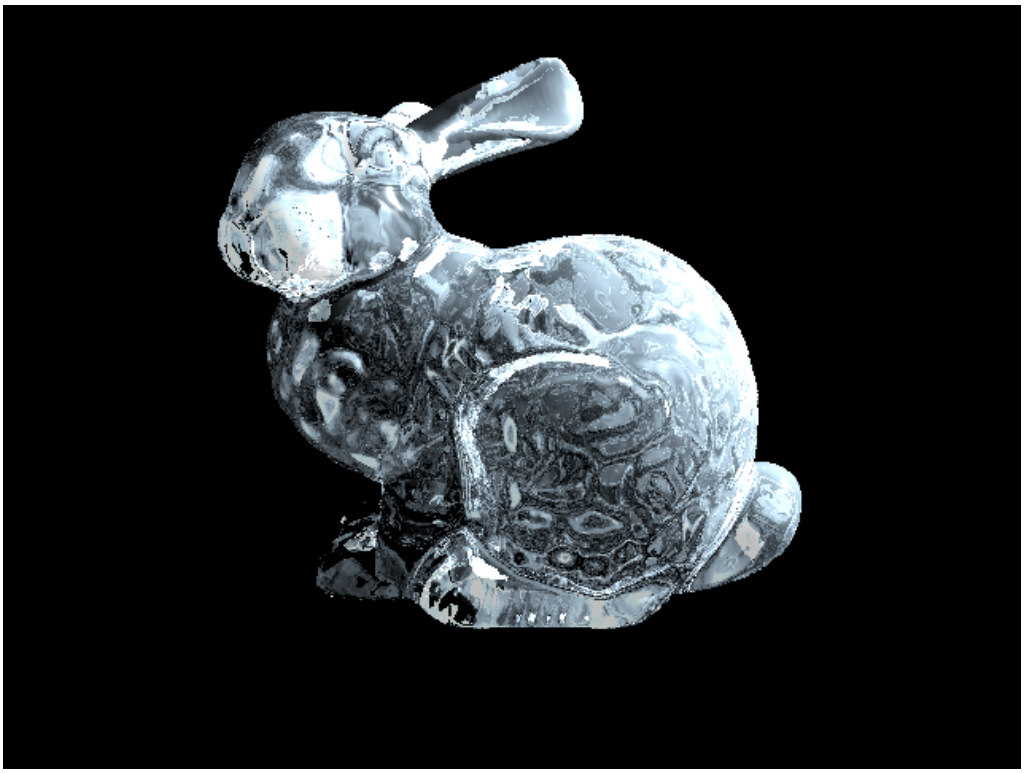
Jorge Mozzino y Matías Domingues

Grupo 5

Ray Tracing



Subdivisión espacial

Se decidió implementar el Kd-tree puesto que existen papers [2] que indican que dicha estructura es superior al resto y que además no parecía haber grandes diferencias con el resto en cuanto a la implementación.
En un principio, se implementó el recorrido del Kd-tree [3] y el armado básico que se realiza en $O(n^2)$ junto con la división en el punto medio. Se utilizó este árbol hasta que comenzó el rendereo de meshes de varios polígonos (10k+). En ese punto quedó claro que el armado del árbol en $O(n^2)$ no era suficiente ya que tardaba demasiado. Para mejorar esta situación, se implementó el armado del árbol en $O(n \log n)$

Requerimientos opcionales

Se implementaron los siguientes requerimientos opcionales:

- Canales texturizables: Tanto diffuse, specular y transparency son texturas. De esta forma se puede setear una textura, en vez de un color para lograr ciertos efectos.
- Spotlight: Se implementó una spotlight. El modelo usado está descripto en [6]. Como LuxRender no aclara qué exponente usa para el decay de la luz, se decidió usar como exponente la diferencia de ángulos que son parámetros de la spotlight.

Detalles de implementación

A continuación se detallan algunas decisiones tomadas.

Se decidió implementar el plano como un plano finito (i.e. cuadrilátero de dos dimensiones). Dicho plano se implementó como un mesh de dos triángulos rectángulos. Se tomó esta decisión por múltiples razones:

- Facilidad de implementación: como ya estaba implementado el mesh, implementar el plano como un mesh era natural y fácil.
- Eficiencia: las meshes utilizan un Kd-tree para calcular las intersecciones y las intersecciones de triángulos están optimizadas por lo que al implementarlo de esta forma queda más eficiente que un plano nativo.
- Conveniencia para la subdivisión espacial: los métodos de subdivisión espacial no se llevan muy bien con objetos “infinitos” y hay que tener en cuenta casos especiales para manejarlos. Consideramos, entonces, que no tener este tipo de elementos permite que el código quede más claro y más rápido.

Al igual que el plano, las cajas se implementaron como meshes de 12 triángulos (6 caras, 2 triángulos por cara).

Se utilizan vectores de 4 dimensiones para representar puntos y direcciones. Esto se hizo con un doble propósito:

- La traslación requiere de un cuarta dimensión si se desea realizar mediante multiplicación de matrices.
- Para utilizar las instrucciones SIMD del procesador. Sin embargo, por motivos que desconocemos, no fue posible lograr utilizar dichas instrucciones. A continuación se muestran el código Java y el código assembler generado

`x += other.x;`
`y += other.y;`
`z += other.z;`
`w += other.w;`
	`vmovsd 0x10(%rsi),%xmm0`
	`vmovsd 0x10(%rdx),%xmm1`
	`vaddsd %xmm1,%xmm0,%xmm0`
	`vmovsd %xmm0,0x10(%rsi)`
	`vmovsd 0x18(%rsi),%xmm0`
	`vmovsd 0x18(%rdx),%xmm1`
	`vaddsd %xmm1,%xmm0,%xmm0`
	`vmovsd %xmm0,0x18(%rsi)`
	`vmovsd 0x20(%rsi),%xmm0`
	`vmovsd 0x20(%rdx),%xmm1`
	`vaddsd %xmm1,%xmm0,%xmm0`
	`vmovsd %xmm0,0x20(%rsi)`
	`vmovsd 0x28(%rsi),%xmm0`
	`vmovsd 0x28(%rdx),%xmm1`
	`vaddsd %xmm1,%xmm0,%xmm0`
	`vmovsd %xmm0,0x28(%rsi)`

Esperábamos que se utilizaran las instrucciones v*pd que son las de packet en vez de las v*sd que son las de scalar.

La intersección de los triángulos se hace utilizando el algoritmo Moller-Trumbore [4]

Problemas encontrados

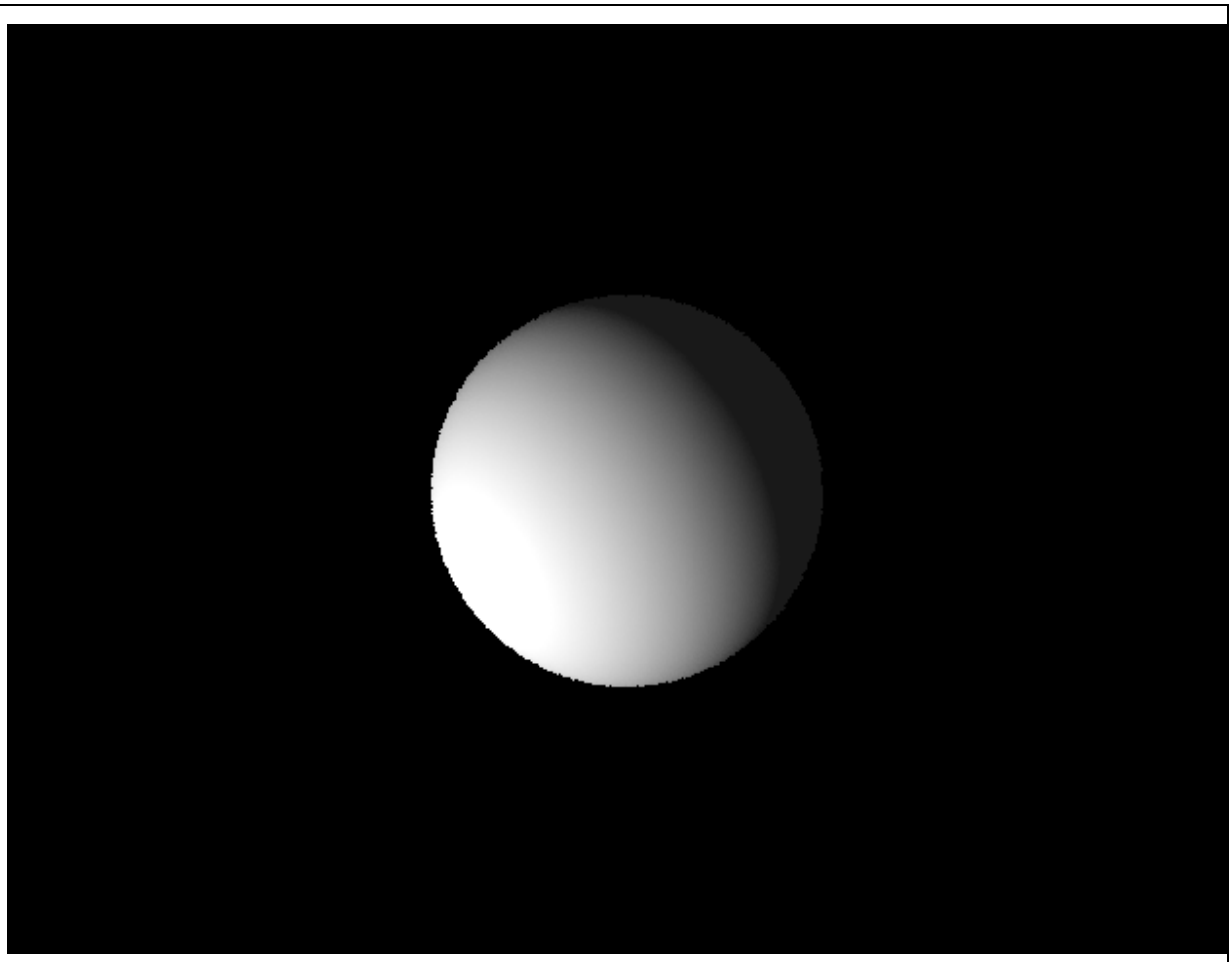
Hubo varios problemas numéricos al realizar el trabajo.

- Si bien se habia considerado sumar un epsilon al momento de lanzar rayos desde una superficie, detectamos un caso borde al recorrer el árbol que hacía que las superficies se vieran manchadas de puntos negros. Esto se daba porque, al recorrer el árbol, la superficie estaba exactamente a la distancia máxima, entonces la precisión, sucedía que a veces estaba antes y a veces después de la distancia máxima, por lo que se generaban las manchas negras. La solución adoptada fue agregar un epsilon a la distancia máxima.

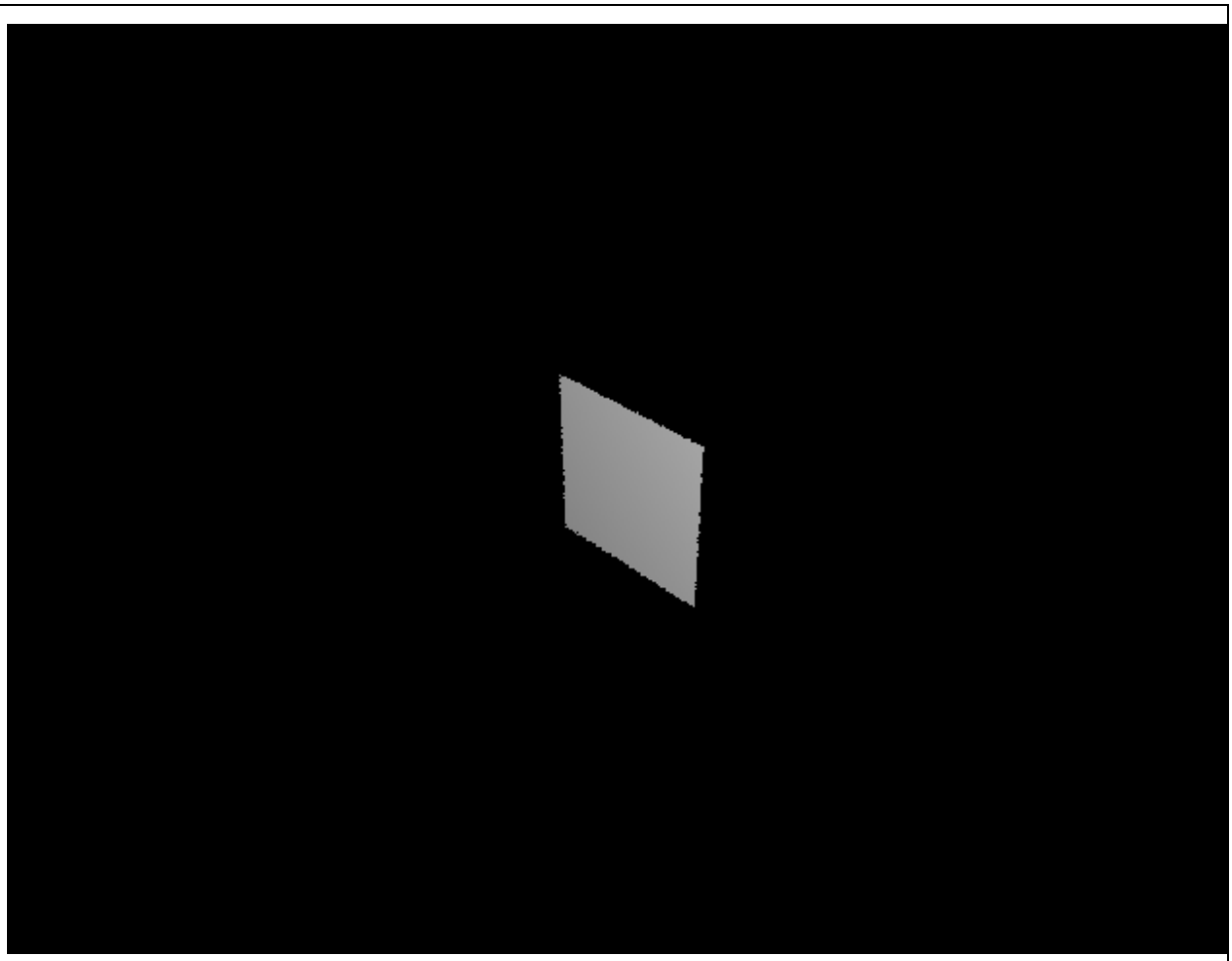
- Al hacer operaciones con colores, cuando se usa el exponente de phong, pasaba que los números daban infinito. Esto hacía que se vieran puntos negros en la imagen. Hubo que agregar el caso especial para manejar números muy grandes.

Fotos

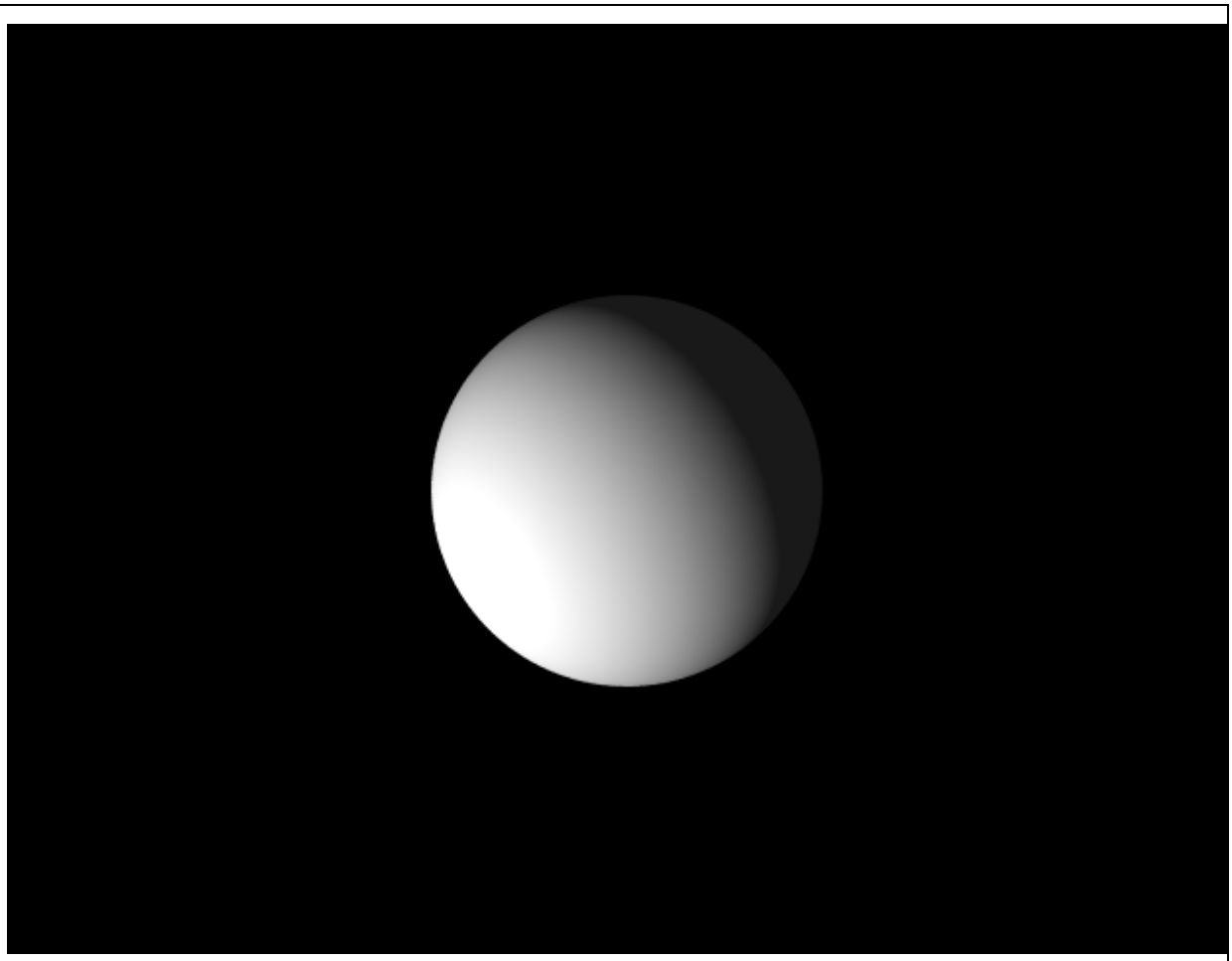
Las siguientes fotos fueron tomadas en una computadora con procesador Intel Core i5-5200U CPU @ 2.20 GHz



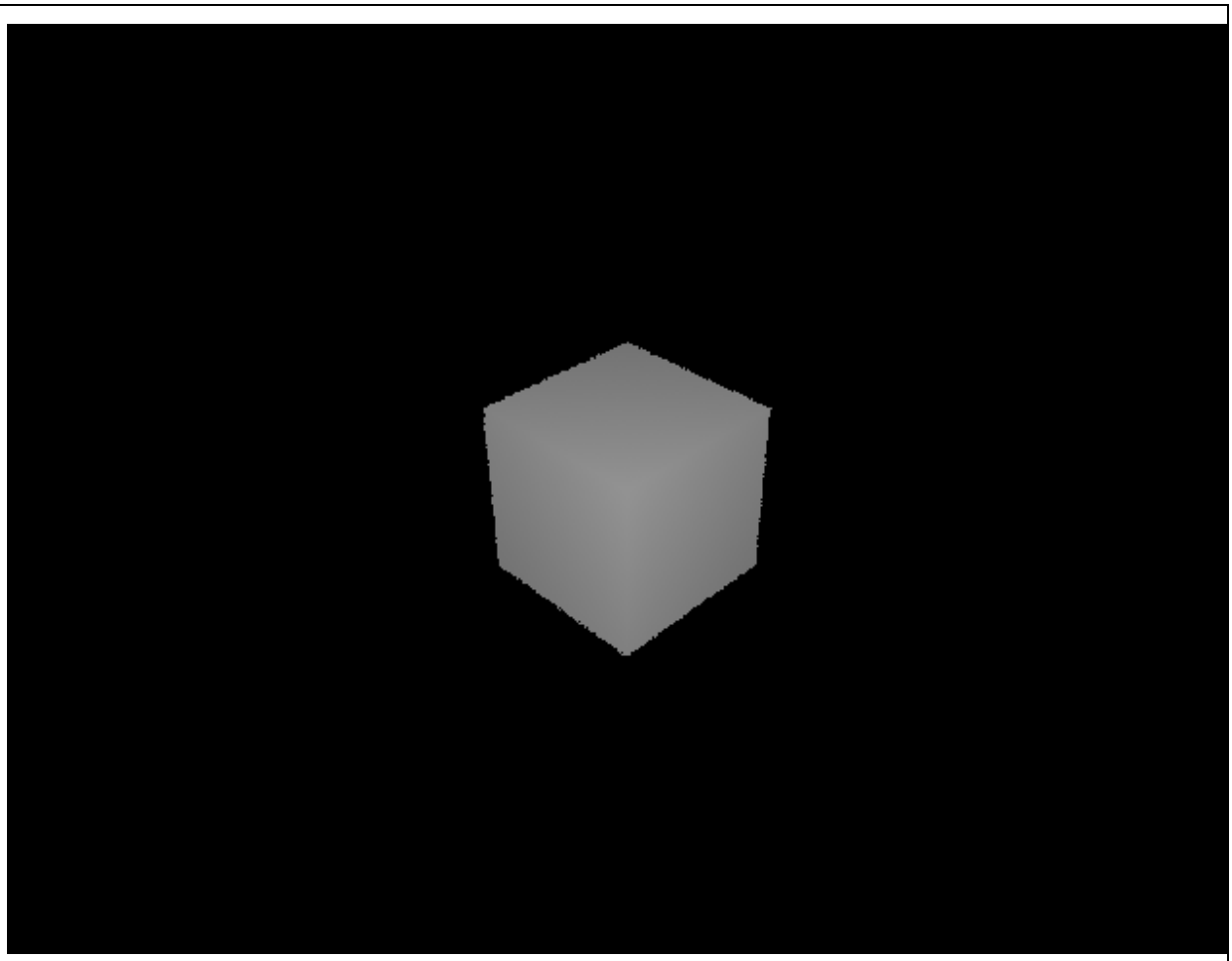
Primitiva de esfera con una luz puntual y con luz ambiental



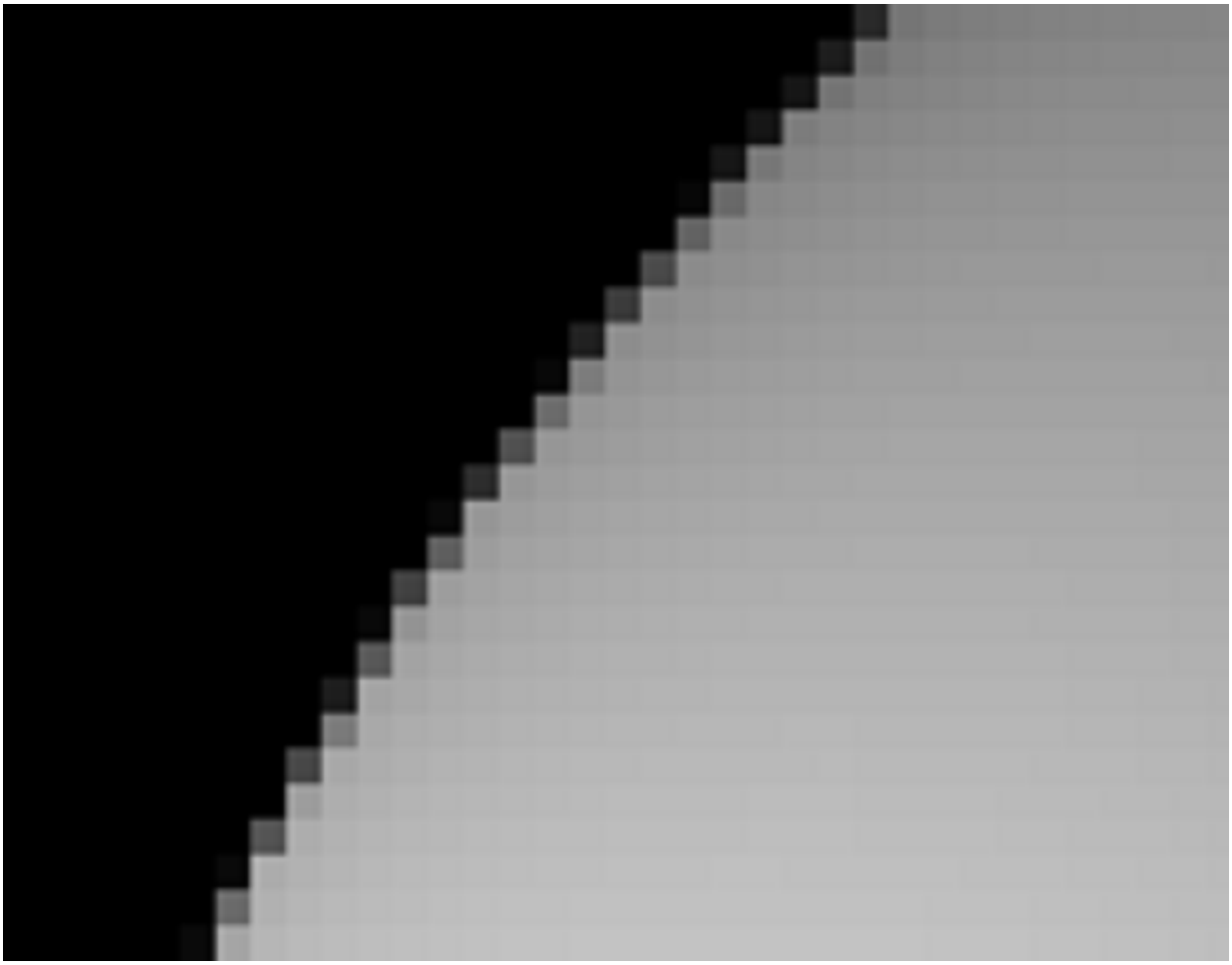
Primitiva de plano ubicada en (0,0,0) vista desde (3,3,3)



Primitiva de esfera con antialiasing

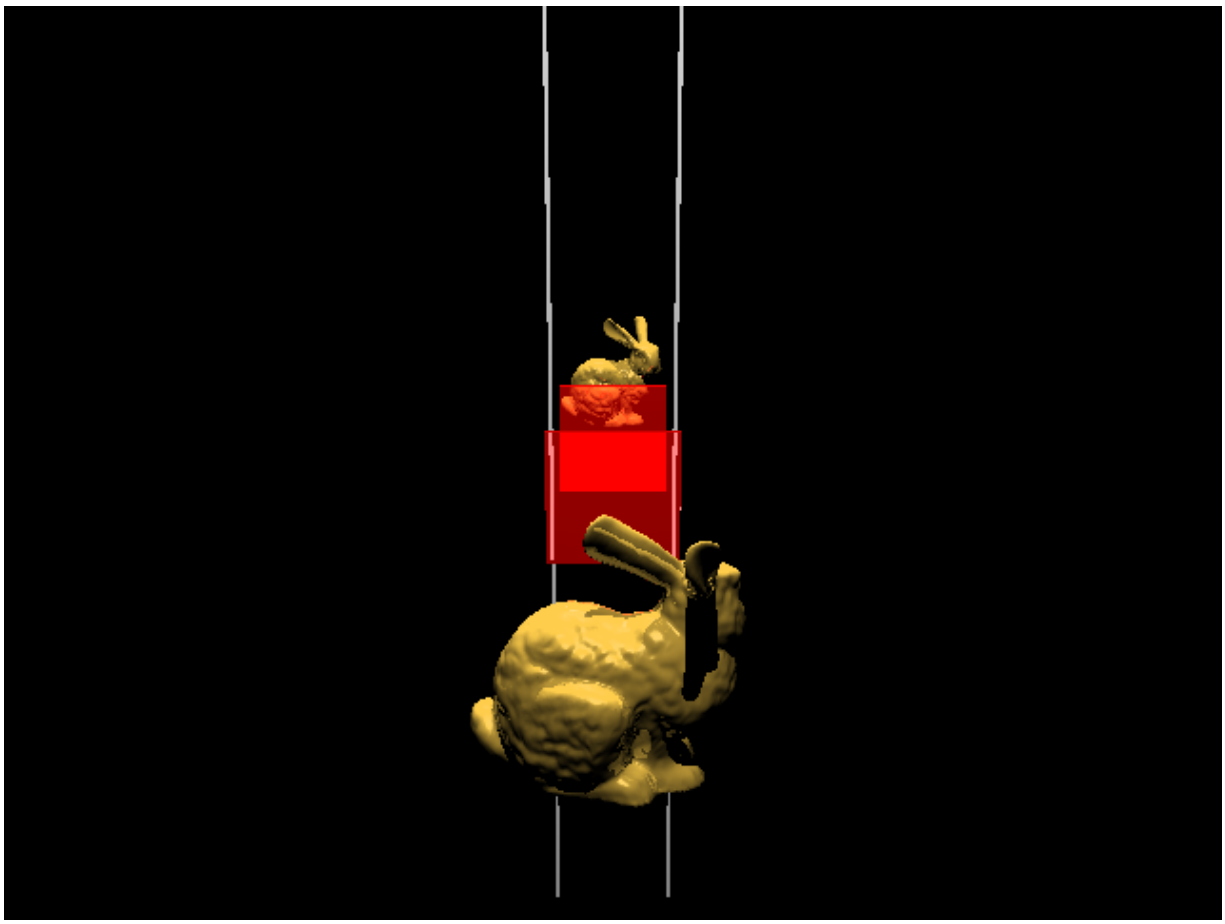


Primitiva de cubo ubicada en el (0,0,0) cuyos lados miden 1 y visto desde (3,3,3)

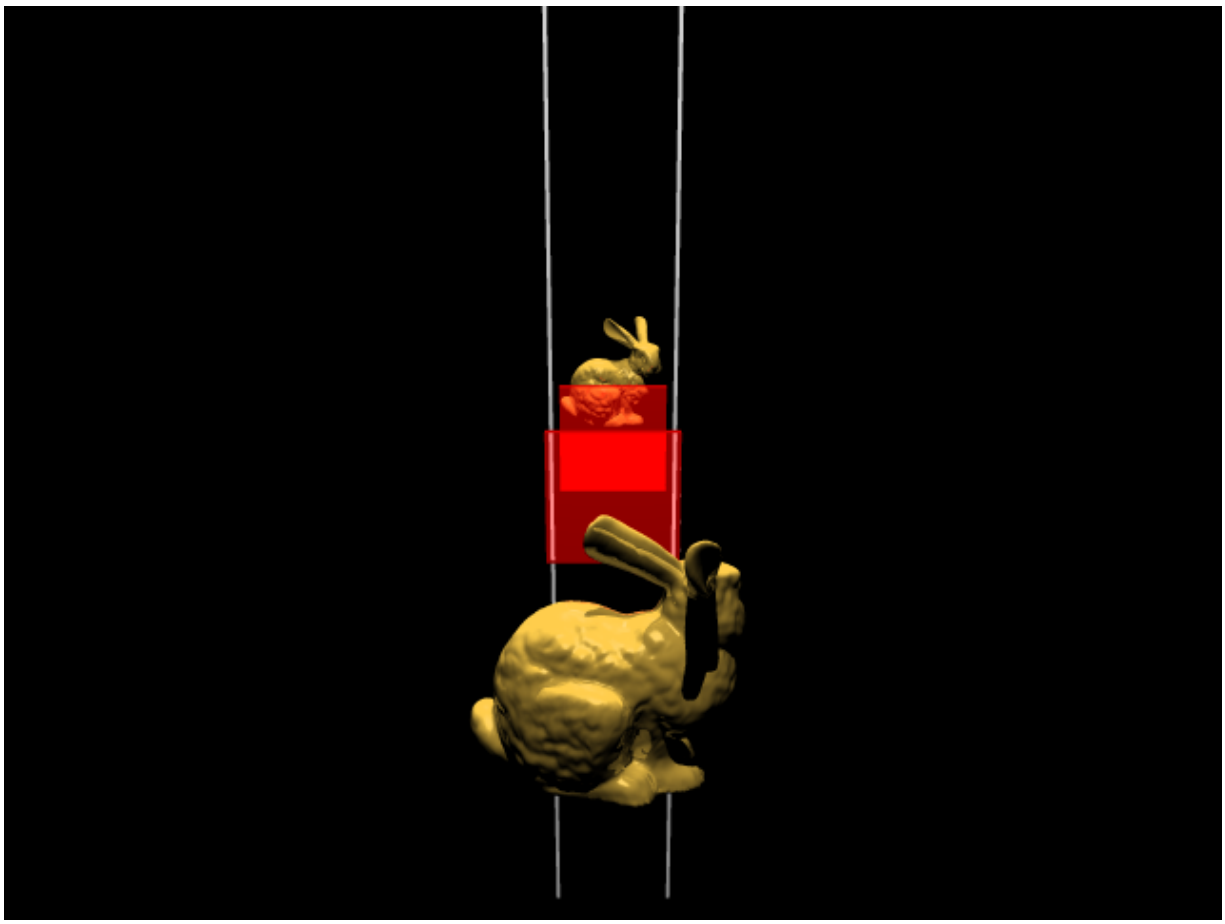


Vista zoomeada de como se ve el borde de la esfera cuando se utiliza antialiasing

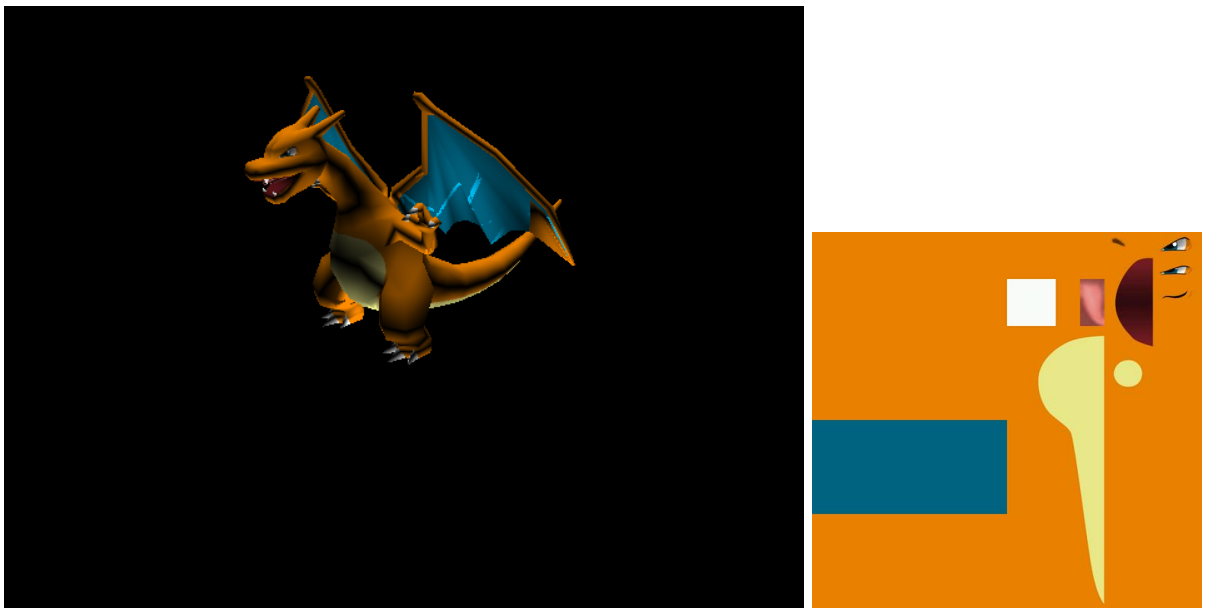
Una spotlight blanca pegándole a un plano blanco



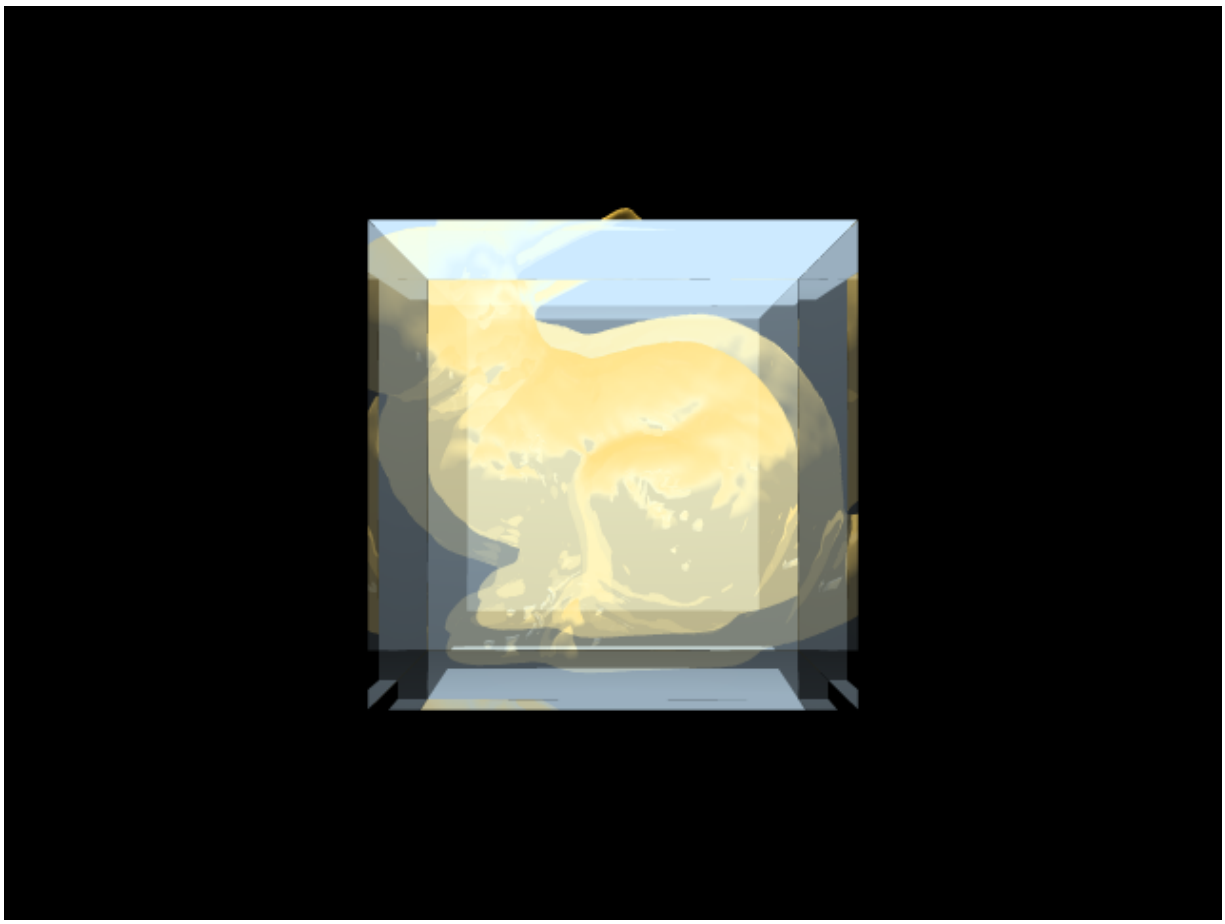
Bunny de oro - plancha de material transparente rojo - espejo. El marco del espejo se logró poniendo un plano blanco detrás del espejo. Hay 2 luces en la escena, una que le pega al bunny de frente y otra que le pega de atrás. Esta imagen se rendereó en 143 ms. La mesh del bunny es de 70k triángulos.



Misma imagen que la anterior pero rendereada con 16 samples de antialiasing. El rendereo tomó 1862 ms. Notar que el tiempo tomado es aproximadamente 16 veces el tiempo empleado en realizar el render con 1 solo sample.



Mesh de charizard con su textura. El rendereo de esta imagen tomó 106 ms



El bunny visto con una caja de vidrio adelante. La luz esta arriba. El tiempo de rendereo fue de 7000ms con un ray depth de 5 y 16 samples de antialiasing.

Mejoras futuras

Algunos temas que se podrían ver para mejorar la velocidad del raytracer pero que no se pudieron investigar a fondo debido a los límites de tiempo son los siguientes:

- Packet ray tracing: Consiste en realizar el recorrido del árbol con un conjunto de rayos, en vez de con un único rayo.
- Instrucciones SIMD: Reescribir el código de alguna manera, o agregar los flags correspondientes a la JVM para que utilice instrucciones SIMD que, como hemos mostrado, no parece estar generándolas.
- Constantes de SAH: En nuestro raytracer hemos utilizado las sugeridas por [1]. Se podría investigar y jugar un poco más con ellas a ver si encontramos un par de constantes que se adaptan mejor a las escenas que rendereamos.

Referencias

[1] Ingo Wald y Vlastimil Havran. 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, 2006.
[2] Vlastimil Havran. Heuristic Ray Shooting Algorithms, Dissertation Thesis. http://dcgi.felk.cvut.cz/home/havran/DISSVH/dissvh.pdf, 2000.
[3] M. Hapala y V. Havran. Review: Kd-tree Traversal Algorithms for Ray Tracing. http://dcgi.felk.cvut.cz/home/havran/ARTICLES/cgf2011.pdf, 2010
[4] Tomas Möller y Ben Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. http://www.graphics.cornell.edu/pubs/1997/MT97.pdf, 1997
[5] Kevin Suffern. Ray Tracing from the Ground Up, ISBN-10: 1568812728, 2007.
[6] https://www.classes.cs.uchicago.edu/archive/2003/fall/23700/docs/project-1.pdf