Computación Gráfica
Ray Tracer
Grupo 2

Guillermo Campelo
Juan Ignacio Goñi
Juan Tenaillon
Santiago Vázquez

May 19, 2010

Abstract

Diseñamos y desarrollamos un motor de Ray Tracing, determinando intersecciones entre rayos y objetos, considerando luminosidad, reflexiones, refracciones, sombras, texturas y anti-aliasing.

1 Introducción

A lo largo de este informe, detallaremos el diseño y desarrollo del motor de Ray Tracing y explicaremos en detalle cómo decidimos implementar las distintas partes que componen a la aplicación.

Para realizar el mismo, partimos del trabajo práctico anterior, que consistió en el desarrollo de un Ray Caster. A partir del motor de ray casting, construimos encima el motor de ray tracing, que toma en consideración muchas cosas que el ray caster no tenía. Entre estas cosas se encuentran las luces, sombras, reflecciones y refracciones, texturas y la posibilidad de tener movilidad en la cámara entre otras.

Además de explicar los detalles del diseño y de la implementación, mostraremos imagenes generadas utilizando el motor haciendo uso de las distintas opciones que presenta.

También desarrollamos técnicas de optimización para hacer más eficiente al motor y, como resultado, generar imágenes de igual calidad a mayor velocidad. En nuestro caso, implementamos Octrees, debido a que nos brinda tanto volúmenes envolventes como también subdivisión espacial. Una explicación más detallada de esto podrá ser encontrada en la sección 4.1.

En la sección 2 detallaremos cada uno de los elementos que componen a una escena, como las cámaras, las luces y los objetos que la pueblan. En la siguiente sección, la 3, explicaremos más en detalle el motor de ray tracing, explicando qué es, cómo funciona, las características del mismo, las optimizaciones que decidimos implementar y una serie de pruebas realizadas con el motor. Por último, en la sección 6, daremos nuestras conclusiones al respecto sobre el motor que implementamos.

2 Escena

En esta sección comentaremos las distintas partes que componen a la escena. Estas son las cámaras, las luces y los objetos mismos que la componen, como los cubos, esferas y figuras complejas.

2.1 Cámara

Desarrollamos dos tipos de cámaras distintas, *pinhole* y *thinlens*.

La cámara *pinhole* es la cámara estándar mientras que la cámara de tipo *thinlens* es la utilizada para modelar la profundidad del campo.

A los efectos de nuestro trabajo y el alcance del mismo, el tipo de cámara no influye.

Una característica intrínseca en la definición de la cámara utilizando $Sunflow$ es que es posible tener una cámara movil. Esto nos permite ver la misma escena desde difrentes perspectivas. Esto lo apreciamos en la figura 1 y 2.

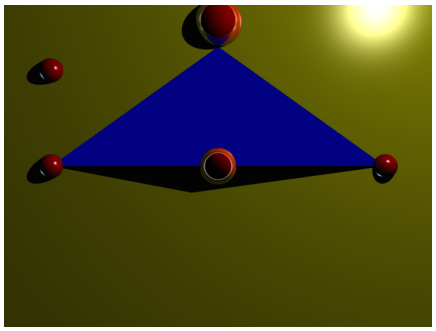

Figure 1: Cámara en (0, 0, 20)

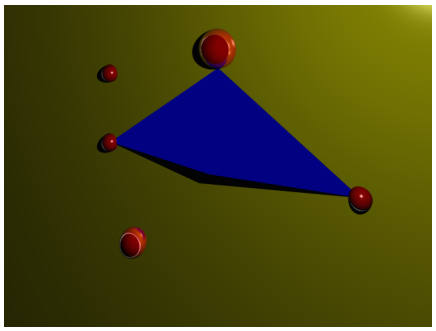

Figure 2: Cámara en (10, 10, 20)

2.2 Luces

Las luces que implementamos fueron dos, una luz de ambiente y una luz puntual.

En caso de no haber una luz puntual definida, la luz ambiente entra en juego para que se pueda apreciar algo de la escena. Esto surgió como una necesidad ante la falta de luces puntuales en algunas escenas. Decidimos implementar una tenue luz ambiental para poder observar las escenas.

La implementación es simple. Al no haber ninguna luz puntual definida en la escena, el motor busca el color que debería tener el pixel y devuelve ese valor oscurecido.

El otro tipo de luz implementada, es una luz puntual. Esta nos permite obtener escenas con sombras, refracciones y reflecciones en la imagen resultante. La luz cuenta con una posición y una potencia y a medida que nos alejamos de ella, la potencia disminuye. Esta disminución fue inicialmente implementada usando la ecuación 1.

$$PotenciadeLuz(d_{punto-luz}) = \frac{P_{luz}}{ln(d_{punto-luz} + 1) + 1} \tag{1}$$

Luego se decidió usar una disminución lineal de la luz, ya que la fórmula 1 nos daba una luz muy fuerte aún con una muy poca potencia. Ésta ecuación se puede ver en 2.

$$PotenciadeLuz(d_{punto-luz}) = 1 - \frac{d_{punto-luz}}{P_{luz}} \tag{2}$$

Obviamente, al existir una luz puntual, también existen sombras. Un pixel se encuentra sombreado si hay algún objeto opaco entre el punto y el centro de la luz. Si el objeto entre medio es translúcido, el punto no estará sombreado.

Con este último punto se nos presentó un error que no fue fácil advertir. En caso de existir un objeto que intersecte el rayo entre el punto y la luz, pero que no esté entre la luz y el punto, sino detrás de la luz, este pixel aparecía sombreado. Luego de que advertimos cual era el problema, tuvimos que realizar un cálculo extra para determinar si la distancia entre la intersección con el objeto era mayor o menor que la distancia entre el punto y la luz.

Figure 3: Sin luz.

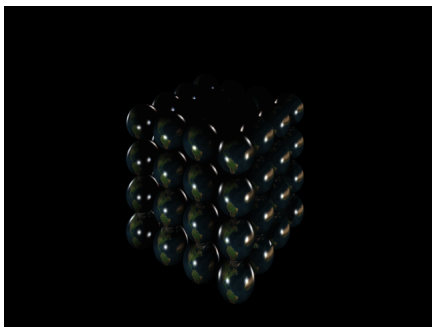

Figure 4: Con 3 luces.

En la figura 3 observamos que se ve muy poco de los 64 mundos, mientras que en la figura 4 las observamos más claramente.

2.3 Primitivas

Las primitivas que desarrollamos fueron tres, estas son esferas, cubos, triángulos y planos. De esta forma se pueden definir figuras más complejas, como explicamos a continuación, que son los *mesh* o mallas de triángulos.

2.3.1 Esferas

El único cambio con respecto a las esferas del primer trabajo práctico, es el hecho de que acá utilizamos un eje imaginario para poder rotar la esfera.

Al crear una esfera, se calcula un eje para luego al rotarla poder calcular qué punto de la textura le pertenece.

Para leer una explicación de la intersección entre un rayo y la esfera, leer el informe del Trabajo Práctico #1.

2.3.2 Triángulo

La representación del triangulo y la intersección de un rayo con el mismo lo comentamos en el informe del Trabajo Práctico #1. Cuando desarrollamos

el motor de ray tracing, tuvimos que agregarle algunas cosas más, como las normales en cada punto y el mapeo con los puntos UV para el caso del mapeo con las texturas.

Para mapear las texturas en los triángulos, con los 3 puntos que definen al triángulo, se crean 2 vectores, entre uno de los puntos y los otros dos, que forman una base para el plano sobre el cual está el triángulo. Con estos vectores, su punto inicial y el punto contra el que hizo intersección el rayo emitido, se toma el componente que multiplica a cada uno de los vectores de la base, que varía entre 0 y 1 obteniendo así, los valores de u y v contra los que se mapea en la textura el punto de intersección.

Por otro lado, las transformaciones en los triángulos son simples. Al momento de aplicarle una traslación, una rotación o un escalamiento, se multiplican los tres puntos del triángulo por una matriz determinada por la acción a tomar, y luego se recalculan las demás variables como las normales y los puntos de mapeo.

2.3.3 Cubos

Para implementar el cubo decidimos usar doce triángulos. De esta forma, el problema de la intersección queda resuelto. Por otro lado, para realizar las transformaciones de la primitiva, se utiliza el mismo concepto, se transforman los doce triángulo uno a uno.

Por definición, un cubo es de lado 1 y está en el origen de coordenadas, y mediante las transformaciones, podemos moverlo hacia otro punto de la escena, rotarlo y escalarlo.

En el caso de las texturas, considerando que los cubos en sí, están formados por 12 triángulos, 2 por cada cara del mismo. Al ser en sí triángulos, la textura ya esta resuelta. El único obstáculo a superar fue la correcta ubicación de los triángulos para que conicidan las texturas.

2.3.4 Planos

También implementamos planos. Para hacerlo, lo definimos como un punto y una normal. De esta forma, el cálculo de la intersección de un rayo con el mismo se simplifica. El cálculo utilizado es el mismo que para los triángulos, pero sin determinar si el punto pertenece al al mismo.

También es posible mapear texturas al plano. De igual forma que con los triángulos se crean 2 vectores utilizando el punto con el que se construye el plano y la normal del mismo. Con dichos vectores se toman los componentes u y v para el mapeo contra la textura.

Pero surge un inconveniente, el plano tiene largo infinito, por lo que no se puede tomar ninguna referencia como un punto máximo o mínimo para corresponder con el máximo y mínimo de la textura. La solución que encontramos para esto, fue hacer una repetición de la textura por cada cierto tamaño fijo.

2.3.5 Figuras Complejas

Para realizar figuras complejas, utilizamos las mallas de triángulos. Mediante la creación de muchos triángulos, es posible generar objetos que no pueden formarse como primitivas en sí. Un ejemplo claro de esto, es el el alambre o la torre de agua que podemos ver en las figuras 5 y 6.

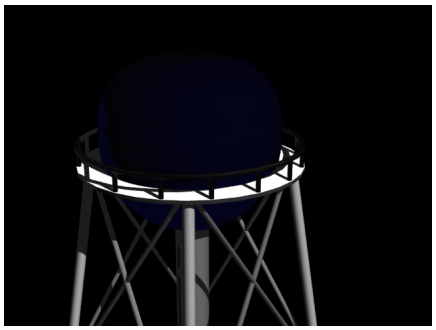

Figure 5: Torre de Agua.

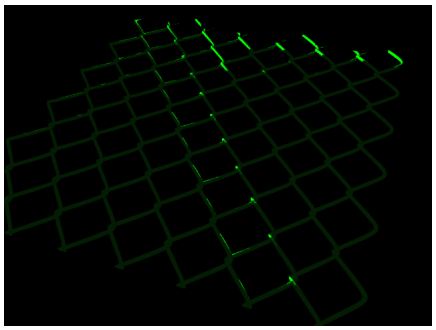

Figure 6: Alambre.

En el caso de los mapeos de las texturas, este punto queda resuelto al igual que en los triángulos, ya que estas figuras complejas son conjuntos de triángulos.

3 Motor de Ray Tracing

3.1 Anti Aliasing

El *anti aliasing* fue resuelto con la creación de varios rayos por cada pixel en la imagen. Dos parámetros determinan la cantidad y disposición de los rayos. Se toma el mayor número del parámetro *aa* para saber la potencia de 2 subdivisiones horizontales y verticales para cada pixel, es decir, si el número es 2, se deben crear 2 por 2 zonas dentro del pixel en las que se deben tirar tantos rayos como el parámetro *samples* determine.

Los rayos creados se lanzan en forma estocástica dentro de los límites creados por las subdivisiones para darle un nivel de realismo a la imagen generada.

En la figura 7 observamos la diferencia entre usar *anti alisasing* y no usarlo. Además, en el figura 8 observamos cómo se subdivide el espacio y por cada división disparamos más rayos que en caso de no utilizar *anti aliasing*.

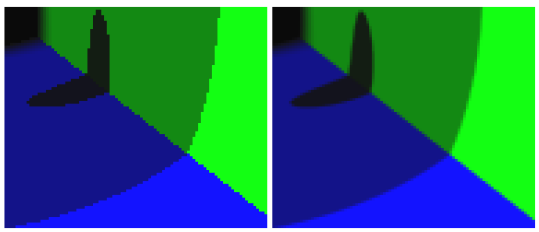

Figure 7: Ejemplo de Anti Aliasing

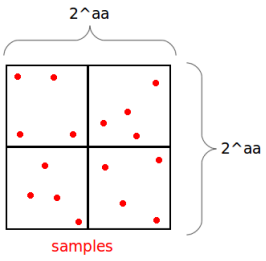

Figure 8: División del Pixel

3.2 Sombras

Cómo explicamos en la sección 2.2, las sombras se determinan si existen objetos entre la luz y el punto. En caso de haber un objeto bloqueando la luz, ese pixel

quedará sombreado, en cambio, si no hay objetos entre la luz y el punto, el pixel no estará sombreado.

Decidimos implementar *soft shadows*, que suaviza un poco las sombras en los puntos cercanos a los expuestos a la luz. Para ésto se crearon luces adicionales alrededor de cada luz de la escena, de forma tal que se generen varias sombras a partir de una luz y así dar un efecto de una sombra más suave. Claramente, esto aumenta el tiempo de rendering, ya que aumenta la cantidad de luces en la escena.

En las figuras 9 y 10 observamos un ejemplo de sombras suaves y sombras simples respectivamente.

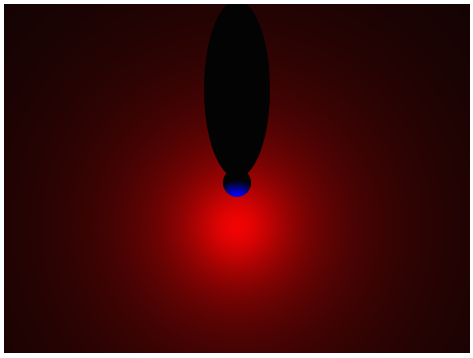

Figure 9: Sombras suaves.

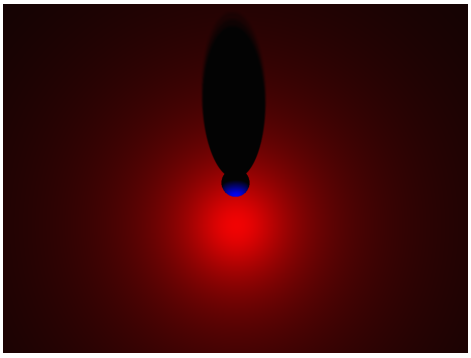

Figure 10: Sombras simples.

3.3 Shaders

Implementamos cuatro shaders distintos, *constant*, *phong*, *glass* y *mirror*. Decidimos implementar uno más que los requeridos para poder probar las escenas de prueba de la cátedra, ya que muchas de ellas no contaban con un shader de los pedidos definido. A continuación explicamos cada uno de ellos.

3.3.1 Constant

Se implementó el shader *constant* para poder representar las escenas de prueba entregadas por la cátedra. Este shader simple consta de un color que esta distribuido de forma uniforme en la superficie de la primitiva.

3.3.2 Phong

En el caso de querer utilizar una textura sobre una figura, hay que utilizar *phong*, aunque también se le puede especificar un color difuso en vez de una textura por si se requiere. Este shader cuenta con una imagen que luego es utilizada para mapear a los distintos objetos. También cuenta con un coeficiente de especularidad, que determina cuánto influye el valor especular de la superficie en el color de reflejo de la misma.

En la figura 11 se puede observar una esfera con una textura, la tierra en este caso. Además, podemos observar como tiene un cierto reflejo, dado por el coeficiente de especularidad asignado en este caso.

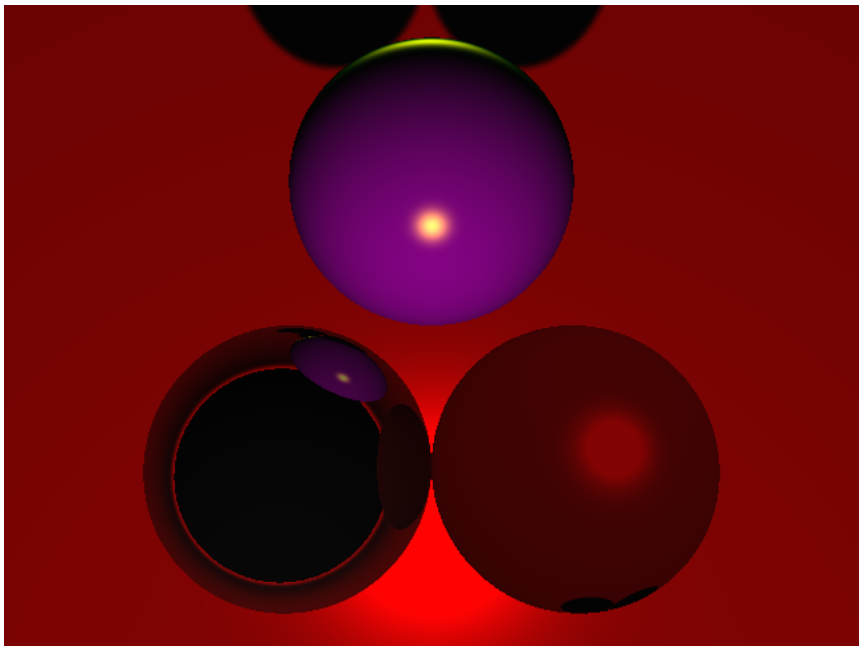

Figure 11: Esferas texturadas utilizando *phong*.

3.3.3 Glass

Este tipo de shader lo utilizamos cuando queremos que la figura en cuestión sea traslucida. Para este tipo de shader hay que definir un índice de refracción que será utilizado para calcular la dirección en la que sale el nuevo rayo una vez que intersecta.

En la figura 12 se puede observar una serie de figuras con unos planos de fondo. Observamos claramente que las esferas son traslucidas y se puede ver detrás de ellas.

3.3.4 Mirror

Este tipo de shader lo utilizamos cuando queremos que el objeto refleje como un espejo. En este caso debemos definir un coeficiente de reflexión ya que los rayos, al golpear serán reflejados en lugar de atravesar la figura.

En la figura 12 podemos observar una esfera que utiliza como shader un mirror.

4 Optimizaciones y Opcionales

Decidimos implementar algunas optimizaciones al algoritmo para hacerlo más eficiente. Tuvimos que implementar un sistema de subdivisión espacial y de volúmenes envolventes y fue por eso que elegimos *octree*.

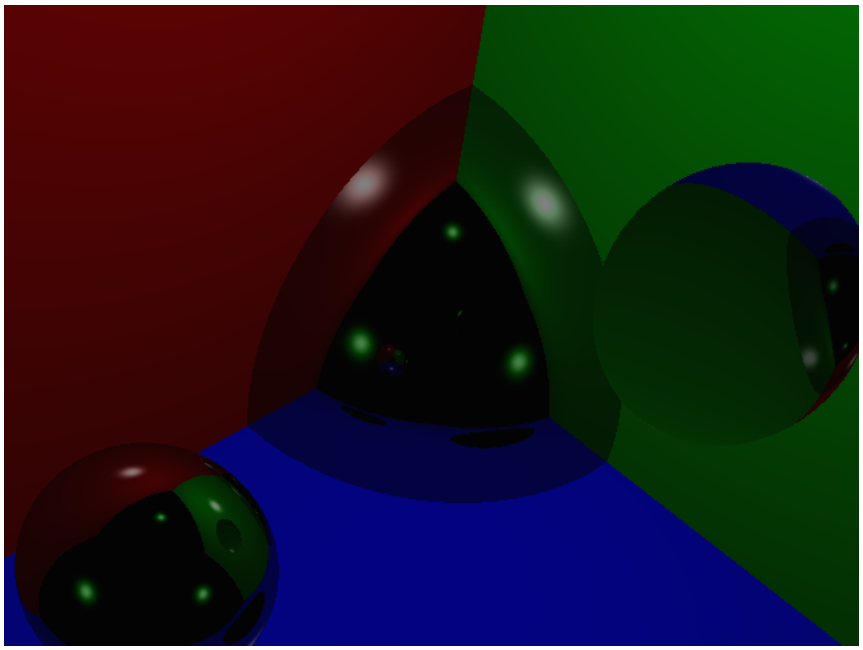

Figure 12: Esferas utilizando *glass*, *mirror* y planos.

4.1 Subdivisión Espacial y Volúmenes Envolventes

Como método de subdivisión espacial se utilizó octree. Para ello se definió su nodo y contenido. El nodo representa un cubo en el espacio (como parte de su subdivisión espacial) y su contenido es una lista de primitivas contenidas por el mismo. Investigando como funciona el octree vimos que este método de subdivisión espacial en definitiva nuclea en volúmenes envolventes AABB (sus nodos) a un conjunto de primitivas. Es por ello que para determinar cómo recorrer el árbol en busca de intersecciones con primitivas, se debe chequear en cada paso intersecciones con un cubo AABB representado por un nodo.

Existen algunas decisiones adoptadas que debemos mencionar. Primero existe el hecho de que nuestro octree respetó la estructura básica de subdivisión en cubos (ancho del lado igual para los tres ejes). En segundo lugar debe mencionarse que como paso previo al armado del octree se recorren las primitivas en busca de los extremos para los ejes y poder determinar la longitud inicial de los lados para la raíz. Por último, los planos por ser de longitud infinita, no se encuentran dentro de la estructura del octree y se analizan por separado (se tiene en cuenta que meterlos en el octree haría que el árbol sea más ineficiente y que en general las escenas tienen mucho menos planos que polígonos).

4.2 Multi Threading

Implementamos también multiples hilos de ejecución. Esto optimiza el poder de los procesadores de varios núcleos de hoy en día y agiliza al motor de ray tracing.

En la sección 5 podemos observar las mejoras respecto a la ejecución con

solo un hilo.

4.3 Progress Bar

Se implementó para informar el progreso del render, un informe progresivo de la cantidad de trabajos o *buckets* realizados en forma de porcentaje completado. Esto ayuda en gran medida a la hora de realizar renders de escenas complejas en cantidad de primitivas o efectos sobre las mismas.

5 Benchmarking

Escena	Tiempo con Octree	Tiempo sin Octree
bunny.sc	11 Seg.	Mayor a 20 Min.
chainlink.sc	Mayor a 20 Min.	Mayor a 20 Min.
hand.sc	7 Min.	Mayor a 20 Min.
marbles.sc	23 Seg.	Mayor a 20 Min.

6 Conclusiones

Del desarrollo del motor concluimos que, de manera relativamente simple, es posible realizar renders de imágenes de muy alta calidad en tiempos razonables. Los tiempos razonables son debido a las optimizaciones implementadas, dado que sin ellas los tiempos eran muy mayores.

También se demuestra a las claras que las técnicas de anti aliasing y sombras suaves ayudar a mejorar aún más la imagen, que se presentá con menos brusquedad en los cambios de color entre una figura y la otra. El refinamiento de los pixeles de la imagen retarda un poco más el render, pero el resultado es bastante mejor que no usarlo.

Por supuesto, hay que considerar que todas estas cuestiones se engloban bajo la disyuntiva entre el tiempo insumido y la calidad que queremos obtener. A más calidad, más tiempo será necesario y lo inverso también se cumple.