

TÉCNICAS AVANZADAS DE GRÁFICOS 3D

Práctica 2:

Trazador de rayos clásico básico

Máster Oficial en Informática Gráfica, Juegos y Realidad Virtual
Universidad Rey Juan Carlos

Luis Gasco Poderoso
Alejandro Ribao Valverde

Abril 2008

I. Parte básica

1.1 Algoritmo

1.1.1 Trazador de rayos básico

Para el trazado de rayos hemos partido de las especificaciones de la cámara en la escena a renderizar.

La secuencia es la siguiente:

1. Se genera un rayo por cada pixel de la imagen a generar.
 - a. Se calcula la posición del pixel en el espacio para el sistema de coordenadas canónico y el plano de la imagen situado a una unidad de distancia en el eje +Z. Es decir, multiplicar la distancia al centro del plano por la tangente del ángulo de visualización.
 - b. Se hace un cambio de base al sistema de coordenadas de la cámara (formado por su posición y sus ejes XYZ).
2. Para cada objeto de la escena se comprueban si el rayo colisiona con él por delante de la cámara (es decir, en $t > 0$). De todas las colisiones (si hay) se escoge la más próxima a la cámara.
3. Se evalúa el color en el punto de la intersección.
 - a. Para cada luz, se comprueba si hay algún obstáculo de por medio.
 - b. Si no hay obstáculos, se suma la aportación de dicha luz para el tipo de material del punto y su relación con la luz (ángulo, distancia, ...)
 - c. Si el material produce reflexiones, se genera un nuevo rayo reflejado y se trata como un nuevo rayo, a no ser que se haya alcanzado el límite de profundidad del árbol.
4. Por último, se combinan todos los colores devueltos por los rayos para generar la imagen final.

1.1.2 Supermuestreo en el lanzamiento de rayos para reducir los bordes dentados

Para el supermuestreo de n muestras hemos trazado n^2 rayos por pixel. También hemos añadido la posibilidad de realizar muestreo estocástico del estilo del *jittering* (transparencia 62 del tema de Aliasing). Es decir, la posición de la n -ésima porción del pixel por la que se lanza el rayo se determina de forma aleatoria.

1.1.3 Sombras

Para el cálculo de sombras se calculan las occlusiones entre cada luz y el punto a evaluar. Si hay algún objeto de por medio se considera sombra respecto de la luz en cuestión. Es un proceso semejante a determinar si desde la posición de la luz se ve dicho punto o no. Si no se ve es porque está en su zona de sombra. Por lo tanto se ignora la aportación de dicha luz.

1.1.4 Reflexiones

Sólo se ha considerado un rayo reflejado en cada punto (en el caso de haber reflexiones). Por lo tanto el proceso, desde un punto en el que hay reflexiones, es equivalente a renderizar una imagen de un pixel y sumarla al color que ya tenía el punto procedente de la iluminación directa.

Genera recursividad en el programa, por lo que hay que tener cuidado con el número de superficies reflectantes y el número de rebotes que se permiten.

1.2 Geometría

Para las cuestiones referentes a las intersecciones con las superficies nos hemos apoyado en la siguiente bibliografía:

Hill, F.S., "Computer Graphics using Open GL", 2nd ed., Prentice Hall, 2001

Lengyel, E., Mathematics for 3D Game Programming & Computer Graphics, Charles River Media, 2004

F. Dunn and I. Parberry. 3D Math Primer for Graphics and Game Development. Wordware Publishing, 2002.

1.2.1 Esferas

La ecuación de una esfera de centro c y radio r es:

$$(x-c_x)^2 + (y-c_y)^2 + (z-c_z)^2 = r^2$$

Para el cálculo de la intersección con las esferas hemos recurrido a igualar las ecuaciones de ambos y ponerlas en función de t . El resultado es una ecuación cuadrática (de grado 2) del tipo $At^2 + Bt + C = 0$.

La ecuación se resuelve con la famosa fórmula:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

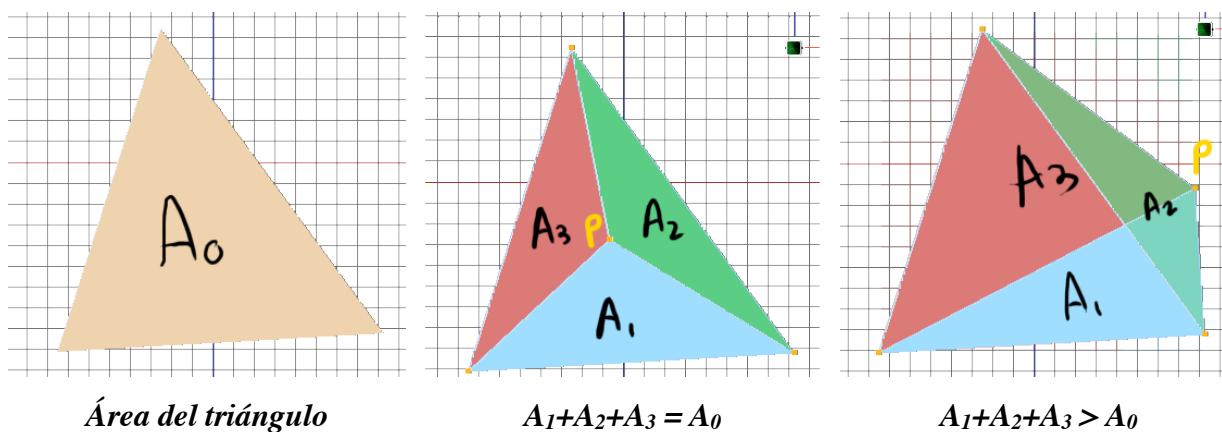
En función del valor del discriminante podemos saber si no existe intersección (discriminante negativo), si el rayo es tangente a la esfera (cero) o si es secante (positivo). En el caso de intersecar en dos puntos se toma aquel con menor valor de t (más cercano al origen del rayo). Los valores de t negativos se descartan por estar por detrás del origen.

Finalmente, el punto de intersección y la normal se calculan evaluando el rayo para ese valor de t (obteniendo ya el punto) y calculando el vector que va del centro de la esfera al punto de intersección (normal). Si se ha calculado bien, el módulo de dicho vector antes de normalizarse debería de ser igual al radio de la esfera.

1.2.2 Triángulos

La intersección con un triángulo se reduce a calcular la intersección entre el rayo y el plano definido por la propia figura, y luego comprobar si cae dentro del propio triángulo.

Para comprobar si está contenido en ella hemos recurrido a las coordenadas baricéntricas del punto. Dado un punto P, hemos calculado las áreas de los 4 posibles triángulos que se pueden formar con los cuatro puntos (los tres vértices y el punto P de intersección con el plano).



Una vez calculadas, la suma de las áreas de los triángulos en los que P es vértice suyo tiene que ser igual a la del triángulo original para que pertenezca a él. De lo contrario es superior.

1.3 Materiales

1.3.1 Lambertianos (mate)

No hemos incluido la componente ambiental en ninguno de los materiales ni luces. En el caso del material mate, se limita a devolver su color difuso multiplicado por el coseno del ángulo que forma la luz incidente con la normal a la superficie.

Es independiente del punto de vista.

1.3.2 Phong

El material *Phong* añade una componente especular con un determinado color y un exponente de brillo. La componente especular es:

$$compSpec = colorSpec \cdot \cos(\text{ángulo(luzreflejada-vista)})^{\text{exp}}$$

Por lo tanto, la componente especular **sí** es dependiente del punto de vista.

1.3.3 ShinyPhong (Phong con reflexión)

Tal y como se ha comentado en el algoritmo de traza de rayos, cuando un rayo topa con un material con reflexiones se genera un nuevo rayo en la dirección de la reflexión y se añade el color resultante del rayo a las componentes difusa y especular que ya incorporaba el material *Phong*.

La reflexión del rayo es perfecta. Es decir, se refleja con el mismo ángulo con el que incide respecto de la normal a la superficie.

1.4 Luces

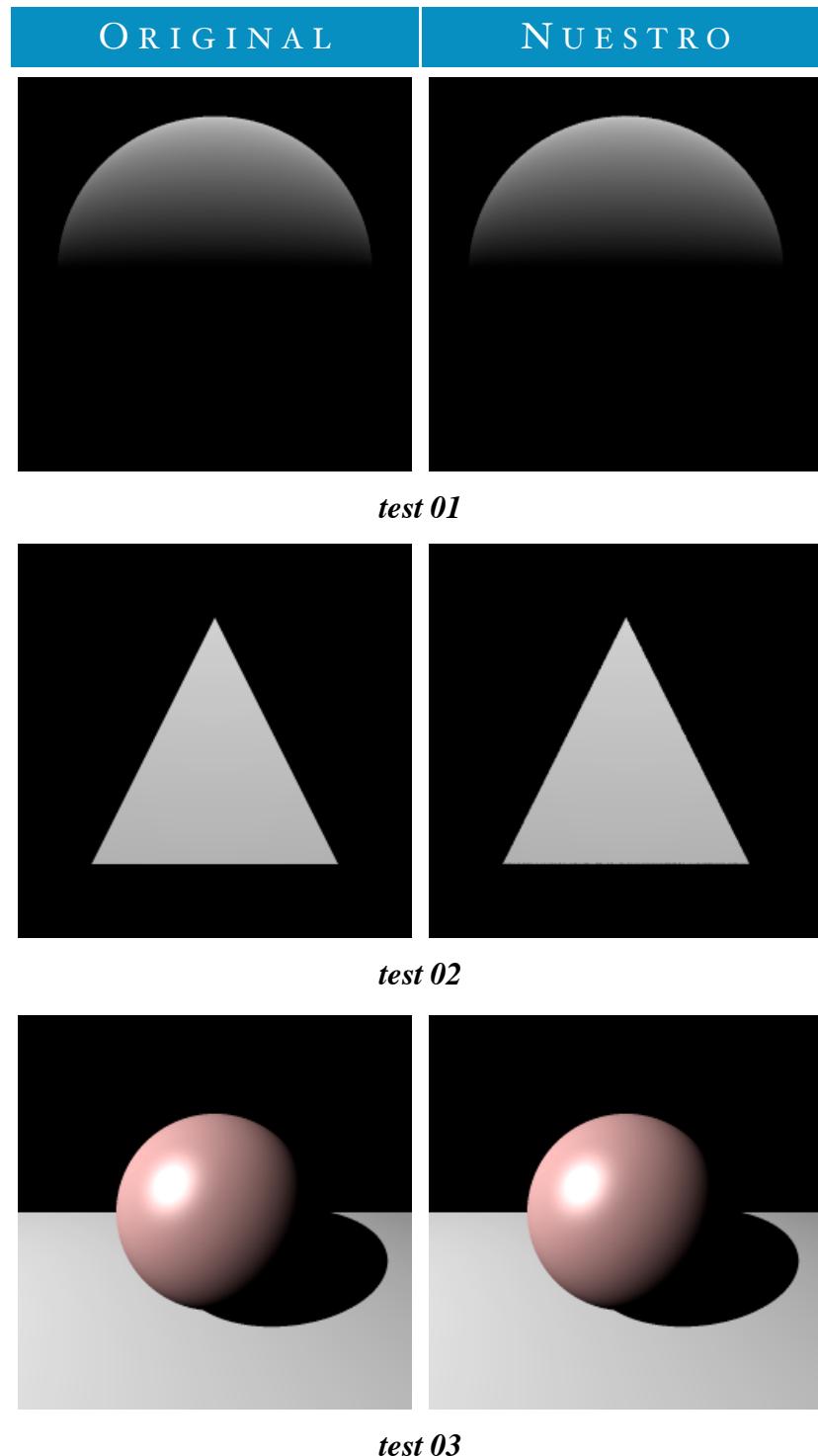
1.4.1 Luces puntuales

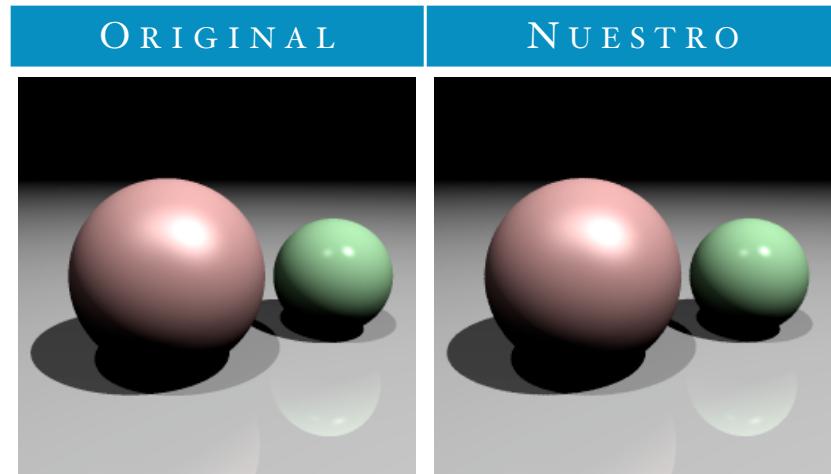
Pese a que no era un requisito, hemos implementado la atenuación respecto de la distancia en las luces puntuales.

Dicho esto, la luz puntual se caracteriza por una posición y un color (intensidad). A la hora de calcular la aportación de la luz en un punto se comprueba su dirección y su distancia, haciendo decaer la intensidad de forma proporcional al cuadrado de la lejanía.

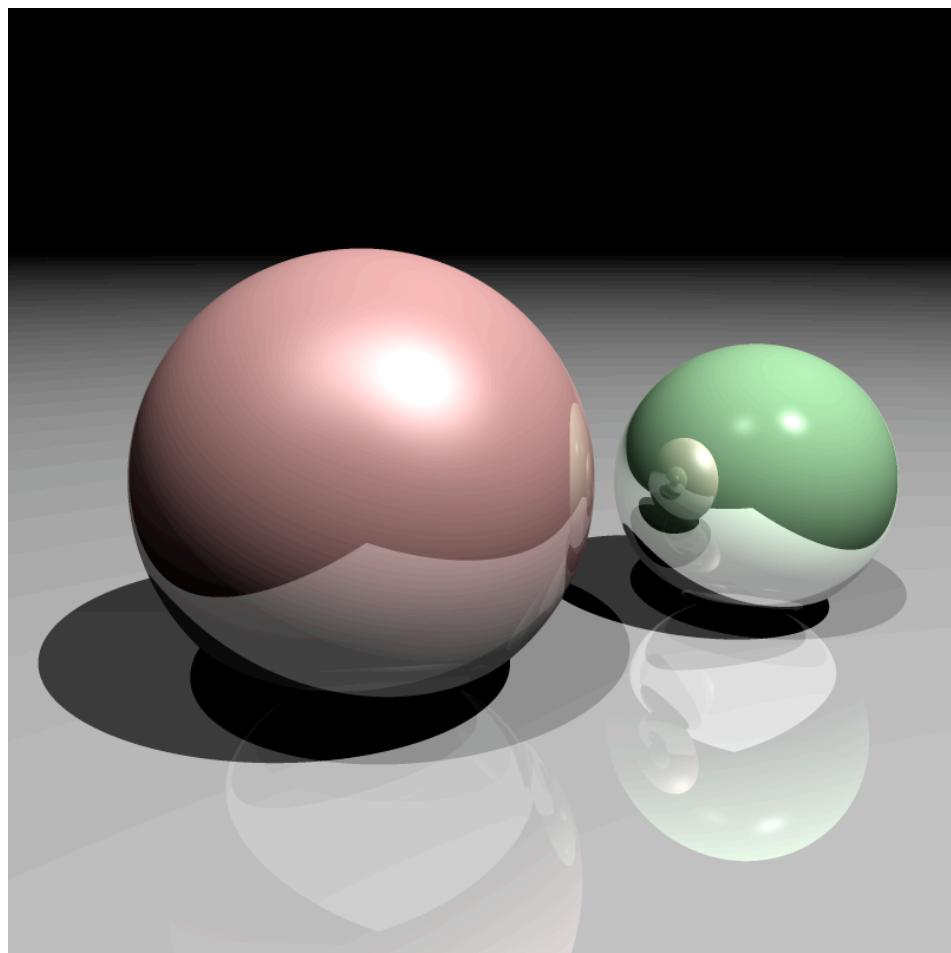
1.4 Resultados

Con esto se da por completada la parte básica de la práctica. Las escenas de prueba proporcionadas generan los siguientes resultados:

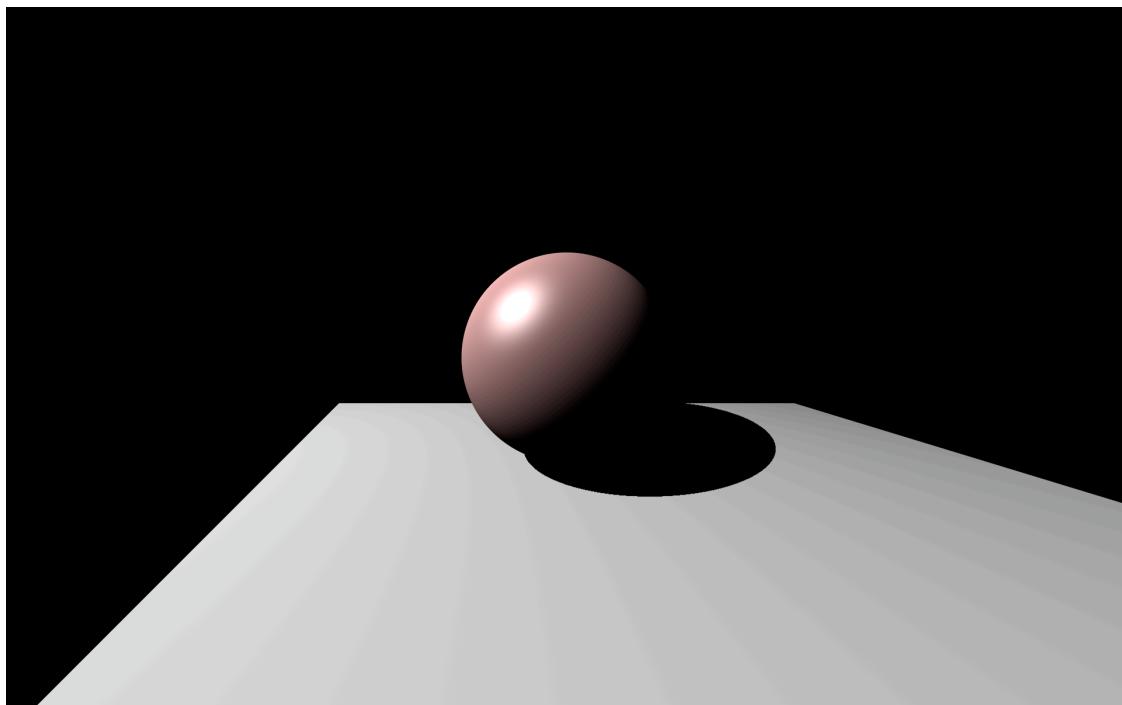




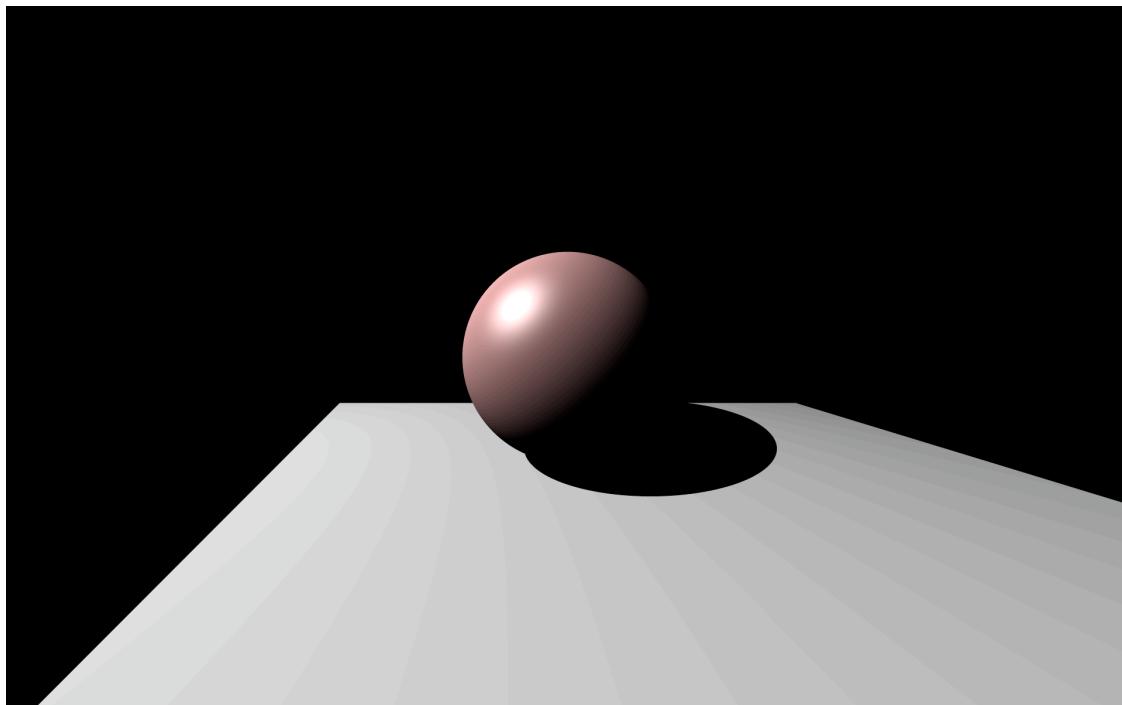
test 04



A modo de curiosidad, compusimos la misma escena del test03 en el programa de modelado 3D *Cinema 4D* y comparamos los resultados (y ya de paso comprobamos que funcionaba bien para resoluciones con relaciones de aspecto distintas a 1:1): (*test032*)



Maxon Cinema 4D



Nuestro motor

2. Partes opcionales

2.1 Geometría

2.1.1 Cilindros

Los cilindros son en cierta manera parecidos a las esferas, pero introducen un nuevo parámetro: el eje.

Las ecuaciones de un cilindro con base en el origen de coordenadas y eje Z son:

$$x^2 + y^2 = r^2$$

$$0 \leq z \leq h$$

Dónde h es la longitud del eje.

En nuestro caso hemos implementado también cilindros no alineados a los ejes. El cálculo de la intersección está dividida en tres partes:

1. Intersección con las “tapas” del cilindro. Se trata de intersecciones recta/plano en las que se comprueba que la distancia al centro de la tapa sea menor que el radio.
2. Intersección con el lateral del cilindro. Son aquellos puntos que están a distancia r del eje. Una vez obtenidos, además deben cumplir que su distancia a la tapa de abajo sea menor o igual que la altura h .
3. Elección del punto de intersección más cercano al origen del rayo.

El código para describir un cilindro puede ser de dos maneras:

Dados el centro de la base (c), el radio (r) y la altura (h):

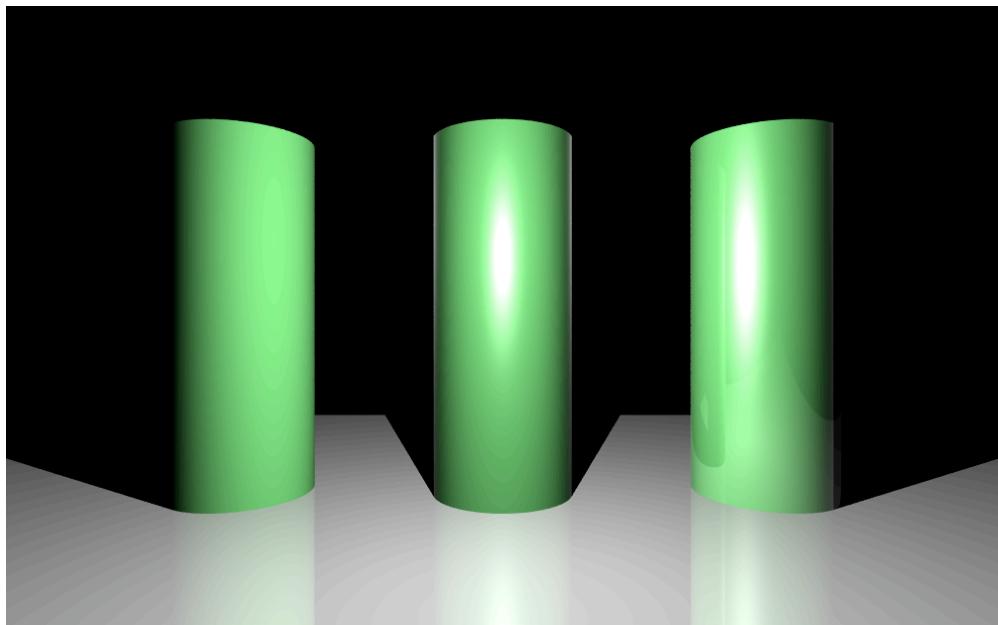
```
<surface class="Cylinder">
  <material> ... </material>
  <center1>cx cy cz</center1>
  <height>h</height>
  <radius>r</radius>
</surface>
```

Se genera un cilindro cuyo eje es paralelo al eje Y de la escena.

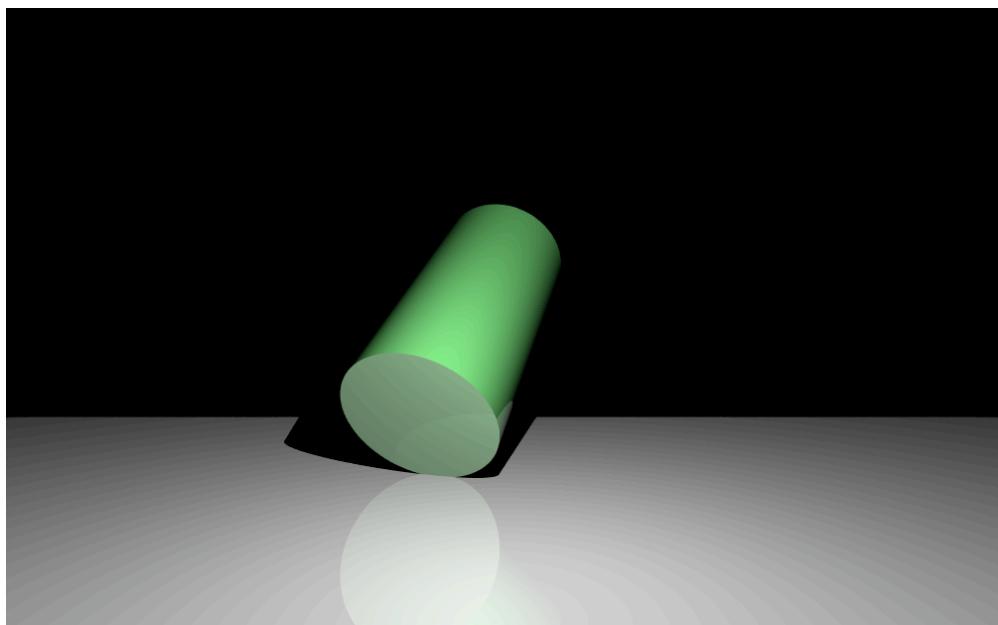
Dados los centros de las dos bases ($c1$ $c2$) y el radio (r):

```
<surface class="Cylinder">
  <material> ... </material>
  <center1>c1x c1y c1z</center1>
  <center2>c2x c2y c2z</center2>
  <radius>r</radius>
</surface>
```

Algunos ejemplos:



*Cilindros alineados al eje Y. Lambert, Phong y ShinyPhong
(testCyl01)*



*Cilindro no alineado al eje Y
(testCyl02)*

2.1.2 Mallas

Las mallas las hemos implementado de manera que se leen de archivos **OFF** (*GeomView Object-File Format*). Existe un programa libre y con versiones para Windows, Mac y Linux que permite visualizar, modificar y convertir archivos *.off* *.wrl* y *.3ds* (entre otros). Se trata de [MeshLab](#).

El tipo Malla sólo es compatible con aquellos ficheros OFF que describan mallas mediante triángulos.

Lo que se hace es leer y almacenar todos los vértices y triángulos, así como calcular las normales en cada vértice.

El cálculo de las intersecciones son tantas intersecciones rayo/tríangulo como facetas tiene la malla. De ellos se escoge el de menor t , como siempre.

Existe la opción de interpolar las normales ponderándolas mediante las coordenadas bari-céntricas o utilizar la del triángulo.

También se puede escalar la malla e invertir las normales.

Hemos incluido además un método que dibuja las normales mediante cilindros.

La descripción en la escena XML es:

```
<surface class="Mesh">
    <material>...</material>
    <position>cx cy cz</position> <!-- desplazamiento -->
    <size>sx sy sz</size> <!-- escalado -->
    <filename>PATH</filename> <!-- dir. del archivo -->
    <breakang>1.0</breakang> <!-- 1 interpolar (defecto) 0 no -->
    <invNormal>0</invNormal> <!-- 1 invertir 0 no (defecto) -->
</surface>
```

A continuación se muestra una malla con y sin interpolación de normales:

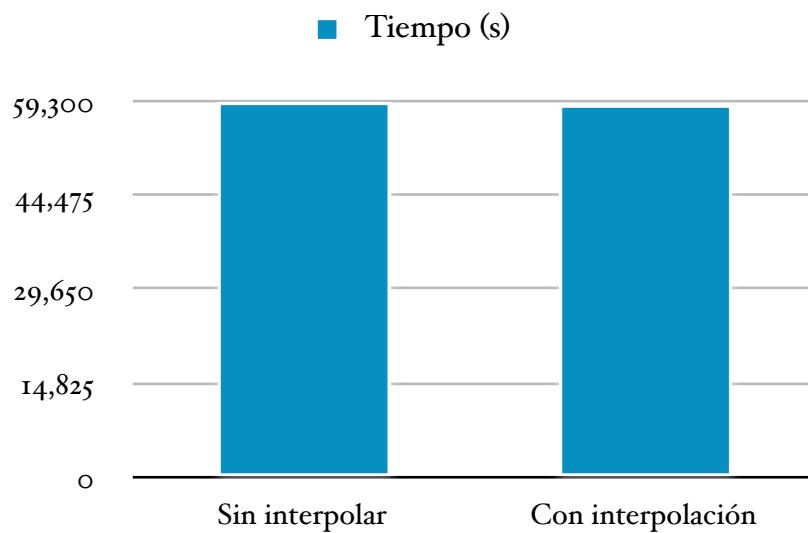


*Tetera sin interpolación de normales
(testM02)*



*Tetera con interpolación de normales
(testM01)*

Para una malla con 1056 facetas y 528 vértices, la penalización por interpolar las normales es la siguiente:



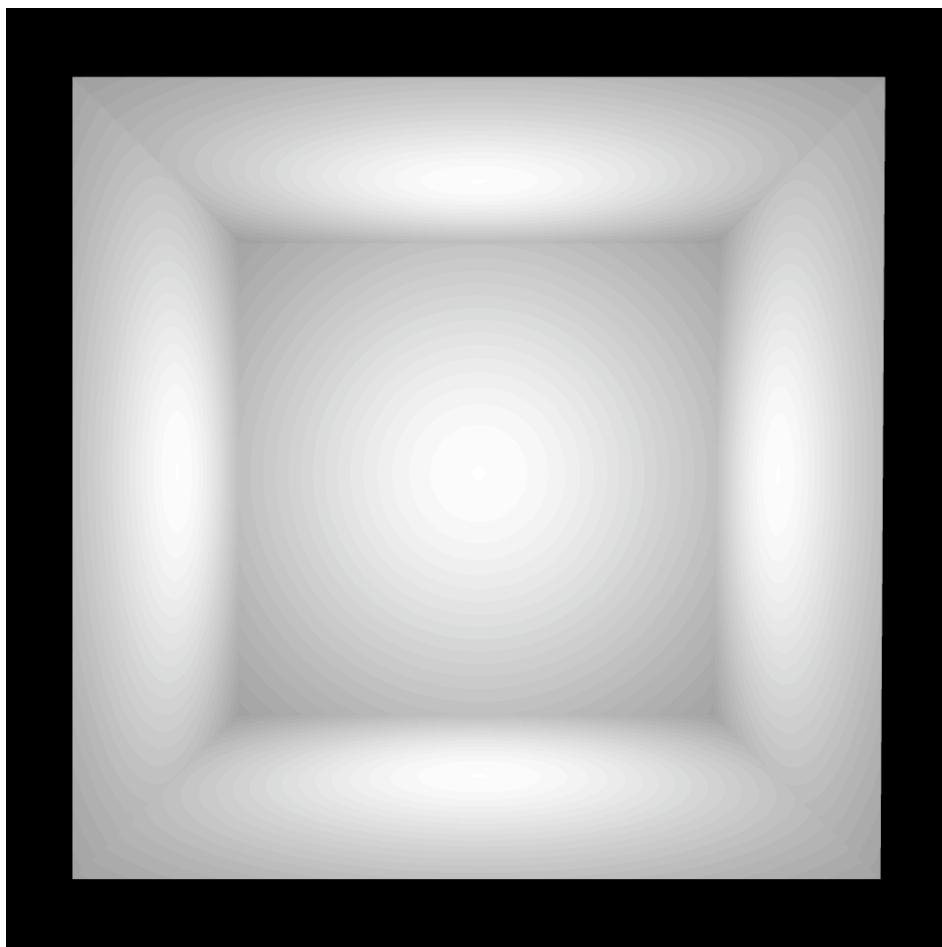
Curiosamente no se aprecia diferencia apenas en el rendimiento.

2.1.3 Quads

Los Quads son simplemente dos triángulos nombrados así para poder simplificar el diseño de las escenas. Su descripción es del tipo:

```
<surface class="Quad">
  <material>...</material>
  <v0>-1.0 -1.0 0.0</v0>
  <v1>-1.0 1.0 100.0</v1>
  <v2>1.0 1.0 100.0</v2>
  <v3>1.0 -1.0 0.0</v3>
</surface>
```

Un simple cubo hecho con quads:



*Quads
(testQuad)*

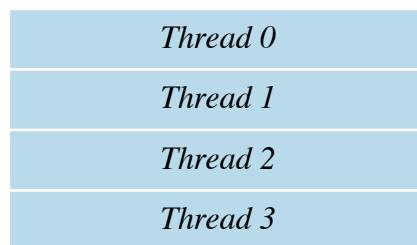
2.2 Optimización del rendimiento

2.2.1 Concurrency

Dadas la naturaleza de las operaciones que se realizan, hemos considerado que resulta fácil y adecuado separar los rayos en varias hebras.

Concretamente, lo que hemos hecho ha sido dividir la imagen final en tantas porciones verticales iguales como hebras se tengan.

Por ejemplo, para cuatro hebras sería algo así:

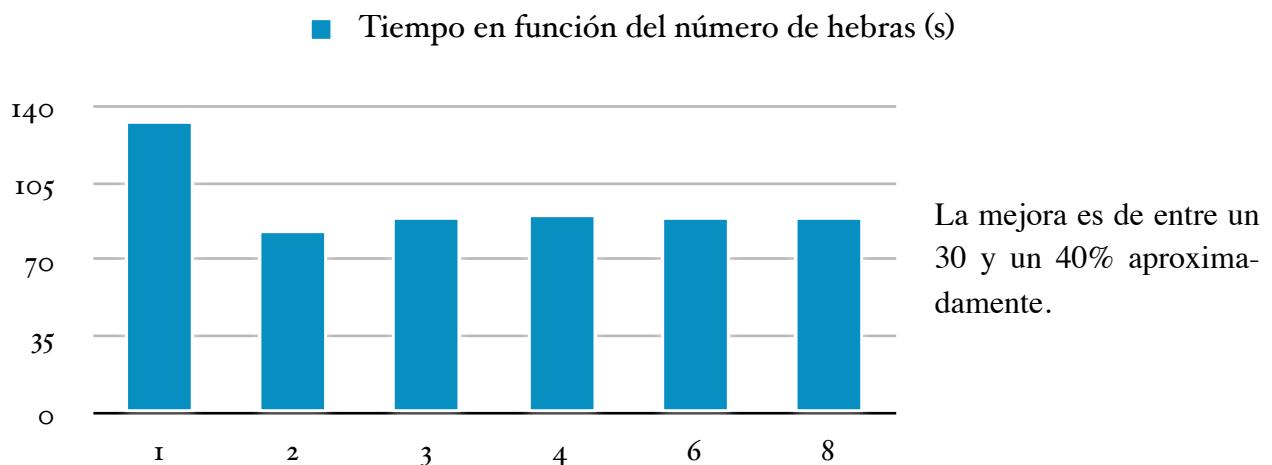


Se puede pensar que lo ideal es tener el mismo número de hebras que de núcleos en el procesador. Sin embargo si, por ejemplo, el procesador tiene dos núcleos, es probable que la carga de proceso de la hebra de una de las mitades sea mucho menor que la de la otra. En ese caso una de ellas estaría ociosa la mayor parte del tiempo mientras que la otra soporta casi todo el render realmente.

Se podría reutilizar la hebra que ha terminado para “ayudar” a la otra, pero es mucho más complicado. De modo que lo más recomendable es dividir en más hebras la escena, siendo el planificador del sistema operativo el que se encargue de asignar cada una a cada núcleo.

El número de hebras hay que especificarlo en el atributo *threads* de la clase *RayTracer*.

A continuación se muestran los tiempos de render con 1, 2, 3, 4, 6 y 8 hebras en un Core2Duo 2.0GHz con 4GB de RAM, a 768x768, 9 muestras por pixel y 14 esferas que llenan la imagen, para un total de 5.308.416 rayos (*testBench*):



2.2.2 Cajas contenedoras alineadas a los ejes

Hemos incluido el cálculo de intersecciones con las cajas contenedoras de las formas Sphere, Cylinder y Mesh. Para ello almacenamos en la figura sus coordenadas mínimas y máximas en cada eje.

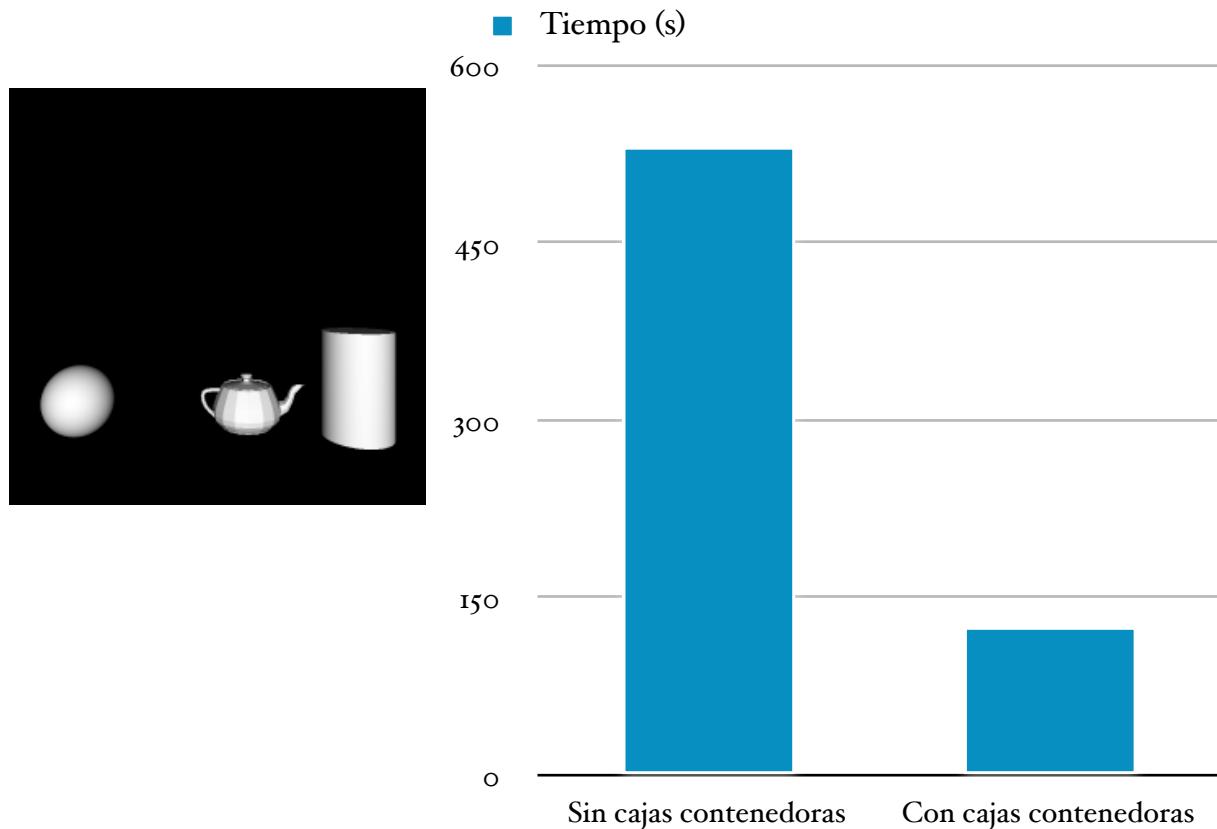
Antes de comprobar si un rayo interseca con una figura se averigua si lo hace con su caja contenedora. En el caso de no ser así no se continua con la comprobación.

En los cilindros no alineados al eje Y la caja contenedora es algo más grande de lo normal. Concretamente, es la equivalente a tener además dos esferas centradas en los centros de cada tapa.

El rendimiento de utilizar las cajas contenedoras, como es de suponer, depende mucho de la porción de imagen que ocupe la figura y de lo ajustada que esté a ella.

Las mejoras son mucho mayores en las escenas en las que las figuras más complejas ocupan menos espacio en la imagen final, y cuanto más complejas sean.

Estos son los tiempos empleados en generar la misma escena con y sin cajas contenedoras (*testBBox.xml*):



La mejora en este caso es bastante apreciable. Un 75%

2.3 Materiales

2.3.1 Mapeado de texturas

2.3.1.1 Cálculo de coordenadas UV

Las coordenadas de textura dependen de la figura sobre la que se apliquen. En el caso de los triángulos y los quads es necesario especificarlas en el archivo XML.

En el caso de la esfera el mapeado es esférico, y en el del cilindro cilíndrico. Para ello, las coordenadas de textura se asocian a los ángulos respecto del centro y del eje respectivamente.

En el caso de las mallas es necesario un archivo más que contenga las coordenadas para cada vértice en cada triángulo. Dado que los archivos OFF no pueden almacenar coordenadas de textura, pero los VRML sí, hemos hecho un script en Python que a partir del archivo .WRL genera la lista de coordenadas de textura. Dicho archivo es fácil de generar con Meshlab.

Por lo tanto, una manera de obtener la malla y las texturas podría ser:

1. Exportar la malla con las coordenadas de textura asociadas desde tu programa favorito en formato, por ejemplo .3DS.

2. Abrir dicha malla en Meshlab y guardarla en formato VRML y OFF.

3. Ejecutar el script:

```
python genUV.py entrada.wrl salida
```

Las texturas se cargan en memoria al inicio de la ejecución. Hemos decidido que no merecía la pena implementar el MipMapping dado que el acceso a las texturas en alta resolución no consume tanto tiempo comparado con el resto de las tareas.

Las texturas de color se asocian a los materiales y se pueden escalar y desplazar.

Para usarlas hay que añadir un nuevo nodo a la escena:

```
<textures>
  <Texture>
    <file>PATH</file> <!-- archivo de textura -->
    <filter>1</filter> <!-- 1 para filtrar 0 no -->
  </Texture>
</textures>
```

Después hay que asociar la textura al material:

```
<material class=...>
  <texture>0</texture> <!-- Orden de la textura en la lista -->
  <toffset>ou ou xx</toffset> <!-- Desplazamiento (opcional) -->
  <tscale>su su xx</tscale> <!-- Escalado (opcional) -->
</material>
```

En el caso de las mallas añadir además:

<UVfile>PATH</UVfile>

En el de los triángulos:

<v0t>u0 v0 xx</v0t>

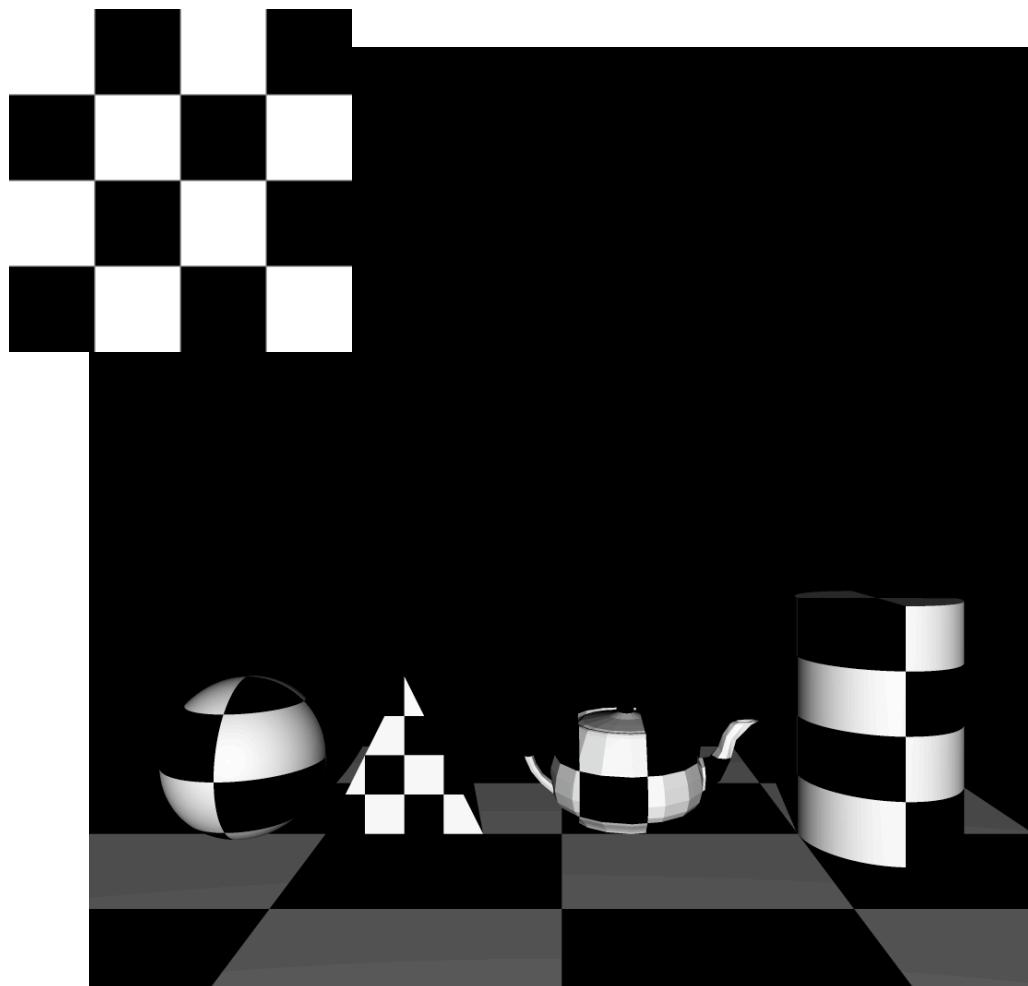
<v1t>u1 v1 xx</v1t>

<v2t>u2 v2 xx</v2t>

Y en el caso de los quads:

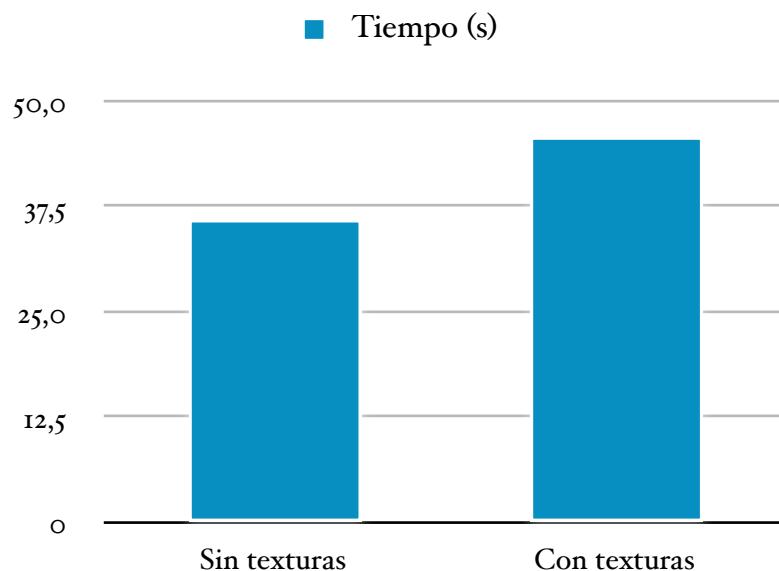
<v3t>u3 v3 xx</v3t>

A continuación se muestra una misma textura mapeada sobre todos los tipos de geometría:



Texturas
(testTex01)

La penalización introducida por el uso de texturas (es decir, cálculo de UV+fetching) es (*testTex03.xml* y *testTex04.xml*):

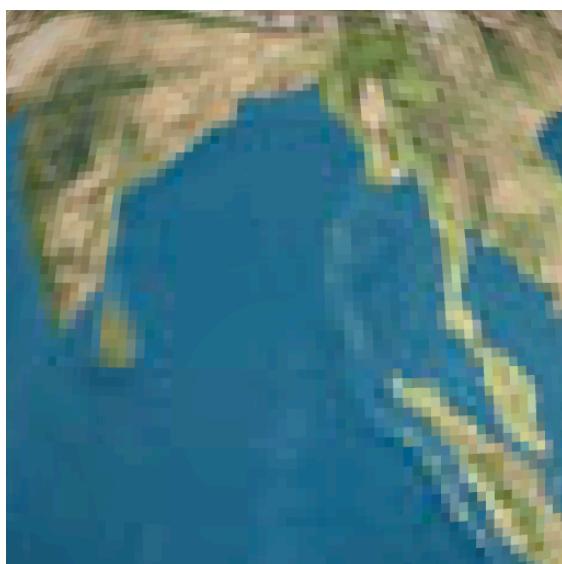


Es algo más de un 25%.

2.3.1.2 Filtrado lineal

Existe la posibilidad de activar o desactivar el filtrado lineal para las texturas. Es decir, tomar cuatro muestras en vez de una e interpolarlas linealmente.

La diferencia en la calidad final es:

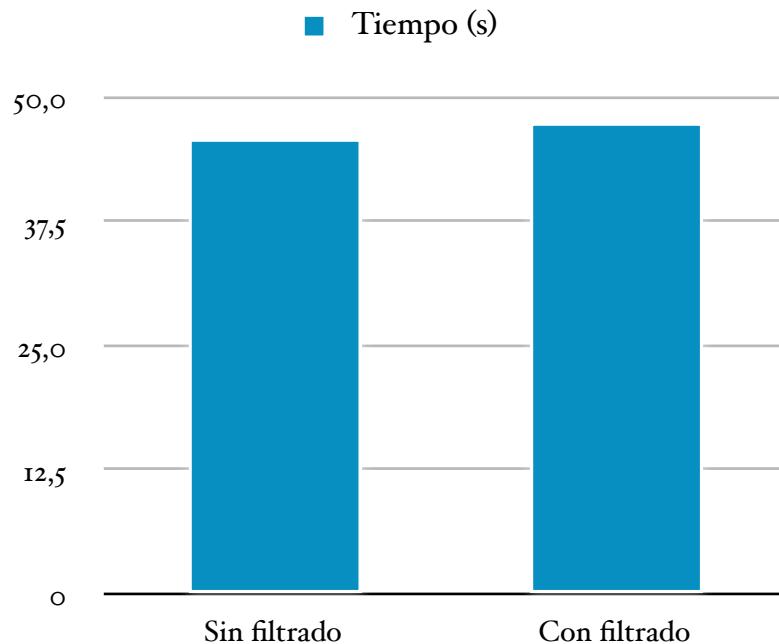


Sin filtrado lineal



Con filtrado lineal

Mientras que la penalización es la siguiente (*testTex02.xml*, *testTex03.xml*):



No llega al 4%.

2.3.1.2 Mapas de normales

Están asociadas a las figuras y también se pueden escalar y desplazar, al igual que las de color.

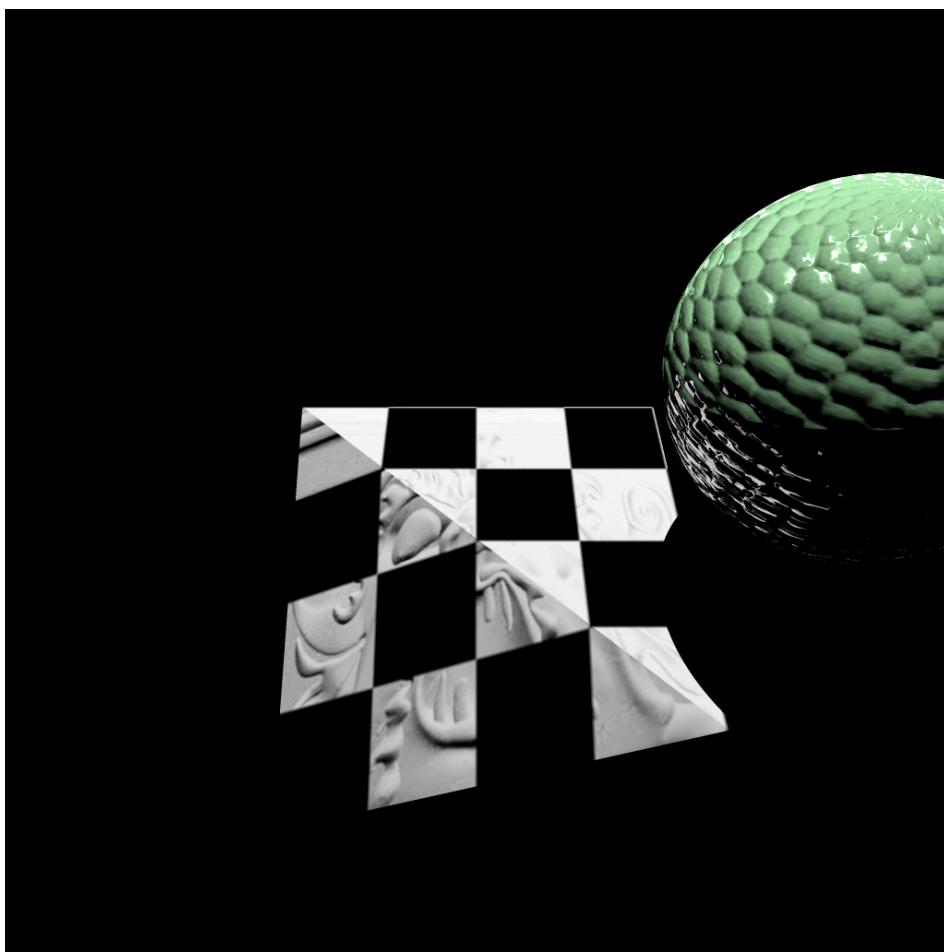
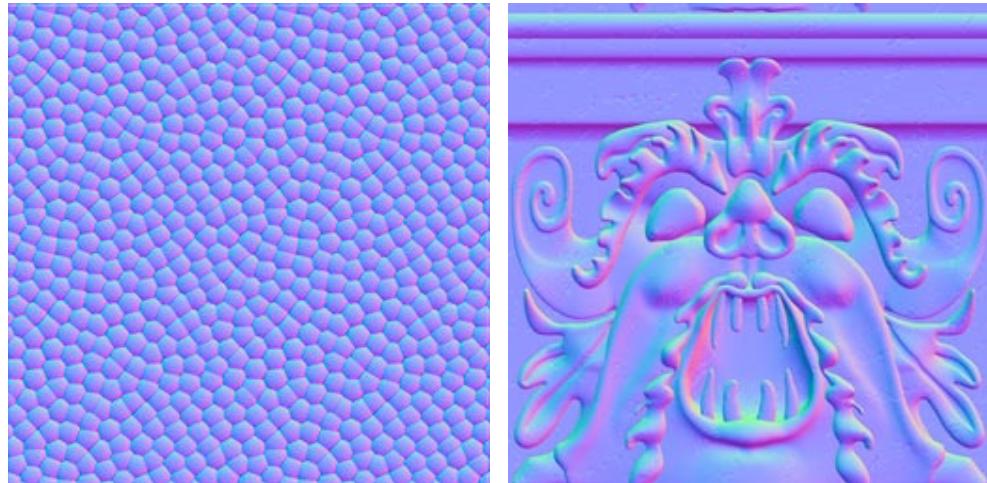
Una vez calculada la normal en el punto de intersección, se obtiene el valor de la textura y se desplaza en los ejes X e Y (respecto de la normal).

Una vez elegidas las texturas y sus coordenadas, hay que añadir en la figura lo siguiente:

```
<bTexture>0</bTexture> <!-- Orden de la textura en la lista -->
<toffset>ou ou xx</toffset> <!-- Desplazamiento (opcional) -->
<tscale>su su xx</tscale> <!-- Escalado (opcional) -->
```

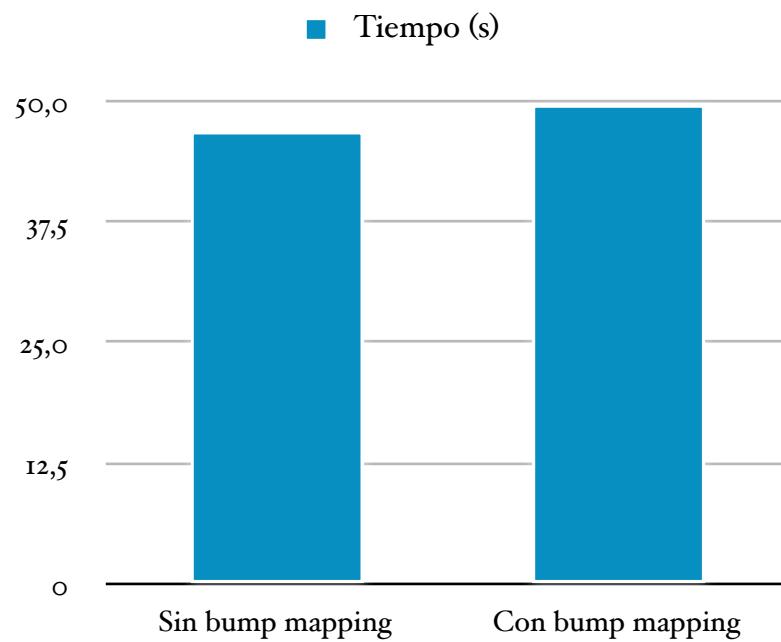
Hay que tener cuidado al modificar las coordenadas UV porque los cambios hechos al bump mapping afectan también a la textura de color (pero no al revés).

El resultado es el siguiente:



*Mapas de normales
(testBump)*

Y la penalización introducida, exclusivamente por la consulta a textura (con filtrado lineal) y desplazamiento de la normal (*testBump01.xml*, *testBump02.xml*):



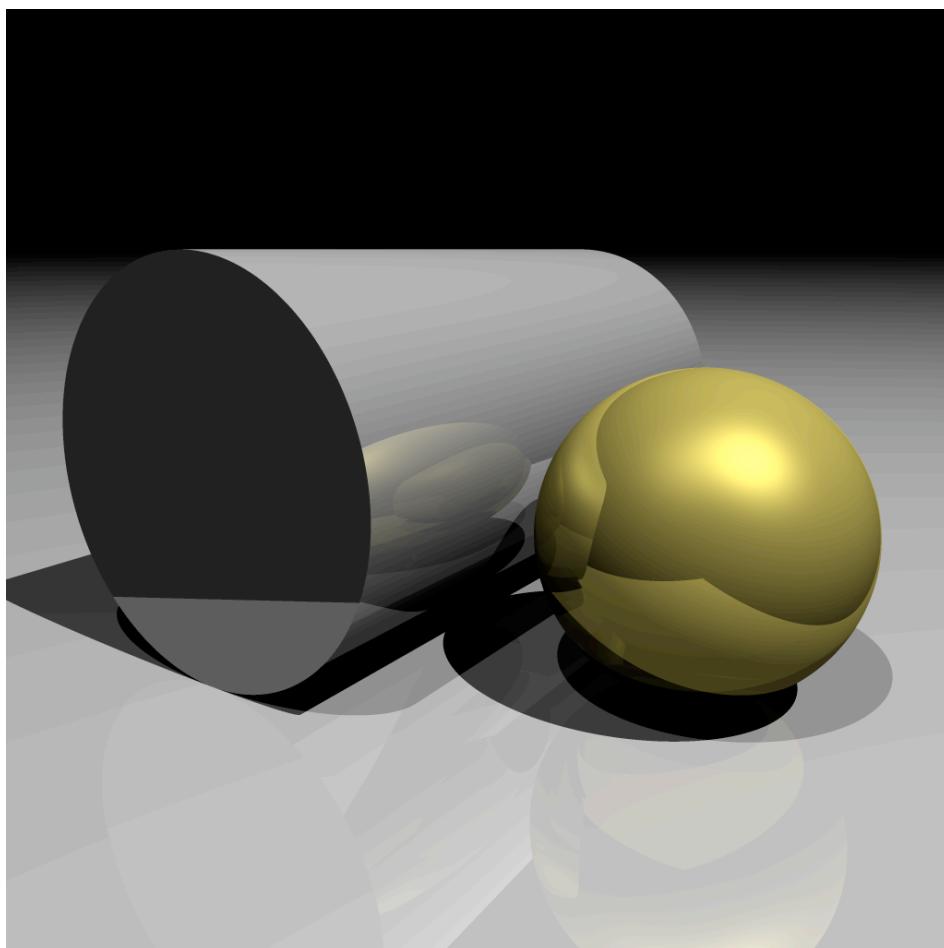
Es de apenas un 6%.

2.3.1 MetalPhong (reflejos metálicos)

El material MetalPhong es un material tipo ShinyPhong en el que las todas sus componentes son tintadas según su color difuso.

Se utilizan de la misma manera que las ShinyPhong.

Queda de la siguiente manera:



Materiales metálicos

(*testMetal*)

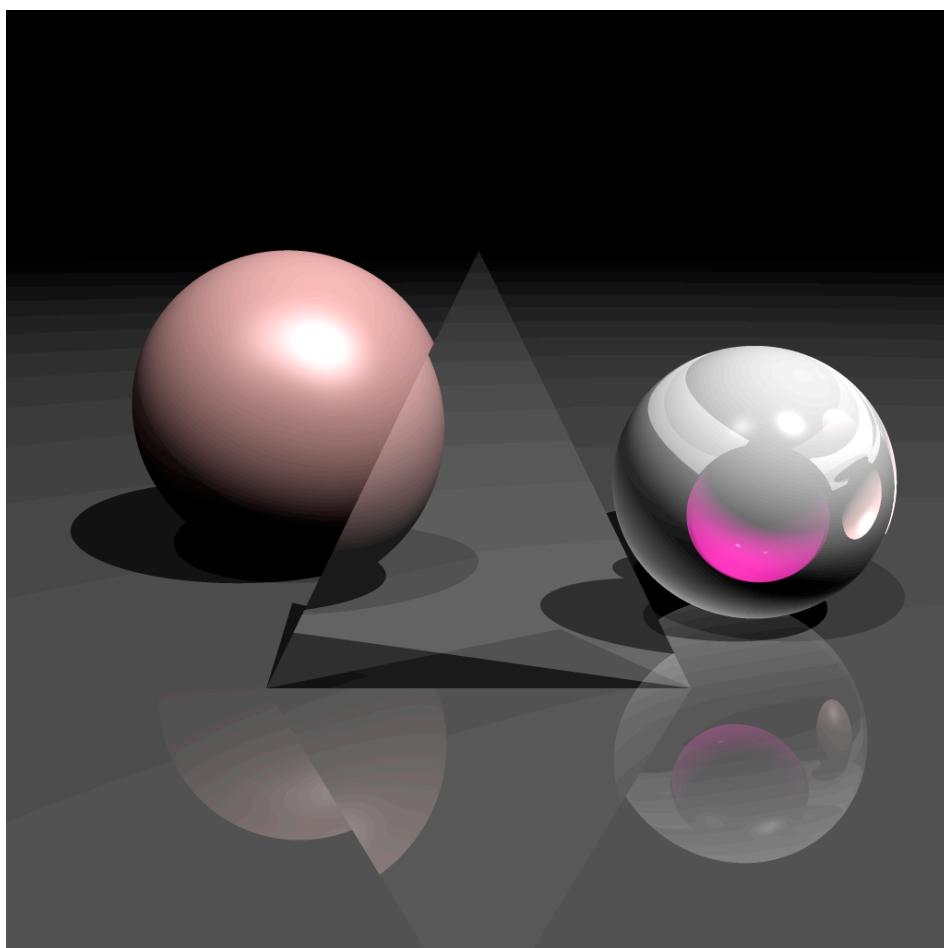
2.3.1 ShinyRefractionPhong (refracción)

Este material añade la desviación de los rayos basándose en el coeficiente de refracción aplicado en la ecuación de *Snell*, que relaciona los coeficientes de refracción y los ángulos cuando un rayo cambia de medio. Si el coeficiente de refracción es equivalente al del medio entonces actúa como un objeto al que se le puede variar la transparencia.

Para utilizarlas se llaman de la siguiente forma:

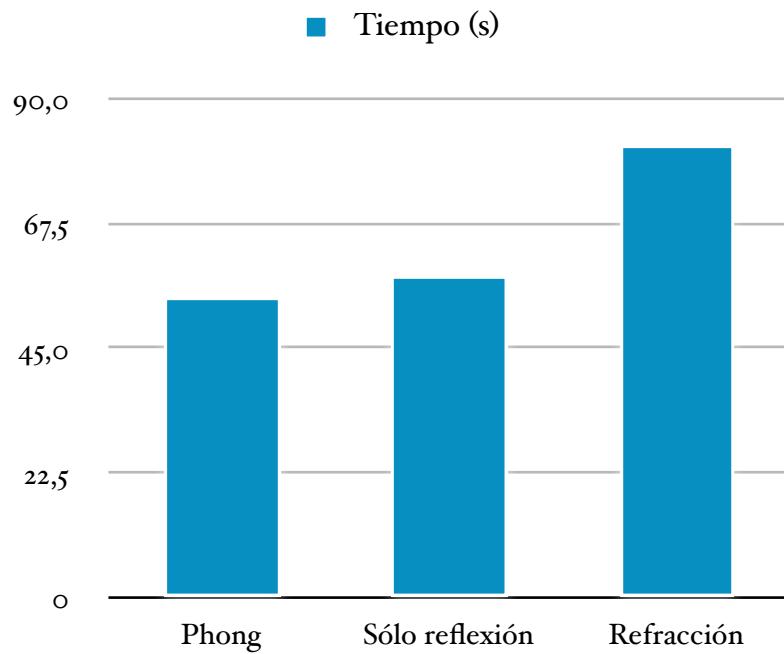
```
<material class="RefractionShinyPhong">
    <diffuse>1 1 1</diffuse>
    <specular>0.7 0.7 0.7</specular>
    <exponent>100.0</exponent>
    <refraction>rfrr rfrg rfrb</refraction> <!-- Intensidad -->
    <refractionCoef>1000</refractionCoef> <!-- coef. de refracción-->
    <reflection>0 0 0</reflection>
</material>
```

Un ejemplo de refracción provocados por un triángulo y una esfera con coeficientes superiores a 1:



*Refracción
(testRef)*

La penalización provocada por utilizar este efecto es (*testRef01.xml*):



La reflexión supone un 7% de penalización, mientras que la refracción lo eleva a un 50% (40% sobre la reflexión).

2.2 Luces

2.2.1 AreaLight (sombras suaves)

Las luces de área las hemos implementado de manera que cuando se calcula la luz actúa como una PointLight, y cuando se calculan las sombras actúa como `nsamples3` luces.

La luz tiene asociada una figura de cualquier tipo, pero en última instancia las posiciones a partir de las cuales se calculan las sombras vienen determinadas por su caja contenedora.

Hay dos formas de utilizarlas:

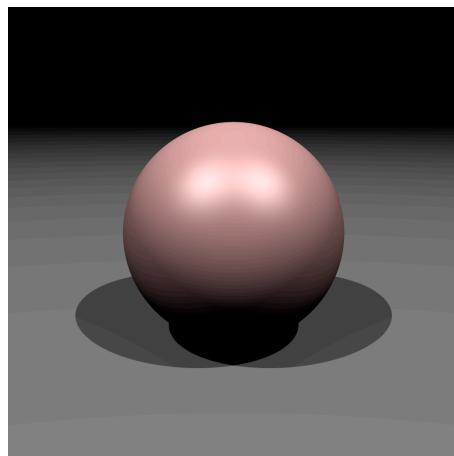
Dando todos los detalles de la figura que la representa:

```
<light class="AreaLight">
    <intensity>0.75 0.75 0.75</intensity>
    <position>6.0 12.0 -6.0</position>
    <object class="...">
    </object>
</light>
```

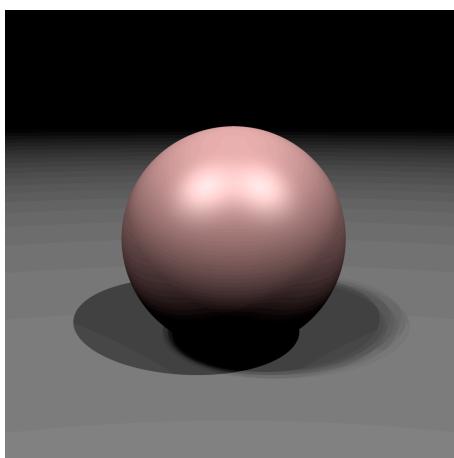
Eligiendo una esfera o un quad y su tamaño, sin necesidad de especificar toda la figura:

```
<light class="AreaLight">
    <intensity>0.75 0.75 0.75</intensity>
    <position>6.0 12.0 -6.0</position>
    <shape>Quad</shape> <!-- quad o sphere -->
    <size>2.0</size> <!-- tamaño -->
</light>
```

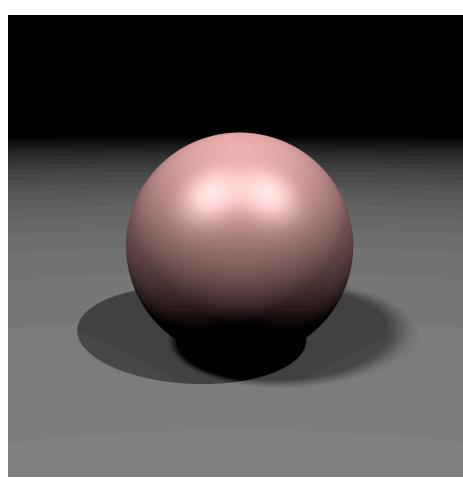
El resultado es el siguiente:



*Luces puntuales
(testArea1)*

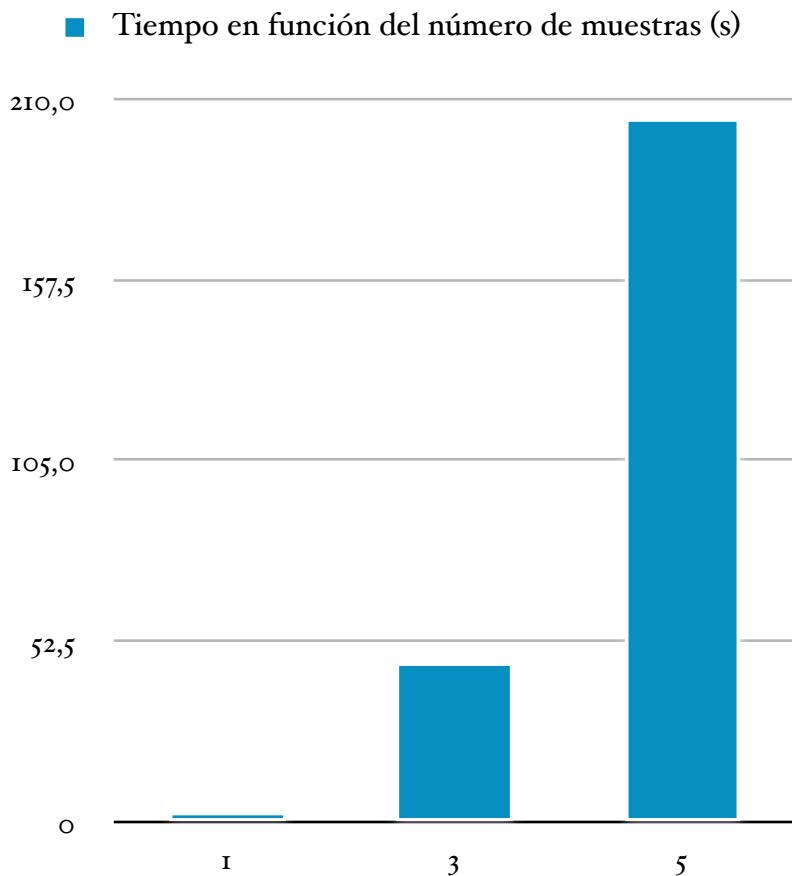


*Luz puntual y luz de área con nsamples=3
(testArea3)*



*Luz puntual y luz de área con nsamples=5
(testArea5)*

La penalización, según el número de muestras (*testArea1.xml*, *testArea3.xml* y *testArea5.xml*):

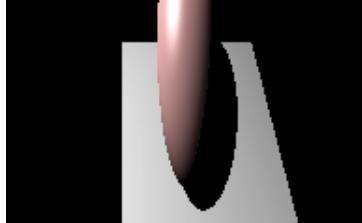


Son penalizaciones muy graves, 15 y 64 veces más lento utilizando 3 y 5 muestras respectivamente. Es decir, 4.5 veces más lento al pasar de 3 a 5 muestras. Tiene sentido ya que $5^3/3^3$ es aproximadamente ese valor.

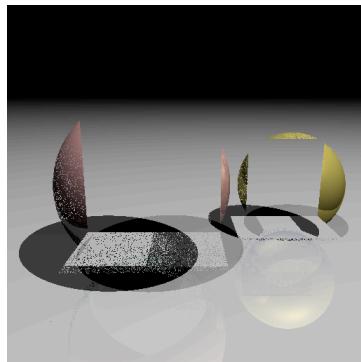
3. Algunas imágenes

A continuación mostramos algunas de las escenas en alta resolución que hemos generado con nuestra versión del trazador de rayos.

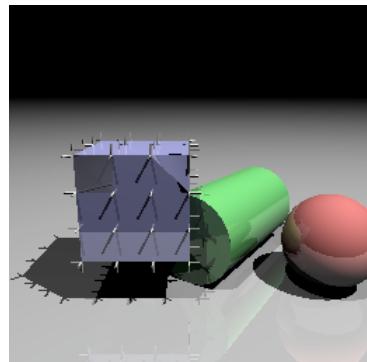
En la carpeta con el código también hay una gran parte de las imágenes generadas a lo largo del desarrollo (más de 1500) hasta que hemos cumplido cada objetivo. Si se ordenan por fecha de creación dan una idea bastante buena de cómo ha sido todo el proceso. Por ejemplo:



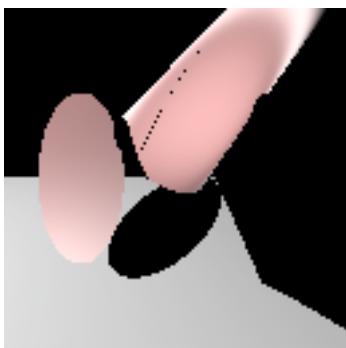
Corrigiendo el aspecto



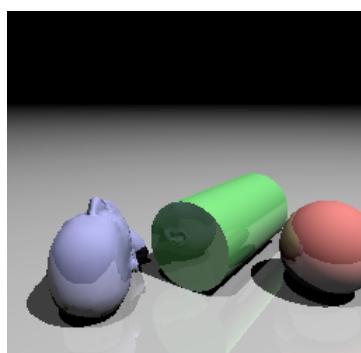
Incorporando cajas contenedoras



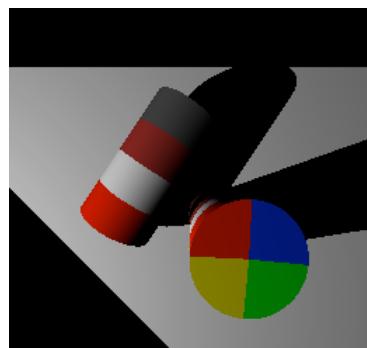
Visualizando las normales



Creando los cilindros

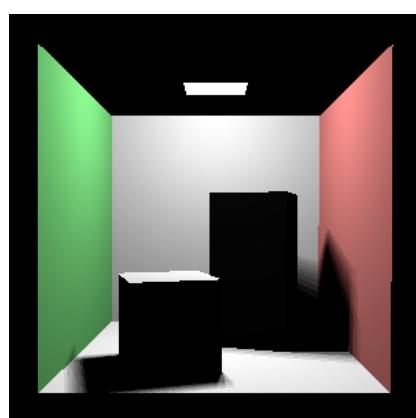


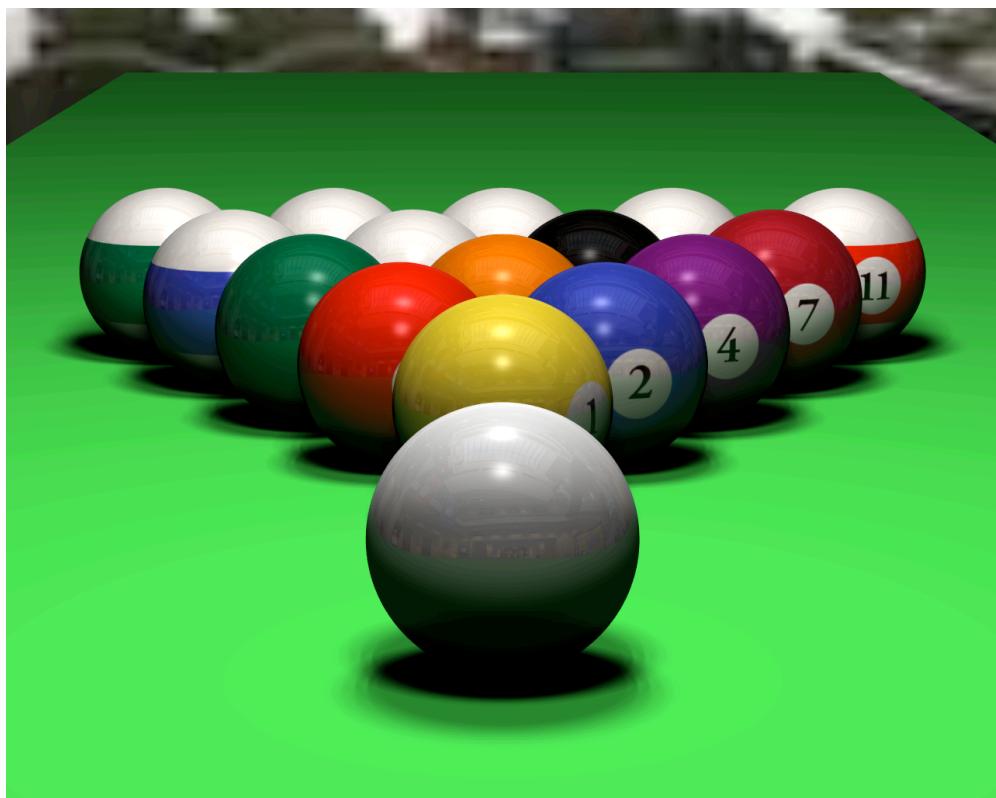
Jugando con las mallas
(es una cabeza)



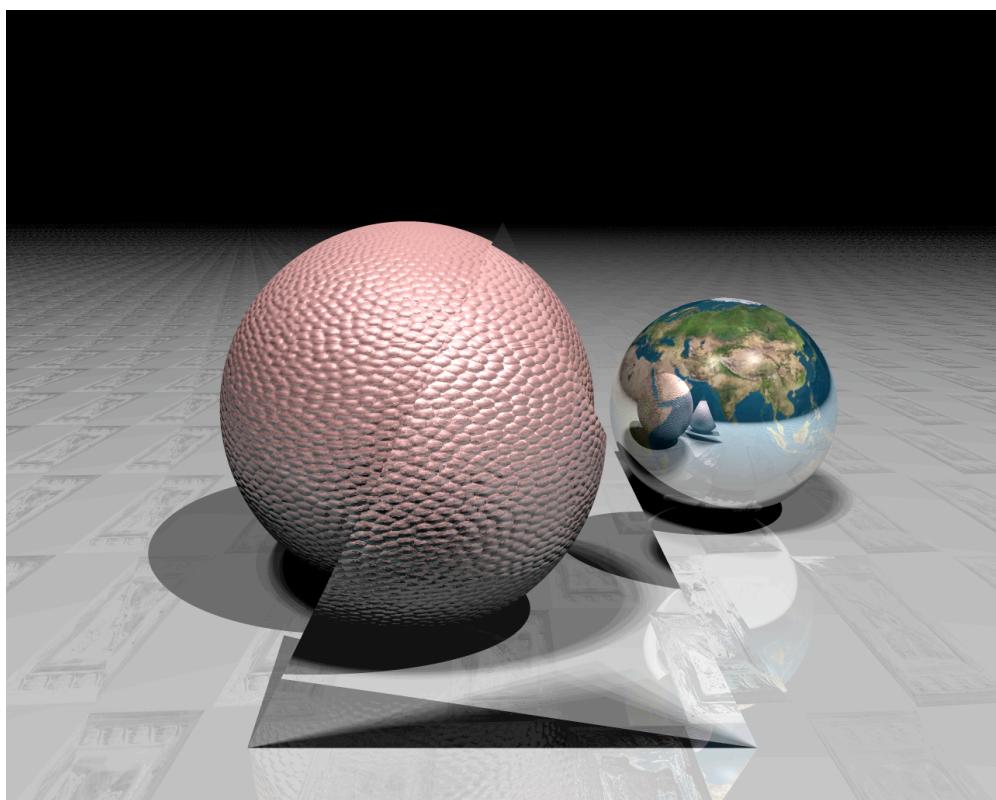
A vueltas con las texturas

Como curiosidad hemos diseñado también una escena con la caja de Cornell, pero al no haber iluminación indirecta el resultado no es demasiado bueno:

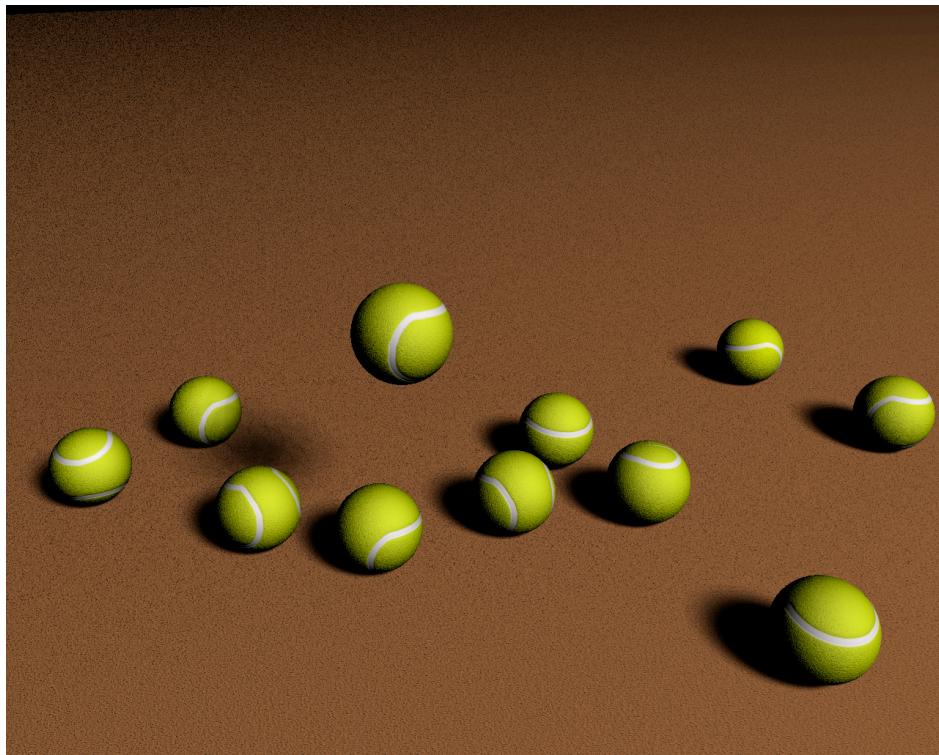




testBillar



testMirror



testTenis



testBasket