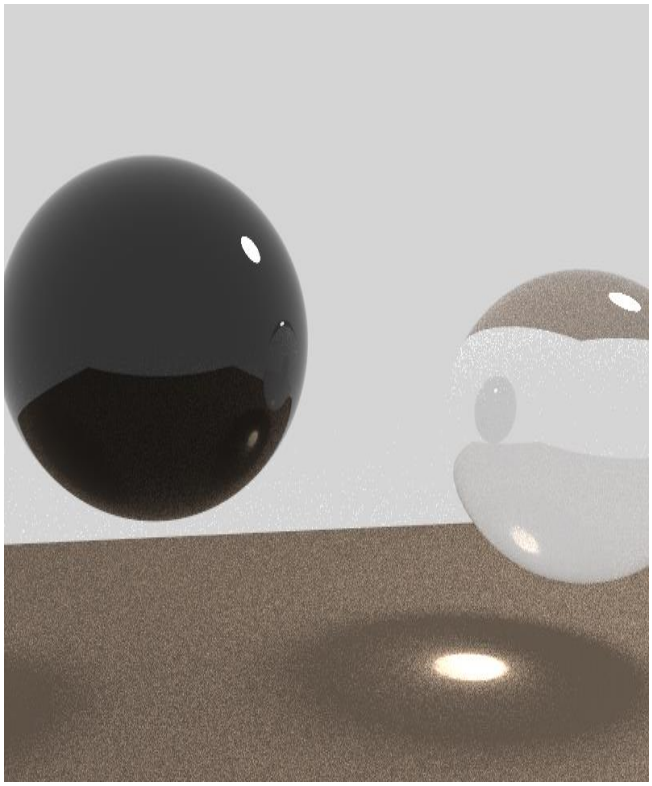


REALTIME PATH TRACER IN WEBGL 2

Rendering Avanzado

DESCRIPCIÓN BREVE

Desarrollo de un Path Tracer en tiempo real para la web usando WebGL 2

Cristian Rodríguez Bernal
Juan Guerrero Martín

Contenido

Introducción 4
Ray Tracer 4
  Objetos básicos 4
    Clase Renderable 4
    Clase Sphere 5
    Clase Box 5
    Clase Plane 6
    Clase Triangle 7
    Clase Light 7
  Sombras 8
  Iluminación 9
    Posición 9
    Intensidad 10
    Color de luz 10
    Energía de mapas de entorno 10
    Iluminación global 10
Path Tracer 11
  Mallas basadas en triángulos 11
    Clase Mesh 11
    Pintado de mallas simples basadas en triángulos 11
    Uso de texturas para subir vértices e índices 12
    Estructuras de aceleración 12
  Algoritmo Path Tracer 13
    Programación del cliente 13
    Programación del shader 13
Parte opcional 15
  Mapas tonales 15
  Escenas 16
    SceneLoader 16
    Cámara 16
    Escenas implementadas 17
  Environments 20
    Constante 20
    Mapas de entorno HDR 20
    Rotación 20
    Energía emitida 20

Materiales 21
Diffuse 21
Attenuate 21
PhongSpecular 22
Metal 22
Transmit 24
SmoothDielectric 25
Emisivo 26
Absorbente 26
Reflectivo 27
FresnelComposite 27
Texturado 27
FormulaTexture 27
ChessTexture 27
FormulaPowTexture 28
WorleyTexture 28
PerlinTexture 28
Normal Mapping 28
Depth of Field 28
Armónicos esféricos 29
Antialiasing y filtrado de texturas 32
Motion Blur 32
Apéndices 34
Cosas a tener en cuenta antes de empezar 34
Doble Buffer 34
Ping-Pong technique 34
Estructura del proyecto 34
Camera 34
Environments 34
Globals 34
HDRLoaders 34
Index 35
Input 35
Integrators 35
Materials 35
Renderable 35
QuadsToneMap 35

Scene 35
SceneLoader 35
ShaderProgram 35
SourceFrag 35
Utility 35
Compilación y desarrollo 35
Typescript 36
Grunt 36
BRDF 37
Esfera utilizando el método de rechazo 37
Esfera con muestreo uniforme 37
Hemiesfera utilizando un método de inversión como técnica de muestreo (Monte Carlo) 38
Librerías auxiliares 39
glMatrix 39
WebGL 2 40
¿Qué es? 40
Activar WebGL 2 en los navegadores actuales 40
Comparación WebGL y OpenGL 41
Problemas sin solucionar y trabajos futuros 41
Conclusiones 42
Bibliografía 43

Introducción

A la hora de tratar todos los puntos que conforman esta práctica, hay que señalar que la separación de tareas que se ha realizado se ha basado en la herramienta de control de Software online Taiga.io.

Utilizando la metodología Scrum[1], hemos separado cada Sprint [2]como cada uno de los diferentes bloques definidos por el profesor en la memoria de prácticas. Así mismo, cada tarea del Sprint lleva asociado un conjunto de subtareas que conforman las distintas "cajas" que forman el producto final.

Gracias a esta separación, nos resultó más fácil ir evolucionando hasta el producto final.

Antes de continuar leyendo es recomendable, para entender algunos conceptos y prácticas utilizadas en la aplicación, leer el apartado de Apéndices, en especial las primeras secciones, donde se detallan algunas técnicas habituales en aplicaciones gráficas para la web.

Se ha grabado un vídeo con el fin de demostrar el funcionamiento de algunas escenas. Es posible visualizarlo a través del siguiente link.

Por otro lado, se dispone de una versión online en una página de Github: http://maldicion069.github.io/PathTracerWebGL2/

Ray Tracer

En el primer sprint había que definir los objetos básicos a renderizar (esferas, cubos, planos y triángulos), así como la generación de sombras y la iluminación basada en una fuente de luz puntual.

Si bien en este sprint deberían formar parte tanto el algoritmo de Ray Tracer y los materiales de tipo Phong o difusos, lo primero se decidió descartar debido a la envergadura del proyecto con una tecnología con escasa documentación y lo segundo se traspasó al último sprint (en estas versiones únicamente se pintaban los objetos con sus normales).

Objetos básicos

Clase Renderable

Debido a la implementación de diversos tipos de objetos a pintar en nuestro Path Tracer, la primera clase en la que centrarse en este apartado es la clase abstracta *Renderable*. Esta clase contiene los distintos métodos y atributos que definen a los objetos renderizables.

En la implementación realizada de *Renderable*, podemos encontrar:

---

[1] Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

[2] En Scrum un proyecto se ejecuta en bloques temporales cortos y fijos (iteraciones de un mes natural y hasta de dos semanas). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto que sea potencialmente entregable, de manera que cuando el cliente (Product Owner) lo solicite sólo sea necesario un esfuerzo mínimo para que el producto esté disponible para ser utilizado.

- Constructor: Requiere tipo, *isDynamic* (permite que el objeto se pueda mover) y material asociado.
- ID, uniformes, prefijo (para identificar junto al ID a los objetos), material, fragmentos de código necesarios.
- *getUniformsDeclararation*, *update*, *getInsideExpression*,…: Estas funciones permiten editar algunos parámetros de los propios objetos renderizables, así como generar el shader final.

Clase Sphere

- Definición: Una esfera se define como un centro y radio. También se requieren los datos del constructor de Renderable:

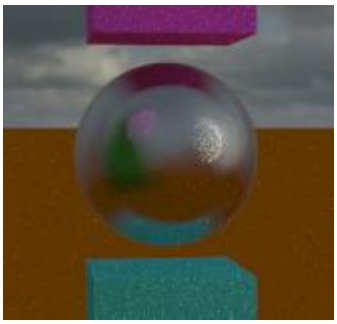
$$super(\text{“}sphere\text{”}, isDynamic, material)$$

- Calcular intersección: Para definir si una esfera ha interseccionado con un rayo, utilizamos la función *rayIntersectSphere*. Dentro de esta función, utilizamos la siguiente ecuación:

$$t = \frac{-d \cdot p \pm \sqrt{(d \cdot p)^2 - (d \cdot d)(p \cdot p - 1)}}{d \cdot d}$$

- Calcular normal: Para calcular la normal de una esfera, utilizamos la función *normalForSphere*. Ésta se define con la ecuación:

$$(posición\_hit \; - \; centro\_esfera) \; / \; radio\_esfera$$

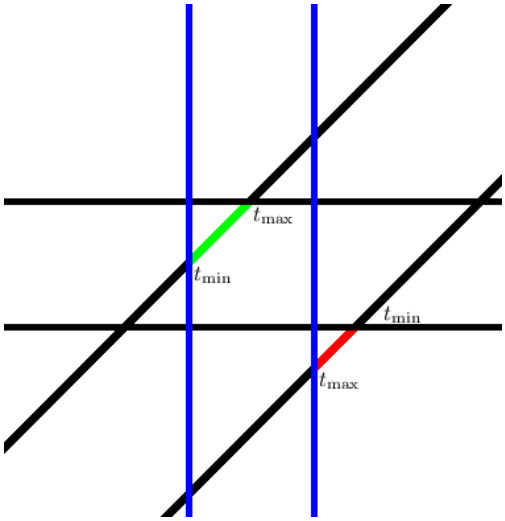


Clase Box

- Definición: Una caja se define como el punto mínimo y máximo de la AABB contenedora. También se requieren los datos del constructor Renderable:

$$super(\text{“}box\text{”}, isDynamic, material)$$

- Calcular intersección: Para definir si una esfera ha interseccionado con un rayo, utilizamos la función *rayIntersectBox*. Esta función se basa en el método conocido como *slab method*. Slab method trata una caja como el espacio en el interior de tres pares de planos paralelos. El rayo se recorta por cada par de planos paralelos y si se ha producido una intersección, se indica que ha habido una colisión entre caja y rayo.



- Calcular normal: Para calcular la normal de una caja, utilizamos la función normalForBox. Ésta se define con la ecuación:

$$(posición\_hit\ -\ centro\_cubo)\ /\ semidiagonal\_cubo$$

Clase Plane

- Definición: Un plano se define con su normal y una constante d. Siendo esta constante:

$$d = dot(planeNormal, p_0)$$

Siendo p_0 un punto cualquiera del plano.

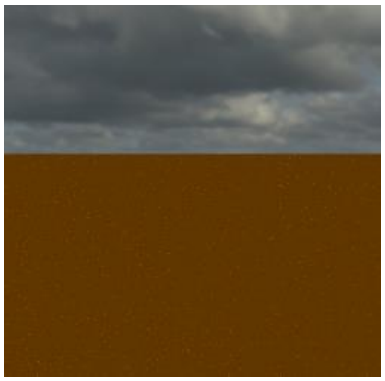
- Calcular intersección: Para calcular la intersección entre plano y rayo utilizamos la función rayIntersectPlane. Dentro de esta función, utilizamos la siguiente ecuación (en forma algebraica):

$$t = \frac{d - dot(planeNormal, rayOrigin)}{dot(planeNormal, rayDir)}$$

Si el denominador es 0, significa que el rayo y el plano son paralelos. En este caso, si el numerador es también 0 el rayo estará contenido totalmente en el plano. Si el numerador no es 0, no existirá intersección entre rayo y plano.

Por otro lado, si el denominador no es 0, habrá un único punto de intersección entre el plano y el rayo. Este punto se extrae sustituyendo el valor calculado de t en la ecuación paramétrica del rayo.

- Calcular normal: Debido a que el plano solo tiene una única normal, no es necesario calcular de alguna manera la normal. Si el objeto interseccionado más próximo al punto de vista del observador/cámara es el plano, la normal del rayo es la normal del propio plano.



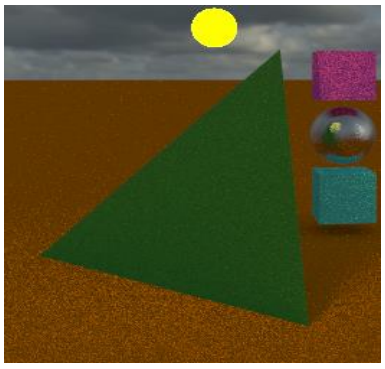
Como podemos comprobar, los planos (por su definición) son infinitos en toda la escena.

Clase Triangle

- Definición: Un triángulo se define con tres vértices. En cuanto a la llamada al constructor padre, utilizamos:

$$super(\text{“}triangle\text{”}, isDynamic, material)$$

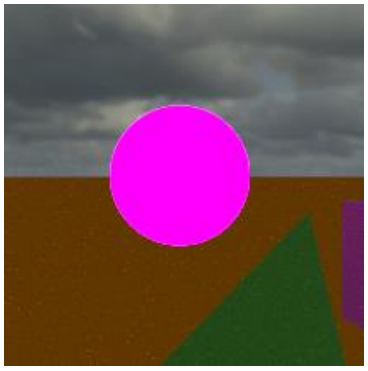
- Calcular intersección: Calcular la intersección de un triángulo resulta una tarea bastante más compleja que en los casos anteriores. Para comprobar si una rayo ha interseccionado con un triángulo se requiere comprobar: Se examina la orientación de la faceta (comentado) y si las coordenadas *uv* del triángulo se encuentran entre *[0.0, 1.0]*. La función que se encarga de esta tarea es *intersectTriangle*.
- Calcular normal: La normal se calcula con la función *normalForTriangle*. Esta función se encarga de realizar el producto vectorial normalizado entre la diferencia de dos parejas de vértices $((v1 - v0)\ x\ (v2 - v0))$. Es importante el orden en el que se realiza el productor vectorial porque se puede marcar como la dirección de la normal en la cara inversa. También es importante remarcar la normalización, ya que en las primeras fases de desarrollo de este algoritmo tuvimos bastantes problemas con ello.



Clase Light

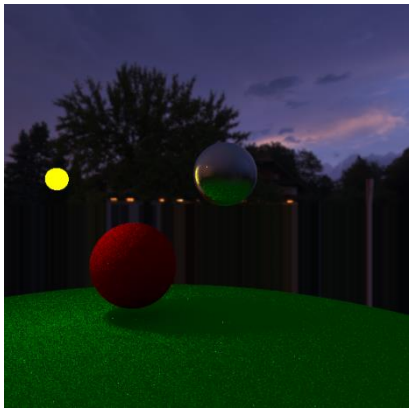
- Un objeto de tipo Light se trata de una extensión de la clase esfera, con los atributos *isDynamic* a true y un material de tipo Emisivo y Absorbente.

- Se requiere al menos una luz para toda la escena, ya que será utilizada para el cálculo de sombras. La configuración de esta se hace a través de los distintos sliders y botones explicados en el apartado de Iluminación que se contará más adelante.
- De forma adicional, se pueden incluir más luces, las cuáles no se podrán configurar, y su principal funcionalidad será mejorar la iluminación global de la escena, así como aportar nuevas sombras o cáusticas.



Sombras

El cálculo de sombras en nuestra aplicación se produce de forma automática, sin necesidad de recalcular las sombras en una segunda interacción.

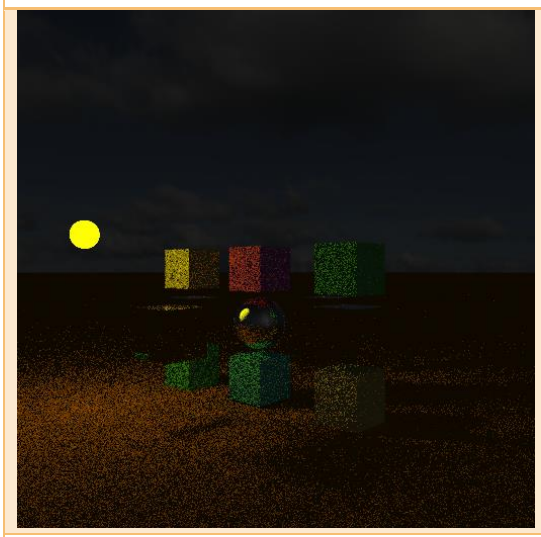
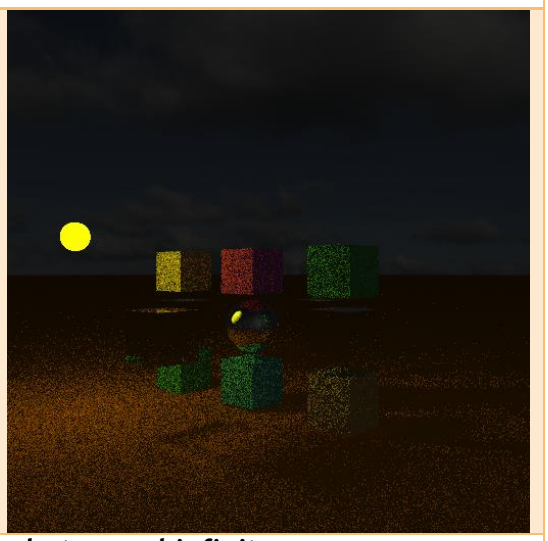


Como se puede comprobar en la anterior imagen, obtenemos sombras suaves que se difuminan según se alejan de los objetos.

Por otro lado, se realizaron dos pruebas con una segunda pasada de sombras, pero los resultados eran bastante menos realistas. En el primer caso las sombras se difuminaban bastante más, y permitía más áreas con menos sombras, y en el segundo las sombras eran más duras (y oscuras), pero a cambio impedía mostrar los mapas de entorno.

Finalmente decidimos optar por la solución de sombreado forzado en cada rebote, ya que permitía sombras menos duras y más suavizadas. Esta implementación siempre requerirá que exista un objeto “sphere1center” que representará la posición de la luz (con el cargador de escenas que se contará más adelante, la luz siempre se carga en primera posición en la lista de objetos de la escena). A diferencia de otros sistemas de renderizado, no existe una luz única, sino que son los propios materiales de los objetos los que determinan la energía captura o resultante.

En cuanto al algoritmo de generación de sombras, se realiza una segunda pasada por cada objeto de la escena (a excepción de planos, ya que las normales afectan a la ecuación de sombreado). Si se detecta una colisión, se marca como sombra y se multiplica con la ecuación básica de iluminación al color final.

Sombreado normal	Sombreado forzado en cada rebote
	

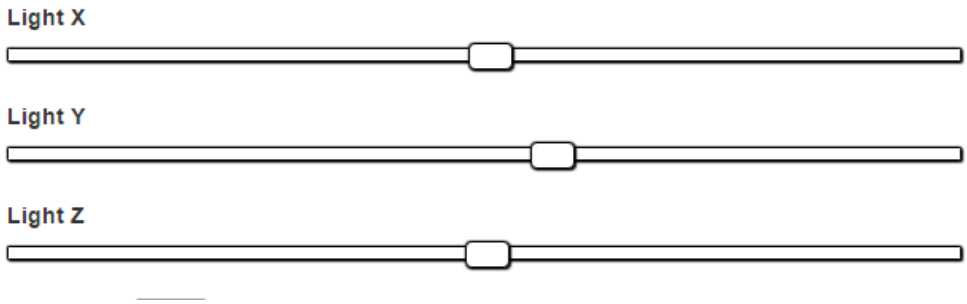
Sombreado forzado en rebote en el infinito

Iluminación

Esta primera tarea se descompone en un total de cuatro subtareas, las cuales son:

Posición

Con el fin de permitir cambiar la posición de las sombras o los efectos producidos por las fuentes de luz. Para ello se han incluido tres sliders en la interfaz que se encargan de gestionar la posición en X, Y e Z, respectivamente, en un área delimitada entre *[-300, 300]* en los tres ejes.



Intensidad

Cuanta más intensidad tenga la fuente de luz, mayor área es capaz de iluminar. Por la misma razón anterior, se incluye un slider que es capaz de aumentar el radio de intensidad entre *[0, 300]*.

Color de luz

Una variación en el color de la luz permite iluminar los objetos de la escena con diferentes mezclas de colores. Debido a las limitaciones entre la intensidad de la luz y el color seleccionado, una gran área de intensidad varía el color del objeto que representa la fuente de luz en los canales RGB para encontrarse entre o bien 0 o bien 1, siendo la estela alrededor del resto de área del color seleccionado.

Por ejemplo, un color (1.0, 0.7, 0.0) representa el color naranja: La fuente de luz se sombrea de color amarillo (1.0, 1.0, 0.0), y la estela de color anaranjado (1.0, 0.7, 0.0).

Botón que modifica el color de la luz

Energía de mapas de entorno

Los mapas de entorno son muy utilizados para simular un paisaje de forma barata y eficiente. Debido a que se han utilizado mapas basados en HDR, la cantidad de energía que emiten es bastante considerable, lo cual afecta al resto de la escena.

Se hablará más adelante sobre los Environments, centrándonos en la explicación de su funcionamiento, así como posibilidad para rotar el cielo y cambiar la cantidad de luz emitida hacía la escena.

Iluminación global

En la aplicación se han utilizado un total de dos tipos de métodos de iluminación global en tiempo real: Path Tracer y Esféricos Armónicos.

Como se podrá comprobar en todo el presente documento, estos dos métodos calculan autosombras y sombras directas e indirectas.

Un ejemplo básico de iluminación global es comprobar una caústica dentro de una mesa, como se puede observar en la foto de portada de este documento.

Path Tracer

Segundo sprint del proyecto. Durante esta fase las tareas principales fueron la integración de mallas de triángulos (a mano y gestionadas con texturas) y el desarrollo del algoritmo de Path Tracer con MonteCarlo.

Mallas basadas en triángulos

Una vez completada la funcionalidad referente a los objetos de tipo triángulo, es hora de representar una malla. Una malla es, por definición en gráficos, un conjunto de triángulos, que pueden estar indexados o no.

Debido a que la representación de mallas no es una tarea trivial, esta tarea se ha separado en un conjunto de cuatro etapas, desde la creación de la clase específica hasta estructuras de aceleración para evitar el uso de algoritmos de fuerza bruta.

Clase Mesh

- Definición: Una malla está definida como un conjunto de triángulos. En la versión utilizando texturas se desechó el conjunto de triángulos a cambio de utilizar texturas. En esta versión solo se almacenan los identificadores de texturas y su tamaño en una dimensión (se hablará de ello en las siguientes líneas). Como en el resto de clases que extienden *Renderable*, la llamada al constructor padre es:

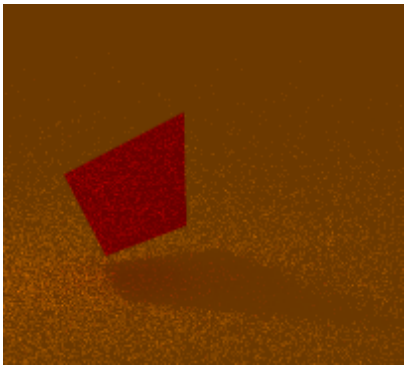
$$super(\text{“}mesh\text{”}, isDynamic, material)$$

- Calcular intersección: Para calcular la intersección utilizamos la función *intersectMesh*. Internamente se encarga de calcular la intersección con cada triángulo a través de un algoritmo de fuerza bruta.
- Calcular normal: La normal de las mallas es igual a la normal de cada uno de los triángulos. Por ello, utilizando la función *normalOnMesh*, calculamos la normal del triángulo con el cual el rayo ha interseccionado.

Pintado de mallas simples basadas en triángulos

Una malla en su definición es un conjunto de triángulos. Es por ello que, en una primera instancia, decidimos fijar varios vértices dentro del propio shader. Dentro de la parte de intersecciones, comprobamos mediante fuerza bruta si el rayo actual ha interseccionado con cada triángulo.

Una vez comprobado que este paso funcionaba, se decidió incorporar una mejor manera de enviar vértices a la tarjeta gráfica sin necesidad de instanciarlos dentro del propio shader.



Uso de texturas para subir vértices e índices

Antes de adentrarse en la subida de vértices a la tarjeta de vídeo, era necesario un parser de ficheros *Wavefront OBJ* a JSON, con el fin de que fuesen fáciles de tratar para un navegador web. Para realizar esta tarea, se creó una tarea automatizada con *Grunt*.

Dentro de la clase *Mesh* mencionada líneas atrás, se incorporó un lector a través de *XMLHttpRequest* para leer el JSON del modelo a tratar.

Para tratar los vértices, creamos una textura 1D de flotantes, mientras que para los índices se creó una textura 1D de *unsigned short*, que se subirían a la tarjeta de vídeo en cada iteración del integrador. Junto a ambas texturas, se requieren dos variables que delimitan la cantidad de vértices e índices.

En cuanto al tratamiento en GPU, la forma de acceder a los triángulos se ha realizado de la siguiente manera:

- Por cada índice desde 0 hasta el número de índices, calculamos la intersección de los triángulos.
- Dentro de la intersección con triángulo, accedemos a los vértices del triángulo a través de las posiciones XYZ obtenidas de la textura de índices y el índice actual del bucle de intersecciones.
- Si interseca, actualizamos la posición del hit más cercano a la cámara. Si no, seguimos hasta pasar por todos los triángulos de la escena.

Nota: Tras pasar varias horas intentando solucionar cuál era el fallo por el que los shaders no llegaban a ejecutarse bien, nos dimos cuenta de que en WebGL no se permite crear bucles de tipo *for* que accedan a texturas (o al menos, no funciona bien en la mayoría de tarjetas gráficas).

Es por esta razón por la que el código asociado a los shaders que usan mallas tienen bastante código repetido.

Estructuras de aceleración

Debido a que en cada rebote es necesario calcular si un rayo ha interseccionado con cada uno de los triángulos de la malla en forma de fuerza bruta, es necesario optimizar esta fuerza bruta a través de una estructura de aceleración que simplifica la intersección con la malla. Es por ello que utilizaremos la estructura de aceleración más básica: AABB.

AABB permite filtrar la fuerza bruta, de forma que si un rayo no ha interseccionado con la caja contenedora, se evita comprobar cada uno de los triángulos.

Los pasos utilizados para esto han sido:

1. Obtención del punto mínimo y máximo del conjunto de vértices de la malla y enviarlo al shader de fragmentos.
2. Dentro del proceso de intersecciones en el Path Tracer, comprobamos si se colisiona contra el AABB.
    a. Si colisiona: Comprobamos cada uno de los triángulos de la malla.
    b. En el resto de casos: No se comprueban los triángulos de la malla.

Existen multitud de técnicas basadas en jerarquías, pero no todas son tan fácil de extrapolar a código de Shader, debido a que requiere grandes cantidades de memoria

para procesar las partes visibles, así como algoritmos puramente recursivos, incapaces de ejecutarse en GLSL.

Algoritmo Path Tracer

El algoritmo de Path Tracer en tiempo real implementado está basado en una técnica muy habitual en gráficos por computador para la web. Debido a que la web no dispone de técnicas como **swapBuffers** (el navegador hace una interpretación propia de esta funcionalidad), se ha simulado mediante una técnica conocida como Ping-Pong (no se detallará nada de esto en este apartado, queda el lector libre de leer el apéndice para saber más).

Programación del cliente

En el lado cliente utilizamos dos texturas y un Framebuffer para simular esta técnica.

El Path Tracer implementado se realiza en dos pasadas:

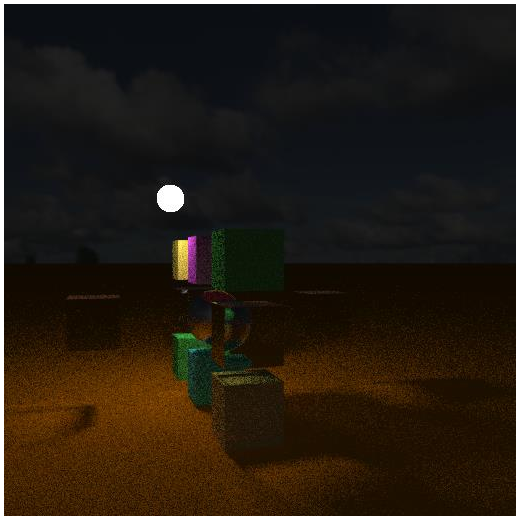
1. Función *step*: Esta primera pasada hace un render sobre una textura con ayuda del FBO el resultado de realizar N número de rebotes (por defecto 8) realizando un mezclado con la textura anterior (la cual se le manda como una textura al shader). Aumenta en uno el número de vueltas (*weight*) realizadas.
2. Función *render*: Utilizando los distintos mapas tonales, carga el contenido del FBO y lo pinta sobre el Framebuffer por defecto. Antes de finalizar, es importante hacer un intercambio de las texturas.
   Debido a la implementación realizada, no es necesario realizar el volcado de una textura sobre la otra, solo se cambia la textura a pintar.

Según el número de pasadas que se requieran sobre la escena, la función *step* dejará de realizar procesos de render. Por defecto realiza un máximo de 32 pasadas, pero mediante la interfaz se pueden lograr desde 0 (infinitas pasadas) hasta 512.
De la misma manera, podemos configurar el número de rebotes por cada pasada. Por defecto son 8 pasadas, pero el usuario puede configurar desde 1 pasada hasta 16 (más de 9-10 no es recomendable si tu ordenador no tiene mucha potencia).

Programación del shader

En cuanto a la programación del shader principal de renderizado (el de la función *step*), los pasos son los siguientes:

1. Desde 1 hasta N número de rebotes, se realiza una comprobación con todos los objetos de la escena para obtener el objeto interseccionado más cercano a la cámara.
2. Una vez identificado el objeto, se aplican las ecuaciones del material o materiales que tiene el objeto, así como la asignación de las normales y (no siempre), la atenuación.
3. Antes de empezar el siguiente rebote, añadimos al acumulador el color difuso respecto a la luz y el sombreado.
4. Debido a que el acumulador tiene bastante energía, se le reduce dividiendo entre el número total de rebotes.
5. Finalmente, mezclamos el color final con el color anterior. De esta forma, vemos como la escena va formando, poco a poco, la imagen final.



Parte opcional

En esta última fase se concentra toda la parte más compleja de la aplicación, pues nos encargamos de crear diferentes mapas tonales, escenas, environment, todos los materiales y algunos extras (normal mapping, depth of field, blur,…). De forma adicional, en la última versión del proyecto, se incluyeron Armónicos Esféricos y varias configuraciones globales del proyecto.

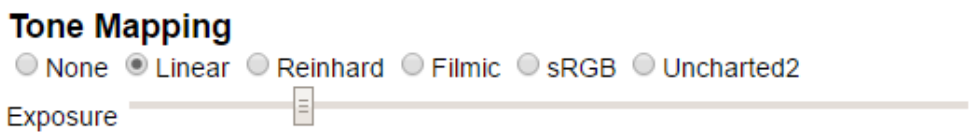
Mapas tonales

Se han incorporado un total de cinco mapas tonales más uno sin cambios en la exposición de la imagen. Estos mapas tonales son:

QuadSimple	QuadGamma/Linear	QuadReinhard
		
QuadFilmic	**QuadsRGB**	**QuadUnchard2**
		

Los mapas tonales se aplican sobre la propia imagen final, por lo que, gracias al sistema de renderizado basado en Ping-Pong, podemos acceder al último render y realizar en esta textura un mapeo tonal.

Se puede cambiar entre un tipo de mapa tonal u otro a través de:



Los distintos mapas tonales incorporados a esta aplicación están disponibles en los siguientes links:

https://www.shadertoy.com/view/lslGzl

http://filmicgames.com/archives/75

Escenas

La tarea relacionada con escenas se ha descompuesto en tres subtareas, las cuáles son un parseador de escenarios (SceneLoader), la cámara y distintas escenas interactuables.

SceneLoader

Esta clase se encarga de cargar diferentes escenas a través de un XML. El constructor, a través de un fichero XML, carga toda la escena y, con los métodos públicos de la clase, muestra una interfaz que devuelve todas las esferas, cajas, planos, mallas y triángulos definidos. También define tanto el foco de luz como la posición de la cámara al iniciar la escena.

Cada fichero de escena está definido a través del siguiente esquema (en este caso se muestra un ejemplo sencillo):

```
<?xml version="1.0" encoding="UTF-8"?>
<SceneLevel dotWeight="true">
	<Camera position="-201.60 65 4.9" up="0, 1, 0" yaw="180.0" pitch="0.0" />
	<Light position="10 125 9" radius="10" color="1 1 1" intensity="100" />
	<Objects>
		<!-- Walls. -->
		<!--
			Outer: 210.
			Inner: 200.
			Height: 125.
		-->
		<Box min="-10 115 -11" max="30 145 29">
			<Emit>
				<Absorb intensity="69" color="0.3 0.3 1" />
			</Emit>
		</Box>

		<Box min="-210 0 100" max="210 125 110"><!-- Left -->
			<Attenuate color="0.0 0.0 1.0"><!-- Blue -->
				<Diffuse />
			</Attenuate>
		</Box>

		<!-- Spheres -->
		<!--
			Height: 60.
			La Z nos es más util que la X.
		-->
		<Sphere center="0 60 30" radius="20">
			<Metal iorComplex="2.1928 3.9379" glossiness="0.1" montecarlo="1.0" />
		</Sphere>
		<Sphere center="0 60 -30" radius="20">
			<Metal iorComplex="2.1928 3.9379" glossiness="0.1" montecarlo="0.0" />
		</Sphere>

	</Objects>
</SceneLevel>
```

Cámara

Para gestionar los movimientos de la cámara, hemos implementado una clase específica que simula una cámara libre.

Debido a que existen muchas implementaciones de cámaras por la red, hemos usado la que utilizamos en la primera práctica de Rendering Avanzado. La implementación de la cámara utiliza ángulos de Euler, y gran parte del código utilizado proviene de http://www.learnopengl.com/#!Getting-started/Camera (salvo algunos detalles de implementación propia como generación de matriz de proyección u otros tipos de movimiento de la cámara).

La cámara se define a través de cuatro datos:

- Posición del observador.
- Vector Up.
- Yaw.
- Pitch.

En cuanto a los atajos de teclado, la configuración utilizada ha sido:

Tecla	Uso
W	Avanza la cámara hacia arriba.
A	Avanza de lado hacia la izquierda.
S	Igual que W pero hacia abajo.
D	Igual que A pero en sentido inverso.
Q	Avanza la cámara hacia delante
E	Igual que Q pero en sentido inverso
Flecha arriba	Gira la cámara en el eje vertical hacia arriba.
Flecha izquierda	Gira la cámara en el eje horizontal hacia la izquierda.
Flecha derecha	Gira la cámara en el eje horizontal hacia la derecha.
Flecha abajo	Gira la cámara en el eje vertical hacia abajo.

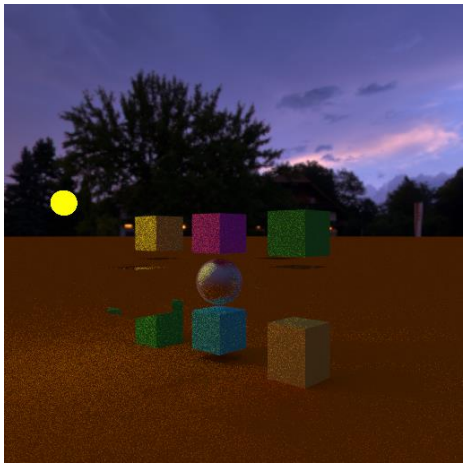
Escenas implementadas

Se han implementado en un total de siete escenas, donde demostramos cada uno de los objetos y materiales implementados.

Dentro de cada escena disponemos de una luz que se puede mover, cambiar de color y variar su intensidad (en las escenas de Corner Box existe además un cubo emisivo extra).

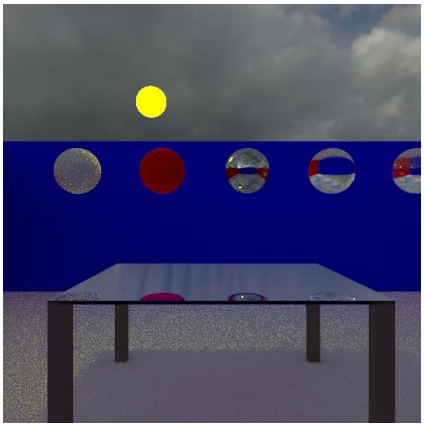
Escena 1: Cubes, dielectric and one metal

La primera escena muestra un conjunto de cubos difusos y dieléctricos sobre un plano anaranjado. Justo en el centro se encuentra una esfera metálica, la cual recibe información de todo el entorno.



Escena 2: Balls crystal table

Esta escena muestra una mesa con un material dieléctrico y un conjunto de esferas de distintos tipos. Según el mapa de entorno utilizado, podemos ver refractado o reflejado el entorno en las distintas esferas y la mesa.
En las paredes podemos ver una textura de tipo ajedrez blanco y negro.

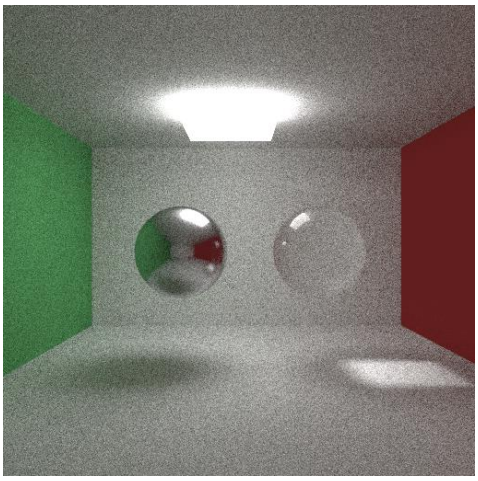


Escena 3: Balls solid table with wood floor

Esta tercera escena es una extensión de la escena segunda. Los dos únicos cambios son el suelo (textura con normal mapping) y una mesa con material atenuado.

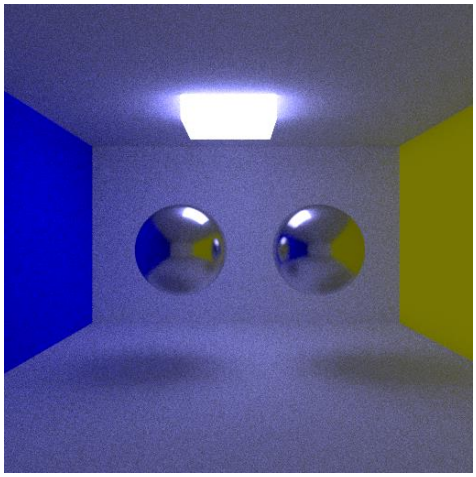
Escena 4: Corner Box Red-Green

En esta escena tenemos la primera implementación del famoso Corner Box. Esta versión utiliza los colores verde y rojo y en su interior encontramos un cubo emisivo (junto a la luz), una esfera metálica con glossy y una esfera dieléctrica.



Escena 5: Corner Box Blue-Yellow

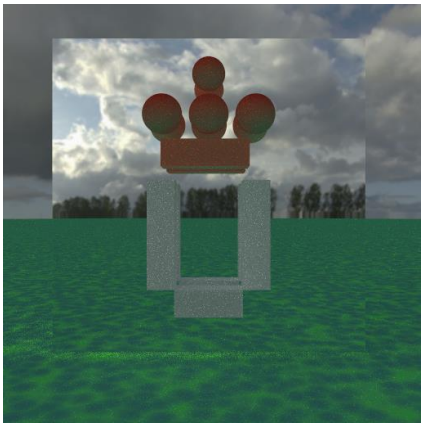
Parecida a la anterior, los colores utilizados han sido azul y amarillo. Las esferas mostradas son de tipo metálico con glossy, pero cada una de ellas utiliza un sampleado diferente (Dot-weight y uniform, respectivamente).



Escena 6: URJC logo

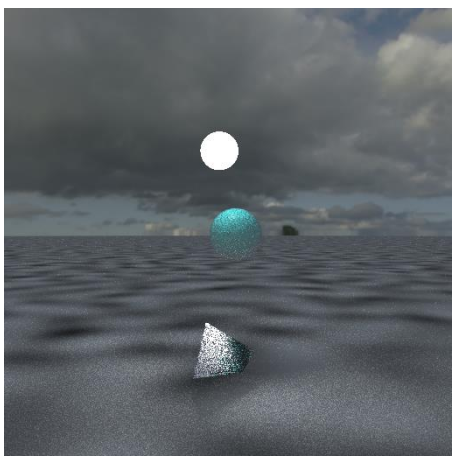
En esta escena podemos ver el logo de la Universidad Rey Juan Carlos conformado por cubos y esferas de material difuso. El suelo está formado por una textura de tipo Worley. Detrás se encuentra un material eléctrico.

Lo más curioso de esta escena es que, poniéndonos de perfil al dieléctrico (espejo) y activamos el motion blur, vemos cómo se emborrona este espejo.



Escena 7: Mesh Object

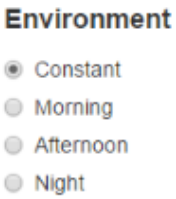
Escena simple que muestra un único objeto sobre un plano de tipo Perlin. Debido a la gran cantidad de código utilizado para generar la malla, esta escena no debería contener muchos más objetos para no superar el tamaño máximo de los shaders GLSL.



Environments

A la hora de mostrar los mapas de entornos, hemos incluido un total de tres mapas de entorno (día, tarde y noche) en HDR más un entorno constante. Todos los mapas de entorno HDR se procesan, como se comentó al inicio de la memoria, a través de las funciones implementadas en HDRLoaders. Como tarea adicional se incluyó, antes de finalizar, la rotación del mapa de entorno, cuyo uso se explicará en las líneas posteriores.

Para poder intercambiar entre uno y otro tenemos las opciones:

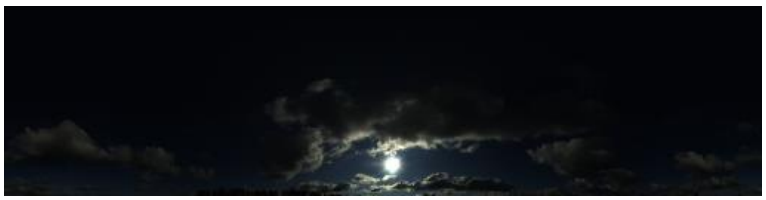


Constante

Fondo constante de color grisáceo.

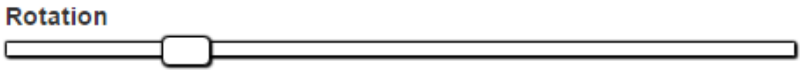
Mapas de entorno HDR

Hemos añadido un total de tres mapas de entorno distintos que pueden intercambiarse mediante la interfaz.



Rotación

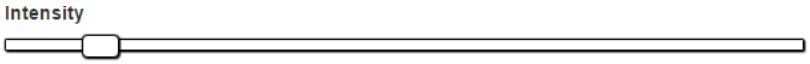
Una variación en la rotación varía, en algunas escenas, iluminación total de la escena. En escenas con objetos de tipo dieléctrico o metálico permite reflejar o refractar el mapa de entorno. El slider permite rotar el mapa de entorno en un rango comprendido entre [0, 720].



Energía emitida

Como se comentó en el primer bloque de la memoria (Ray Tracer), en este punto se detallaría cómo controlar la energía de los mapas de entorno HDR.

Para ello, al igual que la rotación, se ha incluido un slider que permite variar la iluminación del entorno entre [0, 5]. Cada mapa de entorno tiene una energía por defecto, por lo que será necesario modificar el valor del slider al gusto.



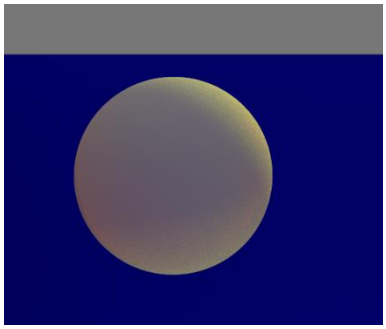
Materiales

Se han implementado varios tipos de materiales que permiten darle un mayor realismo a los objetos renderizados en la escena.

Diffuse

El tipo de material más sencillo es el difuso. Estos materiales pueden absorber la luz o reflejarla en direcciones aleatorias. Para implementar estos materiales en nuestro Path Tracer tenemos que evaluar dónde ha golpeado el rayo en el objeto con el material, calcular la normal del objeto en ese punto y redirigir el rayo hacia un punto aleatorio de la BDRF centrada en dicha normal. En el capítulo de **BDRFs** se explica brevemente que es una BDRF, que BDRFs hemos utilizado en nuestro código y como muestreamos los puntos en dichas BDRFs.

En la siguiente imagen aparece una esfera renderizada utilizando un material de tipo *Diffuse*:

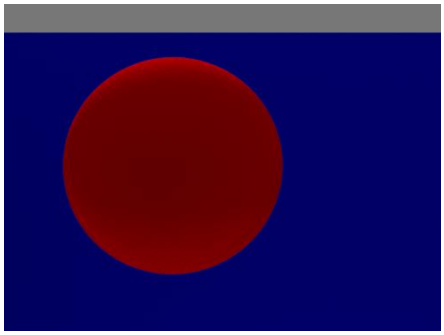


La esfera aparece en blanco ya que estamos reflejando todos los rayos que golpean el objeto (no estamos absorbiendo ninguno).

Attenuate

El siguiente tipo de material no es más que uno difuso pero con color. Lo único que hacemos es multiplicar un vector llamado "*attenuation*" (con el color que queramos definido en el espacio RGB) por la nueva dirección del rayo. Esta multiplicación hará que en vez de ver la esfera de color blanco la veamos del color que hemos definido en la variable "attenuation".

En la siguiente imagen vemos una esfera renderizada con un material de tipo *Attenuate* utilizando un color rojo:

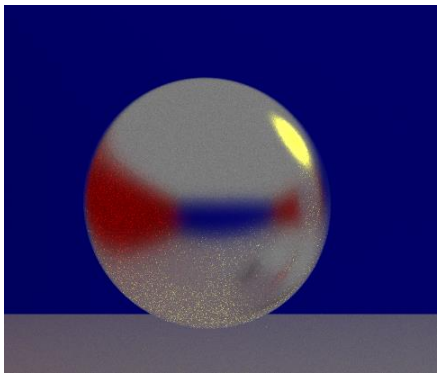


PhongSpecular

Un material difuso refleja la luz en direcciones aleatorias. Hay ciertos materiales (como los metales) que reflejan algunos rayos en direcciones más concretas (de ahí que nos veamos parcialmente reflejados en ellos o veamos ciertos brillos).

La ecuación de iluminación de Phong define una componente especular que es mayor si el vector reflejado por una superficie opaca forma un ángulo cercano a 0 con el eje de visualización de la cámara (en la realidad veríamos un brillo intenso en el objeto porque el rayo reflejado iría directo a nuestros ojos).

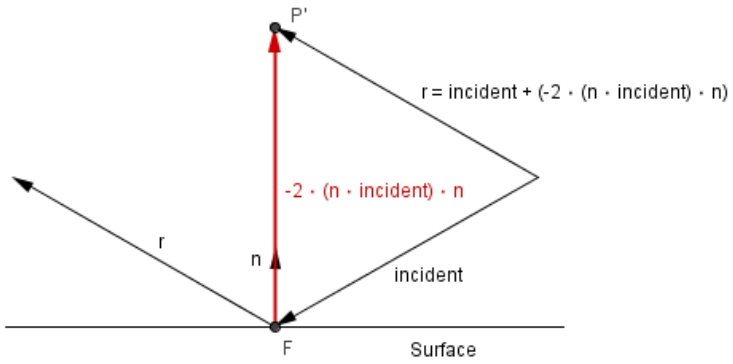
En la siguiente imagen vemos una esfera renderizada con el modelo de reflexión de Phong (n = 100):



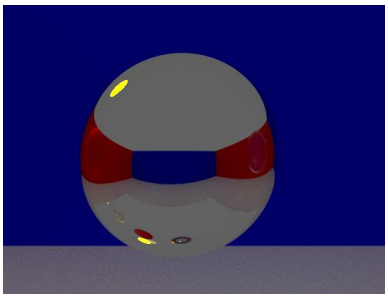
Metal

Los metales son materiales parcialmente reflectantes. Para su implementación en nuestro Path Tracer tenemos que lanzar un rayo al objeto con el metal, calcular la normal al punto donde impacte el rayo y calcular hacia dónde se va a dirigir el rayo tras rebotar con el objeto. En un mundo ideal donde el material tuviese una superficie totalmente pulida la respuesta es: el rayo reflejado **r** cuya dirección viene dada por esta fórmula:

$$r = v - 2 * dot(v, n) * n$$

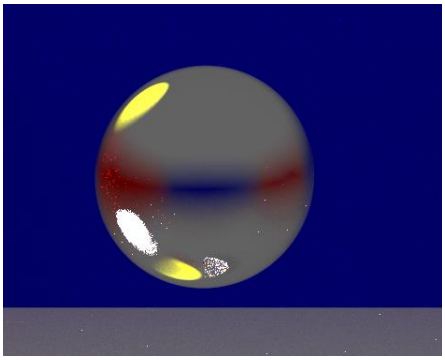


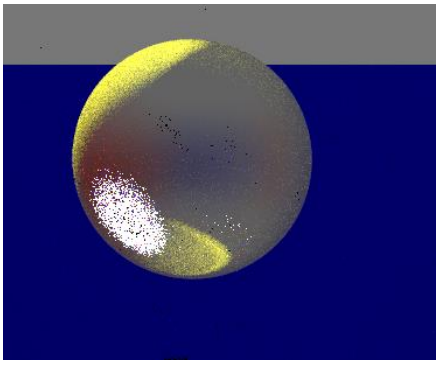
Siendo v el rayo que impacta en la superficie del objeto y n la normal a la superficie en el punto de impacto del rayo. Si dejásemos el rayo rebotado calculado de esta manera, tendríamos un material que refleja su entorno de forma perfecta:



En el mundo real, sin embargo, la superficie del metal no estará perfectamente pulida y tendrá todo tipo de imperfecciones, esto provoca desviaciones en los rayos reflejados y hará que no veamos una reflexión perfecta en la esfera sino algo a medio camino entre una reflexión difusa (color plano) y dicha reflexión ideal. Para conseguir este efecto "*glossy*" en el metal desviaremos los rayos reflejados sumando a la dirección del rayo reflejado perfecto, r, una dirección aleatoria de la BDRF multiplicada por una constante de glossiness que tendrá valores entre 0 y 1. Si glossiness vale 0 tendremos un material perfectamente pulido y con reflexiones ideales y si glossiness vale 1 tendremos un material puramente difuso (que dispersa los rayos de manera totalmente aleatoria).

A continuación, vemos dos esferas metálicas renderizadas con valores de glossiness de 0.3 y 0.7, respectivamente:





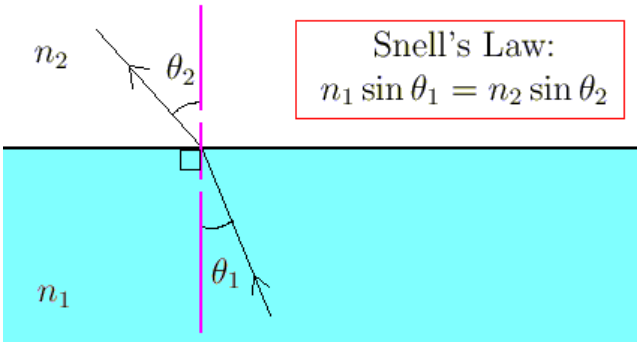
Como se puede observar, la primera esfera reflejará mejor los detalles de su entorno que la segunda esfera.

Transmit

Algunos materiales como el agua o el cristal son dieléctricos. Parte de los rayos de luz que golpean en su superficie salen reflejados y otros atraviesan el objeto con una dirección distinta a la que tenían al entrar (siempre y cuando el material exterior e interior al objeto sean distintos). Este último fenómeno se conoce como refracción. Nuestro material de tipo *Transmit* aplicado a un objeto lo convertirá en puramente refractante, esto es, todos los rayos que golpeen al objeto lo atravesarán y desviarán su dirección.

Dado un rayo que golpea en el objeto podemos calcular la nueva dirección del rayo una vez que atraviesa el objeto con la ley de Snell.

$$n1 * \sin\theta = n2 * \sin\theta'$$



Siendo $n1$ el coeficiente de refracción del primer material y $n2$ el del segundo. θ es el ángulo que forman el rayo entrante y la normal saliente de la superficie y θ' el ángulo entre el rayo desviado y la normal interna a la superficie.

Si el coeficiente de refracción del primer material es mayor que el del segundo se producirá un efecto de reflexión total (si estamos debajo del agua, que tiene un coeficiente de refracción mayor que el del aire y miramos hacia arriba el agua actuará como un espejo perfecto). Si por el contrario, el primer coeficiente es menor que el segundo, podremos calcular la dirección del vector refractado **r** con la siguiente fórmula:

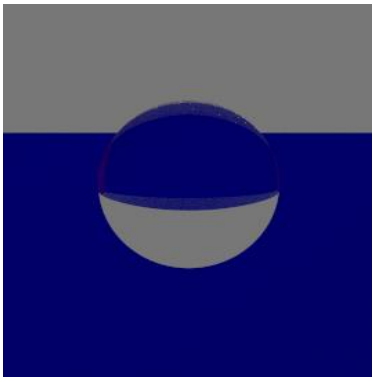
$$discriminante = 1 - \left(\frac{n1}{n2}\right)^2 * (1 - (\cos\theta)^2)$$

$$r = \frac{n1}{n2} * (v - n * \cos\theta) - n * \sqrt{discriminante}$$

Si el discriminante es negativo, tenemos el primer caso, en el que el coeficiente de refracción del primer material es mayor que el del segundo y por tanto tenemos reflexión. Si ese discriminante es positivo, existe refracción y se puede calcular la dirección exacta del vector refractado.

En el caso de *Transmit* solo consideraremos el segundo caso. En el material SmoothDielectric jugaremos con dicho discriminante y además añadiremos el efecto de Fresnel.

En la siguiente imagen vemos una esfera con el material *Transmit*:



Como podemos observar la esfera es totalmente transparente y vemos lo que hay detrás de ella pero dado la vuelta (los rayos de visión lanzados desde la cámara se ven desviados al colisionar con el material refractante).

SmoothDielectric

Los materiales dieléctricos además de dejar pasar la luz a través de ellos, la reflejan con cierta probabilidad. Esta probabilidad de reflectancia va en función del ángulo con el que se mire el objeto. Por ejemplo, si miramos una ventana colocándonos en frente de ella, veremos solo lo que hay en el exterior. Si pegamos nuestra cabeza a la ventana y levantamos la mano, veremos que la ventana reflejará nuestra mano como si de un espejo se tratase (además de poder seguir viendo el exterior). Esto se llama efecto de Fresnel.

Una forma fácil de calcular la probabilidad de reflectancia es la que usamos en el método "*fresnelDielectric*", la cual es una simplificación de la aproximación de Schlick:

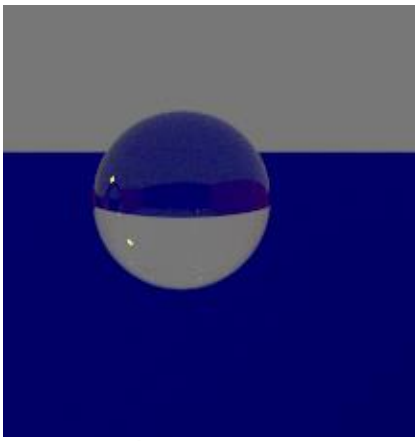
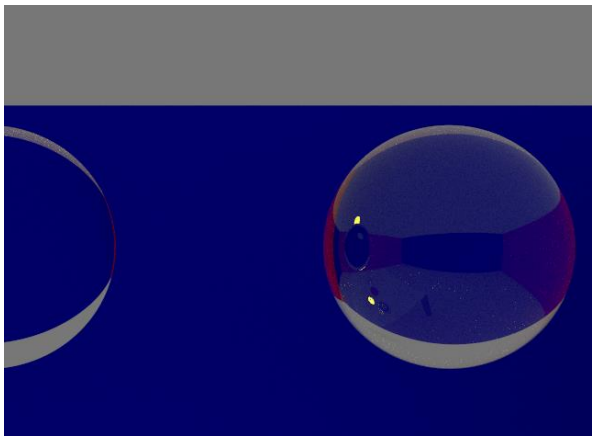
$$probabilidad = r_0 + (1 - r_0) * (1 - \cos\theta)^5$$

$$r_0 = (\frac{n1 - n2}{n1 + n2})^2$$

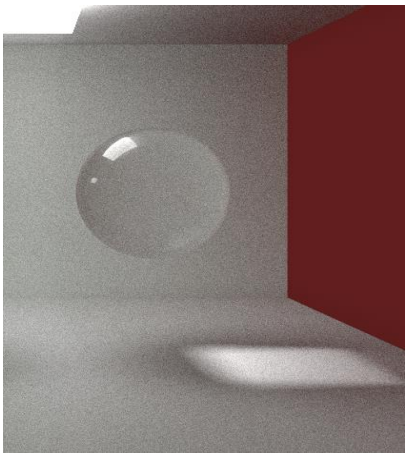
El código para implementar el material *SmoothDielectric* queda de la siguiente manera: primero se comprueba si el discriminante de la fórmula de refracción es negativo (lo hacemos con el método "*totalInternalReflectionCutoff*". Si es así, calculamos la nueva dirección del rayo con la fórmula de reflexión que hemos visto en el capítulo de metales. Si el discriminante es positivo pueden ocurrir dos cosas en función del efecto de *Fresnel*. Si la probabilidad de reflectancia calculada con nuestro método es de 0.8 (en nuestro ejemplo de la ventana, la estaríamos mirando con la cabeza casi pegada a esta), por ejemplo, generaremos un número aleatorio de 0 a 1. Si este número cae por debajo de 0.8 significa que el rayo saldrá reflejado de la superficie, mientras que si cae por encima (con una probabilidad de 0.2 sobre 1) el rayo atravesará la superficie (utilizaremos la fórmula para calcular el vector refractado).

Con la nueva dirección del rayo calculada de esta manera, veremos una esfera con este material igual que como con el material *Transmit* con la diferencia de que cuando nos acerquemos a la esfera por un lado y por tanto varía la probabilidad de reflectancia veremos los demás objetos reflejados en esta esfera, por el nombrado efecto de *Fresnel*.

A continuación se muestra una esfera con el material *SmoothDielectric* (en la segunda se nota más el efecto de Fresnel):

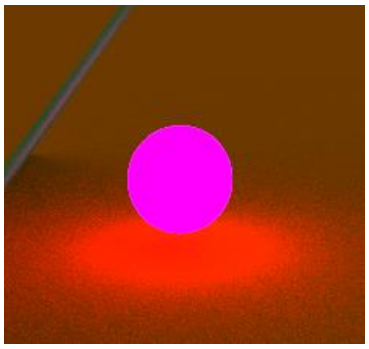



Los materiales dieléctricos concentran una gran cantidad de rayos de luz que pasan a través de ellos en determinadas zonas. Esto se traduce en la aparición de *caústicas* (pequeñas acumulaciones de energía) en las sombras de los materiales dieléctricos.



Emisivo

Un material de tipo emisivo es capaz de emitir (valga la redundancia), de cambiar la acumulación de la escena teniendo en cuenta la intensidad y color definido a la hora de crear dicho material. Requiere de un segundo material de tipo absorbente.



Absorbente

Debido a que todos los materiales necesitan implementar qué ocurre si el rayo ha interseccionado, la única funcionalidad de este tipo de material es capturar el color y parar los rebotes. Es por ello por lo que este material no tiene representación gráfica, solo consiste en capturar toda la energía del rayo y finalizarlo.

Reflectivo

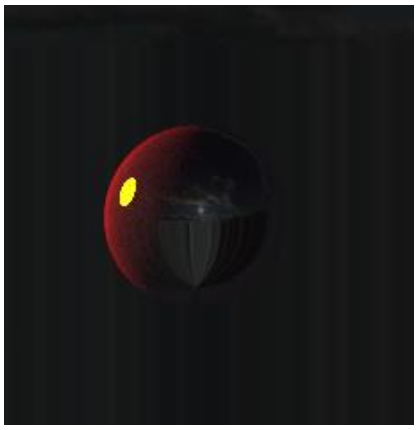
Este tipo de material cambia la dirección del rayo en dirección a la normal del objeto donde se ha realizado el hit más cercano a la cámara.

La posición del rayo se obtiene a través del hit y el sumatorio de cierto valor de error epsilon por la dirección del rayo reflectado.

FresnelComposite

Un material de tipo Fresnel compuesto está formado por la unión de dos materiales: Uno reflectante y otro transmisivo.

Su funcionamiento interno está basado en el material SmoothDielectric utilizando la función "*totalInternalReflectionCutoff*". Primero se comprueba si el discriminante del fresnel de refracción (*na* y *nb* dados, ambos índices de refracción) es negativo. Si es así, el radio es 1.0, si no, es la mitad del *fresnelDielectric*. Según el valor del radio, se utiliza uno de los dos materiales (reflectante o transmisivo).



Material FresnelComposite compuesto por un material atenuado (rojo) y uno reflectante. Los parámetros na y nb son el índice de refracción de la escena y 1.85, respectivamente

Gracias a este tipo de material, podemos producir escenas interesantes, con objetos que mezclan su componente dieléctrica y metálica, entre otros.

Texturado

Este tipo de material muestra un efecto de Normal Mapping y luz especular a través de tres texturas: difuso, especular y normal.

FormulaTexture

Este tipo de material permite asignar una textura procedural a través de una ecuación o conjunto de instrucciones.

ChessTexture

Se trata de una extensión de *FormulaTexture*. La textura final generada es una superficie texelada a través de cubos con dos colores posibles. El ejemplo más realista de este tipo de textura es el tablero de ajedrez o de las damas.

Debido a algún problema con el algoritmo, en algunas escenas o con algunas rotaciones y posiciones de cámara se producen artefactos extraños.

Se pueden utilizar dos colores para las teselas. Por defecto es blanco y negro.

FormulaPowTexture
Esta textura es parecida a *ChessTexture*, pero variando la fórmula utilizada. Según el objeto que contiene este tipo de material, se mostrarán franjas horizontales o verticales, haciendo un efecto parecido al código de barras.

WorleyTexture
Este material reproduce una textura de tipo Worley. Permite utilizar dos colores: fondo y borde.

PerlinTexture
Este material reproduce una textura de tipo Perlin.

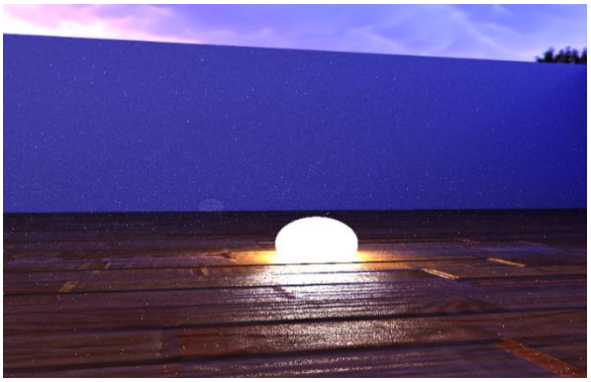
Normal Mapping
Existen diversas formas de simular un relieve 3D sin generar nueva geometría. Entre las técnicas más habituales se encuentra Bump Mapping y Parallax Mapping, las cuáles varían las normales con el fin de simular un efecto de profundidad, generalmente, sobre un plano.

Debido a la envergadura del proyecto, se decidió utilizar una solución intermedia basada en Normal Mapping, ya que nos era mucho más sencillo de implementar con las técnicas de Ray Tracing.

La técnica del Normal Mapping se basa en cambiar la normal según los datos obtenidos de una textura de normales, después de realizar una conversión de [0, 1] a [-1, 1]. Esta variación de las normales permite que se produzcan algunas zonas más oscuras, lo cual engaña a la vista percibiendo una superficie “3D”.

Como todas las técnicas basadas en este estilo, no se genera geometría, por lo que se ahorra mucho tiempo de cómputo, así como espacio en memoria de vídeo. No obstante, estas técnicas también tienen sus problemas, como por ejemplo la generación de mapas de normales o que, enfocando en perfil, no se muestra el nuevo relieve (esto también afecta a las sombras).

Para finalizar, en la siguiente fotografía podemos comprobar como el cambio de las normales simula con bastante buen resultado las sombras o vértices “levantados”.



Depth of Field
A la hora de realizar la profundidad de campo, es necesario tener en cuenta también el valor de la distancia focal.

Ambos conceptos están muy ligados en el mundo de la óptica y la fotografía. A mayor distancia focal, se tiene menor profundidad de campo.

En cuanto a su uso en fotografía, es importante tener en cuenta los dos posibles rangos de valores de la profundidad:

- Profundidad de campo grande: Usado para que toda la escena aparezca enfocada, útil para fotografía de paisajes, montañas,…
- Profundidad de campo pequeña: Usado para captar la atención del espectador y centrarse en un punto concreto. Por ejemplo, enfocar una persona o un cuadro.

A nivel de código, este tipo de desenfoque varía la dirección del rayo (teniendo en cuenta la profundidad de campo y la distancia focal) y la posición del rayo (solo profundidad de campo).

Utilizando los apuntes de Traza de rayos de la asignatura, hemos utilizado un pequeño *jitter* para evitar la creación de bandas visibles. Gracias al *jitter* con un muestreo aleatorio evitamos estos artefactos.

Ejemplo de uso de desenfoque y profundidad de campo

Armónicos esféricos

Diseñado por Ravi Ramamoorthi y Pat Hanrahan en 2001, los esféricos armónicos (SH) son una de las técnicas para renderizado en tiempo real basado en iluminación global que menos gasto computacional produce. En lugar de calcular la contribución de la luz evaluando **Bidirectional Reflectance Distribution Function** (BRDF), estos métodos usan unas imágenes especiales basadas en HDR/RGBE que guardan la información de la luz. La versión implementada está basada en Light Probes.

Un Light Probe es una imagen omnidireccional, de alto rango dinámico (HDR), que registra las condiciones de iluminación incidente en un punto particular del espacio.

Típicamente está diseñado para utilizarse en shaders de vértices con interpolaciones de la luz en la etapa de rasterizado (debido a que hay una variación muy pequeña de la luz, es recomendable que sea el rasterizado quien varíe con interpolaciones el color de los fragmentos) pero, debido a que en este caso nuestra geometría se "genera" en los shaders de fragmentos, en este proyecto se ha traspasado todo a este último shader gracias a la información recogida del shader de *ShaderToy* Spherical Harmonics Lighting del usuario jimmikaelkael.

Para conseguir los coeficientes de radiancia, hemos utilizado distintas imágenes HDR cuadradas. Las imágenes HDR se convierten a imágenes en coma flotante utilizando el software Radiance, en específico, su programa *pvalue* (este software se puede descargar e instalar tanto en Windows como en sistemas operativos Unix) y luego realizan un segundo filtro con el fichero prefilter.c.

Los comandos de uso específicos para cada uno de los dos programas son:

- $pvalue\ -df\ -H\ -h\ imagen\_hdr.hdr\ >\ imagen\_float.float$
- $prefilter.exe\ imagen\_float.float\ [width]$ (dónde width puede valer como máximo 2000. Width hace referencia al ancho o diámetro del light probe)

Debido a que los resultados obtenidos por *prefilter.c* proveen de demasiada iluminación, es necesario reducir el rango producido por un pequeño factor. En este caso se ha utilizado un factor de 0.315. Todos estos datos forman la ecuación final de la iluminación difusa:

$$Diffuse = c_1 \cdot L_{22} \cdot (x^2 - y^2) + c_3 \cdot L_{20} \cdot z^2 + c_4 \cdot L_{20} - c_5 \cdot L_{20} + 2 \cdot c_1 \cdot (L_{22} \cdot xy + L_{21} \cdot xz + L_{21} \cdot yz) + 2 \cdot c_2 \cdot (L_{11} \cdot x + L_{11} \cdot y + L_{10} \cdot z^3)$$

El código modificado de *prefilter.c* se encuentra en *others/prefilter.cpp*, dentro de la raíz del proyecto entregado.

Se puede obtener más información acerca de iluminación basada en esféricos armónicos en el libro naranja de *OpenGL Shading Language*.

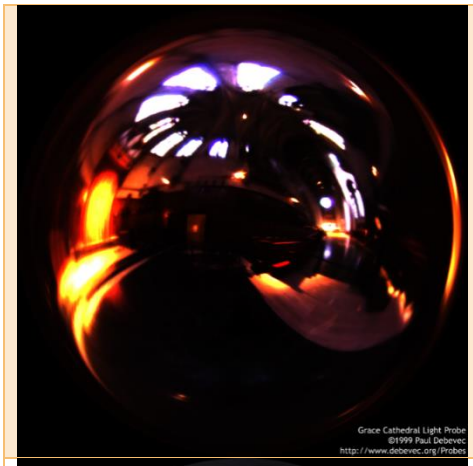
Créditos a la Universidad de Stanford por el código del *prefilter*, así como a *jimmikaelkael* por la aproximación a esféricos armónicos utilizando el shader de fragmentos.

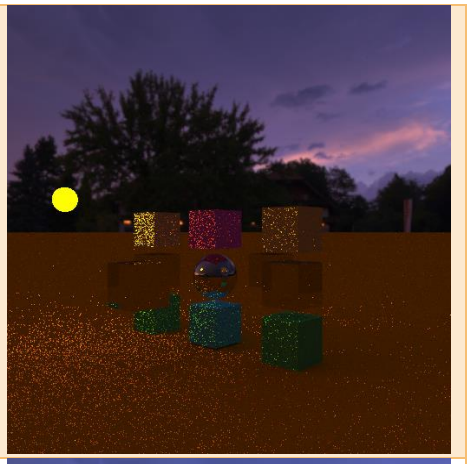
Dentro de nuestra aplicación hemos incorporado un total de cuatro probes, los cuáles se pueden obtener en http://www.pauldebevec.com/Probes/ y http://www.unparent.com/photos_probes.html.

Nombre	**HDR**	**Demo**

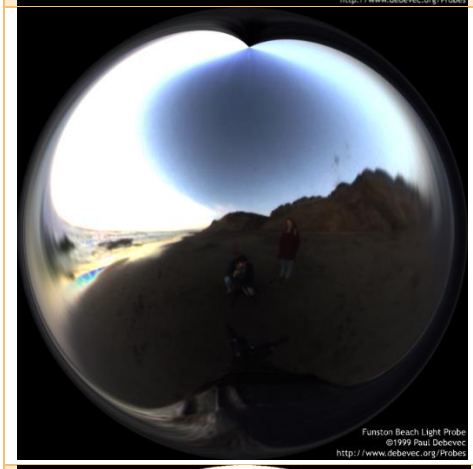
[3] Ecuación obtenida del libro Orange Book – OpenGL Shading Language 2nd Edition.

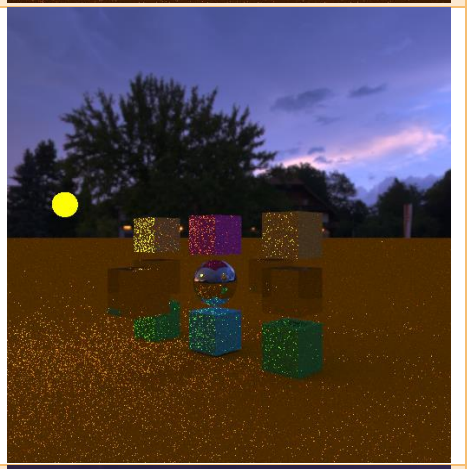
Grace



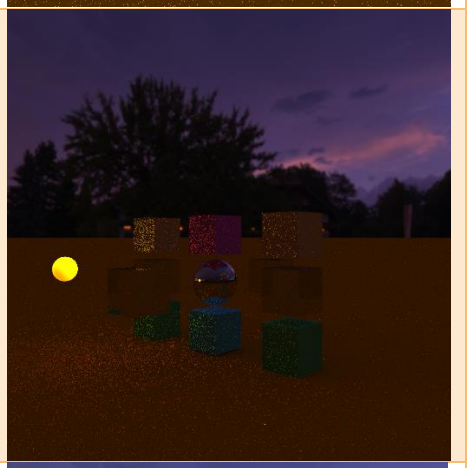


Beach

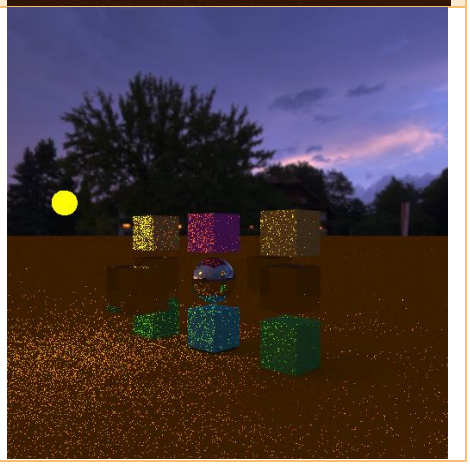




Galileo



Neighbor



Antialiasing y filtrado de texturas

El arreglo del aliasing y de filtrado de texturas en nuestras escenas se ha hecho a través del propio promedio de mezclados entre escenas anterior y actual. De esta forma, los bordes de sierra desaparecen sin necesidad de realizar múltiples pasados con muchos rayos.

Como podemos observar en la anterior imagen, al acercarnos tanto a una esfera y a una caja, no se produce ningún borde de sierra.
Nota: En los objetos de tipo Emisivo-absorbentes, debido a que los rayos mueren en ese momento, se producen pequeños bordes de sierra.

Motion Blur

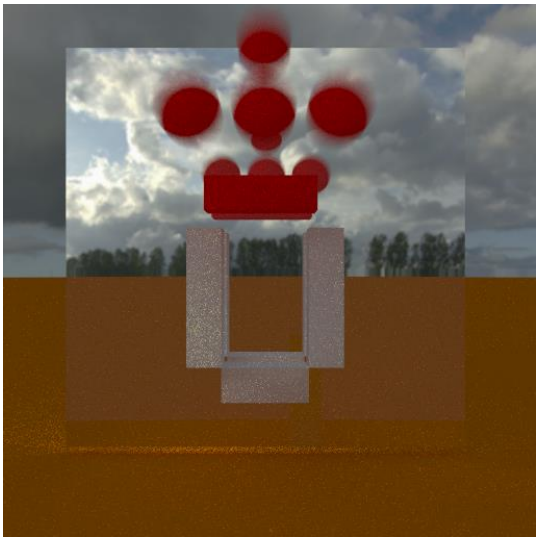
Existen diversas maneras de enfocar el Motion Blur. La más habitual en gráficos bajo OpenGL es realizar un postproceso, pero esta técnica no nos funciona en caso de realizar iluminación global con técnicas de MonteCarlo.

Descartando esta opción, hemos encontrado dos posibles soluciones:

1. Mover la esfera en cada pasada de renderizado: Requiere uso de variables uniformes por cada objeto a mover. El ruido producido no es el más adecuado.
2. Utilizar una variable de tiempo variable entre el tiempo actual y el tiempo futuro. A la hora de realizar la etapa de intersecciones rayo-objetos, multiplicamos la posición actual del objeto con ese tiempo con una ecuación parecida a la siguiente:

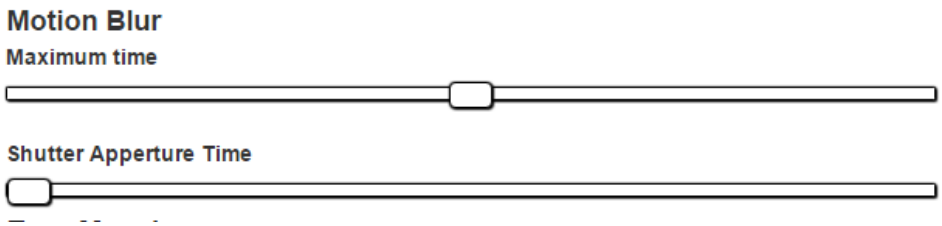
$$position\_center + vec3(-(times.x + (times.y - times.x)) * random\_number(\dots);$$

Finalmente nos decantamos con la segunda opción al ver que era más viable y escalable en el tiempo.
Esta funcionalidad solo se encuentra en objetos de tipo esfera, ya que era la forma más sencilla de demostrar su funcionamiento.



Demo con Motion Blur aplicado sobre cuatro esferas.

En cuanto a la interfaz, disponemos de dos sliders que permiten variar la posición de la esfera y la cantidad de apertura del Motion Blur.



Apéndices

Cosas a tener en cuenta antes de empezar

Si bien existen diversas librerías que facilitan la programación en WebGL para personas novatas como Babylon.js o Three.js, hemos decidido crear el código desde 0, con el fin de controlar todos los pasos de la ejecución. Creemos que este tipo de librerías te abstraen a veces demasiado.

Es recomendable entender la diferencia entre doble buffer y la técnica Ping-Pong, pues esta última es la utilizada para realizar en renderizado de las distintas escenas.

Doble Buffer

Como es conocido, OpenGL utiliza dos buffers (front y back), siendo éste primero el pintado en pantalla. Esto facilita la velocidad de renderizado, de forma que mientras en el back se están recibiendo los datos de la tarjeta de video, en el front tenemos el anterior frame. Gracias a la función de *swapBuffers*, podemos intercambiar estos buffers de forma sencilla.

En WebGL esto no es posible, ya que es el propio navegador quien se encarga de gestionar esta tarea.

Debido a que para utilizar varias pasadas se necesitaba utilizar la textura del frame anterior, utilizamos una técnica muy habitual en gráficos llamada Ping-Pong textures.

Ping-Pong technique

Se trata de una técnica donde se usa la salida de un renderizado para la entrada en otro render. También es posible utilizar esta técnica para simular un doble buffer, funcionalidad no existente en WebGL, aunque no es necesaria, ya que es el propio navegador quien realiza esta tarea. Es necesario disponer de un Framebuffer.

En esta aplicación, se ha utilizado esta técnica para mostrar la evolución de las iteraciones del Path Tracer (también favorece el antialiasing).

Se puede obtener más información en los siguientes links:

http://www.pixelnerve.com/v/2010/07/20/pingpong-technique/

https://hub.jmonkeyengine.org/t/framebuffers-and-the-ping-pong-technique/28459

Estructura del proyecto

Camera

Esta clase gestiona una cámara en perspectiva con un field of view de 45º.

Environments

Esta clase contiene los dos tipos de entornos disponibles: entorno simple (*SimpleEnvironment*) y entorno texturado (*TextureEnvironment*).

Globals

El fichero *globals* incluye gran parte de las variables compartidas por toda la aplicación.

HDRLoaders

Fichero utilizado para cargar las texturas en formato HDR.

Index
Este fichero gestiona todos los eventos de teclado, interfaz, así como de realizar las peticiones de renderizado a los integradores.

Input
Clase singleton que se encarga de simplificar los eventos de teclado. Actualmente no se utiliza.

Integrators
Directorio que incluye los distintos integradores de la aplicación: Ray Tracer, Path Tracer y Spherical Harmonics.

Materials
Este módulo incorpora todos los materiales creados para la aplicación.

Renderable
Directorio que incluye los seis tipos de objetos implementados en la aplicación.

QuadsToneMap
Este fichero incluye el módulo *toneMap*, el cual contiene los diferentes tipos de mapas de tonales implementados en la aplicación.

Scene
Este módulo contiene dos clases: *PerspectiveRays* se utiliza para generar la posición del rayo inicialmente teniendo en cuenta el Depth of field y el focal length; y *Ray*, cuya finalidad era realizar Ray Casting en el cliente, pero se ha quedad como un future.

SceneLoader
Clase utilizada para cargar escenas a partir de un XML. Las escenas están compuestas por una cámara, una fuente de luz y un conjunto de objetos.

ShaderProgram
Clase utilizada para compilar los distintos shaders de la aplicación. Además de compilar y linkar los shaders, provee de funcionalidad para cachear los identificadores de variables uniformes y atributos.

SourceFrag
Esta clase es el núcleo principal con el que se generan los shaders finales. Incluye un conjunto de pequeños fragmentos de código los cuáles son requeridos por las distintas clases de la aplicación. Un *SourceFrag* es un código en GLSL el cuál puede tener (o no) dependencias de otros.

Utility
Módulo de utilidades que se encarga, principalmente, de generar el shader final a partir de los datos de los objetos y materiales. Es la parte que más utiliza la clase *SourceFrag*. También provee de funcionalidad para cargar mallas y texturas.

Compilación y desarrollo
A la hora de trabajar hemos utilizado dos herramientas principales: Typescript y Grunt. El primero es un lenguaje de programación basado en Javascript, mientras que el segundo es un compilador de código Typescript.

A continuación se detallarán un poco más en detalle cada una de las herramientas.

Typescript

Typescript se trata de un lenguaje de programación libre desarrollado por Microsoft. Es un superconjunto de JavaScript que permite que añade tipado estático y orientación a objetos con una sintaxis muy similar a C# y Java.

El código final generado por Typescript se trata de código JavaScript nativo, pero mucho más optimizado que un código JavaScript desarrollo de forma nativa por un desarrollador.

Compilación de código Typescript

A pesar de que Typescript permite generar código JavaScript, es necesario tener un intermediario que se encargue de realizar esta generación y transformación de código para que un navegador pueda interpretarlo. Es por ello por lo que se ha visto necesario integrar un sistema para realizar tareas automáticas.

En el mundo web existen diversas librerías y herramientas para realizar estas tareas utilizando el entorno Node.js, siendo las más utilizadas *Gulp* y *Grunt*. Debido a que uno de los miembros de las prácticas ya tenía tareas automatizadas para otros proyectos externos usando *Grunt*, se decidió utilizar *Grunt* como control de tareas.

Grunt

Grunt.js es una librería JavaScript que nos permite configurar tareas automáticas con el fin de ahorrarnos tiempo de desarrollo y despliegue. Para instalarlo y ejecutarlo se necesita Node.js instalado en el ordenador donde se quiera ejecutar.

La configuración de las tareas disponibles se implementa a través del fichero *Gruntfile*. Podemos llamar a las tareas llamando a "*grunt*" dentro de una tarea.

Gruntfile

Para el desarrollo de esta práctica se han utilizado varias librerías Node.js auxiliar que son tareas automatizadas desarrolladas por terceros para facilitar el trabajo. Las tareas automáticas definidas han sido:

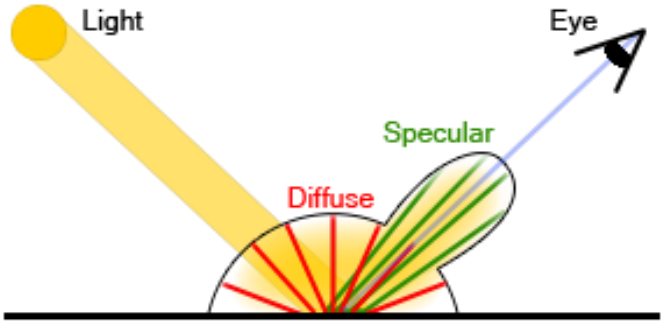
- **ts:** Tarea automatizada que permite compilar el código Typescript, generando código JavaScript junto a ficheros *.map* para facilitar la depuración en el navegador directamente utilizando el código Typescript.
- **connect:** Con el fin de facilitar el desarrollo, se provee de una tarea que se encarga de abrir un servidor y comunicar al navegador por defecto que abra el fichero index.html que contiene la aplicación WebGL. Además utiliza la librería *livereload* que permite recargar la página si se requiere dentro de una tarea.
- **watch:** Uno de los usos más habituales de este tipo de entornos es tener varios directorios en los que Grunt "escuche" si se han realizado cambios. En este caso la tarea que realiza es tener un escuchador de ficheros ts y lanzar la tarea de compilación "ts", además de actualizar el navegador con la tarea "livereload" comentada anteriormente.
- **concat:** Une todos los ficheros JS en uno solo para reducir la cantidad de carga de ficheros dentro de la aplicación web.
- **uglify:** Minifica y ofusca el código JS.
- **open:** Abre la página por defecto.
- **objjson:** Transforma objetos en *Wavefront OBJ* triangulados (con facetas vertex/normal/texCoord) en formato JSON compatible con navegadores web.

BRDF

Una BDRF (siglas en inglés de “Bidirectional Reflectance Distribution Function”) es una función que define como la luz es reflejada en una superficie opaca.

Cada material y cada zona de este pueden tener BDRFs muy distintas. Lo que se suele hacer en el mundo de los gráficos es utilizar aproximaciones conocidas para las formas complejas que pueden tener estas BDRFs (que están definidas en un espacio de cuatro dimensiones).

Para este código hemos usado una BDRF con forma esférica y otra con forma de hemiesfera.



Como se ha comentado antes, estas BDRFs definen como se refleja la luz en una superficie. Si esta BDRF está centrada en la normal al punto de impacto de un rayo en un objeto, se puede calcular el rayo rebotado con origen en el punto de impacto y dirección hacia un determinado punto de la BDRF calculado aleatoriamente.

La dificultad está en calcular un punto que este en esta superficie de la BDRF y que al calcular varios puntos de esta BDRF estén uniformemente distribuidos a lo largo de la superficie de este (de lo contrario, los rayos se concentrarían en una determinada zona y esto para implementar un material puramente difuso es un efecto poco deseado).

Esfera utilizando el método de rechazo

Una de las formas más sencillas de calcular un punto que esté en la superficie de una BDRF (tomándonos esta como una esfera sencilla) es tomar un punto aleatorio en un cubo circunscrito en dicha esfera y evaluar si este punto está dentro o no de la esfera. Si está dentro, nos lo quedamos y lanzamos el rayo rebotado en la superficie del objeto a través de ese punto. Si está fuera, lo intentamos de nuevo. Este método no genera una distribución aleatoria uniforme de puntos sobre la esfera (si tomamos un número alto de muestras) por lo que no lo hemos usado en nuestro código.

Esfera con muestreo uniforme

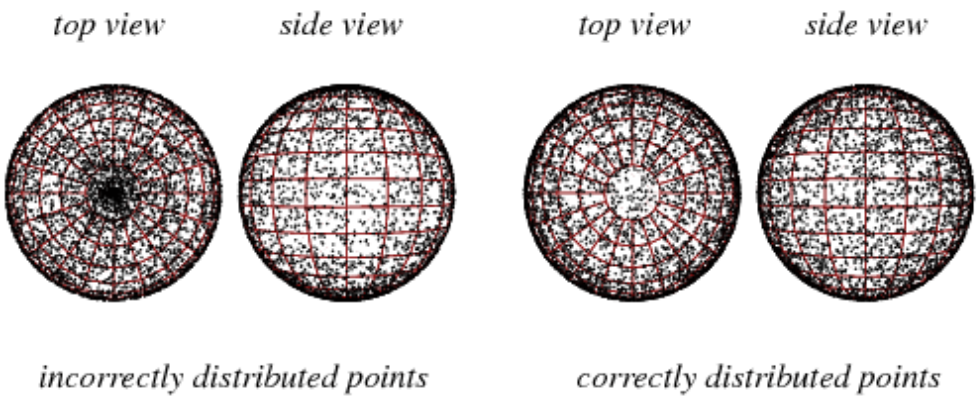
El anterior método acumula muestras en los polos de la esfera (debido a que el área de la esfera es una función de solo una de las dos coordenadas polares que definen cualquier punto de esta, φ y θ). La forma de evitar esto es utilizando los ángulos φ y θ definidos de esta forma:

$$\varphi = 2 * \pi * u$$

$$\theta = \cos^{-1}(2 * v - 1)$$

Siendo u y v dos números aleatorios definidos para el rango [0 1].

Utilizando estos dos ángulos podemos determinar una serie de puntos distribuidos uniformemente por la BDRF haciendo que los rayos se dispersen de forma correcta y tengamos materiales difusos de buena calidad (y por supuesto, el resto de materiales). El método que realiza este muestreo uniforme sobre una esfera se llama "*sampleUniformOnSphere*".



Hemiesfera utilizando un método de inversión como técnica de muestreo (Monte Carlo)

Una mejora sobre el muestreo uniforme de puntos sobre una esfera es utilizar una hemiesfera como BDRF y muestrear los puntos sobre ésta usando un método de inversión. En vez de buscar una densidad uniforme de puntos, buscamos una densidad con una función de probabilidad que conocemos:

$$p(\varphi,\theta) = \frac{n+1}{2*\pi} * (\cos\theta)^n$$

Esta función de densidad está basada en el modelo de iluminación de Phong (ponderado por el coseno). n es el exponente de la ecuación de Phong y θ y φ son las dos coordenadas polares (θ está definida en [0, $\pi/2$] y φ en [0, $2*\pi$]). Partiendo de esta función de densidad, calculamos su distribución acumulada:

$$P(\varphi,\theta) = \int_0^{\varphi}\int_0^{\theta} p(\varphi',\theta') * \cos\theta' * d\,\theta' * d\varphi'$$

Utilizando la inversa de esta distribución acumulada podemos hallar una serie de muestras que sigan una distribución como la definida por la función de densidad que hemos enunciado antes. Esas muestras (puntos en la BDRF) se extraen utilizando los ángulos θ y φ definidos en función de dos variables aleatorias (r1 y r2) y la n (el valor 1 es el más común).

$$\varphi = 2*\pi*r1$$

$$\theta = \cos^{-1}\left((1-r2)^{\left(\frac{1}{n+1}\right)}\right)$$

Como podemos ver en nuestro método "*sampleDotWeightHemisphere*" este recibe dos números aleatorios: xi1 y xi2. La variable "*angle*" es el ángulo φ extraído por el método de inversión mencionado anteriormente. Si tomamos, dada la BDRF, una base de tres vectores (normal centrada en la BDRF, tangente y bitangente) podemos convertir los dos ángulos que hemos visto anteriormente en un vector unitario (cuyo

extremo es precisamente el punto de la BDRF que buscamos) con la siguiente fórmula:

$$vector = \sin\theta\cos\varphi\, u + \sin\theta\sin\varphi\ v + \cos\theta\, w$$

Usando n=1 tenemos que:

$$\theta = \cos^{-1}\left(\sqrt{1-r2}\right)$$

Como necesitamos sin θ hacemos lo siguiente:

$$\cos\theta = \sqrt{1-r2};$$
$$(\cos\theta)^2 + (\sin\theta)^2 = 1;$$
$$(\sin\theta)^2 + 1 - r2 = 1;$$
$$\sin\theta = \sqrt{r2};$$

La variable llamada "*mag*" es precisamente este seno de θ y evita el cálculo computacional de fórmulas trigonométricas, que suele ser lento.
El método de inversión que hemos usado para hallar las muestras deseadas es un método de Montecarlo ya que sigue el siguiente patrón:

Define un dominio de posibles entradas. Puntos sobre la superficie de la BDRF.

Genera entradas de forma aleatoria siguiendo una distribución de probabilidad específica. En este caso tenemos una función de densidad ponderada por el coseno (modelo de iluminación de Phong) sobre una hemiesfera (dominio).

Realiza un cálculo determinista sobre las entradas. Hallamos la CDF (por sus siglas en inglés, "Cumulative Distribution Function") de la función de densidad, la invertimos y dado un número aleatorio Yi buscamos el Xi en la CDF tal que:

$$\text{Xi} = P^{-1}(\text{Yi})$$

Con esto, sabemos que con una sucesión de números aleatorios como Yi tendremos una distribución de eventos con una función de densidad equivalente a la función de la que partíamos.

Eso es justo lo que queremos, que dado un rayo que impacte sobre una superficie, el rayo rebotado lo haga en una dirección muy parecida a la que llevaría en la realidad (que sería la dirección de un vector hacia un punto muestreado sobre una BDRF siguiendo la lógica de la ecuación de iluminación de Phong).

Librerías auxiliares

glMatrix

Se trata de una librería muy optimizada para el cálculo vectorial, principalmente destinada para su uso en aplicaciones gráficas en torno a WebGL y simulaciones físicas.

Debido a que está basada en la librería GLM de C++, es muy fácil su adaptación al entorno JavaScript y WebGL.

Se puede acceder a su documentación a través de su página web:
http://glmatrix.net/

WebGL 2

¿Qué es?

WebGL 2 es una evolución de las especificaciones de WebGL, aún en fase de adaptación por la mayoría de los navegadores. Aunque no se encuentra de forma nativa instalada en los navegadores más habituales (Mozilla Firefox, Google Chrome), es posible activar algunas extensiones propias de estos navegadores para trabajar-

Entre sus principales mejoras, cabe destacar:

- Multiple Render Target: A diferencia de las especificaciones anteriores, *WebGL 2* permite pintar sobre varias texturas en una única llamada.
- Instancing: Permite realizar varias copias de renderizado en una sola llamada.
- *Uniform Buffer Objects*, *Query Object* (*Occlusion Query*), *Transform Feedback* y *Sync Objects*, entre otros.
- La API de GLSL permite utilizar la sintaxis de *OpenGL ES 3.0*, lo que permite incorporar el uso de variables in/out. Como pasa en las versiones modernas de *OpenGL*, en WebGL 2 podemos retornar varios datos en los shaders de fragmentos (esto facilita, por ejemplo, el desarrollo de shaders con FBO). Para activar la nueva sintaxis, es necesario añadir ***#version 300 es*** al inicio de cada shader.
- Incorporación de texturas *iSampler2D* e *uSampler2D*, así como texturas 3D.
- Texturas de flotantes de forma nativa, sin necesidad de extensiones.
- Mejor compatibilidad con texturas de tipo *NPOT* (Non Power of Two Textures).
- Shaders GLSL mucho más grandes, así como mayor cantidad de variables uniformes disponibles.

Debido a que es una especificación bastante reciente (en torno a finales de 2015 algunos navegadores ya implementan gran parte de los requisitos), se encuentra muy poca información en la web.

Casi toda la información recogida para WebGL 2 parte de los siguientes links:

- Especificaciones de Khronos.
- Ejemplos con WebGL 2
- Compatibilidad WebGL 2

Activar WebGL 2 en los navegadores actuales

Este código solo ha sido probado con buenos resultados bajo la última versión de Google Chrome (Versión 50.0.2661.87 m) con fecha 26/04/2016.

Debido a que WebGL 2 sigue en proceso experimental, Google Chrome no permite cargar escenas con este contexto por defecto. Para activar WebGL 2 hay que entrar en los flags de Chrome (en la url: $chrome://flags$) y habilitar la función “Prototipo WebGL 2.0). Una vez activado este flag, por si acaso, es mejor reiniciar el navegador.

Comparación WebGL y OpenGL

- WebGL tiene una pobre implementación de texturas no potencias de 2. Aunque si bien es posible utilizar texturas de tamaños distintos, el rendimiento de estas texturas es muy limitado. En la versión 2 se defiende que se ha mejorado esta funcionalidad.
- WebGL es una API totalmente asíncrona por lo que controlar, por ejemplo, los errores, no es tan fácil.
- En WebGL no existe la técnica del doble buffer. Por lo general se suele utilizar la técnica Ping-Pong que se comentará posteriormente.
- En los shaders de fragmentos es necesario marcar la precisión de los valores flotantes. Esto se hace añadiendo al inicio del shader "precision X float;" siendo X un valor entre *highp* y *mediump*.
- Al tratarse de una versión que parte de OpenGL ES, solo están disponibles shaders de vértices y fragmentos, hecho que imposibilita técnicas más complejas como teselación o generación de geometría al vuelo.
- WebGL no incorpora funciones matemáticas como las antiguas versiones de OpenGL (las nuevas tampoco tienen funciones matemáticas predefinidas). Es por ello necesario utilizar librerías de terceros, siendo la más utilizada gl-matrix.

Se puede obtener más información en:
https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences

Problemas sin solucionar y trabajos futuros

Debido a la falta de tiempo debido a la cantidad de trabajos del Máster, así como del proyecto del Human Brain Project en el que ambos estamos metidos, no hemos podido realizar todas las cosas que nos hubieran interesado. A continuación se mostrará una lista de posibles mejoras:

- Añadir algoritmo Ray Tracing: Con el fin de comparar entre Ray Tracing y Path Tracing, habría sido necesario implementar también dicho algoritmo, pero debido a la envergadura del proyecto decidimos centrarnos en el algoritmo que produce mejores resultados.
- Añadir *Web Workers* con *OffscreenCanvas* para disminuir la carga producida en el hilo principal. Esta funcionalidad solo es permitida con Mozilla Firefox, pero por motivos de tiempo no se pudo llegar a integrar (tampoco queríamos coartar a los usuarios que probasen el código a necesitar un tipo de navegador en específico).
- Más tipos de materiales: Existen multitud de tipos de materiales, pero solo hemos llegado a implementar los más habituales.
- Estereoscopia: Vivimos en un mundo que poco a poco se está metiendo de lleno en la Realidad Virtual. Debido a las limitaciones actuales de WebGL 2 (además de su escasa documentación), tratar de renderizar por duplicado el contenido podría llegar a ser bastante costoso. Esta funcionalidad sería muy fácil de implementar si en algún momento se añadiesen shaders de geometría a las especificaciones de Khronos.
- Depuración: Mientras todavía trabajábamos con la versión de WebGL 1 utilizamos una extensión de Chrome conocida como "WebGL Inspector". Esta extensión nos permitía consultar todos los shaders incluidos en el contexto, así como realizar trazas, obtener buffers o consultar texturas. Debido a que

WebGL 2 sigue en desarrollo, no existen herramientas que lo soporten, así que suponemos que existirá más de un error en el código no controlado.

- Mantener un diseño basado en patrón Singleton. Actualmente muchas variables son públicas globalmente y se podrían llegar a modificar en tiempo real, lo cual podría llegar a "romper" parte de la aplicación.
- Datos compartidos entre programas: Durante el desarrollo de la aplicación, diversas variables uniformes se comparten entre todos los programas en uso. A partir de WebGL 2 se permite el uso de *Uniform Buffer Objects* o *UBO*, una característica muy utilizada en las versiones de *OpenGL*. Nos gustaría, si tuviésemos en un futuro que ampliar la funcionalidad de nuestro Path Tracer, utilizar dicha funcionalidad con el fin de ahorrar todos los envíos posibles a la tarjeta de vídeo de los usuarios.
  Actualmente existe muy poca información al respecto en lo que tiene que ver con WebGL 2 (solo se encuentra información en sus especificaciones en Khronos), pero podrían utilizarse de base los propios ejemplos para OpenGL de escritorio.
- Ray casting: Una de las funcionalidades que nos gustaría haber completado era la técnica de Ray Casting para seleccionar objetos de la escena. Esta técnica, tal y como las hemos definido, utiliza las mismas funciones de intersección *ray-renderable* usadas en el código GLSL. Dicho método es definido en la función $hitRay(ray\colon scene.Ray)$.
  Si un objeto colisionaba con un rayo lanzado desde el puntero del ratón, se quedaría marcado y el usuario podría interaccionar con él.
- WebGL sigue siendo ineficiente con el uso de bucles *for* con acceso a textura. Esto ha provocado que algunos códigos no queden limpios y requieran mayor espacio.

Conclusiones

WebGL es el nuevo estándar de facto para desarrollo multiplataforma. Debido a que JavaScript, gracias a Node.js, se está extendiendo a gran cantidad de aparatos electrónicos, el incluir gráficos por computador en cualquier entorno facilita mucho a los desarrolladores de aplicaciones.

Ha sido un trabajo muy duro, debido a que desconocíamos parte de la tecnología subyacente a WebGL. Hemos aprendido WebGL sobre la marcha, a la par que aprendíamos a desarrollar un motor basado en Path Tracer.

Gracias a todos los conocimientos adquiridos durante las clases de David Miraut y Jorge López en Rendering Avanzado así como de Marcos García en Gráficos 3D y los apuntes y libros de Peter Shirley, hemos conseguido solventar bastantes errores con los que nos topamos desde el principio, como subida de texturas o determinados buffers.

Al igual que en OpenGL, todavía no existe una buena opción para trabajar con índices de texturas, de manera que nunca una textura en un programa pueda ser sustituida por otra, sino por la siguiente que se encuentre vacía. Esto nos ha producido bastantes errores y dolores de cabeza, ya que según el tipo de materiales y objetos definidos en los shaders, los identificadores se solapaban. Este hecho no es tan corriente en sombreado básico de OpenGL, ya que, en el peor de los casos, cada objeto tiene sus propios identificadores y se sustituyen en cada sombreado.

En un futuro nos gustaría poder optimizar todo el código con las nuevas mejoras que WebGL vaya soportando, así como una posible evolución a Shaders de cómputo cuando estos sean permitidos por los navegadores web.

Para finalizar, queremos agradecer la gran labor del profesor Jorge López Moreno y Marcos García Lorenzo por todas las dudas que nos ha resuelto, así como todos los consejos y palabras de ánimo para llevar el trabajo lo más adelante posible. También agradecer las palabras de ánimo de Sergio y Juanjo, nuestros superiores dentro del Human Brain Project.

Bibliografía

- Diseño y desarrollo de un juego en WebGL - http://upcommons.upc.edu/bitstream/handle/2099.1/15478/82457.pdf
- Ray Tracing in One Weekend - Peter Shirley
- Ray Tracing: The Next Week - Peter Shirley
- Realistic Ray Tracing - Peter Shirley
- Ray Tracing from the Ground Up – K. Suffern
- Reflections and Refractions in Ray Tracing - Bram de Greeve
- http://mathworld.wolfram.com/SpherePointPicking.html
- Monte Carlo: https://en.wikipedia.org/wiki/Monte_Carlo_method
- Distintos shaders encontrados en Shadertoy
- OpenGL Development Cookbook - Muhammad Mobeen Movania (Packt Publishing)
- WebGL Programming Guide:Interactive 3D Graphics Programming with WebGL (Opengl)
- Webgl Beginner´s Guide (Packt Publishing)
- Professional WebGL Programming: Developing 3D Graphics for the Web
- La Profundidad de Campo: Gráfico Explicativo - http://www.blogdelfotografo.com/profundidad-de-campo/
- http://blogs.unity3d.com/es/2011/03/09/light-probes/
- Opus 2, GLSL ray tracing tutorial
- http://madebyevan.com/webgl-path-tracing/
- http://jonathan-olson.com/tesserace/tests/3d.html
- http://reindernijhoff.net/2015/04/realtime-webgl-path-tracer/
- http://wulinjiansheng.github.io/WebGL_PathTracer/
- Modeling Illumination Variation with Spherical Harmonics – Ravi Ramamoorthi
- An Efficient Representation for Irradiance Environment Maps – Ravi Ramamoorthi & Pat Hanrahan.