Computación Gráfica

Trabajo Práctico 1 - Ray Tracing

Yet Another Ray Tracer

Grupo 3

Del Giudice Gustavo
Menzella Facundo
Noriega Jose

Indice

Las referencias a las escenas utilizadas, se encuentran el final del documento.

1.Estructuras de aceleración

El trabajo práctico utiliza un ***SAH KD-tree*** como estructura de aceleración principal puesto que presenta muchas ventajas con respecto de las otras estructuras mencionadas en clase. En un primer momento se implementó un KD-tree simple dividiendo por los ejes por la mitad, pero se migró a la implementación final ya que la performance no era la esperada.
Vale la pena remarcar que el algoritmo para la construcción también pasó por muchas etapas. Se comenzó con un armado básico ($O(n^2)$) para terminar con el armado por eventos ($O(n\ log\ n)$).

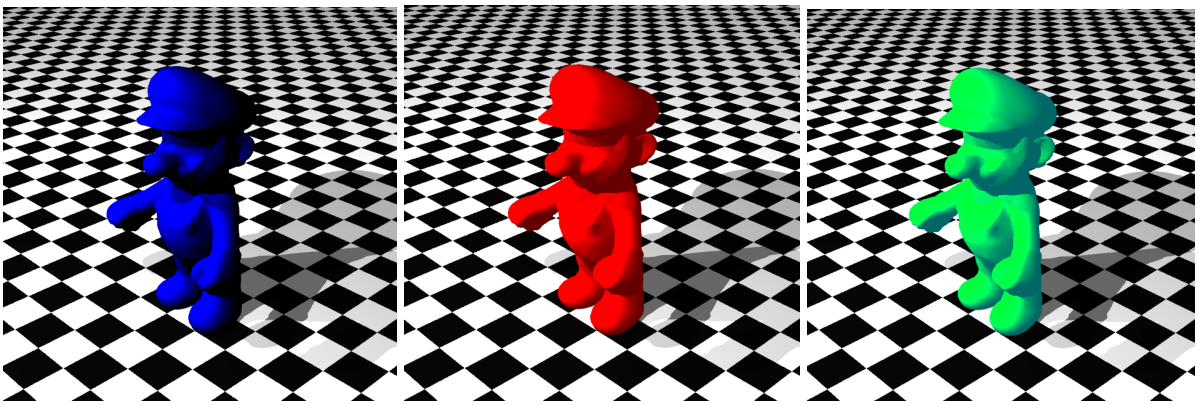
2.Decisiones de implementación

Desde sus comienzos el ray tracer intentó mantener su diseño orientado a objetos. Esto nos daba claridad y orden a la hora de trabajar. Por otro lado, las consecutivas llamadas a los métodos nos daba mucho overhead. Por eso se terminó optando por copiar y pegar código, por ejemplo, en la suma de vectores.

3.Materiales implementados

NOTA: Las siguientes imágenes contienen una luz ambiental, dos luces puntuales, proyección de sombras simples, profundidad de rayo de 4 y antialiasing de 4 samples. Objetos: un mesh y un plano infinito texturizado.

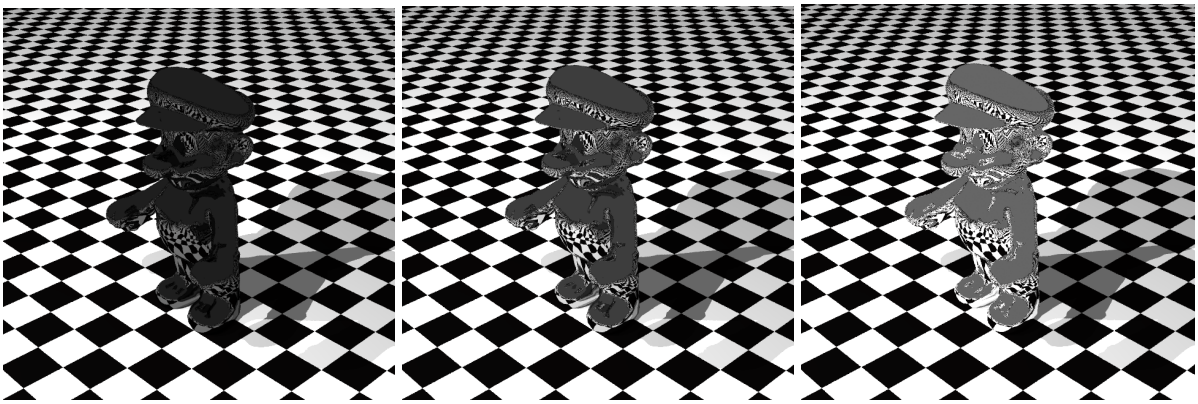
3.a.Matte



Imágenes 1, 2 y 3 con color azul, rojo y verde respectivamente.

El material ***Matte*** utiliza un sombreado ***Lambert***. Posee una componente difusa y una ambiental. Nuestro render permite asignarle la intensidad de la componente ambiental mediante el parámetro *Ka* que le daría "*color*" a las partes sombreadas. Pero como las escenas de Lux Blender, no poseen este parámetro, se le asigna *Ka = 0* y básicamente el color del sombreado depende de la luz ambiental.

3.b.Mirror



Imágenes 4, 5 y 6 con un Kr de 0.3, 0.6 y 1 respectivamente.

El material ***Mirror*** se implementó con la clase ***Reflective***. Esta clase hereda de ***Phong***, para darle un reflejo especular. Pero como Lux Blender no nos pasaba ninguna información acerca de eso, todos los parámetros relacionados al reflejo especular se ponen en 0. El parámetro Cd es negro y el Cr blanco por default, al menos que la escena especifique lo contrario. Su componente *"perfect specular"* (reflejo perfecto) utiliza el parámetro *Kr* (texturizable) que representa la "intensidad" del reflejo. A medida que aumenta, se va pareciendo a un reflejo perfecto.

3.c.Glass

Imágenes 7, 8 y 9 con un Kt de 0.3, 0.7 y 1 respectivamente. Índice de refracción: 1.5, valor default de Lux

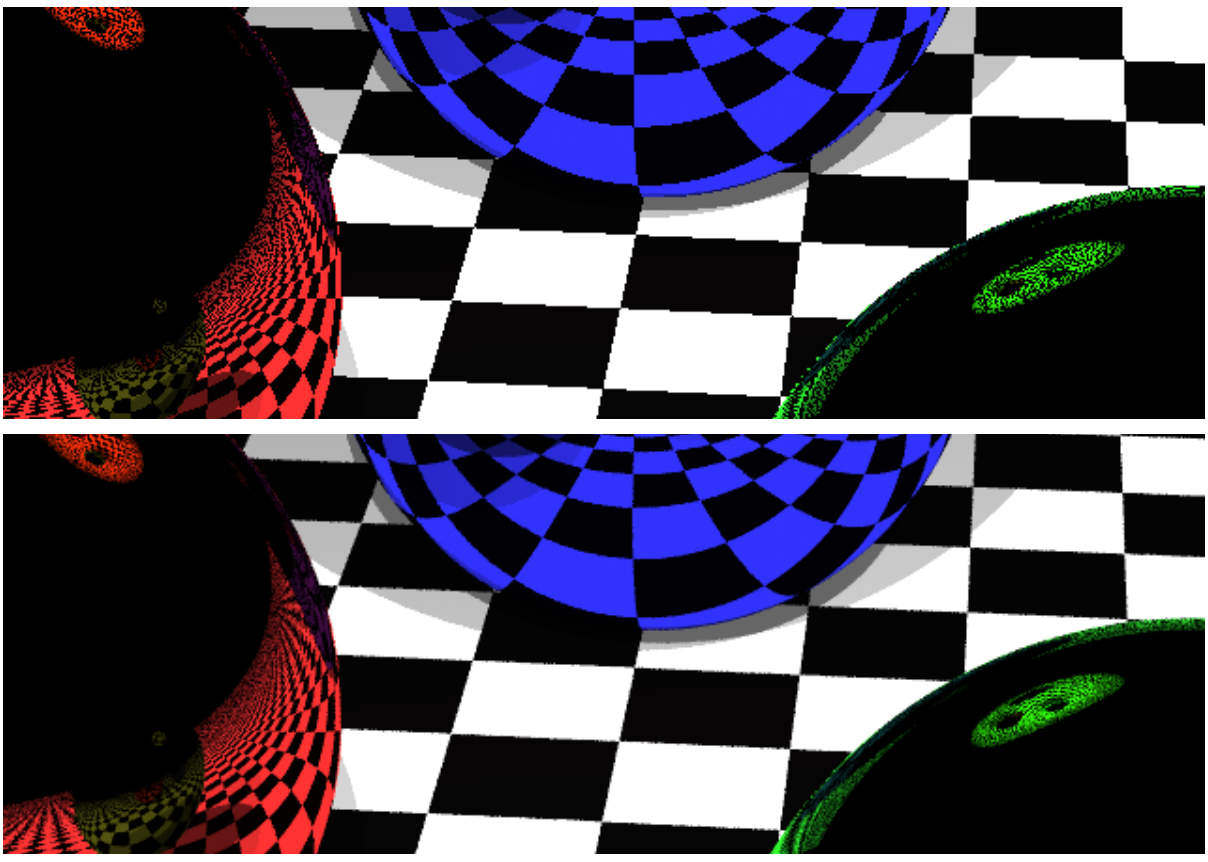
El material ***Glass*** se implementó con la clase ***Transparent***, que posee una componente de transmitancia perfecta. Cómo hereda de Phong, se aplica lo mencionado anteriormente en el material ***Mirror***. El parámetro *Kt* es análogo a *Kr*, representando la "*intensidad*" del reflejo.

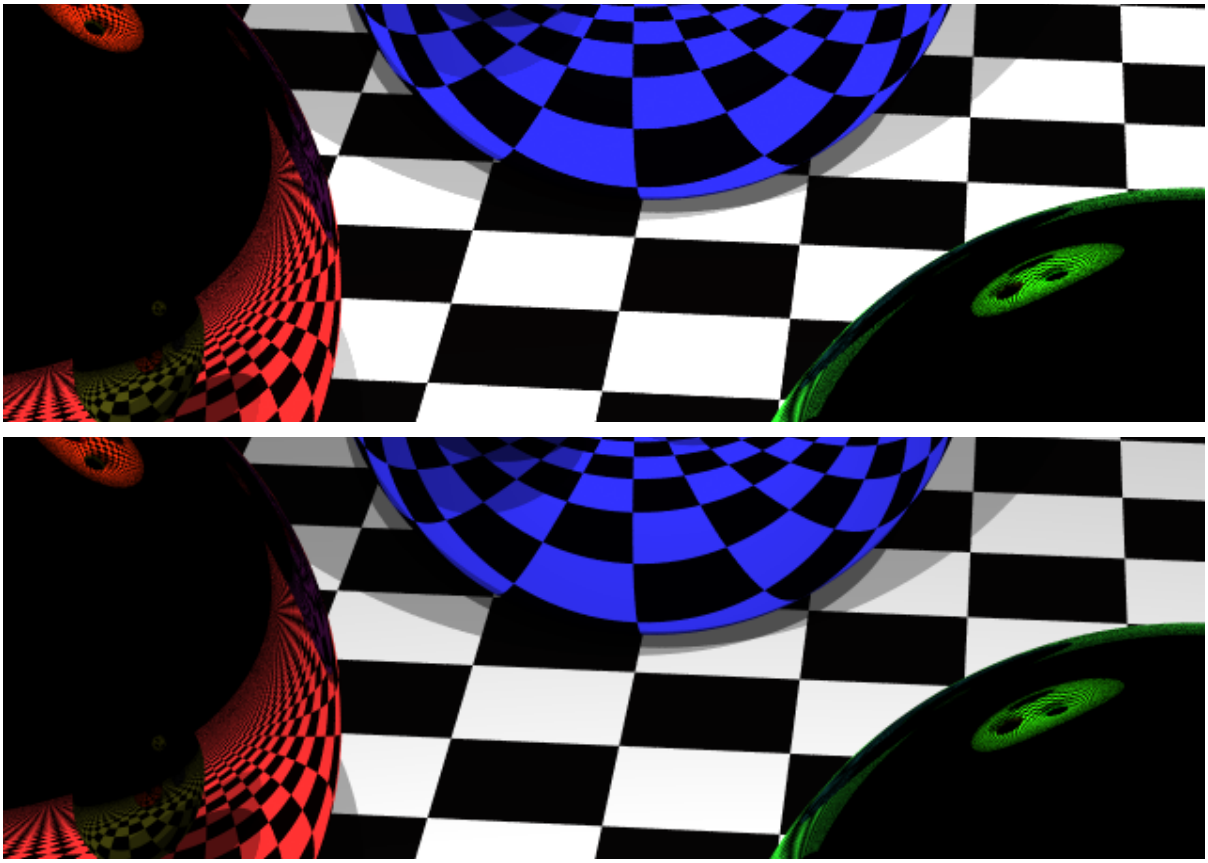
3.d.Metal2

Imágenes 10, 11 y 12 con roughness de 0.1, 0.5 y 0.9 respectivamente.

Por último, ***Metal2*** se implementó usando Reflective. Pero en este caso si utilizamos las propiedades de Phong. Al igual que ***Matte***, posee una componente ambiental (la cual resuelve como Matte) y una difusa. Pero también posee una componente “glossy specular” para el reflejo especular. Lux Blender nos daba *“vroughness”* y *”uroughness”*, entonces se decidió tomar el menor de ambos, para aproximar el exponente de la forma E=1/roughness, y el parametro *Kt = (1 - roughness)/2*. Esto es consistente con las imágenes, ya que a medida que aumenta el roughness, se pierde el reflejo.

4.Antialasing





Imágenes 13, 14, 15 y 16 con muestreo de 1, 4, 16 y 32 respectivamente.

Para antialiasing se probaron 3 tipos distintos de muestreo (sampling); ***regular***, ***random*** y ***jittered***. El muestreo ***regular*** es un muestreo uniforme, aunque preciso, puede generar patrones de Moiré, en especial con texturas pequeñas y/o que se extienden en el infinito, quitándole precisión a la representación de la imagen. Después se implementó el muestreo ***random*** (aleatorio), que en casos de borde, reemplaza por ruido, resultando en una imagen más acertada. Pero como el muestreo de distribuye de forma aleatoria, se puede perder la uniformidad del mismo, dejando “agujeros”. Por eso se eligió como implementación final, el muestro ***jittered***, ya que este hace un muestreo uniforme, manteniendo cierta aleatoriedad. De esta forma, se obtiene lo mejor de ***regular*** y ***random***.

5.Características adicionales opcionales

5.a.Canales Texturizados

En un principio, nuestros parámetros *Kd, Kr y Kt* solo reciben valores numéricos entre 0 y 1. Estos representan distintos niveles de “intensidad”, siendo *Kd* la del color, *Kr* la de refleccion y *Kt* la de transmitancia. Estos valores se “*multiplicaban*” con distintos colores, para modificar su intensidad. Actualmente, al recibir texturas, permiten modificar niveles de intensidad a los distintos canales de un Color. También, con las texturas correctas, se puede “decidir” que partes reflejar y que partes no, como se ve en la siguiente imagen.

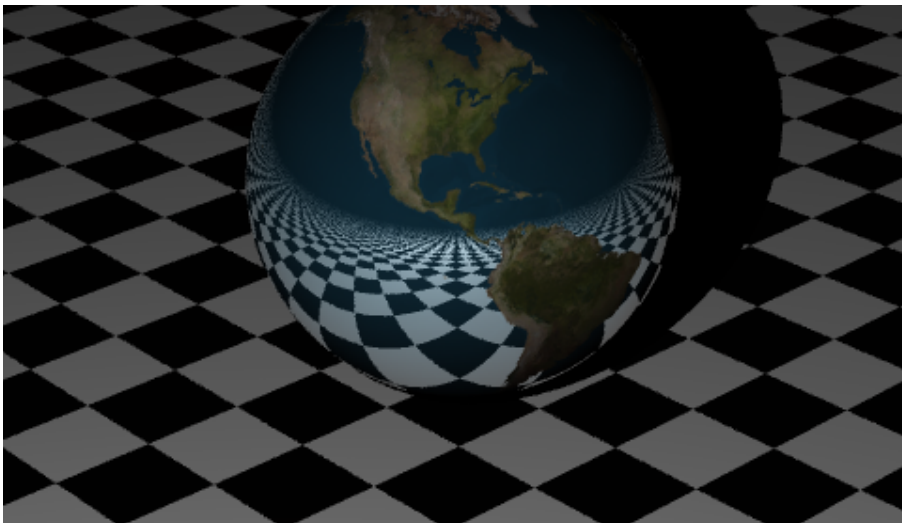


Imagen 17, con el parámetro Kr texturizado.

6.Parámetros de consola

Se implementaron todos los parámetros de consola pedidos por la cátedra. Adicionalmente, se implementaron los siguientes parámetros:

-h
Imprime la lista de parámetros aceptados

-g
(Opcional) Muestra el progreso del render en una ventana. No puede utilizarse en un benchmark.

-th <número de threads>
(Opcional) Especifica la cantidad de threads que se usarán para el render.

-bs <bucket size>
(Opcional) Especifica el tamaño de bucket a usar en el render.

7.Parseo de escenas

Originalmente, se realizaba el parseo de escenas en dos pasadas: la primera leía el archivo y generaba una estructura de árbol con los atributos, identificadores y propiedades en el archivo. La segunda pasada recorría el árbol y configuraba la escena.

Sin embargo, esto resultó ineficiente, y poco mantenible. Se decidió utilizar una estrategia similar a la que usa LuxRender para sus escenas. En nuestra implementación, se cuenta con dos clases: un Parser, y un Builder. El parser se encarga de leer el archivo y encontrar identificadores y atributos. Cada identificador o atributo se corresponde con una

función del Builder, el cual mantiene el estado de la escena que está armando. De esta forma, no es necesario armar una estructura de árbol, y el parseo se realiza en una única pasada.

Aunque esta estrategia fue lo suficientemente eficiente para nuestras necesidades, el parser cuenta con algunas limitaciones. Por cuestiones de tiempo, se tuvo que tuvo que simplificar el parseo, de modo que pueda leer escenas exportadas con LuxBlend. Por lo tanto, tiene las siguientes limitaciones:

- No hay soporte para atributos ObjectBegin/ObjectEnd ni para ObjectInstance. ObjectBegin funciona de la misma forma que AttributeBegin. Tampoco pueden nombrarse sistemas de coordenadas dentro de TransformBegin/TransformEnd.
- El nombre del identificador o atributo debe estar en la misma línea que todos sus parámetros. Ejemplos:
    - Correcto: `LookAt 0 200 50 0 0 0 0 1`
    - Incorrecto: `LookAt`
      `0 200 50`
      `0 0 0`
      `0 0 1`
- El nombre del identificador no puede estar en la misma línea que sus propiedades. Ejemplos:
    - Correcto: `Film “fleximage”`
      `“integer xresolution” [800]`
    - Incorrecto: `Film “fleximage” “integer xresolution” [800]`
- Se puede poner más de una propiedad en una misma línea, pero toda propiedad debe estar contenida dentro de una misma línea. Ejemplos:
    - Correcto: `“integer xresolution” [800] “integer yresolution” [600]`
    - Incorrecto: `“integer xresolution”`
      `[800]`

8.Benchmarks

Los renders en esta sección fueron generados en una PC con las siguientes características:

- **Sistema operativo**: Debian 8 de 64 bits
- **CPU**: Intel Core i3-2330M Quad Core @ 2.20Ghz
- **RAM**: 4GB

Además, a menos que se especifique lo contrario, se utiliza antialiasing de 4 muestras, y ray depth de 4 rebotes.

Para todos los renders se corrieron benchmarks de 5 pasadas. El tiempo en esta tabla corresponde al tiempo promedio de todos los renders, sin tener en cuenta el tiempo de carga de la escena y generación del árbol.

Las imágenes finales se encuentran en la carpeta “images”, junto con el tiempo promedio.

Materiales		**Antialiasing**		**Triángulos**	
Imagen	**Tiempo promedio**	**Imagen**	**Tiempo promedio**	**Imagen**	**Tiempo Promedio**
1	1m 58s	13	852ms	18	3m 40s
2	1m 59s	14	2202ms	19	6m 52s
3	2m 1s	15	8s	20	17m 8s
4	2m 24s	16	12s		
5	2m 29s				
6	2m 26s				
7	2m 15s				
8	2m 12s				
9	2m 15s				
10	2m 25s				
11	2m 25s				
12	2m 25s				

9.Escenas de referencia

Aquí se encuentran las escenas que se corresponden a cada imagen

1. MatteBlue.lxs, 2 primitivas, 7k polígonos aprox.
2. MatteGreen.lxs, 2 primitivas, 7k polígonos aprox.
3. MatteRed.lxs, 2 primitivas, 7k polígonos aprox.
4. MirrorLowReflectivity.lxs, 2 primitivas, 7k polígonos aprox.
5. MirrorMediumReflectivity.lxs, 2 primitivas, 7k polígonos aprox.
6. MirrorHighReflectivity.lxs, 2 primitivas, 7k polígonos aprox.
7. GlassLowTransparency.lxs, 2 primitivas, 7k polígonos aprox.
8. GlassMediumTransparency.lxs, 2 primitivas, 7k polígonos aprox.
9. GlassHIghTransparency.lxs, 2 primitivas, 7k polígonos aprox.
10. MetalLowRoughness.lxs, 2 primitivas, 7k polígonos aprox.
11. MetalMediumRoughness.lxs, 2 primitivas, 7k polígonos aprox.
12. MetalHighRoughness.lxs, 2 primitivas, 7k polígonos aprox.
13. Spheres.lxs, 5 primitivas, 0 polígonos, AA 1
14. Spheres.lxs, 5 primitivas, 0 polígonos, AA 4
15. Spheres.lxs, 5 primitivas, 0 polígonos, AA 16
16. Spheres.lxs, 5 primitivas, 0 polígonos, AA 32
17. TexturedChannels.lxs, 2 primitivas, 0 polígonos
18. StanfordBunny.lxs, 1 primitiva, 69k polígonos aproximadamente
19. 2StanfordBunnies.lxs, 2 primitivas, 138k polígonos aproximadamente
20. 4StanfordBunnies.lxs, 2 primitivas, 276k polígonos aproximadamente

10.Bibliografía

Heuristic Ray Shooting Algorithms
http://dcgi.felk.cvut.cz/home/havran/DISSVH/dissvh.pdf

On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N
http://dcgi.felk.cvut.cz/home/havran/ARTICLES/ingo06rtKdtree.pdf

Ray Tracing with the BSP Tree
http://www.cs.utah.edu/~thiago/papers/BSP_RT08.pdf

Ray-Triangle Intersection Algorithm for Modern CPU Architectures
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.189.5084&rep=rep1&type=pdf

Ray Tracing from the ground up
http://www.raytracegroundup.com/

Physically Based Rendering, from theory to implementation. Second Edition. Matt Pharr & Greg Humphreys.