

- TP 1 -

Path Tracer

Informática Gráfica



755584 Sergio García
758803 Daniel González

Índice

Introducción	3
1. Manipulación de imágenes	4
1.1 Formatos: PPM y BMP (opcional)	4
1.2 Tone mappers	4
1.2.1 Clamp	4
1.2.2 Equalization	5
1.2.3 Clamp && Equalization	6
1.2.4 Curva Gamma	7
1.2.5 Clamp && Curva Gamma	8
1.2.1 Reinhard [RSSF02] (opcional)	9
2. Ecuación de render	10
3. Convergencia	12
3.1 Rayos por pixel	12
3.2 Materiales	13
3.3 Tipos de luz	14
4. Efectos de iluminación	15
4.1 Sombras	15
4.2 Color bleeding	16
4.3 Cáusticas	17
5. + Opcionales	18
5.1 Triangulos + PLY	18
5.2 Paralelización	19
6. Distribución de la carga de trabajo	20
Bibliografía	21

Introducción

A través del tiempo hemos visto cómo han ido evolucionando los algoritmos de renderizado de imágenes. Este trabajo parte sobre una pequeña base de un ray tracing que modelaremos finalmente hasta llegar a un algoritmo de path tracer completo.

La mecánica para el desarrollo del algoritmo ha sido simple: primero comenzamos con la manipulación de imágenes, rango de color y su almacenamiento en memoria. Luego completamos un pequeño ray tracing que solo interseccionaba con el primer objeto que salía desde la cámara, que finalmente terminó por convertirse en el algoritmo del que hablaremos en este trabajo.

Podremos observar como la ecuación de render se adapta en nuestro algoritmo, desde la ecuación teoría hasta la que actualmente se usa pasando por la de ray tracing. También veremos cómo converge el algoritmo implementado si editamos las variables de entrada como los paths per pixel o dependiendo del material. Y finalmente hablaremos de algunos efectos de luz que podemos encontrarnos en la ejecución del algoritmo.

1. Manipulación de imágenes

Antes de poder realizar cualquier algoritmo de renderizado, primero tenemos que adquirir ciertos conocimientos relacionados con las propiedades de los objetos finales de estos algoritmos: las imágenes. Lo más sencillo de todo, será su almacenamiento. En un primer momento, se desarrolla el formato PPM, un formato sencillo propuesto por el profesorado que luego ampliamos con BMP. Tras su almacenamiento, veremos cómo se pueden manipular esas imágenes a través de tone mappers, los cuales sirven para aplicar correcciones en los rangos de luminancia.

1.1 Formatos: PPM y BMP (opcional)

En la segunda práctica de la asignatura nos apareció la necesidad de visualizar las imágenes que generamos con nuestro algoritmo, desarrollamos la funcionalidad de exportar nuestra imagen almacenada en memoria en tuplas RGB a archivo PPM. El formato de archivo PPM nos permite almacenar imágenes HDR y LDR, no utiliza compresión y se almacena en formato texto. Adicionalmente, decidimos añadir a nuestro programa la posibilidad de exportar nuestra imagen a otro formato LDR, en concreto BMP. El formato de archivo BMP incluye una cabecera con información sobre el mapa de bits, es compatible con más programas de lectura de imagen tanto en Windows como Linux, no permite almacenar imágenes HDR, pero es más rápido y consigue archivos más ligeros gracias a 4 niveles de compresión RLE.

Estos son los tiempos y los tamaños conseguidos al exportar una imagen de resolución 2000x2000 en LDR y sin comprimir, tanto en BMP como en PPM.

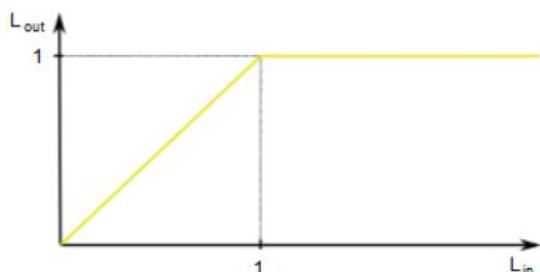
Export BMP:	0.475 segundos	 render.bmp	11.719 KB
Export PPM:	3.863 segundos	 render.ppm	61.699 KB

Figura 1: Comparación tiempos exportando y tamaño final

1.2 Tone mappers

Cuando se almacenan imágenes en algún formato LDR (Low Dynamic Range) puede que se pierda información, lo que implica que en ciertas zonas de la imagen resulten demasiado brillantes hasta el punto de aparecer todo blanco (sobreexposición) o que sean demasiado oscuras que no pueda verse nada (subexposición). Para poder corregir estos valores, lo que hacemos es aplicarles una serie de funciones que permiten pasar de un HDR (High Dynamic Range) a un LDR corregido. A estas funciones les llamaremos tone mappers.

1.2.1 Clamp



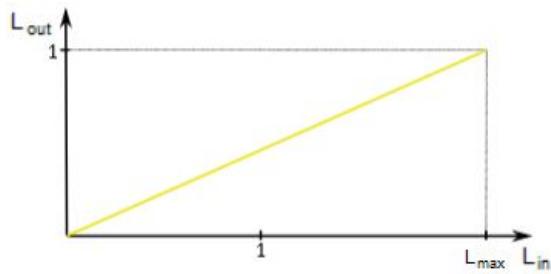
Una de las correcciones más sencillas que podemos hacer es delimitar el máximo valor del rango dinámico. Todos aquellos valores de la imagen que sean superiores a uno se reemplazarán por 1, de tal forma que nos queda una imagen como la Fig. 2, donde las zonas más iluminadas se quedan más blancas:

Figura 2: Función clamp



Figura 3: Imagen con TM clamp

1.2.2 Equalization



Con este tone mapper lo que se pretende es normalizar el rango dinámico de la imagen. En otras palabras, se divide por el máximo valor de la imagen para que el resultado tenga valores sólo entre 0 y 1. Esto implica que si hay algún valor extremadamente alto, el resto de valores disminuyen en proporción y por tanto una imagen más oscura:

Figura 4: Función equalization

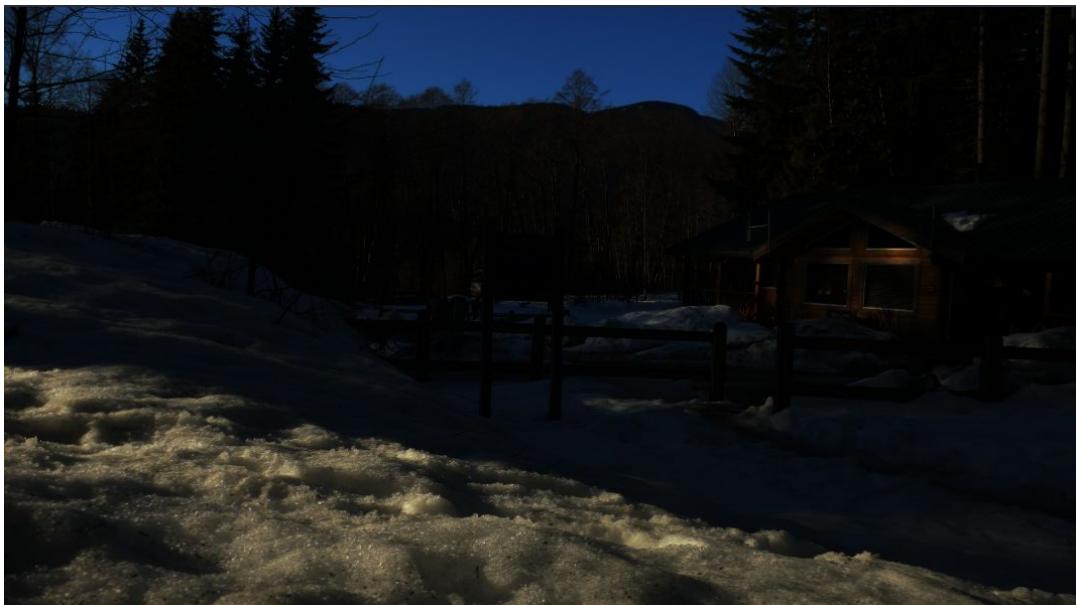


Figura 5: Imagen con TM equalization

1.2.3 Clamp && Equalization

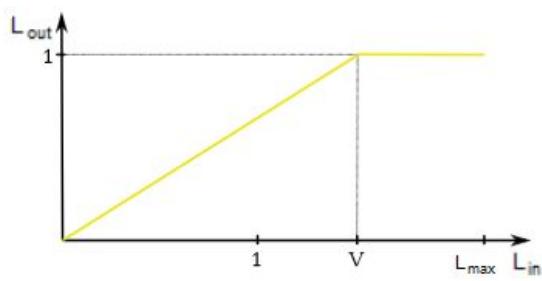


Figura 6: función clamp-equalization

Este es un conjunto de los dos tone mappers mencionados anteriormente, pero con pequeñas diferencias. Ahora el valor que se clamaea no es uno, sino el que nosotros queramos y luego se normaliza en función de ese valor que le hemos asignado. Cuanto menor sea el valor de clamp, más información se pierde y más brillante (blanco) se verá.

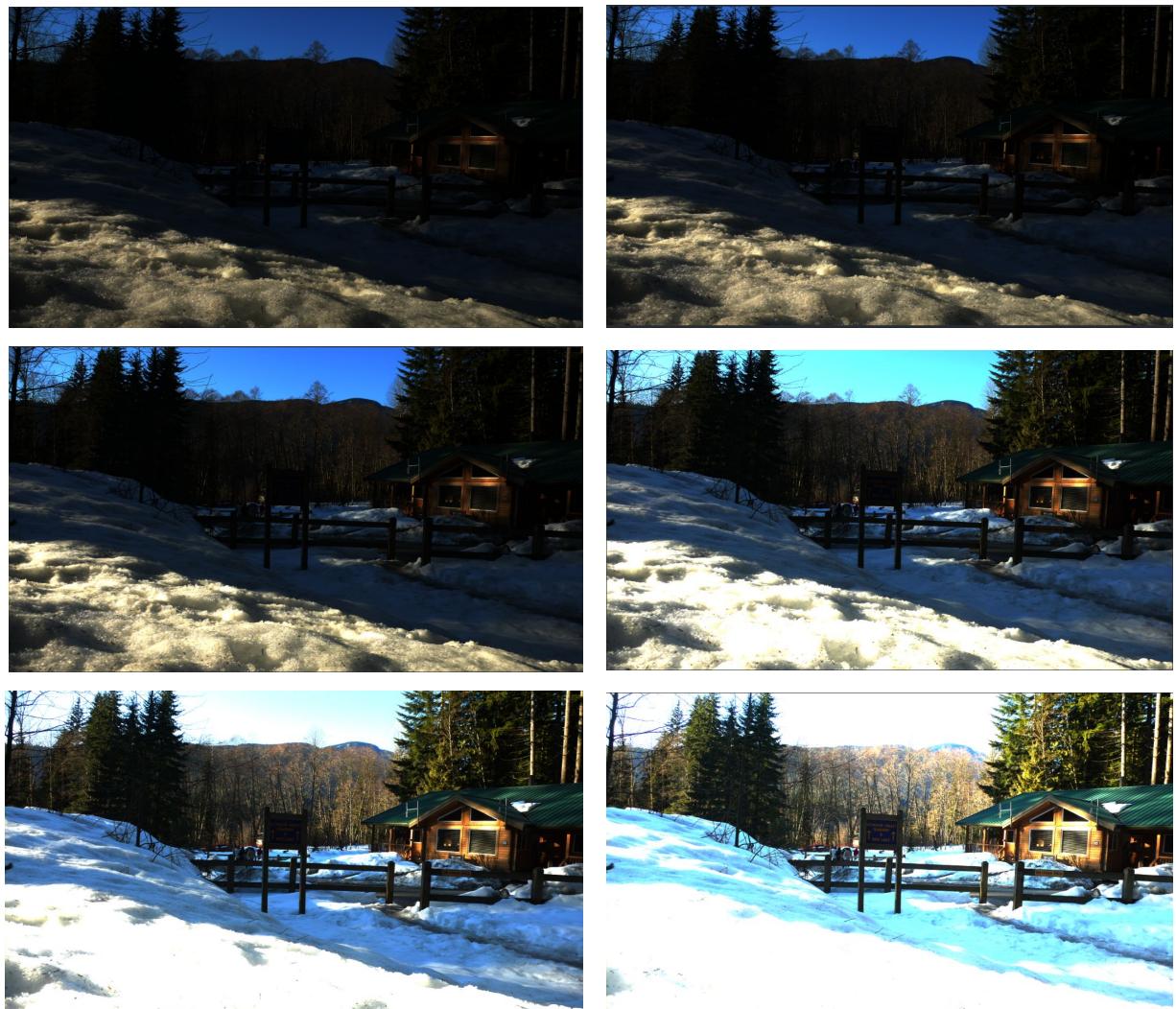


Figura 7: Imagen con TM clamp-equalization con $clamp = max_value * \{ 0.8, 0.6, 0.4, 0.2, 0.1 \text{ y } 0.05 \}$, donde max_value es el valor más alto de las tuplas RGB de la imagen(R, G o B), decrecimiento de clamp de izq a dcha, de arriba a abajo.

1.2.4 Curva Gamma

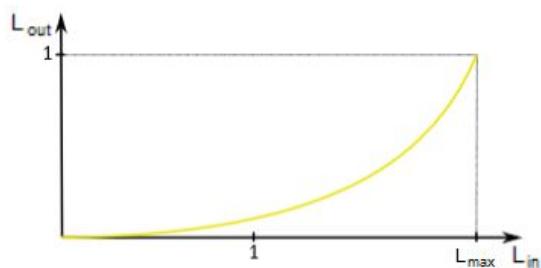


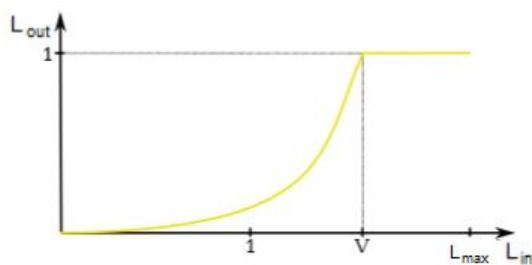
Figura 8: Función curva gamma

La corrección gamma se basa en la ley de potencias. Para calcular la curva, utilizamos la función $L_{out} = \text{CTE} * (L_{in}^{\gamma})$, aunque por defecto dejaremos el valor constante a 1. Para poder realizar esta operación antes hay que normalizar (equalization) (si $\gamma = 1$ es como si solo normalizáramos).



Figura 9: Imagen con TM curva gamma con el valor de gammas = {0.2, 0.4, 0.6, 0.8, 1 y 1.2}
En el caso de gamma = 1 la imagen es igual a la normalizada (1.2.2 Fig. 5)

1.2.5 Clamp & Curva Gamma



Este tone mapper combina la curva gamma con el límite de clamp. Esto puede ser útil cuando la imagen llega a tener pocos valores muy altos que sean despreciables, para que la normalización y luego la curva tengan mejor apariencia.

Figura 10: Función clamp-curva gamma



Figura 11: Imagen con TM clamp-curva gamma con valor clamp = max_value*0.2 y
gamma = { 0.8, 0.6 y 0.4 }

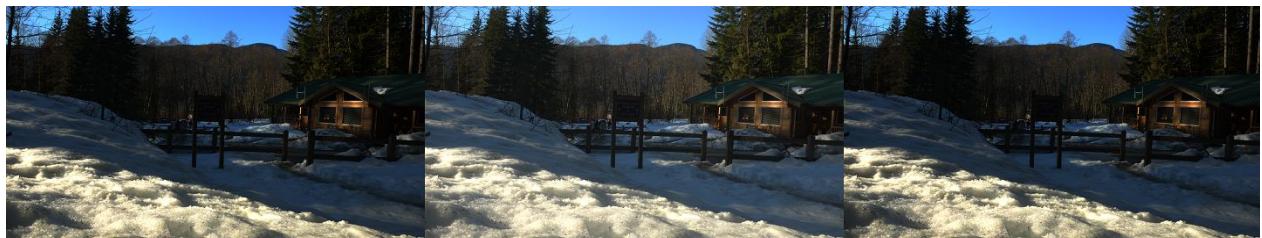


Figura 12: Imagen con TM clamp-curva gamma con valor clamp = max_value*0.4 y
gamma = { 0.8, 0.6 y 0.4 }

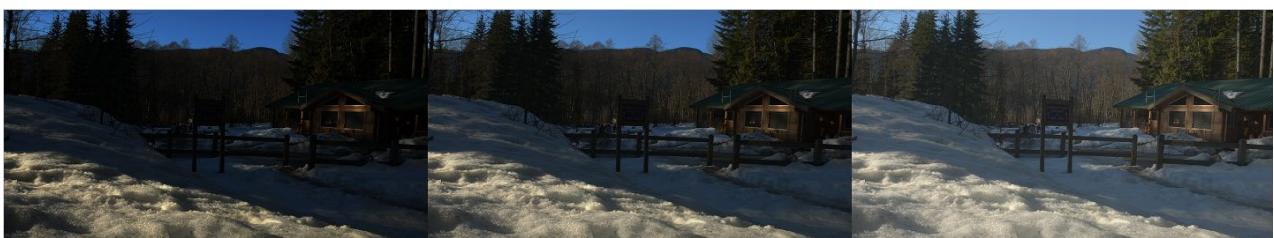
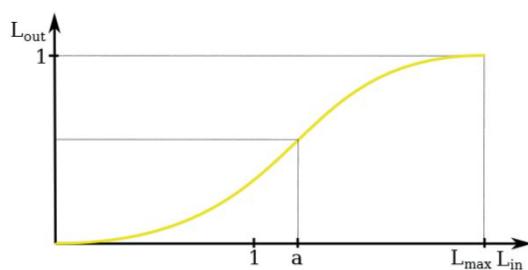


Figura 13: Imagen con TM clamp-curva gamma con valor clamp = max_value*0.8 y
gamma = { 0.8, 0.6 y 0.4 }

1.2.1 Reinhard [RSSF02] (opcional)



Como tone mapper opcional, se ha decidido implementar una aproximación (operador simple) del tone mapper explicado en el doc [RSSF02] [2] [3].

Dicho operador se puede definir como $L'(x,y) = L(x,y)*k / (1 + L(x,y)*k)$, dejando k como variable. Por defecto, la k tiene el valor de 1.

Figura 14: Función clamp-curva gamma



Figura 15: Imagen con TM reinhard con valor $k = \{ 0.5, 0.75, 1, 1.25, 2 \text{ y } 3 \}$, incremento de k de izq a dcha, de arriba a abajo.

2. Ecuación de render

En lo referente a la teoría de este trabajo, lo más importante es entender cómo funciona la ecuación de render:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \int_{\Omega} L_i(x, \vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) |n \cdot \vec{\omega}_i| d\vec{\omega}_i$$

En ray tracing generábamos n rayos en el punto de intersección con la intención de intentar adquirir la cantidad de luminancia más precisa posible, con lo que la ecuación de render obtenía este aspecto:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \frac{1}{N} \sum_1^N \frac{L_i(x, \vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) |n \cdot \vec{\omega}_i|}{p(\vec{\omega}_i)}$$

Recordamos que en el path tracer solo se genera un rayo cada vez que uno anterior intersecciona con algo de la escena y por eso la luz incidente en un punto es la que estimamos que vendrá por el rayo que generamos:

$$L_o(x, \vec{\omega}_o) = L_e(x, \vec{\omega}_o) + \frac{L_i(x, \vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) |n \cdot \vec{\omega}_i|}{p(\vec{\omega}_i)}$$

Si hacemos varias iteraciones y las juntamos en una única operación obtendremos lo siguiente:

$$\begin{aligned} L_o(x, \vec{\omega}_o) &\approx \frac{L_{i1}(x, \vec{\omega}_i) f_r(x, \vec{\omega}_i, \vec{\omega}_o) |n \cdot \vec{\omega}_i|}{p(\vec{\omega}_i)} \\ L_{i1}(x, \vec{\omega}_i) &= L_{o1}(x_1, \vec{\omega}_{o1}) \approx \frac{L_{i2}(x_1, \vec{\omega}_{i1}) f_r(x_1, \vec{\omega}_{i1}, \vec{\omega}_{o1}) |n_1 \cdot \vec{\omega}_{i1}|}{p(\vec{\omega}_{i1})} \\ L_{i2}(x_1, \vec{\omega}_{i1}) &= L_{o2}(x_2, \vec{\omega}_{o2}) \approx \frac{L_{i3}(x_2, \vec{\omega}_{i2}) f_r(x_2, \vec{\omega}_{i2}, \vec{\omega}_{o2}) |n_2 \cdot \vec{\omega}_{i2}|}{p(\vec{\omega}_{i2})} \\ L_{i3}(x_2, \vec{\omega}_{i2}) &= L_{o3}(x_3, \vec{\omega}_{o3}) \approx L_e(x_3, \vec{\omega}_{o3}) \\ L_o(x, \vec{\omega}_o) &= L_e(x_N, \vec{\omega}_N) \prod_{j=0}^N \frac{f_r(x_j, \vec{\omega}_{ij}, \vec{\omega}_{oj}) |n_j \cdot \vec{\omega}_{ij}|}{p(\vec{\omega}_{ij})} \end{aligned}$$

Esto nos da una pista acerca de las posibles implementaciones que puede tener el algoritmo: una recursiva y otra iterativa, pero en la recursiva debido a que pueden ser escenas complicadas con muchas iteraciones no es recomendable por el peso que pueda tener las numerables pilas de activación.

Antes de seguir con el cálculo de la BSDF, falta por mencionar algo relacionado con las luces que inciden en un punto x en la iteración j, ¿y las luces puntuales?. La probabilidad de generar un rayo de forma aleatoria y que sea capaz de intersecar con una luz puntual es 0, por eso en cada iteración calculamos la contribución de cada luz puntual y cada L_i se divide en la suma de la contribución de las luces que no se pueden intersecar y la luz incidente del otro rayo generado.

Si a la contribución de esas luces le llamamos L' (L prima), d a la distancia que separa el punto de intersección de la luz y K el número de este tipo de luces, la ecuación de render final queda así:

$$L_o(x, \vec{\omega}_o) = L_e(x_N, \vec{\omega}_N) \prod_{j=0}^N \frac{f_r(x_j, \vec{\omega}_{ij}, \vec{\omega}_{oj}) |n_j \cdot \vec{\omega}_{ij}|}{p(\vec{\omega}_{ij})} + \sum_{m=0}^K \sum_{j=0}^N \prod_{z=0}^j \frac{f_r(x_z, \vec{\omega}_{iz}, \vec{\omega}_{oz}) |n_z \cdot \vec{\omega}_{iz}| L'_m(x_z)}{p(\vec{\omega}_{iz})} \frac{d_z^2}{d_z^2}$$

BSDF: Para hacer el cálculo de los materiales, se pueden los posibles comportamientos que pueda tener un objeto en una sola ecuación:

$$f_r(x, \vec{\omega}_i, \vec{\omega}_o) = \frac{k_d}{\pi} + k_s(x, \vec{\omega}_o) \frac{\delta_{\vec{\omega}_r}(\vec{\omega}_i)}{n \cdot \vec{\omega}_i} + k_t(x, \vec{\omega}_o) \frac{\delta_{\vec{\omega}_t}(\vec{\omega}_i)}{n \cdot \vec{\omega}_i}$$

Para implementar esto, se ha usado el concepto de ruleta rusa, aplicando probabilidades y que en función de un número aleatorio se elija un evento determinado en base a una probabilidad asignada. Para los materiales lambertianos y especulares (lambertiano puro, specular puro y plásticos) la probabilidad se escoge en función del máximo valor que tenga la constante k de cada evento. En caso de que la suma de ambas probabilidades sea superior a 0.9, se normaliza para dar una pequeña probabilidad del 10% mínimo a finalizar el camino por medio de la ruleta rusa. El caso de la constante k_t es diferente; cuando queremos crear un material que refracte, la probabilidad del evento lambertiano siempre será 0 y las probabilidades k_s y k_t se calcularán en función de las leyes de reflexión y Snell a través de las ecuaciones de Fresnell:

$$\rho_{||} = \frac{\eta_1 \cos \theta_i - \eta_0 \cos \theta_t}{\eta_1 \cos \theta_i + \eta_0 \cos \theta_t} \quad \text{reflected: } F_r = \frac{1}{2} (\rho_{||}^2 + \rho_{\perp}^2)$$

$$\rho_{\perp} = \frac{\eta_0 \cos \theta_i - \eta_1 \cos \theta_t}{\eta_0 \cos \theta_i + \eta_1 \cos \theta_t} \quad \text{refracted: } F_t = 1 - F_r$$

Finalmente, queda hablar de la generación de los rayos en los rebotes por la escena. Cuando el evento elegido es el difuso, se emplea el *Uniform cosine sampling*, que genera dos números aleatorios con los que se calcula un azimuth y un theta con los que se genera un vector en coordenadas locales que tendrá que ser convertido a coordenadas globales (1). Si en cambio el evento es specular, el rayo generado se calcula con la ecuación de la ley de reflexión (2), y si por último el evento es la refracción, usamos la fórmula de snell (3):

$$(1) \theta_i = \arccos \sqrt{1 - \xi_\theta}, \quad \varphi_i = 2\pi \xi_\varphi \quad \vec{\omega}_i = M \begin{pmatrix} \sin \theta_i \cdot \cos \varphi_i \\ \sin \theta_i \cdot \sin \varphi_i \\ \cos \theta_i \end{pmatrix}$$

$$(2) \vec{\omega}_r = 2n(n \cdot \vec{\omega}_i) - \vec{\omega}_i \quad (3) \eta_1 \sin \theta_1 = \eta_2 \sin \theta_2$$

3. Convergencia

3.1 Rayos por pixel

Vamos a analizar cómo converge la imagen generada en función de los rayos por pixel, para ello creamos una escena con una luz de área y diferentes materiales, la cual va a ser generada con una resolución 1000x1000 cambiando el parámetro rayos por píxel, en concreto generamos con los valores 5, 10, 20, 50, 100, 200, 500, 1000, 2000 y 5000.

En la siguiente figura observamos los resultados obtenidos, con valores menores a 50 la imagen tiene mucho ruido, se ven pixeles negros distribuidos uniformemente por la imagen.

Conforme aumentamos el valor de los rayos por pixel el ruido va disminuyendo, y la imagen va mejorando de calidad progresivamente. La mejora cada vez es menor, pero se aprecia más en las zonas sombreadas o más oscuras. La imagen generada con 1000 rayos por pixel no tiene apenas ruido y las generadas con un valor mayor a 1000 no sufren apenas cambios por lo que el valor 1000 es el punto de convergencia de nuestra escena.

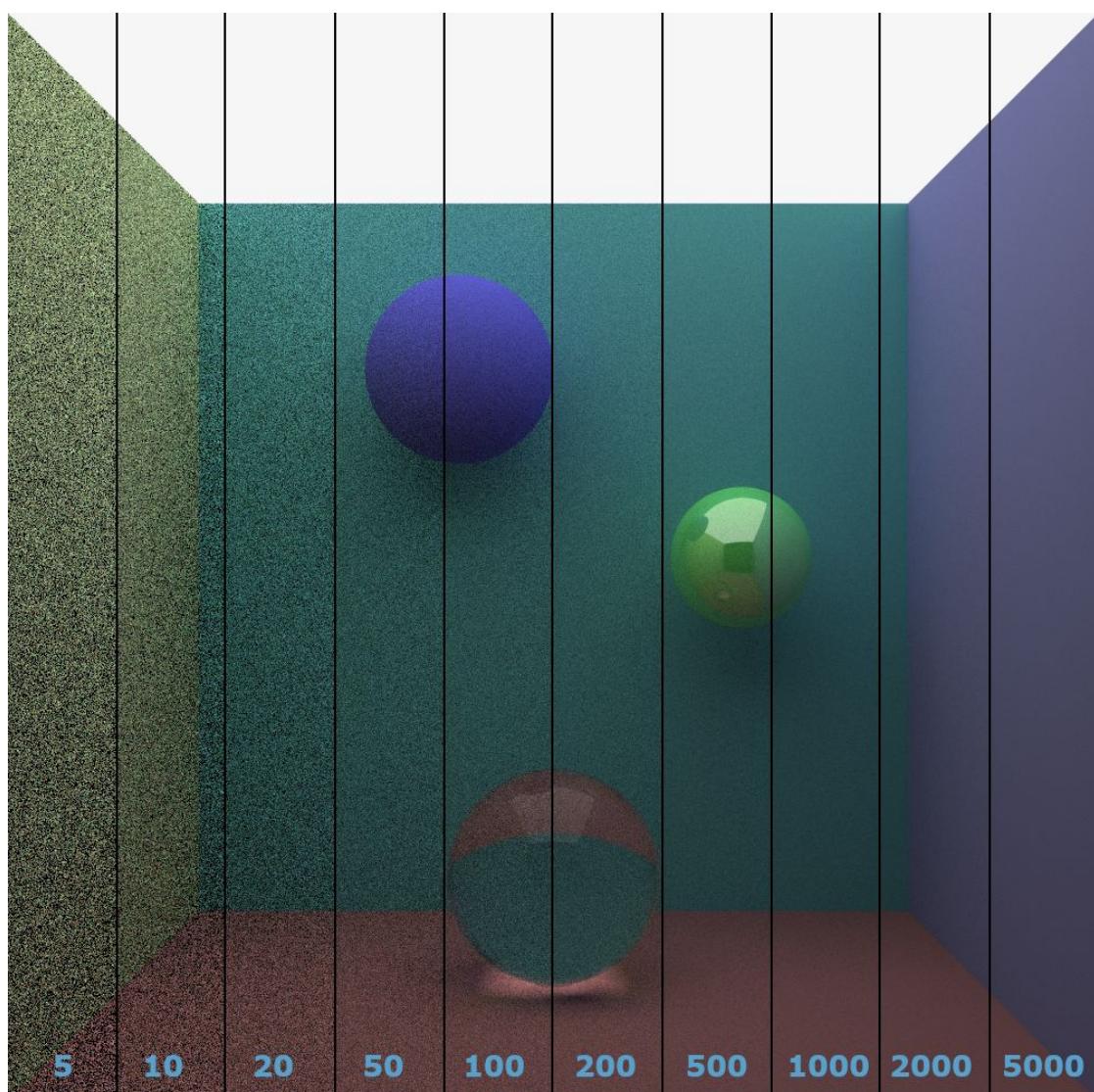


Figura 16: Convergencia de la escena

3.2 Materiales

Creamos nuevas escenas para analizar la convergencia en distintos materiales, en concreto cada una con una esfera de uno de los 4 materiales y vamos a generarlas con distinto número de rayos por píxel, en concreto 5, 15, 100 y 1000.

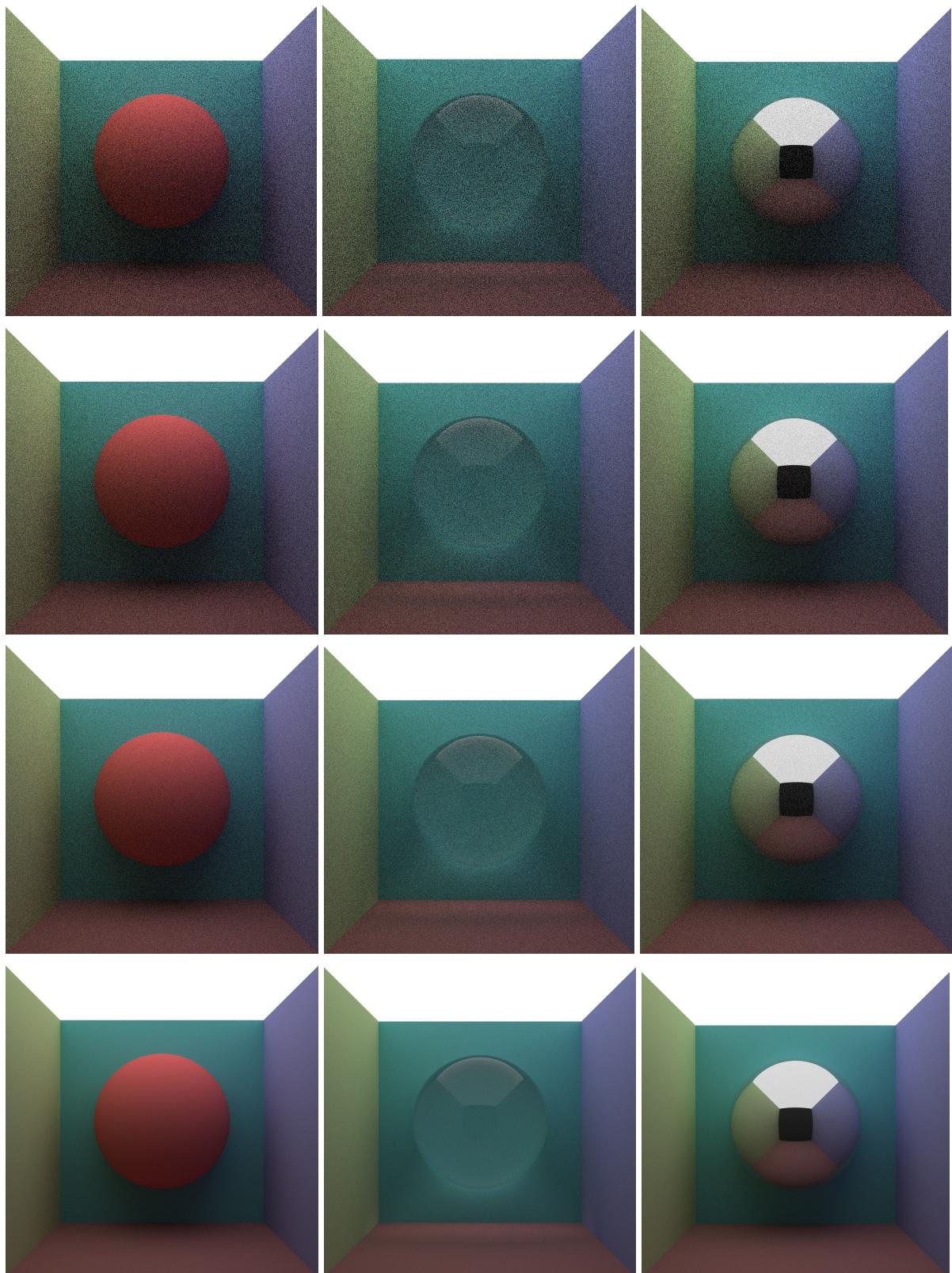
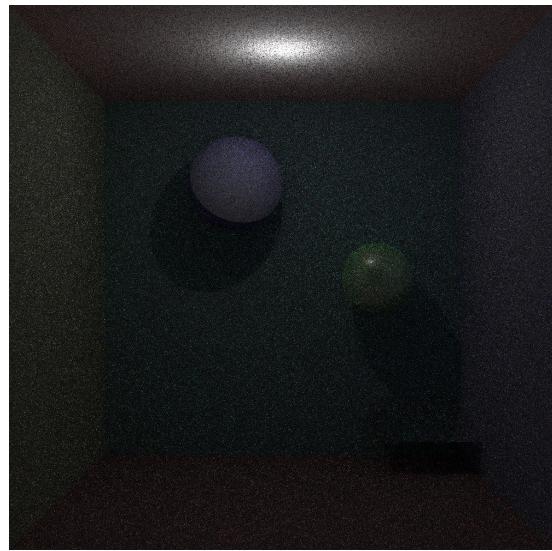
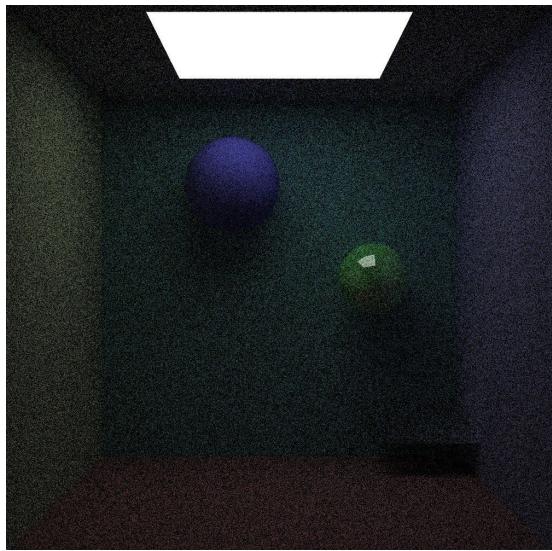
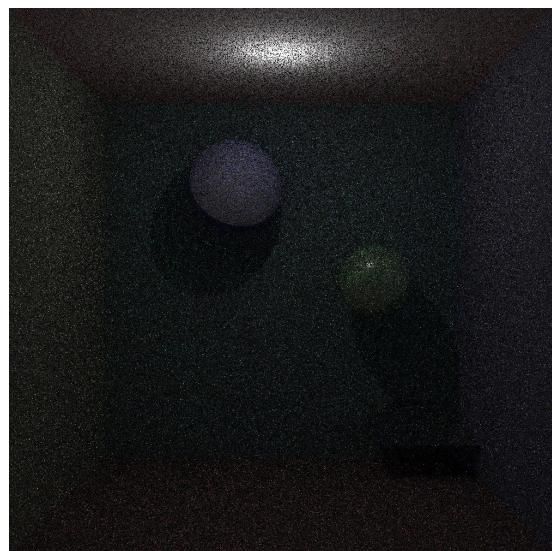
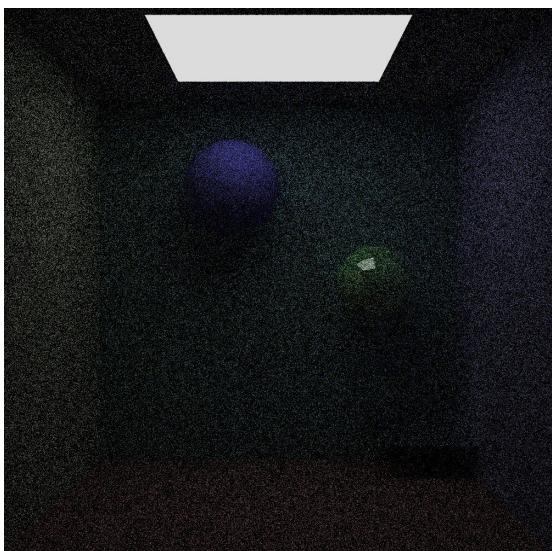


Figura 17: Convergencia de los materiales

Observamos en las imágenes que el ruido se reduce de una manera similar en todos los materiales, pero los materiales que reflejan y refractan convergen más rápido que los difusos ya que los difusos incluyen cierta aleatoriedad en la dirección de un rebote y puede acumular ruido.

3.3 Tipos de luz

Ahora vamos a comprobar la convergencia de la escena dependiendo del tipo de luz, compararemos una escena con una luz de área y una con una luz puntual. Ambas escenas van a ser comparadas con valores de 5, 15, 100 y 1000 rayos por píxel.



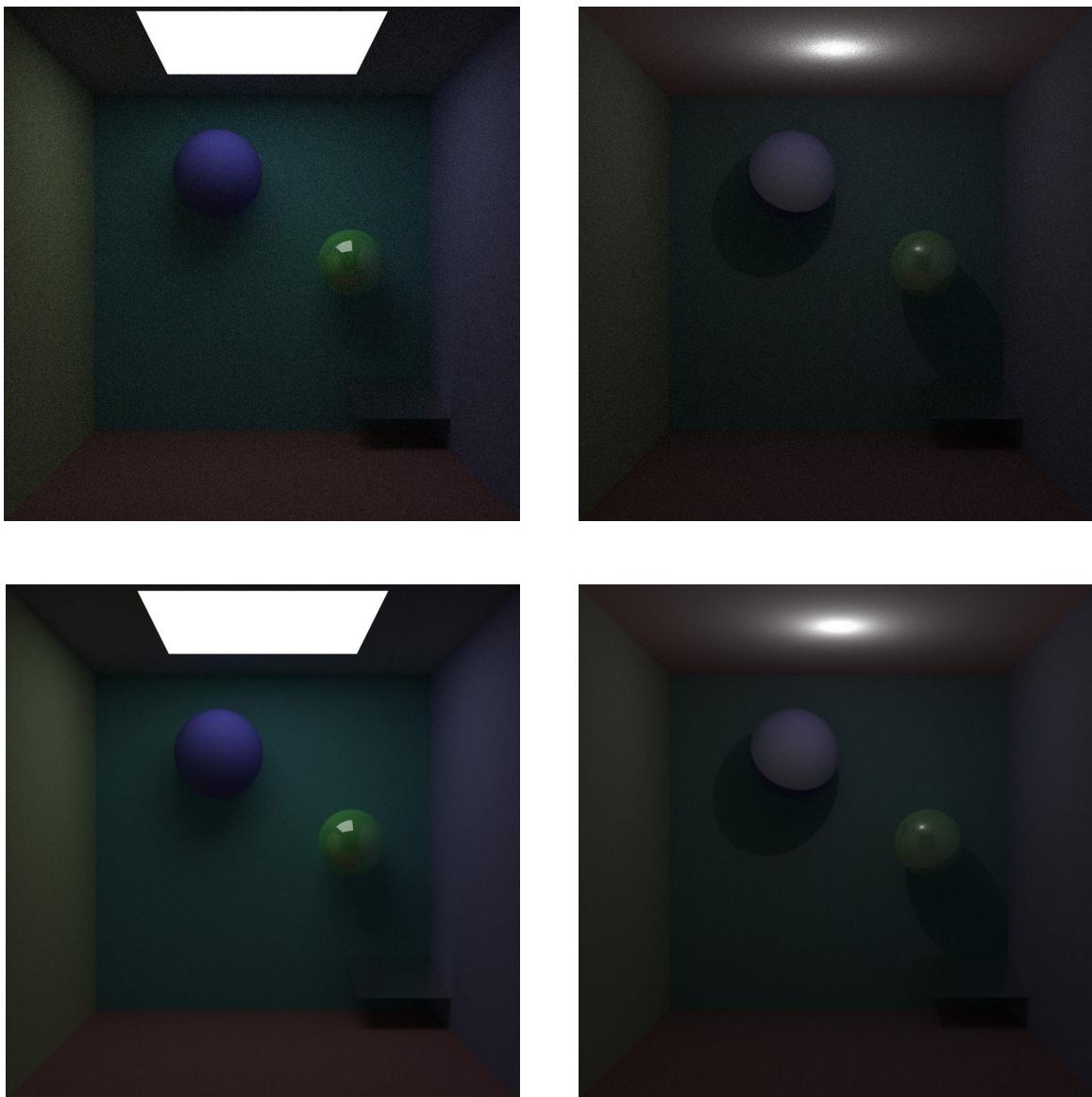


Figura 18: Convergencia de los tipos de luz

Observamos que la escena iluminada por luces puntuales converge más rápidamente, esto se debe a que cuando un punto intersecta con la luz puntual se traza el rayo directamente a la luz, en las luces de área el rayo rebota aleatoriamente en una dirección de la hemisferia.

4. Efectos de iluminación

A continuación explicaremos algunos de los efectos que pueden surgir con diferentes tipos de luz como tipos de sombras diferentes, la absorción y emisión de luz por parte de los objetos o como la luz interactúa con los dieléctricos.

4.1 Sombras

Dependiendo del tipo de luz obtenemos diferentes tipos de sombras y esto se debe a que en la ecuación de render que hemos desarrollado antes, cada tipo de luz usa un sumando

diferente. En especial, las luces de área se espera que el rebote termine por llegar a la fuente mientras que para las luces puntuales se hace la acción de buscarlas.

Para generar un resultado por cada tipo de luz, creamos una cornell box muy sencilla con una esfera sin que llegue a tocar el suelo, mientras que en la parte superior se encontrará la luz del experimento. En el experimento con la luz de área, podemos ver que la sombra generada se va difuminando hasta no tener una forma definida. Este efecto se llama *soft shadow* y se debe a que la luz en un punto puede venir de múltiples puntos en el espacio. Cuanto más se acerca la luz a la esfera o cuanto más grande sea el área de luz, más pequeña y definida será la sombra. En cambio si reducimos el área de luz tendríamos una sombra mucho más grande.

Para el experimento de la luz puntual hemos comentado que en cada punto se va a buscar la luz y por eso el efecto en la sombra es una definición precisa de la proyección de la esfera en el suelo. A este efecto lo llamamos *hard shadow*. Hay que tener en cuenta que el buscar la luz directa no implica en ningún momento que la sombra sea completamente oscura, ya que la iluminación indirecta provoca cierta iluminación en la zona sombreada. En la esfera también vemos el efecto de buscar la luz directamente con la sombra generada.

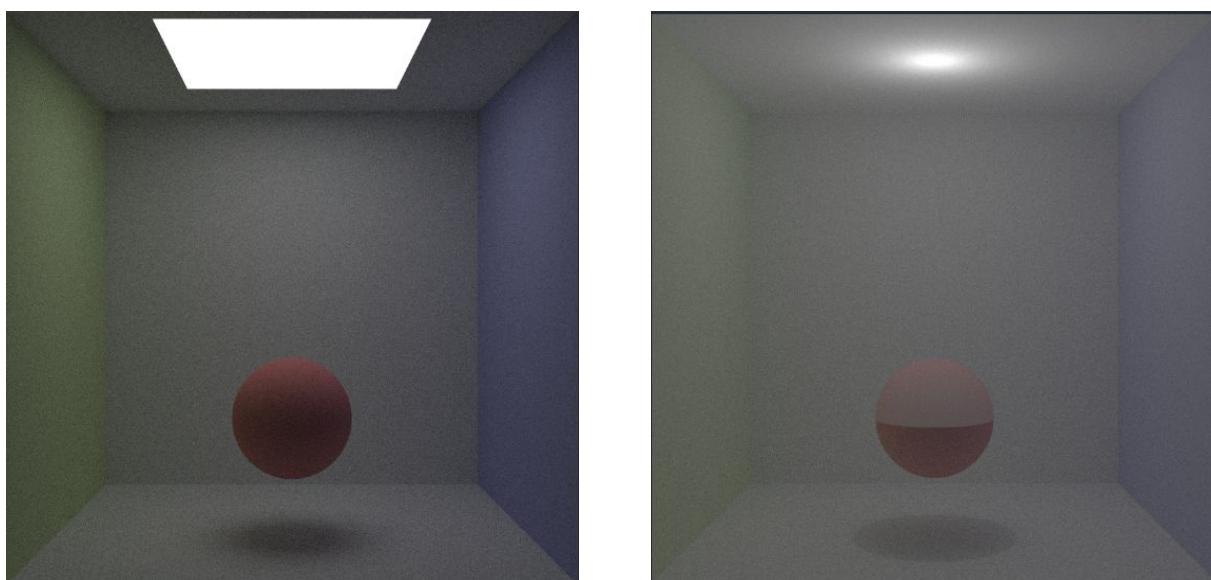


Figura 19: Soft and hard shadow (dcha: Luz de área, izq: luz puntual respectivamente)

4.2 Color bleeding

Cuando la luz incide en un objeto, este tiende a absorber una cierta cantidad de energía y refleja aquella que no absorbe. Esta es una idea un tanto simple de por qué vemos el color de un objeto. Cuando la luz reflejada vuelve a rebotar en otro objeto, esta ya ha perdido cierta energía y puede mostrar otro objeto con un color que no le corresponde.

Para poder ver mejor este efecto se ha creado una cornell box sencilla, con una esfera blanca situada entre dos cartulinas de color, una azul y otra verde. Cuando la luz blanca de esta escena llega a las cartulinas, estas absorben una determinada energía y reflejan otra

pero con distinta longitud de onda. La longitud de onda reflejada debe ser igual a la entrante más la absorbida por el propio objeto (conservación de la energía). Cuando esa nueva energía golpea otro objeto, vemos que imita el color, y esto es debido a que la energía no absorbida en ese momento es precisamente la faltante del rebote anterior, otorgando este efecto.

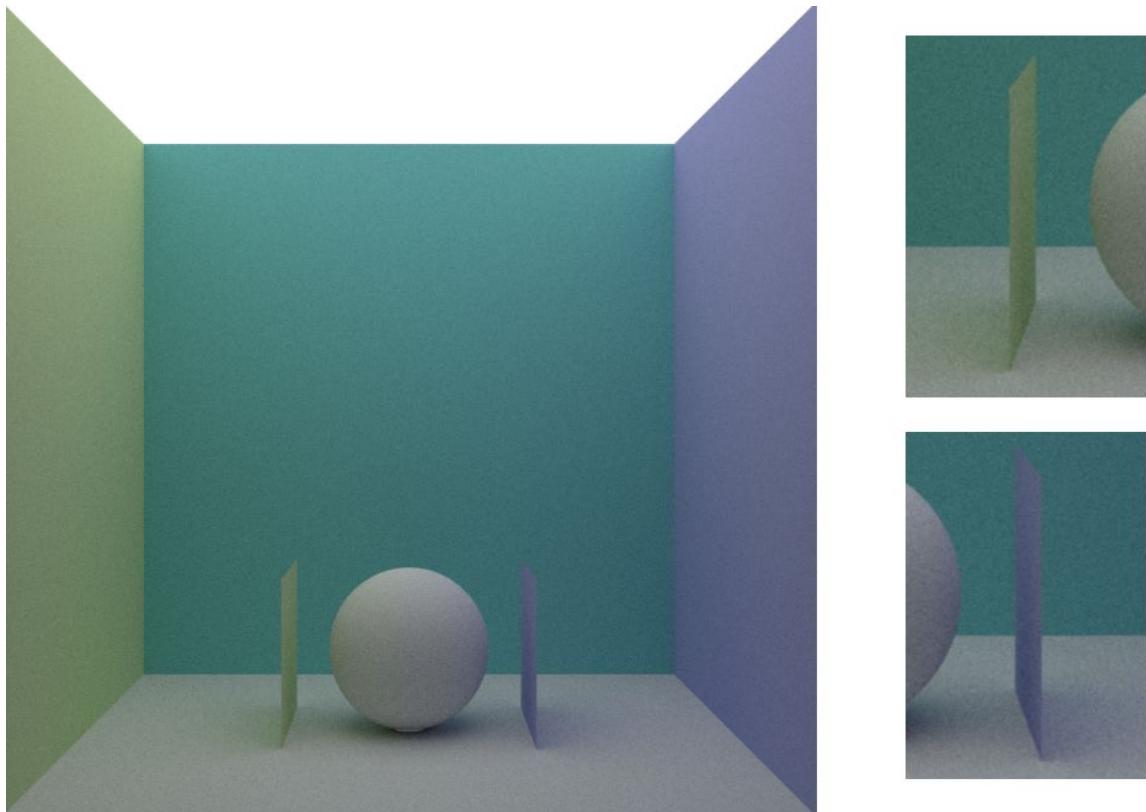


Figura 20: *Color bleeding*: ver como suelo y esfera se tornan del color de las cartulinas.

4.3 Cáusticas

Este efecto se manifiesta cuando la luz pasa a través de elementos que generan refracción, ya que tiene que ver con la distorsión que sufre la luz cuando atraviesa un objeto de este tipo y el resultado es un haz de luz concentrado.

En la imagen se observa que este efecto se ha generado debajo de ambas esferas de cristal y se diferencia la luz origin que ha generado el efecto gracias al color.

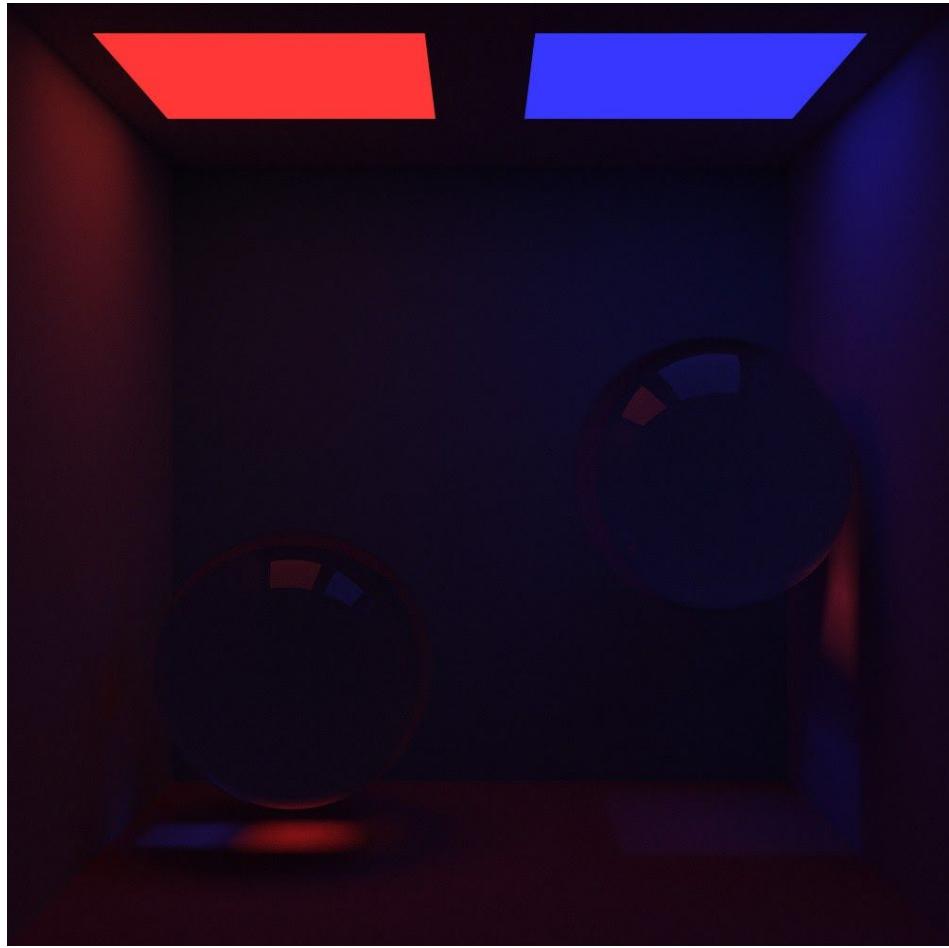


Figura 21: Cáusticas

5. + Opcionales

Mientras que hemos desarrollado el trabajo, hemos observado que ciertos opcionales eran razonables en coste de tiempo y los hemos implementado.

5.1 Triangulos + PLY

Cuando finalizamos la versión pobre del *ray tracer*, intentamos añadir nuevas figuras geométricas, como por ejemplo planos finitos o triángulos. Estos últimos, venían motivados ya que los objetos .ply es un parseo de texto bastante simple que añade triángulos a la escena.

Para los triángulos, encontramos un tutorial [4] que explicaba bastante bien la intersección de un rayo con un triángulo. Los pasos que sigue este ejemplo son la intersección del rayo en el plano en el que se encuentra el triángulo, para luego comprobar si está dentro o no de la forma geométrica.

Tras implementar los triángulos, lo siguiente era el .PLY. La primera decisión de diseño fue que todos los triángulos que constituyan la malla fueran tratados como un solo objeto en la escena, y que tuvieran las mismas propiedades que el resto de objetos. Todos los triángulos

se crean a través del parseado del fichero y se introducen en una lista con una modificación, se obtiene el centro de la figura y se resta del resto de puntos para tratar a los triángulos en un futuro como coordenadas locales. Esto permite usar las funciones de traslación solo actualizando el centro de la figura, mientras que para rotación o escalado si que se procede a modificar todos los triángulos pero desde las coordenadas locales.

Actualmente las mallas de triángulos cuentan en su intersección con un suplemento: para evitar muchas comprobaciones se ha encapsulado la malla en un Axis-Aligned Bounding Box [5]. Esta mejora solo se cuenta para la intersección de la malla, debido a que actualmente no existe una BVH detrás para hacerlo más eficiente -> si un rayo choca contra la malla de triángulos, se ve obligada a buscar la intersección en todos sus componentes. El no implementar el BVH se debe a la falta de tiempo.

Adicionalmente, para pruebas y debug, los triángulos generados tras la lectura del fichero obtienen siempre colores aleatorios.

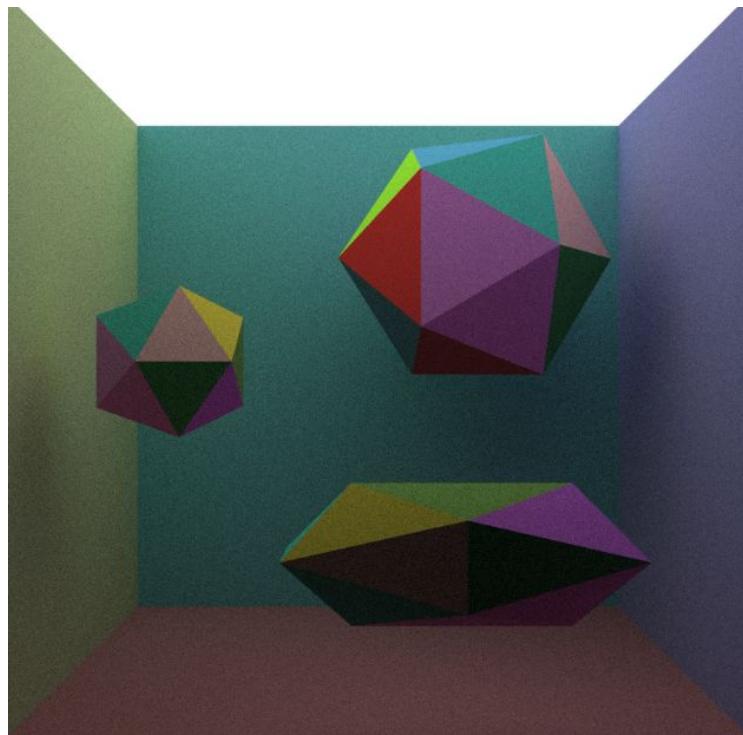


Figura 21: Poliedros introducidos por .PLY y transformaciones (giro, escalado...)

5.2 Paralelización

El algoritmo Path Tracing es un algoritmo costoso computacionalmente, la generación de una imagen con muchos objetos, mucha resolución y muchos rayos por píxel puede tardar horas.

Hemos decidido paralelizar nuestro algoritmo, dividiendo el trabajo en múltiples hilos de ejecución. Hemos probado estrategias de reparto de trabajo como dividir la imagen en cuadrantes o dividirla en filas/columnas pero lo más eficiente es que **cada píxel** se compute en paralelo.

La herramienta **OpenMP** nos facilita la tarea de paralelizar, debemos añadir la opción `-fopenmp` al compilar y usar directivas de compilador en nuestro programa, en nuestro caso escribimos la siguiente directiva en los 2 bucles anidados que controlan el cómputo de los pixeles.

```
#pragma omp parallel for schedule(guided,1)
```

En la directiva usamos el parámetro *guided* para controlar el reparto de hilos ya que logró el mejor rendimiento. Para comparar rendimientos el programa se está ejecutando sobre un Intel i5-4690k 4 cores 4.3GHz, sin paralelización generamos la imagen (1000x1000 px y 100 rppx) en 88.56 segundos mientras que con paralelización la generamos en 24.17 segundos, por lo que obtenemos un **speedup de 3.66**, lo cual se acerca al máximo teórico, que sería 4 ya que nuestro CPU tiene 4 cores.

6. Distribución de la carga de trabajo

Desde un principio siempre se ha intentado ir al día. En las primeras sesiones prácticas, donde se desarrollaron lo básico de los tone mappers o el ray tracing, se establecía como fecha límite las dos semanas desde la impartición de la clase práctica y trabajando siempre de forma paralela.

Para las sesiones de path tracer y photon mapping ya se pensó en otro calendario: noviembre era un mes clave para otras asignaturas por lo que se pospuso debido a la certeza del consumo de tiempo inédito de este trabajo. A partir de este mes, dividímos la carga de trabajo en pequeñas tareas modulares que no tuvieran un peso importante sobre el código para hacerlas cuando cada uno cuando pudiera, ya que si el componente era muy complejo nos reunímos con el fin de hacerlo juntos y tomar las decisiones más importantes entre los dos integrantes.

Durante todo el trabajo, se ha usado un repositorio en github, trabajando en algunas ramas por separado para asegurar siempre que los módulos de trabajo siempre funcionaran bien antes de subirlo a la rama master. En el repositorio se encuentran tantas carpetas como sesiones de prácticas y la compilación de todos estos ficheros se consigue copiando en la terminal el primer comentario de cada fichero (gcc ...).

Bibliografía

- [1] Diapositivas del curso de informática Gráfica
- [2] [RSSF02] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. ACM Trans. Graph., 21(3):267–276, July 2002.
- [3] Shirley, Peter & Robison, Austin & Morley, R.. (2011). A Simple Algorithm for Managing Color in Global Tone Reproduction. Journal of Graphics, GPU, and Game Tools. 15. 10.1080/2151237X.2011.610677.
- [4] Ecuaciones Triángulos. Scratchapixel, Lessons: 3D Basic Rendering. Ray Tracing - Rendering a triangle.
- [5] GDBooks (gitbooks.io) - 3D Collisions. Chapter 3: Raycast AABB