Computación Gráfica

Informe de Trabajo Práctico Final

Extensiones

Federico Tedin - 53048
Javier Fraire - 53023

Julio 2016 - C1

Cambios realizados

Se realizaron los siguientes cambios al proyecto:

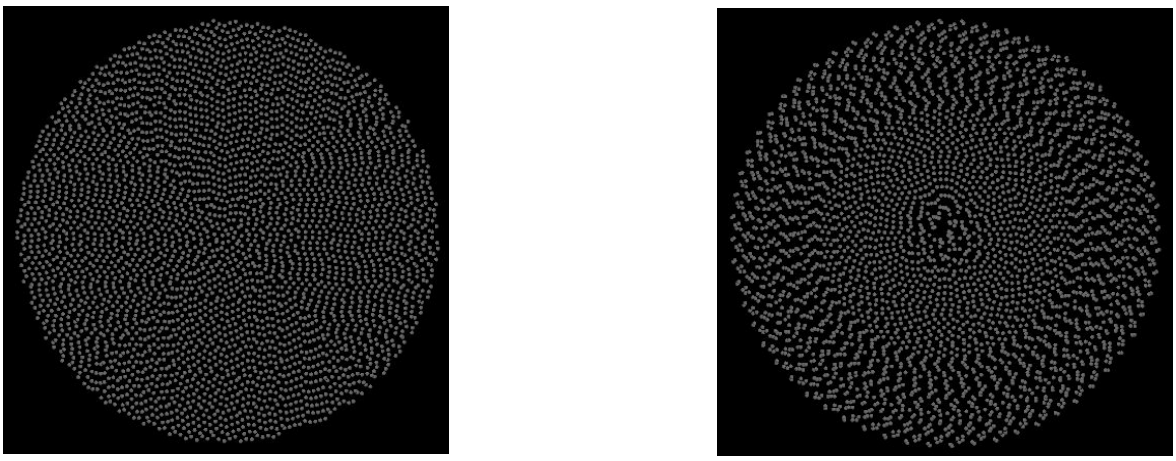
- Se reparó la selección de luces en modo *path tracing.* Debido a un error de *casting*, siempre se elegía la primera luz de la lista de luces en la escena. Para obtener el índice de la luz, se utilizaba *Math.random()* y luego se multiplicaba por la cantidad de luces. El problema era que se realizaba el *cast* a entero sobre *Math.random()* y no sobre el total, por lo que siempre daba 0. Luego de ser reparado, se elige ahora una luz al azar al momento de calcular la iluminación directa.
- Se implementó *Depth of Field* con *Bokeh.*
- Se implementó *Normal Mapping.*
- Se implementaron las cámaras cilíndrica, esférica y *fisheye.*
- Se implementaron texturas procedurales para madera, mármol, *Voronoi* y nubes.

Features implementados

Depth of Field

Una de las funcionalidades implementadas fue *Depth of Field*. Se decidió implementar esta funcionalidad ya que tiene muchas implementaciones y es muy utilizada en el mundo de la fotografía. Para implementarlo se utilizó el método del libro "Physically Based Rendering: From Theory to Implementation". Se utilizan dos parámetros: la apertura y la distancia focal. La apertura se utiliza para determinar el tamaño del disco (apertura de la lente) cuando se obtienen muestras y la distancia focal para establecer la distancia al plano de foco. El algoritmo es muy sencillo: se establece como origen del rayo la muestra obtenida del disco, y luego se utiliza la distancia focal para determinar el punto de foco y orientar la dirección del rayo hacia ella. Para ello, simplemente se divide la distancia focal por la dirección **z** del rayo hacia el pixel, ya que es el plano que nos importa es el **z**. De esta forma, se logra un foco sobre el plano focal y el resto de la imagen se desenfoca.

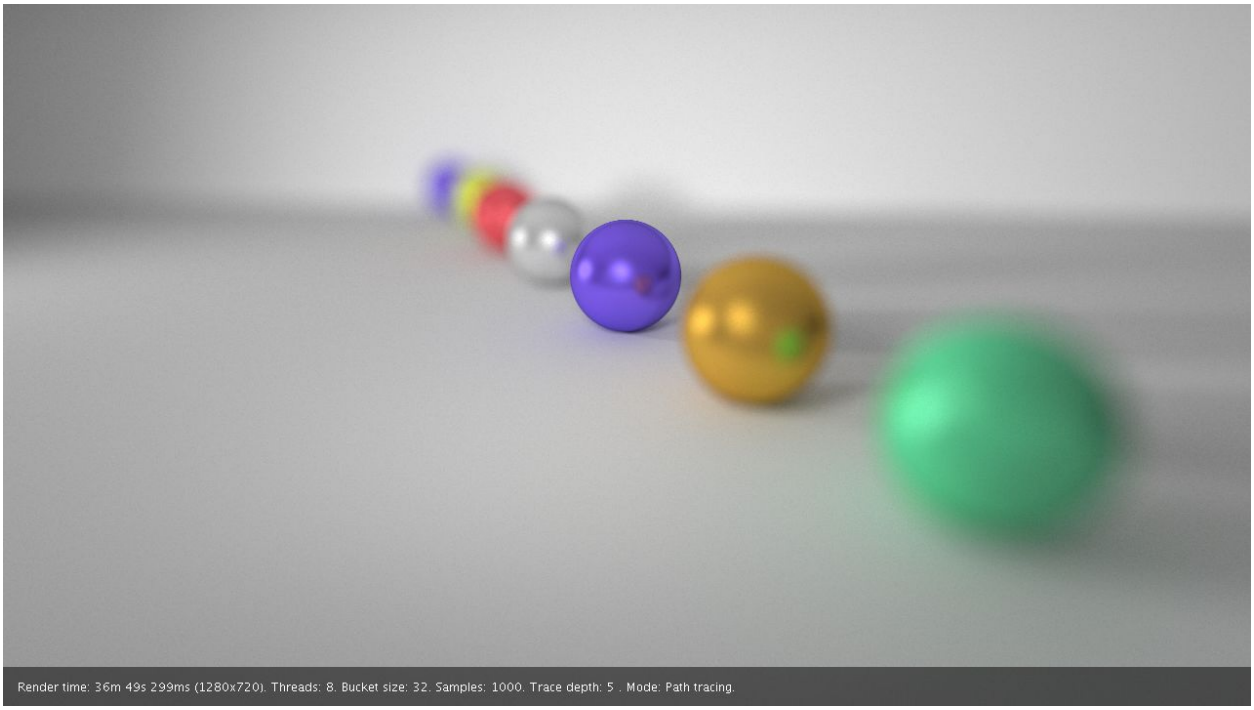
En cuanto al muestreo, se decidió utilizar *concentric mapping* ya que este cuenta con una distribución más uniforme. Se descartó *rejection sampling* debido a que si bien es simple, es costoso descartar muestras. A continuación se muestra una comparación de *concentric mapping* contra realizar un *mapping* polar:



Izquierda: Concentric Mapping, Derecha: Mapping polar

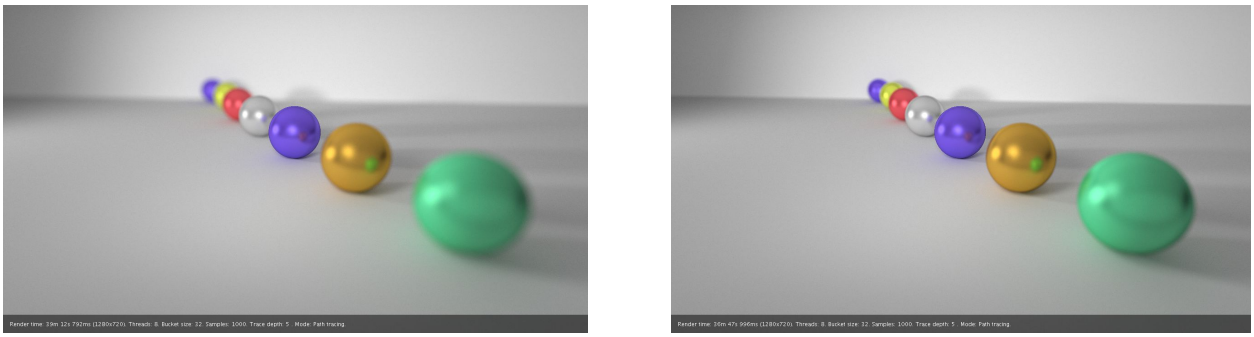
Como se puede observar, *concentric mapping* tiene una distribución más uniforme y el mapeo por coordenadas polares se encuentra *undersampled* en algunas secciones. Se utilizó la implementación de *concentric mapping* de "Ray Tracing from the Ground Up".

En cuanto al formato, se creó un nuevo tipo cámara *ThinLens* y se le agregaron los parámetros *aperture* y *distanceFocus* (ambos medidos en metros). Se decidió crear un nuevo tipo de cámara porque si bien *Pinhole* es un caso particular del *ThinLens* en el que el radio es infinitamente chico, sería muy poco performante generar samples para un radio tan chico cuando no es necesario. Además, la muestra se utiliza para cambiar el destino del rayo en *Pinhole* y en cambio en *ThinLens* para cambiar el origen, lo que rompería la dinámica del código. A continuación se muestra una imagen en la que se aplicó *depth of field* con una apertura de 400 mm y una distancia focal de 8 metros:



Depth of Field (1)

A modo comparativo, se presenta la misma escena con aperturas de 200 mm y 100 mm:



Izquierda: Depth of Field (2) con apertura 200 mm. Derecha: Depth of Field (3) con apertura 100mm.

En las imágenes se puede observar que cuanto mayor es la apertura, mayor es el desenfoque de los elementos que no se encuentran en el plano de foco. Esto se debe a que al tener un lente más amplio, el origen de los rayos varía más. Cuanto mayor es la apertura, menor tiene que ser la distancia de los objetos al plano de foco para que estos se visualicen borrosos.

A continuación se presenta otra imagen con *depth of field* aplicado. En ella, se colocó una gran cantidad de *area lights* fuera de foco. Se observan círculos en lugar de rectángulos debido a que utiliza una apertura de 300 mm, una distancia focal de 10.65 metros y las luces se encuentran a una a más de 12 metros de la cámara.

Stormtrooper (4)

Además, se decidió implementar bokeh ya que se logran efectos interesantes. Para ello, se especifica el id de la textura como “bokehTextureId” en la cámara. Dicha

textura debe ser en blanco y negro y debe ser binaria, ya que se considera al color blanco como parte del lente y al negro como fuera del lente. En cuanto a la implementación, se genera un *cache* de samples de gran tamaño con muestras que se encuentran en la zona blanca de la textura (dentro del lente). Luego, al momento de lanzar los rayos, se prueba utilizar la muestra recibida (obtenida de los *caches*) pero si dicha muestra no se encuentra dentro del lente, se elige al azar una muestra del *cache* de muestras que si se encuentran dentro del lente. De esta forma, se genera una mejor aleatoriedad a la hora de obtener muestras. A continuación se muestra una escena que cuentamente únicamente con *area lights* de distintos tonos de rojo y distintas intensidades. La apertura es de 500 mm y la distancia focal de 8 metros. Se utilizó una imagen de un corazón como lente. Se puede observar que las luces más cercanas no tienen una forma tan marcada debido a que se encuentran más cerca al plano de foco.

Hearts (5)

La siguiente imagen es la misma que *stormtrooper* pero con el detalle que utiliza bokeh en forma de estrella en lugar de un simple disco. De esta forma, se generan estrellas en las fuentes de luz.

Stormtrooper Bokeh (6)

Normal Maps

Otra funcionalidad implementada fue *normal maps*. Se decidió implementar dicha funcionalidad ya que permite tener modelos con mayor precisión sin la necesidad de contar con tantos vértices, lo que disminuye los tiempos de *rendering*.

En el caso de los planos, como estos se encuentran orientados hacia el eje **y**, se utiliza el eje **x** como tangente y luego se calcula la bitangente haciendo el producto vectorial. De forma similar, en el caso del cubo, se retorna el eje **z** si la cara donde se detectó la colisión está alineada con el **x**, el **x** si está alineada con el **y** y el **y** si está alineada con el eje **z**. Todos con sus respectivos signos.

Para la esfera, primero se trabajó en 2 dimensiones utilizando **x** y **z**, es decir, se ignoró el eje **y**. La tangente y bitangente se calcularon de la siguiente forma:

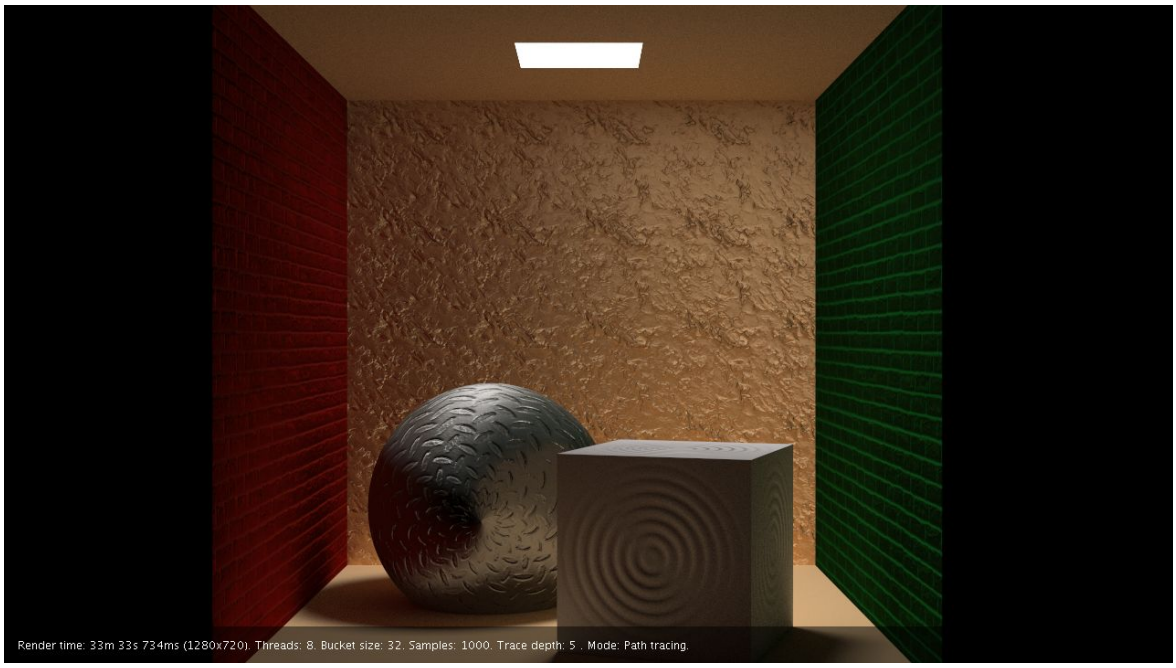
$$N_2 = (N_x,\ 0,\ N_z)$$
$$V = (-N_z,\ 0,\ N_x)$$
$$T = V \otimes N_2$$
$$T = \frac{T}{|T|}$$
$$B = T \otimes N$$
$$B = \frac{B}{|B|}$$

En el caso de *mesh*, se utilizó como referencia el artículo "*Tutorial 13 : Normal Mapping*" referenciado en la bibliografía. Para determinar la tangente y la bitangente, se utilizan 2 aristas del triángulo. A ellos, se los expresa en función de las coordenadas uv y de la tangente y binormal. Es decir,

$$E_1 = \Delta UV_x^1 \times T + \Delta UV_y^1 \times B$$
$$E_2 = \Delta UV_x^2 \times T + \Delta UV_y^2 \times B$$

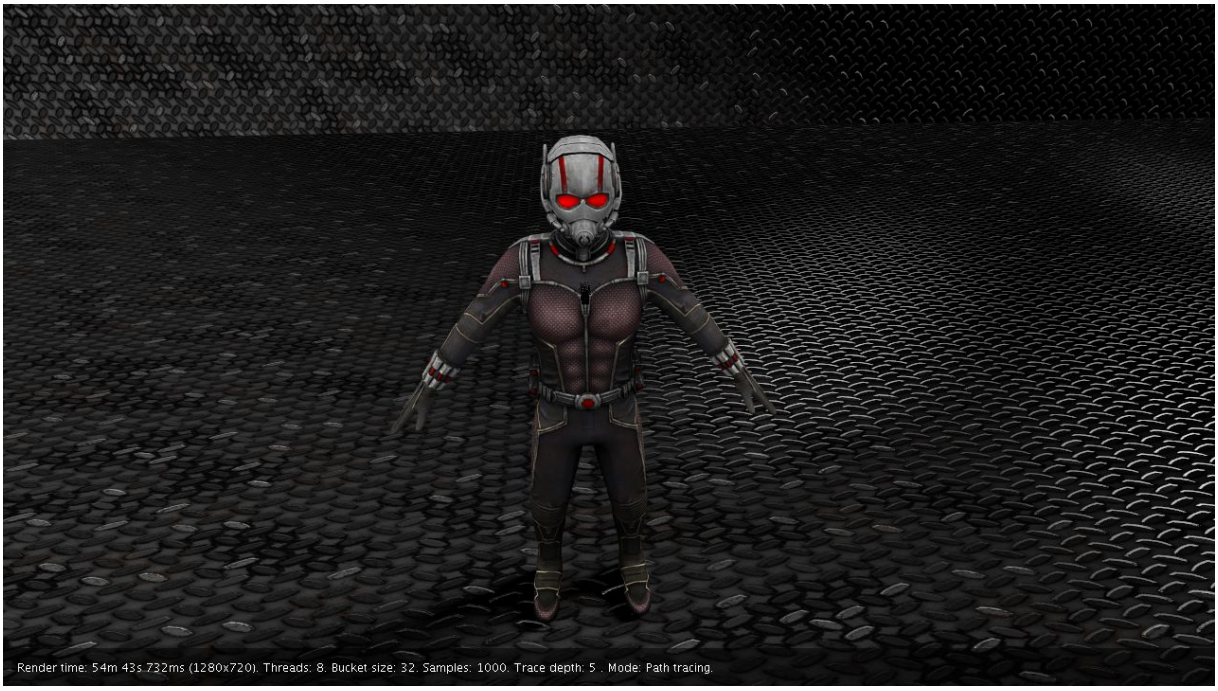
donde, $E_1 = V_2 - V_1$, $E_2 = V_3 - V_1$, $\Delta UV^1 = UV_2 - UV_1$, $\Delta UV^2 = UV_3 - UV_1$ y V_1, V_2 y V_3 son vértices.

Despejando estas ecuaciones, se obtienen la tangente y bitangente. En la siguiente imagen, se pueden observar normal maps sobre las paredes, la esfera y el cubo:



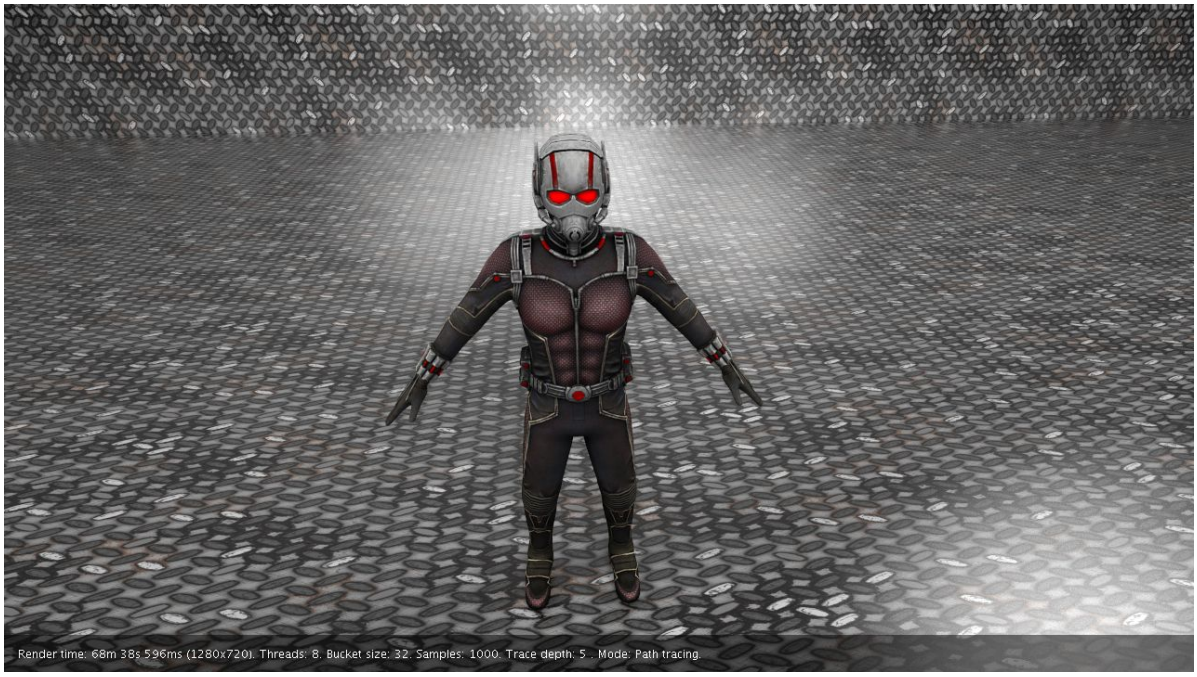
Normal Map Cornell Box (6)

Se puede observar como con geometrías simples, se logra geometría más compleja. La siguiente imagen cuenta con un *mesh* de *Antman* con *normal maps* y con planos que también cuenta con *normal maps*:



Antman (7)

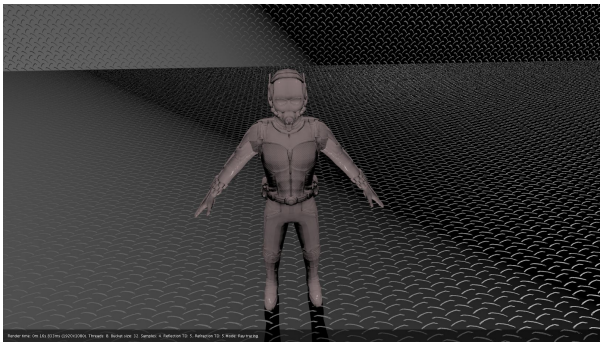
A modo comparativo, se muestra la misma escena, pero sin *normal maps*:

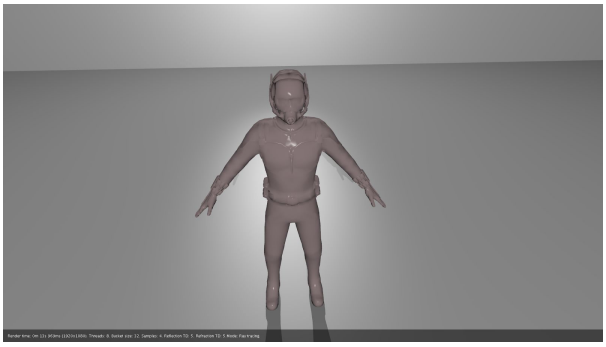


Antman (8)

Se puede observar que el piso es plano por lo que no tiene queda bien la textura que cuenta con relieve. A su vez, el traje del personaje cuenta con mucho más detalle en la imagen que cuenta con *normal maps* y se puede observar el relieve sobre el traje. Además, al tener normales diferentes los reflejos son menos intensos, lo que le aporta más realismo a la imagen.

A continuación se compara la misma escena (renderizada utilizando *raytracing*) pero sin sus respectivas texturas para resaltar más las diferencias:



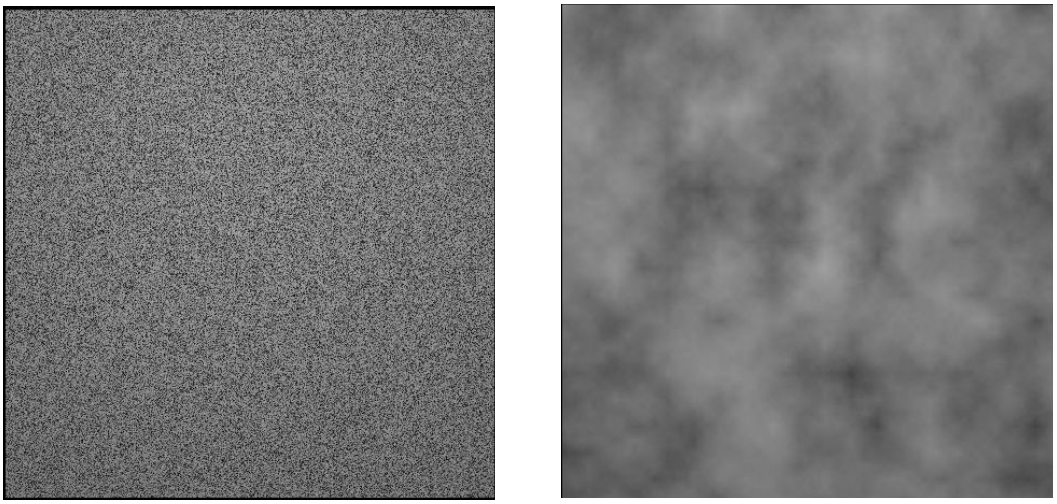


Izquierda: Simple Antman (9) Derecha: Simple Antman (10)

Texturas Procedurales

Otro *feature* implementado fue texturas procedurales. Se implementó dicha funcionalidad porque nos resultaba interesante poder generar texturas sin la necesidad de un artista. Además, al no ser determinísticas cada *render* pasa a ser único. Para la implementación de las texturas procedurales, se utilizó como referencia el tutorial "Texture Generation using Random Noise", el cual detalla cómo implementar de forma fácil los tipos de textura madera, mármol y nubes. Para la implementación de Voronoi, se realizó una implementación propia.

Para comenzar, se creó la clase Noise, la cual almacena una matriz de valores aleatorios entre 0 y 1. Estos valores se utilizan en la generación de texturas de los tipos de madera, mármol y nubes. La clase cuenta con el método *turbulence*, el cual es utilizado para acceder a los valores almacenados, pero de forma promediada. Internamente, el método *turbulence* utiliza otro método llamado *getSmooth*, el cual lee valores de la matriz de numeros, pero no realiza una lectura de un número individual, si no que lo promedia con sus vecinos. Entonces, *turbulence* realiza una suma de N de estos promedios, tomados a distintos niveles de zoom. Los resultados de estos valores interpretados como valores RGB grises se pueden observar en las siguientes imágenes:

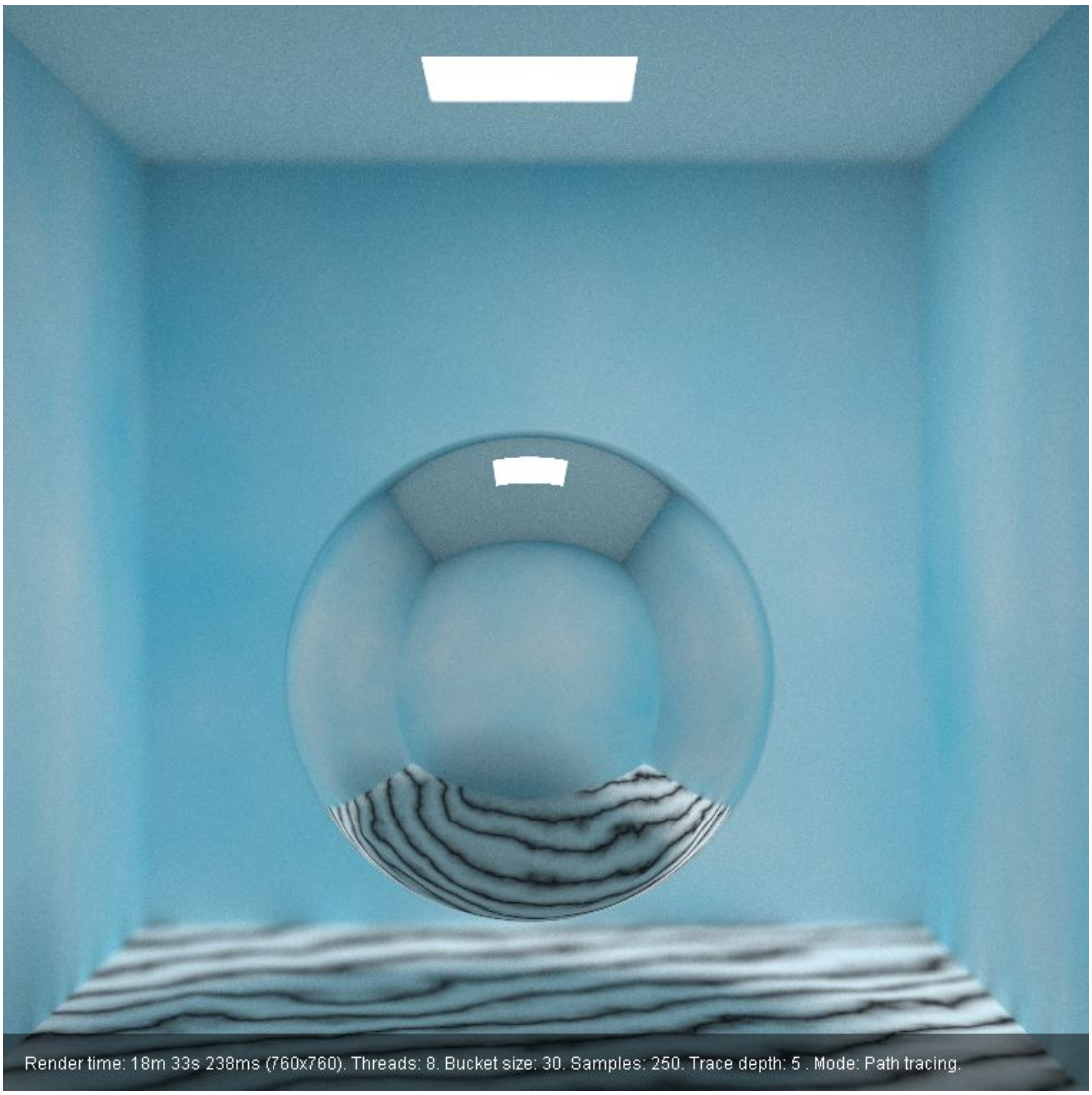


Izquierda: Valores contenidos en Noise sin procesar, Derecha: valores leídos utilizando turbulence

Para la generación de las texturas de nubes, se utilizaron los valores leídos con *turbulence*, con la opción de variar la "densidad" de las nubes (el cual modifica el nivel

de zoom tomado en turbulence). Los valores son luego modificados para que los más bajos se aproximen al color del cielo, y los más altos al color de las nubes.

En el caso del material mármol, se utiliza la función seno en combinación con los valores tomados de *turbulence*. Primero, por cada pixel a generar, se lee un valor para ese *x* e *y* utilizando *turbulence*, y luego se lo suma a dos factores de estiramiento distintos en cada eje. Luego, el valor resultante es evaluado por la función seno, con lo que se obtiene un patrón de líneas distorsionadas, dependiendo de los factores elegidos. A continuación se muestra una escena utilizando texturas de nubes y marmol:



Marble (11)

En el caso de la generación de texturas de madera, se utiliza un método similar al de la generación de texturas de mármol. La mayor diferencia es que se utiliza la

distancia del pixel a procesar a el centro de la textura, y esa distancia se la multiplica por el valor devuelto por *turbulence*, y una vez realizado esto se aplica la función seno. El resultado final es la apariencia de “aros”, formados en los troncos de los árboles. La cantidad de aros a generar es parametrizable. A continuación se muestra un ejemplo de textura de madera:

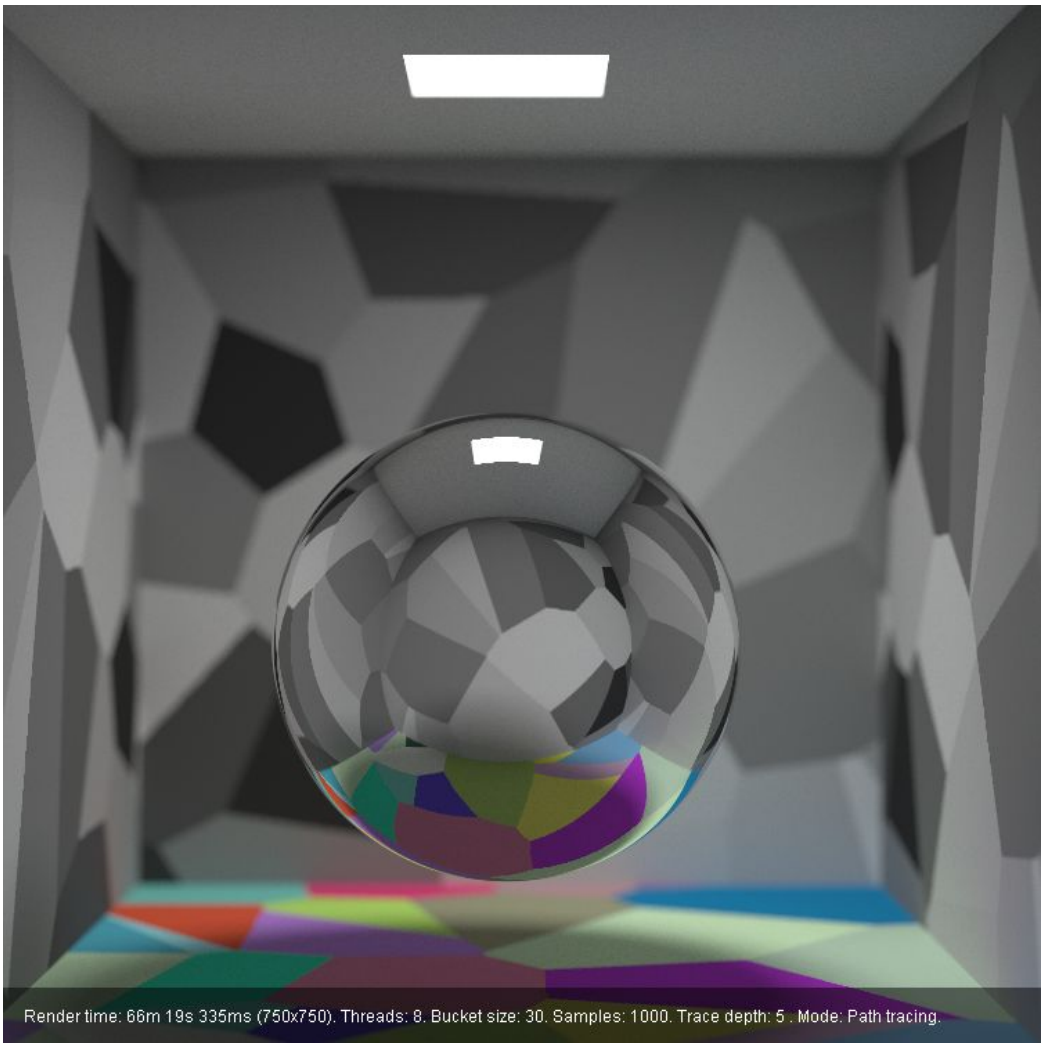
Wood (12)

La generación de texturas Voronoi es completamente distinta a la de las tres mencionadas anteriormente. Aunque existen varios métodos para generar patrones Voronoi, se decidió utilizar un método más simple, pero computacionalmente más exigente. Esto se debe a que se quiso priorizar la facilidad de leer el código y

modificarlo, y se consideró que la performance no era de tanta importancia ya que la generación de texturas se realiza una sola vez, al cargarse la escena.

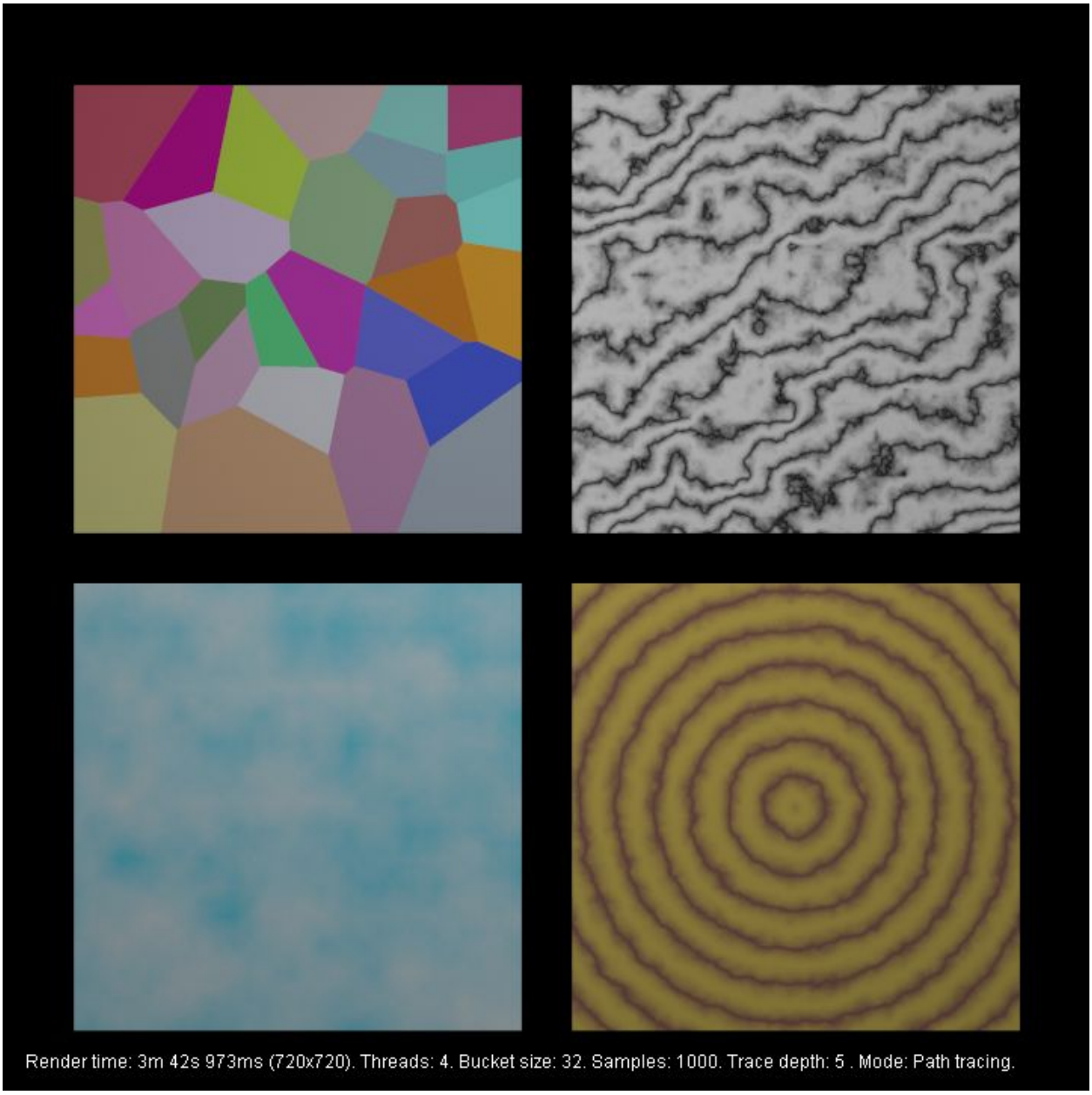
Para efectivamente generar la textura Voronoi, se utiliza primero un conjunto de muestras en un espacio 2D (la cantidad es parametrizable), que serán referidas como celdas. Por cada celda, se elige un color al azar y se lo asigna. Luego, se itera por cada pixel de la textura a generar, y se calcula cual es la celda más cercana. Luego, se asigna al pixel el color de la celda encontrada. La complejidad del algoritmo es alta ya que implica iterar por todos los píxeles, y por cada uno iterar sobre todas las celdas. Ademas, no se garantiza que dos celdas adyacentes tengan distintos colores, pero la probabilidad de que suceda no es significante.

Si se desea una textura Voronoi en escala de grises, se puede especificar utilizando el parámetro *blackAndWhite*. A continuación se muestra una escena que utiliza ambas variantes de texturas Voronoi:



Voronoi (13)

Para concluir, se muestra a continuación las cuatro texturas implementadas lado a lado:



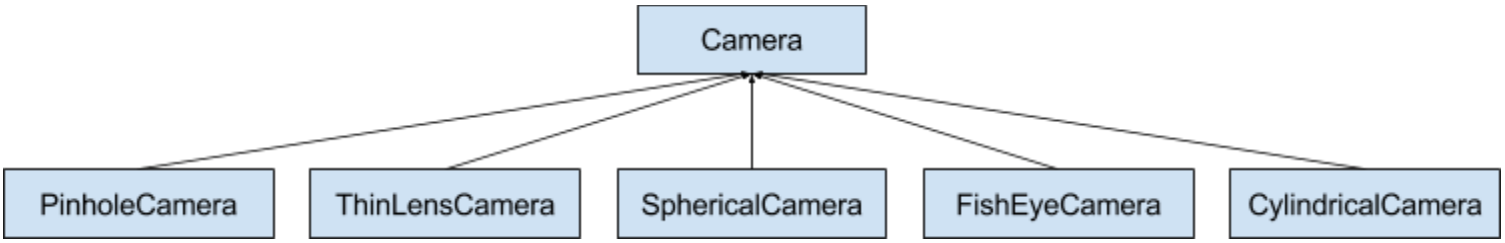
Procedurals (14)

Para poder utilizar las texturas procedurales, se realizaron algunos cambios a la clase *SceneParser*, y al formato de descripción de escenas. Al especificar una textura en la sección de assets, se puede especificar con el campo *textureType* el tipo de textura a utilizar. Si se especifica el valor *Image*, la textura será cargada a partir de los valores codificados con *base64*. Si no se especifica nada, el *default* es *Image*. Para utilizar las texturas procedurales, se puede utilizar los valores *Cloud*, *Wood*, *Voronoi* y

Marble. Se utilizan los parámetros *height* y *width* para especificar el tamaño de la textura. Como se explicó anteriormente, en el caso de *Cloud* se utiliza el parámetro *density*, en el caso de *Wood* el parámetro *rings* y en el caso de *Voronoi*, los parámetros *samples* y *blackAndWhite*. *Marble* no utiliza parámetros extra ya que para otros valores de *turbulence*, dejaba de ser *marble.* Por este mismo motivo, *turbulence* no es parametrizable en *Wood* tampoco.

Cámaras: Esférica, Cilíndrica y FishEye

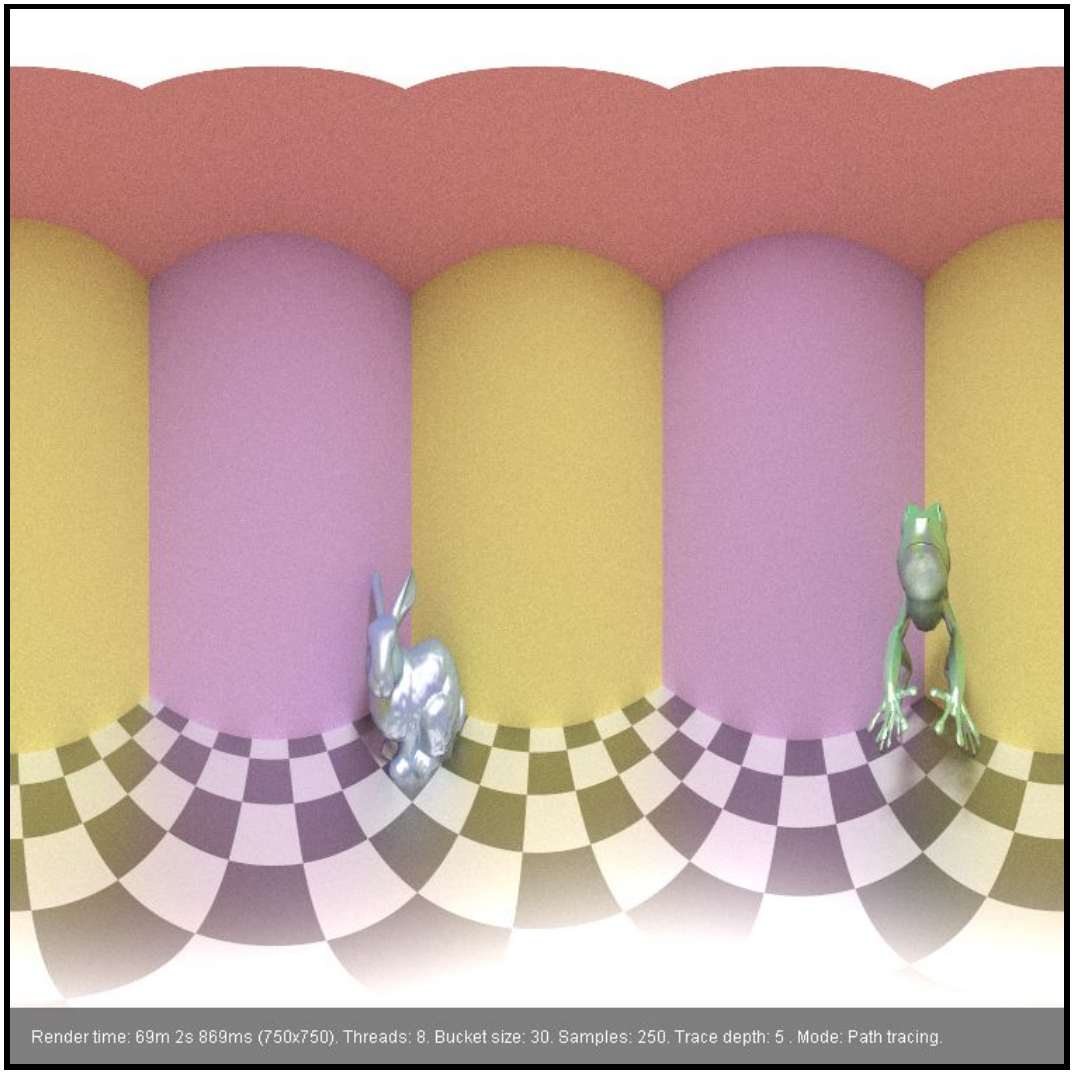
Por último, se implementaron distintos tipos de cámara. Se implementó esta funcionalidad debido a que se pueden generar *renders* interesantes que muestren más de cada escena. Si bien no FishEye no era requerido, se decidió implementar debido a que se utiliza mucho en el mundo de la fotografía. Para la implementación de los distintos tipos de cámaras (incluyendo la anteriormente explicada *ThinLens*), se decidió hacer cambios a la clase *Camera*. En primer lugar, se la convirtió a una clase abstracta, la cual cuenta con el método abstracto rayFor, el cual genera un rayo para un cierto pixel de la imagen a producir. Con qué direccion y origen se genera el rayo, es responsabilidad de la clase implementando *Camera*. Luego, se crearon las nuevas clases, cada una representando a un tipo de cámara (con *Pinhole* siendo la cámara clásica):



La implementación de las cámaras FishEye y esféricas fueron llevadas a cabo utilizando como referencia el libro “Ray Tracing from the Ground Up”. Se decidió que la cámara FishEye tenga un campo de visión de 180 grados ya que es el valor más comúnmente visto en este tipo de cámaras, y ya que no distorsiona demasiado la imagen en sus bordes (a diferencia de un campo de visión de 360 grados). La siguiente imagen muestra un render realizado con cámara FishEye:

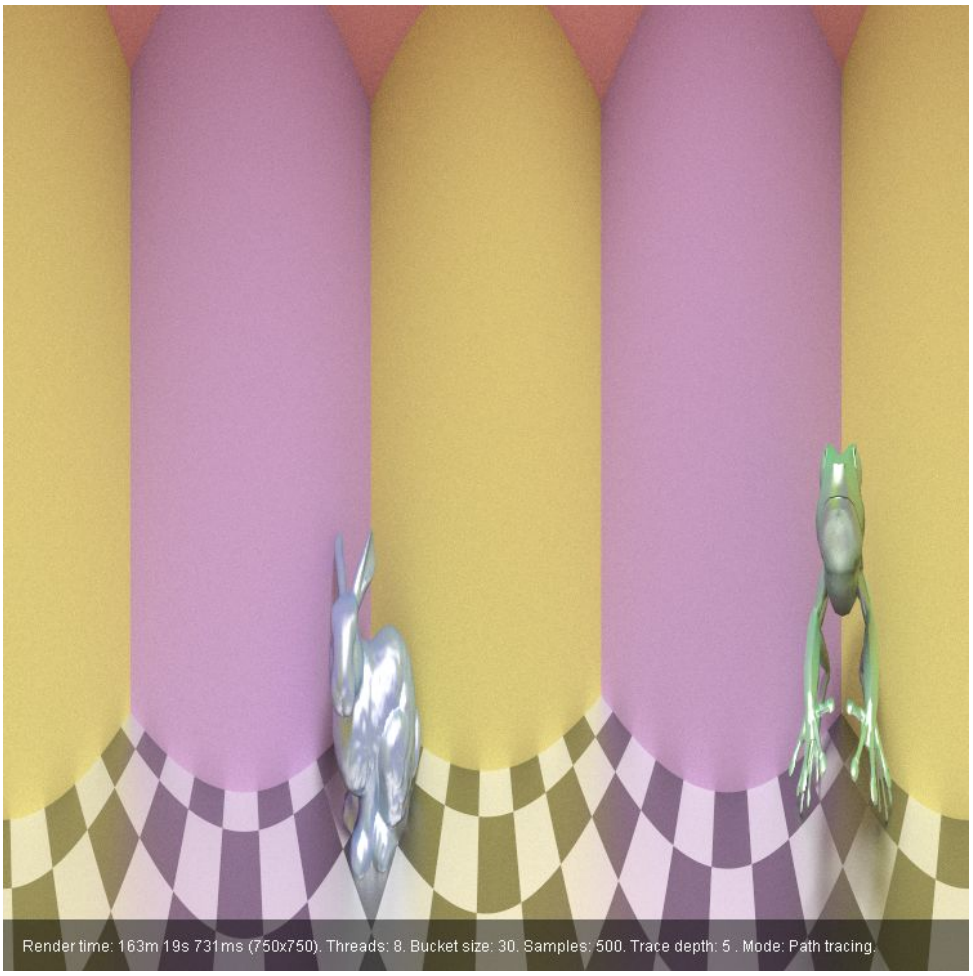
Frog (15)

En el caso de la cámara esférica, se tomo la implementación de libro mencionado, haciendo cambios en la dirección de algunos vectores, ya que se utiliza un sistema de coordenadas con direcciones distintas. Se creó una escena con dos modelos enfrentados para mostrar los tipos de cámara esféricos y cilíndricos. La siguiente imagen muestra la escena renderizada a través de la cámara esférica:



Spherical (16)

Se puede notar en la imagen que se puede ver toda la escena: las cuatro paredes, el techo (con la luz rectangular incluida), y el piso. Esta es la caracteristica principal de la cámara esférica; se captura la escena en todas las direcciones posibles. Por el otro lado, la cámara cilíndrica tiene otro comportamiento. En nuestra implementación, simplemente se generan rayos utilizando la coordenada **y** de la imagen como altura, y distribuyendo la coordenada **x** alrededor de una circunferencia. Los extremos superiores e inferiores de la coordenada **y** están dados por el parámetro de altura de la cámara. La siguiente imagen muestra la misma escena, renderizada a través de la cámara cilíndrica:



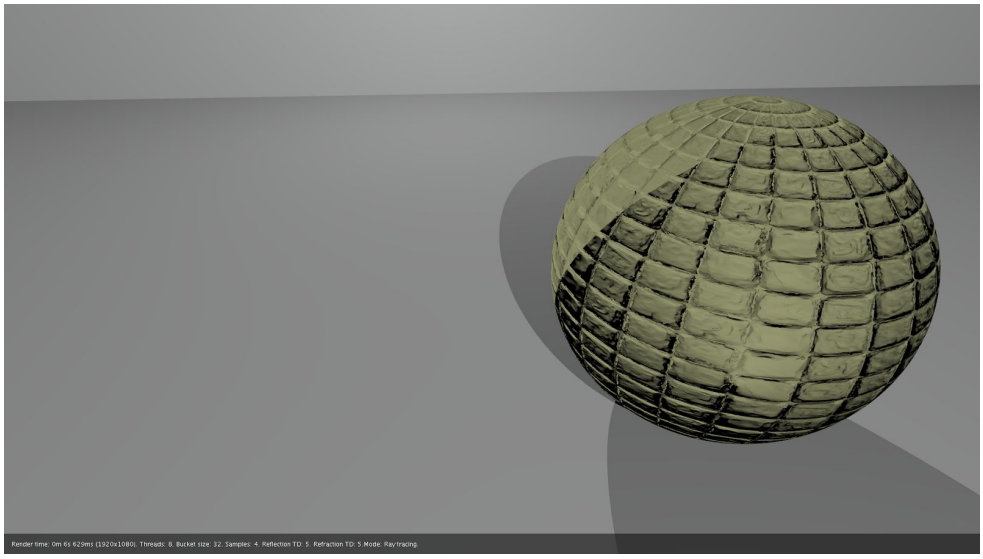
Cylindrical (17)

Se puede observar que la luz ya no es visible, ya que al estar posicionada directamente arriba del cilindro, los rayos no pueden alcanzarla. La parte del piso debajo de la cámara tampoco es visible. Utilizando una altura mayor, se puede lograr un efecto similar al de la cámara esférica, ya que los rayos con mayor y menor coordenada **y** tendrán una direccion casi vertical.

Para especificar el tipo de cámara, se debe usar el campo *cameraType* de la cámara en el archivo de descripción de la escena. Los valores aceptados son: *Pinhole*, *ThinLens* (explicado anteriormente), *Cylindrical* (con el parámetro *height* adicional), *FishEye*, y *Spherical*.

Problemas encontrados:

Uno de los problemas que se encontró fue a la hora de implementar *normal maps* para esferas. Se buscó mucho en internet sin obtener resultados. Una de las alternativas que se probó fue obtener la tangente haciendo el producto vectorial entre el eje con la componente mejor y la normal. El resultado fue el siguiente:



Otro problema que se encontró fue a la hora de implementar las cámaras de tipo cilíndrico y esférico. En algunos casos, como se mencionó anteriormente, la cámara cilíndrica da resultados similares a la de la esférica, dependiendo del parámetro de altura. Por lo tanto, al principio no se pudo decidir si la implementación de la cilíndrica estaba realizado correctamente. Al experimentar con el parámetro de altura, se llegó a la conclusión de que la implementación funcionaba bien, y que simplemente había que elegir la altura de forma tal que los objetos directamente abajo y sobre la cámara no fueran renderizados.

Benchmarks

A continuación se presentan los *benchmarks* de las imágenes presentes en el informe. Se pueden encontrar más imágenes en la carpeta *renders.*

Nombre	Size	Samples	Threads	Bucket Size	Max Trace Depth	Hardware	Tiempo
Depth of Field (1)	1280 x 720	1000	8	32	5	Core i7 2.6ghz (3720QM)	36m 49s 299ms
Depth of Field (2)	1280 x 720	1000	8	32	5	Core i7 2.6ghz (3720QM)	39m 12s 772ms
Depth of Field (3)	1280 x 720	1000	8	32	5	Core i7 2.6ghz (3720QM)	36m 47s 996ms
Stormtrooper (4)	1280 x 720	40000	8	32	5	Core i7 2.6ghz (3720QM)	11h 53m 41s 621ms
Hearts (5)	1280 x720	10000	8	32	5	Core i7 2.6ghz (3720QM)	1h 26m 32s 336ms
Stormtrooper Bokeh (6)	1280 x720	1500	8	128	5	Core i7 3.5ghz (4770K)	2h 14m 31s 549ms
Antman (7)	1280 x720	1000	8	32	5	Core i7 2.6ghz (3720QM)	1m 25s 703ms

Antman (8)	1280 x720	1000	8	32	5	Core i7 2.6ghz (3720QM)	1m 36s 270ms
Simple Antman (9)	1920 x108 0	4 (*Raytraci ng*)	8	32	5	Core i7 2.6ghz (3720QM)	16s 833ms
Simple Antman (10)	1920 X 1080	4 (*Raytraci ng*)	8	32	N/A	Core i7 2.6ghz (3720QM)	13s 969ms
Marble (11)	760x 760	250	8	30	5	Core i5 3.2ghz (4460)	18m 33s 238ms
Wood (12)	750x 750	300	8	30	5	Core i5 3.2ghz (4460)	53m 39s 277ms
Voronoi (13)	750x 750	1000	8	30	5	Core i5 3.2ghz (4460)	1h 6m 19s 335ms
Procedurals (14)	720x 720	1000	8	32	5	Core i5 3.2ghz (4460)	3m 42s 973ms
Frog (15)	750x 750	500	8	30	5	Core i5 3.2ghz (4460)	187m 32s 531ms
Spherical (16)	750x 750	250	8	30	5	Core i5 3.2ghz (4460)	1h 9m 2s 869ms
Cylindrical (17)	750x 750	500	8	30	5	Core i5 3.2ghz (4460)	2h 43m 19s 731ms

Bibliografía:

- Tutorial 13 : Normal Mapping
  - http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

- Physically Based Rendering: From Theory to Implementation - Second edition - Greg Humphreys y Matt Pharr (Julio 2010)
- Ray Tracing from the Ground Up - Kevin Suffern (2007)
- “Texture Generation using Random Noise” de **lodev.org**.
  - URL: http://lodev.org/cgtutor/randomnoise.html, Julio 2016.