustración 83 - Radiosidad mediante hemicubos, la superficie iluminada.

Para terminar con el primer paso del algoritmo de la radiosidad, continuaremos el proceso con todas las demás superficies de la escena. Así aparecerán iluminados patches que podían ver el sol y aparecerán en sombra aquellos que no podían verlo. Así terminado el primer paso del algoritmo ya no sólo tenemos al solo como fuente de luz sino que todos los patches iluminados se han convertido en también en fuentes de luz. Veamos que ahora la vista de los patches ha cambiado.

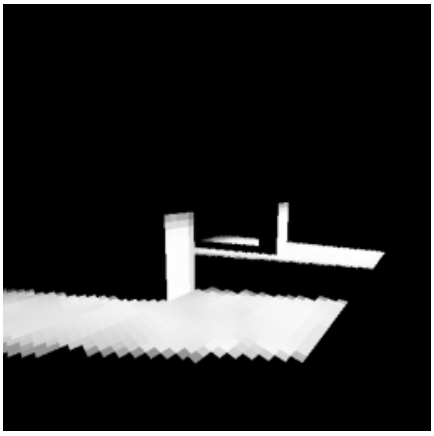

Ilustración 84 - Radiosidad mediante hemicubos, paso 1º.

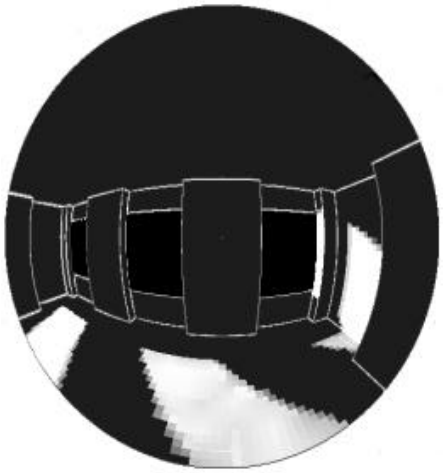

Ilustración 85 - Radiosidad mediante hemicubos, visión despues del primer paso.

Ya que ahora la luz incidente de cada patch ha cambiado, veamos que pasa con las sucesivas repeticiones o pasos del algoritmo de radiosidad. Con el paso 2 ya vemos que esta vez la iluminación de la escena aumenta y parece más realista. Ahora se puede ver el efecto que tiene el que la luz del sol sea reflejada entre las superficies de la escena.

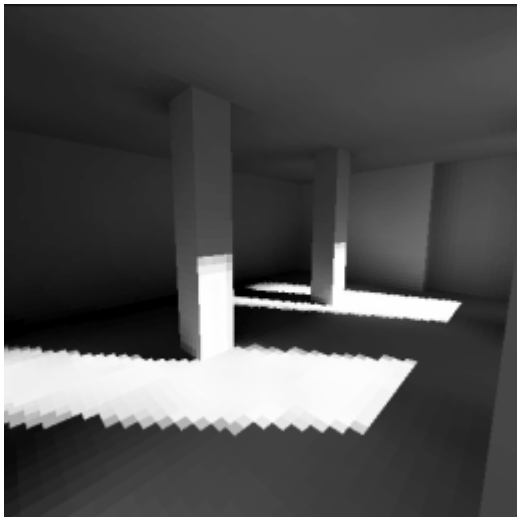

Ilustración 86 - Radiosidad mediante hemicubos, 2º paso.

Con los sucesivos pasos simularemos las siguientes interreflexiones de la luz entre los objetos. Así todo parecerá más o menos igual salvo que la iluminación aumenta un poco más. Llegados al paso 16 la imagen ya no se diferencia de la anterior a simple vista.

El proceso de radiosidad converge pues a una solución. Esta solución será el render que demandábamos y su tiempo de cálculo variará en gran medida dependiendo de la complejidad de la escena, el número de objetos, la posición de las luces iniciales o el tamaño requerido en los patches.

No olvidemos que el tamaño de los patches podrá ser disminuido para obtener una imagen más realista.

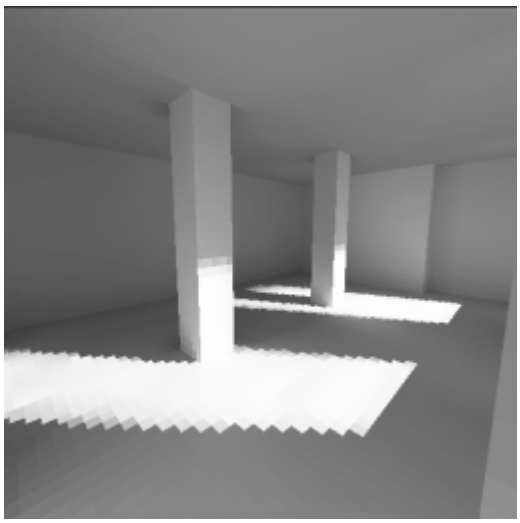

Ilustración 87 - Radiosidad mediante hemicubos, 16º paso.

6.3.3.2 Algoritmos para la radiosidad

En primer lugar definamos los parámetros necesarios que ha de almacenar el patch con el fin de poder llevar a cabo el procedimiento antes expuesto. Un patch así quedará como una estructura que además de contener la posición, tamaños y orientación del patch, almacenará variables numéricas para la emisión, reflectáncia, luz incidente y luz saliente.

La emisión de un patch será aquel parámetro que define si el patch emite luz por si mismo. Es así como en realidad se diferencian las luces de los objetos, las luces tendrán una emisión igual a su color por su intensidad, mientras que los objetos tendrán emisión nula. Esto no se contradice con que todos los objetos pueden ser como fuentes de luz pues los objetos si que podrán reflejar la luz.

La reflectáncia será la proporción de luz que cuando colisiona con un objeto es reflejada por este.

La luz incidente es la suma de todas las luces que el patch puede ver. Esto se completará con la reflectáncia para afirmar que el objeto reflejará una luz igual a la luz saliente. Esta luz saliente será igual a la luz incidente por la reflectáncia del patch.

```
class PISCISRT_API Patch
{
public:
		PRTVector emmision;
		PRTVector excident;
		PRTVector incident;
		PRTVector reflectance;
};
```

Ilustración 88 - Clase Patch del PiscisRT.

De estas estructuras definitorias de los patches saldrá un algoritmo que en pseudocódigo será según [20]:

```
leer la escena y dividir cada superficie en patches de igual tamaño
inicializar los patches:
	para cada patch de la escena
		si el patch es una luz entonces
			patch.emmision = cantidad de luz
		sino
			patch.emmision = negro
		fin si
		patch.excident = patch.emmision
	fin para
bucle:
	para cada patch de la escena
		renderizar la escena desde el punto de vista de este patch
		patch.incident = suma de la luz incidente en el render
	fin para
	para cada patch de la escena
		I = patch.incident
		R = patch.reflectance
		E = patch.emmision
		patch.excident = (I*R) + E
	fin para
	si no hemos hecho suficientes pasos repetir bucle
```

Ilustración 89 - Algoritmo de radiosidad.

Vamos a ver este algoritmo con mayor detenimiento. Al principio todos los patches son negros menos aquellos que emiten luz, por eso estos últimos patches son inicializados en su emisión con esa cantidad de luz. Una vez hecho esto ya podemos pasar a iterar sobre el bucle principal. Este bucle simulará una reflexión de luz en la escena más cada vez que se itere sobre él. El número de iteraciones será pues un parámetro de usuario que definirá en una gran medida la calidad de la radiosidad. Dentro del bucle se simulará el render de lo que ve el patch, la cantidad de luz que le llega a través de ese render y por último la actualización del parámetro de luz saliente.

Podemos observar que una de las partes del algoritmo es, además de complicada de calcular, muy importante para la calidad de la radiosidad. Esta parte es la recolección de luz desde la escena y para ello es para lo que se utilizarán los hemicubos.

6.3.3.3 Algoritmos para los hemicubos

Para la recolección de la luz desde la escena en un patch, lo primero que puede pensarse como solución sería utilizar una vista de tipo ojo de pez. Este tipo de vista podría ser simulado mediante la disposición de una hemiesfera en el patch y renderizando la escena en sobre ella. Entonces, la escena que se vería sobre la hemiesfera sería exactamente lo que se vería desde el punto de vista del patch. Veamos un ejemplo sobre la escena anterior.

Dispondríamos una hemiesfera sobre el patch y reenderezaríamos sobre ella en forma de vista de ojo de pez:

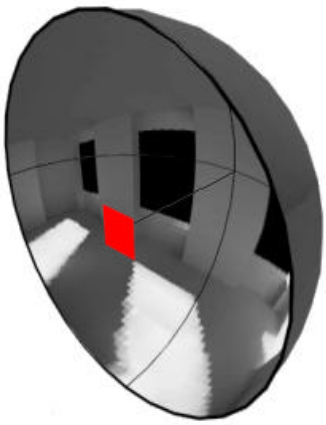

Ilustración 90 - Hemiesfera sobre el patch.

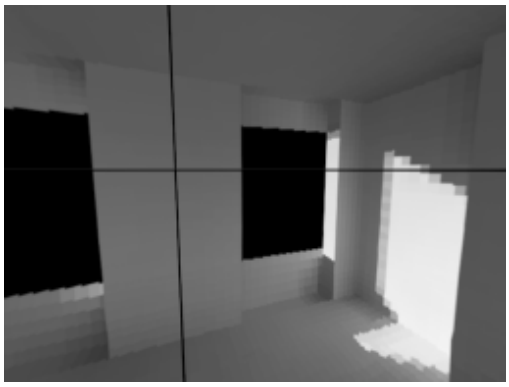

Ilustración 91 - Vista desde el centro de la hemiesfera.

Si pudieramos pues encontrar una manera de renderizar sobre la hemiesfera con una vista de ojo de pez facilmente, entonces sumariamos la iluminación que codifican los píxeles del render sobre la hemiesfera para obtener la luz incidente en el patch. Sin embargo no es fácil renderizar con una vista de tipo ojo de pez. Por eso hemos de buscar otro método más rápido.

Es aquí donde aparece el hemicubo, ya que sorprendentemente, el hemicubo representa exactamente lo mismo desde el punto de vista del patch. Veámoslo con el mismo ejemplo.

Disponemos un hemicubo sobre el patch y renderizamos la escena sobre él en forma de vista perspectiva para cada cara del mismo:

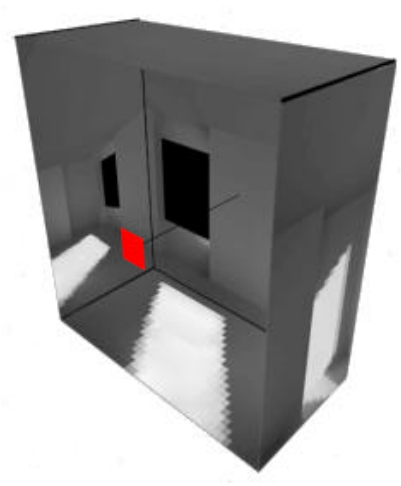

Ilustración 92 - Hemicubo sobre el patch.

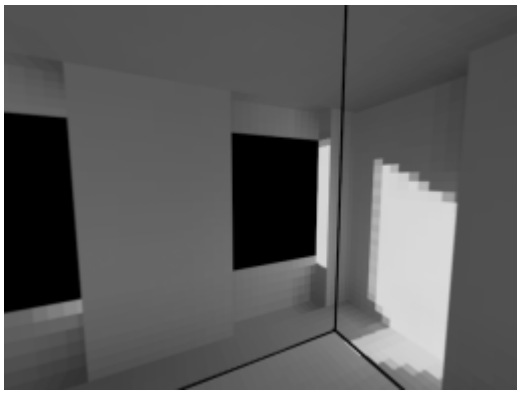

Ilustración 93 - Vista desde el centro del hemicubo.

Por tanto utilicemos el hemicubo para calcular la luz incidente sobre el patch. Este hemicubo estará formado por una imagen cuadrada en el centro y cuatro imagenes rectangulares en los bordes de la cuadrada. La imagen cuadrada representará la vista desde el patch en la dirección de la normal, las otras imágenes rectangulares serán las vistas con un giro de 90 grados hacia arriba, abajo, izquierda y derecha. Por tanto podremos generar estas imagenes situando una cámara en el centro del patch y renderizando hacia adelante, arriba, abajo, izquierda y derecha. Por supuesto, los renders de los lados del hemicubo sólo necesitan la mitad de resolución que el render hacia el frente. Veamos el despiece efectuado sobre el hemicubo:

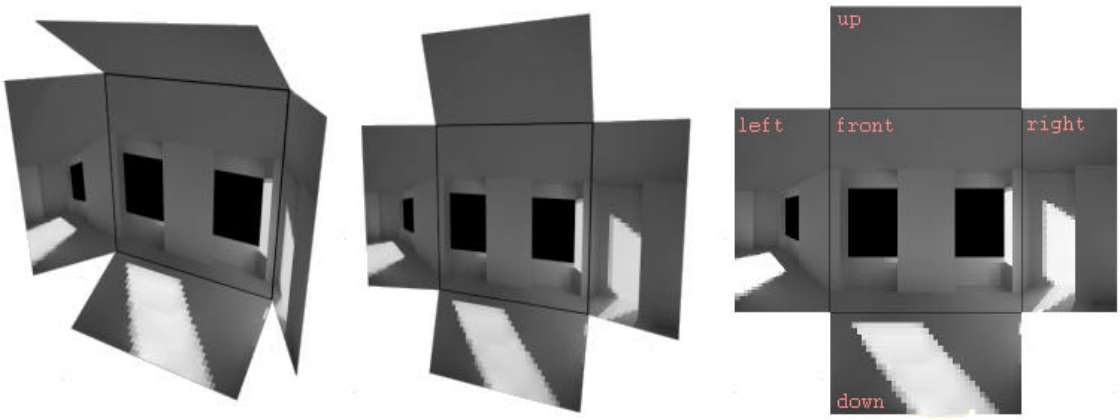

Ilustración 94 - Despiece del hemicubo.

Por tanto tenemos 5 imágenes de distintos tamaños a renderizar en una perspectiva de 90 grados. A partir de estas imágenes renderizadas calcularemos la luz que llega al patch. Pero aún hemos de tener otra cosa en cuenta, y es que debido a la perspectiva de los renders, las imagenes estarán en cierto modo deformadas. Por ejemplo, un render de 3 esferas que están a la misma distancia y se disponen seguidas en horizontal, formarán una imagen donde la esfera del medio parece más redonda y pequeña que las otras dos. Esto nos provoca un problema pues las esferas alargadas de los lados podrían representar más luz para el patch de la que en verdad representan, en el caso que para calcular la luz incidente sumemos los valores de los píxeles del render. Esto habrá que compensarlo de alguna manera.

Para compensar pues que los píxeles más lejanos del centro del render han de contribuir con menos peso, los píxeles de la imagen serán multiplicados por el coseno del ángulo entre la dirección a donde apunta la cámara y la línea desde la cámara al pixel. Esto se podrá ver como otro despiece de hemicubo con la apariencia siguiente:

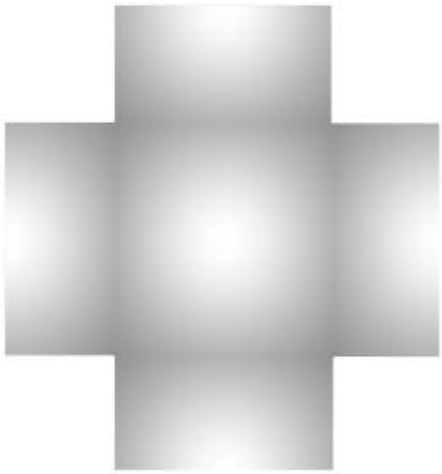

Ilustración 95 - Compensación por perspectiva.

Con esto conseguiremos que toda la visión que se produce desde el patch sea considerada brillante en todas las direcciones por igual, pero esto no ocurre en la realidad, recordemos los BRDF. Hemos de conseguir que el brillo aparente del entorno sea proporcional al coseno entre la normal del patch y la dirección de la luz. Esto se consigue con una nueva compensación que codificará la ley del coseno:

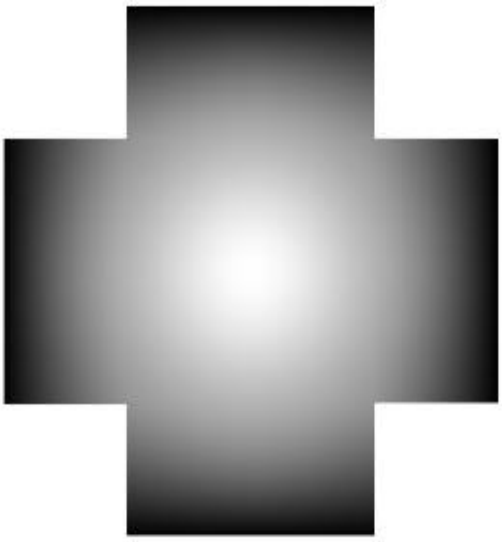

Ilustración 96 - Compensación por ley coseno.

A partir de las dos compensaciones podremos formar por fin un mapa multiplicador (multiplier map) capaz de convertir los renders iniciales en aquellos que nos sirvan para nuestro propósito. Este mapa multiplicador será el resultado de multiplicar la compensación por perspectiva y la compensación por ley coseno, y seguidamente habrá que normalizarlo dividiendo el valor de cada píxel por la suma de los valores de todos los pixeles. Este mapa multiplicador se multiplicará por el valor del hemicubo para dar como resultado los factores que nos servirán para el cálculo de la luz incidente:

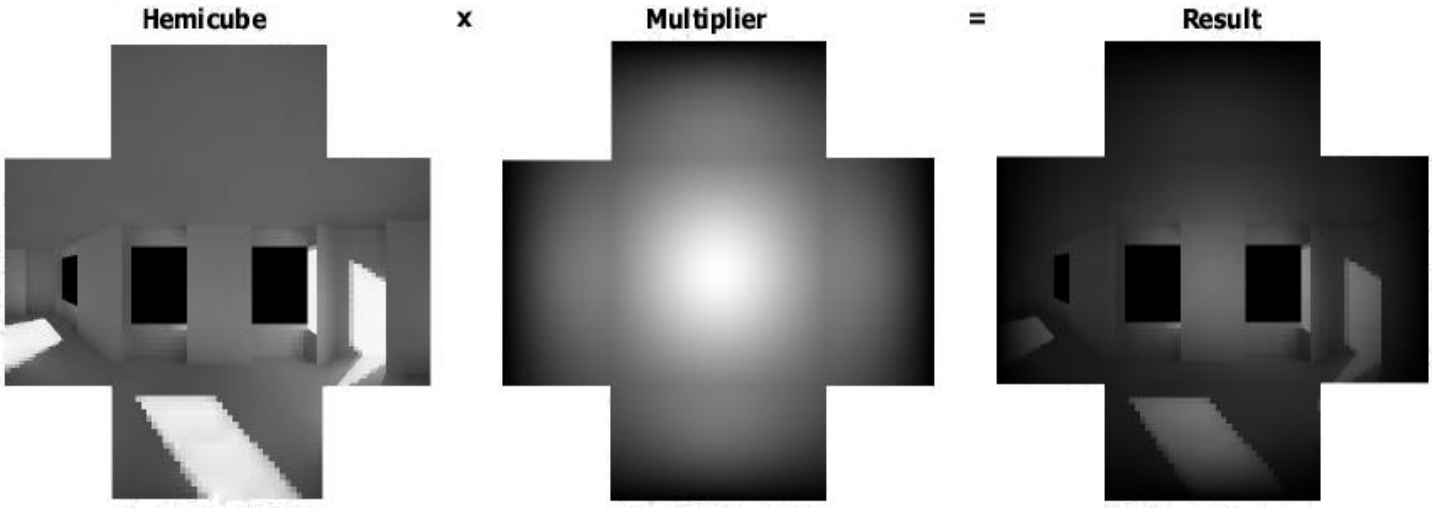

Ilustración 97 - Luz incidete.

Al fin podemos ya crear un algoritmo para calcular la luz incidente. Este algoritmo tomará un punto en 3d o patch y una normal en tal punto. Así, renderizará las 5 caras del hemicubo usando una función *RenderView(point,direction,part)* y las almacenará en una estructura *hemicube* llamada *H* que defina al hemicubo. Seguidamente se multiplicará este hemicubo por el hemicubo multiplicador *M* definido anteriormente dando como resultado al hemicubo *R*. Por fin se sumará el valor total de luz en *R* y se dividirá por el número de píxels que contenga. Este valor será la luz incidente en el patch. El algoritmo queda en pseudocódigo:

```
procedure Calc_Incident_Light(point: P, vector: N)

   light TotalLight
   hemicube H, R, M
   H = empty
   M = Multiplier Hemicube
   R = empty

   div = sum of pixels in M

   camera C
   C.lens = P

   C.direction = N
   H.front = RenderView(C, N, Full_View)

   C.direction = N rotated 90° down
   H.down = RenderView(C, N, Top_Half)

   C.direction = N rotated 90° up
   H.up = RenderView(C, N, Bottom_Half)

   C.direction = N rotated 90° left
   H.left = RenderView(C, N, Right_Half)

   C.direction = N rotated 90° right
   H.right = RenderView(C, N, Left_Half)

   multiply all pixels in H by corresponding
   pixels in M, storing the results in R

   TotalLight = black

   loop p through each pixel in R
     add p to TotalLight
   end loop

   divide TotalLight by div

   return TotalLight
 end procedure
```

Ilustración 98 - Algoritmo para calcular la luz incidente mediante hemicubos.

Con esto, queda explicado pues el hemicubo, su uso para el cálculo de la luz incidente y su uso dentro del método de iluminación global por radiosidad. Pasemos ahora a ver otro método de iluminación global más actual y totalmente distinto al de la radiosidad como es el Photon Mapping.

6.5 Photon Mapping

El concepto de *Photon Mapping* o mapeado de fotones fue introducido por Henrik Wann Jensen en [21] y [22]. Él, igual que nosotros, tuvo la necesidad de un algoritmo que fuera capaz de renderizar imágenes de geometría compleja con iluminación global, un algoritmo que fuera capaz de manejar cualquier tipo de geometría y de *BRDF*.

Ha quedado claro que las técnicas de radiosidad basadas en elementos finitos no satisfacen nuestros requerimientos. Los métodos de radiosidad tienen problemas con los *BRDF* especulares y son demasiado costosos cuando la geometría se complica demasiado.

Alternativas a los elementos finitos aparecieron en forma de técnicas multipaso, *illumination maps* y las técnicas de ray tracing basadas enteramente en *Monte Carlo*. La mejor alternativa sin duda fue la última de ellas pero provocaba cierto ruido en la imagen debido a la varianza de los resultados, además el eliminar este ruido presentaba un sobrecoste demasiado grande.

El *Photon Mapping* fue desarrollado por Jensen como una alternativa a las ténicas de ray tracing basadas enteramente en *Monte Carlo*. El fin era obtener las mismas ventajas pero con un método más eficiente que no sufriera complicaciones por el ruido.

Para ello parece correcto pensar que hay que usar el mismo algoritmo basado en el sampleado por ray tracing. Y además hay que procurar un algoritmo eficiente de transporte de la luz en la escena ya que tanto las luces como el observador son parte primordial de la síntesis de imágenes. También queremos utilizar técnicas de *Monte Carlo* vistas anteriormente pero con la certidumbre de que la componente de radiación se mantiene suave a lo largo de grandes regiones para la mayoría de los modelos. Para estas regiones parece razonable pues almacenar y reutilizar la información extraída sobre su iluminación. Y todo ello sin dividir las superficies en trozos finitos, queremos que nuestro modelo maneje cualquier tipo de objeto.

La primera idea para solucionar el problema es pues la de desacoplar la representación de la iluminación de la geometría. Esto nos permite tanto manejar geometría arbitraria como modelos complejos.

La segunda idea es que la iluminación en la escena puede ser almacenada como puntos en una estructura de datos global, el mapa de fotones o photon map. Muchas alternativas a los puntos fueron consideradas pero todas fallaban en una de las tres condiciones: capacidad de representar cualquier tipo de iluminación, estar desacoplado frente a la geometría y ser compacto. Por eso los puntos son la manera más flexible posible de manejar superficies de cualquier tipo de *BRDF*. Estos puntos además de su posición almacenarán también información sobre la dirección de incidencia de la luz y otros factores que nos facilitarán los cálculos.

El mapa de fotones puede entenderse como una caché de caminos de luces en un *path tracing* direccional y podría ser utilizado ciertamente para tal método. Pero también puede usarse en un método diferente de estimación de la iluminación basada en la estimación de densidad. Esta estimación de densidad tiene la ventaja que su error es de una frecuencia mucho más baja que el que se da en los tradicionales métodos de *Monte Carlo*. Además el método de estimación de densidad es mucho más rápido que un trazador puramente basado en *Monte Carlo*. Sin embargo pagamos el precio de utilizar una estimación de densidad y por tanto un método que no nos dará siempre un render correcto, y que siempre dependerá de que para que converja más a la solución correcta se deberán almacenar y utilizar cuantos más fotones mejor.

Usaremos el nombre *Photon Mapping* para designar el algoritmo que genera, almacena y usa la iluminación como puntos, y el *photon map* como la estructura usada para

procesar estos puntos. Así mismo, *Photon Tracing* será la técnica usada para generar los puntos que representan la iluminación en el modelo.

El método del *Photon Mapping* es un método de dos pasos donde los pasos son:

- *Photon Tracing*, construir el mapa de fotones trazando fotones desde las luces y a través de la escena.
- *Radiance Estimate*, estimar la radiación producida en un punto de la escena mediante estimación de densidad.

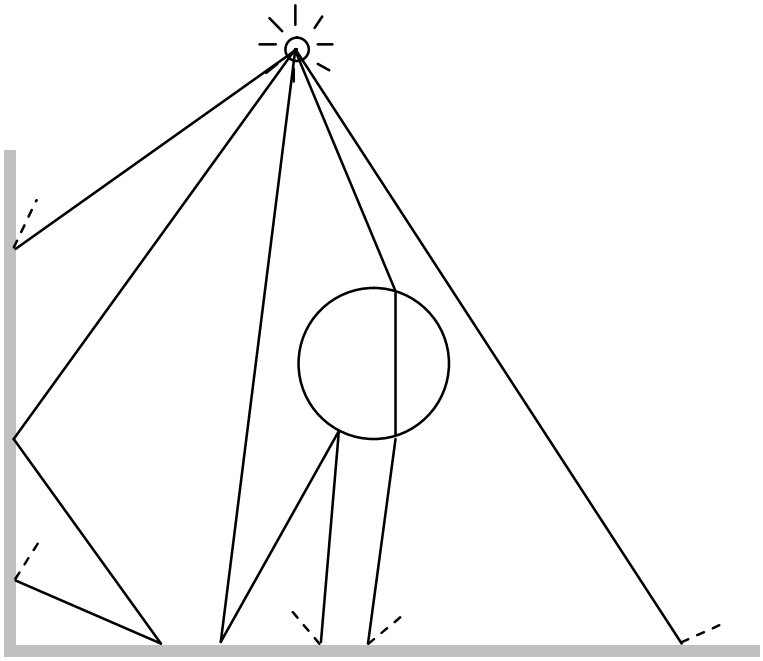

Ilustración 99 - Proceso de creación del photon map.

Así el *photon map* será construido usando *Photon Tracing* donde los fotones son emitidos desde las luces y almacenados al interactuar con las superficies del modelo. Una vez construido el *photon map* lo utilizaremos para calcular la luz radiada.

6.5.1 Photon Tracing

El *Photon Tracing* es el proceso de emitir fotones desde las fuentes de luz y trazarlos a través de la escena. Esta será la técnica usada para construir el *photon map*. Vamos a ver cómo los fotones son generados en las fuentes de luz y cómo podemos seguir su recorrido en la escena de forma eficiente. Siendo esto la base para construir un buen *photon map*.

6.5.1.1 Photon Emission

Vimos en el capítulo 4 que las luces podían ser de muchos tipos y en este apartado veremos como emitir fotones de forma eficiente desde cada una de ellas ya que el *Photon Mapping* admite cualquier tipo de luz.

Así como pasa en la realidad, un enorme número de fotones son emitidos desde cada fuente de luz. La potencia de la luz será dividida entre todos los fotones emitidos por igual, y por eso cada fotón transportará una fracción de la potencia de la fuente de luz inicial. Es importante remarcar aquí que la potencia de los fotones será proporcional al número de fotones emitidos y no al número de fotones almacenados en el *photon map*.

La manera de emitir los fotones desde las luces coincide con lo expuesto en el apartado 4.3. Así la emisión desde una luz puntual será aleatoria en todas las direcciones esféricas alrededor del punto de luz. La emisión en una luz direccional será aleatoria en posición pero con la dirección de la luz. Y la emisión desde luces con formas, por ejemplo esféricas y planares, aleatoria sobre su superficie.

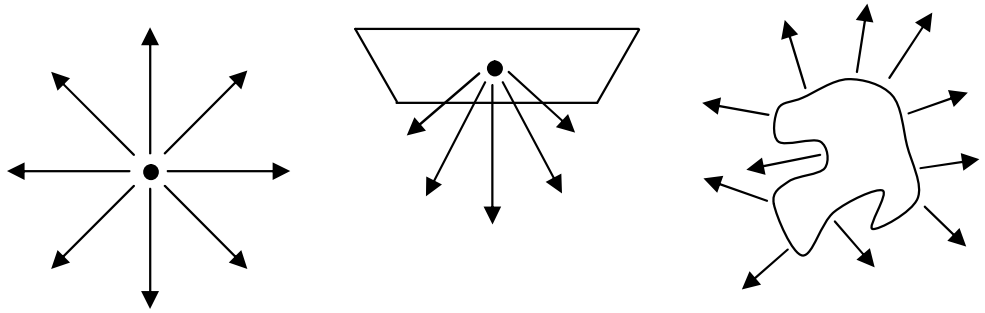

Ilustración 100 - Emisión de fotones desde las fuentes de luz.

6.5.1.2 Photon Scatering

Cuando un fotón colisiona con un objeto, este puede ser con las mismas probabilidades reflejado, transmitido por refracción o absorbido. Que pase una cosa u otra se decide probabilísticamente basándonos en los parámetros del material de la superficie en colisión (capítulo 3). La técnica para decidir el tipo de interacción se conoce como *ruleta rusa*.

La método de la *ruleta rusa* es una técnica estadística que nos servirá para desestimar fotones que no sean importantes a fin de concentrarnos sólo en los que si lo sean. También se usa para asegurarse que los fotones almacenados en el *photon map* tengan aproximadamente la misma potencia de luz. Esto será necesario para una buena estimación de radiación.

La idea básica de la *ruleta rusa* es que podemos tomar muestras aleatorias para eliminar trabajo y aún así obtener un resultado correcto. Es por ello una de las técnicas estándar Monte Carlo.

Con la *ruleta rusa* decidiremos si un fotón en colisión es reflejado o absorbido. Dado un material con reflectividad *d*, y un fotón que colisiona con él con potencia Φ_p:

```
p=d
ξ=random()
if (ξ<p)
        reflejar el fotón con potencia Φp
else
        el fotón es absorvido
```

Ilustración 101 - ¿Reflexión o absorción?.

La idea intuitiva que hay debajo de esto es que si lanzamos 1000 fotones contra una superficie con reflectividad 0.5, podemos bien reflejar los 1000 fotones con la mitad de potencia que tenían antes o reflejar sólo 500 con la potencia intacta. La *ruleta rusa* nos seleccionará esos 500 fotones, reduciéndonos el cómputo requerido por el *Photon Tracing*.

Por tanto a partir de la colisión de un fotón y gracias a la probabilidad lanzaremos otros que le siguen en direcciones de reflexión, refracción o simplemente desecharemos

el fotón porque es absorbido por el material. Sin embargo cada colisión de un fotón provoca que lo almacenemos en el *photon map* como veremos a continuación.

6.5.1.3 Photon Storing

Como hemos mencionado ya, los fotones son almacenados a medida que colisionan contra superficies difusas o más bien no especulares. La razón de esto es que almacenar fotones que colisionan contra superficies especulares o reflectantes, no nos aportan información pues la probabilidad de que un fotón incida por la dirección especular es muy pequeña o cero. Por tanto si queremos simular reflexiones lo mejor será simularlas mediante el ray tracing típico visto en el capítulo 5. Para todas las otras interacciones fotón-superficie, el *photon map* deberá tener información sobre ellas.

Es importante observar que los fotones representan la iluminación (flujo) que llega a las superficies. Esto es una optimización notable que nos da la llave para aproximar la iluminación reflectada en muchos puntos de la superficie.

Veamos como los fotones almacenados en el *photon map* de una escena nos simulan la iluminación y cómo la densidad de fotones es mayor en regiones con fuerte iluminación, sobre todo en la cáustica debajo de la esfera de cristal:

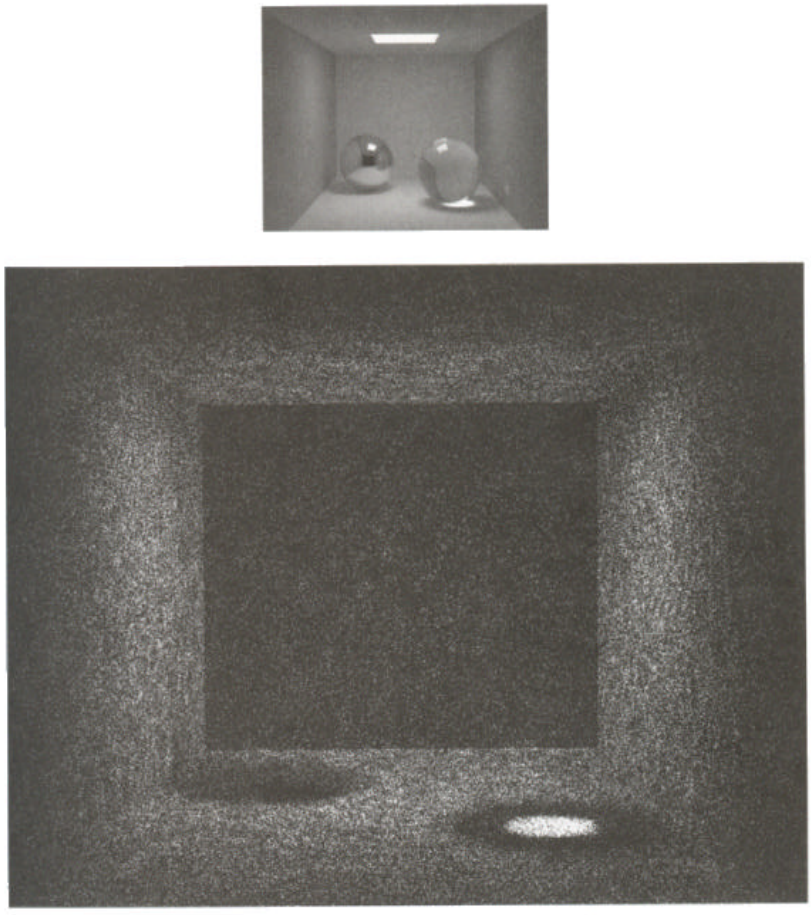

Ilustración 102 - Vista directa del photon map.

Para cada interacción fotón-superficie, serán almacenaos la posición del fotón, la potencia del mismo y la dirección incidente. Veamos como se almacena esto en el PiscisRT:

```
class PISCISRT_API PRTPhoton
{
public:
        PRTFloat pos[3];                    //!< The position of a photon in 3d.
        short plane;                        //!< Axis of balancement.
        unsigned char theta, phi;           //!< Angles of incidence orientation.
        PRTFloat power[3];                  //!< The power of the photon.
};
```

Ilustración 103 - Clase PRTPhoton del PiscisRT.

Dese el momento que queremos que nuestra estructura *photon map* sea util para el algoritmo del *Photon Mapping*, esta deberá ser muy rápida en encontrar los fotones vecinos más cercanos en 3 dimensiones a una posición dada. Para ello Jensen se basó en la estrucutra que vamos a utiliza nosotros también, los Kd-Trees balanceados.

6.5.1.3.1 Balanced Kd-Tree

La complejidad por encontrar un fotón en un *Kd-Tree* balanceado es de O(log *N*), donde *N* es el número de fotones en el *photon map*. Para ver como se gestiona y como se balancea un *Kd-Tree* aconsejo referirse a [23] donde se explica con mayor detenimiento o a [22] donde Jensen propone un algoritmo para los *Kd-Trees* en c.

La eficiencia para encontrar los fotones más cercanos a una posición dad es un punto crítico dentro del *Photon Mapping*. Afortunadamente, la simplicidad de los *Kd-Trees* nos permite implementar un algoritmo sencillo y eficiente de búsqueda para tal fin. Este algoritmo será una extensión directa de los algoritmos de búsqueda binaria estándar.

Para encontrar los vecinos más cercanos en un *Kd-Tree* balanceado, empezaremos por la raíz y añadiremos fotones a la lista de resultados si estos están dentro de una cierta distancia. Para encontrar los *n* fotones más cercanos, la lista de resultados será ordenada como si el que está más lejos pudiera ser eliminado si otro fotón más cercano fuera encontrado. Y así sucederá iterativamente sobre el *Kd-Tree* hasta que encontremos los fotones más cercanos.

Para el algoritmo de búsqueda será necesario que un radio máximo inicial le sea definido con tal de limitar la búsqueda. Un buen radio de búsqueda permitirá al algoritmo realizar una búsqueda óptima reduciendo el número de fotones testeados.

En fin, utilizando el Kd-Tree y un algoritmo de búsqueda eficaz y rápido sobre él, obtendremos la lista de los *n* fotones más cercanos a una posición en 3d dada. Ello nos permitirá pasar a calcular la estimación de radiación sobre ese mismo punto de la forma que veremos a continuación.

6.5.2 Radiance Estimate

La información en el *photon map* puede ser usada para calcular la radiación que sale desde una superficie en una dirección dad. Desde el momento que la dirección de llegada es almacenada con cada fotoón, podremos integrar la información con cualquier BRDF.

Para calcular la radiación L_r, que sale de un punto de intersección *x* en una superficie con BRDF f_r, buscaremos primero los *N* fotones con menor distancia a *x*. Basándonos en la suposición de que cada fotón *p* representa el flujo $\Delta\Phi_p$, que llega a *x* desde la dirección $\Psi_{i,p}$, podremos integrar toda esta información dentro de la ecuación del rendering (6.1) como sigue:

$$L_r(x,y_r) = \int_{\Omega} f_r(x,y_r,y_i) * \frac{d^2\Phi_i(x,y_i)}{dA * dw_i} * dw_i \approx \sum_{p=1}^{N} f_r(x,y_r,y_{i,p}) * \frac{\Delta\Phi_p(x,y_{i,p})}{pr^2} \qquad (6.26)$$

Aquí se usa una aproximación a ΔA donde una esfera centrada en *x* es expandida hasta que contenga a *N* fotones y tenga radio *r*. Luego ΔA será aproximada como πr^2.

El resultado es una ecuación que nos permite computar una estimación sobre la radiación reflejada en cualquier posición de cualquier superficie usando el *photon map*.

$$L_r(x,\vec{w}) \approx \frac{1}{pr^2} \sum_{p=1}^{N} f_r(x,\vec{w}_p,\vec{w}) \Delta\Phi_p(x,\vec{w}_p) \qquad (6.27)$$

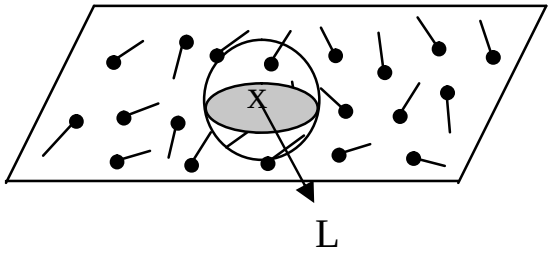

Ilustración 104 - Estimación de L usando el photon map.

Esta será la base de la técnica del *Photon Mapping* y aunque tanto Jensen como otros han hecho pequeñas aportaciones para mejorar el resultado en ciertos casos especiales, la base del método sigue intacta. Por tanto, el *Photon Mapping* será una forma sencilla, rápida y ajustable de calcular cual es la parte de iluminación global que recae sobre una situación dada en 3 dimensiones. Y aunque esto también nos lo podía solucionar la técnica de la radiosidad, el *Photon Mapping* funcionará con cualquier tipo de material y de fuente de luz, y en un tiempo mucho más óptimo que en la anterior técnica.

Por tanto ya tenemos solucionado el problema de la iluminación global, veamos ahora como lo juntamos con el *ray tracing* y la iluminación directa.

6.6 Ray tracing + iluminación global

Vimos en el capítulo 5 como utilizar la técnica del *ray tracing* para resolver el problema de qué color le corresponde a cada píxel de la imagen o plano de visión. Lo resolvimos siempre utilizando la iluminación directa, para reducir el coste de cálculo, y ya vimos por qué. Luego, en el presente capítulo hemos visto métodos para resolver la iluminación global con otro tipo de iluminación, la iluminación indirecta. Por tanto, ahora hemos de ver como poder juntar las dos soluciones de una forma tal que nuestro render contenga una iluminación tanto directa como indirecta. Utilizaremos sin duda el método del *ray tracing* para la directa y cualquiera de los otros dos (aunque el *Photon Mapping* es más rápido) para la indirecta.

El método consistirá en calcular tal y como vimos en el capítulo 5 la iluminación que llega píxel de forma directa pero, añadiendo a la cantidad que nos indiquen los rayos de luz directos, el valor obtenido de la estimación de luz indirecta en el mismo punto de intersección. Es decir, una vez obtenido el punto de intersección lanzaremos rayos de luz directos a todas las luces y además estimaremos la luz indirecta en tal punto por los métodos anteriores de iluminación global, luego sumaremos todos estos valores como la componente de luz en el punto y seguiremos con el proceso de ray tracing.

Con esto conseguiremos un método que contenga lo mejor del ray tracing, con lo mejor de los métodos de iluminación global, en un nuevo método que simulará la iluminación real en una escena dada. Hemos visto también que hemos buscado la forma más rápida y eficaz de computar todas las componentes del método. Por tanto qué menos que implementar todo esto en un programa de síntesis de imágenes, como he hecho yo en mi proyecto y ahora en el siguiente capítulo voy a explicar cómo.

Capítulo 7: Descripción del proyecto

En este capítulo es donde por fin vamos a ver el proceso de creación del proyecto en si. Se utilizarán los conocimientos de los capítulos anteriores para ello y es por eso por lo que se han desarrollado tan ampliamente.

Aquí veremos como se planteó el proceso de creación del proyecto, su planificación y el análisis, diseño e implementación de las partes principales del mismo. De todas maneras para un mayor entendimiento de cómo funciona aconsejo referirse al código del mismo y a su documentación, que dejaré accesible en Internet bajo licencia GPL.

7.1 Introducción

Para la realización del presente proyecto y bajo los objetivos que intento conseguir explicados en el capítulo 1, me basaré en todo momento en la programación orientada a objetos. En mi caso C++ será la base de la programación de todos los componentes del proyecto, principalmente del *PiscisRT* y de los plugins del MAX, ya que la estructura de ambos necesita fuertemente la orientación a objetos.

7.2 Descripción técnica del proceso del proyecto

Vamos a ver ahora cuál fue la descripción técnica inicial que se consideró al empezar el proyecto en si. Esta descripción técnica inicial estará basada en los conocimientos que se tenían al comienzo del proyecto sobre el tema, la estimación de recursos y la planificación de las tareas, y la aplicación práctica que se le quería dar al proyecto.

7.2.1 Información inicial

Para el desarrollo del proyecto, ya se disponía de amplia experiencia en la programación para Windows y linux en C++. Además también se disponía de experiencia en la programación bajo Windows mediante el entorno Visual Studio 6 de Microsoft.

La parte de gráficos y teoría básica de las matemáticas dedicada a ellos se daba por conocida, e incluso la teoría de un trazador de rayos muy básico. En la parte de gráficos del proyecto y sus matemáticas se tenía experiencia por haber estado desarrollando estos últimos años aplicaciones gráficas 2d-3d en tiempo real para usos educacionales, profesionales o por simple curiosidad. Entre estas aplicaciones se encuentran las prácticas de la asignatura informática gráfica, un motor de juegos 3d, un programa para crear paneles de uso cerámico, tratamiento de multirresolución en motores de juegos profesionales, etc. Además en el campo de los trazadores de rayos ya se tenía cierta experiencia cuando hace años intenté crear uno simple junto con un amigo.

En cambio no se tenía experiencia en los métodos de iluminación global ni en la programación de plugins para el 3d Studio Max. Tampoco se tenía experiencia con la librería para la utilización de imágenes jpeg libjpeg ni la que sirve para parsear ficheros de tipo xml xmlparse.

Además cabe destacar los conocimientos que ya se tenían gracias a las asignaturas de la carrera de Ingeniería Informática. Así cabe destacar las asignaturas de tratamiento de estructuras de datos, las matemáticas y físicas, las de programación orientada a objetos, las de sistemas operativos, las de gráficos y las de ingeniería del software.

Así mismo considero una pena que en la carrera no haya ninguna asignatura que se dedique a los trazadores de rayos y toda su teoría, sin duda alguna serviría para que mu-

chos alumnos obtuvieran un mayor conocimiento dentro de la informática gráfica y la síntesis de imágenes.

7.2.2 Estimación de recursos

La estimación de recursos sobre el desarrollo de todos los objetivos puede ser una tarea difícil de acotar pues cualquiera de las tres partes de que consta el proyecto pueden ser mejoradas constantemente. Aún así se considera que el proyecto estará terminado a día de la presentación de esta memoria.

Para ello, se empezó el proyecto a fecha de agosto de 2002 y se termina a fecha de septiembre de 2003, en total 13 meses más o menos. Considerando que se ha dedicado al proyecto 1 hora al día, en total se habrán dedicado unas 400 horas. Nótese aquí que la realización del proyecto en si no se tomó como una obligación sino más bien como un hobbie y de ahí que se hayan dedicado más horas de las estipuladas para la asignatura.

En cuanto a recursos informáticos bastará con un simple ordenador personal, con Windows, Linux, y las librerías externas necesarias para el proyecto. La velocidad del ordenador no nos limitará a la hora de desarrollar el trazador o las otras partes del proyecto. En mi caso utilizaré mi propio ordenador para el desarrollo, siendo este un Pentium II a 233 MHz.

Con todo esto se conseguirá tener terminado el proyecto de la forma que se vio en los objetivos, y además se plantearán sin duda las nuevas bases para las extensiones inmediatamente posteriores a la fecha de entrega del proyecto. Considero a su vez, que continuaré mejorando mi proyecto mientras tenga tiempo para ello pues el tema del mismo es de gran interés para mí.

7.2.3 Planificación de tareas y temporal

Primero de todo habremos de enumerar que partes se pueden extraer de todo el trabajo realizado en el proyecto. A partir de estas partes o tareas podremos hacer un estudio sobre las necesidades temporales que se predicen como necesarias para su realización. Esta estimación temporal depende en gran parte de la estimación de recursos hecha en el punto anterior. Así se observan las siguientes tareas:

- Búsqueda de información asociada al *PiscisRT*.
- Análisis y diseño del *PiscisRT*.
- Implementación del *PiscisRT*.
- Búsqueda de información sobre plugins para 3d Studio Max.
- Análisis y diseño de *PRTMaxPlugin* y *PRTMaxMaterial*.
- Implementación de *PRTMaxPlugin* y *PRTMaxMaterial*.
- Búsqueda de información sobre el uso del *xmlparse*.
- Análisis y diseño del *elements*.
- Implementación del *elements*.

Definamos cada una de estas tareas y estimemos su tiempo necesario de realización.

7.2.3.1 Búsqueda de información asociada al PiscisRT

En esta primera fase nos dedicaremos a buscar información sobre las partes del *PiscisRT* y sus dependencias. Así, hemos de buscar información sobre la librería de tratamiento de imágenes jpeg *libjpeg* a fin de poder utilizar este tipo de imágenes comprimidas en el tratamiento de materiales del *PiscisRT*. Y lo más importante, hay que buscar información sobre muchas de las partes del *PiscisRT* que son lo suficientemente com-

plicadas como para no intentar solucionarlas por uno mismo. Tal es el caso del ray tracing o la iluminación global.

Gran parte de la información necesaria en este punto se ha incluido en la presente memoria a fin de que sea fácil de encontrar para alguien que le interese realizar un proyecto semejante al mío.

Esta tarea es de gran importancia y ya que está en continua actualización, se propone estimarla en la cantidad de tiempo asignada al proyecto en si, 13 meses compartidos con las otras tareas.

7.2.3.2 Análisis y diseño del PiscisRT

El análisis y diseño del *PiscisRT* será la parte donde estructuraremos las clases del *PiscisRT* y su uso. Para ello utilizaremos UML como método de descripción de clases, de sus interacciones y de sus dependencias. Con ello crearemos el esqueleto de lo que habrá de ser nuestro trazador.

Bien es sabido que aunque dediquemos gran parte de tiempo a esto, en algún momento de la implementación es posible que debamos hacer algún retoque. Por tanto dedicaremos el tiempo de análisis y diseño a crear las partes más importantes y relevantes del *PiscisRT* con más dedicación que las otras.

El tiempo estimado para esta tarea será de 2 meses.

7.2.3.3 Implementación del PiscisRT

Una vez realizado el estudio sobre el análisis y el diseño de nuestro trazador *PiscisRT*, pasaremos a implementar las clases que se hayan definido como necesarias. Así implementaremos primero las clases base como son las que gestionan vectores, matrices, funciones matemáticas, listas dinámicas, cuaterniones, etc. Seguidamente a estas se implementarán los tipos de objetos y sus materiales. Será luego cuando empecemos a implementar lo que será la base de nuestro trazador. Cuando el trazador más básico funcione, será cuando iremos añadiéndole las funcionalidades sobre diferentes tipos de luces, diferentes tipos de iluminación, iluminación global, etc.

La implementación del *PiscisRT* será lo que más tiempo dedicará a tiempo preferente, 11 meses.

7.2.3.4 Búsqueda de información sobre plugins para 3d Studio Max

Ya que no se conocía el funcionamiento de los plugins de Max ni su implementación, esta tarea se dedicará a buscar la información necesaria para ello.

Habrá de buscarse información para implementar un plugin de tipo render del Max para desarrollar el *PRTMaxPlugin* y para implementar un plugin de tipo material para el *PRTMaxMaterial*.

A esta tarea se le asigna un tiempo de 1 mes compartido con las otras tareas.

7.2.3.5 Análisis y diseño de PRTMaxPlugin y PRTMaxMaterial

La verdad que debido a que nos hemos de acomodar al diseño que se nos obliga por parte del SDK del Max, esta tarea se verá en cierto modo aliviada. Por tanto hemos de, a partir de la información anterior, realizar un análisis y diseño de como queremos que actúen nuestros plugins acomodándose al estándar de plugins del Max.

El plugin de render, convertirá toda la información poligonal de la escena del Max a triángulos, obtendrá la información de la cámara utilizada y creará la imagen en la pantalla del render del Max de la escena renderizada mediante el *PiscisRT*.

El plugin de material, manejara materiales con los parámetros que hemos definido para los materiales del *PiscisRT* de modo que se ajuste de una forma más estricta a éste.

La tarea se estima en medio mes.

7.2.3.6 Implementación de PRTMaxPlugin y PRTMaxMaterial

Una vez claro cómo hemos de realizar los plugins, sólo falta implementar su código. Una de las cosas que aquí hemos de tener en cuenta es que el 3d Studio Max permite cargar dinámicamente los plugins, pero no descargarlos. Por tanto cada vez que queramos probar algún cambio en nuestros plugins habremos de reiniciar el 3d Studio Max. Esto marcará en cierta medida el tiempo que dedicaremos a esta tarea.

Hay que tener claro que al acoplarse la implementación de los plugins con la implementación del *PiscisRT*, el desarrollo de los plugins dependerán en cada momento de en que momento del desarrollo del trazador estemos.

El tiempo planificado será de 7 meses y medio a tiempo compartido.

7.2.3.7 Búsqueda de información sobre el uso del xmlparse

Para la implementación del programa que parsea un archivo y realiza el render que este define, utilizaremos como lenguaje origen el lenguaje *xml*. El *xml* es el lenguaje base sobre el que se definen por ejemplo las páginas web *html*, y se basa en el uso de etiquetas. Elegiremos pues este lenguaje de *script* para nuestros ficheros *.prt* y ya que no tenemos suficiente conocimiento hasta la fecha en la realización de un buen interprete utilizaremos otro ya hecho, *xmlparse*.

En esta fase buscaremos información para el manejo de la librería *xmlparse* de manera que nos sea útil para nuestro propósito.

El tiempo planificado para la búsqueda de la información es de medio mes.

7.2.3.8 Análisis y diseño del elements

Después de buscar la información sobre como se utiliza el parseador del lenguaje xml, toca analizar y diseñar el programa que a partir de las etiquetas encontradas y de una instancia de la clase PiscisRT sea capaz de crear la escena apropiada, lanzar rayos en las direcciones oportunas y mostrar en una venta la escena final renderizada. Para ello también se utilizará como método de display de la escena la librería de opengl glut. El uso de esta última librería ya era conocido por mí.

El diseño de esta aplicación ha de tener en cuenta que ha de funcionar tanto en plataforma Windows como en plataforma Linux-Unix.

Planificaremos el tiempo necesario para esta tarea en 1 mes.

7.2.3.9 Implementación del elements

Al tener claro como diseñar la aplicación *elements* sólo falta implementarla mediante C/C++. Hay que tener aquí también claro que al acoplarse la implementación del programa *elements* con la implementación del *PiscisRT*, el elements dependerá en cada momento de en que momento del desarrollo del trazador estemos. Tanto es así que sólo funcionará completamente cuando el trazador también esté totalmente completado.

El tiempo planificado para tenrminar esta fase es de 7 meses y medio a tiempo compartido.

7.2.3.10 Estimación temporal total

La estimación temporal de todas las tareas juntas entre el tiempo de inicio a agosto de 2002 y el tiempo de término a septiembre de 2003 puede mostrarse mediante un diagrama como el que sigue:

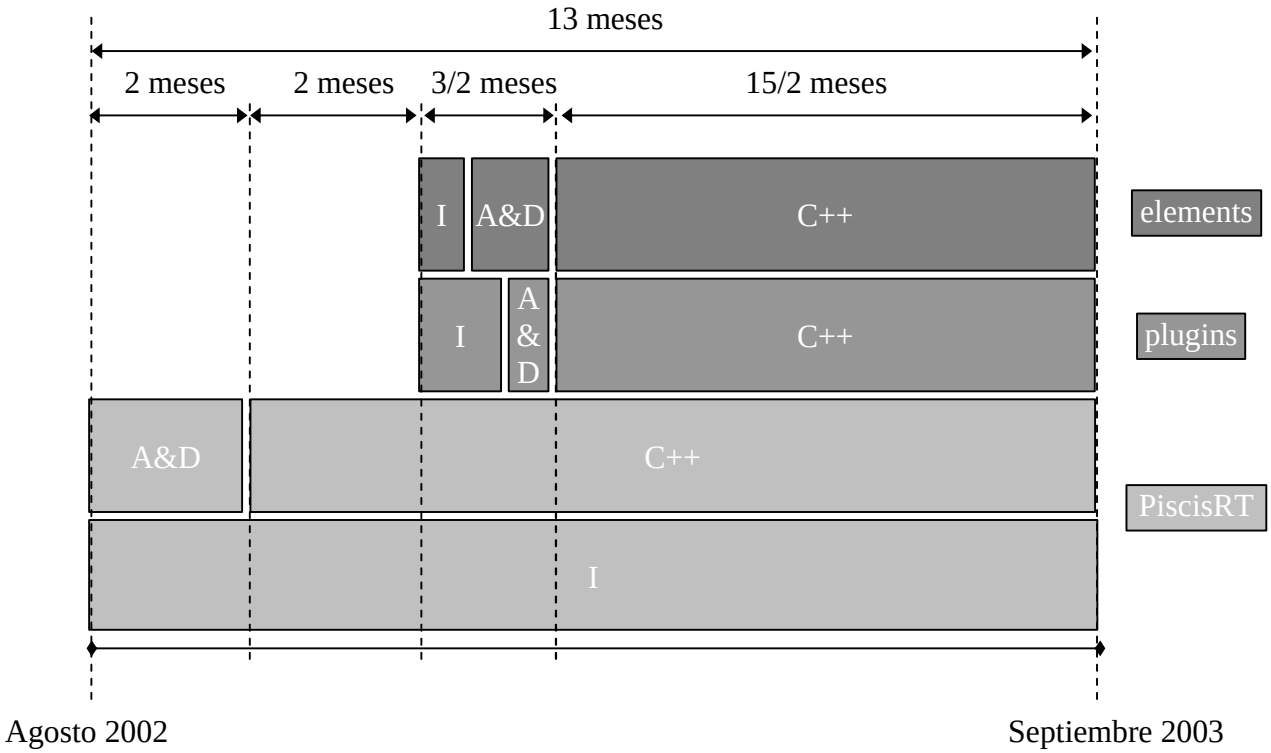

Ilustración 105 - Planificación temporal del proyecto.

La fase de búsqueda de información de cada parte se designa por I, la fase de análisis y diseño por A&D y la de implementación por C++. Además hay que tener en cuenta que existe mucha dependencia e interacción entre las partes del proyecto, por eso se solapan las fases. Empezará la creación del *PiscisRT* con anterioridad y cuando este más o menos funcional empezará las otras dos fases. Al final, todas las partes y fases terminarán en el tiempo de término del proyecto con los objetivos iniciales cumplidos. Sin embargo no se pueden considerar las partes como completamente terminadas y es por eso que a fecha de septiembre de 2003 asumiré el proyecto como en vías de desarrollo y perfeccionamiento. El tiempo libre que tenga a partir de entonces es el que dictará el progreso del trazador *PiscisRT*.

A partir de septiembre de 2003, el camino a seguir estará definido por las posibles extensiones que observe en el proyecto. Estas extensiones serán vistas en el capítulo 9.

7.2.4 Aplicación práctica

Las aplicaciones del trazador *PiscisRT* son muy amplias. Desde el momento que es desarrollado como una librería dinámica y sus dependencias de otras aplicaciones son nulas, el *PiscisRT* está preparado para servir como trazador de rayos a cualquier aplicación que lo necesite.

En el proyecto hemos visto que tanto una aplicación pequeña como el *elements* o un plugin del 3d Studio Max, pueden sacarle jugo al trazador de forma fácil y eficaz.

Sin duda alguna creo que un trazador como el que he realizado con el *PiscisRT* puede servir a cualquier ámbito de la informática, por ejemplo para crear los lightmaps de un motor de juegos, iluminación por vértice realista en una aplicación en tiempo real o cualquier tipo de programa para generar escenas con sintéticas.

Además, ya que todo el proyecto estará accesible bajo licencia GPL, el código podrá servir de ayuda a cualquier persona que esté interesada en el tema que abarco con el mismo.

7.3 Análisis y diseño

Vamos a ver en este apartado como se ha realizado en análisis y diseño de las principales partes del proyecto. No vamos pues a ver todas las partes y por eso aconsejo referirse al código y la documentación del mismo si es que se quiere tener mayor conocimiento en lo referente al análisis y diseño de mi proyecto.

7.3.1 PiscisRT

El proyecto *PiscisRT* estará compuesto por las siguientes clases y en los siguientes ficheros:

Ficheros	Descripción
PRTCircle.h, PRTCircle.cpp	**PRTCircle** – clase que representa un círculo paramétrico mediante su centro, su normal y su radio
PRTCone.h, PRTCone.cpp	**PRTCone** – clase que representa un cono paramétrico mediante su centro, su altura y el radio de su base
PRTConvexHull.h, PRTConvexHull.cpp	**PRTConvexHull** – clase que gestiona los convex hulls de los objetos tanto en forma de cubo orientado en los ejes como de esfera envolvente
PRTCsgObject.h, PRTCsgObject.cpp	**PRTCsgObject** – clase para representar un objeto realizado a partir de una operación csg entre otros dos objetos
PRTCylinder.h, PRTCylinder.cpp	**PRTCylinder** – clase que representa un cilindro paramétrico mediante su centro, su altura y su radio
PRTDinamicList.h, PRTDinamicList.cpp	**PRTDinamicList,** – clase que gestiona listas dinámicas de cualquier tipo de instancias **PRTListMember** – miembro de la lista dinámica
PRTDirectionalLight.h, PRTDirectionalLight.cpp	**PRTDirectionalLight** – clase que representa una luz direccional
PRTImpExp.h, PRTImpExp.cpp	conjunto de estructuras y funciones para hacer posible la creación de plugins de lectura y escritura bajo el PiscisRT
PRTLight.h, PRTLight.cpp	**PRTLight** – clase que sirve de interfaz para todos los tipos de luz
PRTLine.h, PRTLine.cpp	**PRTLine** – clase que representa una línea en tres dimensiones a partir de dos puntos
PRTMain.h, PRTMain.cpp	**PRTMain** – clase principal del PiscisRT, sirve de interfaz hacia todos su funcionamiento, parámetros y funciones
PRTMaterial.h, PRTMaterial.cpp	**PRTMaterial** – clase que representa el material de cualquier objeto
PRTMath.h,	conjunto de definiciones y funciones que nos servirán de base

PRTMath.cpp	para las operaciones matemáticas dentro del PiscisRT
PRTMatrix.h, PRTMatrix.cpp	**PRTMatrix** – clase que define a una matriz 4x4 homogenea, la cual nos servirá para representar rotaciones
PRTObject.h, PRTObject.cpp	**PRTObject** – clase que sirve de interfaz a todos los tipos de objetos aceptados por el PiscisRT
PRTOcTree.h, PRTOcTree.cpp	**PRTOcTree,** – clase que gestiona la utilización de un OcTree dentro del PiscisRT **PRTOcTreeNode** – nodo del árbol OcTree
PRTPatches.h, PRTPatches.cpp	**PRTPatches,** – clase que representa el tipo de render por radiosidad mediante patches **PRTObjectPatches,** – clase que define los patches de cada objeto **PRTPatch** – clase que define un patch
PRTPhotonMap.h, PRTPhotonMap.cpp	**PRTPhotonMap,** – clase que define un photon map **NearestPhotons,** – clase que representa la lista de fotones más próximos a una posición en 3 dimensiones **Photon** – clase que define un fotón
PRTPhotonMapping.h, PRTPhotonMapping.cpp	**PRTPhotonMapping** – clase que representa el tipo de render por Photon Mapping
PRTPlane.h, PRTPlane.cpp	**PRTPlane** – clase que representa un plano infinito mediante un punto del plano y la normal del mismo
PRTPlugin.h, PRTPlugin.cpp	**PRTPlugin,** – clase interfaz para los plugins de PiscisRT **PRTPluginImportar** – clase interfaz para los plugins de importación de geometría del PiscisRT
PRTPointLight.h, PRTPointLight.cpp	**PRTPointLight** – clase que representa una luz puntual a partir de la posición de la misma
PRTQuadric.h, PRTQuadric.cpp	**PRTQuadric** – clase que representa el tipo de objeto cuádrica
PRTQuartic.h, PRTQuartic.cpp	**PRTQuartic** – clase que representa el tipo de objeto cuártica
PRTQuaternion.h, PRTQuaternion.cpp	**PRTQuaternion** – clase para utilizar cuaterniones en el proyecto
PRTQuatJulia.h, PRTQuatJulia.cpp	**PRTQuatJulia** – clase para especificar el tipo de objeto fractal de quaternion julia
PRTRandom.h, PRTRandom.cpp	**PRTRandom.h,** – clase interfaz para los generadores de números aleatorios **PRTRandomMersenne,** – generador de números aleatorios de tipo Mersenne twister **PRTRandomMotherOfAll,** – generador de números aleatorios de tipo Mother of All **PRTRandomBGenerator,** – generador de números aleatorios de tipo ranrot B **PRTRandomWGenerator** – generador de números aleatorios de tipo ranrot W
PRTRay.h, PRTRay.cpp	**PRTRay,** – clase que representa un rayo **PRTIntersectPoint** – clase que representa un punto de intersección
PRTRayTracing.h, PRTRayTracing.cpp	**PRTRayTracing** – clase que define el tipo de render por ray tracing
PRTRender.h,	**PRTRendern** – clase interfaz para los tipos de render, ray tra-

PRTRender.cpp	cing, patches o photon mapping
PRTSphere.h, PRTSphere.cpp	**PRTSphere** – clase que representa una esfera paramétrica mediante su centro y su radio
PRTSphereLight.h, PRTSphereLight.cpp	**PRTSphereLight** – clase que define una luz en forma esférica
PRTTexCoord.h, PRTTexCoord.cpp	**PRTTexCoord** – clase que representa una coordenada de textura u,v
PRTTexture.h, PRTTexture.cpp	**PRTTexture** – clase que encapsula una textura, tanto de tipo tga como de tipo jpg
PRTThread.h, RTThread.cpp	**PRTThread** – clase que representa un hilo de ejecución, se asocia con una función
PRTTriangle.h, PRTTriangle.cpp	**PRTTriangle** – clase que representa un triángulo en 3 dimensiones mediante la posición de sus 3 vértices y sus normales
PRTTriangleLight.h, PRTTriangleLight.cpp	**PRTTriangleLight** – clase que define un tipo de luz triangular
PRTUtil.h, PRTUtil.cpp	conjunto de definiciones y funciones dependientes de la arquitectura en la que se compila el PiscisRT
PRTVector.h, PRTVector.cpp	**PRTVector,** – clase que representa un vector en 3 dimensiones x,y,z **PRTVector4** – clase para los vectores de 4 componentes x,y,z,w

Siendo las clases más importantes las clases PRTMain, PRTObject, PRTLight y PRTRender. Por eso vamos a verlas ahora con más detenimiento.

7.3.1.1 Clase PRTMain

El diagrama de colaboración de la clase PRTMain es:

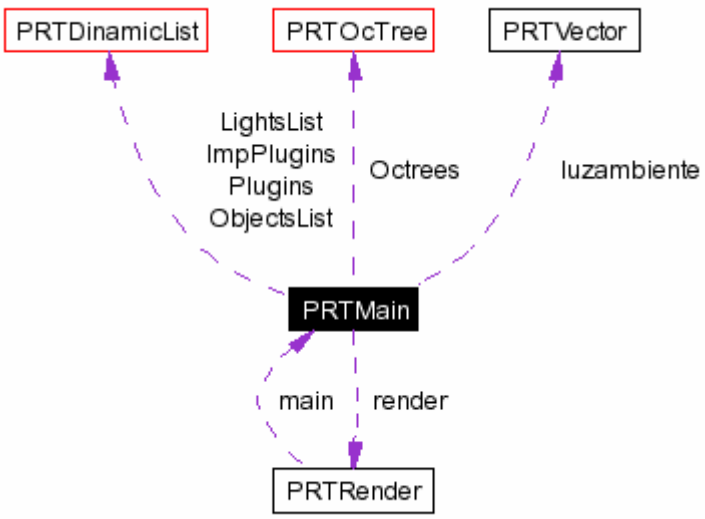

Ilustración 106 - Diagrama de colaboración de PRTMain.

La clase PRTMain será a todas luces la clase principal del trazador de rayos. En ella se contempla el análisis principal del trazador y de su funcionamiento de cara al usuario. Nos servirá como interfaz al PiscisRT siendo sus funciones y atributos:

Public Member Functions

- PRTMain (int type=PRT_RAYTRACING)
  A constructor of the PRTMain.
- void AddTriangle (PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTMaterial *)
  Add one triangle to the scene.
- void AddTriangle (PRTVector, PRTVector, PRTVector, PRTMaterial *)
  Add one triangle to the scene.
- void AddTriangle (PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTMaterial *)
  Add one triangle to the scene.
- void AddTriangle (PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTTexCoord, PRTTexCoord, PRTTexCoord, PRTMaterial *)
  Add one triangle to the scene.
- void AddSphere (PRTVector, PRTFloat, PRTMaterial *)
  Add a sphere to the scene.
- void AddSphere (PRTVector, PRTFloat, PRTFloat, PRTFloat, PRTMaterial *)
  Add a sphere to the scene.
- void AddQuadric (PRTVector, PRTFloat, PRTFloat, PRTFloat, int, PRTMaterial *)
  Add a quadric to the scene.
- void AddCylinder (PRTVector, PRTFloat, PRTFloat, PRTFloat, PRTFloat, PRTMaterial *)
  Add a cylinder to the scene.
- void AddCircle (PRTVector, PRTVector, PRTFloat, PRTFloat, PRTFloat, PRTMaterial *)
  Add a circle to the scene.
- void AddPlane (PRTVector, PRTVector, PRTFloat, PRTFloat, PRTMaterial *)
  Add a plane to the scene.
- void AddQuartic (PRTVector, PRTFloat, PRTFloat, int, PRTMaterial *)
  Add a quartic to the scene.
- void AddCone (PRTVector, PRTFloat, PRTFloat, PRTFloat, PRTFloat, PRTMaterial *)
  Add a cone to the scene.
- void AddLine (PRTVector, PRTVector, PRTMaterial *)
  Add a line to the scene.
- void AddCsgObject (PRTObject *, PRTObject *, int)
  Add a csg object to the scene.
- void AddQuatJulia (PRTVector, PRTQuaternion, int, PRTMaterial *)
  Add a quaterion julia to the scene.
- PRTObject * GetObjectAtPos (int)
  What is the object at pos i?.
- void AddPointLight (PRTVector, PRTVector)
  Add a light of type pointlight.
- void AddTriangleLight (PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector, PRTVector)
  Add a light of type trianglelight.
- void AddTriangleLight (PRTVector, PRTVector, PRTVector, PRTVector)
  Add a light of type trianglelight.
- void AddSphereLight (PRTVector, PRTFloat, PRTVector)
  Add a light of type spherelight.
- void AddDirectionalLight (PRTVector, PRTVector)
  Add a light of type directionallight.

- PRTLight * GetLightAtPos (int)
  What is the light at pos i?.
- PRTVector RayTrace (PRTRay, int)
  Trace a ray and return what color is seen.
- bool BuildOctrees (void)
  Construct the OcTree about the objects in the scene.
- PRTIntersectPoint FindNearestIntersection (PRTRay, PRTObject *d=NULL)
  Solve the nearest intersection of the ray with all the objects.
- PRTVector ComputeReflection (PRTRay, PRTIntersectPoint, PRTVector, int, PRTObject *)
  Compute a reflection.
- PRTVector ComputeRefraction (PRTRay, PRTIntersectPoint, PRTVector, int, PRTObject *)
  Compute a refraction.
- PRTVector ComputeTransparence (PRTRay, PRTIntersectPoint, PRTVector, int, PRTObject *)
  Compute a transparence.
- PRTVector ComputeReflectedLight (PRTVector, PRTVector, PRTVector, PRTObject *, PRTVector, PRTVector)
  Compute the reflected light.
- IMPORT_EXPORT_DATA PluginLeer (LPCSTR, LPCSTR)
  Load a plugin.
- void CargaPlugins (LPSTR)
  Loads all the plugins on the path.

Public Attributes

- bool BTextures
  Textures actived or not.
- bool BReflections
  Reflections actived or not.
- bool BRefractions
  Refractions actived or not.
- bool BSpecular
  Specular actived or not.
- bool BAlpha
  Alpha transparence actived or not.
- bool BShadows
  Shadows actived or not.
- bool BDoubleSided
  Double sided actived or not.
- bool BNInter
  Normal interpolation actived or not.
- bool BCInter
  Color interpolation actived or not.
- bool BOctrees
  Octrees actived or not.
- bool BConvexHulls
  Convex hulls actived or not.
- bool BIndirectLight
  Indirect light actived or not.
- bool BGiReflections
  Global illumination reflections actived or not.

- bool BGiRefractions
  Global illumination refractions actived or not.
- bool BGiAlpha
  Global illumination transparences actived or not.
- bool BGlossy
  Glossy reflections actived or not.
- bool BBump
  Bump actived or not.
- bool MejoraPorDistancia
  Enhanced algorithm.
- PRTVector luzambiente
  The ambient light.
- int shadowpass
  Shadow passes.
- PRTDinamicList ObjectsList
  The objects list.
- PRTDinamicList LightsList
  The lights list.
- PRTDinamicList ImpPlugins
  The import plugins list.
- PRTDinamicList Plugins
  The plugins list.
- PRTRender * render
  The renderer.
- PRTOcTree * Octrees
  The PRTOcTree.
- int octreesdeep
  The OcTree deeping.
- int numrayos
  Number of traced rays.
- unsigned long int numintertest
  The number of intersections solved.

Vemos como gran parte de los atributos del PRTMain se dedican a activar o desactivar opciones globales y comunes al trazador y el tipo de renderización que se quiere. Este tipo de renderización se definirá mediante el constructor del PRTMain y mediante el uso de su parámetro render del tipo PRTRender. Así mismo, gran parte de las funciones de la clase se dedican a añadir objetos PRTObject y luces PRTLight a la escena. Por último cabe destacar que existirá una función RayTrace para lanzar rayos que será la que nos resolverá todo el problema de síntesis de imágenes al devolver para cada rayo el color que es visto por el dentro de la escena.

7.3.1.2 Clase PRTObject

Hemos visto que la utilización de objetos dentro del PiscisRT se hace mediante una interfaz que engloba a todos los tipos de objetos. Esta interfaz es PRTObject y su diagrama de colaboración es:

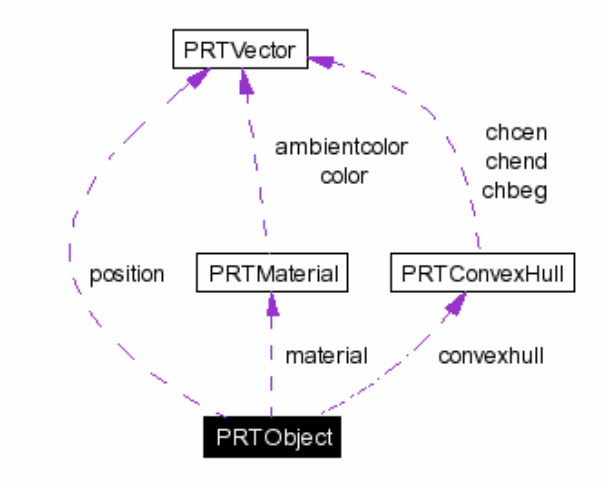

Ilustración 107 - Diagrama de colaboración de PRTObject.

A su vez el diagrama de herencia del la clase PRTObject será:

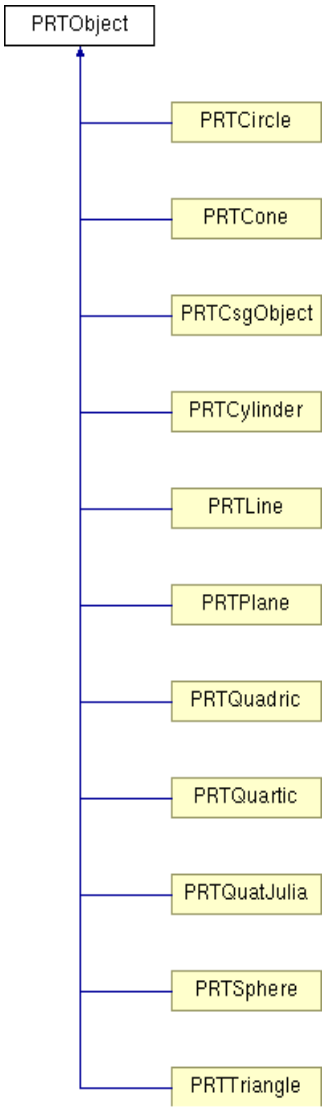

Ilustración 108 - Diagrama de herencia de PRTObject.

Y las funciones de la clase y sus parámetros son:

Public Member Functions

- PRTObject (PRTVector position, int type, PRTMaterial *material)
  A constructor of the PRTObject.
- PRTObject ()
  Empty constructor for PRTObject.
- virtual PRTTexCoord ComputeTexCoord (PRTVector point)=0
  Computes a texture coordinates for a point on 3d.
- virtual PRTVector ComputePoint (PRTTexCoord texcoord)=0
  Computes a point in 3d for a texture coordinates.
- virtual PRTVector ComputeNormal (PRTVector point)=0
  Computes a normal vector for a point on 3d.
- virtual PRTVector ComputeTangent (PRTVector point)=0
  Computes a tangent vector for a point on 3d.
- virtual PRTVector ComputeBinormal (PRTVector point)=0
  Computes a binormal vector for a point on 3d.
- virtual PRTVector ComputeColor (PRTVector point)=0
  Computes a color for a point on 3d.
- virtual PRTIntersectPoint ComputeIntersection (PRTRay ray, bool doublesided)=0
  Computes the intersection of the PRTObject with a PRTRay.
- virtual void Rotate (PRTVector axis, PRTFloat angle, PRTVector point=PRTVector(0, 0, 0))=0
  Rotate the PRTObject in 3d.
- virtual void Translate (PRTVector vector)=0
  Translate the PRTObject in 3d.

Public Attributes

- PRTVector position
  Position of the PRTObject, center of it.
- int type
  The type of the PRTObject.
  #define PRT_TRIANGLE 1001
  #define PRT_SPHERE 1002
  #define PRT_QUADRIC 1003
  #define PRT_CYLINDER 1004
  #define PRT_CIRCLE 1005
  #define PRT_PLANE 1006
  #define PRT_QUARTIC 1007
  #define PRT_CONE 1008
  #define PRT_LINE 1009
  #define PRT_CSG_OBJECT 1010
  #define PRT_QUAT_JULIA 1011
- PRTMaterial * material
  The PRTMaterial assigned to the PRTObject.
- PRTFloat despu
- PRTFloat despv
  Displacement of the texture in u and v coordinates.
- PRTFloat repu
- PRTFloat repv
  Repetition of the texture in u and v coordinates.

- PRTConvexHull convexhull
  Convex hull of the PRTObject.

Vemos así como la clase PRTObject define los parámetros y las funciones que representan enteramente a un objeto en el trazador. Un objeto ha de tener una posición, un tipo, un material y un convex hull definido sobre él. Además habrá de definir dos funciones que sean capaces de trasladar y rotar el objeto y unos parámetros de desplazamiento y escalado del material sobre el objeto. El motor de render manejará al objeto mediante las funciones que han de definirse para calcular primero la coordenada de textura que se asocia con un punto del objeto y viceversa. También ha de definirse la normal, tangente y binormal que se asocian a un punto del objeto. Y por último definir una de las funciones más importantes del trazador, la intersección que provocará un rayo sobre el objeto.

7.3.1.3 Clase PRTLight

Del mismo modo que la clase PRTObject era la interfaz para los objetos, PRTLight es la interfaz para las luces. Veamos su diseño:

El diagrama de colaboración de la clase PRTLight es:

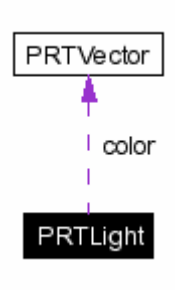

Ilustración 109 - Diagrama de colaboración de PRTLight.

El diagrama de herencia del la clase PRTLight será:

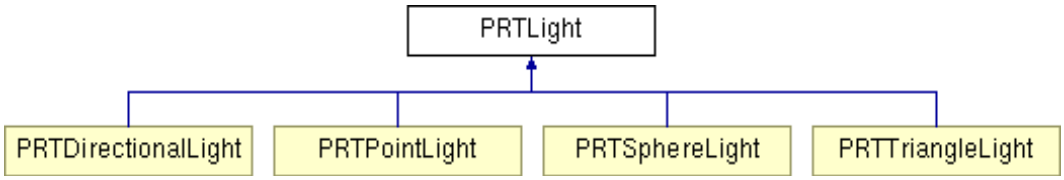

Ilustración 110 - Diagrama de herencia de PRTLight.

Las funciones y los parámetros de la clase serán:

Public Member Functions

- PRTLight (const int type, const PRTVector &color)
  A constructor of the PRTLight.
- PRTLight ()
  Empty constructor for PRTLight.
- virtual PRTVector ComputeWhatPointLight (PRTVector &point)=0
  Computes what point light represents the PRTLight at a given position.
- virtual PRTVector ComputeLightRay (PRTRay &ray, PRTIntersectPoint &ipoint, PRTVector &normal, const PRTObject *object, PRTMain *main)=0

Computes the light ray. What light intensity is seen by the ray.

Public Attributes

- int type
  The type of the PRTLight.
  #define PRT_POINTLIGHT 2001
  #define PRT_TRIANGLELIGHT 2002
  #define PRT_SPHERELIGHT 2003
  #define PRT_DIRECTIONALLIGHT 2004
- PRTVector color
  Color of the light. (Diffuse color).
- PRTFloat attenuation0
  Attenuation factor 0.
- PRTFloat attenuation1
  Attenuation factor 1.
- PRTFloat attenuation2
  Attenuation factor 2.
- PRTFloat intensity
  Intensity/power of the light.

Cualquier fuente de luz dentro del trazador de rayos habrá de tener un tipo, un color, una intensidad y una atenuación. Además para el manejo de su funcionamiento y de su forma habrá de definir una función que decida que punto de su superficie iluminará a un punto dado y qué cantidad de luz verá un rayo, es decir calcular el rayo de luz.

7.3.1.4 Clase PRTRender

Después de las clases que hemos visto, la siguiente en importancia es la clase PRTRender. Esta será la interfaz para los métodos de render que se quieran implementar dentro del PiscisRT.

El diagrama de colaboración de la clase PRTRender es:

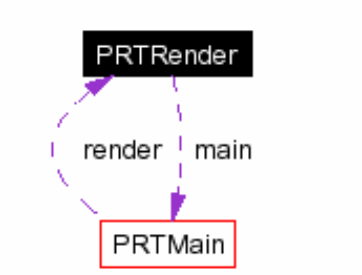

Ilustración 111 - Diagrama de colaboración de PRTRender.

El diagrama de herencia del la clase PRTRender será:

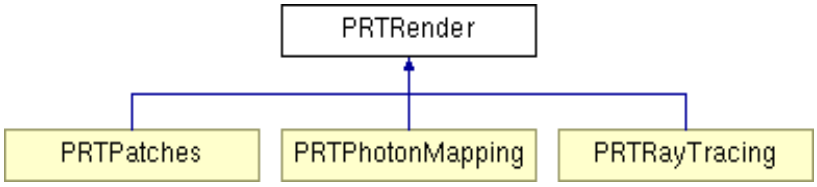

Ilustración 112 - Diagrama de herencia de PRTRender.

Las funciones y parámetros de la clase PRTRender son:

Public Member Functions

- PRTRender (PRTMain *, int)
  A constructor of the PRTRender.
- virtual PRTVector RayTrace (PRTRay, int)=0
  An interface to trace rays.

Public Attributes

- int type
  The type of the renderer.
  #define PRT_RAYTRACING 7001
  #define PRT_PHOTONMAPPING 7002
  #define PRT_PATCHES 7003
- PRTMain * main
  A pointer to the PRTMain object.

Cualquier clase que herede de PRTRender deberá definir que tipo de método de render es, y una función para trazar rayos sobre él. Los tipos de render que se quieren son ray tracing, patches y photon mapping.

7.3.2 Plugins para Max

7.3.2.1 PRTMaxPlugin

El proyecto *PRTMaxPlugin* está compuesto por las siguientes clases y en los siguientes ficheros.

Ficheros	Descripción
asciitok.h	lista definiciones para los identificadores que necesitaremos del Max SDK
DllEntry.cpp	funciones y parámetros para la interfaz del Max SDK
maxincl.h	lista de includes necesarios para compilar el plugin
PRTMaxPlugin.h, PRTMaxPlugin.cpp, PRTMaxPlugin.rc	**MyView,** – clase que maneja la transformación de la vista **PRTMaxPlugin,** – clase principal de manejo del plugin tipo render **PRTRenderParams** – clase que representa los parámetros que se piden en el render por parte del usuario
PRTMaxPluginDlg.h, PRTMaxPluginDlg.cpp	**PRTMaxPluginDlg** – clase que define el comportamiento del diálogo de render específico del PRTMaxPlugin
resource.h	fichero de identificación de los recursos gráficos utilizados

Siendo la clase más importante la clase PRTMaxPlugin.

7.3.2.1.1 Clase PRTMaxPlugin

El diagrama de colaboración de PRTMaxPlugin es:

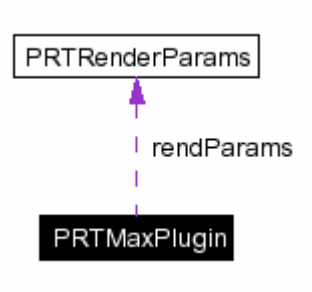

Ilustración 113 - Diagrama de colaboración de PRTMaxPlugin.

Sus funciones y atributos de clase serán:

Public Member Functions

- void GetViewParams (INode *vnode, ViewParams &vp, TimeValue t)
- BOOL nodeEnum (INode *, int, TimeValue)
- void ExportMesh (INode *, int, TimeValue)
- void ExportGeomObject (INode *, int, TimeValue)
- void ExportLightObject (INode *, int, TimeValue)
- IOResult Load (ILoad *iload)
- IOResult Save (ISave *isave)
- Class_ID ClassID ()
- void GetClassName (TSTR &s)
- RefTargetHandle Clone (RemapDir &remap)
- void DeleteThis ()
- PRTMaxPlugin ()
- void ResetParams ()
- RendParamDlg * CreateParamDialog (IRendParams *ir, BOOL prog)
- void Close (HWND hwnd)
- int Open (INode *scene, INode *vnode, ViewParams *viewPar, RendParams &rp, HWND hwnd, DefaultLight *defaultLights=NULL, int numDefLights=0)
- int Render (TimeValue t, Bitmap *tobm, FrameRendParams &frp, HWND hwnd, RendProgressCallback *prog=NULL, ViewParams *viewPar=NULL)

Public Attributes

- BOOL bOpen
- INode * scene
- INode * pViewNode
- ViewParams view
- Interface * ip
- PRTMain trazador
- PRTVector camarapos
- PRTVector camaradir
- PRTFloat ancho
- PRTFloat alto
- PRTFloat fov
- PRTDinamicList Materiales
- PRTRenderParams rendParams
- bool BTextures
- bool BReflections
- bool BRefractions
- bool BAlpha
- bool BSpecular

- bool BShadows
- bool BDoubleSided
- bool BNInter
- bool BCInter
- bool BOctrees
- bool BConvexHulls
- int OctreesDeep
- bool BIndirectLight
- bool BGiReflections
- bool BGiRefractions
- bool BGiAlpha
- bool BGlossy
- bool BMejoraPorDistancia
- int NumPhotones
- int GroupPhotones
- float ScalePhoton

Podemos ver que el plugin funcionará convirtiendo la información que nos pasan por el Max en la información que ha de recibir el PiscisRT y a la hora de renderizar utilizar el PiscisRT en lugar del Max.

7.3.2.2 PRTMaxMaterial

A su vez, el proyecto *PRTMaxMaterial* está compuesto por las siguientes clases y en los siguientes ficheros.

Ficheros	Descripción
DllEntry.cpp	funciones y parámetros para la interfaz del Max SDK
PRTMaxMaterial.h, PRTMaxMaterial.cpp, PRTMaxMaterial.rc	**PRTMaxMaterial** – clase que maneja el tratamiento de los parámetros del material del PiscisRT en forma de diálogo gráfico
resource.h	fichero de identificación de los recursos gráficos utilizados

Siendo la clase más importante la clase PRTMaxMaterial.

7.3.2.2.1 Clase PRTMaxMaterial

Sus funciones y atributos de clase serán:

Public Member Functions

- ParamDlg * CreateParamDlg (HWND hwMtlEdit, IMtlParams *imp)
- void Update (TimeValue t, Interval &valid)
- Interval Validity (TimeValue t)
- void Reset ()
- void NotifyChanged ()
- void SetAmbient (Color c, TimeValue t)
- void SetDiffuse (Color c, TimeValue t)
- void SetSpecular (Color c, TimeValue t)
- void SetShininess (float v, TimeValue t)
- Color GetAmbient (int mtlNum=0, BOOL backFace=FALSE)
- Color GetDiffuse (int mtlNum=0, BOOL backFace=FALSE)
- Color GetSpecular (int mtlNum=0, BOOL backFace=FALSE)
- float GetXParency (int mtlNum=0, BOOL backFace=FALSE)
- float GetShininess (int mtlNum=0, BOOL backFace=FALSE)

- float GetShinStr (int mtlNum=0, BOOL backFace=FALSE)
- float WireSize (int mtlNum=0, BOOL backFace=FALSE)
- float GetSelfIllum (int mtlNum=0, BOOL backFace=FALSE)
- BOOL GetSelIllumColorOn (int mtlNum=0, BOOL backFace=FALSE)
- Color GetSelIllumColor (int mtlNum=0, BOOL backFace=FALSE)
- void Shade (ShadeContext &sc)
- float EvalDisplacement (ShadeContext &sc)
- Interval DisplacementValidity (TimeValue t)
- int NumSubMtls ()
- Mtl * GetSubMtl (int i)
- void SetSubMtl (int i, Mtl *m)
- TSTR GetSubMtlSlotName (int i)
- TSTR GetSubMtlTVName (int i)
- int NumSubTexmaps ()
- Texmap * GetSubTexmap (int i)
- void SetSubTexmap (int i, Texmap *m)
- TSTR GetSubTexmapSlotName (int i)
- TSTR GetSubTexmapTVName (int i)
- BOOL SetDlgThing (ParamDlg *dlg)
- PRTMaxMaterial (BOOL loading)
- IOResult Load (ILoad *iload)
- IOResult Save (ISave *isave)
- Class_ID ClassID ()
- SClass_ID SuperClassID ()
- void GetClassName (TSTR &s)
- RefTargetHandle Clone (RemapDir &remap)
- RefResult NotifyRefChanged (Interval changeInt, RefTargetHandle hTarget, PartID &partID, RefMessage message)
- int NumSubs ()
- Animatable * SubAnim (int i)
- TSTR SubAnimName (int i)
- int NumRefs ()
- RefTargetHandle GetReference (int i)
- void SetReference (int i, RefTargetHandle rtarg)
- int NumParamBlocks ()
- IParamBlock2 * GetParamBlock (int i)
- IParamBlock2 * GetParamBlockByID (BlockID id)
- void DeleteThis ()

Public Attributes

- IParamBlock2 * pblock
- Mtl * submtl [NSUBMTL]
- BOOL mapOn [NSUBMTL]
- Color color
- Color ambiente
- Color especular
- float shininess
- float xparency
- float shinstr
- float wiresize
- float spin
- float illumfloat
- BOOL illumbool
- Color illumcolor
- float reflection

- float refraction
- float alpha
- float gireflection
- float girefraction
- float gialpha
- PRTMaterial * mat
- Interval ivalid

En el plugin del material habrá funciones para interactuar con la interfaz de usuario de modo que al final se tengan los parámetros del material actualizados. Estos parámetros servirán para crear el material definitivo dentro del PiscisRT.

7.3.3 Elements

El *elements* estará formado por el siguiente fichero:

Ficheros	Descripción
elements.cpp	funcionamiento del programa parseador y renderizador

El *elements* no es basado en clases pero aún así su funcionamiento se puede explicar mediante los siguientes puntos:

- Crear una instancia de la clase PRTMain
- Parsear el fichero de entrada y a cada etiqueta hacer lo que sea oportuno, guardando información o interactuando con la instancia del PRTMain
- Crear la ventana glut donde se mostrará el render especificado en el fichero
- Lanzar los rayos de manera correcta y a cada color devuelto por estos actualizar la ventana de render con otro pixel más (será óptimo aquí utilizar múltiples threads)

7.4 Implementación

Como ya he comentado, el código fuente de todo el proyecto y la documentación del mismo serán accesibles publicamente. Por tanto sobre el código y a salvedad de ciertas partes incluidas en los primeros capítulos, no voy a incluir nada. Dejaré pues al lector la posiblidad de hecharle una ojeada al código o no, sobre todo porque no creo que sea conveniente poner código fuente en la memoria y en cambio si es más conveniente explicar cuál es la teoría que se ha utilizado.

7.5 Resultados

Echemos ahora una ojeada a cómo ha quedado el proyecto a fecha de la entrega de esta memoria. Ya que el trazador en si no es visible, vamos a ver su funcionamiento a partir de las otras aplicaciones *PRTMaxPlugin* y *elements*.

7.5.1 Plugins para Max

Los plugins creados para el 3d Studio Max dan al PiscisRT la capacidad de gestionar escenas de gran tamaño y complejidad de forma mucho más sencilla que como lo haríamos con el elements. Así por ejemplo podremos gestionar la siguiente escena:

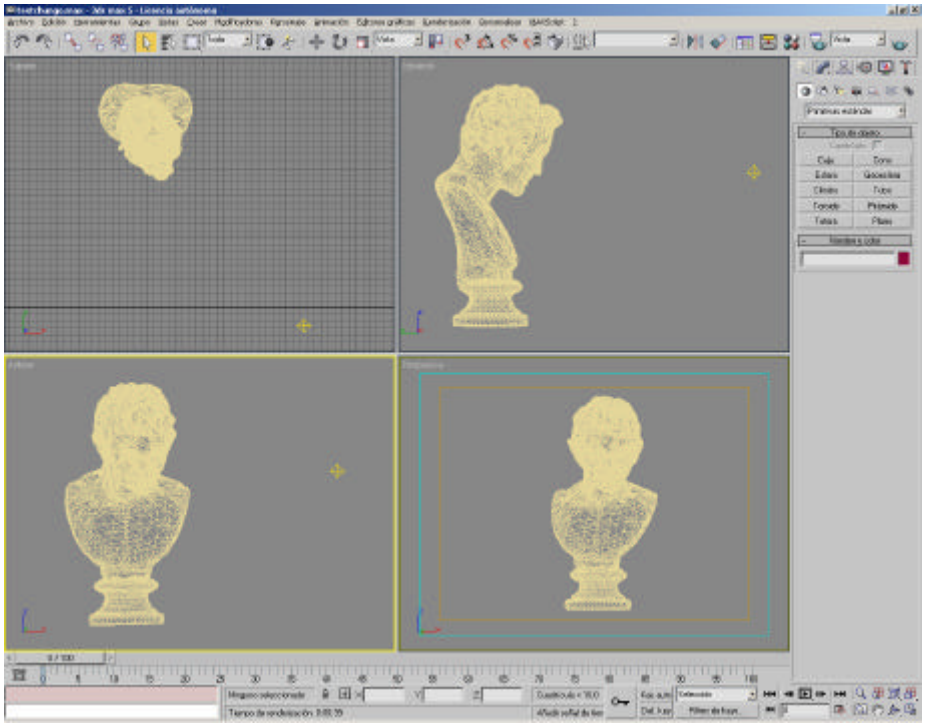

Ilustración 114 - Escena de ejemplo del 3d Studio Max.

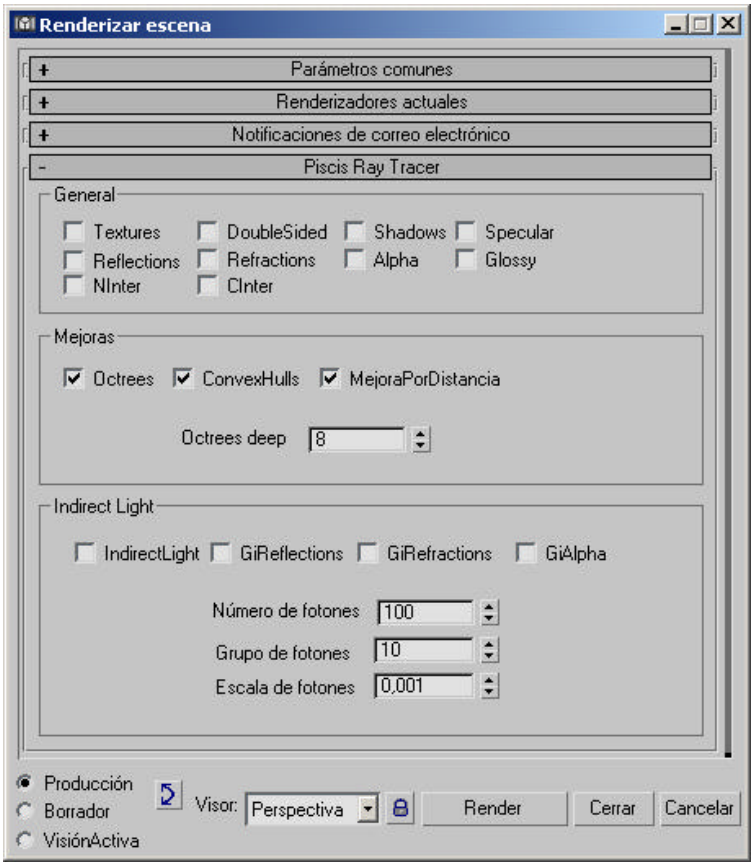

Ilustración 115 - Opciones elegidas en el PRTMaxPlugin.

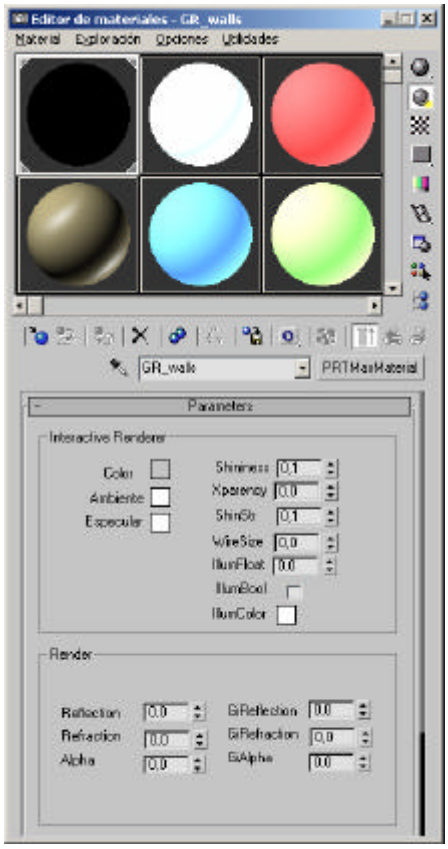

Ilustración 116 - Opciones elegidas en el PRTMaxMaterial.

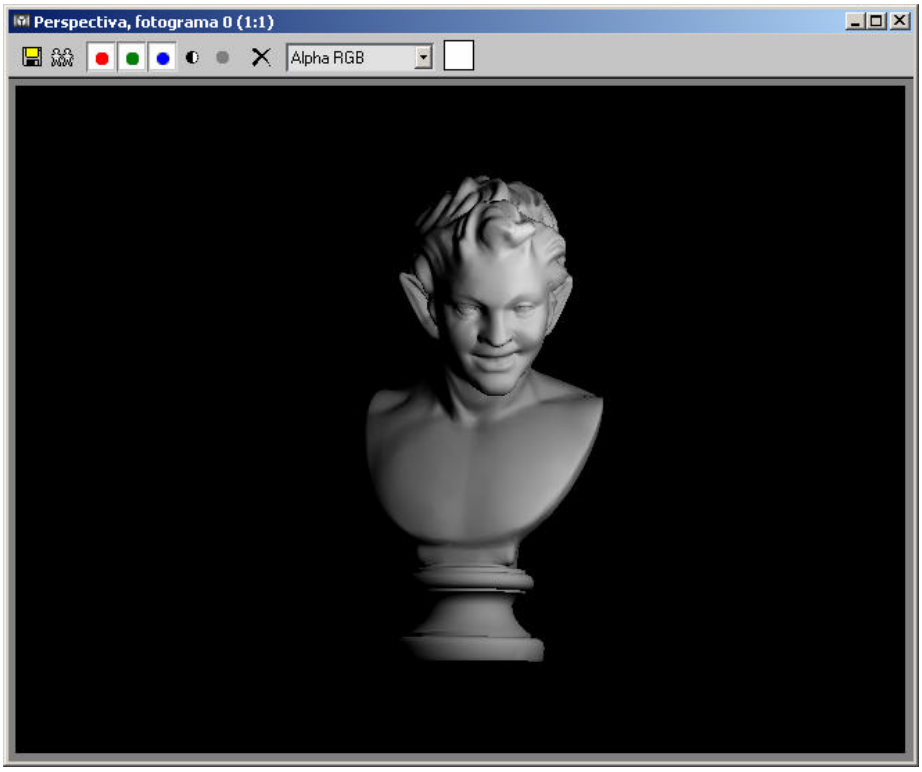

Ilustración 117 - Escena del Max renderizada mediante PiscisRT.

7.5.2 Elements

Para probar el elements elegiremos la escena típica utilizada en ambientes de iluminación global, el cornell test. Dicha escena tendrá como fichero descriptivo .prt el siguiente código xml:

```
<piscisrt>

	<scene
				name="cornell"
				width="120"
				height="120"
				camerapos="278,473,1000"
				cameraend="278,273,0"
				camerafov="35"
				deep="2"
				BShadows="true"
				shadowpass="1"
				randomsamples="false"
				rendersamples="1"
				BBump="false"
				stereo="false"
				stereowidth="20"
				BDoubleSided="false"
				photonmap="true"
				numphotones="1000"
				groupphotones="100"
				scalephoton="0.005">


<!--				MATERIALES				-->

	<material name="luz" color="1,1,1" forced="true"></material>

	<material name="mat" ambientcolor2="0.2,0.2,0.2" color="1.0,0.87,0.45" gireflection="1.0" bumpmap="bump.jpg"></material>

	<material name="mat2" ambientcolor2="0.2,0.2,0.2" color="1.0,0.87,0.45" gireflection="1.0" refraction2="0.8" bumpmap="bump.jpg"></material>

	<material name="matright" ambientcolor2="0.2,0.2,0.2" color="0.21,0.35,0.21" gireflection="1.0" bumpmap="bump.jpg"></material>

	<material name="matleft" ambientcolor2="0.2,0.2,0.2" color="0.72,0.201,0.201" gireflection="1.0" bumpmap="bump.jpg"></material>



<!--				OBJETOS				-->

<!--				FLOOR				-->

	<triangle p1="552.8,0.0,0.0" p2="549.6,0.0,-559.2" p3="0.0,0.0,-559.2" n1="0,1,0" n2="0,1,0" n3="0,1,0" t1="1,0" t2="1,1" t3="0,1" material="mat"></triangle>

	<triangle p1="0.0,0.0,-559.2" p2="0.0,0.0,0.0" p3="552.8,0.0,0.0" n1="0,1,0" n2="0,1,0" n3="0,1,0" t1="0,1" t2="0,0" t3="1,0" material="mat"></triangle>

<!--				LIGHT				-->

	<triangle p1="343.0,548.6,-332.0" p2="343.0,548.6,-227.0" p3="213.0,548.6,-227.0" n1="0,-1,0" n2="0,-1,0" n3="0,-1,0" t1="1,0" t2="0,0" t3="0,1" material="luz"></triangle>

	<triangle p1="213.0,548.6,-227.0" p2="213.0,548.6,-332.0" p3="343.0,548.6,-332.0" n1="0,-1,0" n2="0,-1,0" n3="0,-1,0" t1="0,1" t2="0,0" t3="1,0" material="luz"></triangle>


<!--				CEILING				-->

	<triangle p1="556.0,548.8,-559.2" p2="556.0,548.8,0.0" p3="0.0,548.8,0.0" n1="0,-1,0" n2="0,-1,0" n3="0,-1,0" t1="1,0" t2="1,1" t3="0,1" material="mat"></triangle>
	<triangle p1="0.0,548.8,0.0" p2="0.0,548.8,-559.2" p3="556.0,548.8,-559.2" n1="0,-1,0" n2="0,-1,0" n3="0,-1,0" t1="0,1" t2="0,0" t3="1,0" material="mat"></triangle>
```

```
<!--                    BACK WALL                    -->

        <triangle p1="0.0,548.8,-559.2" p2="0.0,0.0,-559.2" p3="549.6,0.0,-559.2" n1="0,0,1" n2="0,0,1" n3="0,0,1" t1="0,1" t2="0,0" t3="1,0" material="mat"></triangle>

        <triangle p1="549.6,0.0,-559.2" p2="556.0,548.8,-559.2" p3="0.0,548.8,-559.2" n1="0,0,1" n2="0,0,1" n3="0,0,1" t1="1,0" t2="1,1" t3="0,1" material="mat"></triangle>

<!--                    LEFT WALL                    -->

        <triangle p1="0.0,548.8,0.0" p2="0.0,0.0,0.0" p3="0.0,0.0,-559.2" n1="1,0,0" n2="1,0,0" n3="1,0,0" t1="0,1" t2="0,0" t3="1,0" material="matright"></triangle>

        <triangle p1="0.0,0.0,-559.2" p2="0.0,548.8,-559.2" p3="0.0,548.8,0.0" n1="1,0,0" n2="1,0,0" n3="1,0,0" t1="1,0" t2="1,1" t3="0,1" material="matright"></triangle>

<!--                    RIGHT WALL                    -->

        <triangle p1="556.0,548.8,-559.2" p2="549.6,0.0,-559.2" p3="552.8,0.0,0.0" n1="-1,0,0" n2="-1,0,0" n3="-1,0,0" t1="0,1" t2="0,0" t3="1,0" material="matleft"></triangle>

        <triangle p1="552.8,0.0,0.0" p2="556.0,548.8,0.0" p3="556.0,548.8,-559.2" n1="-1,0,0" n2="-1,0,0" n3="-1,0,0" t1="1,0" t2="1,1" t3="0,1" material="matleft"></triangle>

<!--                    SHORT BLOCK                    -->

        <triangle p1="240.0,165.0,-272.0" p2="82.0,165.0,-225.0" p3="130.0,165.0,-65.0" n1="0,1,0" n2="0,1,0" n3="0,1,0" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="130.0,165.0,-65.0" p2="290.0,165.0,-114.0" p3="240.0,165.0,-272.0" n1="0,1,0" n2="0,1,0" n3="0,1,0" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
        <triangle p1="240.0,165.0,-272.0" p2="290.0,165.0,-114.0" p3="290.0,0.0,-114.0" n1="0.95340009693182,0,-0.30170889143412" n2="0.95340009693182,0,-0.30170889143412" n3="0.95340009693182,0,-0.30170889143412" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="290.0,0.0,-114.0" p2="240.0,0.0,-272.0" p3="240.0,165.0,-272.0" n1="0.95340009693182,0,-0.30170889143412" n2="0.95340009693182,0,-0.30170889143412" n3="0.95340009693182,0,-0.30170889143412" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
        <triangle p1="290.0,165.0,-114.0" p2="130.0,165.0,-65.0" p3="130.0,0.0,-65.0" n1="0.29282578030182,0,0.95616581323044" n2="0.29282578030182,0,0.95616581323044" n3="0.29282578030182,0,0.95616581323044" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="130.0,0.0,-65.0" p2="290.0,0.0,-114.0" p3="290.0,165.0,-114.0" n1="0.29282578030182,0,0.95616581323044" n2="0.29282578030182,0,0.95616581323044" n3="0.29282578030182,0,0.95616581323044" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
        <triangle p1="130.0,165.0,-65.0" p2="82.0,165.0,-225.0" p3="82.0,0.0,-225.0" n1="-0.95782628522115,0,0.28734788556635" n2="-0.95782628522115,0,0.28734788556635" n3="-0.95782628522115,0,0.28734788556635" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="82.0,0.0,-225.0" p2="130.0,0.0,-65.0" p3="130.0,165.0,-65.0" n1="-0.95782628522115,0,0.28734788556635" n2="-0.95782628522115,0,0.28734788556635" n3="-0.95782628522115,0,0.28734788556635" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
        <triangle p1="82.0,165.0,-225.0" p2="240.0,165.0,-272.0" p3="240.0,0.0,-272.0" n1="-0.28512090676781,0,-0.95849155892157" n2="-0.28512090676781,0,-0.95849155892157" n3="-0.28512090676781,0,-0.95849155892157" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="240.0,0.0,-272.0" p2="82.0,0.0,-225.0" p3="82.0,165.0,-225.0" n1="-0.28512090676781,0,-0.95849155892157" n2="-0.28512090676781,0,-0.95849155892157" n3="-0.28512090676781,0,-0.95849155892157" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>

<!--                    TALL BLOCK                    -->

        <triangle p1="314.0,330.0,-456.0" p2="265.0,330.0,-296.0" p3="423.0,330.0,-247.0" n1="0,1,0" n2="0,1,0" n3="0,1,0" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="423.0,330.0,-247.0" p2="472.0,330.0,-406.0" p3="314.0,330.0,-456.0" n1="0,1,0" n2="0,1,0" n3="0,1,0" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
        <triangle p1="472.0,330.0,-406.0" p2="423.0,330.0,-247.0" p3="423.0,0.0,-247.0" n1="0.95564896144166,0,0.29450817050718" n2="0.95564896144166,0,0.29450817050718" n3="0.95564896144166,0,0.29450817050718" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
        <triangle p1="423.0,0.0,-247.0" p2="472.0,0.0,-406.0" p3="472.0,330.0,-406.0" n1="0.95564896144166,0,0.29450817050718" n2="0.95564896144166,0,0.29450817050718" n3="0.95564896144166,0,0.29450817050718" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
        <triangle p1="314.0,330.0,-456.0" p2="472.0,330.0,-406.0" p3="472.0,0.0,-406.0" n1="0.30170889143412,0,-0.95340009693182" n2="0.30170889143412,0,-0.95340009693182" n3="0.30170889143412,0,-0.95340009693182" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
```

```
            <triangle p1="472.0,0.0,-406.0" p2="314.0,0.0,-456.0" p3="314.0,330.0,-456.0" n1="0.30170889143412,0,-0.95340009693182" n2="0.30170889143412,0,-0.95340009693182" n3="0.30170889143412,0,-0.95340009693182" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
            <triangle p1="265.0,330.0,-296.0" p2="314.0,330.0,-456.0" p3="314.0,0.0,-456.0" n1="-0.95616581323044,0,-0.29282578030182" n2="-0.95616581323044,0,-0.29282578030182" n3="-0.95616581323044,0,-0.29282578030182" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
            <triangle p1="314.0,0.0,-456.0" p2="265.0,0.0,-296.0" p3="265.0,330.0,-296.0" n1="-0.95616581323044,0,-0.29282578030182" n2="-0.95616581323044,0,-0.29282578030182" n3="-0.95616581323044,0,-0.29282578030182" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>
            <triangle p1="423.0,330.0,-247.0" p2="265.0,330.0,-296.0" p3="265.0,0.0,-296.0" n1="-0.29620907081072,0,0.95512312628763" n2="-0.29620907081072,0,0.95512312628763" n3="-0.29620907081072,0,0.95512312628763" t1="0,0" t2="1,0" t3="1,1" material="mat2"></triangle>
            <triangle p1="265.0,0.0,-296.0" p2="423.0,0.0,-247.0" p3="423.0,330.0,-247.0" n1="-0.29620907081072,0,0.95512312628763" n2="-0.29620907081072,0,0.95512312628763" n3="-0.29620907081072,0,0.95512312628763" t1="1,1" t2="0,1" t3="0,0" material="mat2"></triangle>


<!--                    LUCES                    -->

            <trianglelight p1="343.0,545.0,-332.0" p2="343.0,545.0,-227.0" p3="213.0,545.0,-227.0" n1="0,-1,0" n2="0,-1,0" n3="0,-1,0" color="1,1,1" intensity="100000"></trianglelight>
            <trianglelight p1="213.0,545.0,-227.0" p2="213.0,545.0,-332.0" p3="343.0,545.0,-332.0" n1="0,-1,0" n2="0,-1,0" n3="0,-1,0" color="1,1,1" intensity="100000"></trianglelight>

            </scene>

</piscisrt>
```

Ilustración 118 - Ejemplo de fichero de entrada para el elements, testcornell.

A la cual, el *elements* se comportará del siguiente modo:

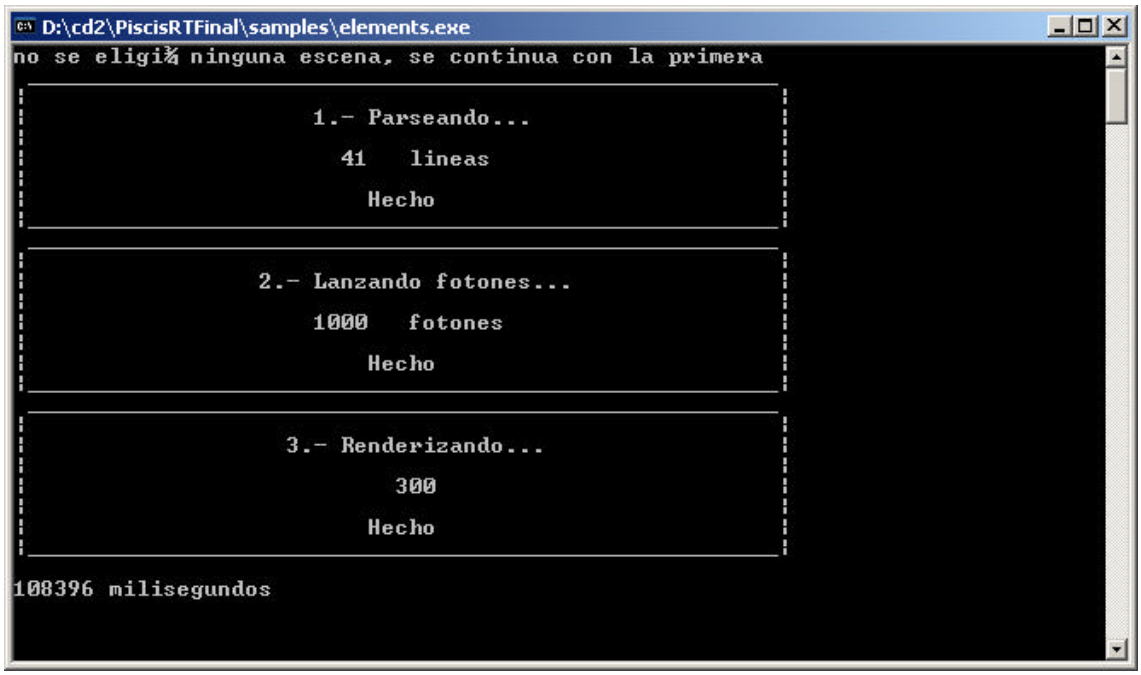

Ilustración 119 - Funcionamiento del elements.

Renderizando una escena como la que sigue:

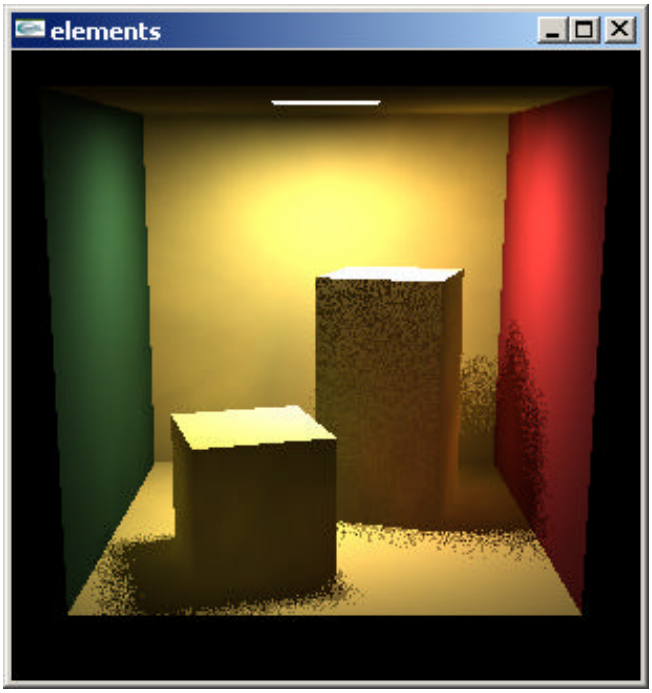

Ilustración 120 - Escena test cornell con iluminación global.

Del mismo modo se podrán utilizar muchas otras etiquetas xml que añadiran funcionalidad al *elements* siempre gracias al *PiscisRT*. Con ello también seremos capaces de renderizar escenas como las que siguen:

Test Cornell sin iluminación global.

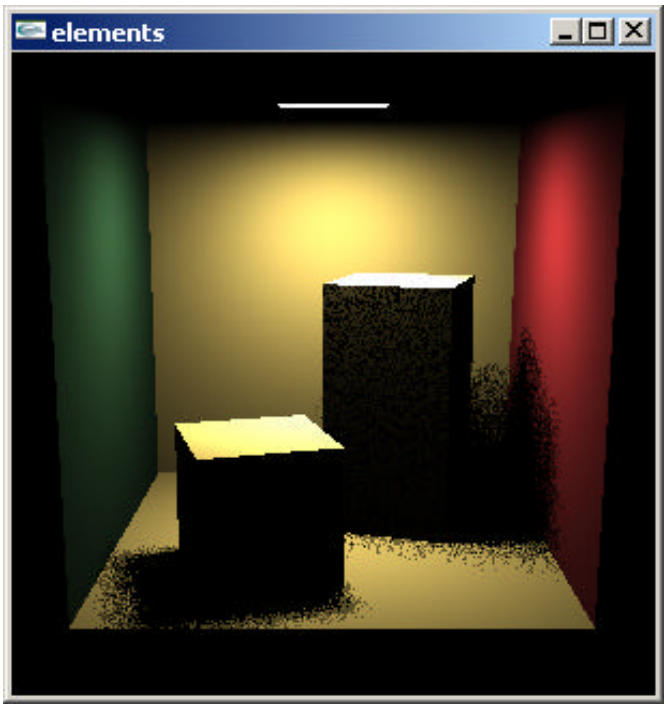

Ilustración 121 - Escena test cornell sin iluminación global.

Escena con esferas reflectantes.

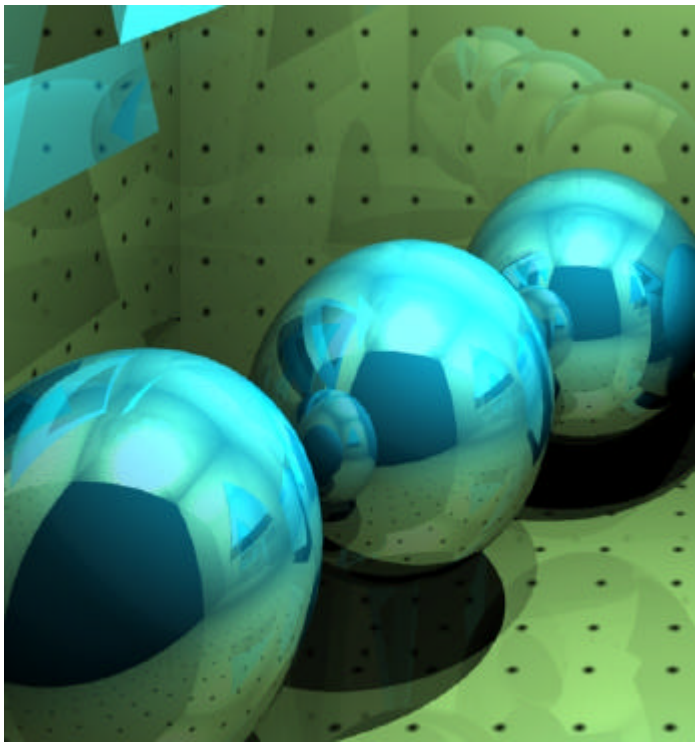

Ilustración 122 - Escena con iluminación simple y esferas reflectantes.

Una escena donde se ven multitud de tipos de objetos.

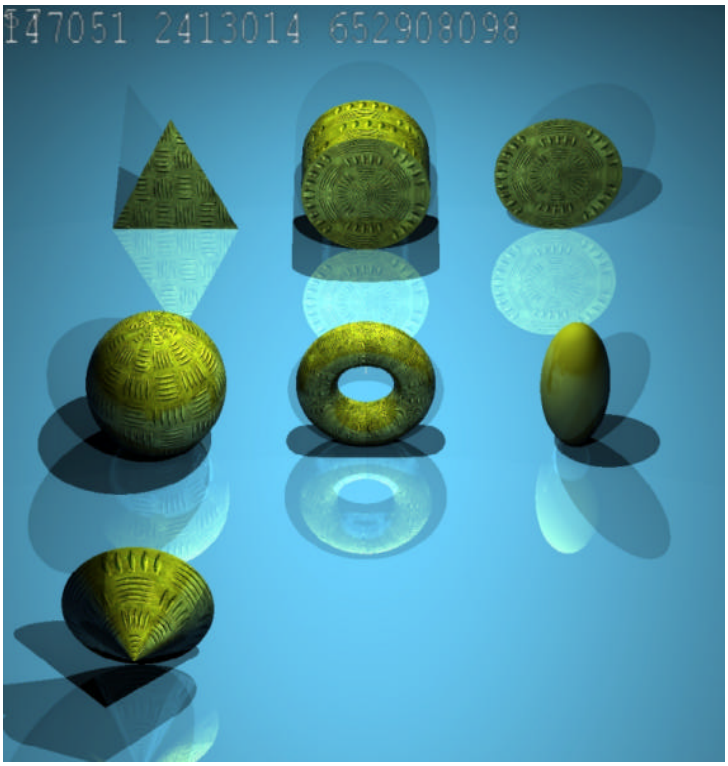

Ilustración 123 - Escena para probar los tipos de objetos.

Una esfera en una escena con luz esférica.

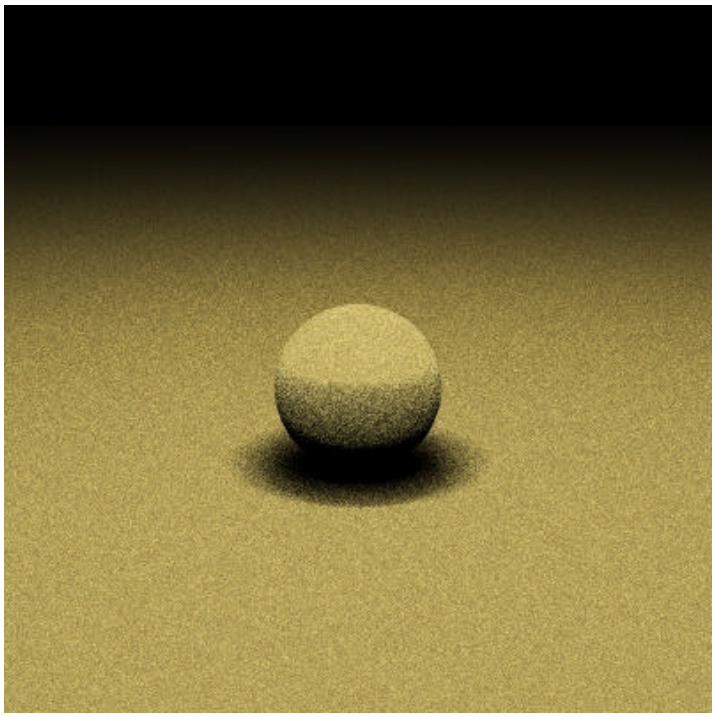

Ilustración 124 - Escena con luz esférica.

Test Cornell con objetos refractantes e iluminación global:

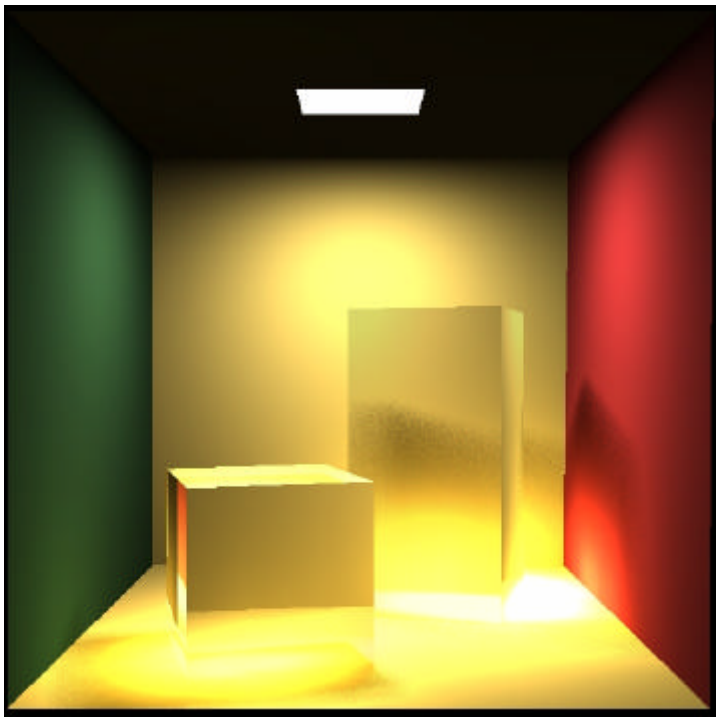

Ilustración 125 - Test Cornell con objetos refractantes.

A la vista del funcionamiento de las aplicaciones del proyecto, concluimos que los objetivos han sido cubiertos con éxito.

Capítulo 8: Conclusiones

El desarrollo del presente proyecto y la búsqueda de toda la información que era necesaria para él ha servido para sobre todo tener mayor conocimiento del entorno de los trazadores de rayos y de la iluminación global. Además, se ha profundizado en los aspectos de la programación orientada a objetos y la creación de plugins para otras aplicaciones externas.

Hemos conseguido crear un trazador de rayos con multitud de posibilidades que además es apto para incluirse en cualquier otra aplicación grande o pequeña. Un trazador de rayos que puede utilizar métodos de iluminación avanzados y que es fácil de actualizar y mejorar.

Además hemos creado las aplicaciones que demuestran que es fácil utilizar el trazador y que extraen de él todo su potencial. Una aplicación que se asemeja a la forma típica de trabajar con los trazadores, parseando un fichero y interpretándolo como lo hacía el PovRay. O una aplicación mucho más actual como es un plugin de render para el programa de modelado y síntesis de imágenes 3d Studio Max.

Se ha creído necesario que el trazador además sea compatible Windows y Linux y que su código sea de libre acceso bajo licencia GPL para que el proyecto pueda servir de ayuda a cualquiera. Esto quizá al principio del proyecto no se consideró así pero a lo largo de su desarrollo se ha creído que el proyecto en si es muy útil, y que puede ser de mucha ayuda para aquellos que busquen un trazador de rayos no tan complicado como los profesionales.

En general yo mismo me siento muy satisfecho por el trabajo realizado y con muchas ganas de continuarlo en el futuro inmediato. De verdad que creo que el proyecto ha sido una fuente inacabable de conocimientos nuevos y espero que así sea también para cualquiera que se interese por él.

Al principio del proyecto se marcaron unas metas que al finalizar el proyecto se ven cumplidas. Ahora por tanto creo necesario que para que el proyecto avance aún más tendremos que marcarnos más metas y que si la situación lo permite podremos cumplirlas de aquí en adelante. Estas nuevas metas se verán en el siguiente capítulo.

Capítulo 9: Posibles extensiones

Con el desarrollo del proyecto se ha visto que sería conveniente añadir multitud de extensiones al análisis que se hizo en un principio. Entre esas extensiones se consideran algunas de poco impacto y otras que tendrían un impacto más grande sobre el proyecto.

Entre las extensiones de corto impacto se puede considerar añadir nuevos objetos, nuevas luces o nuevos modos de render. Estas extensiones se acomodarían a la interfaz que se hizo para tal efecto y no causarían cambios sustanciales al trazador.

Sin embargo hay otras extensiones que se podrían realizar o incluso se han empezado a desarrollar y deberían acabarse. Por ejemplo el uso masivo de plugins como partes del *PiscisRT*. Se ha desarrollado ya la inclusión de plugins de importación de geometría pero se considera interesante también añadir plugins de exportación, de render, o de efectos visuales. Más en la línea de programas de síntesis de imágenes profesionales.

También se considera muy interesante y que incluso se encuentra ya en desarrollo es la ejecución paralela del *PiscisRT*. Mediante una función que lance los rayos en threads diferentes, obtendremos una mejora sustancial del proceso de render.

En lo que se refiere tanto al *elements* como a los plugins del Max, las mejoras que se pueden realizar sobre ellos son mucho más numerosas que con el *PiscisRT*. Esto es porque se pretende que ambas aplicaciones vayan mejorando en posibilidades y facilidad de manejo, y tanto es así que el trabajo puede llegar a no tener fin. En general se reconoce como extension principales la posibilidad de que en todo momento, todo el potencial del PiscisRT sea accesible mediante ambas aplicaciones.

También se ha considerado que ya que el *PiscisRT* es tan independiente y flexible, puede incluirse en otras aplicaciones que no sean tan apropiadas para el uso de trazadores de rayos dentro de ellas, como son los motores 3d en tiempo real o las aplicaciones basadas en web.

Con el tiempo se verá si todas estas extensiones se pueden llegar a desarrollar y qué otras extensiones aparecerán como posibles al llegar a cubrir las anteriores.

Bibliografía

[1] Mitchell, D.a.P., Fast Algorithms for 3D Computer Graphics, *PhD Thesis*, University of Sherffield, July 1990.

[2] Norton, V.A., Generation and rendering of geometric fractals in 3-D, *Computer Graphics 16, 3, 61-67*, 1982.

[3] Mandelbrot, B. B., Fractal aspects of the iteration of $z \rightarrow \lambda z$ (1-z) for complex λ and z. *Annals New York Academy of sciences 35g , 249-259*, 1980.

[4] Hart, J.C., Image Space Algorithms for Visualizing Quaternion Julia Sets, *Master's thesis*, University of Illinois at Chicago, 1989.

[5] Glassner, A.S., An Introduction To Ray Tracing, 1993.

[6] Bui-Thong, Phong, Illumination for Computer-generated Pictures, *Comm. ACM*, 18(6), June 1975.

[7] Lafortune, Eric P.F., Foo Sing-Choong, Torrance K.E., Greenberg D.P, Non-Linear Aproximation of Reflectance Functions, Cornell University, 1996.

[8] Watt A., Watt M., Advanced Animation and Rendering Techniques, Theory aand Practice, 1992.

[9] Moller T., Trumbore B., Fast, Minimum Storage Ray/Triangle Intersection, 1997.

[10] Schwarze J., Cubic and Quartic Roots, p. 404-407, code: p. 738-786, Graphics Gems.

[11] Turk G., Generating Random Points in Triangles, p. 24-28, code: p. 649-650, GraphicsGems.

[12] John C., Daniel J., Louis H., Ray Tracing Deterministic 3-D Fractals, University of Illinois at Chicago, 1989.

[13] Massimino P., 4-D Fractals, http://skal.planet-d.net/fractals, 2003.

[14] Sillion F., Puech C., Radiosity and Global Illumination, Morgan *Kaufmann Publishers, Inc.*, 1994.

[15] Kajiya J., The rendering equation, *Computer Graphics Proceedings, Siggraph'86*, pages 143-150, 1986.

[16] Cohen M.F., Wallace J.R., Radiosity and Realistic Imagen Synthesis, *Academic Press Professional, Boston, MA*, 1993.

[17] Dutré P., Mathematical Frameworks and Monte Carlo Algorithms for Global Illumination in Computer Graphics, *Ph.D. thesis*, Department of Computer Science, Katholieke Universitiet Leuven, Belgium, 1996.

[18] Goral C.M, Torrance K.E, Greenberg D.P, Battaile B., Modeling the Interaction of Light Between Diffuse Surfaces, *In Computer Graphics (ACM SIGGRAPH '84 Proceedings*, vol. 18, no. 3, pp. 212-222, 1984

[19] Cohen M., Greenberg D., The hemi-cube: A radiosity solution for complex enviroments, *Computer Graphics*, 19(3), pages 31-40, 1985.

[20] Elias H., Radiosity, http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm, 2000.

[21] Jensen H.W., Global Illumination using Photon Maps, *Rendering Techniques '96*, pages 21-30, 1996.

[22] Jensen H.W., Realistic Image Synthesis Using Photon Mapping, *A K Peters LTD.*, 2001.

[23] Jon L.B., Multidimensional binary search trees used for associative searching, *Communications of tha ACM*, 18(9): 509-517,1975.

[24] Elias H., http://freespace.virgin.net/hugo-elias/models/m_perlin.htm, 2000.

[25] Rise raytracer, http://rise.sourceforge.net/, 2003.

[26] BitchX raytracer, http://www.bitchx.org, 2003.
[27] Eurographics, Computer graphics forum volume 20, 2001.
[28] PiscisRT, http://www.terra.es/personal/sergiosancho, 2003.

Tabla de ilustraciones