DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Ray Tracing en C++

10 de febrero de 2022 — Fundamentos de la Computación Gráfica

Grupo 13

Integrante	LU	Correo electrónico
Julian Recalde Campos	502/17	Recaldej@hotmail.com.ar
Jonathan Teran Carballo	643/18	jonathan.nerat@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Resumen

En este trabajo implementaremos un Ray Tracer basado en las guías de Shirley [1] capaz de renderizar distintos objetos (esferas, cubos, planos y mallas de triángulos) y materiales (dieléctricos, metales y colores difusos). Además veremos una de las estructuras más utilizada para optimizar el renderizado cuando tenemos muchos objetos en una escena, el KD Tree.

1. Introducción

En el mundo de la computación gráfica, existen distintos algoritmos para renderizar objetos 3D en imágenes 2D. Se pueden clasificar en 2 categorías: *image-order rendering* y *object-order rendering*. En *object-order rendering*, se considera cada objeto de una escena y se modifican los pixeles que este influye. Por otro lado, en *image-order rendering*, se recorre cada pixel de la imagen, y se calcula su valor considerando a todos los objetos que influyen en el.

El Ray Tracing es un ejemplo de un algoritmo de *image-order rendering*. Para generar una imagen, se lanzan *rayos* hacia cada pixel y se calcula el color del mismo como una combinación de todos los objetos con los que este rayo colisiona.

2. Objetos

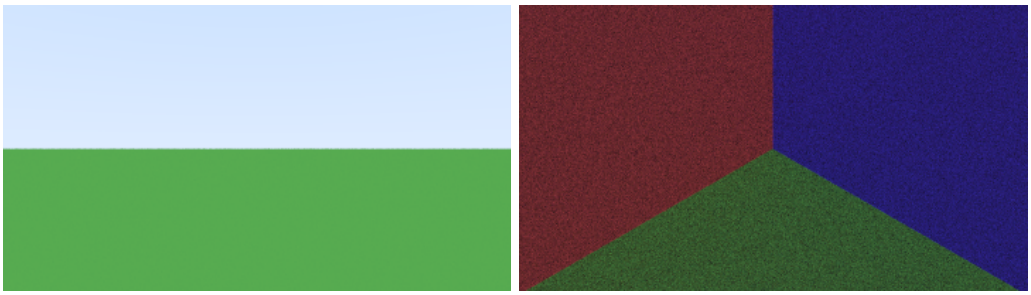
Para esta implementación, creamos 4 tipos de objetos básicos representables en nuestra escena:

- Esferas
- **Planos**
- **Cajas**
- **Triángulos**

A continuación explicaremos brevemente la implementación de aquellos que están en negrita, ya que el resto podemos encontrarlos en la guía de Shirley.

2.1. Planos

Representamos a un plano como un par de 2 vectores, uno apuntando a un punto de referencia (P) que pertenezca al plano, y uno perpendicular al plano (N). Consideremos un rayo $r(t) = O + tD$, donde O es el origen y D es la dirección del mismo, entonces la intersección entre el plano y el rayo se da en $t = \frac{(P-O)\cdot N}{D\cdot N}$. Notar que la intersección se da siempre y cuando $D \cdot N$ no de 0, es decir, el rayo no sea paralelo al plano.



(a) Render del plano x=0 (b) Render de los planos x=0, y=0 y z=0

Figura 1: Planos

[1] https://raytracing.github.io/books/RayTracingInOneWeekend.html

En la figura 1a vemos el plano $x = 0$ en verde, mientras que en la figura 1b vemos los 3 planos $x = 0$, $y = 0$ y $z = 0$ en rojo, verde y azul respectivamente.

2.2. Cajas

Las cajas las implementamos con 6 planos, cada uno representando a la cara correspondiente. Para que la implementación sea simple, decidimos que las caras estarían alineadas a los ejes cartesianos. Para verificar si un rayo colisiona con la caja, recorremos cada cara y calculamos el punto en el que el rayo y el plano se intersecan. Si este punto pertenece a alguna cara de la caja, lo consideramos para calcular el punto final, que sería el más cercano. Si ningún punto pertenece a una cara de la caja, consideramos que no hay colisión.

Las cajas se definen por 2 vertices opuestos, el que tiene las coordenadas menores y el que tiene las coordenadas mayores. En la figura 2 podemos ver una caja con un vértice rojo $P = (0, ,1, 0)$ y un vértice amarillo $Q = (1, 1,1, 1)$.

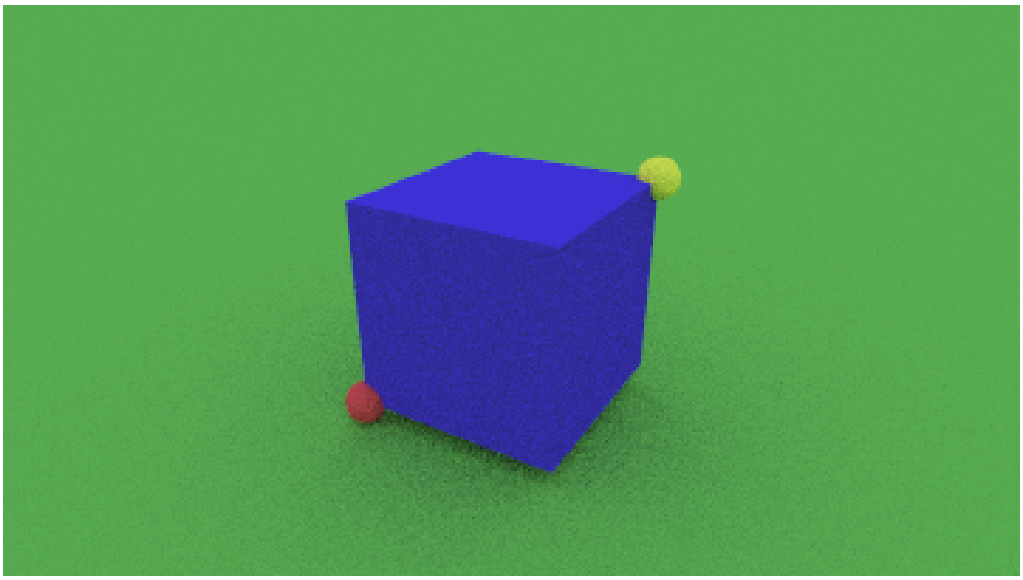


Figura 2: Render de una caja, indicando los vertices que lo identifican

2.3. Triángulos

Los triángulos los definimos con 3 vertices, P_0, P_1 y P_2. Para calcular la intersección con un rayo, primero calculamos la intersección con el plano que contiene al triangulo y luego verificamos si este punto P se encuentra dentro del triangulo. Para hacer esto, calculamos las normales de los triángulos P_0P_1P, P_1P_2P y P_2P_0P. Si estas normales apuntan hacia el mismo lado que la normal del triangulo original, entonces el punto P se encuentra dentro.

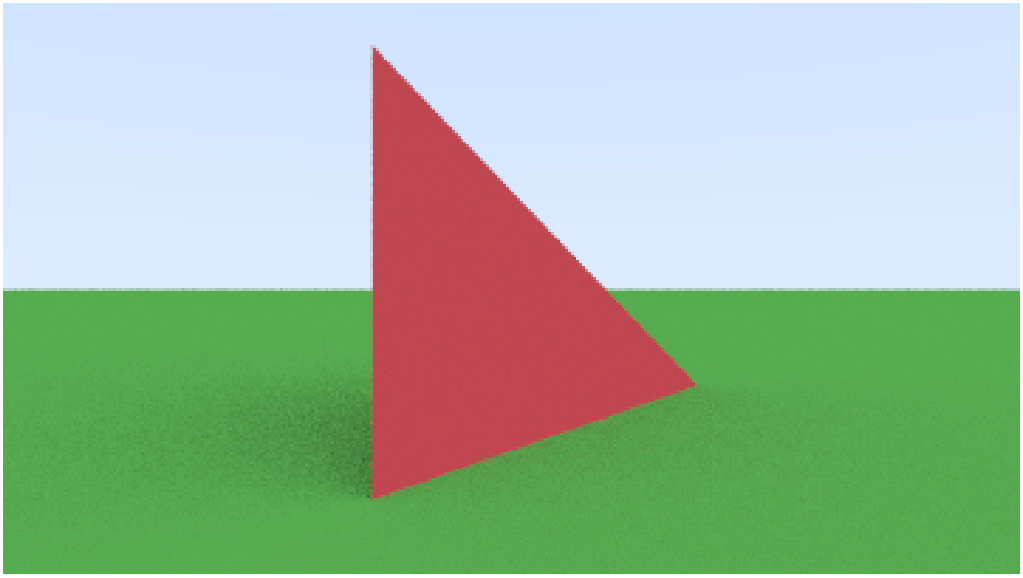


Figura 3: Render de un triángulo

En la figura 3 vemos un triangulo de vertices $(1, 0, 0)$, $(0, 1, 0)$ y $(0, 0, 1)$.

3. Estructuras

Además, de los objetos mencionados, agregamos 3 estructuras adicionales para facilitar el renderizado de objetos más complejos:

- Lista de objetos
- Malla de triángulos
- KD Tree

A continuación explicamos brevemente la implementación de cada uno.

3.1. Lista de objetos

Tiene una implementación muy directa, ya que como su nombre indica, solo almacena una lista de objetos (básicos u otras estructuras). La colisión de un rayo contra este se calcula como el punto más cercano de colisión entre el rayo y alguno de los objetos que almacena, si es que hubiese colisión.

Es una estructura de utilidad para facilitar el renderizado de escenas o para crear otras estructuras más complejas, como las que veremos a continuación.

3.2. Malla de Triángulos

Las mallas de triángulos por lo general consisten de miles de objetos, por lo que la implementación de esta estructura toma como parámetro un archivo en el que se definen los mismos. El archivo consiste de lineas de tipo vértice, en el que se indican las coordenadas de cada vértice, y de tipo cara, en el que se indican que vertices componen a cada triangulo. Dejando de lado esa diferencia, la implementación es prácticamente la misma que la lista de objetos, solo que almacena triángulos.

En la figura 4 podemos ver el render del modelo de un pato.

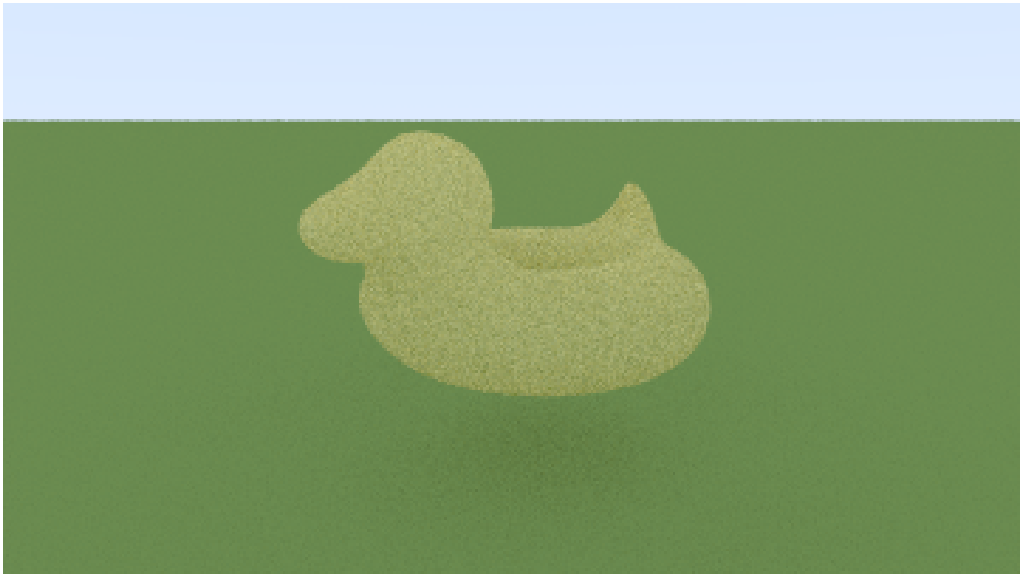


Figura 4: Render de una malla de triángulos.

Sin embargo, debido a la sencillez de la implementación de la lista de objetos, esta estructura resulta muy lenta ya que se debe recorrer los miles de triángulos uno por uno para detectar colisiones. Para poder mitigar el tiempo de procesamiento, utilizamos la siguiente estructura para calcular menos colisiones, el KD Tree.

3.3. KD Tree

El problema que tiene la implementación anterior es que la complejidad para saber si un rayo colisiona con algún triángulo de la malla crece con la cantidad de objetos en la lista.

Los *axis-aligned bounding boxes* (AABB) son estructuras que mejoran la complejidad dependiendo del tipo (Octree, BPS Tree, KD Tree, etc.). Logran esto dividiendo el espacio en partes iguales, y encerrando cada parte en una caja que contenga todos sus objetos. Luego cada partición del espacio se divide de la misma forma, creando así una jerarquía de cajas asociada a cada nodo de un árbol. La raíz del árbol representa la caja más grande que contiene todos los objetos, y los nodos hijos están asociados a cada partición del espacio, con sus respectivas cajas.

El *KD Tree* es una estructura AABB en el que en cada paso de la división del espacio, se hace un único corte en el eje donde la longitud de la caja es mayor. Esto significa que el árbol resultante es un árbol binario, ya que cada espacio se divide en 2.

Esta nueva estructura recibe como parámetro una lista de objetos y un entero indicando cuantos elementos debe tener las hojas del árbol. Luego se procede a dividir el espacio hasta llegar a nodos que contengan como máximo la cantidad indicada. De esta forma, si se especifica como lista de objetos una malla de 16000 triángulos y como cantidad máxima 500 objetos, el espacio se divide hasta llegar a un árbol de altura 6 con 32 hojas, cada una con 500 triángulos.

Esta estructura nos permitió acelerar el renderizado tanto de las mallas de triángulos, como de escenas complejas que cuentan con muchos objetos.

4. Resultados y Conclusión

En la figura 5 podemos observar un render con todos los objetos y estructuras mencionadas en este trabajo.

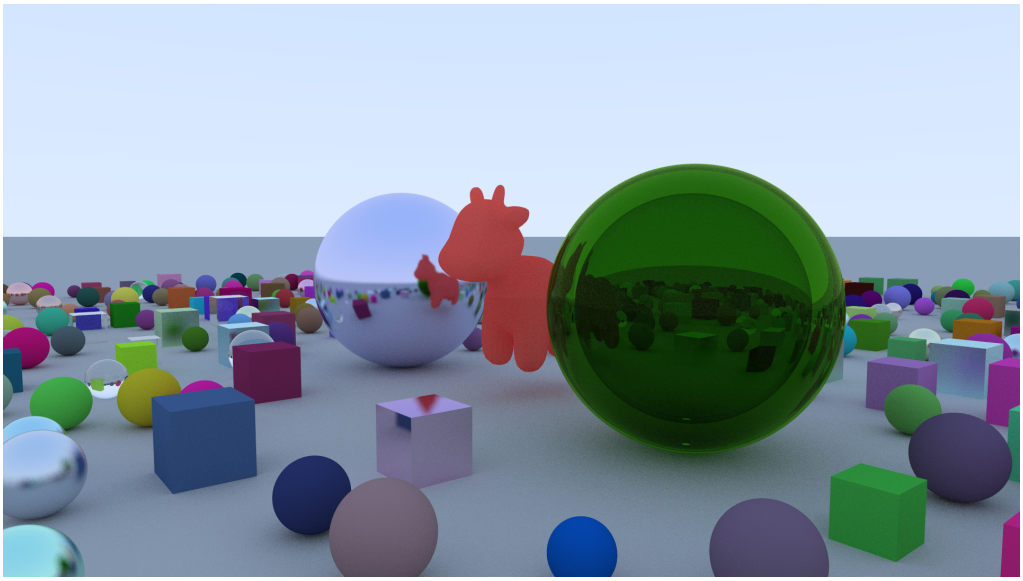


Figura 5: Render de una malla de triángulos.

Luego de ver en detalle dos técnicas de renderizado, una basada en *object-order rendering* (rasterización), y otra en *image-order rendering* (ray tracing), las diferencias son claras.

La rasterización cuenta con algoritmos que al día de hoy han sido optimizados a tal punto que existe todo un pipeline dentro de la mayoría de las tarjetas gráficas. Es necesario conocer este pipeline y la librería correspondiente para interactuar con el mismo, pero su optimización resulta en renders de buena calidad en muy poco tiempo.

Por otro lado, los algoritmos de ray tracing suelen ser más intuitivos en cuanto a la idea general y su implementación, y no requieren (en principio) de conocimientos de ninguna librería. Si bien en la actualidad surgen nuevas tarjetas gráficas que incluyen unidades para realizar la intersección de los rayos de manera optimizada, la realidad es que sin estos recursos el ray tracing resulta lento en comparación a la rasterización. Sin embargo, la calidad de las imágenes resultantes son claramente superiores y más realistas.

El futuro del ray tracing y su aplicación tanto en videojuegos como en producciones audiovisuales resulta muy prometedor. Las nuevas tarjetas gráficas capaces de realizar ray tracing en tiempo real permitirán que esta rama de la computación gráfica avance a pasos agigantados y que la calidad del contenido multimedia que producimos y consumimos crezca de manera acorde.