Computación Gráfica

Informe de Trabajo práctico Nº 1

Ray Tracing

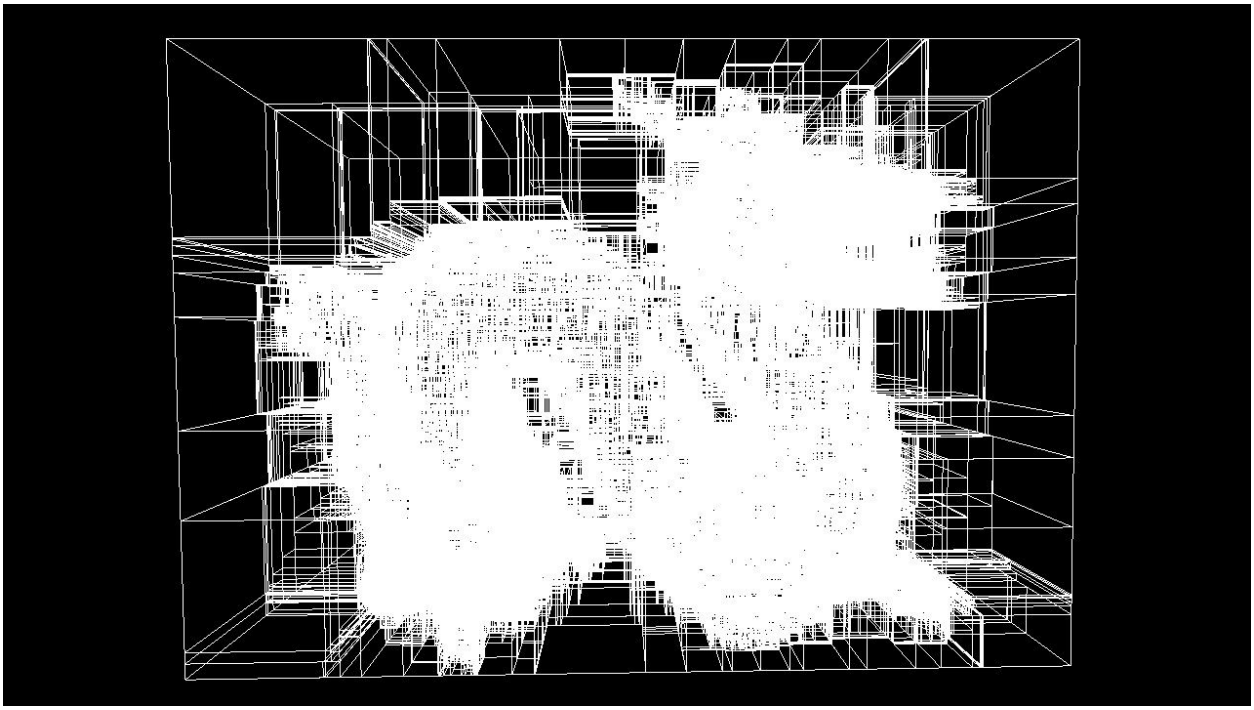
Federico Tedin - 53048
Javier Fraire - 53023

Mayo 2016 - C1

Decisiones de Diseño:

Estructuras de Aceleración:

Como estructura de aceleración se decidió utilizar *KD-Tree* y no un *Octree*. Esto se debe a que el *Octree* subdivide la escena en planos de igual tamaño (es decir, en cubos), por lo que es menos flexible. El *Octree* asume una subdivisión uniforme de los objetos, mientras que el *KD-Tree* permite subdividir en puntos arbitrarios, por lo que se adapta mejor a la geometría de los objetos. Como se puede observar en la siguiente imagen, si hubiésemos utilizado un *Octree*, el cuadrante superior izquierdo estaría innecesariamente dividido en muchos cubos más pequeños. Al utilizar un *KD-Tree* se logra una subdivisión más eficiente y granular. Además, el *KD-Tree* garantiza búsqueda $O(\log_m (n))$.

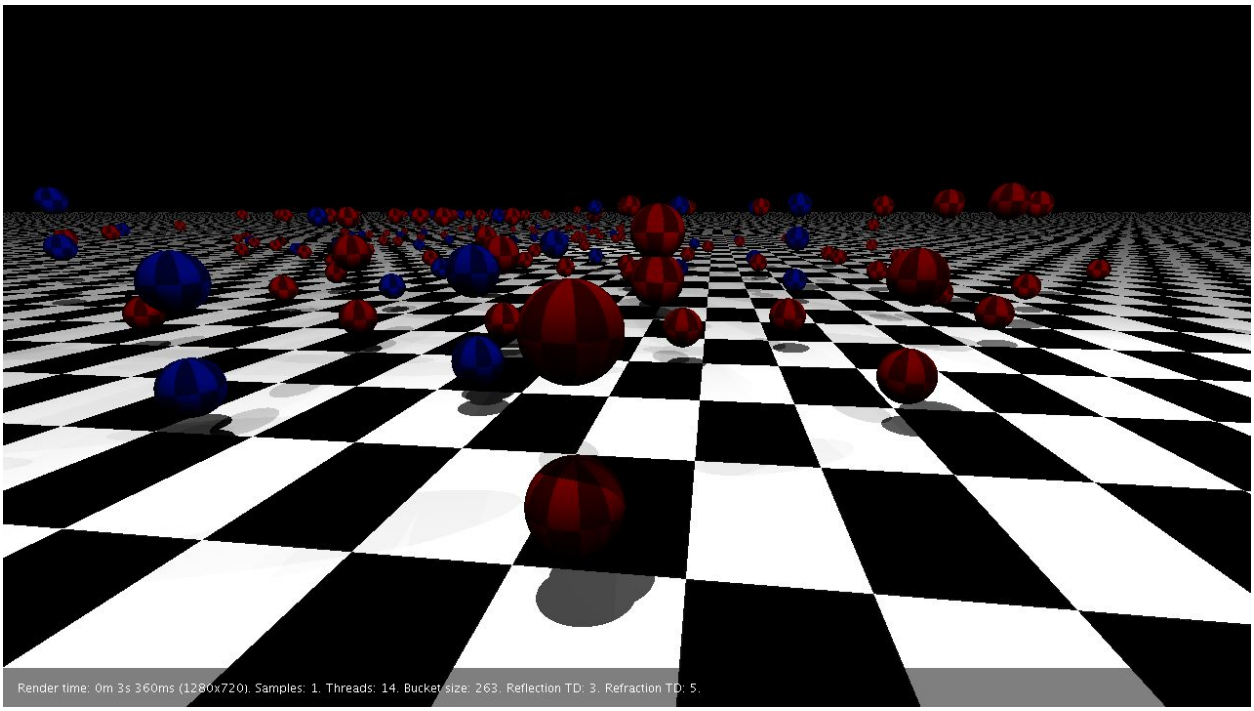


Para construir el árbol se decidió utilizar la mediana, ya que es simple de implementar, pero a su vez es efectivo. Si bien se sabe que existen mejores métodos, estos eran mucho más difíciles de implementar.

Dentro del *KD-Tree*, se realizaron pequeñas optimizaciones. Una de ellas fue calcular los *bounds* de la escena y luego, al tirar un rayo, verificar rápidamente si el rayo intersecta con ella. De esta forma, se evita recorrer el *KD-Tree* en casos en los que no era innecesario. Otra decisión que se tomó fue utilizar un stack en lugar de

hacer el proceso recursivo para que sea más performante. De esta manera, se evita el constante cambio de contexto, al saltar a un nuevo llamado de función. El uso de un objeto de stack también permite un menor uso de memoria comparado a la alternativa recursiva. A su vez, se decidió realizar dos implementaciones de *KD-Tree*, una para primitivas y otra para *meshes*. Esto se debe a que se necesitaba guardar distinta información: para primitivas se utiliza el bounding box y la lista de primitivas, y para *meshes* los índices de los triángulos y su centro. No se quiso realizar una abstracción intermedia ya habría traído un pérdida de performance.

En cuanto a la profundidad y la cantidad de elementos por hoja, se determinó para el *KD-Tree* de primitivas utilizar 3 primitivas por hoja y una profundidad máxima de 14. De esta forma, el árbol estaría completo con 2^{16} x 3 = 49,152 primitivas. El número de primitivas se determinó corriendo varias pruebas, entre ellas, se armó una escena con más de 200 esferas y se probaron diversos valores. Utilizando 3 primitivas por hoja, se obtenía un tiempo de menos de 3 segundos, en cambio en los demás casos este se encontraba por encima de los 3 segundos. El objetivo fue entonces encontrar un balance en el cual no hubieran demasiadas primitivas por hoja, pero a su vez que hubiese un número significante, ya que esto habría resultado en una búsqueda más larga. A continuación se presenta una de las escenas utilizadas para elegir la cantidad de primitivas por hoja:

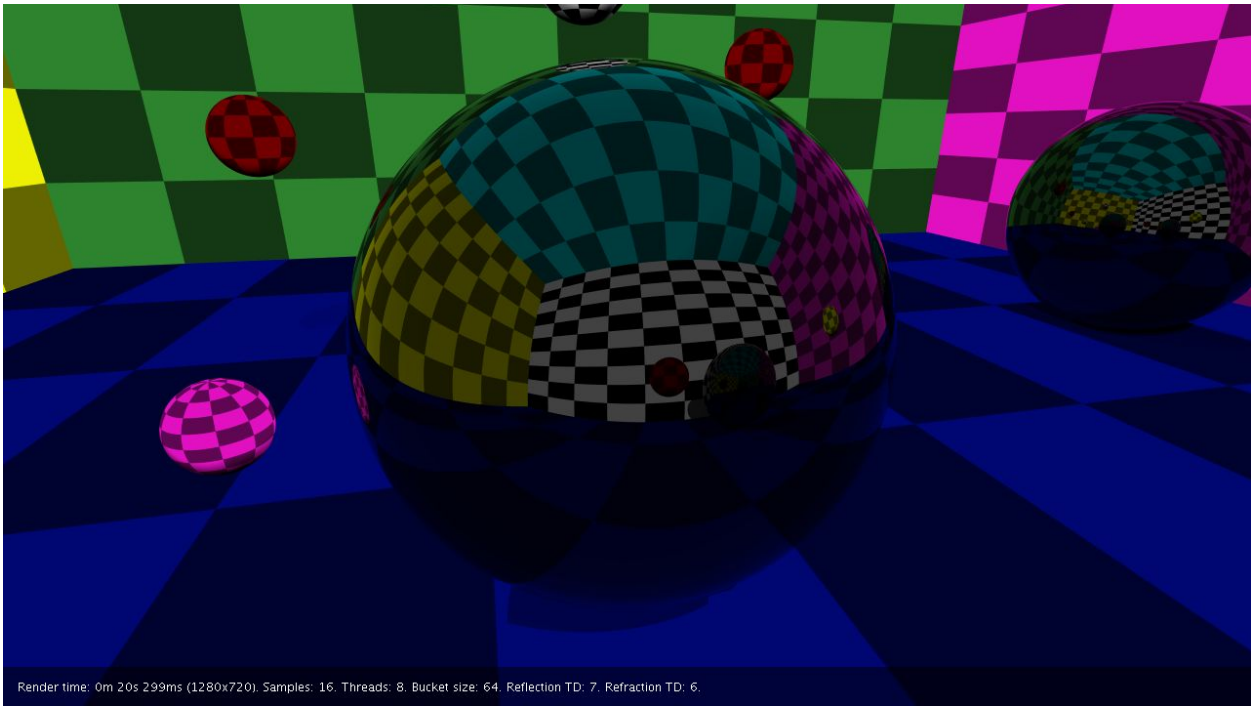


Spheres (1)

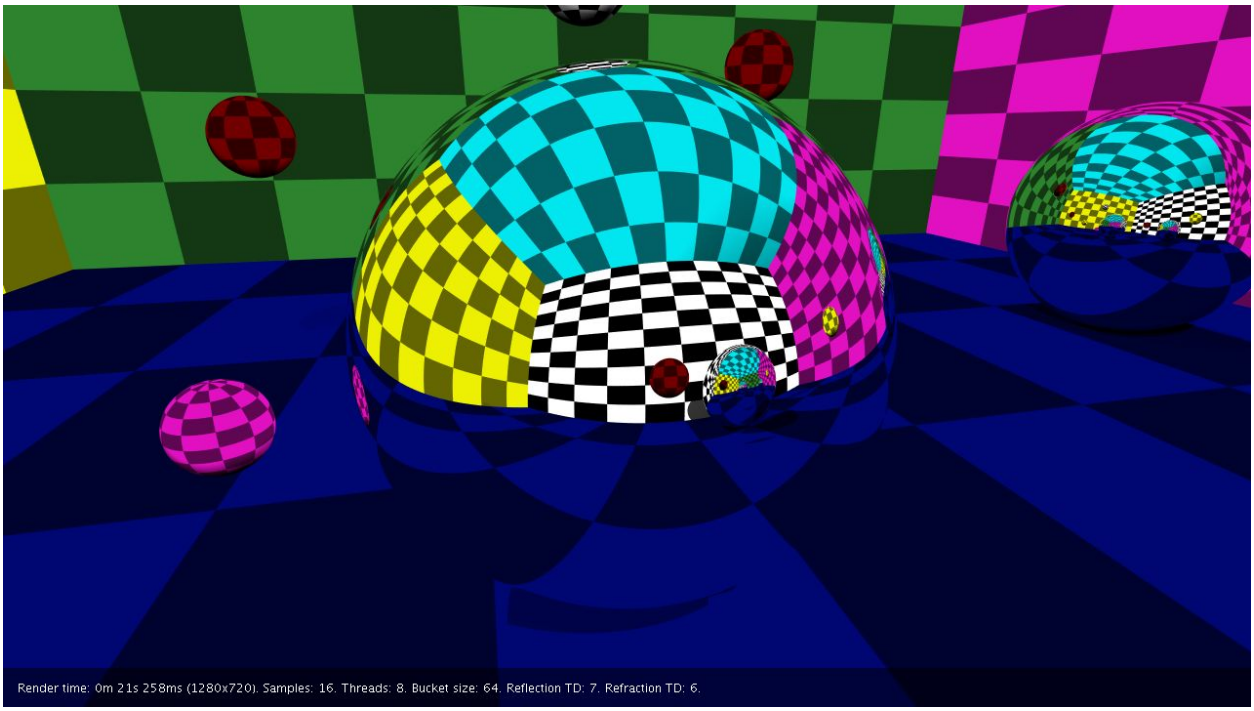
De forma análoga, para el *mesh KD-Tree* se determinó una máxima profundidad de 20 y una cantidad de triángulos por hoja de 25. En este caso, el número es mayor ya que una *mesh* tiene miles de triángulos y si se utilizara un número muy chico, la profundidad del árbol sería demasiado grande. En este caso, el árbol estaría completo con 2^{20} x 25 = 26,214,400 triángulos. En cuanto a las primitivas *unbounded*, como por ejemplo el plano infinito, se decidió no incorporarlas al *KD-Tree* ya que ocuparían un lugar en todas las hojas causando que el árbol sea más grande.

Materiales:

En el caso del material refractivo, se decidió utilizar la aproximación de Schlick ya que esta es un 30% más rápida que Fresnel, obteniendo de todas formas buenos resultados. Para el material reflectivo, se probó utilizar Fresnel (Schlick) con el índice de refracción del cromo para lograr que la reflexión sea más realista, pero en el resultado esta se veía muy oscura por lo que se optó por no utilizarlo. A continuación se presentan dos imágenes, la primera utilizando Fresnel, y la segunda sin Fresnel:



Fresnel Reflection (2)



Non Fresnel Reflection (3)

En cuanto a las sombras generadas por materiales refractivos, se decidió ignorar la presencia de sombras cuando se estaba calculando el color de un material refractivo. Así, se evitó que el color del material refractivo fuera de color negro.

Opcionales:

En cuanto a los requerimientos opcionales, se implementaron la luz *spotlight* y los canales texturizables. Se implementó la luz spotlight porque se cree que es un tipo de luz fundamental. En el caso de los canales texturizables, se decidió implementarlos para tener más flexibilidad a la hora de procesar e instanciar los materiales. Aplicar texturas a distintos canales puede tener resultados interesantes.

Para el procesamiento de canales de materiales, se optó por crear una clase *Channel* que modelara un canal generico. Un canal tiene dos componentes principales: la primera, que puede ser un color o un valor escalar, y la segunda, que es una textura opcional. La textura, a su vez, tiene también tiene propiedades de escala y *offset*. Luego, los materiales creados simplemente son conjuntos de canales: por ejemplo, el material de tipo Phong tiene un canal para la componente de color, otro para la componente especular, y otro para el exponente. De esta forma, es trivial

modificar qué canales son utilizados por cada material, asi también como crear materiales nuevos.

Optimizaciones:

Una de las optimizaciones implementadas fueron las *quick collisions*. Originalmente, al ocurrir una colisión contra una primitiva, se calculaba la normal, los uv, etc., cuando esto no era completamente necesario ya que era posible que la colisión no fuera la más cercana. Por lo tanto, para evitar cálculos innecesarios y para evitar almacenar tanta información, se decidió implementar una versión más simplificada de la colisión que solo cuente con el rayo, la primitiva y la distancia *t*. Luego, si efectivamente esa colisión es la más cercana, se calculan el resto de los parametros necesarios. En el caso de las *meshes*, se guardan las coordenadas baricéntricas y el índice del triángulo para evitar recalcular dichas coordenadas y para evitar buscar el triángulo nuevamente, a la hora de completar la colisión.

También se decidió implementar la librería matemática de vectores y matrices en lugar de utilizar alguna existente para poder aprender y entender las distintas técnicas que se deben utilizar para obtener un mejor rendimiento. La clase *Matrix4*, por ejemplo, cuenta con 16 variables en lugar de usar *arrays* ya que esto es más performante al tener una indirección de memoria menos cada vez que se accede a un valor. Los vectores fueron implementados de la misma manera. Además, las clases se implementaron de forma inmutable para evitar problemas de concurrencia.

Para el sampleo de puntos en el espacio bidimensional, se opto por utilizar la tecnica de Correlated Multi-Jittered Sampling, que se especifica en el paper de Pixar referenciado en la bibliografía. Por un lado, el método de Jittered Sampling permite tomar puntos pseudoaleatorios de una grilla de forma rapida, pero cuenta con el problema de clustering cuando se contempla sólo uno de los dos ejes. Para evitar este problema, se puede aplicar la tecnica de N-Rooks, obteniendo entonces Multi-Jittered Sampling. Finalmente, se puede agregar otra modificación para obtener Correlated Multi-Jittered Sampling, que reduce el aglomeramiento de las muestras generadas en el plano. La técnica mencionada también fue elegida por su facilidad de implementación.

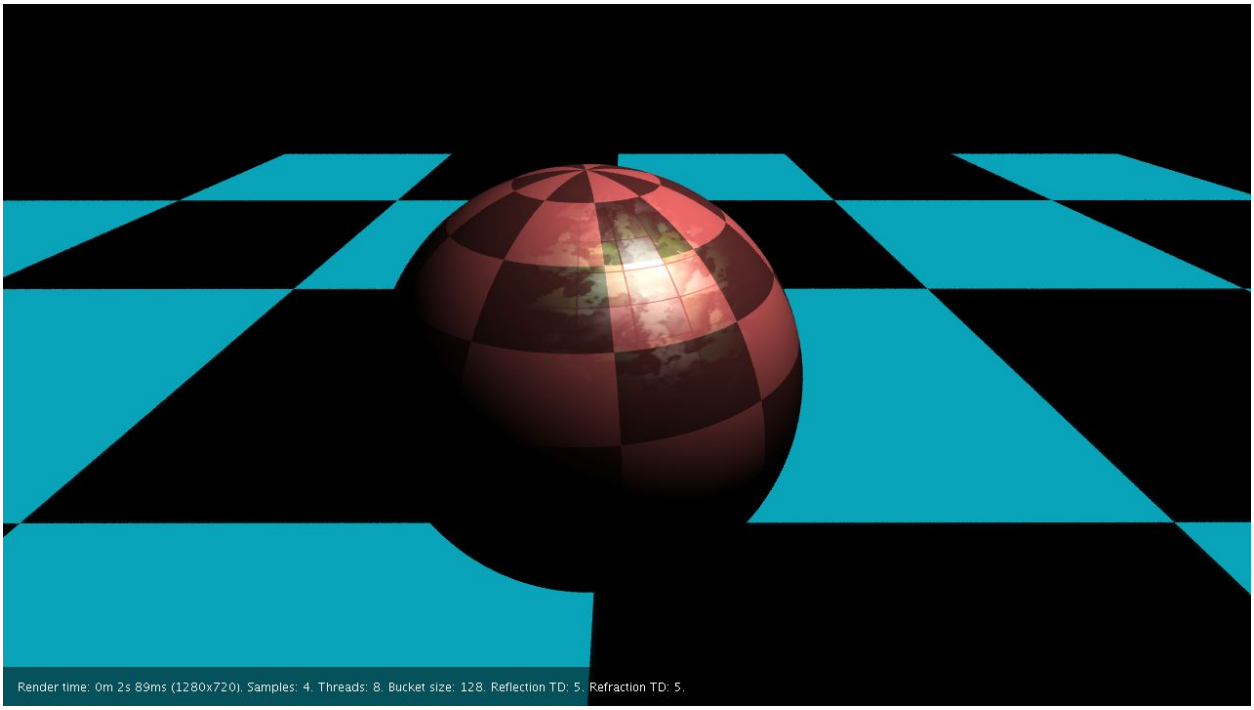
Para intentar reducir la cantidad de memoria reservada durante el proceso de renderizado, se decidió que *MultiJitteredSampler* mantenga un estado interno, junto a memoria pre-reservada. Así, se reutilizan los arreglos en vez de crear nuevos cada vez que se necesita generar nuevas muestras. Por el otro lado, esto trae problemas de concurrencia, por lo que a la hora de implementar *multithreading*, se crearon tantos s*amplers* como *threads,* para que cada thread utilice un *sampler* distinto.

Análogamente, se crearon tantos *arrays* de rayos como *samplers* y *threads* por el mismo motivo. Otro aspecto importante de *multithreading* es que se decidió recorrer los buckets de forma secuencial utilizando una cola ya que pre-asignar los threads a los buckets no era conveniente, debido a que se podían encontrar casos en los que hayan *threads* inactivos.

A la hora de implementar el *parser*, se optó por una aproximación tolerante a fallas. En caso de que un valor sea inválido, se utiliza un valor default. Por ejemplo, en el caso de que la textura sea inválida, se utiliza una textura de color rosa. En el caso de los archivos OBJ, en caso de que sean inválidos, se saltean todos los objetos que hacen referencia a ese *mesh*. Es decir, dependiendo del caso o se asigna un valor default o se descarta el objeto.

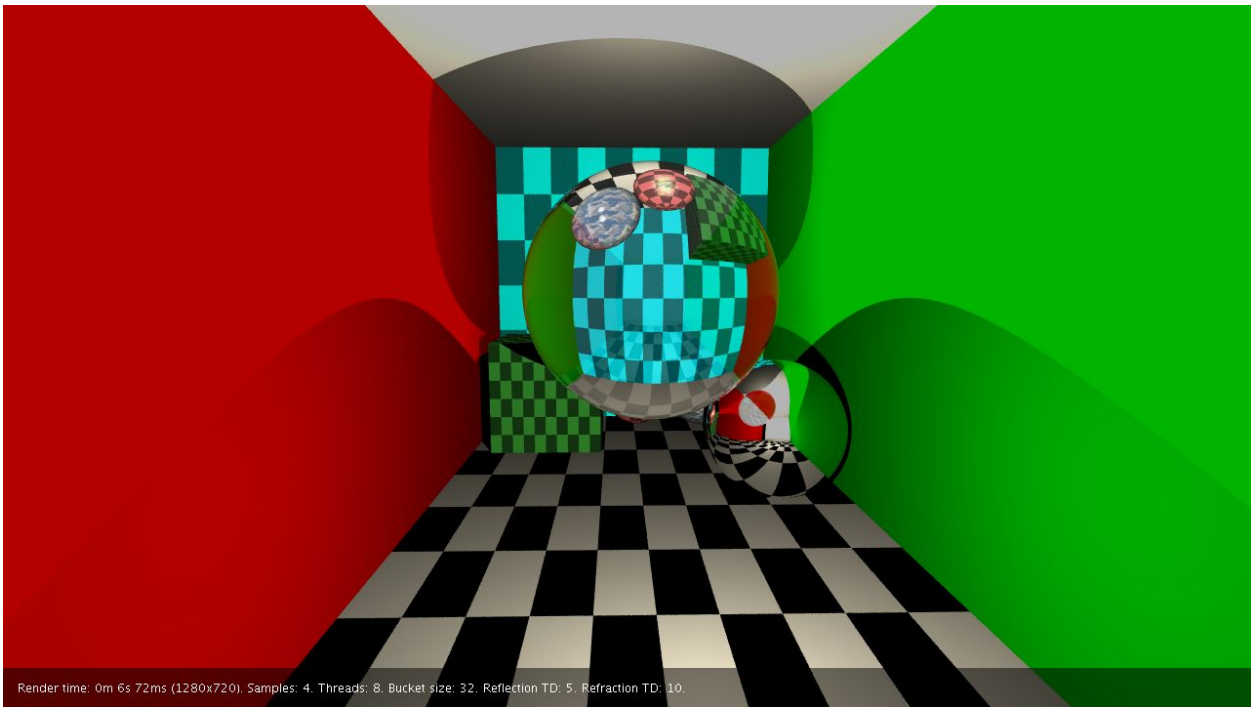
Imágenes de Ejemplo:

La siguiente imagen muestra una esfera con un material *phong* en el cual todos los canales tienen una textura. Se puede observar un *checker* en la componente difusa, un *grid* en la componente specular y finalmente nubes en el exponente.



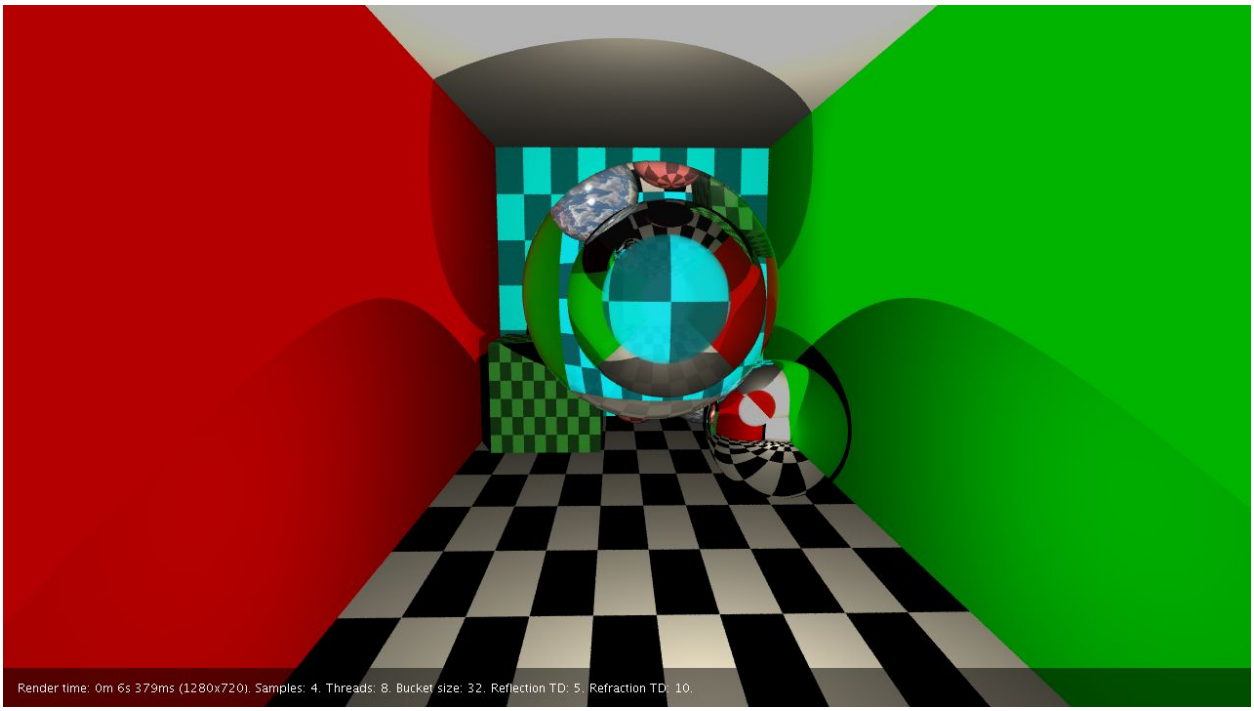
Basic Phong (4).

La siguiente imagen muestra una refracción de una esfera con un índice de 2.42. La imagen también cuenta con una esfera reflectiva en la que se puede observar el reflejo de la esfera refractiva a través de la cual se puede ver la escena.



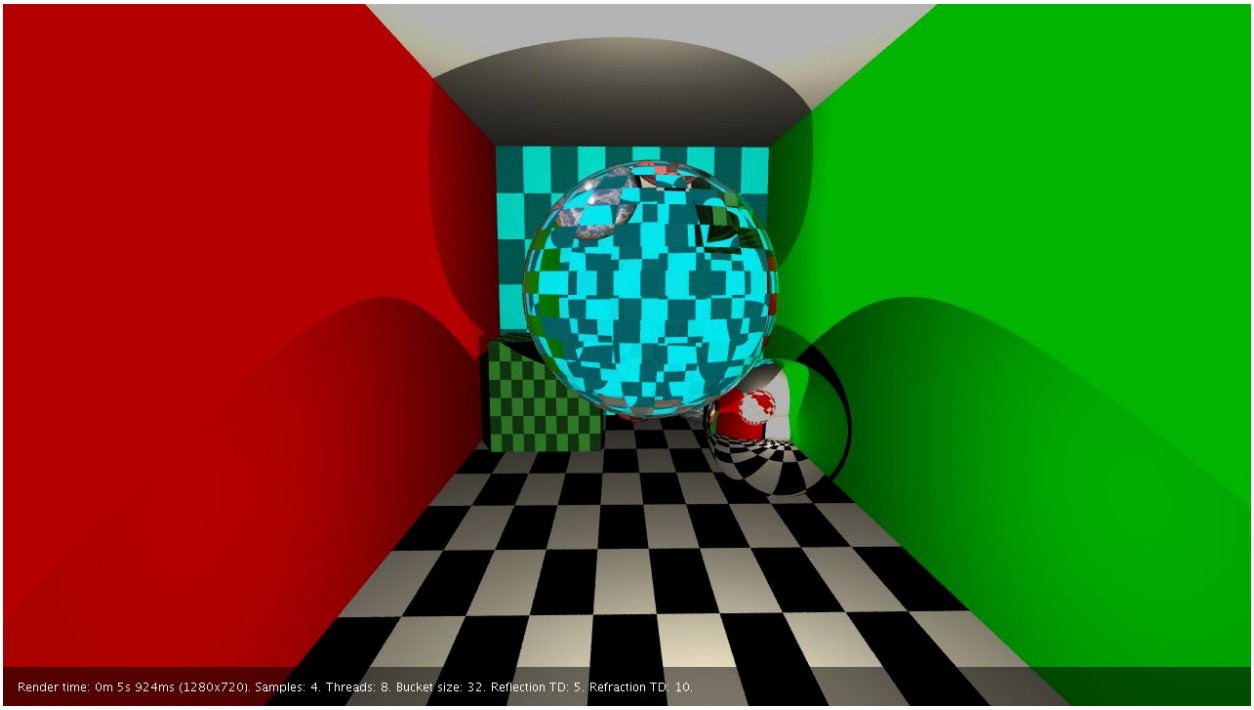
Simple Refraction (5)

La siguiente imagen es igual a la escena anterior, excepto por el detalle de que cuenta con otra esfera refrativa detrás. De esta forma, se puede observar un aumento mucho mayor en el centro. En este caso ambos esferas cuentan con un índice de refracción igual a 1.52.



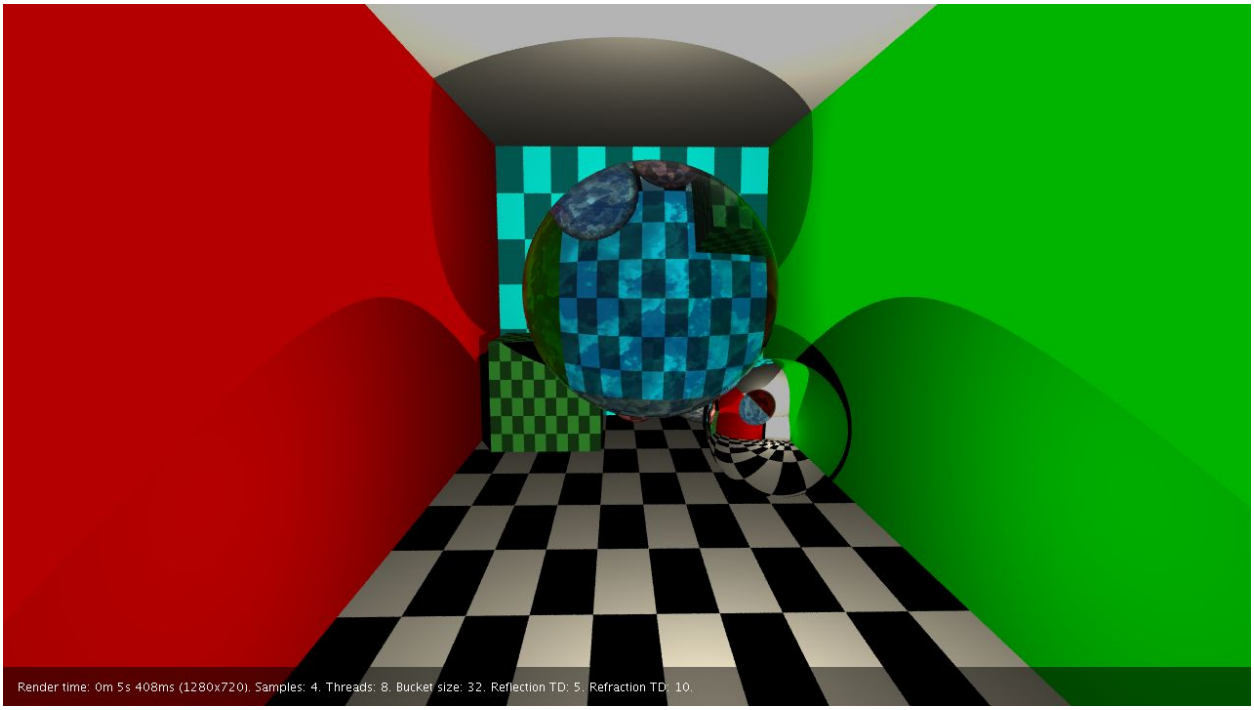
Double Refraction (6)

A continuación se muestra una imagen en la que se utilizó un *checker* como textura para el índice de refracción. Se observa un efecto extraño en las refracciones, tanto en la esfera refractiva como en los reflejos de la reflectiva.



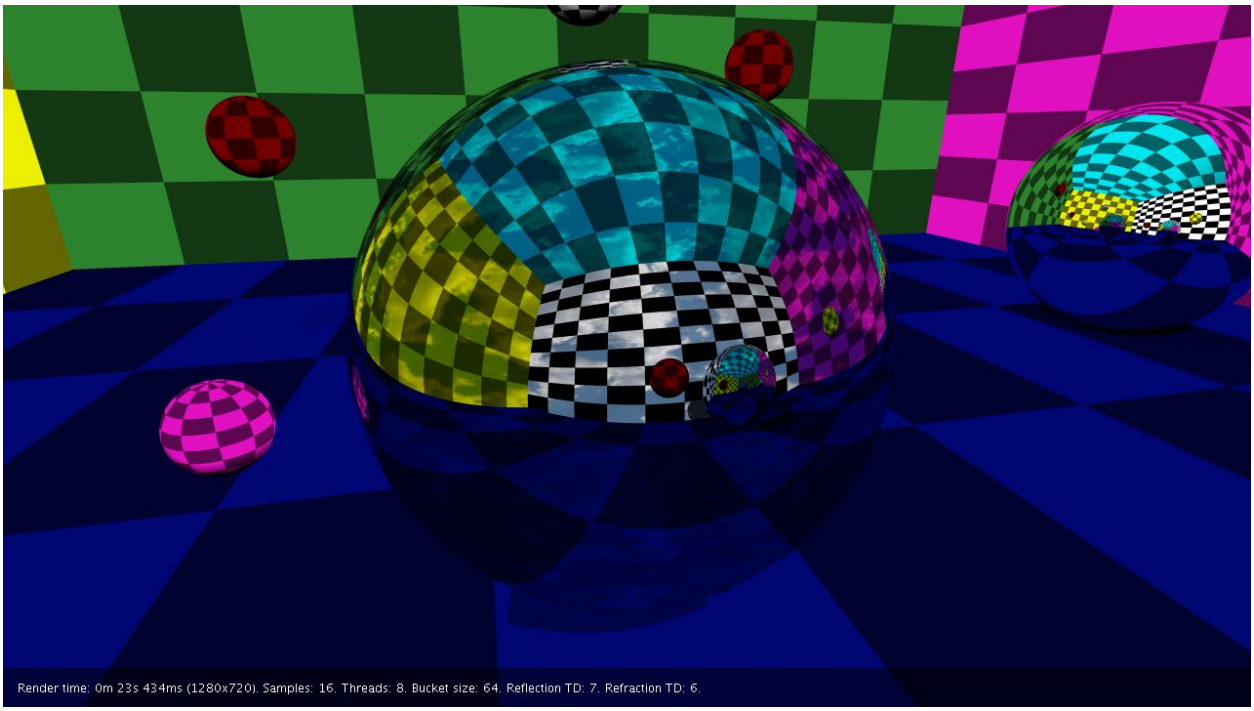
IOR Texture Refraction (7)

“Sky refraction”, contiene una esfera refractiva con una textura del cielo en su canal refractivo, y a su vez, el color del mismo es celeste. De esta manera, se logra un efecto cielo sobre la refracción.



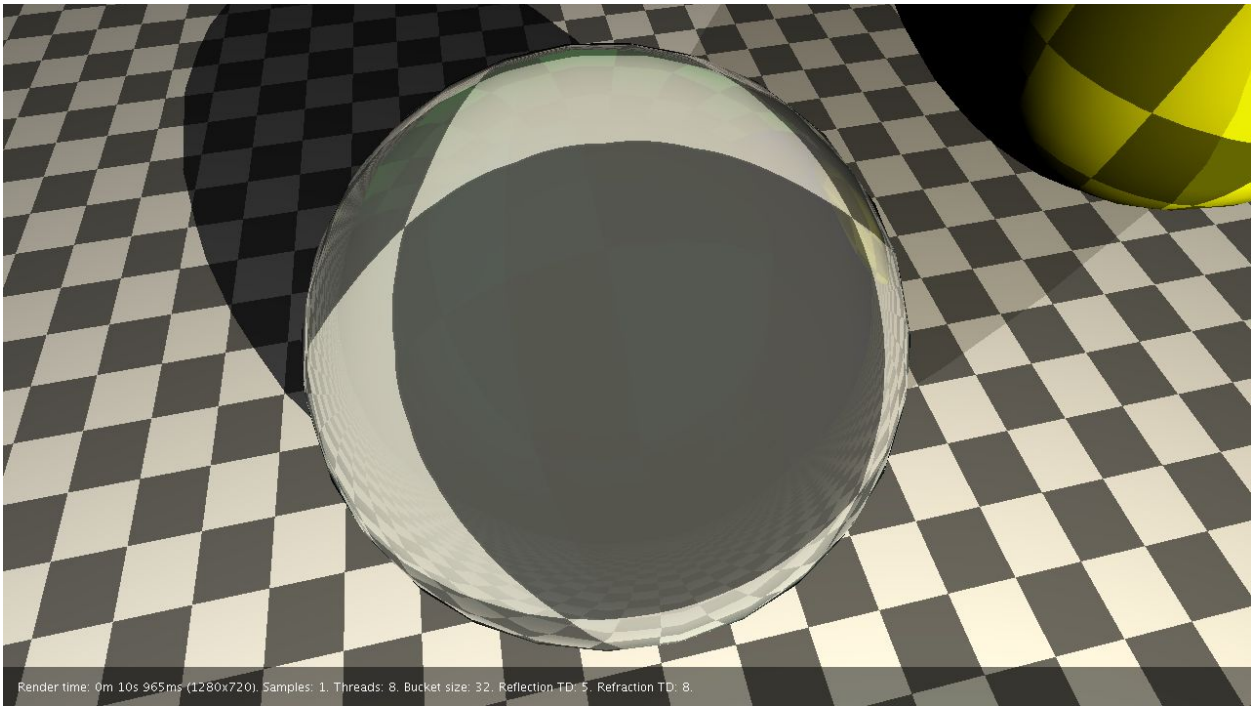
Sky Refraction (8)

Se utilizó la misma técnica en una esfera reflectiva en la siguiente imagen:



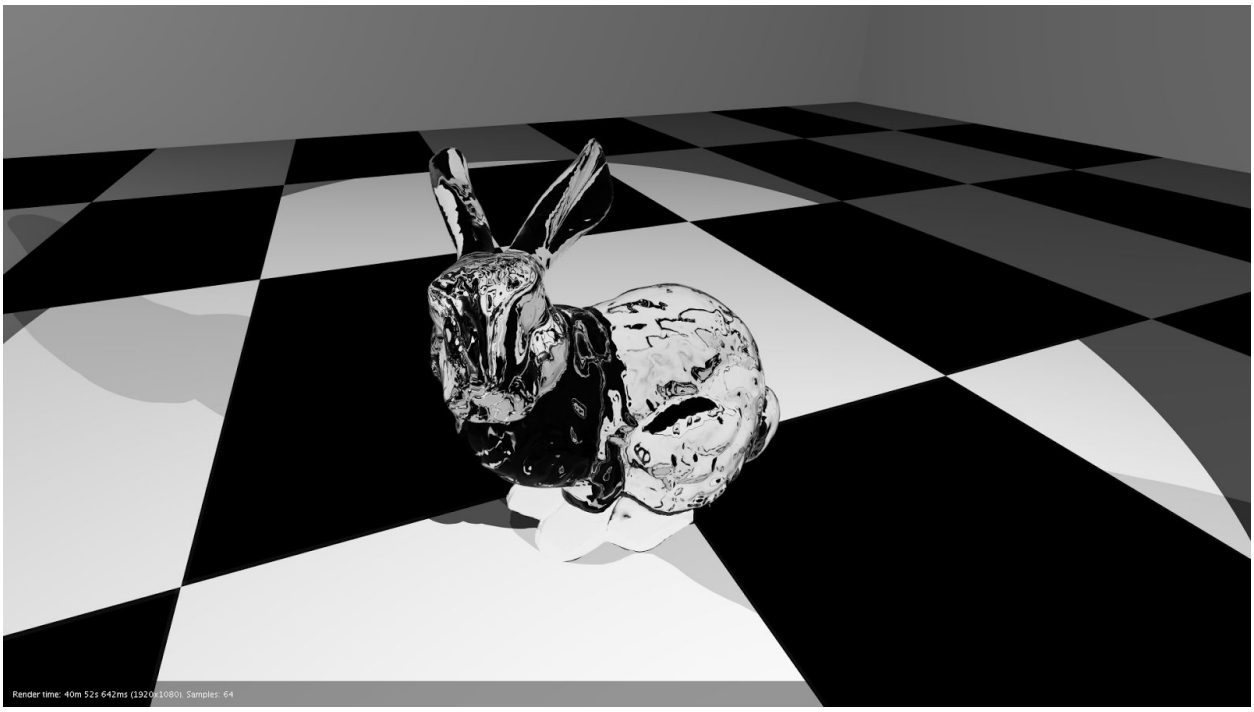
Sky Reflection (9)

La siguiente imagen muestra una refracción desde una distancia más corta y con otro ángulo. El índice de refracción utilizado fue 1.33. Se puede observar el efecto *zoom* que se genera y a su vez se pueden observar los bordes reflectivos que reflejan objetos como el plano, una pared verde, la esfera amarilla, etc.

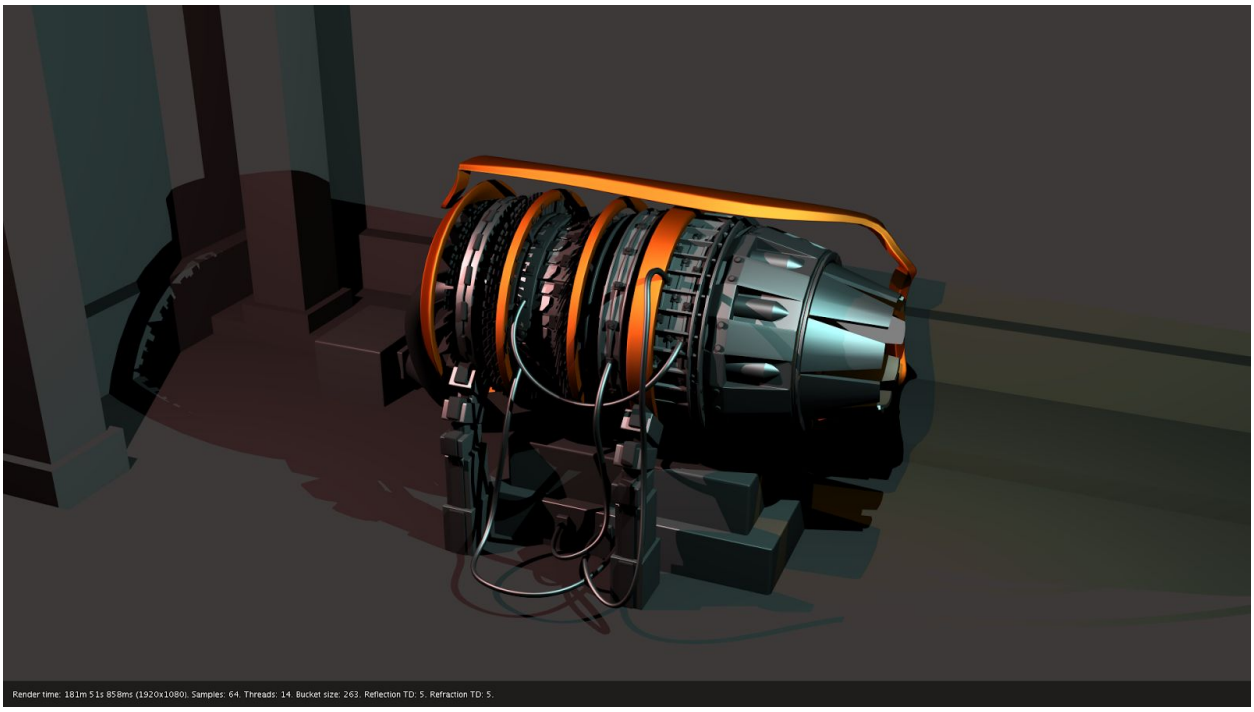


Refraction Zoom (10)

En la siguiente imagen se observa un material refractivo con un índice de refracción de 1.33 iluminado por dos *point lights* y una *spot* light. Se utilizaron 64 muestras para *antialiasing* para lograr un efecto más suave.



Refractive bunny (11)

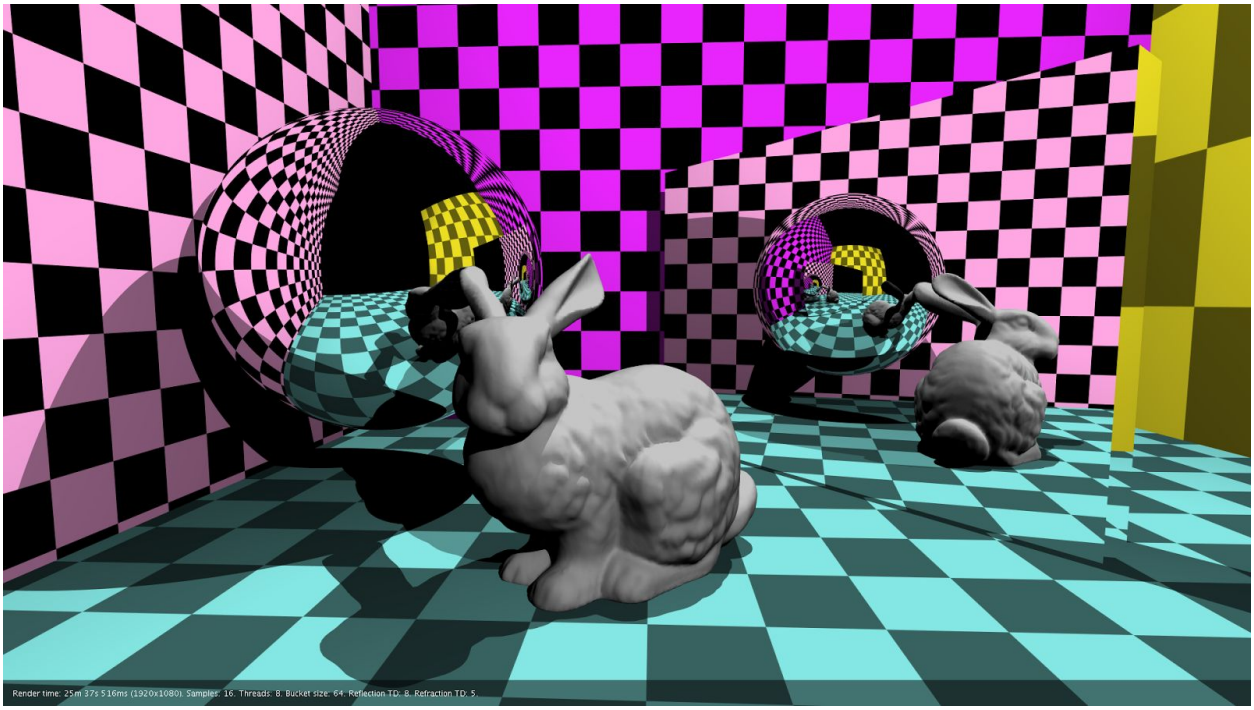


Bonus Scene 1 (12)

Bonus Scene 2 (13)

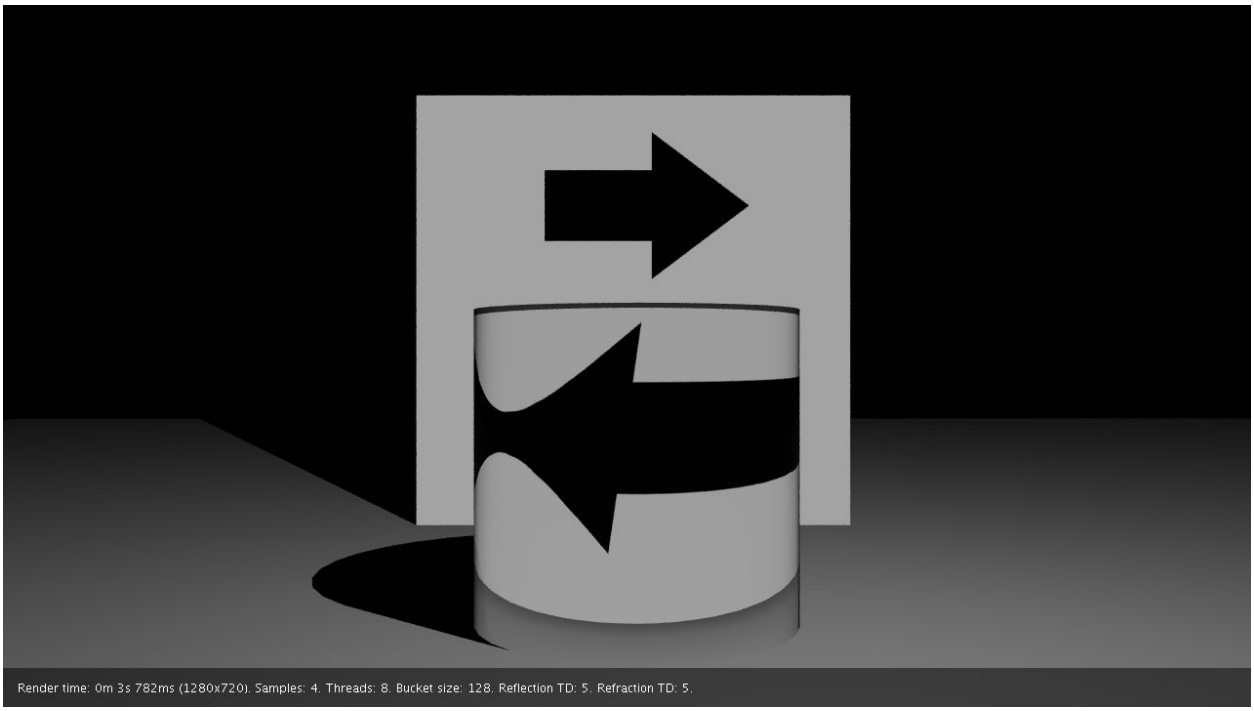
Thor (14)

La siguiente imagen muestra los efectos de la reflexión. En el espejo se puede observar el reflejo de la esfera y a su vez en la esfera reflejada en el espejo, se puede observar el espejo.



Bunny Mirrors (15)

Statue (16)



Refraction (17)

Benchmarks:

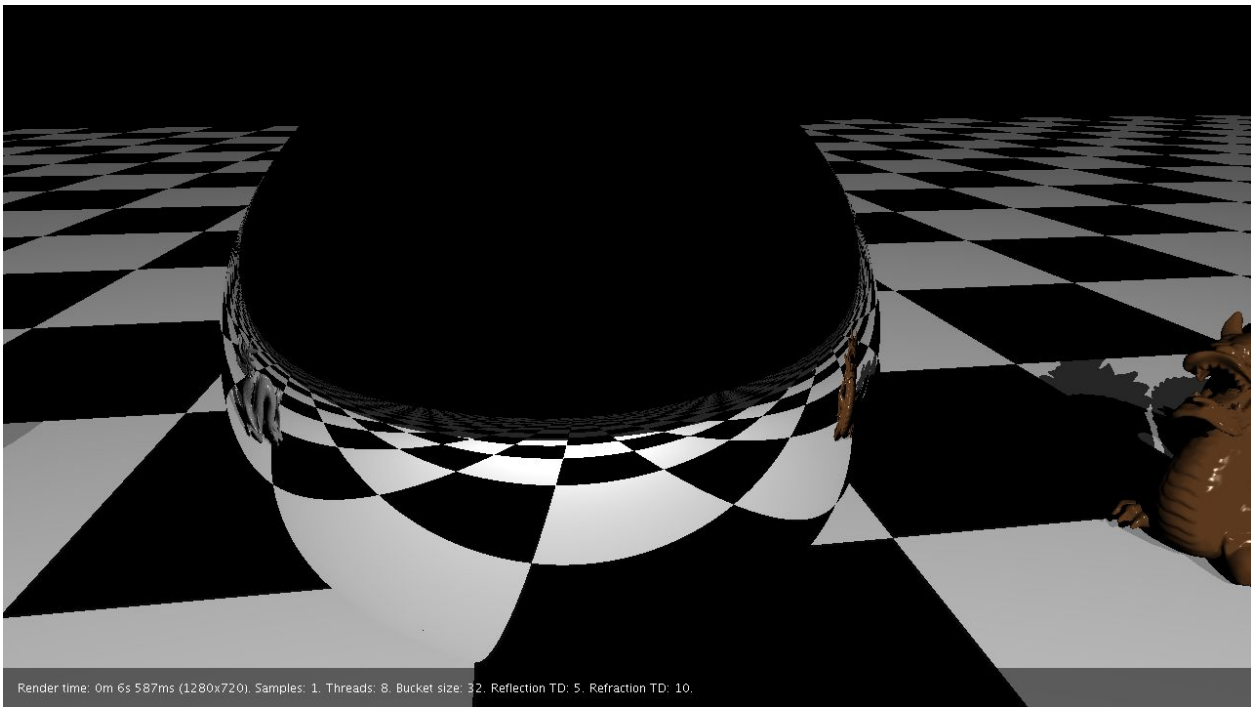
Cabe destacar que accidentalmente la construcción del *KD-Tree* de primitivas se realiza al comenzar el *render* por lo que esto afecta levemente los benchmarks de tiempo. Por lo que el tiempo de construcción del *KD-Tree* de primitivas se encuentra dentro del *render time*. Todos los *benchmarks* fueron realizados utilizando un procesador Intel Core i7 2.6ghz (3720QM) de 4 núcleos y con 4 núcelos de *hyper-threading* (virtuales).

Nombre	**Size**	**Samples**	**Threads**	**Bucket Size**	**RLTD**	**RRTD**	**Tiempo**
Spheres (1)	1280 x720	1	14	263	3	5	2s 861ms
Fresnel Reflection (2)	1280 x720	16	8	64	7	6	20s 299ms
Non Fresnel Reflection (3)	1280 x720	16	8	64	7	6	21s 258ms
Basic Phong (4)	1280 x720	4	8	128	5	5	2s 89ms
Simple Refraction (5)	1280 x720	4	8	32	5	10	6s 72ms
Double Refraction (6)	1280 x720	4	8	32	5	10	6s 379ms
IOR Texture Refraction	1280 x720	4	8	32	5	10	5s 924ms

(7)							
Sky Refraction (8)	1280 x720	4	8	32	5	10	5s 408ms
Sky Reflection (9)	1280 x720	16	8	64	7	6	23s 434ms
Refraction Zoom (10)	1280 x720	1	8	32	5	8	10s 965ms
Refractive Bunny (11)	1920 x1080	64	8	64	6	4	40m 52s
Bonus Scene 1(12)	1920 x1080	64	14	263	5	5	181m 51s
Bonus Scene 2 (13)	1920 x1080	4	32	64	5	5	2m 24s 832 ms
Thor (14)	1920 x1080	16	12	32	6	4	36m 21s
Bunny Mirrors (15)	1920 x1080	16	8	64	8	5	25m 37s
Statue (16)	1920 x1080	16	8	64	8	7	6m 26s 840ms
Refraction (17)	1280 x720	4	8	128	5	5	3s 782ms

Tabla 1: Benchmarks. RLTD = Reflection Trace Depth, RRTD = Refraction Trace Depth

Una de las imágenes que se utilizó para probar distintas alternativas de multithreading fue la siguiente:



Dragons (18)

Se utilizó dicha imagen ya que cuenta con reflexiones y a su vez con *meshes* con una cantidad de polígonos media. La siguiente tabla muestra los resultados de las distintas configuraciones:

Threads	**Bucket Size**	**Tiempo**
16	128	23s 823ms
8	128	6s 486ms
4	128	7s 318ms
8	64	5s 788
8	256	32s 651ms
4	180	11s 894ms
8	32	6s 587ms

16	32	6s 358ms
6	32	7s 667ms

Tabla 2: Multithreading benchmarks dragons

Otra escena que se utilizó para probar distintas configuraciones de *mulithreading* es *spheres.* Ya que esta cuenta con muchas primitivas*.* A continuación se presenta una tabla con resultados:

Threads	**Bucket Size**	**Tiempo**
4	64	3s 148ms
8	64	2s 866ms
16	64	3s 65ms
10	64	2s 985
8	32	2s 600ms
8	256	3s 219ms
8	10	2s 934ms
16	16	3s 313ms

Tabla 3: Multithreading benchmarks spheres

La tercera escena que se utilizó para probar distintas combinaciones de *multithreading* fue *Bonus Scene 1* ya que esta era compleja y pesada. A continuación se listan los resultados:

Threads	**Bucket Size**	**Tiempo**
14	263	1m 45s 815ms
14	32	1m 23s 271ms
14	64	1m 30s 29ms
8	32	1m 22s 695ms

4	32	1m 38s 583ms
32	32	1m 36s 645ms
8	48	1m 38s 58ms
2	32	2m 25s 221ms
8	16	1m 17s 171ms
64	32	1m 36s 64ms
6	32	1m 30s 297ms
1	1280x720	5m 16s 283ms

Tabla 4: Multithreading benchmarks Bonus Scene 1

Es fácil observar que si se tienen menos threads que la cantidad de núcleos del procesador, el tiempo de *rendering* va a ser más alto. Esto se debe a que no se aprovecha todo el poder de procesamiento del procesador, que cuenta con varios núcleos. Particularmente en el caso de 1 *thread* en *Bonus Scene 1*, este tarda más de cuatro veces más que utilizando 8 *threads*. También, se puede observar que utilizando el doble de threads que la cantidad de núcleos del procesador, se obtienen buenos resultados. Sin embargo, el tamaño del *bucket* también tiene un gran impacto en el tiempo de renderizado. Se puede notar que aumentando la cantidad de threads (hasta cierto punto) no tiene un impacto tan significativo como variar el tamaño del *bucket*. Los buckets de gran tamaño enlentecen el proceso. Al utilizar un mínimo de 8 *threads*, se están aprovechando los 4 núcleos físicos y los 4 núcleos virtuales del procesador, lo que permite obtener una mejor performance. Por este motivo utilizar 6 *threads* resultó más lento que utilizar 8. Es importante destacar que el tamaño del *bucket* y la cantidad de *threads* son factores que tienen un impacto distinto según el tipo de escena que se este renderizando.

Problemas Encontrados:

Uno de los problemas que se encontró fue que las esferas se veían negras. Al no encontrar el problema empezamos a utilizar distintas técnicas de *debugging* (como por ejemplo utilizar como color la normal). Se continuó buscando el problema pero no se encontraba. Luego al revisar la implementación de la esfera, se observó que faltaban analizar algunas condiciones que podrían significar que haya o no una colisión. La condición en cuestión era que t_0 no debe ser mayor que la distancia máxima del rayo y t_1 no debe ser menor que 0. Por este motivo, se obtenían colisiones con la esfera en su interior en lugar del borde, por lo que al trazar un rayo hacia la luz, la esfera chocaba contra sí misma y la función de visibilidad retornaba siempre falso.

Otro gran problema se encontró al implementar el material refractivo. Se observaba que el material funcionaba en esferas utilizando un epsilon muy grande para trazar el rayo hacia adentro de la primitiva. Además su funcionamiento no era el mismo que con una esfera de tipo *mesh.* Investigando, se aprendió que la resolución de las ecuaciones cuadráticas no es del todo precisa. Por lo tanto se cambió el método de calcular la colisión de la esfera por uno más performante y más preciso. De esta manera se logró el correcto funcionamiento. Además se encontraron varios problemas de signos. En el cubo se encontró un problema similar al de la esfera pero no se pudo resolver y este no funciona adecuadamente con el material refractivo. Se cree que tiene que ver con la precisión del cálculo de la colisión con las cajas.

El *KD-Tree* también fue otro lugar de muchos problemas. Este se re implementó múltiples veces hasta lograr su correcto funcionamiento. Uno de los problemas fue, por ejemplo, no añadir la primitiva a ambos hijos cuando el plano la cortaba transversalmente. Otro fue descartar incorrectamente una rama del árbol cuando todavía podía ser factible que esta cuente con primitivas más cercanas. Estos problemas se solucionaron implementando el *KD-Tree* de forma iterativa utilizando un *stack* utilizando como referencia el libro *Physically-based rendering.* Un problema que no se pudo solucionar se presenta en la escena *reflection.ssd* provisto por la cátedra. Esta cuenta con una semiesfera. Nuestra implementación cuenta con un problema que la mitad de la media esfera no se renderiza adecuadamente.

Otro problema encontrado fue la aparición de píxeles negros en algunas partes de la escena. Esto se solucionó al convertir todo de *float* a *double* para tener menos errores de precisión.

Bibliografía:

- Pixar Technical Memo 13-01 - Correlated Multi-Jittered Sampling (5 de Marzo, 2013)
  - http://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf

- Reflections and Refractions in Ray Raytracing - (13 de Noviembre, 2006)
  - https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf

- Physically Based Rendering: From Theory to Implementation - Second edition - Greg Humphreys y Matt Pharr (Julio 2010)