DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico - Grupo 5

Ray Tracing in One Weekend

8 de diciembre de 2024 Introducción a la Visualización y Simulación Interactivas e Inmersivas

Integrante	LU	Correo electrónico
Arrosio, Valeria	223/20	vale.arrosio@gmail.com
Rosenzuaig, Florencia	118/21	f.rosenzuaig@gmail.com
Valentini, Nicolás	86/21	nvalentini2000@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

1. Introducción teórica

Dada una escena en un espacio 3D, Ray Tracing es una técnica que genera una imagen 2D de ella. Este método emula cómo nuestros ojos ven la realidad. En el espacio 3D, definimos una cámara que sirve de punto de mira. Luego, desde la cámara vamos a emitir rayos que van hacia la escena. Cada rayo que emitimos va a luego colorear un pixel de la imagen resultante. Los rayos pueden chocarse o no con los objetos de la escena.

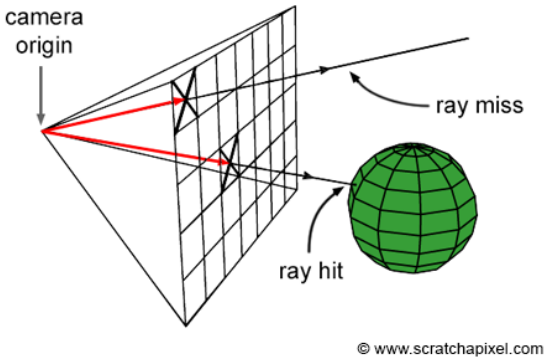


Figura 1: Los rayos van hacia la escena y pueden chocarse o no contra objetos.

En caso de que se choquen, según las propiedades del objeto, vamos a definir cómo se colorea el pixel. Por ejemplo, si el objeto es reflectante, entonces el rayo va a rebotar y el color del pixel va a estar definido por el resultado del rebote.

2. Desarrollo

Para el desarrollo de este proyecto, seguimos las instrucciones del libro Ray Tracing in One Weekend"[2]. En el libro la implementación está en C++, pero nosotros decidimos hacerlo en Python. Elegimos Python porque nos resulta un lenguaje conocido y cómodo.

2.1. Problemas con Python

Una limitación que nos encontramos con Python fue que las funciones no toman parámetros por referencia, por ende en muchos casos tuvimos que devolver tuplas emulando los valores transformados. Otro problema que nos encontramos, es que es más lento que C++, por lo tanto intentamos utilizar un par de optimizaciones.

2.2. Optimizaciones

Desde un principio notamos que hacer una implementación directa del libro podría generar problemas de performance en Python, particularmente en la cantidad de multiplicaciones entre vectores.

La primer solución fue utilizar la librería de Numpy, la cual de por sí trae consigo múltiples optimizaciones para calculo matricial y con performance más cercana a la de C++. Claramente fue necesario hacer múltiples adaptaciones al código para que utilice la librería. Particularmente fue necesario adaptar Vec3, el cual actualmente es Wrapper de un NDarray de 3 elementos.

Esto trajo mejorías para la performance del render, sin embargo, investigamos si había más formas de optimizar dicho código. La siguiente iteración fue la integración de Numba, otra librería de Python pensada para optimizar funciones que utilizan funciones de Numpy. Numba lee codigo de python y luego utiliza un compilador LLVM (Low Level Virtual Machine), el cual genera código para la CPU el cual corre a una velocidad similar a C, haciendo que dicha función corra significativamente más rapido [1]. Esta librería es muy restrictiva sobre qué elementos pueden ser partes de la función, por ende decidimos que unas pocas funciones sean las candidatas a tener el tag `@njit` de Numba. Con este tag se especifica qué funciones son las que serían optimizadas por Numba. La funciones elegidas fueron: obtener la norma de un vector, el producto cruz, el producto punto y el reflejo entre dos Vec3.

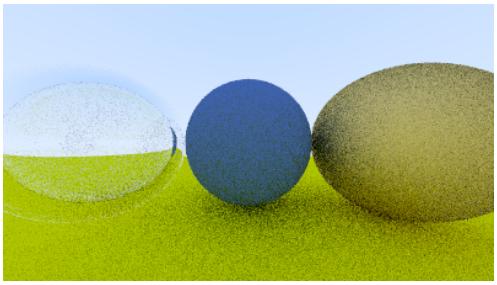
Una vez terminada la implementación del libro, ya utilizando Numpy y Numba, notamos que el render final podía llevar una excesiva cantidad de tiempo, alrededor de 12 horas, por ende quisimos pensar más optimizaciones. La más importante fue sin duda la aplicación de multiprocessing. Fuimos capaces de utilizar multiprocessing ya que el calculo del color de cada rayo es un proceso independiente, es decir que no depende de otros procesos para saber de qué color se tiene que pintar un pixel. Este concepto nos fue fundamental ya que nos abrió la posibilidad de utilizar la librería de python multiprocessing. Con esta librería pudimos separar el renderizado en la cantidad de procesos que requiera el usuario. La idea es separar en partes iguales a la imagen e ir escribiendo el resultado de cada render en archivos temporales separados. Una vez estén todos listos, unirlos para tener la imagen final.

Exploramos utilizar también multithreading para paralelizar el render pero este no tuvo mejoría en perfomrance, y en algunos casos hasta la empeoró. Intentamos crear un ThreadPool para paralelizar distintos aspectos: el render de cada fila de la imagen, el render de cada pixel, el color de cada pixel. Los 3 experimentos dieron resultados negativos, entendemos que esto se puede deber a que se generan un numero muy grande de threads, los cuales de por sí no hacen cálculos muy complejos, pero sí se hace un cambio de contexto muy seguido, y esto debe impactar negativamente en performance.

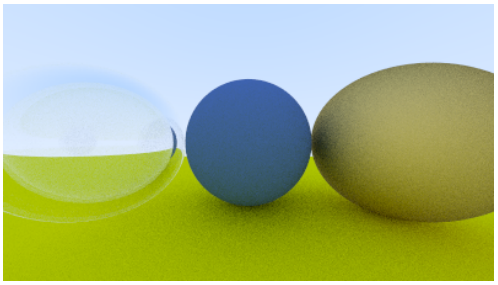
3. Resultados y discusión

A lo largo del proyecto, fuimos generando todas las imágenes a la par que el libro, obteniendo resultados similares (salvo diferencias causadas por usar algunos parámetros menores con el fin de disminuir el runtime).

Un ejemplo de esto puede verse en la Figura 2, correspondiente a la imagen 18 del libro. La primera imagen fue generada con un `sample_per_pixel` de 10 mientras que la segunda utilizó un valor de 100. La diferencia es muy notoria, siendo la segunda imagen mucho más suave en relación a la primera.



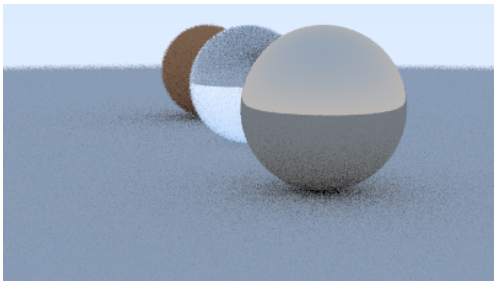
(a) Imagen con sample 10



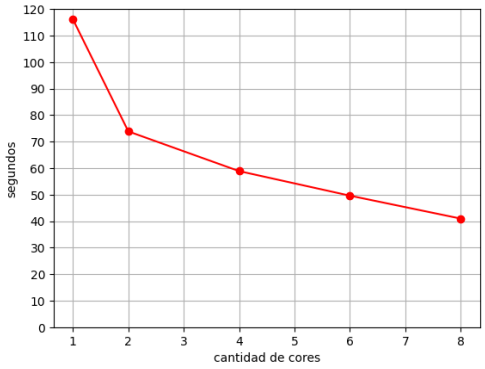
(b) Imagen con sample 100

Figura 2: Diferencia que hace el número de samples

Otra comparación interesante surgió al implementar paralelismo con múltiples cores. En la Figura 3 mostramos la imagen que generamos con distintas cantidades de cores (la imagen siendo una versión básica de la imagen final, sin esferas aleatorias). Notamos como con un solo core, la imagen tardó casi dos minutos en renderizarse (específicamente 116 segundos), y este tiempo se reduce considerablemente al aumentar la cantidad de cores utilizados. Al usar solo un core más, la renderización fue un 36 % más rápida, completándose en unos 73 segundos. Esta disminución se ralentiza luego de esto, pero de todos modos, al usar 8 cores, la imagen se genera en 41 segundos, 64 % más rápido que al usar un solo core.



(a) Imagen a generar



(b) Tiempo en generar la imagen con n cores

Figura 3: Comparativa de tiempo por cantidad de cores

Como muestra final del trabajo, presentamos la Figura 4, análoga a la imagen 24 del libro, siendo la última imagen generada del proyecto. La imagen tiene 400 pixeles de ancho con un `aspect_ratio` de 16 : 9 lo que resulta en 225 pixeles de altura. Utilizamos 100 `samples_per_pixel` y un `max_depth` de 10. La imagen fue generada utilizando 8 cores, y tardó 226 minutos (3 horas y 46 minutos). Un dato interesante que resalta la eficiencia de utilizar más cores para el renderizado, si tomamos que con 8 cores tardamos el 36 % de lo que tardaríamos usando 1 solo core, esta misma imagen hubiera tardado casi 11 horas en renderizarse.

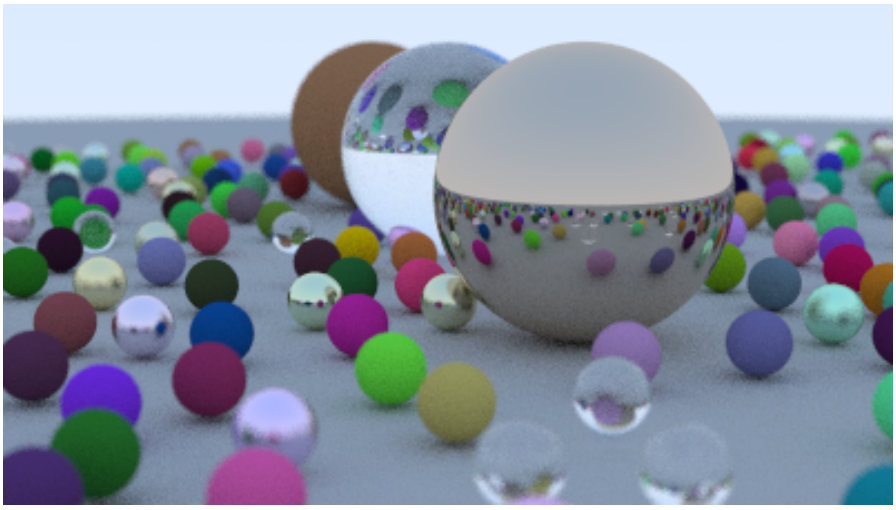


Figura 4: Imagen Final

4. Conclusiones

Creemos que obtuvimos muy buenos resultados con python en vista de la calidad de las imágenes generadas, siendo en muchos casos prácticamente idénticas a las del libro. Sin embargo nos parece importante remarcar que python es un lenguaje mucho menos performante que C++ y eso representó una limitación a la hora del volumen de experimentos a realizar, tanto por el tamaño de las imágenes como la cantidad de samples. Hacer un render final con los parámetros idénticos al del libro, con 500 `samples_per_pixel` y una imagen de 1200 pixeles de ancho en `aspect_ratio` de $16:9$, estimamos que hubiese tardado alrededor de 56 horas.

De todas las optimizaciones que realizamos para obtener imágenes en un tiempo medianamente razonable, creemos que la más importante fue la aplicación de multiprocessing. Creemos que esta idea puede ser aplicable a lenguajes más allá de python y pueden representar una mejoría significativa en performance en cualquier escenario, ya que sería una forma de aprovechar que el color de cada pixel se calcula independientemente.

Referencias

[1] Anaconda. *Numba Documentation*. 2020. URL: `https://numba.readthedocs.io/en/stable/index.html`.

[2] Steve Hollasch Peter Shirley Trevor David Black. *Ray Tracing in One Weekend*. Ago. de 2024. URL: `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.