Proyecto Final de Carrera – Tipo A

Visualización realista por trazado de rayos

Realizado por:
Josep Maria Tomas Sendra

Dirigido por:
Roberto Agustín Vivó Hernando

Índice

1. Motivación

El visualizado realista de objetos 3d es un proceso muy útil hoy en día en muchos sectores comerciales, como por ejemplo empresas que desean obtener una representación realista de su producto en formato digital.

Este proceso también es realmente útil, si no necesario, dentro del campo del ocio, desde películas o cortos de animación, efectos especiales en películas de acción real o la representación de diversas figuras en 3d de manera realista para un uso artístico.

Cada vez más aplicaciones de gráficos por computador en tiempo real emplean métodos de trazado de rayos para representar un gran abanico de efectos, por ejemplo reflejos. También conforme avanza la tecnología, en especial las GPU, comienzan a verse los primeros ejemplos de visualizadores en tiempo real basados completamente en trazado de rayos.

Partiendo de los conocimientos previos que se tienen sobre el trazado de rayos, este proyecto servirá para profundizar en el mundo del trazado de rayos.

2. Problema

Se plantea realizar un programa que sea capaz de obtener imágenes realistas de una escena compuesta por diferentes tipos de objetos.

Para este problema han ido surgiendo diferentes soluciones a lo largo del tiempo.

Ray Tracing

Para este método se parte de un punto de vista, o cámara, y un plano en el espacio que representa la imagen resultante. El primer paso consiste en trazar rayos desde la cámara hacia cada píxel de la imagen, rayos de vista, y calcular las intersecciones de este rayo con los objetos de la escena. La intersección más cercana corresponderá al objeto más cercano. Con el punto obtenido se trazan rayos en dirección a las fuentes de luz, rayos de sombra, para obtener la iluminación en el punto. Adicionalmente se pueden trazar rayos para calcular reflejos, refracciones, iluminación indirecta u otros.

Con este método se obtiene imágenes muy buenas, más aún con modelos concretos de iluminación y materiales con propiedades físicas.

La principal contra de este método es que se necesitan trazar muchos rayos para que la calidad de la imagen resultante sea alta. Con un número de rayos bajo, ciertos cálculos, como la iluminación indirecta, provocan que el resultado sea una imagen con ruido.

De este método parten otros como *Path Tracing*.

Photon Mapping

Este método consiste en emitir fotones desde las fuentes de luz de la escena. Los fotones chocarán con los elementos de la escena y podrán ser reflejados, refractados o absorbidos. Los fotones pueden rebotar un número determinado de veces y una vez acaban, se añaden a una estructura de tipo árbol-Kd.

El proceso de visualización consiste en trazado de rayos para encontrar el objeto más cercano, pero la iluminación aquí se calcula a partir de los fotones más cercanos recorriendo el árbol.

Point Based Rendering

Se basa en construir, a partir de la escena, una nube de puntos que representa una aproximación de la geometria de la escena. Cada punto contiene información sobre la geometría, el color y la iluminación en ese lugar de la escena. Posteriormente, cuando se quiere obtener la imagen resultante, el color de cada pixel se deduce a partir de varios puntos de la nube.

Este método ha demostrado ser muy rápido para calcular oclusiones entre objetos e iluminación indirecta.

En este proyecto se va a utilizar el método de trazado de rayos.

3. Solución

3.1 Introducción

En este punto se va a explicar la solución ideada para resolver el problema. Se van a exponer los elementos que componen la solución desde un punto de vista general.

3.2 Diseño

La solución consiste en la realización de un software que sea capaz de, mediante el método de trazado de rayos, obtener una imagen realista de una escena definida mediante un fichero de entrada.

La solución diseñada consta de varias etapas.

- En primer lugar se realizará la lectura del fichero escena. Se construirá la escena al mismo tiempo que se va leyendo el fichero de entrada. En esta etapa la escena reside en la memoria principal del procesador.
- Una vez finalizada la lectura y construcción de la escena se pasará a transferir la escena a la memoria global de la GPU.
- En este punto la GPU ejecutará un proceso que realizará el trazado de rayos sobre la escena, asignando el valor de cada píxel a la imagen resultado.
- La imagen resultado se guardará en un archivo y después se mostrará por pantalla.

Estos pasos se ilustran en la siguiente figura.

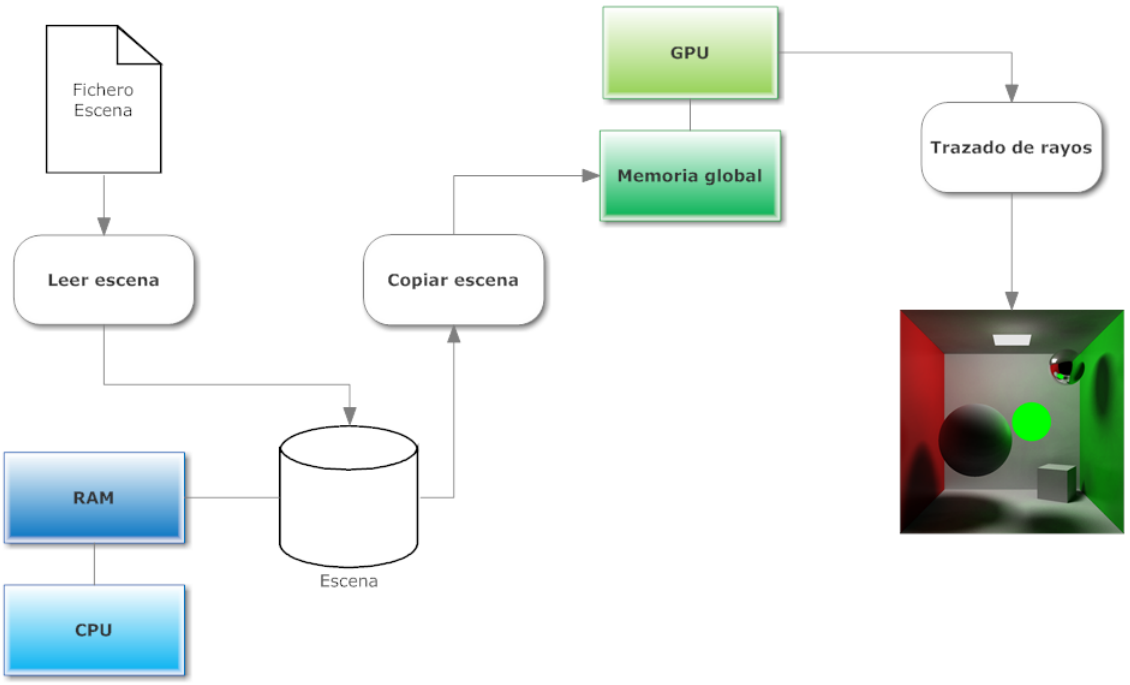


Figura 1. Etapas de la solución.

3.3 Trazado de rayos

En este punto se va explicar cómo se ha planteado el proceso de trazado de rayos así como cada característica que se va a calcular.

3.3.1 General

Se va a partir del algoritmo de trazado de rayos de Whitted, cuyo pseudocódigo se encuentra a continuación, y se ampliará con nuevas funcionalidades.

Para cada pixel traza un rayo primario de dirección V desde la cámara hacia la escena
Recorrer todos los objetos de la escena y obtener la intersección de mínima distancia
Si existe intersección
Traza rayos hacia las fuentes de luz con dirección L y calcular iluminación
Traza rayos reflejados con dirección R y calcular reflejos
Traza rayos refractados en dirección T y calcular refracciones
Combinar los elementos anteriores y devolver el color en ese píxel
Si no existe intersección
Devuelve el color de fondo

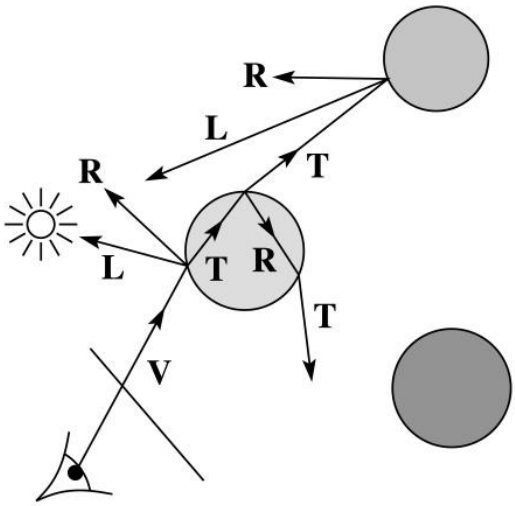


Figura 2. Esquema de los rayos trazados por este algoritmo.

En este documento se va a considerar intersección entrante a la intersección más cercana de un objeto e intersección saliente a la intersección más lejana, tal y como se muestra en la siguiente figura.

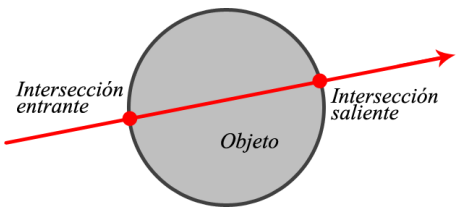


Figura 3. Intersección entrante y saliente en una esfera.

3.3.2 Visibilidad

Es el primer paso en el proceso de trazado de rayos y se ha implementado posteriormente de la siguiente manera siguiendo el esquema de Whitted.

- Primero se trazan rayos desde la cámara hacia cada píxel. A estos rayos los llamaremos rayos principales.

- Cada rayo recorrerá la escena y calculará las intersecciones con los objetos que la componen quedándose únicamente con la intersección de menor distancia, la más cercana.

A partir de este punto termina la fase de visibilidad y según si se ha encontrado o no intersección se realizará lo siguiente.

- Si el rayo intersecta con algún objeto, para el punto correspondiente a la intersección de menor distancia se efectuarán los cálculos necesarios para obtener el color en ese punto.

- Si el rayo no intersecta, se asignará el color de fondo de la escena al color del píxel.

3.3.3 Iluminación

A partir del punto de intersección más cercano, uno de los cálculos que se van a realizar consiste en calcular si este punto está iluminado por alguna fuente de luz y en cuanta medida. En esta sección vamos a distinguir dos tipos de iluminación.

3.3.3.1 Sombreado

El proceso de sombreado consiste en alterar el color del objeto en base a un vector de luz y una intensidad dadas.

Existen numerosos algoritmos para cumplir este propósito. Entre todos ellos se han elegido los siguientes debido a que se basa en utilizar propiedades físicas para obtener una mejor representación.

- Para la componente difusa se ha elegido el algoritmo de Oren-Nayar.
- Para la componente especular se ha elegido el algoritmo de Cook-Torrance.

3.3.3.2 Iluminación directa

En este tipo de iluminación se ha de calcular si una fuente de luz ilumina directamente un objeto.

Desde el punto de intersección más cercano se lanzan rayos con dirección a las fuentes de luz, rayos de sombra. Posteriormente se ha de recorrer la escena y comprobar si hay algún objeto que intersecta con este rayo, es decir, si existe algún objeto entre el punto y la luz.

Si no existe ningún punto de intersección eso significa que el punto está iluminado por la luz. En este momento se calcula el sombreado mediante el algoritmo que hemos elegido. Este proceso se repite para cada luz, siendo la iluminación en el punto la suma de este cálculo para todas las luces de la escena.

3.3.3.3 Iluminación indirecta

Dado un objeto de la escena, su iluminación indirecta corresponderá a la luz que rebota en otros objetos y luego incide sobre éste.

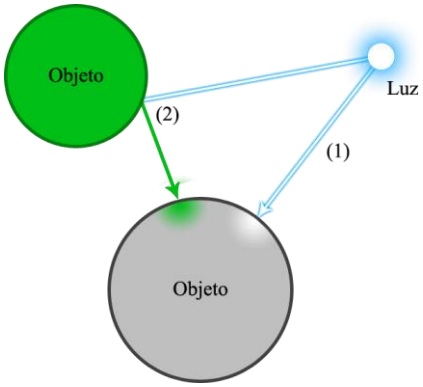


Figura 4. Tipos de iluminación sobre un objeto.

La figura muestra un rayo que ilumina directamente (1) y un rayo que ilumina indirectamente (2).

Para obtener la iluminación indirecta en un punto se ha pensado el siguiente procedimiento.

- Partiendo del punto de intersección se trazan un número determinado de rayos con forma de semiesfera.
- Cada uno de estos rayos recorrerá la escena del mismo modo que un rayo principal y se obtendrá un punto de intersección, si se da el caso.
- Si existe intersección.
    - Se calculará la iluminación directa en ese punto del objeto, resultando en un color determinado.
    - Se considera este nuevo punto de intersección como una fuente de luz siendo su intensidad el color calculado en el paso anterior y se calcula la iluminación directa sobre el punto original.
    - La iluminación indirecta calculada para cada rayo será proporcional al número de rayos, es decir, que si trazamos 200 rayos, cada rayo solo podrá aportar 1/200 de la iluminación indirecta a ese punto.
- Si no existe intersección.
    - No se hace nada más con ese rayo.

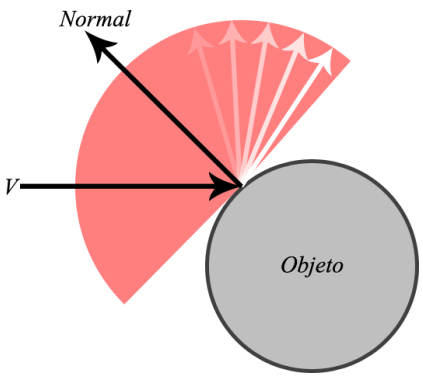


Figura 5. Rayos trazados en iluminación indirecta.

3.3.4 Reflejos

3.3.4.1 Reflejos simples

Dado un rayo de dirección V, un punto P de intersección en el objeto y el vector normal N de la superficie en ese punto, los reflejos se calculan de la siguiente manera.

- A partir de V negado y N se calcula el vector de reflexión R.
- Se traza un rayo con origen en P y dirección igual a R.
- Se aplica el algoritmo de trazado de rayos con este rayo (calculando todos los elementos: iluminación, reflejos, etc.) sobre la escena.
- El color resultante del anterior paso será el reflejo.
- Finalmente el color reflejado se multiplica por el color especular en ese punto.

Es posible que el rayo R al intersectar con otro objeto sea reflejado de nuevo, si se da el caso de tener dos objetos reflejantes. Esto implica que el calcular los reflejos es un proceso recursivo. Para limitar la recursividad se ha fijado un número máximo de rebotes, que se especifica en el fichero escena, que se pueden efectuar en el cálculo de reflejos.

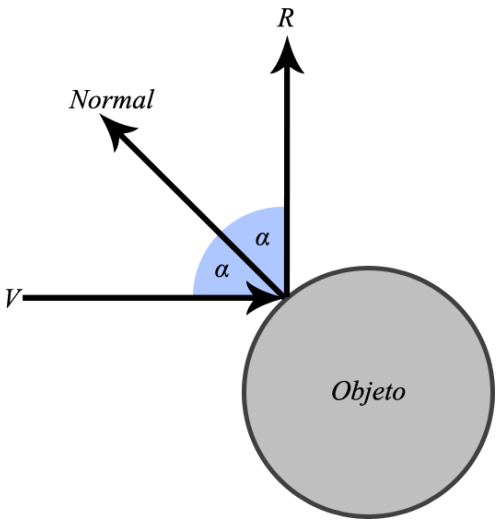


Figura 6. Reflejo simple.

3.3.4.2 Reflejos con ángulo sólido

La obtención de reflejos con ángulo sólido se basa en el proceso de obtener varios reflejos simples.

- En primer lugar se obtiene el vector R como en la sección anterior.
- A partir de este vector trazamos un número determinado de puntos dentro de un ángulo sólido especificado.
- El resultado de cada rayo se va añadiendo y se obtiene un color.
- Este color se divide entre el número de rayos para obtener el color final.

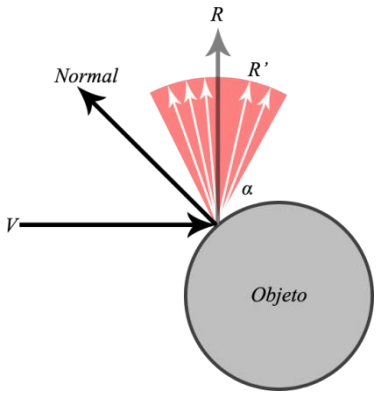


Figura 7. Reflejo con ángulo sólido.

3.3.5 Refracciones

Cuando un rayo pasa de un medio a otro con distinta densidad éste se desvía. Para calcular este fenómeno se necesita que el objeto no sea opaco. La refracción se calcula siguiendo estos pasos.

- En primer lugar, teniendo el punto P de la intersección entrante, el vector V correspondiente a la dirección del rayo y la normal N a la superficie se calcula, mediante la ley de Snell, el rayo de dirección T transmitido.
- Desde el punto P se traza un rayo de dirección T dando como resultado las siguientes opciones.
    - El rayo intersecta con otro objeto que se encuentra total o parcialmente dentro del transparente.
    - El rayo atraviesa el objeto y se desvía de nuevo, T'. Se calcula la nueva dirección del rayo mediante la ley de Snell y se traza sobre la escena para obtener el color. Durante el cálculo de T' se puede detectar el fenómeno de reflexión interna, en cuyo caso el rayo no se desviará sino que rebotará en la superficie interna.

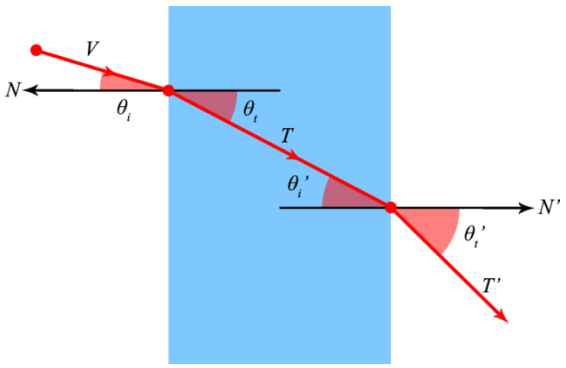


Figura 8. Rayo atravesando un objeto transparente con un índice de refracción > 1.

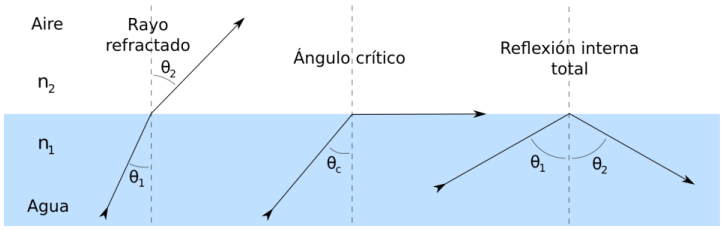


Figura 9. Fenómeno de la reflexión interna (Fuente: Wikipedia).

3.3.6 Antialiasing

En imagen digital aparecen una serie de artefactos en una imagen cuando se tiene que representar una figura de infinitos puntos, como cualquier primitiva gráfica, en un espacio finito, en este caso una imagen.

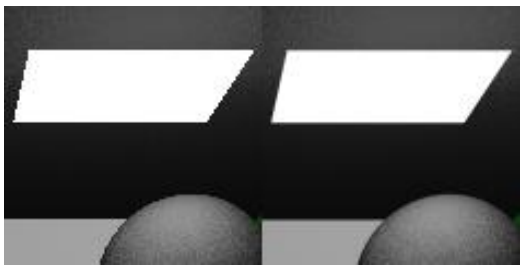


Figura 10. Imagen con aliasing (izquierda) frente a imagen sin aliasing (derecha).

Para eliminar estos artefactos se utilizará la técnica de antialiasing por sobremuestreo. Concretamente se va a poder utilizar sobremuestreo uniforme de 4 o 16 muestras por píxel. Esta técnica también permitirá eliminar ruido si estuviera presente en algunas partes de la imagen.

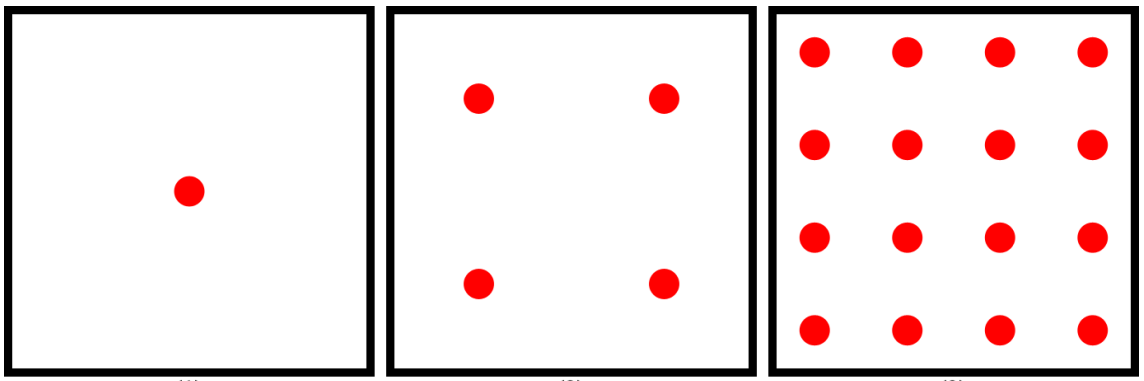


Figura 11. Diferentes muestras por píxel.

En esta figura los cuadrados representan un píxel y los círculos rojos representan las muestras, en el caso del trazado de rayos serían los rayos que se trazan. En (2) y en (3) se aprecia el muestreo uniforme, donde todas las muestras están equidistantes.

Para realizar el antialiasing se trazan los diferentes rayos para cada muestra dentro del píxel. El color resultado del trazado de cada rayo se va sumando en cada píxel. Posteriormente se realiza la media del valor final para el número de muestras definido.

El resultado final es una imagen de más calidad dependiendo del número de rayos, a más rayos más calidad.

3.4 Programación

Se ha decidido utilizar el modelo de programación CUDA C++ para implementar el trazador de rayos.

El trazado de rayos es un proceso altamente paralelizable debido a que cada rayo es independiente de los demás. Para aprovechar esto se ha decidido implementar este método de visualización mediante CUDA.

El modelo de programación de CUDA trata de aprovechar el gran paralelismo y ancho de banda de las GPU NVidia y se basa en lanzar un gran número de hilos de ejecución, *threads*, que realizan una misma tarea, *kernel*.

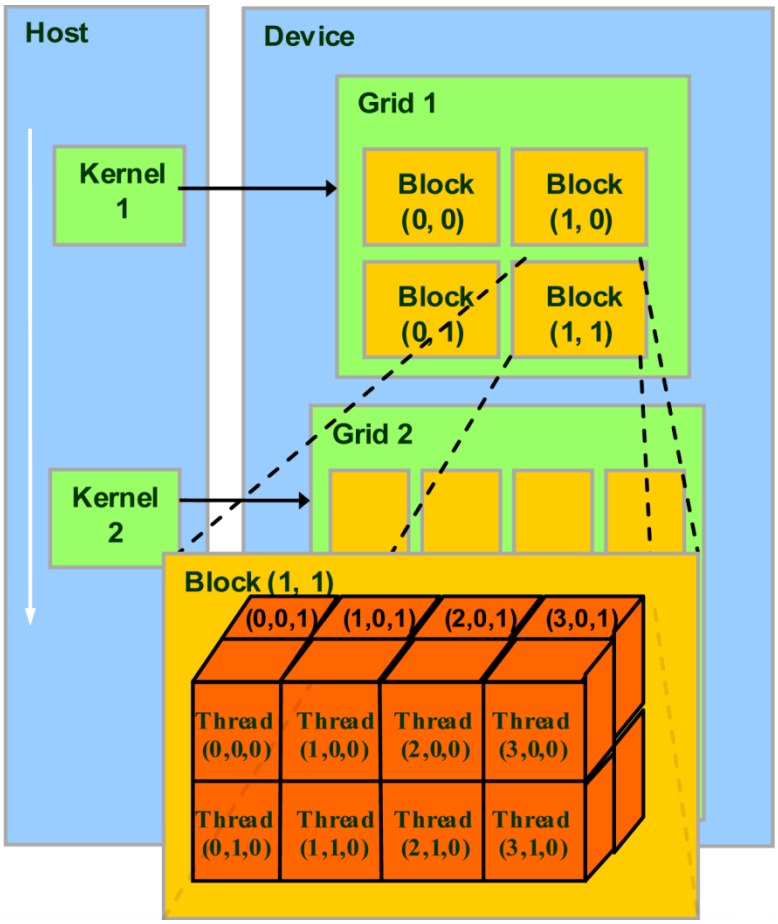


Figura 12. Esquema de bloques y threads en CUDA.

La figura muestra 2 partes diferenciadas (en azul).

- Host. Corresponde a la CPU y es la parte que se encarga de lanzar los kernels.
- Device. Es la GPU, donde se ejecuta cada kernel.

Los threads se lanzan agrupados en bloques, *Blocks*, y éstos pueden tener como máximo 3 dimensiones.

Los bloques se lanzan en lo que se llama *Grid*, una rejilla de bloques. Esta rejilla puede albergar bloques de ejecución en hasta 2 dimensiones siendo todos los bloques del mismo tamaño.

Cuando se lanza un kernel a ejecución se lanza con unos parámetros concretos: el número de bloques y el número de threads por bloque.

Las GPU tienen varios niveles de memoria, a continuación se enuncian de más rápido a más lento.

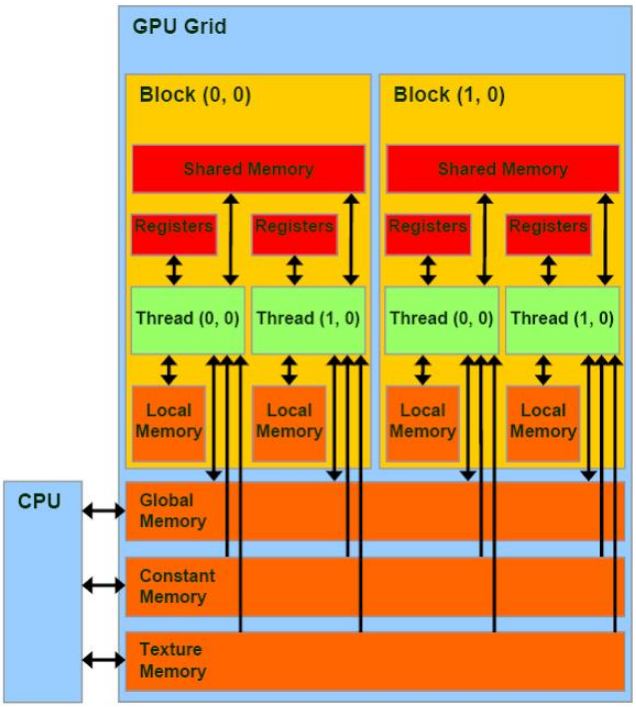


Figura 13. Niveles de memoria en GPU.

- Registros. Variables locales en el hilo de ejecución.
- Memoria local. Memoria que reserva un hilo.
- Memoria compartida. Es una memoria compartida únicamente por los hilos de un bloque.
- Memoria constante. Memoria sólo de lectura.
- Memoria de textura. Memoria optimizada para patrones de acceso 2d.
- Memoria global. Memoria de lectura/escritura.

4. Descripción de escenas

4.1 Introducción

En esta sección se va a explicar el formato de fichero creado para describir escenas. Se detallarán todos los posibles elementos que pueden conformar una escena, así como su representación dentro del fichero.

4.2 Fichero escena

El fichero que describe una escena será un fichero de texto plano. Dentro de este fichero se describirá la escena siguiendo unas pautas.

El contenido del fichero estará formado por bloques o módulos que representarán los distintos elementos de la escena. Cada módulo constará de un identificador, que indicará qué tipo de elemento es, y una serie de parámetros, formados por el nombre del parámetro y unos valores asociados que definirán el elemento. El conjunto de las propiedades irá delimitado por los caracteres "{" y "}". El siguiente ejemplo sirve para ilustrar la estructura de los módulos.

ElementoX
{
    Propiedad1 *valor1* *valor2* … *valorN*
    Propiedad1 *valor1* *valor2* … *valorN*
    …
    PropiedadN *valor1* *valor2* … *valorN*
}

Casi todos los parámetros de los elementos de la escena tendrán unos valores por defecto asignados, de manera que no será necesario indicar todas las propiedades en los distintos elementos. Tampoco importará el orden en el que se escriban los parámetros.

En el fichero escena se pueden escribir comentarios pero con la restricción de que solo pueden estar escritos fuera de los elementos (entre la declaración de un elemento y otro). Su formato consiste en el carácter "#" seguido de un espacio y el texto que se quiera escribir.

 # Esto sería un comentario

4.3 Propiedades de la escena

Las propiedades o ajustes de la escena sirven para definir los siguientes parámetros:

- Color de fondo (*BackgroundColor*). El color que se utilizará cuando un rayo no intersecta con ningún objeto. Viene definido por tres valores reales que representan el color en formato RGB.

- Ancho y Alto (*RenderWidth*, *RenderHeight*). Tamaño de la imagen resultante, compuesto por dos parámetros definidos con un valor de tipo entero cada uno.

- Calidad de reflejos (*ReflectionQuality*). Afecta al número de rayos que se trazan para calcular reflejos con ángulo sólido. El valor de este parámetro será entero y el número de rayos será igual a 2*valor^2.

- Pasos de reflejos (*ReflectionSteps*). Indica el número de rebotes máximo a la hora de calcular los reflejos, valor entero.

- Calidad de iluminación indirecta (*IndirectQuality*). Afecta al número de rayos que se trazan para calcular la iluminación indirecta. El valor de este parámetro será entero y el número de rayos será igual a 2*valor^2.

- Sangrado de color (*ColorBleeding*). Es un factor, valor real, que se multiplica por la cantidad de iluminación indirecta que se recibe.

- Suavizado (*Antialiasing*). Es un valor de tipo entero que servirá para indicar la calidad de antialiasing, el número de muestras, para el renderizado de la escena. Viene dado por un valor de tipo entero y debe tener los valores 1, 4 o 16, cualquier valor que no sea uno de estos se descartará quedando el valor por defecto.

A continuación se muestra la estructura de este nodo en el fichero, así como sus valores por defecto.

```
SceneSettings
{
        BackgroundColor     0.0   0.0   0.0
        RenderWidth         512
        RenderHeight        512
        ReflectionQuality   3
        ReflectionSteps     1
        IndirectQuality     3
        ColorBleeding       1
        Antialiasing        1
}
```

4.4 Cámara

La escena dispondrá de una cámara, que definirá el punto de vista del observador y el punto de interés en la escena, así como el campo de visión.

La cámara servirá para elegir qué queremos ver de la escena y desde donde. Se define una cámara mediante los siguientes parámetros.

- Punto de vista (*POV - Point Of View*). Posición de la cámara en el espacio. Se compone de tres valores reales correspondientes a los valores de las coordenadas x, y, z de la posición.

- Punto de interés (*POI - Point Of Interest*). Punto en el espacio donde apunta la cámara. Se compone de tres valores reales que corresponden a los valores de las coordenadas x, y, z del punto.

- Campo de vision (*FOV - Field Of View*). Ángulo que forma la cámara respecto al plano de proyección. Es el equivalente al angular de una cámara real. Este parámetro viene dado por un único valor real, el ángulo en grados sexagesimales.

Solo habrá una cámara en la escena, así que si se escriben varias cámaras en el fichero escena se utilizará la última, es decir, reescribir un nodo cámara implica sobrescribir el anterior.

La estructura de la cámara en el fichero junto con sus valores por defecto se muestra a continuación.

```
Camera
{
        POV     0       1       1
        POI     0       0       0
        FOV     45
}
```

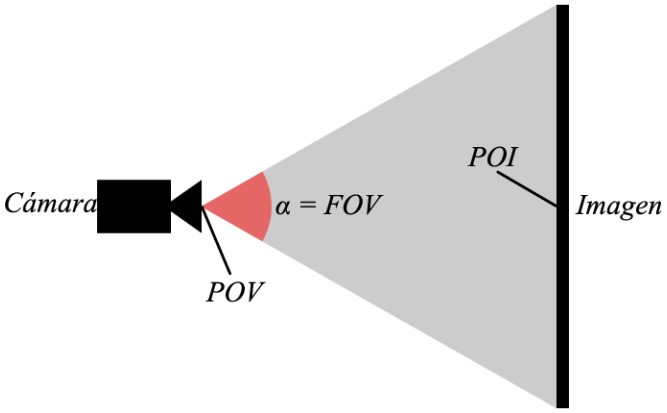


Figura 14. Parámetros de la cámara.

4.5 Texturas

Las texturas se pueden utilizar en los materiales para representar el color difuso, el especular y/o el emisivo. También se podrán utilizar texturas como mapas de normales para alterar las normales del objeto y simular relieves.

Para cargar texturas en la escena se necesitarán dos parámetros.

- Nombre (*Name*). Servirá para identificar a la textura dentro de la escena y no podrá repetirse. Su valor será una cadena de caracteres sin espacios.

- Origen (*Source*). Corresponde a la ruta del archivo imagen que se cargará y viene dado por una cadena de caracteres delimitada por dobles comillas.

Este elemento se escribe en el fichero de la siguiente forma.

```
Texture
{
        Name        nombreSinEspacios
        Source      "directorio/archivo.formato"
}
```

En este caso no hay valores por defecto. Al leer la escena se comprobará que el nombre no existe y que la imagen se lee y se guarda correctamente.

4.6 Materiales

Para darle aspecto a los objetos se definirán materiales en la escena y tendrán varias propiedades.

- Nombre (*Name*). Al igual que en las texturas, el nombre es único y sirve para identificar al material dentro de la escena. Vendrá dado por una cadena de caracteres sin espacios.

- Color difuso (*DiffuseColor*). Es el color propio del objeto. Este parámetro se define mediante tres valores reales que representan los canales RGB del color.

- Textura difusa (*DiffuseTexture*). Vendrá dada por una cadena de caracteres sin espacios que corresponderá al identificador de la textura. Si se utiliza esta textura, la componente difusa del material será esta textura multiplicada por el color difuso (anterior parámetro).

- Color especular (*SpecularColor*). Es el color que representa la especularidad, es decir, la medida en que el material es capaz de reflejar luz. Este parámetro se define mediante tres valores reales correspondientes a los canales RGB del color.

- Textura especular (*SpecularTexture*). Vendrá dada por una cadena de caracteres sin espacios que corresponderá al identificador de la textura. Si se utiliza esta textura, la componente especular del material será esta textura multiplicada por el color especular (anterior parámetro).

- Color emisivo (*EmissiveColor*). Es el color que emite un objeto sin necesidad de ser iluminado. Este parámetro viene dado por tres valores reales que representan los canales RGB del color.

- Textura emisiva (*EmissiveTexture*). Representa lo mismo que el parámetro anterior, solo que en forma de textura. Vendrá dada por una cadena de caracteres sin espacios que corresponderá al identificador de la textura. Si se utiliza esta textura, la componente emisiva del material será esta textura multiplicada por el color emisivo (anterior parámetro).

- Textura de normales (*NormalTexture*). Esta textura se utiliza en el proceso de normal mapping. Este proceso consiste en, a partir de un mapa de normales, alterar las normales del objeto para simular relieves. Vendrá dada por una cadena de caracteres sin espacios que corresponderá al identificador de la textura.

- Rugosidad (*Roughness*). El valor de este parámetro será real e indicará lo pulida (valor cercano a 0) o lo rugosa (valor cercano a 1) que es la superficie del material.

- Opacidad (*Opacity*). Indica si un objeto es transparente y en cuanta medida lo es. Viene dado por un valor real, donde 0 es totalmente transparente y 1 es totalmente opaco.

- Índice de refracción (*RefractionIndex*). Este índice indicará la cantidad de desviación de la luz al pasar de un medio externo al medio del material. Este parámetro se especifica mediante un valor real.

- Ángulo de reflexión (*ReflectionAngle*). Mediante un valor real se indica el ángulo sólido en grados sexagesimales a la hora de calcular los reflejos.

- Cantidad de reflejos (*ReflectionAmount*). Es un factor, entre 0 y 1, que se multiplica por la componente especular para obtener la cantidad de reflejos de un material.

Este elemento se escribe de la siguiente manera en el fichero.

```
Material
{
    Name            nombreDelMaterial
    DiffuseColor    0.9    0.9    0.9
    SpecularColor   0.8    0.8    0.8
    EmissiveColor   0.0    0.0    0.0
    DiffuseTexture  texturaDifusa
    SpecularTexture texturaEspecular
    EmissiveTexture texturaEmisiva
    NormalTexture   texturaDeNormales
    Roughness       0.5
    Opacity         1.0
    RefractionIndex 1.0
    ReflecionAngle  0.0
}
```

Aunque se indiquen nombres de textura, para estos no hay valores por defecto. Adicionalmente, las texturas utilizadas en los materiales deben de haber sido declaradas previamente en el fichero, de lo contrario la escena fallará al cargar.

4.7 Primitivas

Se pueden añadir a la escena una serie de primitivas gráficas.

Las primitivas no están mapeadas, es decir, dado un punto no se saben las coordenadas de textura en este, de manera que los materiales asignados no harán uso de texturas, únicamente lo harán de los colores.

4.7.1 Esfera

Se define una esfera mediante dos parámetros.

- Material (*Material*). Cadena de caracteres sin espacios que corresponderá a un identificador de material previamente declarado.

- Centro (*Center*). El centro de la esfera es un punto en el espacio representado mediante tres valores reales, que corresponden a las coordenadas x, y, z de éste.

- Radio (*Radius*). Será un valor real que indicará el radio de la esfera.

Su declaración dentro del fichero escena y sus valores por defecto se muestran a continuación.

```
Sphere
{
    Material    nombreDelMaterial
    Center      0.0    0.0    0.0
    Radius      1.0
}
```

4.7.2 Cubo

El cubo estará formado por 8 vértices y 6 caras. Cada cara tendrá asociados 4 vértices.

Se definirán cubos mediante transformaciones. Inicialmente se crea un cubo con centro en el origen de coordenadas y con unos valores de anchura, altura y profundidad iguales a 1. Posteriormente mediante translaciones, rotaciones y escalados se podrá modificar el cubo.

Los parámetros para definir el cubo son los siguientes.

- Material (*Material*). Cadena de caracteres sin espacios que corresponderá a un identificador de material previamente declarado.

- Translación (*Translation*). Sirve para aplicar un desplazamiento a los vértices y viene dado por 3 valores reales que corresponden al desplazamiento en cada uno de los ejes.

- Rotación (*Rotation*). Sirve para aplicar una rotación a los vértices. La rotación se realiza respecto al origen de coordenadas. Este parámetro se define mediante 4 valores reales, los 3 primeros corresponderán a las tres componentes del eje de rotación, un vector, y el último valor será el ángulo de rotación en radianes.

- Escalado (Scale). Sirve para aplicar un escalado, respecto al origen de coordenadas, a los vértices. Se define mediante 3 valores reales que serán las escalas en cada uno de los ejes (X, Y, Z).

Dentro del fichero su declaración será como se muestra a continuación. No hay valores por defecto.

```
Box
{
    Material        nombreDelMaterial
    Translation     0.0     0.0     0.0
    Rotation        0.0     1.0     0.0     3.14
    Scale           1.0     1.0     1.0
    …
}
```

Las transformaciones se van aplicando en el orden en el que están escritas y no hay límite en el número de transformaciones que se pueden aplicar.

4.7.3 Cilindro

Se define un cilindro mediante los siguientes parámetros.

- Material (*Material*). Cadena de caracteres sin espacios que corresponderá a un identificador de material previamente declarado.

- Base (*Base*). Un punto en el espacio correspondiente al centro de la base del cilindro. El punto viene dado por 3 valores reales, coordenadas X, Y, Z del punto.

- Eje (*Axis*). Sirve para orientar el cilindro. Este parámetro corresponde a un vector de 3 componentes que viene especificado por 3 valores reales.

- Altura (*Height*). Viene dada por un valor real que indica la altura del cilindro.

- Radio (*Radius*). Viene dado por un valor real que indica el radio del cilindro.

La siguiente figura ilustra los diferentes parámetros.

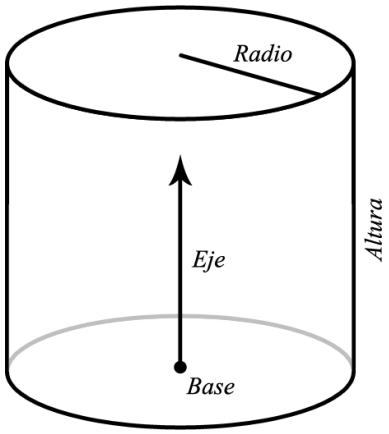


Figura 15. Parámetros del cilindro.

A continuación se muestra la definición de un cilindro en el fichero, junto con sus valores por defecto.

```
Cylinder
{
    Material    nombreDelMaterial
    Base        0.0     0.0     0.0
    Axis        0.0     1.0     0.0
    Height      1.0
    Radius      1.0
}
```

4.8 Árboles CSG

La geometría solido-constructiva (*Constructive Solid Geometry, CSG*) utiliza operadores binarios (unión, diferencia e intersección) sobre primitivas gráficas para obtener distintos objetos.

La composición de operaciones y primitivas se representa en forma de árbol.

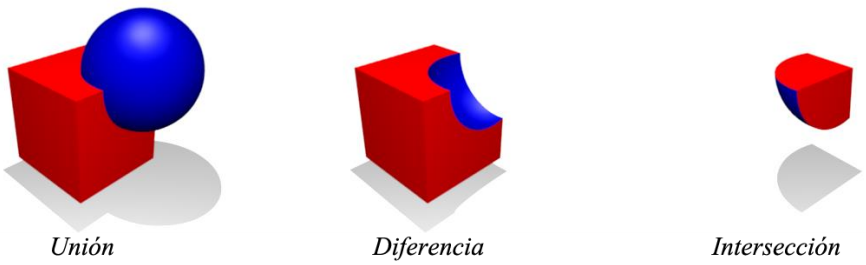


Figura 16. Ejemplos de las operaciones binarias para un cubo y una esfera (Fuente: Wikipedia).

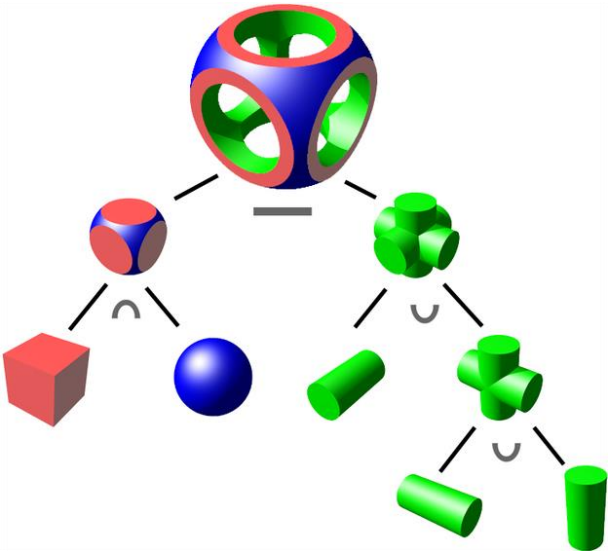


Figura 17. Ejemplo de árbol CSG (Fuente: Wikipedia).

En este proyecto los árboles CSG se importarán desde un fichero de formato .CSG. Este formato de fichero es de un programa propio que permite crear y editar árboles CSG (ver Anexo C).

Para definir un árbol CSG se necesitan los siguientes parámetros.

- Material (*Material*). Cadena de caracteres sin espacios que corresponderá a un identificador de material previamente declarado.

- Fichero origen (*Source*). Viene dado por una cadena de caracteres delimitada por dobles comillas y especifica la ruta del fichero que contiene el árbol CSG. Con la ruta al fichero se procede a importar el contenido.

Su declaración en el fichero escena se muestra a continuación.

```
CSG
{
    Material    nombreDelMaterial
    Source      "directorio/fichero.csg"
}
```

4.9 Mallas poligonales

Se permitirá el importado de mallas poligonales formadas únicamente por triángulos. El software leerá ficheros de tipo .OBJ y extraerá las posiciones de los vértices, las coordenadas de textura, las normales y los polígonos.

Para poder realizar ajustes sobre la malla de polígonos importada existe la posibilidad de aplicar transformaciones a los vértices que forman la malla. Estas transformaciones serán translaciones, rotaciones y escalados.

Los parámetros necesarios para definir una malla poligonal son los siguientes.

- Material (*Material*). Cadena de caracteres sin espacios que corresponderá a un identificador de material previamente declarado.

- Fichero origen (*Source*). Viene dado por una cadena de caracteres delimitada por dobles comillas y especifica la ruta del fichero OBJ. Con la ruta al fichero se procede a importar el contenido.

- Translación (*Translation*). Sirve para aplicar un desplazamiento a los vértices y viene dado por 3 valores reales que corresponden al desplazamiento en cada uno de los ejes.

- Rotación (*Rotation*). Sirve para aplicar una rotación a los vértices. La rotación se realiza respecto al origen de coordenadas. Este parámetro se define mediante 4 valores reales, los 3 primeros corresponderán a las tres componentes de un vector, eje de rotación, y el último valor será el ángulo de rotación en radianes.

- Escalado (*Scale*). Sirve para aplicar un escalado, respecto al origen de coordenadas, a los vértices. Se define mediante 3 valores reales que serán las escalas en cada uno de los ejes (X, Y, Z).

Las transformaciones se deben especificar después del fichero origen, de manera que primero se carga la malla de polígonos y posteriormente se transforma. Las transformaciones se aplicarán en el mismo orden en el que se han escrito y se podrán utilizar tantas como se quieran.

Se especifica una malla poligonal en el fichero de la siguiente manera. No hay valores por defecto.

```
Mesh
{
    Material        nombreDelMaterial
    Source          "directorio/fichero.obj"
    Translation     0.0     0.0     0.0
    Rotation        0.0     1.0     0.0     3.14
    Scale           1.0     1.0     1.0
}
```

4.10 Luces

Las luces se utilizan para iluminar a los objetos de la escena. Al mismo tiempo estas luces generan sombras.

4.10.1 Luces puntuales

Estas luces iluminan desde un único punto del espacio. Como se puede ver en la siguiente figura este tipo de luces generan sombras sin penumbra, esto implica que el borde de las sombras no sea suave.

Las luces puntuales se definen mediante dos parámetros.

- Posición (*Position*). Corresponde a la posición de la luz en la escena. Este parámetro viene dado por 3 valores reales, coordenadas x, y, z del punto.

- Color (*Color*). Es el color de la luz y al mismo tiempo la intensidad de la misma. Este parámetro se define mediante 3 valores reales correspondientes a los canales RGB del color.

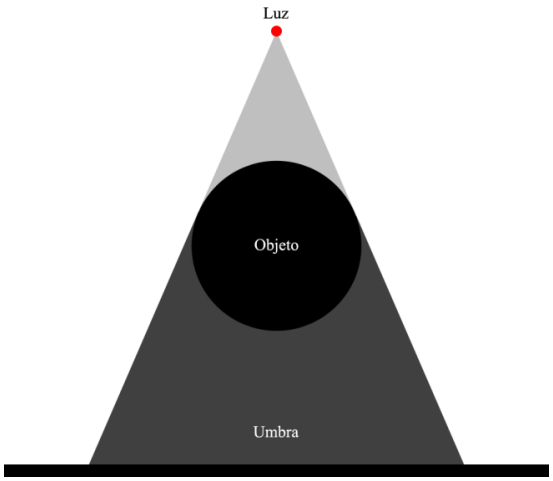


Figura 18. Luz puntual.

Su declaración en el fichero junto con sus valores por defecto es la siguiente.

```
PointLight
{
    Position    0.0    0.0    0.0
    Color       1.0    1.0    1.0
}
```

4.10.2 Luces de área

Estas luces iluminan los objetos de la escena desde muchos puntos de una superficie. Debido a que en una superficie hay infinitos puntos, se ha decidido discretizar la superficie y pasar a tener un conjunto de puntos que representan dicha superficie. Esto implica que a un mayor número de puntos mayor calidad tendrán los resultados.

Solo se han implementado luces con forma de plano rectangular.

El hecho de que tengamos una superficie que ilumina a la escena permite simular sombras con umbra y penumbra, lo que dota de un mayor realismo a las sombras provocadas por este tipo de luces. La siguiente figura ilustra cómo estas luces generan las sombras.

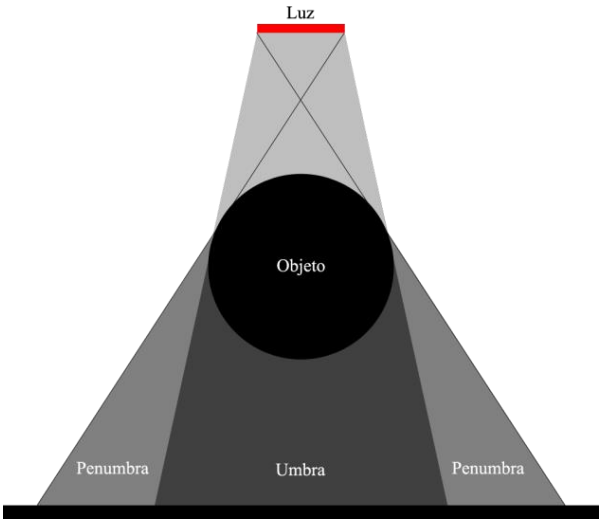


Figura 19. Luces de área generando umbra y penumbra.

Para definir una luz de área con forma de plano se especificarán los siguientes parámetros.

- Color (*Color*). Corresponde al color de la luz y a su intensidad. Se define mediante 3 valores reales, los canales RGB del color.

- Posición (*Position*). Es el punto central del plano en el espacio y viene dado por 3 valores reales correspondientes a las coordenadas x, y, z del punto.

- Dirección (*Direction*). Corresponde a la normal del plano y sirve para orientarlo. Este parámetro se define mediante 3 valores reales que representan a un vector de 3 componentes.

- Ángulo de giro (*RollAngle*). El plano se puede girar utilizando su dirección como eje. Este parámetro especifica el ángulo de giro en radianes mediante un valor real.

- Anchura (*Width*). Tamaño horizontal del plano definido mediante un valor real.

- Altura (*Height*). Tamaño vertical del plano definido mediante un valor real.

- Subdivisión (*Subdivision*). Este parámetro se especifica mediante dos valores enteros que indicarán la cantidad de puntos, a lo ancho y a lo alto, con que se discretiza la superficie.

La siguiente figura ilustra el significado de los parámetros anteriormente definidos.

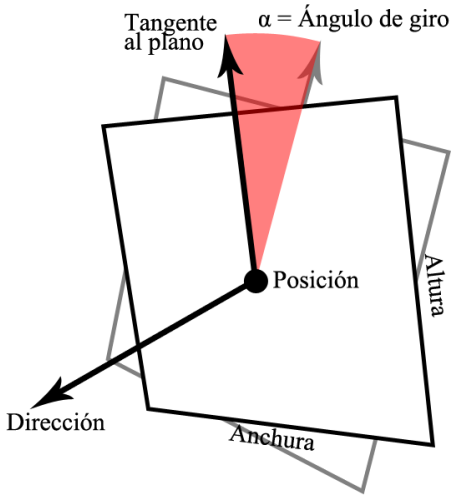


Figura 20. Luz de área con forma de plano.

Estas luces se declaran en el fichero como se describe a continuación junto con sus valores por defecto.

```
AreaLight
{
    Color       1.0     1.0     1.0
    Position    0.0     0.0     0.0
    Direction   0.0     -1.0    0.0
    RollAngle   0.0
    Width       1.0
    Height      1.0
    Subdivision 5       5
}
```

5. Implementación

5.1 Introducción

En este punto se van a describir de forma más detallada los módulos implementados.

5.2 Recursos matemáticos

Para poder realizar los numerosos cálculos necesarios en el proceso de trazado de rayos se necesita disponer de tipos de datos y funciones matemáticas. Todo esto se encuentra en el fichero *Algebra.cuh*.

5.2.1 Datos

Los tipos de datos implementados son los siguientes

- Datos con múltiples valores enteros. Sirven para representar tuplas de valores enteros. En concreto se han implementado tuplas de enteros sin signo con 2 y 3 componentes (tipos u_int2 y u_int3).

- Datos con múltiples valores reales. Sirven para representar tuplas de valores reales, generalmente vectores. Las tuplas que se han implementado son de 2,3 y 4 valores (tipos vec2, vec3 y vec4).

- Matrices. Están formadas por varias tuplas, dependiendo del tamaño de la matriz. Se han implementado dos tamaños de matriz (tipos mat3 y mat4). El primer tamaño de matriz es 3x3 y se compone de 3 tuplas de 3 valores. El otro tamaño de matriz es 4x4 y está formado por 4 tuplas de 4 valores.

Se ha creado otro tipo de dato de tipo estructura, face3, que se utiliza en el proceso de importar mallas poligonales. Este tipo de dato representa los índices de cada vértice en el vector de posiciones, de normales y de coordenadas de textura.

Las librerías de CUDA también proporcionan tipos de datos similares a estos implementados pero los constructores son incómodos de utilizar y no hay funciones implementadas. Igualmente se hace uso de los tipos de datos int2 e int4 en sitios concretos del programa.

5.2.2 Funciones

Las funciones matemáticas que operan con los tipos definidos se han definido en el mismo fichero. No existen todas las funciones y operadores para todos los tipos, esto es debido a que las funciones se han ido implementando conforme ha surgido la necesidad de utilizarlos.

Todos los tipos de datos poseen una serie de constructores con diferentes argumentos. Esto resulta útil, por ejemplo, cuando se quiere crear un vector de 3 componentes y el valor de todas ellas es el mismo, en este caso habrá un constructor con un solo argumento que asignará el mismo valor a las 3 componentes.

También se pueden construir los datos a partir de otros tipos, por ejemplo, la creación de un vector de 3 componentes a partir de un vector de 2 componentes y un valor real.

Las tuplas de valores reales disponen de operadores aritméticos sobrecargados para suma, resta, multiplicación y división. También disponen de operadores comparativos sobrecargados: igual, distinto, mayor, menor, etc. Para las tuplas que se utilicen como vectores se han implementado funciones como el producto escalar de vectores o el producto vectorial de vectores entre otras.

Para las matrices se han implementado las funciones de resta de matrices, multiplicación de matrices y trasponer matrices.

Dentro de este módulo se han añadido dos funciones adicionales:

- el cálculo del vector reflejado R utilizado en el trazado de reflejos. Recibe la dirección del rayo, L, y la normal en la superficie, N, y devuelve el vector reflejado.

- el cálculo de un vector T de transmisión para el cálculo de refracciones. Se le proporcionan el índice de refracción de los dos medios, la dirección del rayo y la normal en la superficie y devuelve el vector refractado.

5.3 Sombreado

Para calcular el sombreado se han implementado una serie de funciones dentro del fichero *Shading.cuh*.

```
vec3 getOrenNayar( vec3 vLight, vec3 vEye, vec3 vNormal,
                   vec3 lightIntensity, vec3 albedo, float roughness);
```

Esta función permite obtener la componente difusa mediante el modelo de sombreado Oren-Nayar. La implementación se ha basado en el código HLSL de este modelo dentro del libro ShaderX2 [7](pag. 125).

```
vec3 getCookTorrance( vec3 vLight, vec3 vEye, vec3 vNormal,
                      vec3 lightIntensity, vec3 albedo, float roughness);
```

Esta función permite obtener la componente especular mediante el modelo de sombreado de Cook-Torrance. La implementación se ha basado en el código HLSL de este modelo dentro del libro ShaderX2 [7] (pag. 134).

```
void getOrenNayarCookTorrance(vec3 vLight, vec3 vEye, vec3 vNormal,
                              vec3 lightIntensity, vec3 albedo, vec3 spec,
                              float roughness, vec3 &diffuse, vec3 &specular);
```

Esta función sirve para obtener las componentes difusa y especular mediante los dos modelos anteriores (pero utilizando solo una función).

5.4 Elementos de la escena y clases

En esta sección se describen los elementos de la escena desde un punto de vista técnico, con detalles específicos de su implementación junto con una breve descripción de la clase que representa al elemento.

5.4.1 Aspectos generales

La definición de las clases y funciones y su implementación se ha realizado dentro del mismo fichero.

Ha habido casos en que las estructuras de datos utilizadas en una clase eran incompatibles con CUDA, de manera que se han realizado dos clases, una para CPU y otra para GPU. En los elementos que se explicarán en las secciones posteriores se especificará que hay dos versiones, de lo contrario se asume que solo hay una, compatible tanto para CPU como GPU.

Generalmente la estructura de datos incompatible ha sido la de vectores STL de C++. En la versión GPU esta estructura se sustituye por un puntero a un *array* de elementos y una variable entera que indica el número de elementos.

Al estar implementado el trazador de rayos en CUDA, antes de la definición, o implementación si no se define una cabecera, de cada función en el proyecto se han añadido unos calificadores.

__global__: indica que la función es un kernel.

__host__: indica que la función solo se puede ejecutar en CPU.

__device__: indica que la función solo se puede ejecutar dentro de un kernel, es decir, en GPU.

__host__ __device__: indica que se puede ejecutar tanto en CPU como en GPU.

5.4.2 Intersecciones

Muchos de los cálculos del proceso de trazado de rayos consisten en encontrar intersecciones rayo-objeto. Posteriormente se necesitarán saber cierta información correspondiente al punto de intersección para realizar operaciones diversas. Para este propósito se ha ideado una clase *Intersection* que servirá para devolver información sobre la intersección.

La información contenida en la clase *Intersection* es la siguiente.

- Tipo (*type*). Es el tipo de la intersección, entrante, saliente o sin intersección.
- Tipo del objeto (*objectType*). Indicará el tipo del objeto con el que ha intersectado.
- Índice del objeto (*objectIndex*). Corresponde a la posición del objeto dentro del vector de elementos del mismo tipo, por ejemplo, un objeto de tipo esfera con un índice 1 corresponde a la segunda esfera en la lista de esferas de la escena.
- Material (*material*). El índice del material del objeto en la lista de materiales de la escena.
- Distancia (*distance*). Distancia entre el punto de intersección y el origen del rayo.
- Punto (*point*). El punto de intersección con el objeto.
- Normal (*normal*). Vector normal a la superficie en el punto de intersección.
- Tangente (*tangent*). Vector tangente a la superficie en el punto de intersección.
- UV's (*uv*). Coordenadas de textura en el punto de intersección.

5.4.3 Cámara

La cámara ha sido implementada en la clase *PerspectiveCamera*, que contiene los campos descritos en la sección 4.4.

Esta clase incluye un método que permite obtener la dirección de un rayo primario con origen en la cámara, dada la posición (i, j) de un píxel y el tamaño de la imagen.

```
vec3 visualPixel(float i, float j, int width, int height);
```

5.4.4 Texturas

La implementación de las texturas se ha dividido en dos partes, la versión CPU y la versión GPU.

5.4.4.1 Texturas (CPU)

La versión de la implementación de las texturas en CPU corresponde a la clase *HostTexture*, que incluye campos correspondientes al contenido de la imagen y a información sobre esta.

El propósito de esta clase es cargar las imágenes desde un fichero especificado y guardarla. Para ello se ha hecho uso de la librería FreeImage, con la que se ha leído el fichero de la imagen y se ha guardado su contenido en un buffer.

5.4.4.2 Texturas (GPU)

La implementación en GPU de las texturas se ha definido la clase *DeviceTexture*.

Esta clase contiene una copia, en la memoria de la GPU, de la imagen que se ha cargado en CPU. Su única función en el proceso de trazado de rayos es devolver el color en cierto punto de la imagen a partir de unas coordenadas de textura mediante el método.

```
vec3 at(vec2 uvs);
```

5.4.5 Materiales

La clase *Material* incluye todos los campos mencionados en la sección 4.6. Las texturas se referencian mediante un número entero que indica el índice de la textura en la lista de texturas de la escena. Si no hay textura asociada entonces el índice es -1.

5.4.6 Primitivas

Las primitivas gráficas (esfera, cubo y cilindro) se han implementado a partir de lo que se ha definido en la sección 4.7. Los parámetros necesarios para definir las primitivas corresponden a campos en las clases.

En el punto 4.7.2 se dice que el cubo tiene 8 vértices y 6 caras. En la implementación se han añadido los vectores normales a cada cara y los vectores tangentes a cada cara. La clase cubo también dispone de un método para calcular las normales y las tangentes a partir de los vértices y las caras, que se utiliza cuando se aplican transformaciones.

Todas las clases correspondientes a primitivas tienen un campo en común que indica el material asociado. Este campo se ha implementado mediante una variable entera que referencia al material en una lista de materiales contenidos en la escena.

Finalmente todas estas clases disponen de dos métodos para el trazado de rayos.

```
bool rayIntersection(vec3 origin, vec3 ray, Intersection& intersection);
bool rayIntersectionOut(vec3 origin, vec3 ray, Intersection& intersection);
```

Los dos métodos sirven para comprobar si un rayo, dado un punto de origen (*origin*) y una dirección (*ray*), intersecta con la primitiva. El primer método devuelve la intersección entrante y el segundo la saliente.

5.4.7 CSG

5.4.7.1 Primitivas

Las implementaciones de las primitivas CSG son iguales que las primitivas normales exceptuando que sólo contienen un método para calcular su intersección, en vez de dos.

```
bool rayIntersection(vec3 origin,vec3 ray, Intersection &in, Intersection &out);
```

Este método devuelve la intersección entrante y la saliente.

5.4.7.2 Operaciones

Las operaciones se han implementado mediante una única clase, *CSG_Node*, que contiene un campo que indica qué operación es.

Al tratarse de un nodo operación éste debe referenciar a su hijo izquierdo y a su derecho. Para esto se tienen unos campos que especifican el tipo del nodo izquierdo y del derecho y una serie de campos de tipo puntero que referencian a los objetos que correspondan al nodo izquierdo o derecho.

Al igual que las primitivas, esta clase dispone de un campo, *material*, que indica el material que se utiliza para representar el objeto resultante.

También se incluyen dos métodos para calcular intersecciones que se explican con más detalle en la sección 5.9.1.6.

5.4.7.3 Árbol

Esta clase, *HostCSGTree*, se utiliza únicamente en la CPU y su propósito es guardar los datos, primitivas y operaciones, que se extraen de un fichero CSG.

Los datos que se guardan en esta clase son listas de primitivas (una por cada tipo de primitiva) y una lista de nodos operación.

Dispone de un único que método que muestra por consola información sobre el contenido (número de nodos de cada tipo).

El árbol en la escena GPU viene representado mediante el nodo raíz, a partir de él se podrán obtener los cálculos que se deseen.

5.4.8 Mallas poligonales

La implementación de las mallas poligonales se ha realizado en dos versiones.

5.4.8.1 Mallas poligonales (CPU)

La versión CPU de las mallas poligonales, clase *HostMesh*, sirve para cargar mallas desde ficheros de tipo OBJ y aplicarles transformaciones.

En esta parte se ha realizado una optimización para el posterior cálculo de intersecciones con la malla debido a que recorrer todos los polígonos de una malla calculando intersecciones es muy costoso.

La optimización ha consistido en la partición espacial del espacio que ocupa la malla mediante cubos. Estos cubos, clase *HostBoundingBox*, parten de la implementación de la primitiva del cubo a la que se le ha añadido una lista de vértices asociados. Estos cubos poseen también una lista de cubos "hijo", resultado del proceso de división.

El proceso que se sigue para la división espacial es el siguiente.

1. Se parte de un cubo con una referencia a la lista de polígonos de la malla.
2. Se cuenta el número de polígonos que están dentro del cubo.
3. Si el número es 0 se marca como "vacio".
4. Si el número es menor que un umbral establecido.
    a. Se añaden los vértices que están dentro del cubo a la lista de vértices asociados.
    b. El nodo se marca como "hoja".
5. Si el número es mayor que el umbral
    a. El cubo actual se divide en 8 cubos más pequeños
    b. Se construyen estos nuevos cubos desde el paso 1
    c. Para cada cubo construido se comprueba si está vacío. Si lo está no se añade como hijo. En caso contrario se añade a la lista de hijos.

Como se puede apreciar, el proceso de construcción de estos cubos es recursivo y el resultado es un árbol donde solo en los nodos marcados como "hoja" se calcularan intersecciones con polígonos.

Para establecer el valor del umbral se han realizado unas pruebas sobre una escena de prueba. El resultado de estas pruebas se muestra en la siguiente tabla.

Umbral	Cubos generados	Memoria ocupada (Bytes)	Tiempo (segundos)
100	2791	212116	8.7404
200	765	58140	8.7492
300	416	31616	8.7527
400	337	25612	8.7699
500	261	19836	8.7795

Tabla 1. Resultados de la prueba de valores de umbral.

Las gráficas extraídas de estos muestran que el coste espacial crece en mayor cantidad frente al coste temporal que decrece muy levemente.

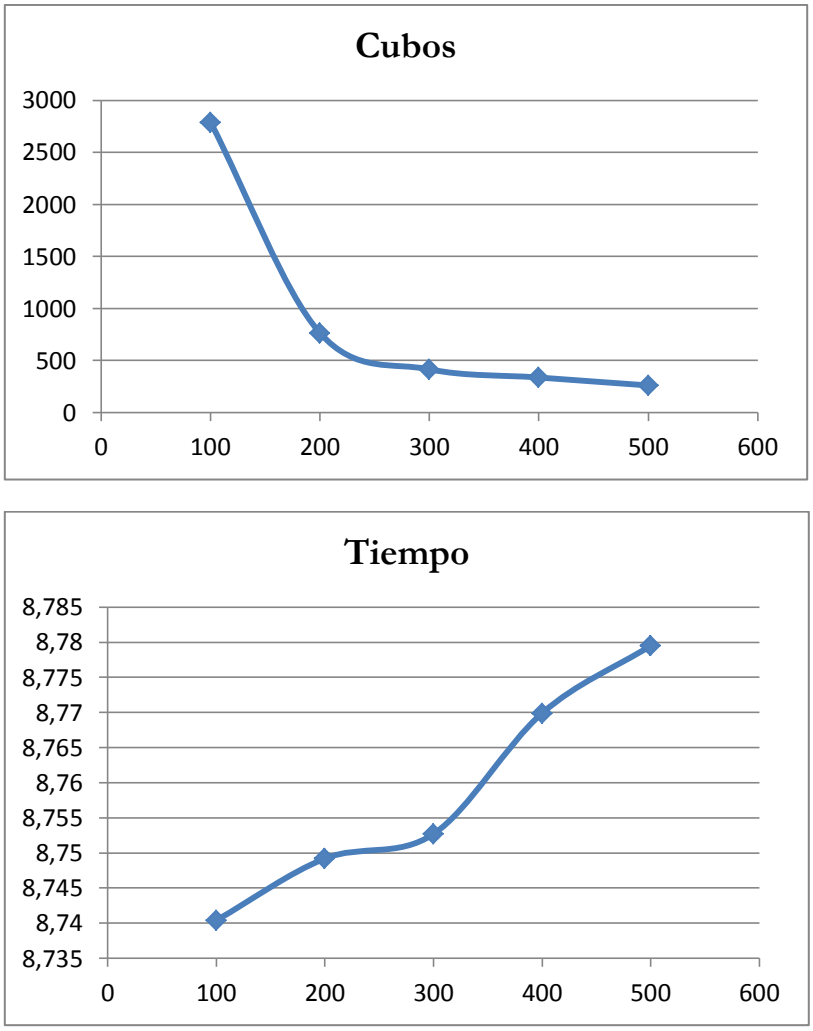


Figura 21. Gráficas de los resultados obtenidos en la prueba del umbral.

Después de evaluar los resultados se ha decidido utilizar un umbral de 300 polígonos.

5.4.8.2 Mallas poligonales (GPU)

La versión para GPU de la implementación de las mallas poligonales, clase *DeviceMesh*, contiene los elementos importados del modelo (posiciones, coordenadas de textura, normales, etc.), una referencia al cubo raíz y un método para obtener la intersección más cercana.

El cubo que se referencia en esta clase es la versión GPU, *DeviceBoundingBox*, y contiene los métodos para calcular intersecciones con los cubos y con los polígonos asociados.

En la implementación GPU los cubos que dividen el espacio siguen formando un árbol pero no de la misma manera. En esta versión todos los cubos se encuentran en una lista de cubos y cada cubo referencia a sus hijos mediante una lista de índices que corresponden a posiciones en esta lista.

Los detalles sobre la conversión de una malla CPU a GPU junto con el cálculo de intersecciones se detallan posteriormente en la sección 5.6.4.

5.4.9 Luces puntuales

Las luces puntuales se han implementado en la clase *PointLight*, que contiene los campos que se han definido previamente la sección 4.10.1.

5.4.10 Luces de área

La implementación de las luces de área se ha dividido en dos versiones.

5.4.10.1 Luces de área (CPU)

Esta versión se implementa en la clase *HostAreaLight* y su propósito es el construir la lista de puntos que discretizan la superficie. Para ello dispone de un método que construye una lista de puntos, dados los argumentos de entrada descritos en la sección 4.10.2.

5.4.10.2 Luces de área (GPU)

La versión GPU de la luz de área corresponde a la clase *DeviceAreaLight*. Dicha clase contiene los mismos datos que la versión CPU pero cuenta con algunos elementos más.

La clase cuenta con una referencia la clase plano (*Plane*, clase que define un plano definido por 4 puntos en el espacio). Esto es porque el plano que representa la luz será dibujado como un objeto más de la escena.

Como es un objeto que se puede dibujar, implica que se tendrán que calcular intersecciones sobre el plano. Para ello se dispone de un método que calcula la intersección de un rayo con el plano.

```
bool rayIntersect(vec3 origin, vec3 ray, Intersection &intersection);
```

5.4.11 Propiedades de la escena

Las propiedades de la escena se han implementado mediante una estructura con los mismos campos que los parámetros definidos en la sección 4.3.

5.4.12 Escena

La implementación de la escena se ha realizado en dos versiones.

5.4.12.1 Escena (CPU)

La clase *HostScene* contiene todos los elementos que forman la escena. En el caso de los elementos que tienen 2 versiones, en esta clase solo se almacenan las versiones CPU.

5.4.12.2 Escena (GPU)

La clase DeviceScene contiene todos los elementos de la escena y una gran cantidad de métodos para realizar diferentes cálculos entre los que destacan.

- Obtener la intersección más cercana para un rayo determinado.
- Obtener la iluminación en un punto de la escena.
- Obtener el color (difuso, especular o emisivo) en un punto de la escena.

5.5 Función principal

El fichero kernel.cu contiene la función principal del programa que sigue los pasos de la solución propuesta en la sección 3.2.

```
int main(int argc, char **argv)
```

La función principal recibe dos argumentos correspondientes a los argumentos de ejecución del programa (un contador de argumentos y los argumentos en sí). El programa debe recibir un único argumento, la ruta al fichero escena, o dos, la ruta al fichero escena y la ruta a la imagen resultado.

Esta función realiza una serie de tareas en el orden en el que se presentan a continuación de manera general, pues se explican con más detalle individualmente en posteriores secciones.

- Se declaran las variables locales necesarias y se inicializan (en el caso).
- Se comprueban los argumentos de entrada (se deben recibir 1 o 2). En caso de no recibir el número de argumentos adecuado se informa por consola con un texto cómo se ejecuta el programa correctamente. En caso positivo pasamos al siguiente punto.
- Con el primer argumento de entrada se intenta cargar el fichero entrada. Si se produce algún error, ya sea porque la ruta no es correcta o porque el fichero contiene errores, el programa acaba (el motivo del error se habrá mostrado por consola). Si la escena se ha cargado correctamente se sigue la ejecución.
- Se transfiere la escena de la memoria principal de la CPU a la memoria global de la GPU.
- Se inicializan unas estructuras de CUDA que permiten la generación de números aleatorios.
- En esta parte se realizan una serie de tareas antes de lanzar a ejecución el kernel.
    - Se reserva espacio en la GPU para la imagen resultado.
    - Se establecen los parámetros de lanzamiento del kernel según los parámetros de configuración de la escena.
- Se ejecutan los kernels.
- Se recupera la imagen resultado copiando desde la memoria de la GPU a la memoria de la CPU.
- La imagen resultado se guarda en el fichero especificado como segundo argumento. Si no hay segundo argumento se guarda en un fichero por defecto “out.bmp”.
- La imagen resultado se muestra por pantalla.

5.6 Lectura del fichero escena

5.6.1 General

La función encargada de leer y construir la escena tiene la siguiente forma.

```
bool parseSceneFile(char *filename, HostScene *scene, SceneSettings *settings);
```

La lectura de los diferentes elementos se realiza línea a línea. La función leerá una línea, extraerá el primer token, o palabra, y comprobará si se corresponde con algún identificador de elemento o con algún parámetro dentro de un elemento. En caso negativo la función acabará devolviendo FALSO. En caso afirmativo procesará la línea y continuará leyendo el fichero.

Esta función espera leer los elementos, los argumentos y los valores tal y como se han definido en la sección 4, de lo contrario fallará indicando la línea donde se ha producido el error y una pequeña descripción del error.

5.6.2 Mallas poligonales

Las mallas poligonales se han importado desde un fichero de tipo OBJ (ver anexo B). Los diferentes elementos se leen y al final se obtienen 3 listas: de posiciones, de coordenadas de texturas y de normales.

Posteriormente se realiza un procesado de los datos leídos.

- En primer lugar se obtiene una lista de vértices únicos, es decir, una lista de polígonos donde para cada uno el conjunto de posición, coordenada de textura y normal no se repite en ningún otro polígono.
- Se calculan las normales en los polígonos (el fichero OBJ solo contiene normales en los vértices).
- Se calculan las tangentes mediante la aproximación de Lengynel.

5.6.3 CSG

Se ha implementado una función que lee ficheros de tipo CSG, este tipo de ficheros se leen de manera muy similar al fichero escena. Este fichero se especifica en el anexo C. Los nodos leídos y procesados se guardan en una clase *HostCSGTree*.

5.7 Copiado de la escena

5.7.1 Introducción

En esta sección se describe el proceso de copiar los datos correspondientes a la escena desde la memoria de la CPU a la memoria de la GPU. La mayoría de elementos de la escena siguen un proceso similar de copiado, sin embargo para otros elementos el proceso de copiado es más complejo. A continuación se explicarán el método general y los específicos a ciertos elementos.

Se han utilizado dos funciones que proporciona CUDA para realizar el copiado de la escena.

- cudaMalloc. Permite reservar espacio en la memoria global de la GPU. Se le proporcionan un puntero, que posteriormente apuntará a la zona de memoria que se habrá reservado, y el tamaño de lo que se quiere reservar.
- cudaMemcpy. Permite copiar datos. Se le proporcionan una dirección destino, una dirección origen, el tamaño de los datos y el tipo de copia, de CPU a GPU, de GPU a CPU, de GPU a GPU o de CPU a CPU.

El proceso de copiado de la escena se realiza mediante la siguiente función que se encuentra en el fichero *Scene.cuh*.

```
void copySceneHostToDevice(HostScene *hostScene, DeviceScene *deviceScene);
```

El puntero que recibe del tipo *DeviceScene* está en memoria de la CPU, esto se debe a que esta clase representa las listas de elementos mediante un puntero a una lista y un número de elementos. Estos punteros se deben inicializar en la instancia de *DeviceScene* en CPU con punteros en el espacio de GPU para posteriormente copiar la *DeviceScene* a GPU.

En las funciones específicas de copiado de cierto tipo de datos esta *DeviceScene* es la misma, una instancia en CPU.

5.7.2 General

El proceso general de copiado de un elemento es el siguiente.

- Se quieren copiar un número *n* de elementos de un tipo determinado.
- Se reserva espacio para albergar una lista de *n* punteros a ese tipo de elemento.
- Para cada elemento
    - Se reserva espacio para 1 clase de este tipo
    - Se copia el elemento
    - Se copia la referencia a ese objeto en su posición de la lista
- Esta lista de elementos se asigna a su correspondiente lista dentro de una clase *DeviceScene*.

5.7.3 Texturas

La clase *DeviceTexture* tiene un campo, un puntero, que contiene la imagen que se ha cargado. Esto implica que, antes de copiar una clase de este tipo, el puntero previamente ha de apuntar a donde están los datos en la memoria de la GPU. Para ello se tiene la siguiente función.

```
void copyTexturesHostToDevice(HostScene *hostScene, DeviceScene *deviceScene);
```

Esta función sigue los siguientes pasos.

- Se quieren copiar *n* texturas.
- Se reserva espacio para albergar una lista de *n* punteros a textura.
- Para cada textura
    - Se reserva espacio para los datos de la imagen.
    - Se copian los datos de la imagen desde la correspondiente *HostTexture*.
    - Se crea una instancia de *DeviceTexture* en CPU y se asignan los valores del tamaño de la imagen y el puntero con los datos que se acaban de copiar.
    - Se reserva espacio para una *DeviceTexture* en GPU.
    - Se copia la *DeviceTexture* en CPU a GPU.
    - Se copia la referencia a esta textura en su posición en la lista de punteros a textura.
- Esta lista de texturas se asigna a su correspondiente lista dentro de una clase *DeviceScene*.

5.7.4 Mallas poligonales

En la clase *DeviceMesh* se tiene el mismo problema que anteriormente con las texturas, hay unos campos que requieren de punteros ya inicializados a posiciones de memoria en GPU. Adicionalmente se ha de convertir el árbol de cubos que dividen el espacio en una lista enlazada. Todo ello se encuentra en la función:

```
void copyMeshesHostToDevice(HostScene *hostScene, DeviceScene *deviceScene);
```

Para el copiado de mallas se siguen los siguientes pasos.

- Se quieren copiar *n* mallas.
- Se reserva espacio para albergar una lista de *n* punteros a malla.
- Para cada malla
    - Se crea una instancia de *DeviceMesh* en CPU.
    - Se reserva espacio para los polígonos de la malla, las posiciones de los vértices, las coordenadas de textura, las normales de los vértices, las normales de las caras y las tangentes de los vértices.
    - Todos estos elementos se copian desde la correspondiente malla *HostMesh*.
    - Se procesan los cubos que dividen el espacio (ver abajo).
    - Se asignan todas las listas que se han copiado a los campos correspondientes de la *DeviceMesh* en CPU.
    - Se reserva espacio para una DeviceMesh en GPU.
    - Se copia la DeviceMesh de CPU en GPU.
    - Se copia la referencia a esta malla en su posición en la lista de punteros mallas.
- Esta lista de mallas se asigna a su correspondiente lista dentro de una clase *DeviceScene*.

El procesado de los cubos se realiza siguiendo estos pasos.

- En primer lugar se recorren todos los cubos del árbol y se marcan con un índice único. Al mismo tiempo un cubo con nodos hijo se guardará los índices de estos.
- Con los cubos marcados se vuelve a recorrer el árbol. Esta vez cada cubo añadirá sus hijos a una lista de cubos en las posiciones que indiquen sus índices.
- Se reserva espacio para *n* cubos.
- Para cada cubo de la lista
    - Se crea una instancia de *DeviceBoundingBox* en CPU.
    - Se reserva espacio para la lista de vértices asociados y se copian.
    - Se reserva espacio para los índices correspondientes a los hijos y se copian.
    - Se reserva espacio para los polígonos, vértices y normales que definen el propio cubo y se copian.
    - Se define el *DeviceBoundingBox* en CPU con las referencias a todos los elementos de los puntos anteriores.
    - Esta instancia se copia a la lista de cubos en GPU.
- La referencia a la lista de cubos se añade a la *DeviceMesh* en CPU.

La siguiente figura ilustra mediante un árbol más simple el proceso de marcado de los cubos y su paso a forma de lista.

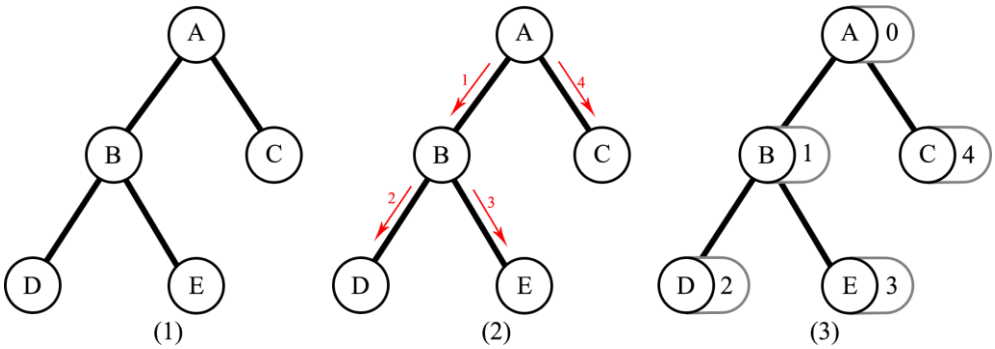


Figura 22. (1) Árbol original. (2) Recorrido. (3) Índices asociados.

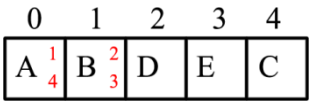


Figura 23. Nodos añadidos a la lista y las referencias a sus hijos (rojo).

5.7.5 CSG

La copia de árboles CSG se realiza con la función:

```
void copyCSGTrees(HostScene *hostScene, DeviceScene *deviceScene);
```

El proceso de copiado de una clase *HostCSGTree* a GPU se realiza de la siguiente manera.

- Se quieren copiar *n* árboles CSG.
- Se reserva espacio para una lista de *n* punteros a la clase *CSG_Node*.
- Para cada árbol CSG
    - Para cada tipo de nodo (esfera, cubo, cilindro y operación) se reserva una lista diferente y se copian los diferentes nodos en ella. (Los nodos operación no disponen de referencias a sus hijos, más allá de índices y tipos, de manera que se deben obtener los punteros y asignarlos).
    - Para cada nodo operación
        - Se comprueba el tipo de hijo izquierdo y se asigna el valor de la referencia mediante un puntero a una de las listas del punto anterior.
        - Se hace lo mismo para el nodo derecho.
    - Se vuelven a copiar los nodos operación, se sobreescriben, ahora con las referencias a sus hijos establecidas.
    - Se copia la referencia al nodo raíz en la lista de punteros.
- Se asigna la lista de punteros a la escena *DeviceScene*.

5.7.6 Luces de área

El copiado de luces de área se realiza en la misma función mencionada en la sección 5.6.1 y sigue el siguiente procedimiento.

- Se quieren copiar *n* luces de área.
- Se reserva espacio para una lista de *n* punteros a *DeviceAreaLight*.
- Para cada *HostAreaLight* en la escena CPU.
    - Se crea una instancia de *DeviceAreaLight* en CPU.
    - Se reserva espacio para la lista de puntos y se copian los puntos de la actual luz.
    - Se reserva espacio para el plano que define la luz y se copia.
    - Se establecen diferentes parámetros de la luz (se asignan valores a los campos de la clase).
    - Se reserva espacio para una DeviceAreaLight en GPU.
    - Se copia la instancia en CPU a GPU.
    - Se copia la referencia a esta DeviceAreaLight en la lista de punteros.
- Se asigna la lista de punteros a la escena *DeviceScene*.

5.8 Kernels de CUDA

Como se ha comentado en la sección 3.4 los kernels se lanzan en bloques, donde cada bloque tiene un número determinado de hilos. En el lanzamiento de kernels se distingue entre el trazado sin antialiasing y el trazado con antialiasing.

CUDA dispone de mecanismos para identificar a los *threads* dentro de una función definida como kernel.

- threadIdx (estructura con 3 campos: x, y, z). Permite obtener la "posición" del hilo dentro del bloque, por ejemplo, si es el hilo(1,2,3) de un bloque de 3 dimensiones.
- blockIdx (estructura con 2 campos: x, y). Permite obtener la "posición" del bloque dentro de la rejilla, por ejemplo, si es el bloque (2,5) de una rejilla de 2 dimensiones.
- blockDim (estructura con 2 campos: x, y). Permite obtener el tamaño del bloque.

5.8.1 Generación de números aleatorios

Diferentes cálculos del trazado de rayos necesitan disponer de números aleatorios. Para poder generar números aleatorios se ha utilizado una estructura de datos que provee CUDA, *curandState*. Cada *curandState* corresponde a un estado diferente dentro de la función matemática que genera los números pseudo-aleatorios. En este proyecto se ha necesitado una estructura de este tipo para cada rayo primario.

La función utilizada para obtener números aleatorios dentro del kernel es la siguiente.

```
__device__ float curand_uniform (curandState *state);
```

Esta función devuelve un número real aleatorio entre 0 y 1, lo que resulta útil para posteriormente ajustar el rango de este valor

5.8.2 Trazado sin antialiasing

En el trazado de rayos sin antialiasing se utilizan bloques de 2 dimensiones y una rejilla de 2 dimensiones. Si se parte de la imagen resultante, los bloques corresponden a zonas de la imagen (de manera que una imagen se compone de X*Y bloques) y los hilos corresponden a píxeles concretos. Como cada píxel se obtiene mediante el resultado de trazar un rayo principal, se establece que un hilo corresponde a un rayo principal.

El trazado de rayos sin antialiasing utiliza solo un kernel.

```
void rayTraceAA1(unsigned char *u_raster,
                 DeviceScene *scene,
                 curandState *states,
                 SceneSettings settings    )
```

Este kernel recibe un puntero al buffer imagen (un vector de enteros de 8 bits sin signo), un puntero a la escena, un puntero a las estructuras de números aleatorios y los parámetros de la escena. El procedimiento que realiza es el siguiente.

- Obtiene, con los mecanismos de la sección anterior, la posición (i, j) del píxel que le corresponde calcular.
- Comprueba que la posición obtenida se encuentra dentro de los límites de la imagen. En caso afirmativo continua, en negativo no hace nada.
    - Obtiene la posición del pixel en el buffer imagen.
    - Inicializa su correspondiente estructura *curandState*.
    - Obtiene la dirección del rayo principal mediante la cámara de la escena.
    - Se realiza el proceso de trazado del rayo.
    - Se obtiene el resultado en forma de 3 valores reales
    - Estos valores se restringen entre 0 y 1.
    - Los valores se convierten al rango entero [0, 255].
    - El resultado se almacena en el buffer.

5.8.3 Trazado con antialasing

En el trazado de rayos con antialiasing se utilizan bloques de 3 dimensiones y una rejilla de 2 dimensiones. Al igual que en el trazado sin antialiasing, los bloques corresponden a zonas de la imagen resultante y los hilos de ejecución a píxeles concretos.

Como se ha visto en la sección 3.3.6 se requieren de varias muestras, en este caso rayos, para obtener el valor de un píxel. La dimensión Z del bloque indica la muestra que se tiene que trazar.

El trazado de rayos con antialiasing requiere del lanzamiento de dos kernels, siendo el primero de ellos explicado a continuación.

```
void rayTraceAA4(float *f_raster,
                 DeviceScene *scene,
                 curandState *states,
                 SceneSettings settings,
                 vec2 *offsets   )
```

Este kernel recibe un puntero al buffer imagen (un vector de números reales), un puntero a la escena, un puntero a las estructuras de números aleatorios, los parámetros de la escena y un puntero a un vector de

desplazamientos. El vector de desplazamientos sirve para desplazar el rayo, centrado en el pixel, a la posición de la muestra.

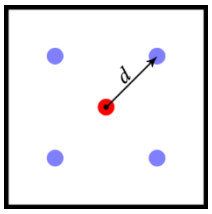


Figura 24. Desplazamiento *d* de la posición central del píxel (rojo) a la posición de la muestra (azul).

El procedimiento que realiza es el mismo que en el kernel sin antialiasing salvo por unos detalles.

- En la obtención de la dirección del rayo se aplica un desplazamiento a la posición correspondiente a la muestra.
- El resultado se acumula en el buffer de números reales mediante la función *atomicAdd*. Esta función proporcionada por CUDA sirve para serializar accesos concurrentes a variables por parte de varios hilos, en este caso serializa las operaciones lectura, incremento y escritura.

El segundo kernel es el siguiente.

```
void pixelMean(  unsigned char *u_raster,
                 float *f_raster,
                 SceneSettings settings,
                 int numSamples      )
```

El kernel recibe un puntero al buffer de números reales con los resultados acumulados, un puntero al buffer imagen resultado, los parámetros de la escena y el número de muestras con que se ha trazado.

La función que cumple este kernel es la de obtener la media en cada píxel y asignar ese valor al buffer de la imagen resultado.

5.9 Trazado

En esta sección se detallan los métodos que se han seguido para los diferentes cálculos relacionados con el trazado de rayos.

5.9.1 Intersecciones con los objetos de la escena

5.9.1.1 Esferas

Para obtener la intersección de un rayo con una esfera se ha utilizado el método descrito en el libro Graphics Gems (pag. 388) "*Intersecion of a Ray with a Sphere*".

Este método permite obtener la distancia y el punto de intersección. Adicionalmente se ha calculado la normal a la superficie como el vector (centro – punto) normalizado y la tangente a la superficie como el producto vectorial de (0, 1, 0) y la normal.

5.9.1.2 Cubos

La intersección de un rayo con un cubo se ha obtenido utilizado el método descrito en el libro Graphics Gems II (pag. 247) "*Fast Ray-Convex Polyhedron Intersection*".

Este método permite obtener la distancia, el punto de intersección y el vector normal a la superficie. El vector tangente se obtiene mediante la tangente de la cara del cubo más cercana.

5.9.1.3 Cilindros

Para obtener la intersección de un rayo con un cilindro se ha utilizado el algoritmo descrito en el libro Graphics Gems IV (pag. 356) "*Intersecting a Ray with a Cylinder*".

Este método permite obtener la distancia y el punto de intersección. El vector normal y el vector tangente se han calculado en función de la superficie de intersección del cilindro.

- Plano superior. La normal corresponde al vector eje y la tangente al producto vectorial del vector (0, 1, 0) y el eje.
- Plano inferior. La normal corresponde al vector eje negado y la tangente al producto vectorial del vector (0, 1, 0) y el eje.
- Lado. La normal se obtiene realizando operaciones trigonométricas (ver figura) y la tangente mediante el producto vectorial de la normal y el eje.

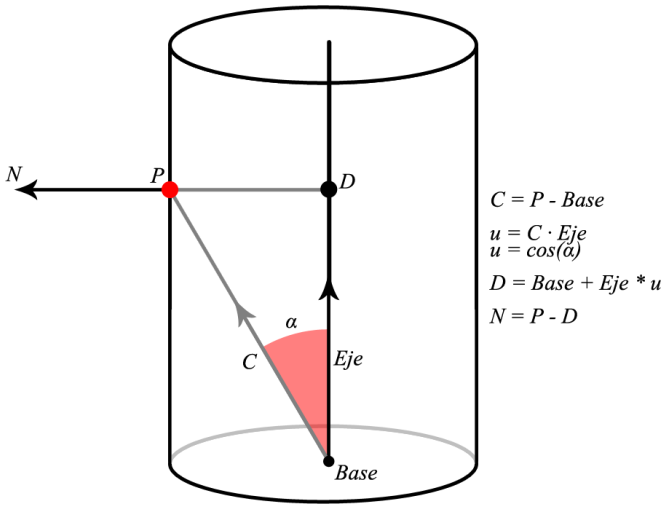


Figura 25. Obtención del vector normal (N) en un punto de la superficie de un cilindro.

5.9.1.4 Mallas poligonales

La intersección de un rayo con una malla poligonal se realiza a través de los cubos que delimitan su espacio. El proceso parte del cubo raíz y sigue el siguiente procedimiento.

- Si no existe intersección con el cubo
    - No se hace nada
- Si existe intersección con el cubo.
    - Se comprueba si es un cubo "hoja", es decir, si tiene o no polígonos asociados.

- Si es un cubo "hoja"
    - Se calculan las intersecciones con todos los polígonos asociados
- Si no es un cubo "hoja"
    - Se ejecuta este proceso para todos sus cubos hijo
    - Se devuelve el resultado

5.9.1.5 Polígonos

Los polígonos en este proyecto siempre son triángulos. La intersección de un rayo con un triángulo se ha implementado siguiendo el método propuesto en el libro Graphics Gems (pag. 390) "*An Efficient Ray-Polygon Intersection*".

Este proceso consta de 2 pasos.

- Encontrar el punto de intersección con el plano que contiene el triángulo.
- Evaluar si el punto se encuentra dentro del triángulo.

Si el punto se encuentra dentro del triángulo se deben de obtener los vectores normal y tangente y las coordenadas de textura en ese punto. Para conseguir esto se han interpolado los valores de estos vectores en los 3 puntos que definen el triángulo respecto al punto de intersección.

Para realizar la interpolación en primer lugar se ha calculado el área del triángulo. El punto P de intersección divide el triángulo en 3 triángulos más pequeños. La suma del área de estos triángulos corresponde al área del triángulo original como ilustra la siguiente figura.

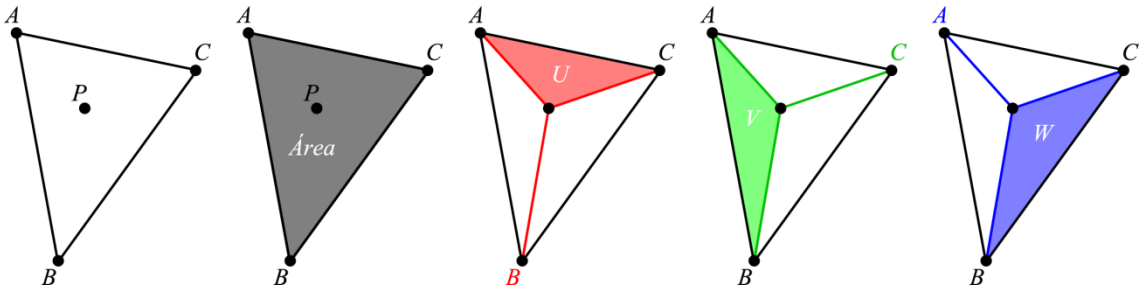


Figura 26. Áreas de un triángulo.

El resultado de la interpolación será la suma de cada uno de los vectores de los puntos del triángulo multiplicados por un factor, el área del correspondiente triángulo pequeño dividido por el área del triángulo. Como se puede ver en la figura el resultado es:

$$Interpolación = \left(A \times \frac{W}{Área}\right) + \left(B \times \frac{U}{Área}\right) + \left(C \times \frac{V}{Área}\right)$$

5.9.1.6 CSG

El cálculo de intersecciones de rayos sobre sobre árboles CSG se realiza calculando una lista con todas las intersecciones de las primitivas y procesando esa lista según las operaciones.

El proceso se inicia desde el nodo raíz. En primer lugar se crea una lista de intersecciones cuyo tamaño es 2 veces el número de primitivas en el árbol, esto es porque se necesitan las intersecciones entrantes (IN) y salientes (OUT) de cada primitiva. También se crean unas referencias a las posiciones inicial y final, que se utilizan para establecer la parte de la lista sobre la que se va a trabajar. Estos primeros pasos se realizan en el método:

```
bool rayIntersection(vec3 origin, vec3 ray, Intersection &intersection);
```

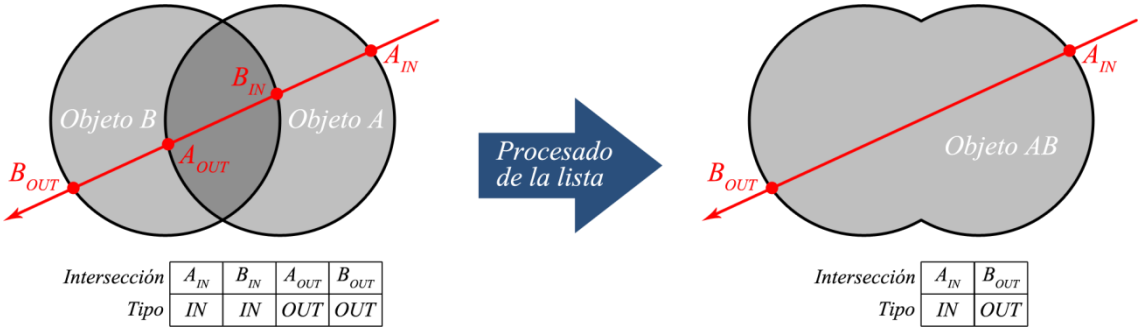
A partir de aquí se ejecuta el siguiente método:

```
bool rayIntersection( vec3 origin, vec3 ray,
                      Intersection *intersections, int2 *arrayIndex);
```

Cada nodo realiza una serie de operaciones en este último método:

- Se crean dos referencias o índices. Estos índices marcan las partes de la lista que ha devuelto el hijo izquierdo y derecho.
- Se calculan las intersecciones del hijo izquierdo y si ha habido intersección se incrementan los índices acorde con el número de intersecciones. Si no habido intersección y el nodo es una diferencia o una intersección entonces no hay intersecciones para este nodo.
- Se calculan las intersecciones del hijo derecho y si ha habido intersección se incrementan los índices acorde con el número de intersecciones. Si además el nodo es una diferencia, en todos los nodos de la lista correspondientes a este hijo se cambia el tipo de las intersecciones (de entrantes a salientes y viceversa) y se invierten sus normales.
- Se realizan unas comprobaciones.
    - Si no hay intersección en el nodo izquierdo ni en el derecho no hay intersección en este nodo.
    - Si hay intersección en el hijo izquierdo pero no en el derecho.
        - Si el nodo es una unión o una diferencia el resultado son las intersecciones del hijo izquierdo.
        - Si no, no hay intersección.
    - Si no hay intersección en el hijo izquierdo pero si en el derecho.
        - Si el nodo es una unión el resultado son las intersecciones del hijo derecho.
        - Si no, no hay intersección.
    - Si hay intersecciones en ambos hijos se continua la ejecución
- La zona de la lista de intersecciones correspondiente a las intersecciones devueltas por ambos hijos se ordena crecientemente en función de la distancia, mediante el algoritmo *ShellSort*.
- Se eliminan de la lista las distancias negativas, de manera que si tras este paso, la lista tiene un número intersecciones impar no habrá intersecciones en este nodo.
- En este punto se realizarán tareas según el tipo de operación.

    - **Unión**
        - Si dentro de la lista hay dos intersecciones consecutivas cuyos tipos son IN e IN se borra la segunda (la más lejana).
        - Si dentro de la lista hay dos intersecciones consecutivas cuyos tipos son OUT y OUT se borra la primera (la más cercana).

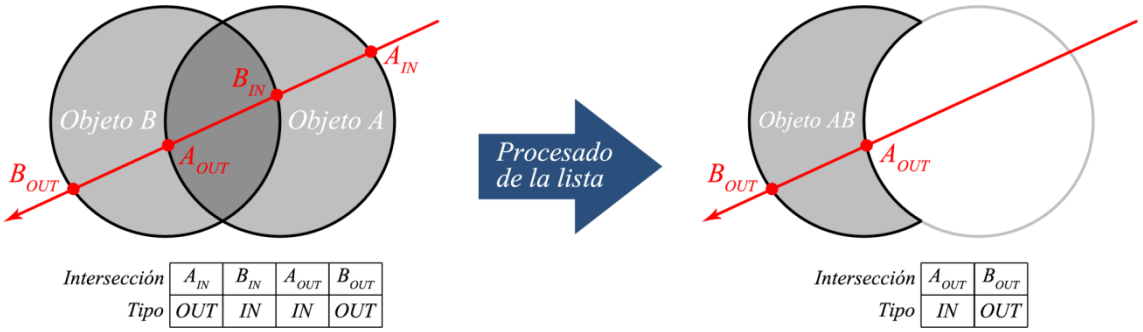


Intersección	A_{IN}	B_{IN}	A_{OUT}	B_{OUT}
Tipo	IN	IN	OUT	OUT

Intersección	A_{IN}	B_{OUT}
Tipo	IN	OUT

Figura 27. Procesado de una lista para el caso Unión(A, B).

- **Diferencia**
  - Si dentro de la lista hay dos intersecciones consecutivas cuyos tipos son OUT e IN se borran los dos.

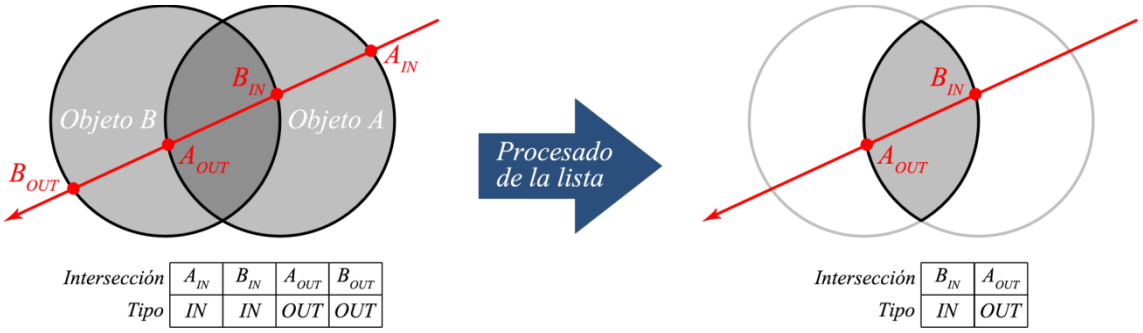


Intersección	A_{IN}	B_{IN}	A_{OUT}	B_{OUT}
Tipo	OUT	IN	IN	OUT

Intersección	A_{OUT}	B_{OUT}
Tipo	IN	OUT

Figura 28. Procesado de una lista para el caso Diferencia (B, A).

- **Intersección**
  - Si dentro de la lista hay dos intersecciones consecutivas cuyos tipos son IN e IN se borra la primera (la más cercana).
  - Si dentro de la lista hay dos intersecciones consecutivas cuyos tipos son OUT y OUT se borra la segunda (la más alejada).



Intersección	A_{IN}	B_{IN}	A_{OUT}	B_{OUT}
Tipo	IN	IN	OUT	OUT

Intersección	B_{IN}	A_{OUT}
Tipo	IN	OUT

Figura 29. Procesado de una lista para el caso Intersección (A, B).

La intersección más cercana de todo el objeto corresponderá a la primera intersección de la lista.

5.9.2 Algoritmo de trazado general

La función principal de trazado de rayos se encuentra implementada en el fichero *RayTracing.cuh*.

```
vec3 rayTrace( vec3 origin, vec3 ray,
               DeviceScene *scene, curandState *state,
               int reflectionQuality, int indirectQuality,
               u_int2 steps, float colorBleeding)
```

Esta función recibe el rayo principal junto con la escena y diferentes propiedades de la escena.

El primer paso que realiza el algoritmo es encontrar la intersección del objeto más cercano. Esto se ha implementado mediante un método en *DeviceScene* (ver sección 5.9.3).

Si no existe la intersección se devuelve el color de fondo. En caso contrario se procede a calcular el color en el punto de intersección. Si el objeto intersectado fuera un plano de luz de área se devuelve el color de la luz.

Se comprueba si el objeto intersectado es una malla y si tiene el material tiene un mapa de normales asignado, en cuyo caso se calcula el nuevo vector normal.

Para obtener el color primero se han de calcular diferentes componentes.

- En primer lugar se calcula la iluminación directa (ver sección 5.8.3).
- Si el material es transparente se calcula la transmisión o refracción (ver sección 5.8.7).
- Si el material no es totalmente transparente (opacidad = 0) y la calidad de iluminación indirecta es > 2 se calcula la iluminación indirecta (ver sección 5.8.5).
- Si el componente especular del material no es negro (0, 0, 0), si el material no es totalmente transparente, si la calidad de reflejos es mayor que 0 y el número de rebotes actual es menor que el máximo entonces se calculan los reflejos (ver sección 5.8.6).

A partir de estas componentes el color final se calcula como:

$$I_{directa} = Iluminación\ directa$$
$$I_{indirecta} = Iluminación\ indirecta$$
$$R = color\ de\ reflejos\ \times especular$$
$$T = color\ de\ transmisión$$
$$Lerp(x, y, a) = x * a + y * (1 - a)$$

$$Color = Lerp(\ I_{directa} + I_{indirecta}\ , R\ , cantidad\ de\ reflejos)$$
$$Color = Lerp(T, Color, opacidad)$$

5.9.3 Recorrido de la escena

La clase *DeviceScene* dispone de dos métodos para obtener la intersección rayo-objeto más cercana. Esto se hace calculando las intersecciones del rayo con cada objeto de la escena y devolviendo, si hay intersección, la más cercana.

```
bool getNearestIntersection(vec3 origin, vec3 ray, Intersection &in);
bool getNearestIntersectionAll(vec3 origin, vec3 ray, Intersection &in);
```

Los dos métodos devuelven TRUE si ha habido intersección y FALSE en caso contrario. Cumplen la misma función pero la diferencia principal es que la primera no tiene en cuenta los planos de las luces de área como objeto. Esto es así porque los planos de las luces de área no se tienen en cuenta para calcular sombras.

5.9.4 Iluminación directa

La iluminación directa en un punto se calcula como la suma de las iluminaciones producidas por todas las luces de la escena. La clase *DeviceScene* tiene una serie de métodos para obtener esto.

```
vec3 getDirectLighting(Intersection &intersection, vec3 eye, curandState *state);
vec3 getPointLightIlluminationAt(int index, Intersection &intersection, vec3 eye,
                                 bool diffuse, bool specular);
vec3 getAreaLightIlluminationAt(int index, Intersection &intersection, vec3 eye);
vec3 getAreaLightIlluminationAt(int index, Intersection &intersection, vec3 eye,
                                 vec2 offset, bool diffuse, bool specular);
```

El primer método calcula la iluminación ejecutando los métodos posteriores en cada fuente de luz según el tipo.

El segundo método corresponde al cálculo de la iluminación en un punto dada una luz puntual. Se siguen los pasos diseñados en la sección 3.3.3.2 y se añaden dos variables para indicar si se quieren calcular las componentes difusa y/o especular.

El segundo y tercer método corresponden al cálculo de la iluminación en un punto dada una luz de área. El procedimiento que se sigue es:

- Calcular la iluminación producida por cada punto de la luz de área, tratando cada punto de la luz como una luz puntual pero acumulando el resultado.
- El resultado se divide entre el número de puntos de la luz.

En el tercer método también se permite elegir si se quiere calcular la componente difusa y la especular.

El motivo por el que se tienen dos métodos es porque al implementar de la iluminación de luces de área se ha visto que los resultados de por sí no eran del todo correctos.

Para solventar este "problema" se ha alterado ligeramente la posición de cada punto mediante la generación de números aleatorios. El máximo desplazamiento corresponde al espacio entre puntos. En la siguiente figura se muestra el porqué de utilizar este nuevo método.

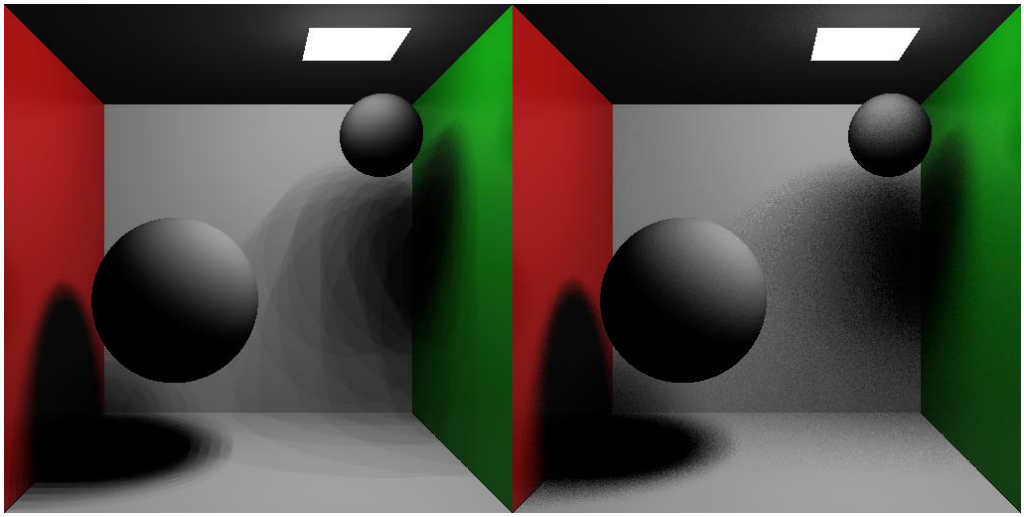


Figura 30. Comparativa de métodos para una luz de área con 5x5 puntos.

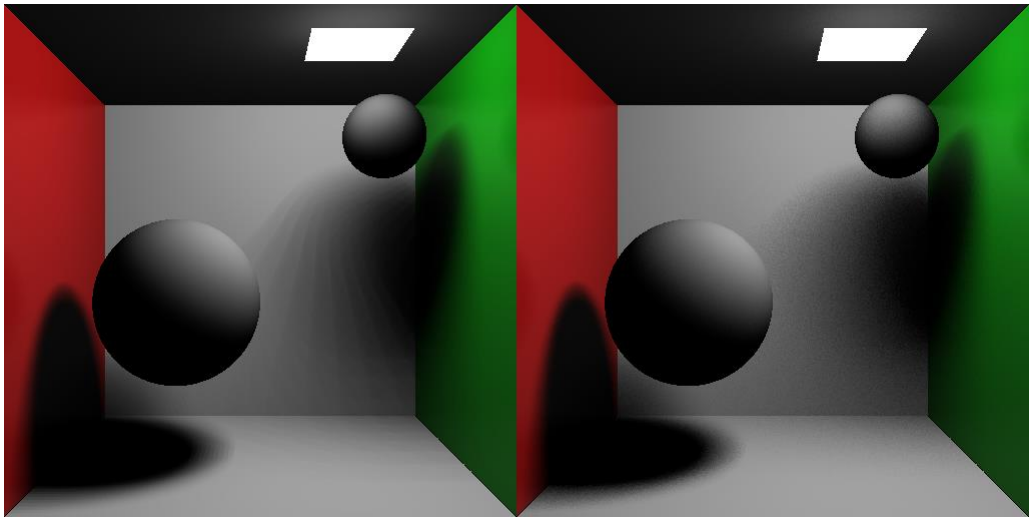


Figura 31. Comparativa de métodos para una luz de área con 10x10 puntos.

Se puede apreciar en las imágenes que sin alterar ligeramente los puntos de la luz las sombras producen resultados extraños, al contrario que alterando los puntos donde se ve que se produce una especie de efecto de *dithering* y produce unos resultados mucho mejores.

Con el primer método es posible deshacerse de estas malas representaciones de sombras aumentando significativamente el número de puntos con que se discretiza la luz de área, pero esto conlleva un mayor coste temporal de ejecución únicamente para obtener resultados similares al segundo método con un número de puntos muy menor.

5.9.5 Iluminación indirecta

El cálculo de la iluminación indirecta se realiza en el algoritmo de trazado general siguiendo los pasos descritos en la fase de diseño (sección 3.3.3.3). En la implementación se ha sumado toda la iluminación calculada para cada rayo y se ha dividido ese valor por el número de rayos trazado. Posteriormente se ha multiplicado por un factor para acentuar o atenuar esta iluminación, el sangrado de color o *color bleeding*.

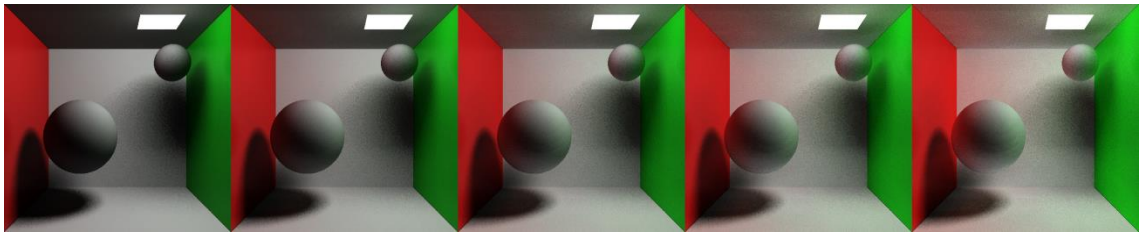


Figura 32. Diferentes valores de *Color Bleeding*.

El único detalle concreto relativo a la implementación en esta sección corresponde a cómo se trazan los rayos con forma de semiesfera.

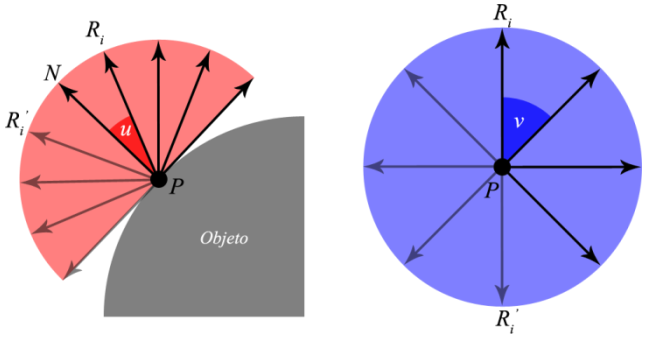


Figura 33. Obtención de los rayos en forma de semiesfera para una calidad indirecta = 4.

Se parte del vector normal *N* y se le aplican dos rotaciones para obtener el vector *R*, una con eje en la tangente y con un ángulo *u* y otra con eje en la misma normal y con un ángulo *v*. El vector *R'* se obtiene aplicando una rotación de 180º al vector *R*.

El ángulo *u* se obtiene dividiendo $\pi/2$ radianes (90º) entre el valor de la calidad indirecta. El ángulo *v* se obtiene dividiendo π radianes (180º) entre el valor de la calidad indirecta.

El siguiente pseudocódigo muestra el cálculo de estos vectores para toda la semiesfera.

```
Rotación (vector, eje, ángulo)

Para i=1 hasta calidadIndirecta
    Para j=1 hasta calidadIndirecta
        R = Rotación(N, Tangente,  j*u)
        R = Rotación(R, N, i*v)
        R' = Rotación (R, N, π)

        # Cálculo de la iluminación indirecta para R y R' #

    Fin para
Fin para
```

En la implementación se ha tenido que añadir un factor aleatorio a la rotación de los vectores para obtener *R*. El motivo de hacer esto es que en el resultado se han producido efectos similares a las sombras producidas por luces de área sin el desplazamiento de los puntos. El nuevo cálculo de *R* es:

```
R = Rotación(N, Tangente, j*u + aleatorio*u)
R = Rotación(R, N, i*v + aleatorio*v)
```

Las siguientes imágenes ilustran la diferencia entre aplicar el factor aleatorio y no aplicarlo.

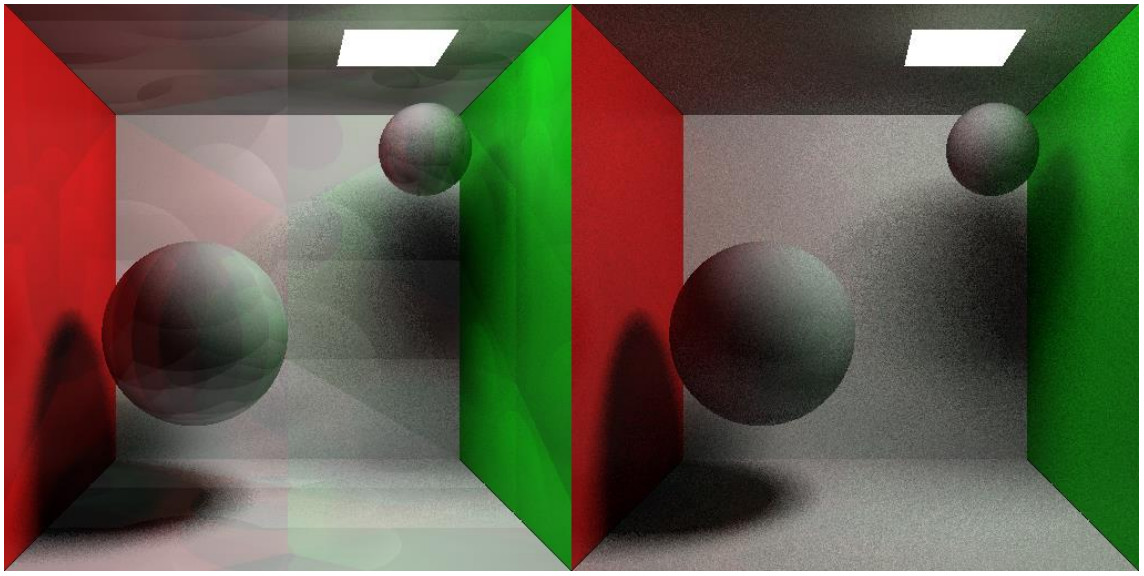


Figura 34. Comparativa entre utilizar o no el factor aleatorio para una calidad indirecta = 3.

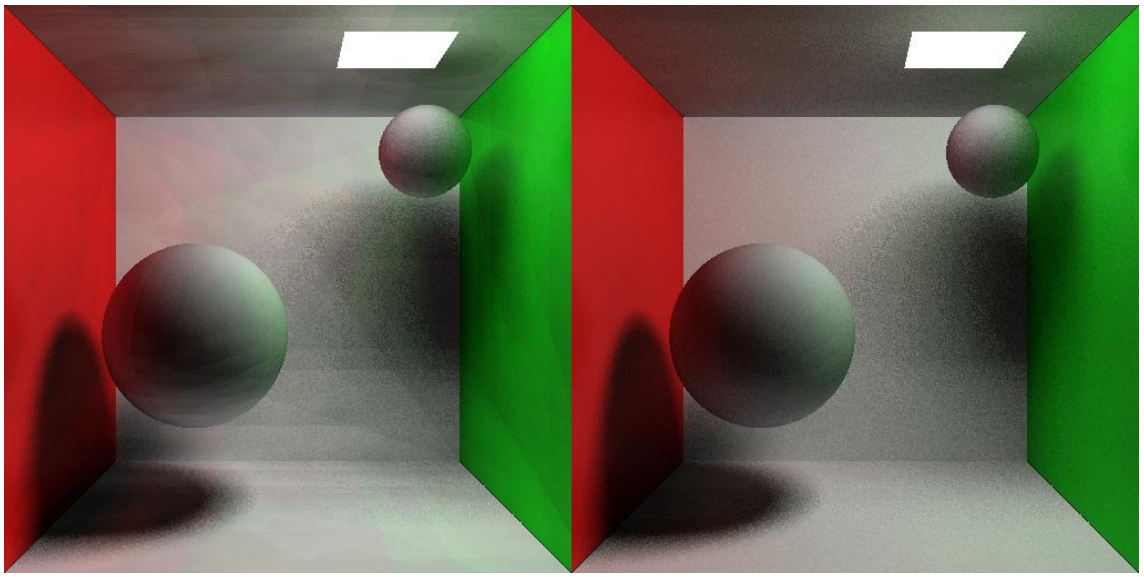


Figura 35. Comparativa entre utilizar o no el factor aleatorio para una calidad indirecta = 5.

5.9.6 Reflejos

La implementación del trazado de los reflejos sigue los pasos diseñados en la sección 3.3.4.

Para los reflejos con ángulo sólido se sigue el mismo esquema que la sección anterior, pero en esta parte se calcula el ángulo u como:

$$u = \left(\frac{\acute{a}ngulo\ de\ reflejo}{2} \times \frac{\pi}{180}\right)\ radianes$$

La siguiente imagen muestra el resultado de calcular reflejos con distintos valores de ángulo sólido.

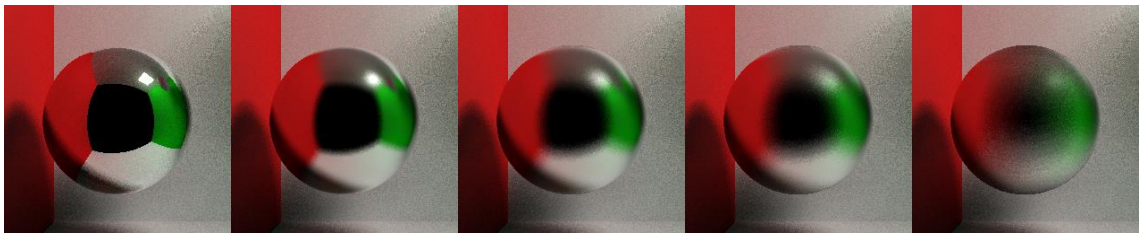


Figura 36. Reflejos con ángulo sólido (0º, 15º, 30º, 45º y 90º).

En la implementación se ha tenido que añadir el mismo factor aleatorio a la rotación de los vectores que en la iluminación indirecta. El motivo de realizar este ajuste ha sido los malos resultados obtenidos de los reflejos en cuanto a calidad (es más fácil apreciarlo en las siguientes imágenes).

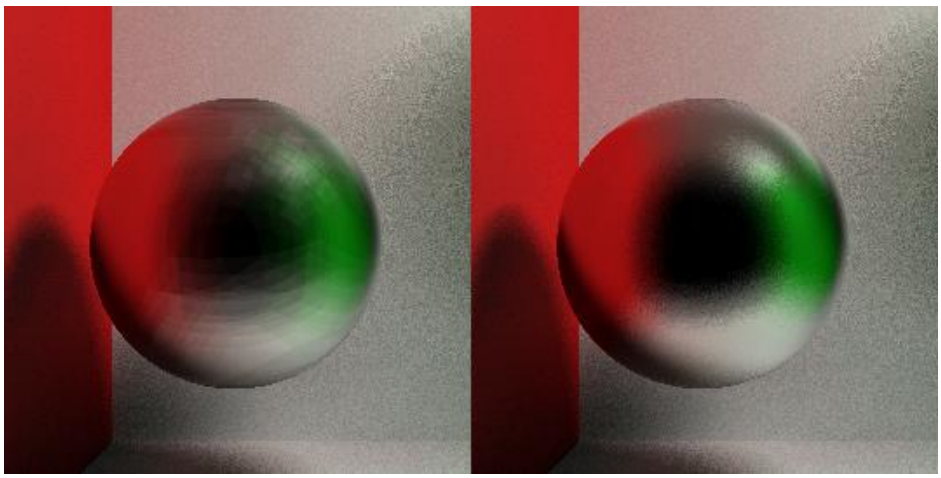


Figura 37. Comparativa sin el ajuste y con el ajuste para un ángulo de 45º y una calidad de reflejos = 3.

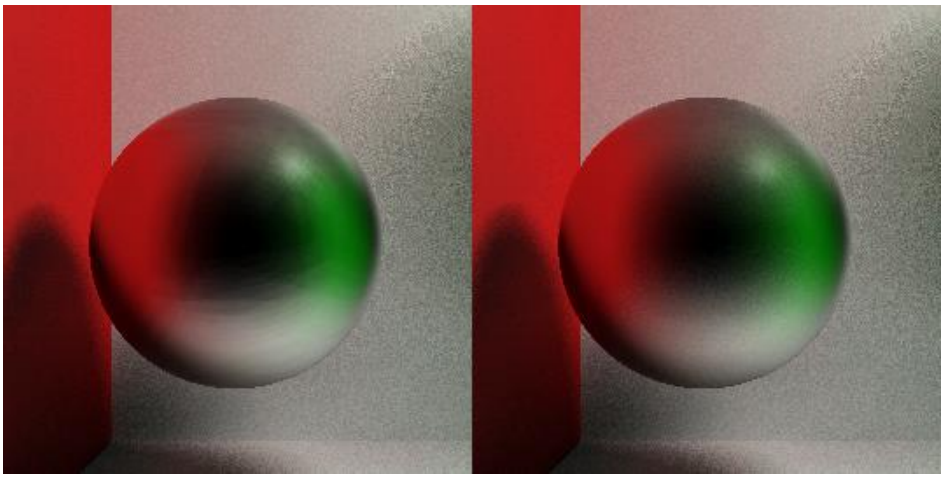


Figura 38. Comparativa sin el ajuste y con el ajuste para un ángulo de 45º y una calidad de reflejos = 5.

Como se puede apreciar en las imágenes el ajuste realizado es necesario y sirve para obtener un mejor resultado visual.

5.9.7 Transmisión

Para la implementación de la transmisión se han realizado mediante el algoritmo de trazado general y un método en la clase *DeviceScene*.

En primer lugar, en el algoritmo principal, tras encontrar la intersección más cercana se calcula el vector de transmisión para el rayo y se ejecuta el siguiente método de *DeviceScene*.

```
int  getTransmission(vec3 transmissionRay, Intersection &in, Intersection &out, vec3 &rayOut);
```

La función sigue el procedimiento descrito en la fase de diseño (sección 3.3.5) y recibe el vector de transmisión, la intersección del objeto más cercano, un puntero a una intersección (resultado de la intersección saliente) y la dirección del rayo al salir del objeto.

Posteriormente se ejecuta el algoritmo general para obtener el color en ese punto de la escena. Este color se asigna al color de transmisión.

5.9.8 Problemas

Uno de los problemas encontrados en el trazador de rayos ha sido la "autosombra". Este fenómeno ocurre cuando, debido a la precisión limitada del hardware, el punto de intersección calculado no se corresponde con el real sino que se encuentra desplazado ligeramente.

Para solucionar esto todos los puntos de intersección calculados se desplazan mínimamente en dirección opuesta al rayo.

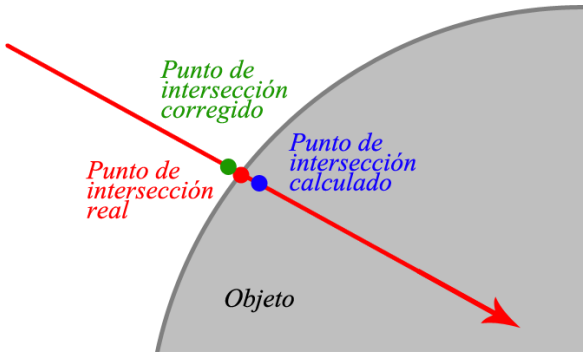


Figura 39. Cálculo erróneo que produce autosombra y punto corregido.

6. Resultados

En esta sección se van a presentar una serie de escenas de ejemplo para ilustrar todos los efectos que se consiguen con el programa realizado. Se van a indicar una serie datos relativos a cada ejemplo:

- En primer lugar se va a realizar una breve descripción de la escena, los elementos que la componen y las propiedades de escena establecidas.
- Se indicará la resolución de la imagen resultante, ancho x alto, en píxeles.
- Se indicará el número de rayos que se generan por píxel. Este número podrá no ser concreto puesto que no todos los elementos de la escena requerirán de los mismos cálculos.
- Se indicarán una serie de tiempos obtenidos.
    - Crear escena: tiempo empleado en leer el fichero escena, procesarlo y construir la escena.
    - Copiar escena: tiempo empleado en copiar la escena de CPU a GPU.
    - Ejecución: tiempo empleado en el proceso de trazado de rayos (ejecución del kernel).

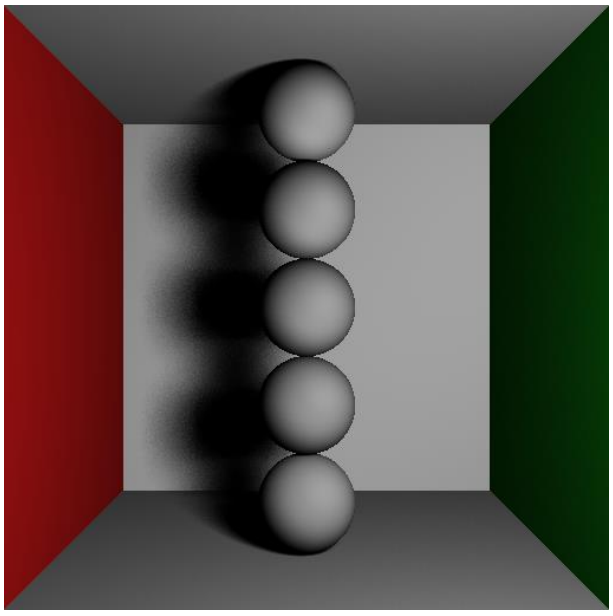
Los diferentes ficheros de escena correspondientes a los ejemplos se encuentran en el anexo D.

Los resultados se han obtenido mediante una GPU NVIDIA GTX 560 con las siguientes características:

- 336 núcleos CUDA a 1.7Ghz.
- Versíon del driver/versión del runtime/versión de capacidad de CUDA: 6.0 / 5.0 / 2.1
- Memoria total: 1024 MB (2 niveles de cache) a 2.1 GHz

Escena 1

Esta escena está formada por 5 cubos que forman una *Cornell Box* improvisada, 5 esferas, 3 materiales y una luz de área con 100 puntos. En esta escena solo hay iluminación directa.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101
- 1 rayo principal
- 100 rayos de sombra

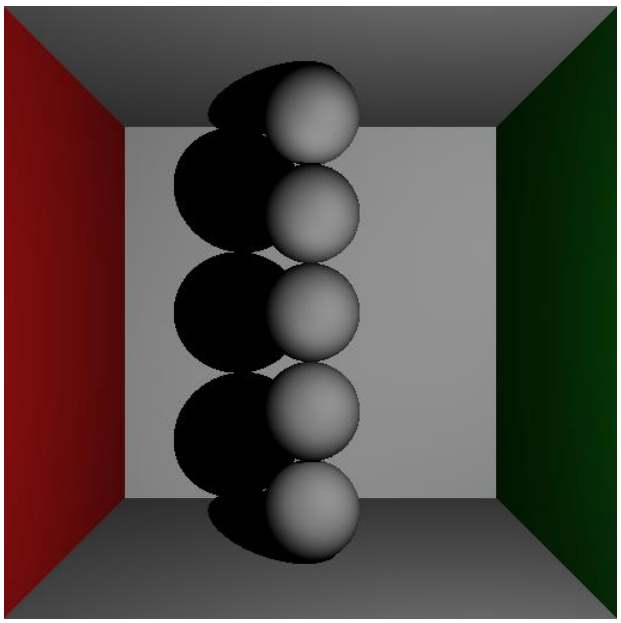
Crear escena: 0.0123 segundos

Copiar escena: 0.0218 segundos

Ejecución: 0.5203 segundos

Figura 40. Escena 1.

En esta escena se puede observar el efecto de penumbra producido por la luz de área. Si se cambia la luz por una luz puntual en la misma posición y con la misma intensidad el resultado es el siguiente.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 2
- 1 rayo principal
- 1 rayo de sombra

Crear escena: 0.0130 segundos

Copiar escena: 0.0069 segundos

Ejecución: 0.1981 segundos

Figura 41. Escena 1 con una luz puntual.

La calidad de las sombras es muy superior con la luz de área y el coste temporal, siendo 100 rayos más, no es 100 veces mayor.

Se va a aprovechar esta escena para cambiar el parámetro *Antialiasing* de las propiedades de la escena e ilustrar el efecto que tiene el sobremuestreo.

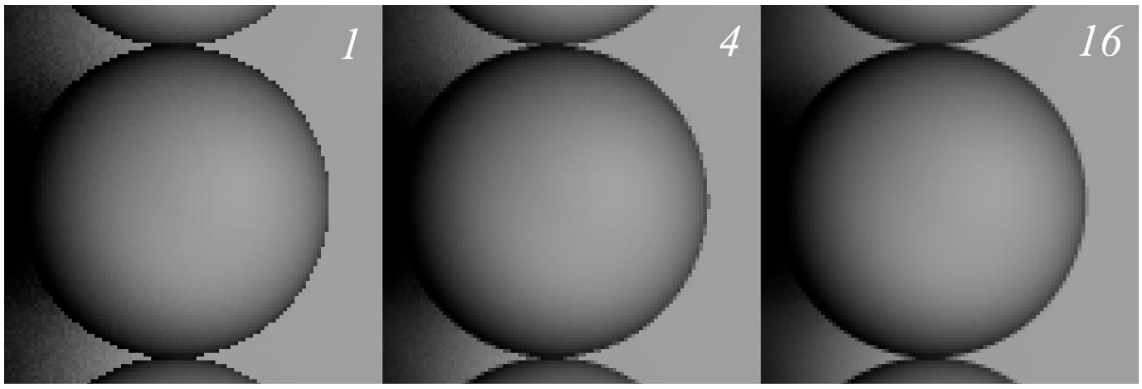


Figura 42. Escena 1 con diferentes valores de antialiasing (nº de muestras).

En la imagen se puede observar una mejora significativa de la calidad al aplicar antialiasing, tanto por suavizar los bordes de los objetos como por eliminar el ruido presente en la penubra. Mirando los tiempos de ejecución se puede ver que el incremento del coste temporal es lineal, considerando que para el antialiasing se lanzan dos kernels a ejecución.

	Luz de área	Puntual
Tiempo de ejecución (sin):	0.5203 segundos	0.1981 segundos
Tiempo de ejecución (4 muestras):	2.0563 segundos	0.7770 segundos
Tiempo de ejecución (16 muestras):	8.2076 segundos	3.0927 segundos

Escena 2

En esta escena se utilizan los 5 cubos de la escena anterior, una luz de área con 100 puntos y una malla poligonal de 70064 polígonos (modelo original http://graphics.stanford.edu/data/3Dscanrep/, versión OBJ obtenida de https://dt-grid.googlecode.com/files/stanford_bunny.obj).

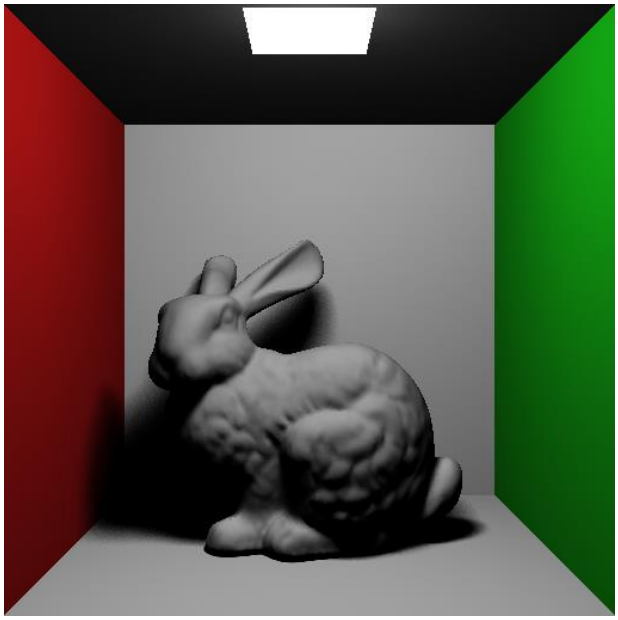


Figura 43. Escena 2.

Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101

- 1 rayo principal
- 100 rayos de sombra

Crear escena: 13.0486 segundos

Copiar escena: 0.4435 segundos

Ejecución: 5.0687 segundos

En esta escena también se va a cambiar la luz por una luz puntual y se va a evaluar el coste temporal.

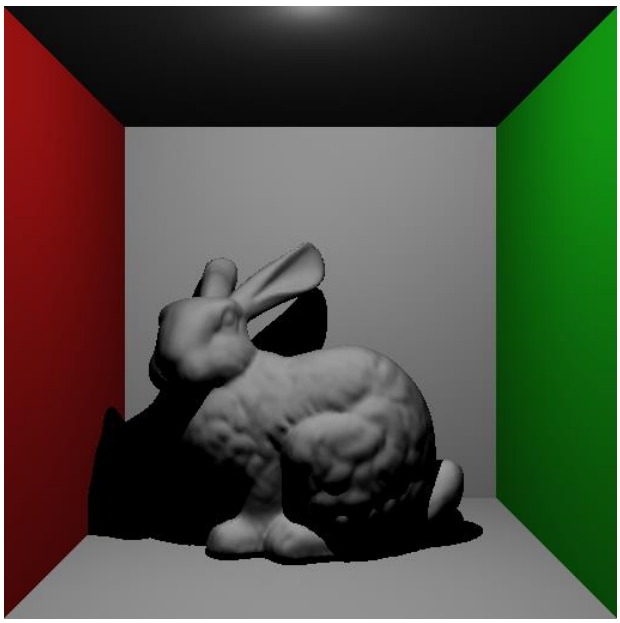


Figura 44. Escena 2 con una luz puntual.

Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 2

- 1 rayo principal
- 1 rayo de sombra

Crear escena: 13.0789 segundos

Copiar escena: 0.4260 segundos

Ejecución: 0.2709 segundos

En los tiempos obtenidos se puede observar el impacto que tiene el utilizar mallas poligonales. Por una parte el procesado de la malla emplea un tiempo considerable, aunque el número de polígonos también es muy alto. Por otra parte se puede ver que el cálculo de intersecciones con mallas poligonales es más lento que el cálculo de intersecciones con primitivas, aun así el resultado es muy bueno comparando con recorrer todos los polígonos por fuerza bruta.

Los tiempos de ejecución también muestran que, para las mallas poligonales, las luces de área tienen un coste temporal mayor que para las primitivas siendo este coste aproximadamente unas 20 veces superior.

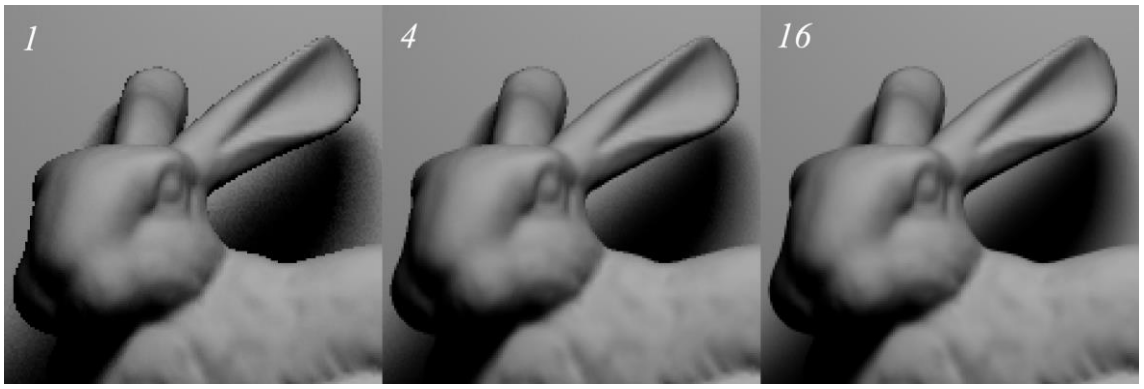


Figura 45. Escena 2 con diferentes valores de antialiasing (nº de muestras).

Aumentar el valor de antialiasing en el caso de las mallas también conlleva un aumento lineal del coste temporal, al igual que en la escena anterior.

	Luz de área	Luz puntual
Tiempo de ejecución (sin):	5.0687 segundos	0.2709 segundos
Tiempo de ejecución (4 muestras):	19.9315 segundos	1.0679 segundos
Tiempo de ejecución (16 muestras):	79.2492 segundos	4.2579 segundos

En esta escena se va a aprovechar para ver el efecto que tiene el modificar el parámetro "rugosidad" (roughness) del material *matWhiteSpecular* definido en la escena y que únicamente está asignado a la malla.

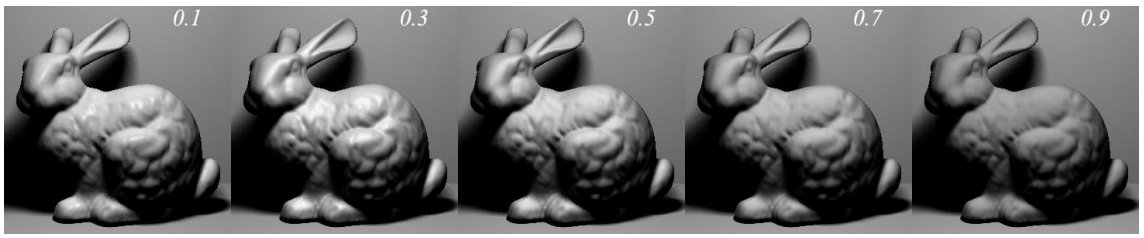
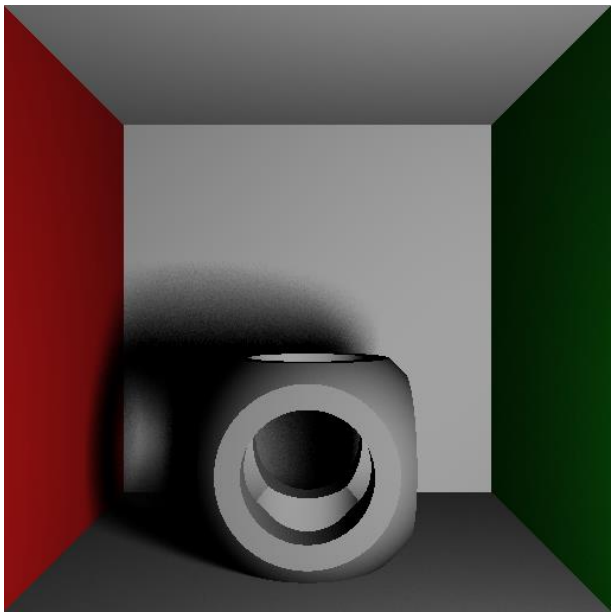


Figura 46. Comparativa de diferentes valores de rugosidad en un material de color blanco.

En la imagen se puede observar cómo afecta el parámetro *roughness* a las componentes del material, sobre todo a la especular.

Escena 3

Esta escena parte de la escena 1 y se han sustituido las esferas por un objeto CSG con los mismos elementos que el de la figura 17 (3 cilindros, 1 esfera, 1 cubo y 4 operaciones). Se van a comparar los tiempos de ejecución para una luz de área y para una luz puntual.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101

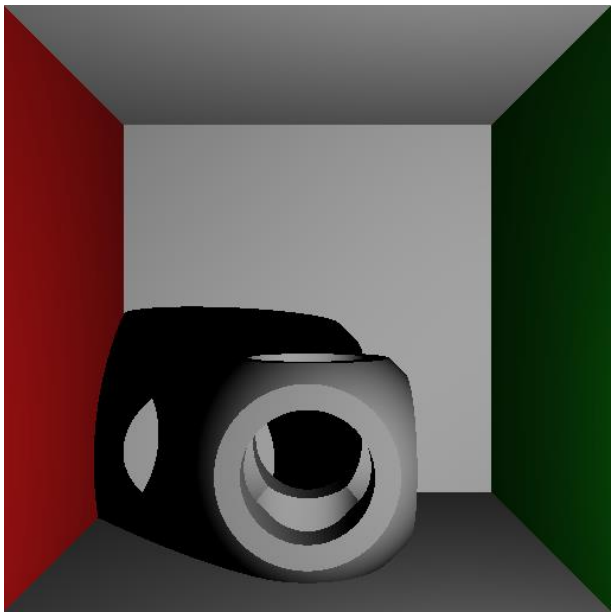
- 1 rayo principal + 100 rayos de sombra

Crear escena: 0.0143 segundos

Copiar escena: 0.0202 segundos

Ejecución: 15.7502 segundos

Figura 47. Escena 3 con luz de área.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 2

- 1 rayo principal + 1 rayo de sombra

Crear escena: 0.0132 segundos

Copiar escena: 0.0073 segundos

Ejecución: 0.4630 segundos

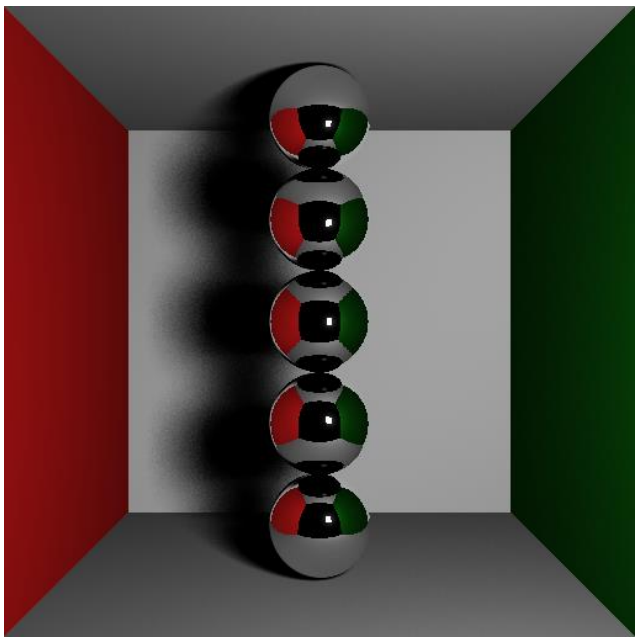
Figura 48. Escena 3 con luz puntual.

Los tiempos de ejecución han aumentado al utilizar objetos CSG respecto a utilizar primitivas. Esto se debe a que cada rayo ha de reservar espacio dinámicamente, recorrer los nodos y procesar la lista.

Escena 4

Para esta escena se ha partido de la escena 1 y se ha modificado el material de las esferas para que sean totalmente reflejantes (reflejos simples). El número máximo de rebotes (*ReflectionSteps*) se ha establecido en 1.

Se puede observar que al no poder rebotar los rayos de reflejo, el reflejo de una esfera no reproduce los reflejos de las demás esferas. Aumentando el número de rebotes se consigue un mejor resultado.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101 o 102 (2 casos)

- 1 rayo principal + 100 rayos de sombra (objetos no reflejantes)
- 1 rayo principal + 1 rayo de reflejo + 100 rayos de sombra (objetos reflejantes)

Crear escena: 0.0194 segundos

Copiar escena: 0.0252 segundos

Ejecución: 0.5544 segundos

Figura 49. Escena 4.

A continuación se ha variado el número de rebotes.

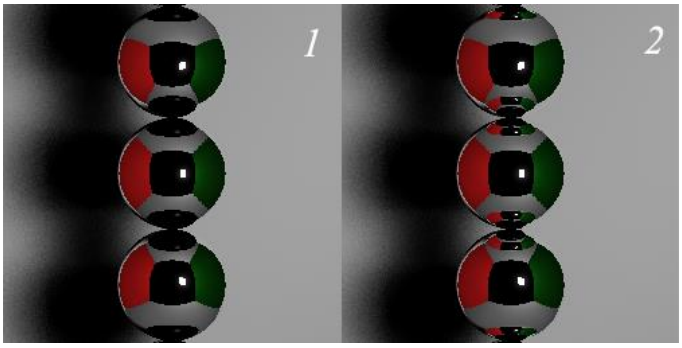


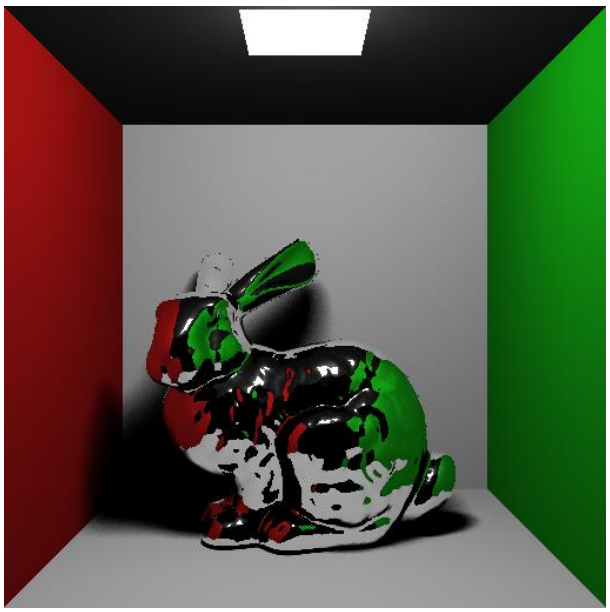
Figura 50. Escena 4 con diferente número de rebotes para los reflejos.

Con un mayor número de rebotes se pueden observar los reflejos que produce una esfera sobre otra esfera. Se produce un mínimo incremento en el tiempo de ejecución debido al mayor número de rayos.

Tiempo de ejecución (1 rebote):	0.5544 segundos
Tiempo de ejecución (2 rebotes):	0.5611 segundos

Escena 5

Para esta escena se parte de la escena 2 y se asigna un material reflejante (reflejos simples) para la malla.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101 o 102 (2 casos)

- 1 rayo principal + 100 rayos de sombra (objetos no reflejantes)
- 1 rayo principal + 1 rayo de reflejo + 100 rayos de sombra (objetos reflejantes)

Crear escena: 13.1108 segundos

Copiar escena: 0.4410 segundos

Ejecución: 7.9027 segundos

Figura 51. Escena 5.

A continuación se ha variado el número de rebotes.

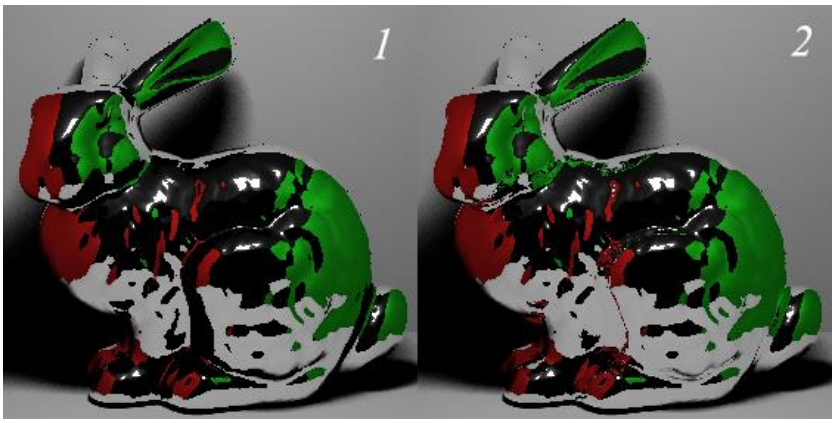


Figura 52. Escena 5 con diferente número de rebotes para los reflejos.

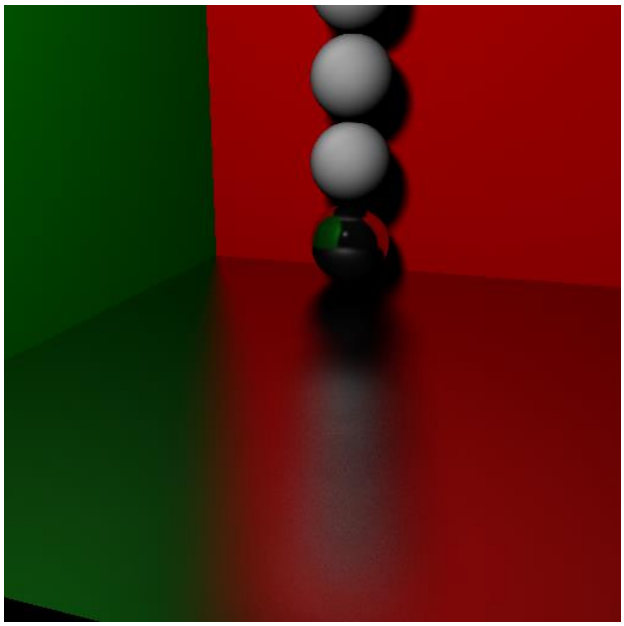
Con un mayor número de rebotes se pueden observar los reflejos que produce la malla sobre sí misma. Se produce un pequeño incremento (mayor que en el caso de la escena 4) en el tiempo de ejecución debido al mayor número de rayos.

Tiempo de ejecución (1 rebote): 7.9027 segundos

Tiempo de ejecución (2 rebotes): 8.4511 segundos

Escena 6

Esta escena se compone de 5 esferas y 3 cubos. Se asigna un material parcialmente reflejante, con ángulo sólido de 15º, a uno de los cubos y a una de las esferas para evaluar el efecto del ángulo sólido sobre los reflejos obtenidos. Se establece una calidad de reflejos igual a 5.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101 o 5101 (2 casos)

- 1 rayo principal + 100 rayos de sombra (objetos no reflejantes)
- 1 rayo principal + 100 rayos de sombra + (50 rayos de reflejo x 100 rayos de sombra por cada rayo de reflejo) (objetos reflejantes)

Crear escena: 0.0121 segundos

Copiar escena: 0.0218 segundos

Ejecución: 8.2046 segundos

Figura 53. Escena 6.

Al igual que en las escenas anteriores si se aumenta el número de rebotes la calidad de los reflejos es mayor.

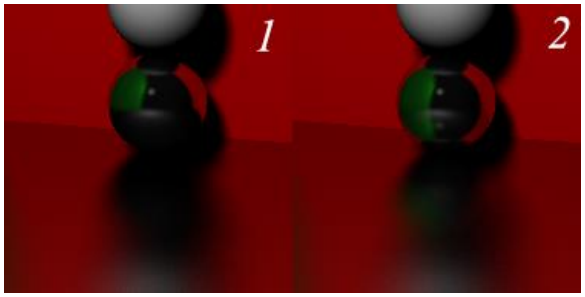


Figura 54. Escena 6 con diferente número de rebotes para los reflejos.

Los tiempos de ejecución no muestran una linealidad en el coste temporal al aumentar el número de rebotes.

Tiempo de ejecución (1 rebote): 8.2046 segundos

Tiempo de ejecución (2 rebotes): 24.6366 segundos

Escena 7

Esta escena se compone de una serie de cubos, unas esferas totalmente transparentes (con un índice de refracción de 1.5) y una luz de área.

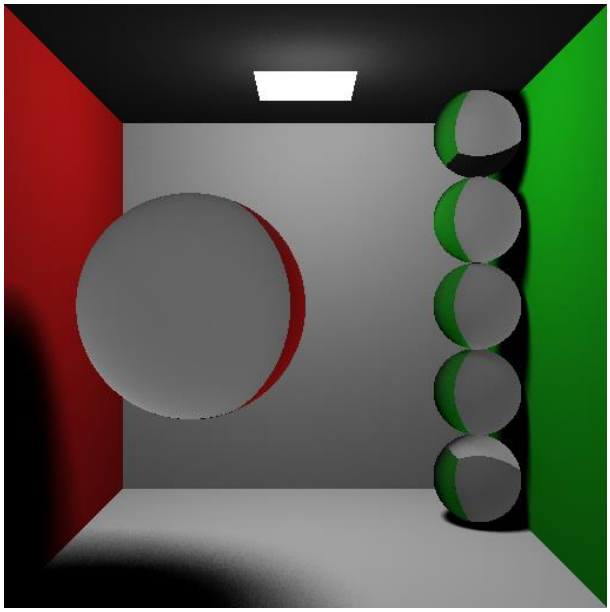


Figura 55. Escena 7.

Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 102

- 1 rayo principal + 1 rayo de transmisión + 100 rayos de sombra.

Crear escena: 0.0158 segundos

Copiar escena: 0.0232 segundos

Ejecución: 0.6008 segundos

Se produce un pequeño incremento en el tiempo de ejecución al incrementar en 1 el número de rayos trazados.

Para diferentes valores del índice de refracción se obtienen los siguientes resultados.

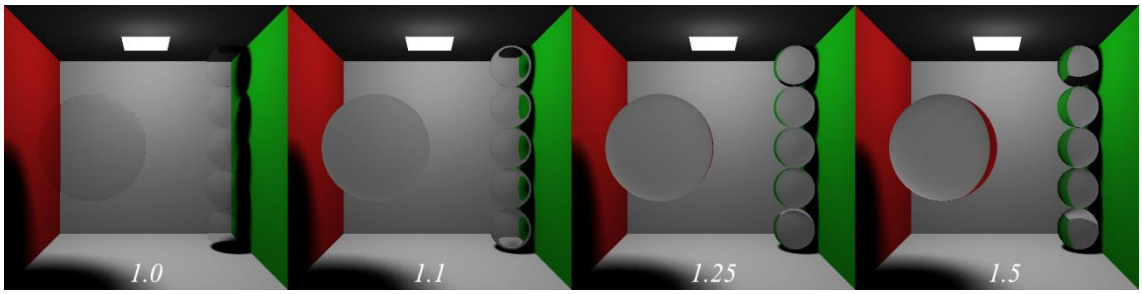
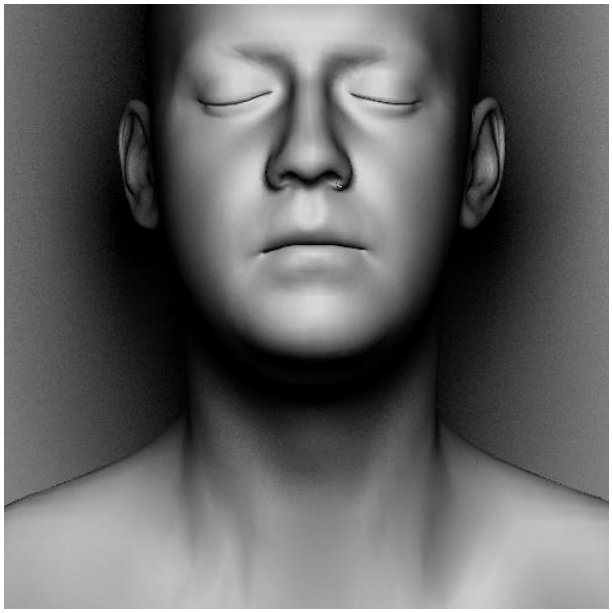


Figura 56. Escena 7 con diferentes valores de índice de refracción.

Escena 8

En esta escena se va utilizar una malla de 14581 polígonos y varias texturas. La malla se ha obtenido de http://ir-ltd.net/ . En primer lugar se va a renderizar la escena con la malla sin texturas.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101

- 1 rayo principal + 100 rayos de sombra.

Crear escena: 1.9409 segundos

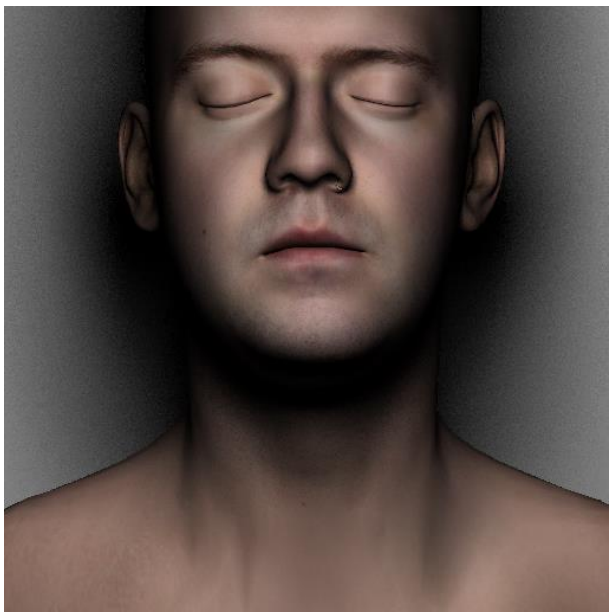
Copiar escena: 0.2271 segundos

Ejecución: 9.3449 segundos

Figura 57. Escena 8. Malla sin texturas.

Comparando con la escena 2 el tiempo de procesado y de copiado es menor al tener una malla de menos polígonos. Se puede observar también que el tiempo de ejecución es mayor en esta escena que en la escena 2, esto es debido a que si un rayo no intersecta con el cubo raíz que divide el espacio entonces no hay intersección y no se realizan más cálculos (en la escena 2 la malla no ocupa toda la imagen y por eso se realizan menos cálculos).

A continuación se va a renderizar la escena con una textura difusa asignada al material de la malla.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101

- 1 rayo principal + 100 rayos de sombra.

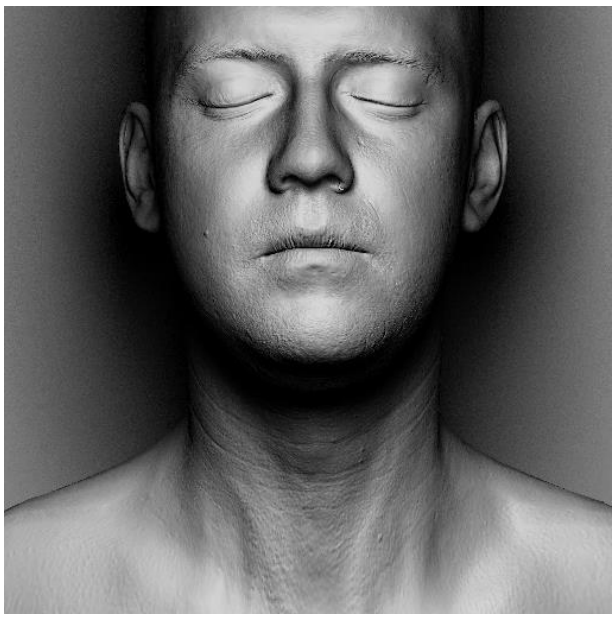
Crear escena: 2.0008 segundos

Copiar escena: 0.2176 segundos

Ejecución: 9.3534 segundos

Figura 58. Escena 8. Malla con textura difusa.

Se va a renderizar la escena con un mapa de normales asignado al material de la malla, sin textura difusa.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101

- 1 rayo principal + 100 rayos de sombra.

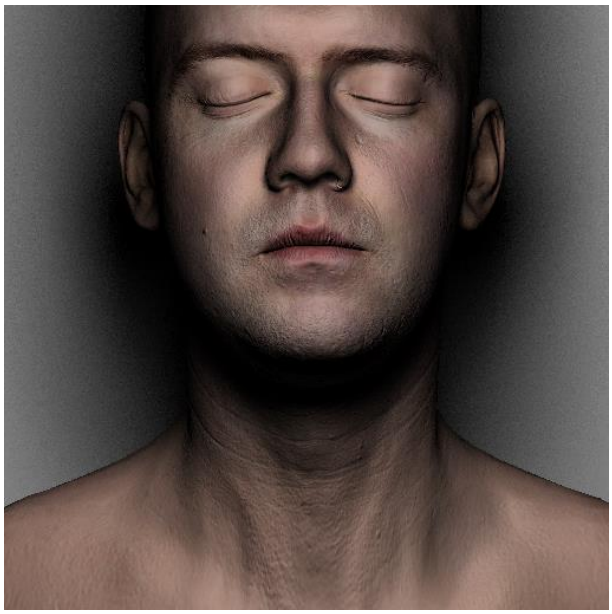
Crear escena: 1.9942 segundos

Copiar escena: 0.2185 segundos

Ejecución: 9.3393 segundos

Figura 59. Escena 8. Malla con mapa de normales.

Para finalizar con esta escena, se va a renderizar la escena con todas las texturas anteriores.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 101

- 1 rayo principal + 100 rayos de sombra.

Crear escena: 1.9868 segundos

Copiar escena: 0.2282 segundos

Ejecución: 9.3504 segundos

Figura 60. Escena 8. Malla con todas las texturas.

Viendo los tiempos de ejecución no existe un incremento apreciable del coste temporal al hecho de utilizar o no texturas.

Escena 9

Se ha utilizado la escena 1 como base, pero con la luz de área con 25 puntos, y se ha activado la iluminación indirecta con un factor de *ColorBleeding* igual a 4.

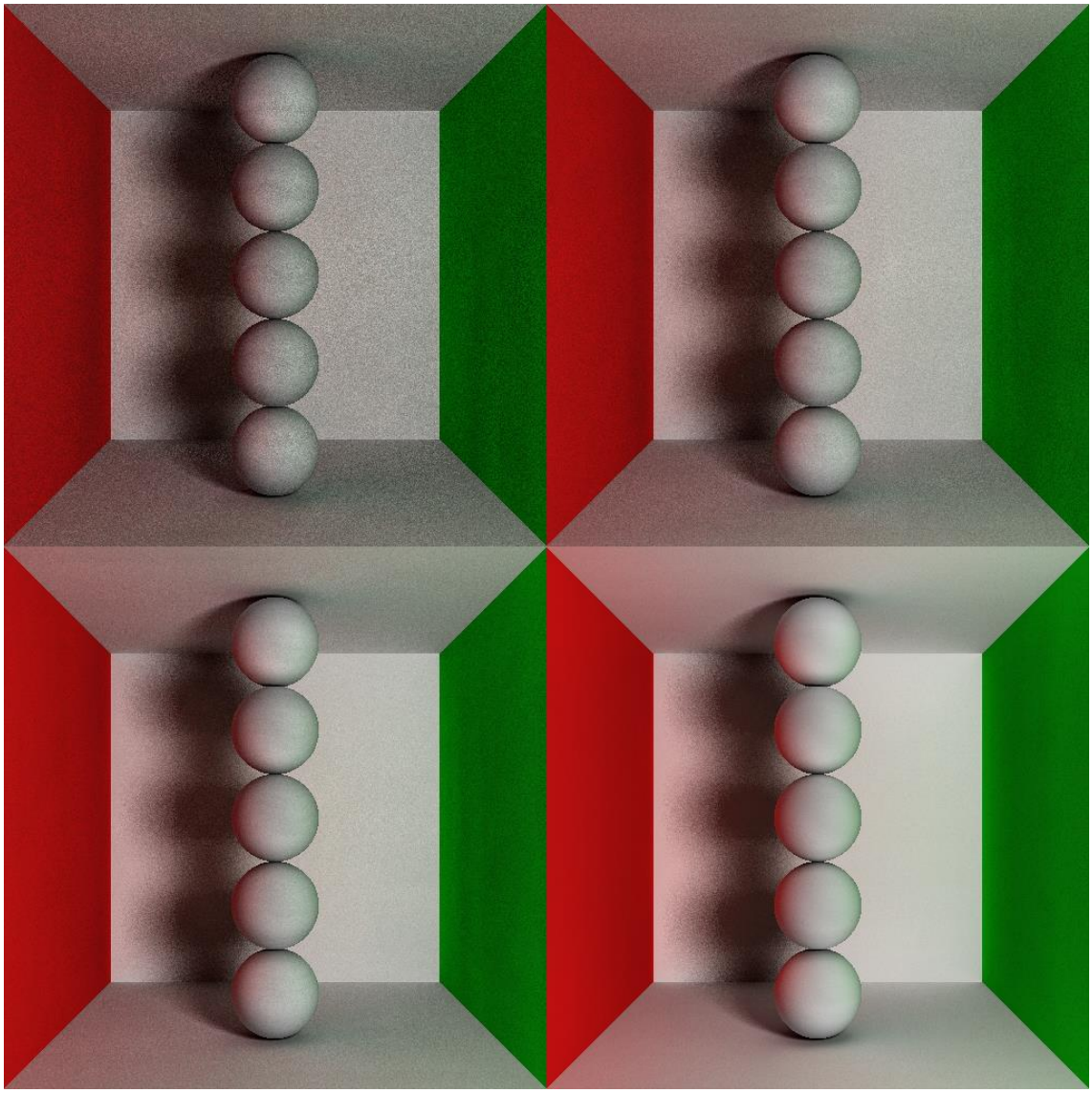


Figura 61. Escena 9 con una luz de área y con 4 valores diferentes de calidad indirecta (de izquierda a derecha y de arriba abajo): 3, 4, 5, 10.

Para una resolución de 512 x 512 sin antialiasing los tiempos de ejecución y el número de rayos son:

	Rayos por píxel	Tiempo de ejecución
Calidad = 3:	1+25+18+(18*25) = 494	1.7885 segundos
Calidad = 4:	1+25+32+(32*25) = 858	2.8930 segundos
Calidad = 5:	1+25+50+(50*25) = 1326	4.2947 segundos
Calidad = 10:	1+25+100+(100*25) = 2626	15.4304 segundos

El número de rayos se ha calculado de la siguente manera: 1 rayo primario + N rayos de sombra + M rayos de iluminación indirecta + (M rayos de iluminación indirecta x N rayos de sombra por cada rayo).

En las imágenes se puede ver como las esferas están iluminadas por la luz de área y por el rebote de esta luz sobre las paredes roja y verde. También se observa que para un número de calidad bajo el resultado no es muy bueno ya que resulta en una imagen con ruido (ruido provocado por el factor aleatorio que se ha introducido al cálculo de la iluminación indirecta, sección 5.9.5). Para eliminar este ruido existen dos opciones: aumentar la calidad, trazar más rayos, o renderizar con antialiasing.

Se ha cambiado la luz de área por una luz puntual para evaluar las diferencias en los tiempos de ejecución.

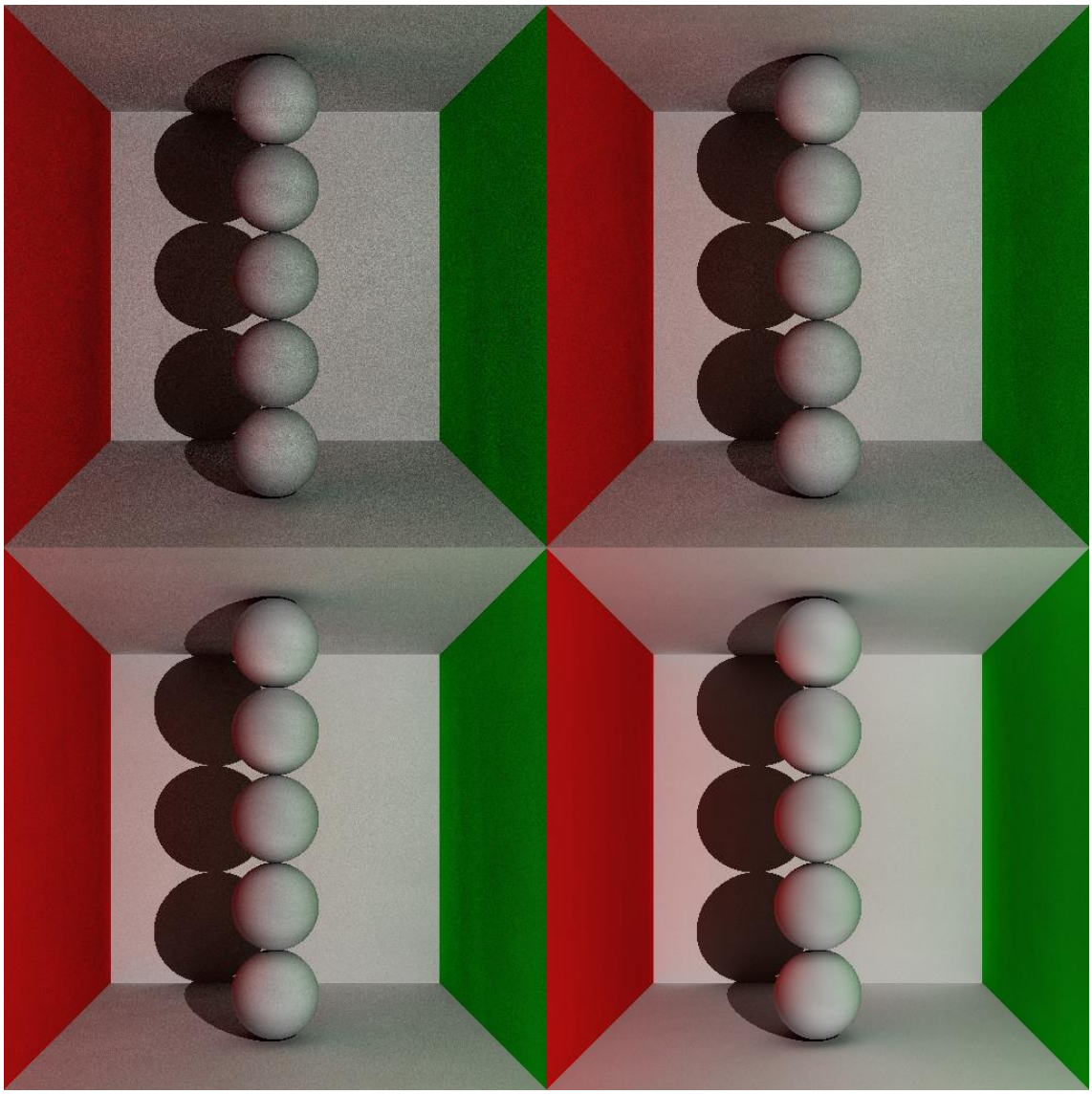


Figura 62. Escena 9 con una luz puntual y con 4 valores diferentes de calidad indirecta (de izquierda a derecha y de arriba abajo): 3, 4, 5, 10.

Para una resolución de 512 x 512 sin antialiasing los tiempos de ejecución y el número de rayos son:

	Rayos por píxel	Tiempo de ejecución
Calidad = 3:	1+1+(18)*2 = 38	0.3628 segundos
Calidad = 4:	1+1+(32)*2 = 66	0.4816 segundos
Calidad = 5:	1+1+(50)*2 = 102	0.6306 segundos
Calidad = 10:	1+1+(100)*2 = 202	1.8122 segundos

Escena 10

Se ha utilizado la escena 2 como base y se ha activado la iluminación indirecta con un factor de *ColorBleeding* igual a 4. Se va a evaluar el tiempo de ejecución y la calidad obtenida para distintos valores de calidad indirecta al igual que en la escena 9.

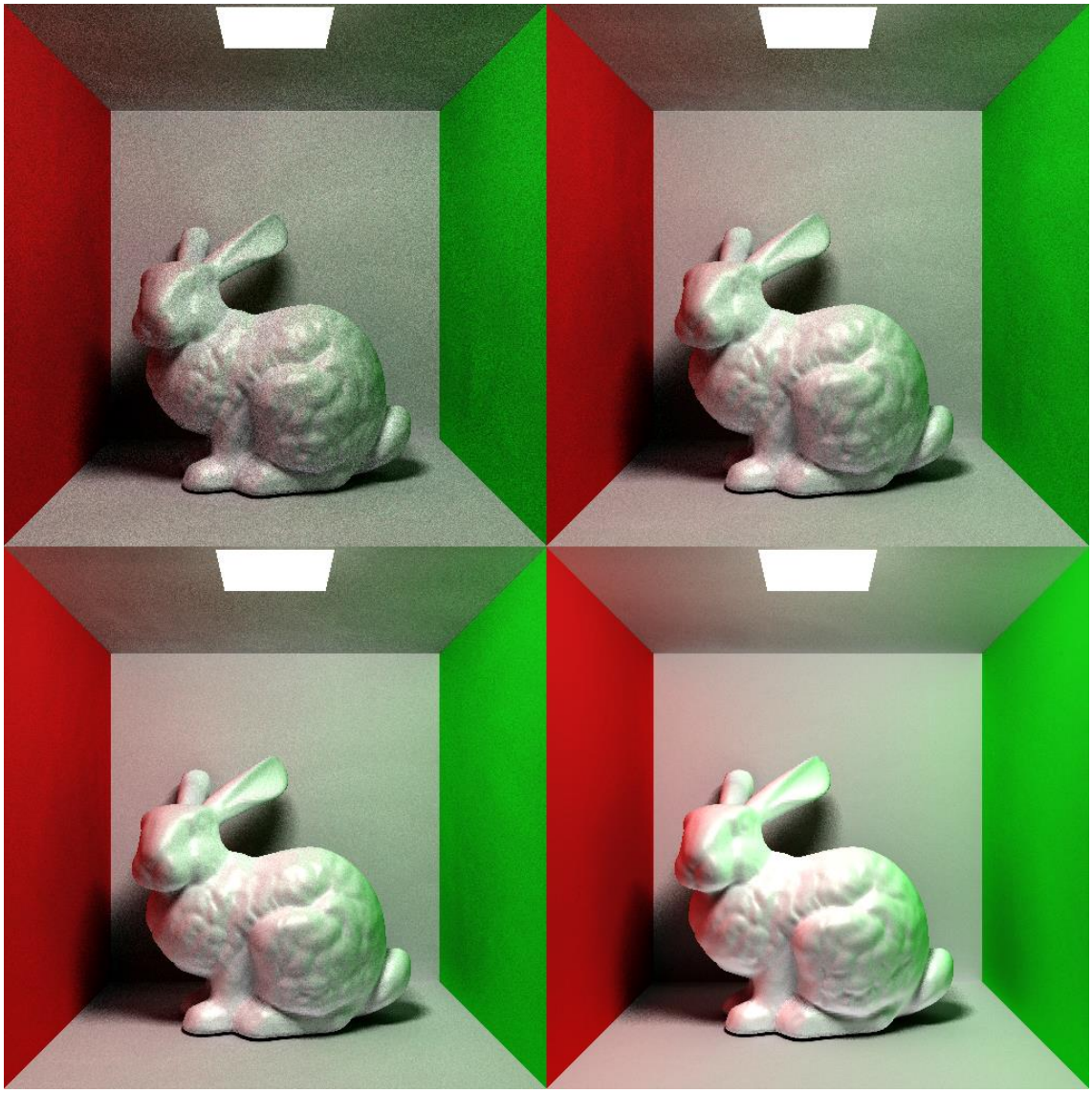


Figura 63. Escena 10 con una luz de área y con 4 valores diferentes de calidad indirecta (de izquierda a derecha y de arriba abajo): 3, 4, 5, 10.

Para una resolución de 512 x 512 sin antialiasing los tiempos de ejecución y el número de rayos son:

	Rayos por píxel	Tiempo de ejecución
Calidad = 3:	1+25+18+(18*25) = 494	82.5812 segundos
Calidad = 4:	1+25+32+(32*25) = 858	135.6396 segundos
Calidad = 5:	1+25+50+(50*25) = 1326	196.7165 segundos
Calidad = 10:	1+25+100+(100*25) = 2626	607.5739 segundos

Se ha cambiado la luz de área por una luz puntual para evaluar las diferencias en los tiempos de ejecución.

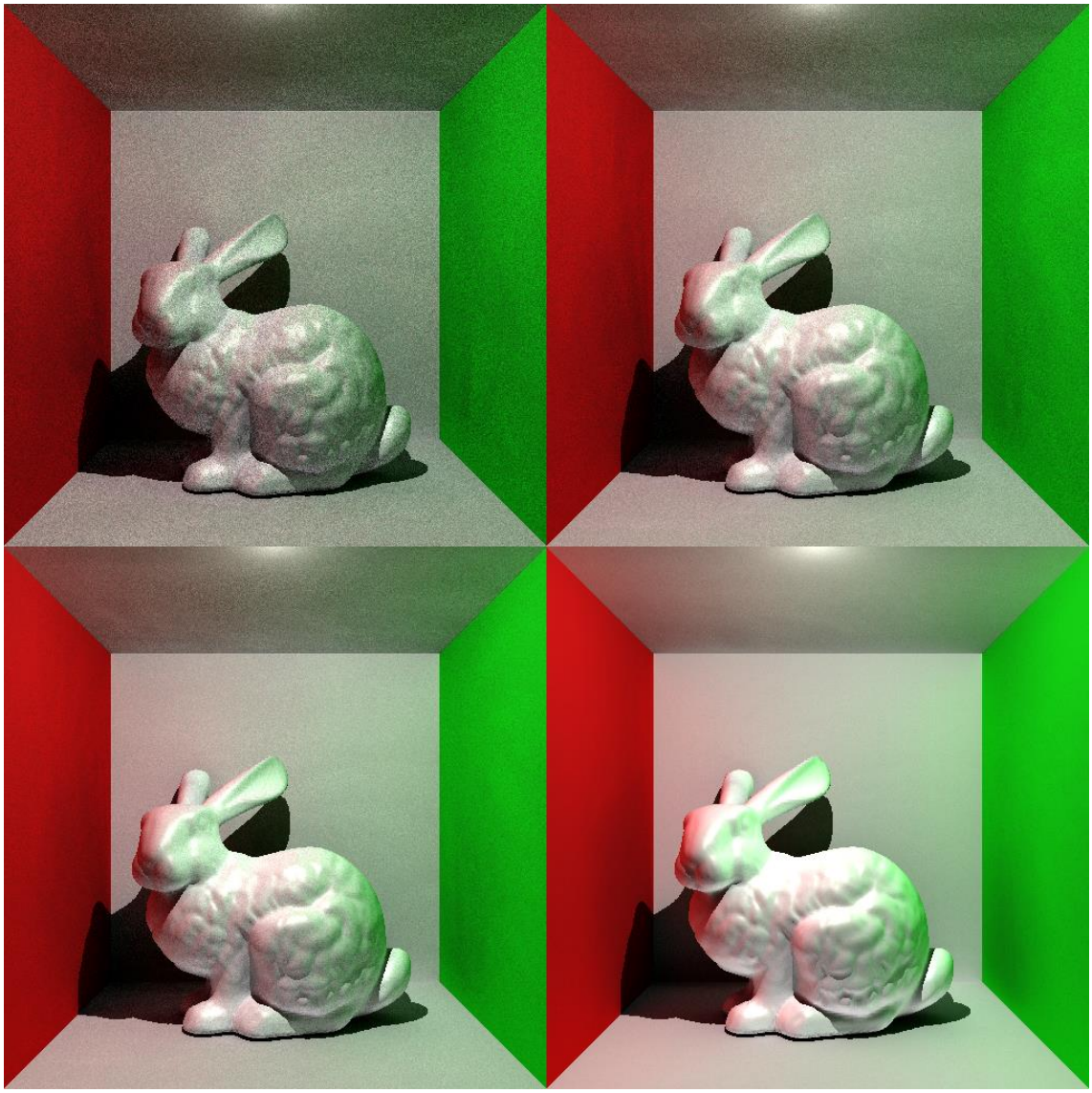
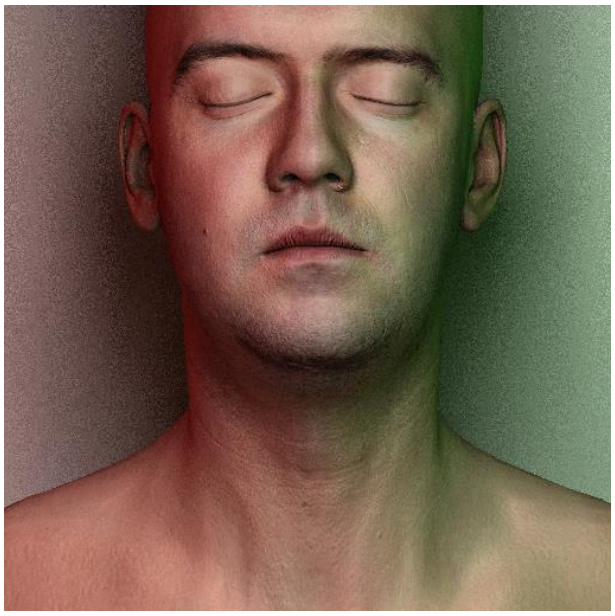


Figura 64. Escena 10 con una luz puntual y con 4 valores diferentes de calidad indirecta (de izquierda a derecha y de arriba abajo): 3, 4, 5, 10.

Para una resolución de 512 x 512 sin antialiasing los tiempos de ejecución y el número de rayos son:

	Rayos por píxel	Tiempo de ejecución
Calidad = 3:	1+1+(18)*2 = 38	7.2876 segundos
Calidad = 4:	1+1+(32)*2 = 66	11.4689 segundos
Calidad = 5:	1+1+(50)*2 = 102	16.5288 segundos
Calidad = 10:	1+1+(100)*2 = 202	48.5596 segundos

Otras escenas



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 1325

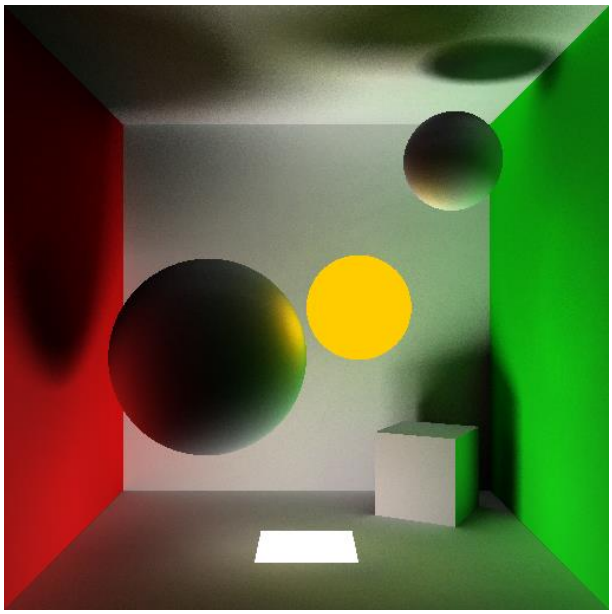
Crear escena: 1.9181 segundos

Copiar escena: 0.2177 segundos

Ejecución: 128.3698 segundos

Esta escena es la misma que la escena 7, pero en esta se ha reducido el número de puntos de la luz de área de 100 a 25 y se ha calculado la iluminación indirecta con una calidad = 5.

Figura 65. Escena extra 1.



Resolución: 512 x 512 (sin antialiasing)

Rayos por píxel: 20301

Crear escena: 0.0154 segundos

Copiar escena: 0.0204 segundos

Ejecución: 60.1217 segundos

En esta escena se han creado una serie de primitivas y una luz de área con 100 puntos. La calidad de la iluminación indirecta es 10 y no hay reflejos. Se ha definido una de las esferas con un material emisivo amarillo para ver el efecto de este tipo de materiales.

Figura 66. Escena extra 2.

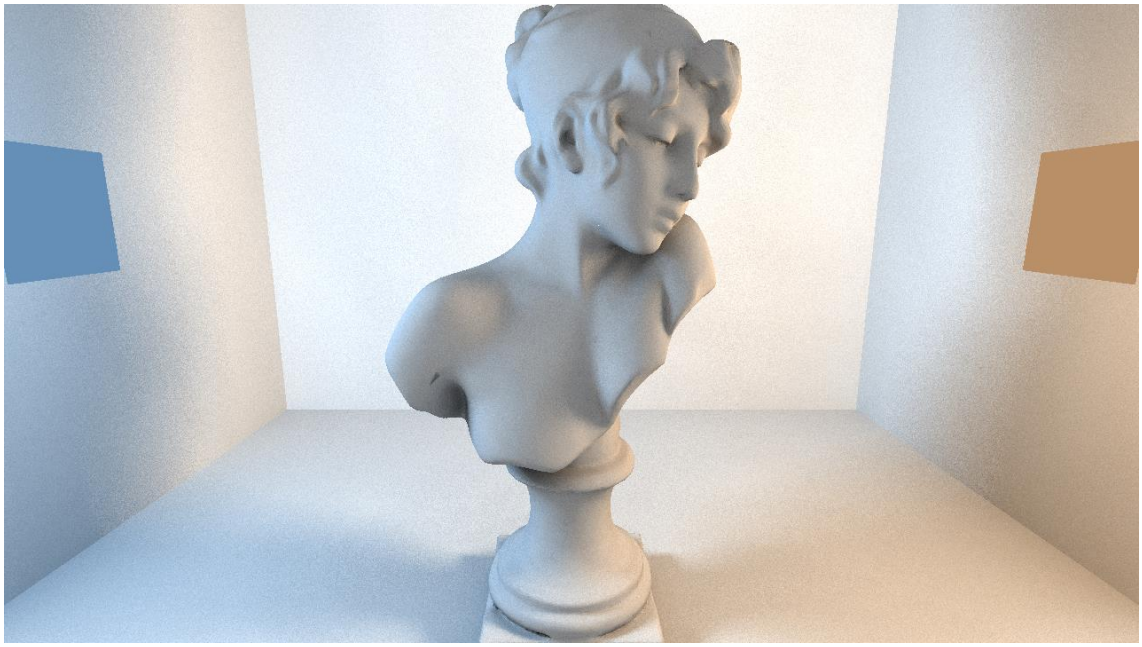


Figura 67. Escena extra 3.

Figura 68. Escena extra 4.

Se han creado dos escenas formadas por una malla (de 56321 polígonos en la primera imagen y de 79978 polígonos en la segunda), 6 cubos y dos luces de área de 25 puntos cada una. Calidad de iluminación indirecta = 5, sin reflejos. Los resultados han sido:

	Primera imagen	Segunda imagen
Resolución:	1280 x 720 (sin antialiasing)	1280 x 720 (sin antialiasing)
Rayos por píxel:	2601	2601
Crear escena:	8.6641 segundos	16.6365 segundos
Copiar escena:	0.4545 segundos	0.9944 segundos
Ejecución:	1265 segundos	1785 segundos

7. Conclusiones

El objetivo de este proyecto era el de desarrollar un software para obtener imágenes realistas y se ha cumplido con los objetivos marcados obteniendo un buen resultado final.

Todas las características requeridas del trazador se han conseguido implementar satisfactoriamente, a excepción de la transmisión, donde solo se tienen en cuenta unos pocos casos y no está disponible para todos los tipos de objetos. Aun así, la cantidad de elementos y cálculos implementados son considerables.

Se ha implementado el trazador en CUDA aprovechando todas las capacidades que el hardware disponible y el tiempo destinado al proyecto han permitido. Se ha tenido que decidir y establecer un tamaño de pila de ejecución para cada hilo de ejecución relativamente bajo, sacrificando recursión por tamaño de imagen.

Los resultados obtenidos, en tiempo de ejecución, son buenos si se tiene en cuenta el número de rayos trazados total en la escena (por ejemplo la escena extra 3 tarda 21 minutos en renderizar pero se trazan casi 2400 millones de rayos). Por otra parte los resultados a nivel de calidad de imagen son muy buenos.

8. Ampliaciones futuras

En este punto se proponen futuras ampliaciones para este proyecto desde varios puntos.

8.1 CUDA

Las posibles ampliaciones que se realicen sobre CUDA tendrán como propósito el de mejorar el rendimiento de la aplicación actual.

- Utilizar paralelismo dinámico. Esto es lanzar un kernel dentro de otro kernel y podría utilizarse por ejemplo en el proceso de trazar rayos para la iluminación indirecta. Esta característica no se ha implementado en el proyecto porque el hardware del que se disponía no lo soportaba.

- Aplicar optimizaciones en el trazado de rayos y/o optimizaciones relacionadas con CUDA como accesos más eficientes a memoria.

- Adaptar este proyecto para su ejecución en clústeres de GPUs.

8.2 Trazado

Las ampliaciones en el trazado de rayos consistirán en el cálculo de nuevos efectos.

- De la forma similar que se han implementado los reflejos con ángulo se podría implementar la transmisión con ángulo sólido para, por ejemplo, simular objetos transparentes con la superficie rugosa, de manera que dispersa la luz y las refracciones no son nítidas.

- En la cámara se podría implementar profundidad de campo. Bastaría con modificar la posición de la cámara en patrones concretos manteniendo el punto de interés en el mismo sitio. El resultado sería el del enfoque de una cámara, los objetos más alejados del foco se ven más borrosos.

- Los dos puntos anteriores se podrían ampliar más aun añadiendo aberración cromática separando cada rayo en 3 (R, G, B).

- Se podría añadir el factor de *fresnel* a los reflejos para que la cantidad de reflejo en una superficie dependa también del punto de vista del observador.

- Se podría añadir el cálculo de *Subsurface Scattering* para objetos translucidos y conseguir efectos como la cera o la piel.

8.3 Otros

Se podría realizar un editor de escenas, al estilo de programas tipo 3dsMax, Maya o Blender pero más simple, que generase los ficheros de escena de una manera más visual y amigable de cara al usuario.

9. Referencias

9.1 Libros

[1]Andrew S. Glassner. *An Introduction to Ray Tracing*, Academic Press Inc., 1990.

[2] Andrew S. Glassner. *Graphics Gems*, Academic Press Inc., 1990.

[3] James Arvo. *Grapics Gems II*, Academic Press Inc., 1991.

[4] Paul S. Jeckbert. *Graphics Gems IV*, Academic Press Inc., 1994.

[5] Alan W. Paeth. *Graphics Gems V*, Academic Press Inc., 1995.

[6] Philip J. Schneider, David H. Eberly. *Geometric Tools for Computer Graphics*, Morgan Kaufmann Publishers, 2003.

[7] Wolfgang F. Engel. *Shader X²: Introductions & Tutorials with DirectX9*, Wordware Publishing, Inc., 2004.

[8] David Wolff. *OpenGL 4.0 Shading Language Cookbook*, Packt Publishing, 2011.

9.2 Referencias web

[9] Curso online "Intro to parallel programming" de Udacity. https://www.udacity.com

[10] CUDA Toolkit Documentation, NVIDIA. http://docs.nvidia.com/cuda/cuda-c-programming-guide

ANEXO A: Manual de usuario

Requisitos

Para ejecutar este software se requiere:

- Una GPU NVIDIA con una versión de *CUDA Compute Capability* >= 2.0. Esta versión depende únicamente del hardware, https://developer.nvidia.com/cuda-gpus.
- Tener instalados los drivers de CUDA https://developer.nvidia.com/cuda-downloads.

Elaboración de escenas

Hay que escribir un fichero de texto plano (sin un formato de archivo específico, puede ser .txt u otro) siguiendo las líneas descritas en la sección 4 (los ficheros de ejemplo del anexo D son un buen ejemplo también).

Ejecución

Para ejecutar el fichero lo primero que se ha de hacer es abrir una consola en el directorio donde esté el ejecutable (pulsando Shift + botón derecho del ratón aparecerá una opción en el menú contextual "Abrir ventana de comandos aquí").

A continuación se ha lanzar el ejecutable con unos argumentos. Tenemos dos opciones.

> cudaRayTracer.exe "ruta del fichero escena/fichero escena"

Se escribe el nombre del ejecutable seguido de la ruta correspondiente al fichero escena. Se realizará el trazado de rayos, la imagen resultado se guardará en un fichero por defecto "out.bmp" y se mostrará en una ventana.

> cudaRayTracer.exe "fichero escena" "ruta de la imagen/imagen resultado.formato"

También se puede lanzar el ejecutable especificando donde se quiere guardar la imagen resultado y como. Se escribe el nombre del ejecutable seguido de la ruta correspondiente al fichero escena y de la ruta correspondiente al archivo donde se guardará la imagen resultado. Se realizará el trazado de rayos, la imagen resultado se guardará en el fichero especificado y se mostrará en una ventana.

Hay que tener en cuenta que el formato especificado para guardar la imagen influye en la calidad de la imagen guardada. Así mismo si el formato del archivo de salida no se corresponde con ningún formato de imagen válido el guardado fallará, resultado un archivo vacío.

Consideraciones

El trazador de rayos utiliza toda la capacidad posible de la GPU, lo que implica que la salida por pantalla podría bloquearse durante el proceso.

Errores

Se pueden producir una serie de errores al ejecutar el programa.

- Errores de lectura del fichero escena. Se indica la línea y el motivo, principalmente debidos a una incorrecta definición de los elementos.
- Errores de lectura del fichero OBJ. Se producen cuando un fichero OBJ no contiene coordenadas de textura o normales.
- Out of memory. Indica que el tamaño de la imagen resultante es excesivo. Básicamente no tiene suficiente memoria para asignar a todos los threads.

ANEXO B: Fichero OBJ

En este proyecto las mallas poligonales se han importado desde ficheros de tipo OBJ. En este anexo se comentará brevemente la estructura de estos ficheros.

Los ficheros OBJ son ficheros de texto plano que contienen información sobre mallas poligonales. Los diferentes elementos que extraeremos de estos ficheros, así como su representación en el mismo se describen a continuación.

Comentarios

Los comentarios que aparecen en el fichero abarcan únicamente una línea. Se definen mediante el carácter "#" seguido de un espacio y a continuación el texto del comentario.

```
# File Created: 29.01.2014 22:12:34
```

Posiciones de los vértices

Cada posición de vértice se especifica en una línea diferente. Estas posiciones vienen dadas mediante el carácter "v" seguido de un espacio. A continuación se encuentran tres valores reales que representan las coordenadas x, y, z de la posición.

```
v  15.135800 68.190697 -4.227500
```

Normales de los vértices

Cada vector normal se especifica en una línea diferente. Estos vectores vienen dados por los caracteres "vn" seguidos de un espacio. A continuación se encuentran tres valores reales que representan las componentes x, y, z del vector.

```
vn 0.951577 0.082960 0.296004
```

Coordenadas de textura de los vértices

Cada coordenada de textura se especifica en una línea diferente. Estas coordenadas vienen dadas por los caracteres "vt" seguidos de un espacio. A continuación se encuentran tres valores reales que representan los valores de las coordenadas en x, y, z. Generalmente solo se utilizan las componentes x e y.

```
vt 0.679800 0.822800 0.000000
```

Grupo

El nombre del grupo viene dado por el carácter "g" seguido de un espacio y a continuación el nombre.

```
g 3dHead
```

Caras

Cada cara se especifica en una línea del fichero diferente. Una cara viene dada en primer lugar por el carácter "f" seguido de un espacio. A continuación se especifican los componentes de cada vértice (posición, coordenada de textura, normal) mediante índices que referencian estos elementos dentro de su conjunto.

Cada vértice se especifica de la siguiente forma.

posición/coordenada de textura/normal

Cada cara tendrá un número de vértices mayor o igual a 3.

f 1/1/1 2/2/2 3/3/3

Consideraciones

- El número de decimales (precisión) de los valores en los elementos es fijo en cada fichero y se especifica al exportar.
- Hay que tener en cuenta que los índices de los vértices de las caras empiezan por 1 y no por 0.

ANEXO C: Editor CSG y fichero

En este punto se va a comentar la herramienta con la que se generan ficheros de tipo CSG.

El programa CSGEditor permite crear y editar árboles binarios CSG. El programa también permite el guardado, cargado y exportado de los árboles.

Este programa requiere de tener instalado Qt http://qt-project.org/downloads.

1. Interfaz

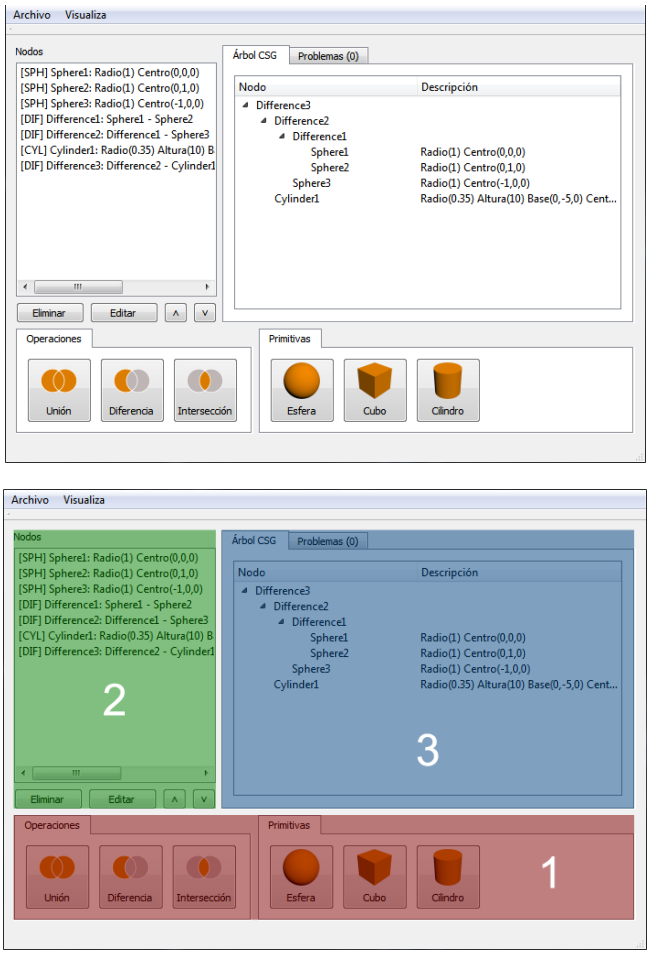


Figura 69. Ventana principal del editor CSG.

En la interfaz se diferencian 3 zonas marcadas en la figura.

1- Creación de primitivas y operaciones. Para crear primitivas y operaciones basta con pulsar el botón y definir sus propiedades.

2- Lista de nodos y botones de acciones sobre el nodo. Una vez seleccionado un nodo de la lista éste se puede editar, eliminar o reordenarlo dentro de la lista.

3- Árbol y problemas. Corresponde a un panel de dos pestañas. En la primera pestaña se representan las relaciones de los nodos en forma de árbol y en la segunda hay una lista de problemas asociados al árbol, como por ejemplo nodos operaciones sin hijos asignados.

2. Creación y edición

2.1 Primitivas

Las primitivas se crean y editan mediante unos diálogos.

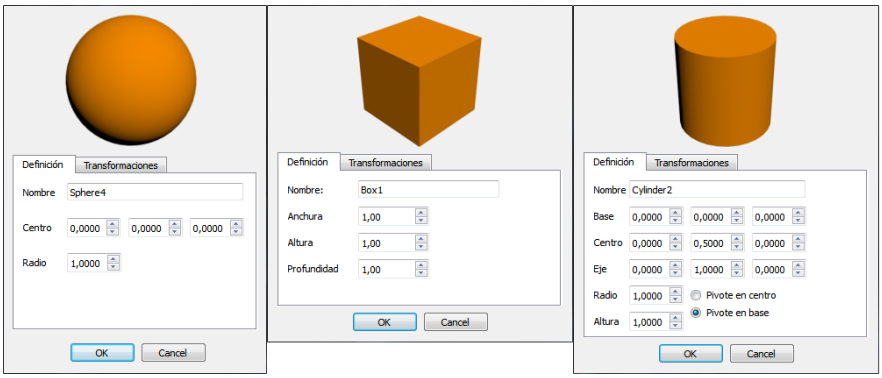


Figura 70. Diálogo de creación/edición para todas las primitivas.

Todas las primitivas se definen mediante unos parámetros (al igual que las primitivas de la escena) y tienen asociada una lista de transformaciones.

2.2 Operaciones

Las operaciones se crean y editan mediante su correspondiente diálogo.

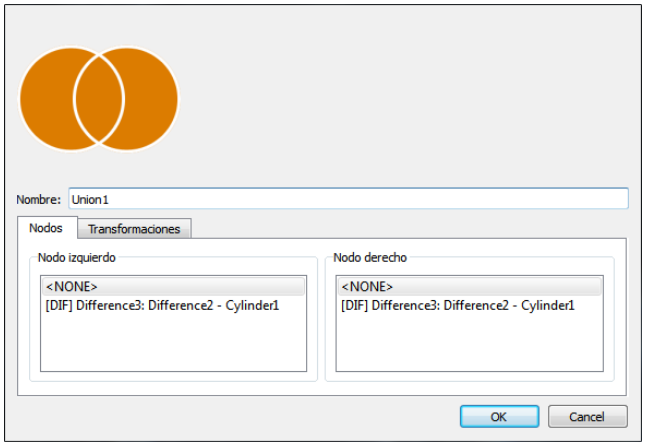


Figura 71. Diálogo de creación/edición de operaciones.

En este diálogo se muestran para una operación una lista de nodos disponibles que puede añadir como hijo izquierdo o como hijo derecho.

Las operaciones también disponen de una lista de transformaciones asociada.

2.3 Transformaciones

Dentro del diálogo correspondiente a cada elemento hay una pestaña dedicada a la lista de transformaciones asociada. Esta lista se puede editar añadiendo, eliminando, modificando o reordenando transformaciones.

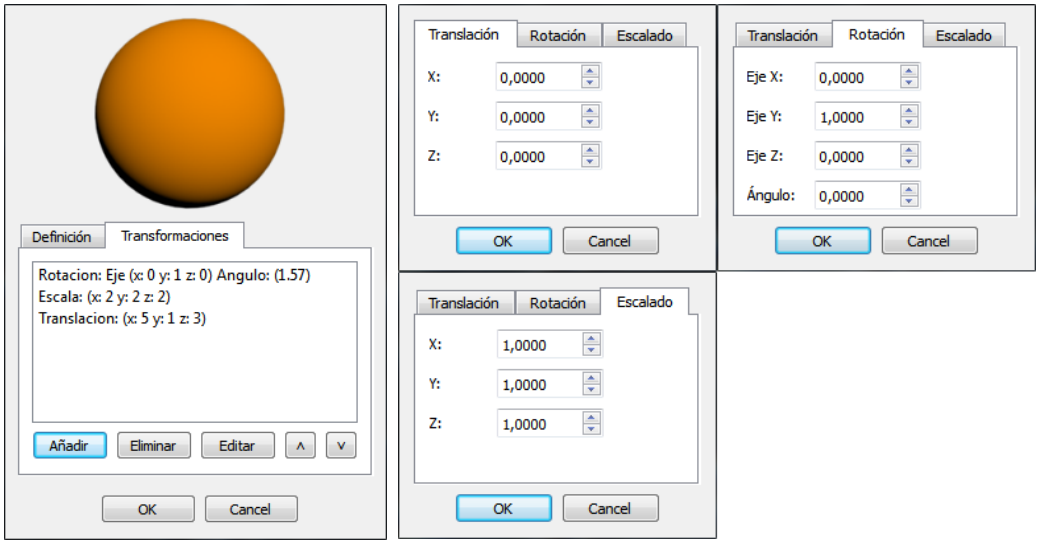


Figura 72. Lista de transformaciones en un nodo esfera y diálogos correspondientes a las transformaciones.

Las transformaciones se crean mediante el diálogo de la figura. Según la pestaña que se seleccione se creará un tipo de transformación diferente.

3. Guardar/Cargar/Exportar

En la ventana principal del programa se incluye un menú con varias opciones.

Las opciones de guardar y abrir fichero sirven para guardar el árbol en el formato de fichero del editor.

La opción de exportar es la que se debe utilizar para que este árbol se pueda utilizar en una escena.

4. Formato de fichero CSG

El formato del fichero exportado CSG es similar al fichero escena y los elementos del fichero tienen la misma estructura que la descrita en la sección 4.2.

Las primitivas se definen exactamente igual que en la sección 4.7.

Los nodos operación tienen los siguientes identificadores: Union, Difference y Intersection, y se definen con los siguientes parámetros.

- Tipo izquierdo (*LeftType*). El tipo de nodo del hijo izquierdo.
- Índice izquierdo (*LeftIndex*). Índice del nodo izquierdo en la lista de nodos de ese tipo.
- Tipo derecho (*RightType*). El tipo de nodo del hijo derecho.
- Índice derecho (*RightIndex*). Índice del nodo derecho en la lista de nodos de ese tipo.

ANEXO D: Escenas de ejemplo

Escena 1

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        0
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name                     matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name                     matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name                     matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Sphere
{
        Material matWhite
        Center   0.0 1.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 3.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 5.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 7.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 9.0 0.0
        Radius   1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

AreaLight
{
        Color 1 1 1
        Position 3.0 5.0 8.0
        Direction 0.0 0.0 -1.0
        RollAngle 0.0
        Width    2
        Height   2
        Subdivision 10 10
}

# PointLight
# {
#       Color 0.9 0.9 0.9
#       Position 3.0 5.0 8.0
# }
```

Escena 2

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        0
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name                     matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name
        matWhiteSpecular
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.9 0.9 0.9
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}


Material
{
        Name                     matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name                     matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

Mesh
{
        Material matWhiteSpecular
        Source "Resources/bunny.obj"
        Scale 0.7 0.7 0.7
        Translation 1 0 -2
}

AreaLight
{
        Color 1 1 1
        Position 0.0 9.5 3.0
        Direction 0.0 0.0 -1.0
        RollAngle 0.0
        Width   2
        Height  2
        Subdivision 10 10
}

# PointLight
# {
#       Color 0.9 0.9 0.9
#       Position 0.0 9.5 3.0
# }
```

Escena 3

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        3
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name             matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name                     matReflective
        DiffuseColor     0.0 0.0 0.0
        SpecularColor    1.0 1.0 1.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.1
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
        ReflectionAmount 1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

CSG
{
        Material matWhite
        Source "Resources/csgExampleExport.csg"
}

PointLight
{
        Color 1 1 1
        Position 3.0 3.0 8.0
}
```

Escena 4

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        3
        ReflectionSteps          2
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name             matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matReflective
        DiffuseColor     0.0 0.0 0.0
        SpecularColor    1.0 1.0 1.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.1
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
        ReflectionAmount 1.0
}

Sphere
{
        Material matReflective
        Center   0.0 1.0 0.0
        Radius   1.0
}

Sphere
{
        Material matReflective
        Center   0.0 3.0 0.0
        Radius   1.0
}

Sphere
{
        Material matReflective
        Center   0.0 5.0 0.0
        Radius   1.0
}

Sphere
{
        Material matReflective
        Center   0.0 7.0 0.0
        Radius   1.0
}

Sphere
{
        Material matReflective
        Center   0.0 9.0 0.0
        Radius   1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

AreaLight
{
        Color 1 1 1
        Position 3.0 5.0 8.0
        Direction 0.0 0.0 -1.0
        RollAngle 0.0
        Width    2
        Height   2
        Subdivision 10 10
}
```

Escena 5

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        3
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name             matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matWhiteSpecular
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.9 0.9 0.9
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}


Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matReflective
        DiffuseColor     0.0 0.0 0.0
        SpecularColor    1.0 1.0 1.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.1
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
        ReflectionAmount 1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

Mesh
{
        Material matReflective
        Source "Resources/bunny.obj"
        Scale 0.7 0.7 0.7
        Translation 1 0 -2
}


AreaLight
{
        Color 1 1 1
        Position 0.0 9.5 3.0
        Direction 0.0 -1.0 0.0
        RollAngle 0.0
        Width   2
        Height  2
        Subdivision 10 10
}
```

Escena 6

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        5
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV -5 5 -15
        POI 0 0 0
        FOV 45
}


Material
{
        Name             mat1
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matReflective
        DiffuseColor     0.0 0.0 0.0
        SpecularColor    1.0 1.0 1.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  15
        Opacity          1.0
        RefractionIndex  1.0
        ReflectionAmount 0.75
}

Material
{
        Name             matRed
        DiffuseColor     0.9 0.0 0.0
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.0 0.9 0.0
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

# Bottom box
Box
{
        Material matReflective
        Scale 20 1 20
        Translation 0 -0.5 0
}

# Left Box
Box
{
        Material matGreen
        Scale 1 20 20
        Translation 5 5 0
}

# Back box
Box
{
        Material matRed
        Scale 20 20 1
        Translation 0 5 5
}

Sphere
{
        Material matReflective
        Center 0 1 3
        Radius 1
}

Sphere
{
        Material mat1
        Center 0 3 3
        Radius 1
}

Sphere
{
        Material mat1
        Center 0 5 3
        Radius 1
}

Sphere
{
        Material mat1
        Center 0 7 3
        Radius 1
}

Sphere
{
        Material mat1
        Center 0 9 3
        Radius 1
}

AreaLight
{
        Color 1 1 1
        Position 0.0 10.0 -10.0
        Direction 0.0 -1.0 1.0
        RollAngle 0.0
        Width   2
        Height  2
        Subdivision 10 10
}
```

Escena 7

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        0
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name             matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matRefractive
        DiffuseColor     1.0 1.0 1.0
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        ReflectionAmount 0.0
        Opacity          0.1
        RefractionIndex  1.5
}

Sphere
{
        Material matRefractive
        Center -2 5 2
        Radius 2
}

Sphere
{
        Material matRefractive
        Center 4 9 -2
        Radius 1
}

Sphere
{
        Material matRefractive
        Center 4 7 -2
        Radius 1
}

Sphere
{
        Material matRefractive
        Center 4 5 -2
        Radius 1
}

Sphere
{
        Material matRefractive
        Center 4 3 -2
        Radius 1
}

Sphere
{
        Material matRefractive
        Center 4 1 -2
        Radius 1
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

AreaLight
{
        Color 1 1 1
        Position 0.0 9.5 0.0
        Direction 0.0 -1.0 0.0
        RollAngle 0.0
        Width    2
        Height   2
        Subdivision 10 10
}
```

Escena 8

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        0
        ReflectionSteps          1
        IndirectQuality          0
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 20
}

Texture
{
        Name     headDiffuse
        Source   "Resources/diffuse2048.tga"
}

Texture
{
        Name     headNormal
        Source   "Resources/normal2048.tga"
}

Material
{
        Name             matHead
        DiffuseTexture   headDiffuse
        DiffuseColor     1.0 1.0 1.0
        NormalTexture    headNormal
        Roughness        0.5
        ReflectionAngle  0.0
        Opacity          1.0
        ReflectionAmount 0.0
}

Material
{
        Name             mat1
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

# Bottom box
Box
{
        Material mat1
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material mat1
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material mat1
        Scale 10 10 1
        Translation 0 5 -5.5
}

Mesh
{
        Material matHead
        Source "Resources/3dHead.obj"
        Scale 0.1 0.1 0.1
        Translation 0 1 0
}

AreaLight
{
        Color 0.9 0.9 0.9
        Position 0.0 7.0 5.0
        Direction 0.0 0.0 -1.0
        RollAngle 0.0
        Width   2
        Height  2
        Subdivision 10 10
}
```

Escena 9

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        0
        ReflectionSteps          1
        IndirectQuality          3
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name             matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Sphere
{
        Material matWhite
        Center   0.0 1.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 3.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 5.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 7.0 0.0
        Radius   1.0
}

Sphere
{
        Material matWhite
        Center   0.0 9.0 0.0
        Radius   1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

AreaLight
{
        Color 1 1 1
        Position 3.0 5.0 8.0
        Direction 0.0 0.0 -1.0
        RollAngle 0.0
        Width    2
        Height   2
        Subdivision 5 5
}

#PointLight
# {
#       Color 0.9 0.9 0.9
#       Position 3.0 5.0 8.0
#}
```

Escena 10

```
SceneSettings
{
        BackgroundColor          0.0 0.0 0.0
        RenderWidth              512
        RenderHeight             512

        ReflectionQuality        0
        ReflectionSteps          1
        IndirectQuality          3
        ColorBleeding            4
        Antialiasing             1
}

Camera
{
        POV 0 5 15
        POI 0 5 0
        FOV 45
}

Material
{
        Name             matWhite
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.0 0.0 0.0
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.9
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matWhiteSpecular
        DiffuseColor     0.9 0.9 0.9
        SpecularColor    0.9 0.9 0.9
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.2
        ReflectionAngle  0.0
        Opacity          1.0
        RefractionIndex  1.0
}


Material
{
        Name             matRed
        DiffuseColor     0.9 0.1 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

Material
{
        Name             matGreen
        DiffuseColor     0.1 0.9 0.1
        SpecularColor    0.1 0.1 0.1
        EmissiveColor    0.0 0.0 0.0
        Roughness        0.85
        ReflectionAngle  45.0
        Opacity          1.0
        RefractionIndex  1.0
}

# Bottom box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 -0.5 0
}

# Top Box
Box
{
        Material matWhite
        Scale 10 1 10
        Translation 0 10.5 0
}

# Left Box
Box
{
        Material matRed
        Scale 1 10 10
        Translation -5.5 5 0
}

# Right Box
Box
{
        Material matGreen
        Scale 1 10 10
        Translation 5.5 5 0
}

# Back box
Box
{
        Material matWhite
        Scale 10 10 1
        Translation 0 5 -5.5
}

Mesh
{
        Material matWhiteSpecular
        Source "Resources/bunny.obj"
        Scale 0.7 0.7 0.7
        Translation 1 0 -2
}


PointLight
{
        Color 1 1 1
        Position 0.0 9.5 3.0
}
```