Título del trabajo: Photo Realistic Rendering (PBR) on the web using WebGL

Hermann Plass Pórtulas

TRABAJO FIN DE GRADO

GRADO: INGENIERIA INFORMÁTICA

ESCUELA SUPERIOR POLITÉCNICA UPF

AÑO 2016

Director del trabajo
Javi Agenjo Asensio

Dedicado a los que me quieren, a los que faltan y a los que confiaron en mí.

Agradecimientos

Si he podido llevar a cabo este proyecto ha sido gracias a todos los que han confiado en mí. En mi madre por no lanzarme por la ventana. A Laura por las palomitas cuando necesitaba un respiro. A Tomás por su apoyo fuese cual fuese la hora. A mi hermana que siempre está atenta de mí. Y a toda esa gente loca que ha compartido mis alegrías y mis traumas.

Y a Javi! Que me ha dedicado su paciencia y sus broncas cuando más las necesitaba para corregir el camino.

Resumen

Fragmento de la película “Big Hero 8”, Disney

Physically Based Rendering (PBR) es una técnica avanzada de iluminación foto-realista que tiene como fin la simulación más precisa sobre el comportamiento de la luz, en contraposición con técnicas como Phong o Blinn shading, cuando interactúa con los diferentes materiales de una escena.

Existen técnicas como “image based lighting”(IBL) o diferentes modelos aproximados del BRDF, permiten aproximar el resultado final a un coste menor, contraponiendo un cierto nivel de precisión en los resultados y pudiendo así llevar a cabo este algoritmo a velocidades de renderizado interactivas.

La ubicuidad de la web, la evolución del motor de JavaScript en los navegadores y la existencia de APIs como WebGL para comunicarnos directamente con la GPU nos permite portar y crear aplicaciones 3D en contexto del navegador. El uso de los programas Shader permiten delegar una importante carga del algoritmo en la GPU.

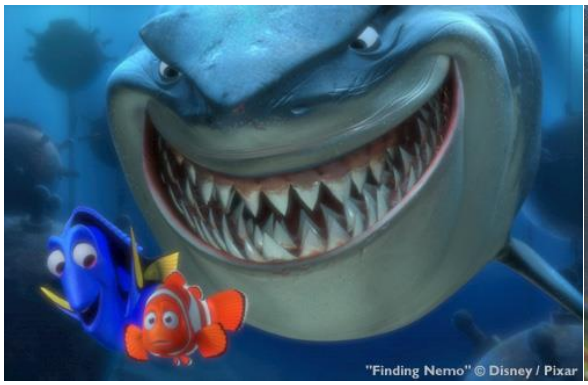
El objetivo de este proyecto es portar un render que implemente la técnica de PBR al entorno web mediante el uso de JavaScript y de la API WebGL, haciendo uso de diferentes métodos de optimización además de los ya mencionados.

Prólogo

Tras analizar diferentes técnicas de visualización 3D existentes, el resultado de un motor gráfico recae en tres propiedades: **tasa de muestreo**, **nivel de precisión de la muestra** y **memoria** utilizada. Estos tres parámetros forman parte de un triángulo donde cada uno de los valores contrapone a los otros dos.

La **velocidad de muestreo** hace referencia al número de muestras que el motor puede generar en un determinado periodo de tiempo. Para la producción de largometrajes es habitual ver contenidos generados por ordenador que hacen uso de técnicas de iluminación global como puede ser el “ray-tracing”, el “path-tracing” o “radiosity”, cada una con sus ventajas y desventajas.

Estas técnicas tienen como objetivo llevar a cabo con la mayor precisión posible la interacción de la reflexión / refracción de la luz a lo largo de la escena, teniendo en cuenta cada minuciosa contribución de luz.



“Buscando a Nemo”, Disney / Pixar (2002)


“Shrek 2”, Dreamworks (2004)

Películas como “Buscando a Nemo” Disney (2002) o “Shrek 2” DreamWorks (2004) son algunos de los primeros largometrajes animados que hemos podido ver que utilizaban la técnica de **ray-tracing**.

Estas técnicas si bien logran una **alta precisión**, el tiempo de cómputo tiende a ser muy elevado. A pesar de que se pueden hacer uso de “granjas” para el cálculo de cada frame, el tiempo requerido para renderizar un solo frame en estos largometrajes puede durar varias horas o incluso superar las 24 horas. Hay que notar sin embargo que el tiempo depende directamente de la complejidad de la escena. Adicionalmente, para llevar a cabo este algoritmo, se tienen que tener cargado en memoria todas las superficies de la escena a la vez.

Los motores para videojuegos sin embargo, se enfocan en algoritmos que proporcionen una tasa de muestreo interactiva (25 - 60fps). Para conseguirlo se suele acudir a aproximaciones menos precisas sobre la interacción de la luz.

El método más habitual y utilizado es el pre-computo. Guardar los valores que debería tomar un pixel en pantalla y almacenarnos en un fichero en memoria para más tarde poder acceder a este fichero sin más coste que el acceso a memoria.

A medida que pasan los años y el hardware evoluciona, donde antes leer una textura ya podía significar algo relativamente pesado como para limitar el número de texturas o la información almacenada en ella, el hardware que tenemos hoy en día nos permite almacenar una gran cantidad de información directamente en la memoria de la propia tarjeta gráfica, pudiendo almacenar además del color, la contribución ambiental, reflejos, parámetros pre-calculados, diferentes versiones de la misma textura, etc. (Un dato curioso que averigüé por ejemplo acerca de la limitación de la memoria es que algunos cartuchos de las primeras generaciones como Atari 2600 o NES por tal de reducir el uso de memoria, excluían de las fuentes los caracteres del alfabeto que no eran utilizados)

La evolución de los recursos disponibles hace que sea posible implementar el algoritmo de PBR como una alternativa viable al modelo Blinn-Phong en los motores a tiempo real.

La técnica de PBR descrita en “Physically Based Rendering: From Theory to Implementation” (Pharr, Humphreys) hace uso de la técnica de ray-tracing para implementar un algoritmo cuyo objetivo es acercar la rama óptica física a los modelos de renderizado y llevar a cabo una simulación precisa de como la luz debería interactuar con los diferentes materiales de las superficies de una escena.

Resumen del proyecto

Se realizará un análisis sobre el algoritmo de PBR, cuales son los fundamentales del algoritmo, las propiedades de los materiales y como debe interactuar la luz dentro del sistema. Se realizará un primer análisis de cómo integrar este algoritmo dentro de un sistema diseñado con JavaScript. Para ello partiré de la descripción incluida en el libro “Physically Based Rendering – From theory to implementation. De Matt Pharr i Greg Humpreys” además de otras fuentes de refuerzo.

En el libro, los autores implementan el PBR basándose en la técnica de ray-tracing, una técnica ya ideada en el siglo XVI por Alberch Dürer y que más tarde se adaptó para la generación de imágenes por ordenador. Sin embargo, las diferentes técnicas propias al sistema de PBR pueden ser extrapoladas a otro tipo de algoritmo sin que sea necesario la técnica de ray-tracing. La base del proyecto se trata en hallar una alternativa eficaz al algoritmo de ray-tracing capaz de ser implementado bajo un sistema basado en JavaScript y capaz de correr en la web.

Motivación

Como estudiante de último curso, siento gran interés en el sector de la imagen generada por ordenador, la animación. Por otra parte, JavaScript se convierte en un lenguaje de programación relativamente asentado y con mucha potencia que definitivamente seguirá creciendo en los próximos años, lo cual lo convierte fruto de mi interés ya que muchas salidas profesionales requieren un conocimiento amplio de JavaScript.

Si bien la técnica de PBR no es para nada novedosa, su aplicación dentro de los entornos web ha sido muy poco explorada, pocos de los ejemplos que podemos encontrar tienen implementaciones ofuscadas que tienen poco sentido didáctico. Este proyecto me brinda la oportunidad de aportar un pequeño granito de arena en la comunidad compartiendo mi experiencia sobre las dificultades de implementar este modelo de iluminación foto realista.

Calendario de proyecto

El proyecto se va a dividir en cuatro grandes partes cruciales a desarrollar a lo largo de la vida del proyecto: Análisis, Diseño, Desarrollo y Conclusión.

Análisis	Diseño
Setiembre 2015 – Octubre 2015	Noviembre 2015 – Enero 2016
- Estudio de la librería de OpenGL y el pipeline. - Shaders en OpenGL ES. - Análisis del Algoritmo de PBR. - Familiarización con JavaScript.	- Descomposición del algoritmo en alto nivel. - Decidir que librerías se pueden utilizar. - Prever los ejemplos que se van a realizar. - Estudio de los materiales de PBR.

Desarrollo	Experimentos y Conclusión
Febrero 2016 – Abril 2016	Mayo 2016 – Junio 2016
- Desarrollo incremental del framework de desarrollo. Implementación de los diferentes componentes del algoritmo.	- Prueba de rendimiento del sistema. - Prueba de funcionalidad del sistema. - Prueba de los experimentos si cumplen las estimaciones. - Realización de la memoria y conclusiones del proyecto.

Índice

Agradecimientos 2

Resumen 4

Prólogo 6

Resumen del proyecto 7

Motivación 8

Calendario de proyecto 8

1. ANÁLISIS 12

1.1. El comportamiento de la luz 12

1.2. Modelos de iluminación tradicionales 13

1.3. WebGL y el pipeline de OpenGL 14
a) Primitive Processing 15
b) Vertex shader 16
c) Fragment Shader 17

1.4. Physically Based Rendering 18
a) Descripción de la técnica 19
b) Propiedades de los materiales 20
c) Ecuaciones de Fresnel 25
d) BSDF, BRDF y el BTDF 27
e) Image Based Lighting (IBL) 29

2. Diseño 30

1.5. Estructura del Algoritmo 30
a) Implementar el algoritmo de PBR. 31

1.6. Descripción de funcionalidades 36
a) Cargar diferentes escenarios. 36
b) Cargar modelos con componentes anidados. 37
c) Visualizar los diferentes canales del modelo. 37
d) Visualizar el modelo bajo diferentes influencias de luz. 37

3. Implementación 38

3.1. Implementación del modelo Cook-Torrance 38

3.2. Shaders adaptados a cada tipo de material 47

3.3. Librerías utilizadas 48

3.4. Problemas encontrados durante el desarrollo 49

3.5. Conclusión 50

Bibliografía 52

Anexo 55

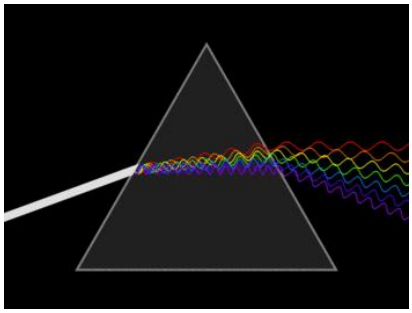
1. ANÁLISIS

En esta sección del proyecto voy a discutir sobre mis hallazgos, las partes que componen mi proyecto, como funcionan y que relevancia tienen dentro de este.

1.1. El comportamiento de la luz

La base del proyecto y de la técnica de PBR es el correcto análisis y entendimiento de que es la luz, como funciona y las formas en que la observamos.

La luz es el conjunto de dos señales perpendiculares, eléctrica y magnética usualmente referida como radiación electro-magnética la cual se ha demostrado la interacción tanto eléctrica (células fotovoltaicas) como magnética de la luz (efecto Faraday).



Dispersión de la luz a través del prisma (Wikipedia)

Las principales propiedades de la luz son: la intensidad, la dirección de propagación, la frecuencia y la polarización. La velocidad de propagación de la luz fue definida oficialmente como una constante universal en el sistema internacional de unidades en 1983 y su velocidad es de $299.792.458\ metros/segundo$. Uno de los parámetros importantes

Habitualmente el termino luz visible hace referencia al espectro de frecuencias que el ojo humano es capaz de percibir: El rango se halla entre 400 y 700 nanómetros (nm) de longitud de onda, por debajo quedan los infra-rojos y por encima los ultra-violados.

Las fuentes de luz más habituales son la luz solar, luz artificial, fuentes fluorescentes o fuentes bioluminiscentes. Estas últimas, las fuentes bioluminiscentes se deben a una reacción proteica en algunos animales o plantas marinas.

Más adelante, cuando revisemos las ecuaciones de Fresnel, veremos la relación entre la reflexión/refracción y la polarización, fase y amplitud de la luz.

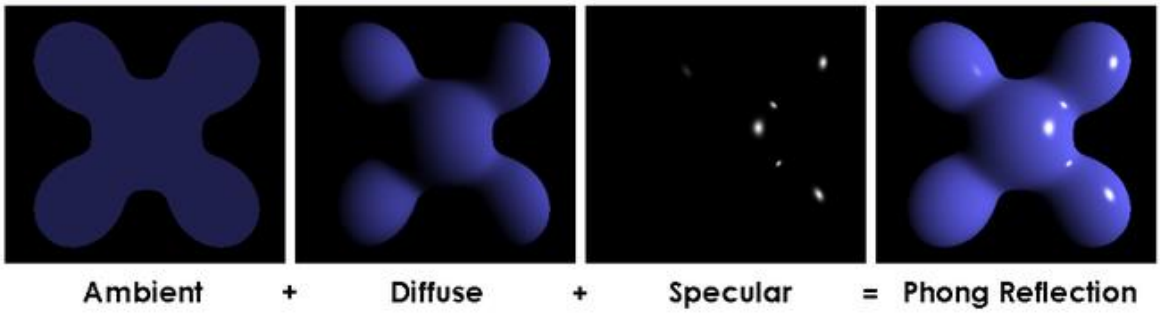
1.2. Modelos de iluminación tradicionales

Phong es uno de los modelos más útilizados y sencillos de iluminación que se puedan haber utilizado a lo largo del tiempo desde su aparición en 1973. Aparece como una solución al flat shading. Este modelo interpola entre las normales de los vértices para crear una iluminación suavizada a lo largo de la geometría. Este modelo es actualmente utilizado como el modelo de iluminación por defecto de OpenGL y Direct3D. El modelo de Phong sigue la ecuación:

$$k_a + k_d * NdotL + k_s * VdotR^{gloss}$$

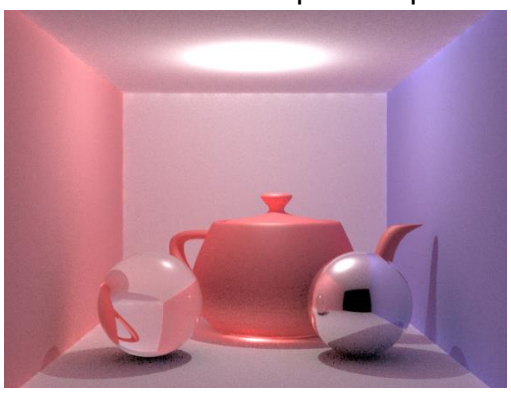
$donde$:

- k_a $contribución\ de\ la\ luz\ indirecta\ a\ causa\ de\ la\ dispersión\ de\ la\ luz\ en\ el\ medio$
- k_d $reflejos\ difusos, color\ base$
- k_s $reflejos\ especulares, rayos\ de\ la\ fuente\ de\ luz\ que\ son\ reflejados\ en\ la\ superficie$
- $NdotL$: $producto\ escalar\ entre\ la\ normal\ a\ la\ superficie\ y\ el\ vector\ luz.$
- $VdotR$: $producto\ escalar\ entre\ el\ vector\ desde\ donde\ se\ observa\ un\ punto\ p\ (ojo - p) y\ el\ vector\ reflejado\ de\ la\ luz.$
- $gloss$: $Exponente\ que\ determina\ la\ escala\ del\ brillo\ de\ la\ superficie.$



Diferentes componentes de la iluminación Phong (Wikipedia)

A pesar de no ser un modelo foto realista, para una gran mayoría de casos es más que suficiente para representar el volumen de un objeto de forma correcta. Muchas son las técnicas de iluminación avanzadas que han partido de este modelo.



Muestra generada usando iluminación Phong con Ray-Tracing (Wikipedia)

El algoritmo de PBR quiere ser un modelo que mejore el actual modelo Phong usando términos más precisos, el efecto Fresnel, conservación de la luz, rugosidad de los materiales…

1.3. WebGL y el pipeline de OpenGL

La API de OpenGL (Application Programming Interface), es el principal puente de comunicación que disponemos entre nuestro código y la GPU.

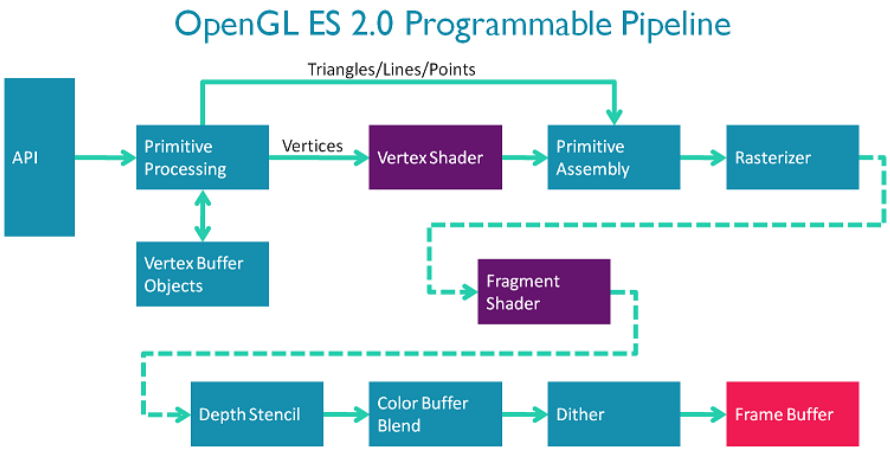
El núcleo de OpenGL está diseñado en diferentes módulos que actúan a modo de pipeline para llevar a cabo diferentes fases dentro del proceso de renderizado. A partir de la versión de OpenGL 2.0, se permite modificar la lógica interna del pipeline, pudiendo escribir programas que definen nuevos comportamientos sobre las diferentes fases del pipeline, son los **Shaders**. Solo como apunte, el término "Shader" proviene de la abreviación inglesa de "shading program".

A partir de 2009 Mozilla y Khronos se unirían en el WebGL Working Group para desarrollar una versión adaptada de OpenGL para los navegadores web. En 2011, WebGL sale a la luz en algunos navegadores (G.Chrome y M.FireFox). Esta versión está basada en la versión para dispositivos móviles OpenGL ES 2.0, se trata de una versión simplificada de OpenGL con un set de instrucciones reducido. Un manifiesto publicado por Khronos está en el anexo de este proyecto para mayor referencia.

Esta API nos permite mediante un set de instrucciones pasar información básica sobre el modelo que se quiere visualizar, y bajo qué circunstancias debe hacerlo. Alguna de esta información es:

- La primitiva con la que vamos a interpretar los VertexBuffers.
- ArrayBuffers con la información del modelo: vértices, normales, uv, color…
- Flags con para cambiar atributos del pipeline: alpha, depth, stencil, antialias…
- Texturas2D (mapeadas con vec2) y cubemaps (mapeadas con vec3)

Entendemos por rasterizado 3D como el método que permite proyectar sobre un plano 2D (la pantalla) un modelo 3D. A continuación hablare sobre las fases más importantes del pipeline.

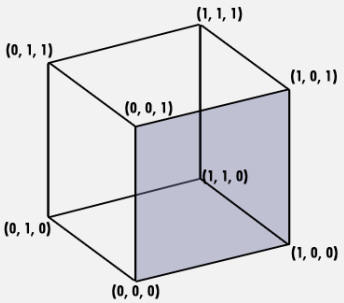


Pipeline de OpenGL ES 2.0

a) Primitive Processing

Las propiedades básicas que tenemos que transferir a la GPU para que podamos visualizar un modelo son los **arraybuffers** y la **primitiva**. Estos arraybuffers contienen datos mínimos que la GPU debe interpretar para poder rasterizar el modelo posteriormente. Uno de estos arraybuffers es el VertexBuffer, el cual contiene la información sobre la posición de los vértices en algún sistema de coordenadas 3D, habitualmente el sistema de coordenadas locales ya que no suelen modificarse.

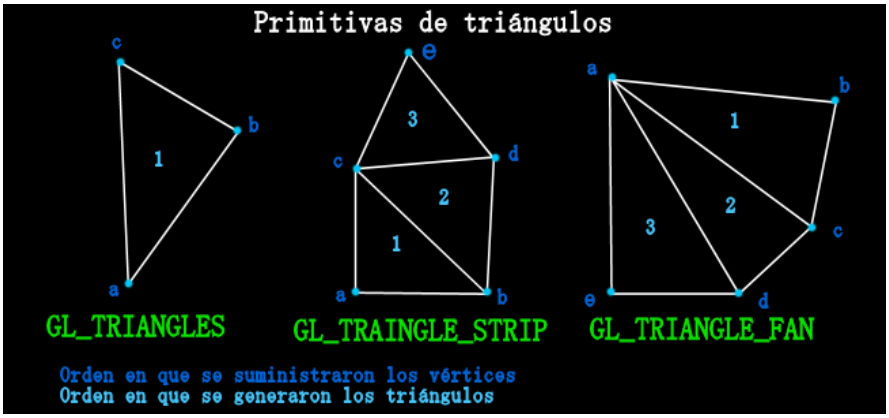
```
vertexBuffer = [
[ 0.0, 0.0, 1.0 ],
[ 1.0, 0.0, 1.0 ],
[ 1.0, 1.0, 1.0 ],
[ 0.0, 1.0, 1.0 ],
[ 0.0, 0.0, 0.0 ],
[ 1.0, 0.0, 0.0 ],
[ 1.0, 1.0, 0.0 ],
[ 0.0, 1.0, 0.0 ]
```



El modo en que el vertex buffer tiene que interpretado va directamente ligado a la primitiva que hayamos definido:

- **Puntos:** solo se tiene en cuenta la posición de cada vértice individualmente.
- **Líneas:** se interpreta la información a pares de vértices, dos puntos = una línea.
- **Triángulos:** La primera de las primitivas poligonales, la información se procesa por paquetes de tres vértices, lo cual forma el conjunto mínimo para obtener una superficie cuyos vértices siempre serán coplanarios. Tenemos tres diferentes flags para la misma primitiva:

- **GL_TRIANGLES**: Se interpreta el vertex buffer como triángulos independientes, tres vértices = 1 triangulo.
- **GL_TRIANGLE_STRIP**: En esta ocasión los triángulos están conectados. Los tres primeros vértices definen el primer triangulo, el cuarto vértice forma el segundo triangulo combinado con el segundo y tercer vertices, y así sucesivamente.
- **GL_TRIANGLE_FAN**: El primer vértice define el centro de la geometría y cada par de vértices que añadimos añade un triángulo al objeto.



Primitivas en OpenGL ES

b) Vertex shader

El Vertex Shader es el primer módulo reprogramable que vamos a encontrar en el pipeline. El objetivo de esta fase del rasterizado es proyectar las coordenadas de nuestro modelo 3D de cada vértice en el plano 2D de la pantalla. Para ello podemos hacer uso de una serie de matrices de cambio de base, en concreto queremos transformar los ejes de coordenadas del espacio global (múndo) al sistema de coordenadas de cámara, donde el front de la cámara define la profundidad y el top y el right el eje de ordenadas y abscisas respectivamente. De esta base pasamos al sistema de coordenadas de pantalla, donde se tiene en cuenta el aspecto de pantalla. Es importante esta transformación ya que nos permitirá almacenar la profundidad de cada punto en el espacio, obteniendo el Z-Buffer que posteriormente podemos utilizar para detectar oclusiones (que objeto queda ocluido por la presencia de otro).

Mediante el uso de programas Shader, podemos modificar el comportamiento de este módulo, haciendo uso de diferentes variables. El input que recibe nuestro programa son los atributos inherentes al vértice (posición, normal y coordenadas), y las variables que podamos haber cargado en memoria (int, float, vec2, 3, 4, texturas, etc) denominadas **Uniforms**. Estas variables se cargan previamente a la fase de renderizado y se alojan en memoria, la GPU únicamente enlaza la memoria con el programa mediante la directiva “uniform” delante de cada variable declarada. Mientras estemos renderizando los arraybuffers el valor de entrada de las uniforms vertex shader se mantiene constante.

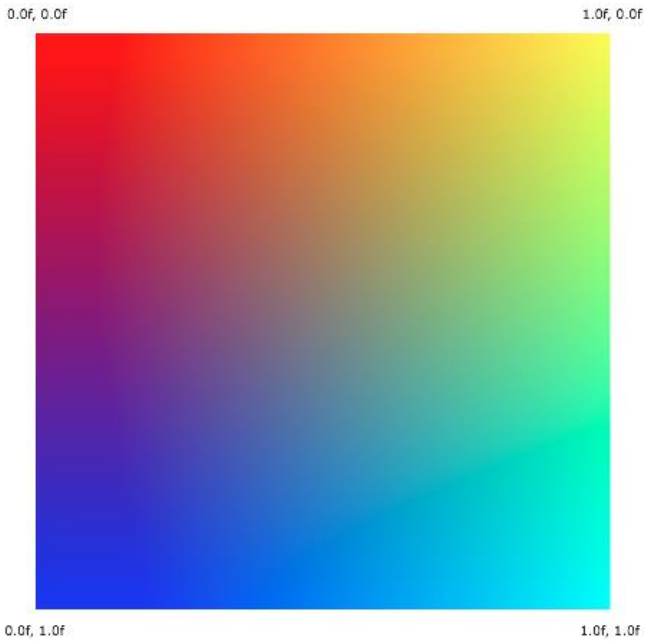
Adicionalmente existe un segundo tipo de variables que podemos definir en nuestro vertex Shader, las variables de salida de tipo **Varyings**. Estas permiten pasar información desde el vertex shader hasta el fragment shader (una fase posterior del pipeline).

Estas varying tienen una particularidad y es que como su nombre indica, varían linealmente en función del valor que toma en cada llamada del vertex shader. El valor que recibe el fragment shader es la interpolación lineal entre el valor definido entre los vértices de la primitiva utilizada y la distancia a cada uno de ellos. El uso de las varyings suelen estar destinado a variables capaces de hacerlo como valores float, vectores o matrices.

c) Fragment Shader

El Fragment shader es el segundo módulo reprogramable al que tenemos acceso en OpenGL ES 2.0. La entrada de esta fase son principalmente los varyings que podamos haber pasado desde el vertex shader así como el acceso a las mismas uniforms. La salida es el color que debería recibir el pixel en caso de que sea visible.

Como comentaba antes, las varyings son un tipo de variables que se interpolan linealmente en función de los vértices de la primitiva original. El siguiente ejemplo muestra el resultado de interpolar linealmente el color definido por vértice, generado el degradado. Esta interpolación es básica para implementar el modelo de iluminación de Phong.



Interpolación lineal de color (Wikipedia)

1.4. Physically Based Rendering

“Physically Based Rendering” (PBR), también llamado muchas veces como “Physically Based Shading” (PBS) es un conjunto de técnicas que simulan la interacción de la luz con los diferentes materiales de la escena del mismo modo que se debería comportarse en el mundo.

PBR se basa en el estudio y la formalización del comportamiento de la luz. La conservación de la energía y los patrones de dispersión de la luz son propiedades intrínsecas que el algoritmo debe tener en cuenta, así como definiciones físicamente precisas de los materiales.

Los materiales en PBR suelen hacer uso de mapas de texturas que contienen valores pre-calculados sobre cómo se verá el modelo. Tenemos el mapa de albedo, que nos define el color base del modelo; el mapa de rugosidad, que indica que tan propensa es una superficie a dispersar la luz cuando rebota; y el mapa metálico, el cual nos indica que partes del modelo son metálicos o no-metálicos.

El uso de estos mapas no es nada novedoso, ya se utilizaban antes mapas similares con otros nombres: difuso, especular o brillo. Existen más mapas que se pueden utilizar como el mapa de normales, oclusión ambiental… Incluso estos mapas se utilizan para pre-calcular varios valores.

La técnica de PBR busca un modelo sencillo de iluminación que mejore el estándar actual del modelo **blinn-phong** sin suponer por ello un elevado impacto en el rendimiento. PBR utiliza ecuaciones no tan diferentes de las que ya eran utilizadas previamente, sigue teniendo su componente ambiental, difusa y especular, únicamente estas ecuaciones calculan un resultado de un modo más preciso a como la luz debería comportarse en el mundo real.

Para el artista, encargado de generar los modelos y texturas, existían ocasiones en las que debía ajustar los assets manualmente debido a que el resultado presentaba artefactos o imprecisiones en según qué entornos de luz. La técnica de PBR termina con esto y ofrece un modelo donde unos materiales debidamente medidos en entornos de prueba concretos dan lugar a un único asset capaz de ser visualizado correctamente bajo cualquier entorno y por lo tanto puede ser reaprovechado para cualquier escenario.

a) Descripción de la técnica

El análisis de PBR se basa en dar solución a la ecuación de la luz. Esta ecuación tiene en cuenta la interacción de la luz con las propiedades de material. Describo cual es esta ecuación y a que corresponde cada parte.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f(p, \omega_o, \omega_i) \cdot L_i(p, \omega_i) \cdot |\cos \theta_i| \, d\omega_i$$ [1]

Se lee: *"La radiáncia total percibida al observar un punto p en dirección ω_o es igual a la radiáncia emitida por la superficie en el mismo punto p y en en la misma dirección ω_o sumado al sumatorio de todas las contribuciones de radiáncia incidentes en la superficie en este punto p multiplicado por el BRDF y un cierto factor coseno."*

- $\boldsymbol{L_o(p, \omega_o)}$ **:** Es la cantidad de radiáncia percibida por el ojo al observar el punto p en la dirección ω_o, en nuestro sistema se traduce como el color final que toma un pixel.

- $\boldsymbol{L_e(p, \omega_o)}$ **:** Cantidad de luz emitida por el propio material en la dirección ω_o, debido principalmente a la **fluorescencia** (radiáncia que ha incidido un tiempo anterior y que es liberada de forma retardada), **bioluminiscencia** (reacción proteica que ocurre en ciertos animales y plantas marinas) o iluminación artificial eléctrica.

- $\boldsymbol{f(p, \omega_o, \omega_i)}$ **:** El **BSDF** es encargado de determinar el porcentaje de luz incidente (en dirección ω_i) que es reflejada en dirección ω_o tras incidir en el material. En función de la rugosidad del material la luz se verá más o menos dispersada.

- $\boldsymbol{L_i(p, \omega_i)}$ **:** Luz incidente en el punto p que proviene a lo largo de la dirección $\boldsymbol{\omega_i}$ **.** Existen diferentes fuentes de luz que pueden contribuir:

  - luz directa: luz paralela, luces puntuales o de área.
  - luz indirecta: (debido a los rebotes de la luz directa en las superficies de la escena)

- $\boldsymbol{|\cos \theta_i|}$ **:** Es el valor absoluto del coseno del ángulo existente entre la normal a la superficie y el ángulo de incidencia de la luz.

[1] Formula extraída del libro Physically Based Rendering de (Humphreys & Pharr)

b) Propiedades de los materiales

Un material es el conjunto de propiedades que dan apariencia a una superficie, describe las bases de cómo debe interactuar la luz al incidir sobre la superficie para luego ser percibida por nuestro ojo. En la ecuación de la luz, el BRDF se encarga de determinar esta interacción de la luz incidente con el material. PBR hace hincapié en el uso de materiales medidos con precisión para que estos se vean bien bajo cualquier contexto, sin necesidad de alteraciones o ajustes.

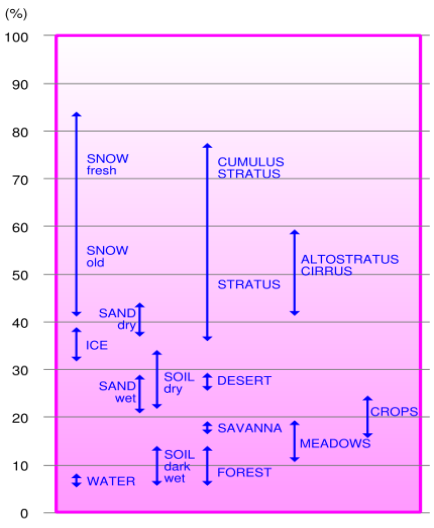
"*Las propiedades ópticas de una superficie están estrechamente relacionadas a las propiedades eléctricas de la misma. Tenemos tres principales grupos que podemos diferenciar: conductores (metálicos), dieléctricos (no metálicos) y semiconductores. Dado que los materiales semiconductores son raros de observar, por razones prácticas se simplifica en dos grupos: metálicos y no-metálicos*"

Naty Hoffman, Siggraph 2015

b.1 Albedo

Albedo es un término derivado del latín "albus" que significa "blanco". Este término detalla el coeficiente de reflexión de la luz de una superficie o que tan "blanco" se ve. El albedo se utiliza tanto para medir el color difuso en las superficies de los materiales no-metálicos (dieléctricos) y la reflectividad especular en los materiales metálicos (conductores).

En física, el albedo define el porcentaje de los fotones que son reflejados respecto al total incidente, y son los que posteriormente serán interpretados por un sensor o nuestro ojo en forma de color. Del resto de neutrones que son capaces de penetrar la superficie del material, una parte de estos serán absorbidos por la materia y transformados en calor, mientras que la otra parte eventualmente rebotarán internamente hasta salir de nuevo, es lo que observamos como el color difuso del material.



Escala Albedo (Wikipedia)

Por una parte tenemos la tierra, un material con un albedo entorno al 37-44%. Un 37-44% de la radiación incidente es absorbido por el material en forma de calor. La radiación restante es reflejada y la percibimos como el color rojizo característico de la tierra.

Por otra parte tenemos la nieve fresca, la cual tiene un albedo alto, entorno al 65-85%. Esto hace de la nieve un material frio que refleja muy bien la luz, ya que absorbe poca de la radiación del sol.

El primer principio a tener en cuenta en el algoritmo foto-realista de PBR es la conservación de la energía. El total de luz que puede ser reflejada por un material es siempre inferior o igual al total de luz incidente. La cantidad de luz no reflejada es absorbida por el material.

b.2 Rugosidad

El parámetro de la rugosidad define que tan irregular es una superficie a micro escala. Una superficie cualquiera, al ser observada a nivel microscópico, presenta irregularidades en su superficie que alteran la dirección en que los rayos de luz incidentes son reflejados, dispersando la luz.

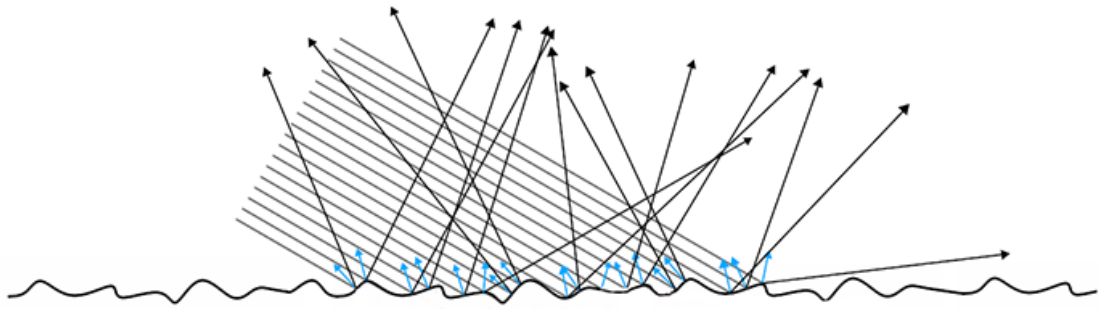
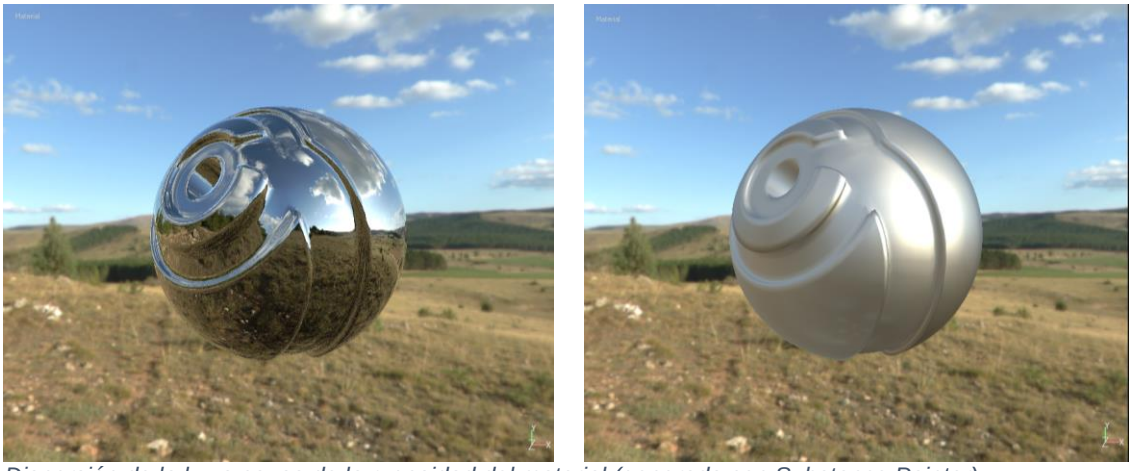


Ilustración 1 - Rugosidad de la superficie

El grafico nos muestra una sección de una superficie cualquiera vista desde el lado. Se puede ver una luz incidente paralela, como la luz del sol, que incide sobre una superficie rugosa. Los rayos reflejados se reflejan de forma irregular debido a la rugosidad del material.



Dispersión de la luz a causa de la rugosidad del material (generada con Substance Painter)

En la esfera de la izquierda, tenemos material liso, cuyos microfacets presentan una variación de la pendiente mínima y reflejos especulares perfectos. En la esfera de la derecha, los microfacets tienen una pendiente pronunciada que provocan la dispersión de la luz a nivel macroscópico.

Supongamos que tenemos dos esferas, una de ellas rugosa (como la de la derecha). La superficie aparenta ser rugosa a simple vista debido a que la luz incidente, al rebotar, es dispersada y produce el efecto de que la luz se difumina al rebotar. Del mismo modo, si tuviésemos una esfera completamente lisa, rugosidad 0, pero hubiésemos puesto un filtro difusor a la luz incidente, ambas esferas aparentarían ser iguales. Este concepto será clave a la hora de implementar nuestra iluminación mediante image based lighting (IBL), donde podemos tener diferentes texturas (que representan la contribución de luz incidente) con diferentes grados de difuminado y en función de la rugosidad del material, mapeamos la luz incidente de una textura u otra.

b.3 Índice de Reflectividad

El índice de reflectividad de un material indica el porcentaje de luz que la superficie es capaz de reflejar cuando la miramos de frente. El albedo nos determina el índice de reflexión para los reflejos difusos, la parte de la luz reflejada que consigue penetrar la capa de la superficie más externa para permitirnos ver la componente difusa del material. Sin embargo, como veremos más adelante, las ecuaciones de Fresnel detallan el fenómeno óptico en el que el índice de reflexión de una superficie varía en función del ángulo en que es observado ω_o. El término F0 (también descrita en algunos sistemas como F90 en función de si consideramos el ángulo entre la normal o entre la tangente a la superficie) detalla el índice de reflexión especular de una superficie al ser vista de frente.

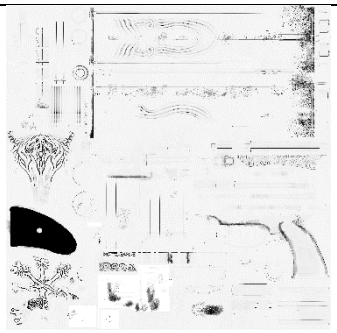
b.4 Workflows de PBR

Existen diferentes implementaciones sobre el algoritmo de PBR que se han llevado a cabo. Entre estas las diferentes implementaciones existen dos principales modos (los “workflows”) en los que alimentar el algoritmo, los canales utilizados (texturas de entrada que alimentan el shader). Dos principales workflows: el workflow “*metalness – roughness*” y el workflow “*specular – glossiness*”. Cada uno con sus pros y sus contras.

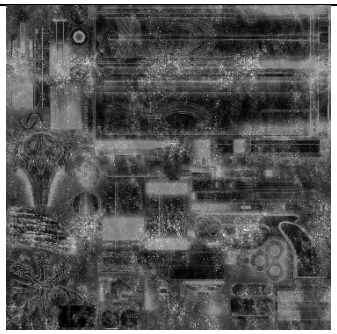
Workflow: Metalness – Roughness

(McDermott, the Comprehensive PBR Guide by Allegorithmic - vol. 2)

Albedo (sRGB)



Metalicidad (linear)



Rugosidad (linear)

La rugosidad, el color base (albedo) y la metalicidad son tres principales atributos que nuestro algoritmo y este workflow tienen en cuenta:

- **El mapa Albedo** (sRGB): contiene la información tanto para los reflejos difusos para materiales no-metálicos como para los reflejos especulares en materiales metálicos. Si el material es dieléctrico, el texel leído debe interpretarse como el índice de reflectividad difusa del material (color base), estableciendo el índice de reflexión especular a 0,04 (la mayoría de los materiales dieléctricos comunes tienen un índice de reflexión entre 0% y 8%, este workflow simplifica a un valor medio de 0.04). Si el material es conductor (metálico), el texel leído representa la reflectividad especular y se establece el índice reflectividad difusa en 0.0 (color base negro).

	Color Base : R^3	**Reflectividad (F0) : R^3**
Materiales Dieléctricos	Variable: albedo	Constante: 0.04
Materiales Metálicos	Constante: [0,0,0]	Variable: albedo

- **El mapa Rugosidad** (lineal): un valor entre 0 y 1 indica que tan rugosa es la superficie del material. Un material liso (rugosidad 0.0) tiene todos los microfacets alineados a la tangente a la superficie y se apreciará como un espejo. Una esfera cromada por ejemplo donde la rugosidad es baja, la imagen se ve perfectamente reflejada. En un material rugoso (1.0), la luz rebotada es dispersada (cambia su dirección respecto a la dirección que tomaría el reflejo perfecto) y provoca el efecto de que la luz se vea difuminada. Es el término inverso a lo que en el workflow tradicional denominaríamos glossiness (brillo). El mapa de rugosidad permite trabajar el detalle de un material sin alterar la reflectividad especular.

- **El mapa Metálico** (lineal): en función del texel registrado, el mapa indica que partes del modelo corresponden a materiales metálicos (conductores):1.0, o no-metálicos (dieléctricos): 0.0. Adicionalmente el mapa metálico se encarga de balancear la reflectividad tanto difusa como especular del modelo.

Pros:

- No rompe la ley de conservación de la energía, debido a que el índice de reflexión especular y difusa comparten una misma texura (albedo) El artista no puede alterar la textura para que tenga un alto índice para los reflejos especulares y difusos a la vez.

- Utiliza menos memoria dado que el mapa de rugosidad y el mapa metálico únicamente son una escala lineal de grises

Contras:

- Falta de control de la reflectividad especular (F0) en los materiales dieléctricos.

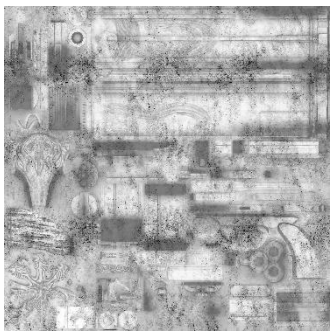
Workflow: Specular – Glossiness
(McDermott, the Comprehensive PBR Guide by Allegorithmic - vol. 2)

Este workflow tiene una terminología familiar, componente difusa, especular y glossiness (brillo). Sin embargo esto puede llevar a error porque no es exactamente lo mismo: el color difuso debe ser negro en materiales metálicos y la reflectancia difusa debe ser correctamente alojada en el mapa de especulares.

En la página de Marmoset.co se explica un tutorial sobre como portar las texturas de un workflow a otro sin pérdida de información.


Reflejos Difusos


Reflejos Especulares


Glossiness

- **El mapa Difuso** (sRGB): contiene el índice de de reflectividad difusa, también denominado color base.

- **El mapa Especular** (sRGB): contiene el índice de reflectividad especular de la superficie. (F0, reflectividad especular cuando miramos de frente la superficie)

- **El mapa Glossiness** (lineal): define el brillo (“gloss”) del material en función de que tanto rugoso es una superficie. El valor es directamente inverso a la rugosidad (1 - rugosidad).

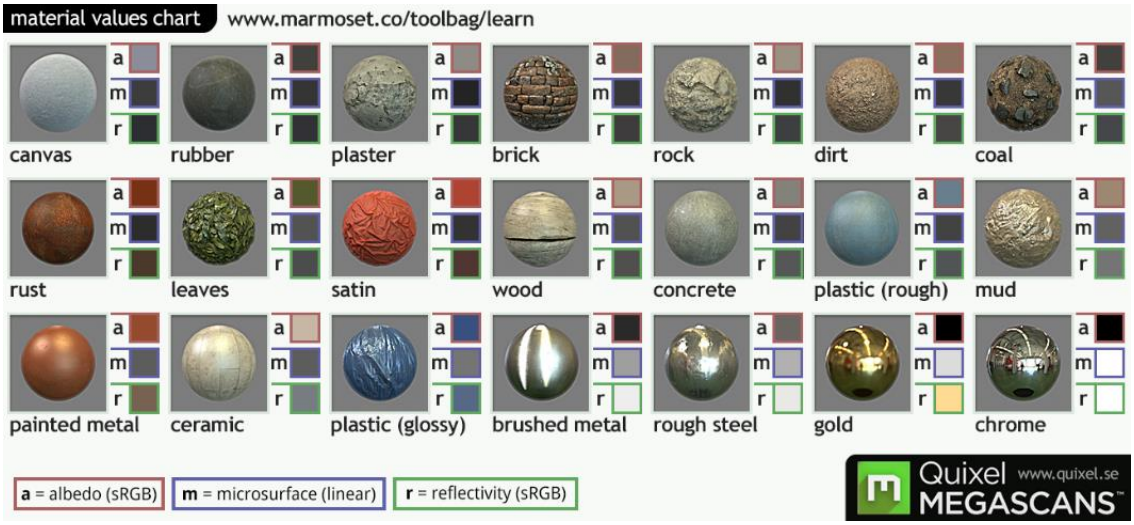
Pros:

- Control sobre el índice de reflexión especular F0 para los dieléctricos.

Contras:

- Como cada componente para los reflejos se halla en un canal diferente, el workflow es susceptible de romper la ley de la conservación de la luz si no se crean las texturas a conciencia.(Si alojamos un pixel blanco en las dos texturas tendríamos que el material tiene reflejos 100% difusos y 100% especulares cosa que es imposible)

- Utiliza más memoria ya que en este caso estamos utilizando dos texturas sRGB.



Carta de materiales PBR elaborada en entornos preparados en un laboratorio. (Marmoset.co)

c) Ecuaciones de Fresnel

Las ecuaciones de Fresnel (Augustin-Jean Fresnel) describen un comportamiento particular de la reflexión especular de la luz entorno a la superficie de un material. El cálculo describe la cantidad de luz reflejada en función del índice de refracción del medio incidente, del medio transmitido y del ángulo de incidencia.

Habitualmente cuando uno se refiere a reflectancia o transmitancia habla de un valor que es constante a lo largo de toda la superficie. Sin embargo en las ecuaciones de Fresnel, estos términos dependen directamente del ángulo en que incide la luz en la superficie.

El fondo del lago se aprecia cada vez menos a medida que alejamos la vista (National Geographic)

Podemos diferenciar su comportamiento en funcion de la polarización de la luz (giro rotacional a lo largo de la direccion de propagacion de la onda electromagnética).

- **Polarización perpendicular**: El campo eléctrico está alineado perpendicularmente al plano incidente.
- **Polarización paralela**: El campo eléctrico está alineado paralelamente al plano incidente.

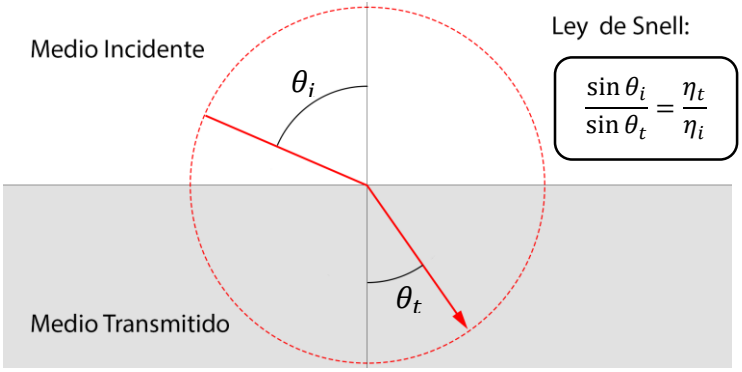
La reflectancia para los materiales no-metálicos (dieléctricos) y luz con polarización perpendicular y paralela son respectivamente:

$$r_{\perp} = \frac{\eta_i \cdot cos\theta_i - \eta_t \cdot cos\theta_t}{\eta_i \cdot cos\theta_i + \eta_t \cdot cos\theta_t} \qquad r_{\parallel} = \frac{\eta_t \cdot cos\theta_i - \eta_i \cdot cos\theta_t}{\eta_t \cdot cos\theta_i + \eta_i \cdot cos\theta_t}$$

$Donde$:

- $\boldsymbol{\eta_i}$ **y** $\boldsymbol{\eta_t}$: Índices de refracción del medio incidente y transferido.

- $\boldsymbol{\theta_i}$**y** $\boldsymbol{\theta_t}$**:** Ángulo entre la normal a la superficie y la dirección de la luz incidente/transmitida. El ángulo de la luz transmitida se puede calcular mediante la ley de Snell para la refracción de la luz:



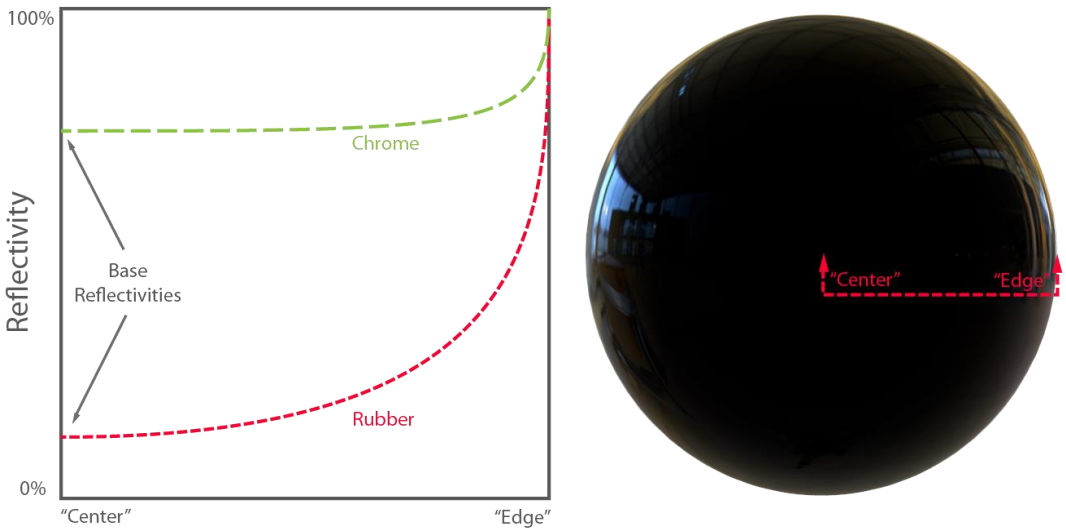
Para la luz no polarizada, la reflectancia se puede calcular como:

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2)$$

Debido a la ley de la conservación de la energía, la energía transmitida en un material dieléctrico es $1 - F_r$.
En el caso de los materiales metálicos (conductores), la parte de la energía que penetra la superficie no se ve reflejada como color difuso, es sin embargo absorbida por el material en forma de calor. El libro de PBR “From Theory to Implementation” se habla de una ecuación ampliamente utilizada para la reflectividad en materiales conductores:

$$r_{\perp}^2 = \frac{(\eta^2 + k^2) - 2\eta \cdot cos\theta_i + cos\theta_i^2}{(\eta^2 + k^2) + 2\eta \cdot cos\theta_i + cos\theta_i^2} \qquad r_{\parallel}^2 = \frac{(\eta^2 + k^2) \cdot cos\theta_i^2 - 2\eta \cdot cos\theta_i + 1}{(\eta^2 + k^2) \cdot cos\theta_i^2 + 2\eta \cdot cos\theta_i + 1}$$

donde k es el coeficiente de absorción del material en que incide la luz



Índices de refracción del cromo y la goma en función del ángulo de incidencia (Nvidia CGDC)

Estas ecuaciones pueden tener un factor de complejidad elevado si queremos evaluar la reflectividad por cada pixel que ocupe la superficie. Existe sin embargo una reformulación simplificada de la ecuación descrita por Christophe Schlick. Esta nueva fórmula permite aproximar el fenómeno descrito por Fresnel de forma más simple tomando por parámetro los índices de refracción de los medios implicados y el ángulo de incidencia de la luz:

$$F(\omega_o) = F_0 + (1 - F_0)(1 - \cos\theta)^5, \qquad F_o = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2$$

d) BSDF, BRDF y el BTDF

“Bi-directional Scatering Distribution Function” es el nombre completo de las siglas que literalmente se traduce como: “función de dispersión distributiva bidireccional”. Esta función es la principal encargada de describir el comportamiento la luz cuando incide en una superficie y da solución a la componente $\boldsymbol{f(p, \omega_o, \omega_i)}$ de la ecuación de la luz.

Un fotón al incidir sobre una superficie ve su energía dispersada. Dos principales comportamientos capturan esta dispersión: Reflexión (BRDF) y Refracción (BTDF).

Nota: $BSDF = BRDF(f_r) + BTDF(f_t)$

d.1 f_r : BRDF

Son las siglas de “Bidirectional Reflectance Distribution Function”, función que define el comportamiento de la luz que se refleja en la capa más externa de una superficie. Podemos diferenciar entre dos tipos de reflejos: Especulares y Difusos.

Los reflejos especulares son la porción de la luz incidente que no consigue penetrar la superficie del material y por lo tanto rebota como si se tratase de un espejo.
Los reflejos difusos sin embargo si consiguen penetrar la superficie del material e inmediatamente rebotan, la parte de la luz la cual luego podremos apreciar como el color difuso del material. El comportamiento del **BRDF** fue definido por **Fred Nicodemus** en 1965 como:

$$f_r(\omega_i, \omega_o) = \frac{d\, L_o(\omega_o)}{dE_i(d\omega_i)} = \frac{d\, L_o(\omega_o)}{L_i(\omega_i)\, cos\theta_i\, d\omega_i}$$

$d\, L_o(\omega_o)$ es la fracción de luz que es reflejada respecto la luz irradiada $dE_i(d\omega_i)$
Propiedades del BRDF:

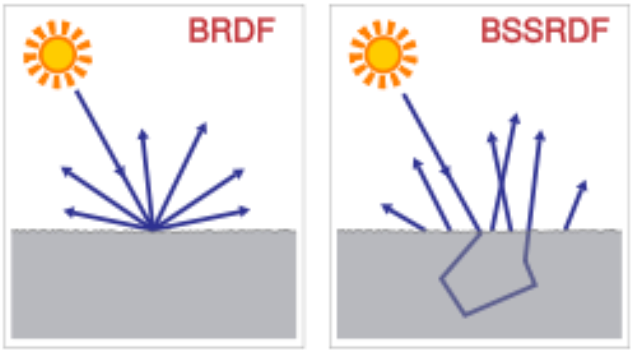
- El valor es siempre positivo: $f_r(\omega_i, \omega_o) \geq 0$
- Obedece la ley de reprocidad de Helmholtz: $f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i)$
- Y la ley de conservación de la energía: $\forall\, \omega_r, \int_\Omega f_r(\omega_i, \omega_o)\, cos\theta_i\, d\omega_i \leq 1$

d.2 f_t : BTDF

Esta es la función que permite determinar la cantidad de luz que ha conseguido penetrar la superficie totalmente, el porcentaje de luz que no ha sido reflejado como luz difusa. Esta luz sin embargo sufrirá dos posibles destinos: absorción o refracción.

- **Absorción**: una cantidad de luz es atrapada en el material y se transforma en calor del propio objeto.
- **Refracción**: luz que penetra el material con una dirección de propagación desviada debido a la nueva velocidad de propagación del medio (índice de refracción del medio). Y que eventualmente atraviesa el material para salir de nuevo a la superficie.

Hay que notar que estos comportamientos no son del todo exactos. El modelo BSSRDF (Bidirectional Subsurface Scatering Reflection Distribution Function) define como la luz que penetra la superficie experimenta también reflexión / refracción interna antes de ser absorbida o reflejada de nuevo hacia fuera del material, y es habitual de que el punto donde emerge de nuevo sea distinto al que penetró.



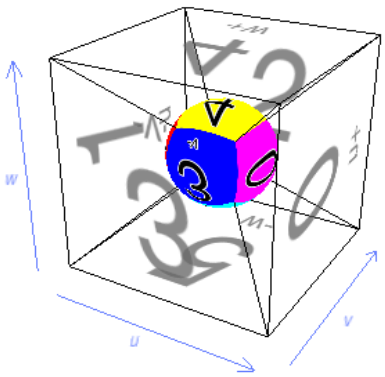
Comparativa BRDF vs BSSRDF (Wikipedia)

e) Image Based Lighting (IBL)

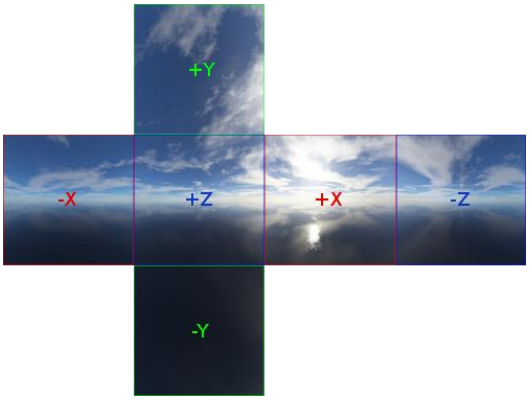
Image Based Lighting (IBL) es una técnica de iluminación que utiliza texturas, en nuestro caso utilizaremos cubemaps, para aproximar los reflejos de una superficie. En la textura se almacena la luz que incide entorno a un cierto punto de la escena p, el environment map. Es posible recrear el efecto de como la luz se dispersa tras rebotar en la superficie del material en función de dicha rugosidad.

Un cubemap es una textura especial que integra 6 texturas cuadradas de las 6 caras de un cubo en una única textura que se puede mapear mediante un vector3 (R^3).

Para generar una cara del environment map podemos acudimos a una cámara (FoV de 90º) que realiza una instantánea (o bien un volcado del framebuffer) mientras apunta en cada una de las caras del cubo desde el centro del mismo (derecha, izquierda, arriba, abajo, frente y trasera).



2El cubemap sirve de aproximación esférica (fuente:3dpanda)



Caras de un cubemap en un cubecross (fuente:http://www.txutxi.com/)

2. DISEÑO

Para llevar a cabo el diseño hay que tener algunas limitaciones claras. La aplicación deberá correr bajo el lenguaje de la web: JavaScript. Este lenguaje es muy versátil, el hecho de ser código interpretado hace que el código pueda contener errores y no mostrarse en función de la casualística, hay que ser riguroso con el código y las metodologías para escribir código: de qué forma se definen las variables y funciones, con que “scope” trabajamos en cada momento, cuando deja de utilizarse una variable…

Se busca implementar un visualizador de modelos a tiempo real (velocidades interactivas entre 25-60 fps). Si bien existen ejemplos de aplicaciones del algoritmo de ray-tracing o radiosity en los que se consiguen velocidades interactivas, habitualmente esta velocidad interactiva se debe a la simplicidad de la escena usando primitivas simples.

1.5. Estructura del Algoritmo

Veamos el orden lógico de la estructura principal del algoritmo para luego darle implementación.

1 **Petición por el fichero de la escena:** Una petición HTTP va a buscar el fichero almacenado en memoria y verifica su contenido.

2 **Parseo y carga de elementos de la escena:** Se parsean los diferentes elementos descritos en el fichero de escena, se crean las instancias necesarias y se añaden a la lista de nodos cargados.

3 **Cargar los assets asociados:** Cada elemento de la escena puede tener varias dependencias: meshes, materiales y texturas.

4 **Generar y asignar programas de shader a cada elemento en función del material:** Cada material puede tener una serie de propiedades que definen el modo en que se toman los valores del material. Mediante el uso de directivas se pueden activar y desactivar fragmentos del código específicos para cada tipo de material.

5 **Asignar la captura de eventos de los periféricos teclado y ratón:** Eventos que permiten manejar la cámara y la iluminación de la escena.

6 **Pre cálculo de las texturas y valores que sean necesarios:** Probablemente hayan diferentes valores que necesiten ser pre-calculados antes de la fase de renderizado por tal de optimizar la tasa de muestreo.

7 **Cómputo de las úniforms generales:** Posición de las luces puntuales, matrices de transformación, texturas relacionadas con el entorno, etc.
8 **Cómputo de las úniforms de cada elemento:** matriz modelo, propiedades de los materiales, texturas etc

9 **Renderizado del fondo, el skybox:** Se pinta el fondo para todos los elementos en primera instancia

10 **Renderizado de los modelos mediante shader especifico de cada elemento:** Se recorren todos los elementos que han de ser renderizados y se lanza la operación de la API de WebGL para que se renderize el modelo usando las uniforms generales y especificas con el programa Shader definido por el elemento. Cada programa Shader implementa el algoritmo de PBR.

a) Implementar el algoritmo de PBR.

El libro de *"Physically Based Rendering: From Theory to Implementation"* utiliza la técnica de ray-tracing que es incompatible con la tasa de muestreo interactiva, así que se han de buscar otros modelos posibles para llevar a cabo esto.

Necesitamos implementar un modelo de PBR que solucione la ecuación de la luz, no rompa la ley de la conservación de la energía y que funcione a una velocidad interactiva entre 25 - 60 fps (*frames per second*). El principal problema de la ecuación es solventar la integración de la luz sobre la semiesfera superior (Ω) alrededor del punto (p) evaluado. Ha de tener en cuenta a su vez como combinar el BRDF con los principales tipos de iluminación que podemos encontrar: Luz directa e indirecta.

Dos alumnos de la Universidad de Granada publicaron "An overview of BRDF models" han catalogado hasta 26 modelos diferentes para el BRDF[2]. En el paper, de la rama de los modelos teóricos, hablan de algunos modelos del BRDF que dan resultados físicamente plausibles.

En el libro de PBR "From Theory to Implementation" se implementan algunos de los modelos de BRDF basados en **microfacets**: *"Una superficie comprendida de micro-caras (microfacets) es esencialmente un mapa de alturas, donde la distribución de las caras (facets) es descrita de forma estocástica."* Los modelos basados en microfacets que utilizan son: **Oren-Nayar, Torrance-Sparrow y Cook-Torrance**. Se ha de ver por encima que facultades tiene cada uno igual y como se adapta a las necesidades del proyecto.

[2] Anexo: Diagrama sobre los diferentes modelos para el BRDF

a.3 Oren-Nayar

Este modelo se enfoca en la descripción de los **reflejos difusos**. La pareja Oren y Nayar se fijaron en que las superficies rugosas tienden a no parecerse tanto al modelo Lambertiano para reflejos difusos. El modelo describe sin embargo las superficies rugosas como un conjunto de surcos en forma de V y donde cada cara del surco (los facets del surco) si presenta reflejos perfectos Lambertianos. La distribución de estos microfacets se basa en una distribución gausiana con un único parámetro σ .El modelo se describe como:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}\,(A + B * \max(0, cos(\phi_i - \phi_o)) * \sin\alpha * \tan\beta)$$

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)}, \qquad B = \frac{0.45\,\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = \max(\theta_i, \theta_o)\,, \qquad \beta = \min(\theta_i, \theta_o)$$

Comparativa del modelo Oren/Nayar (fuente: wikipedia)

Si bien este modelo aproxima unos mejores reflejos difusos que el modelo convencional lambertiano, uno de los principales problemas del modelo Oren-Nayar es que presenta un bajo rendimiento debido a las operaciones de seno y tangente de la ecuación. Esto no supone un inconveniente para el sistema de ray-tracing que describen en “PBR: From Theory to Implementation” ya que su objetivo es la precisión purista sin importar el tiempo de renderizado, en nuestro proyecto queremos conservar unos fps’s altos y esta técnica parece a priori no ser conveniente.

a.4 Torrance-Sparrow

En este modelo para **reflejos especulares**, al igual que en el modelo Oren-Nayar, se define la superficie como un conjunto de microfacets, donde en este caso los microfacets están orientados de forma aleatoria y siguiendo una determinada función de distribución $D(\omega_h)$, que indica que porcentaje de microfacets están alineados con el vector ω_h.

$$f_r(p, \omega_o, \omega_i) = \frac{k_d}{\pi} + k_s \frac{D(\omega_h)\,F_r(\omega_o)\,G(\omega_o, \omega_i)}{4\pi\,(n \cdot l)}$$

Donde:

- $D(\omega_h)$, es la función de distribución NDF(“normal distribution function”): esta función determina que tan brillante se ve una superficie a causa de la orientación de los microfacets en torno al vector “halfway” H. Este valor puede tomar entre $[0, \infty]$

$$D(\omega_h) = \frac{e^{\left(\frac{\cos\beta^2 - 1}{m^2 \cos\beta^2}\right)}}{m^2 cos\beta^4}$$

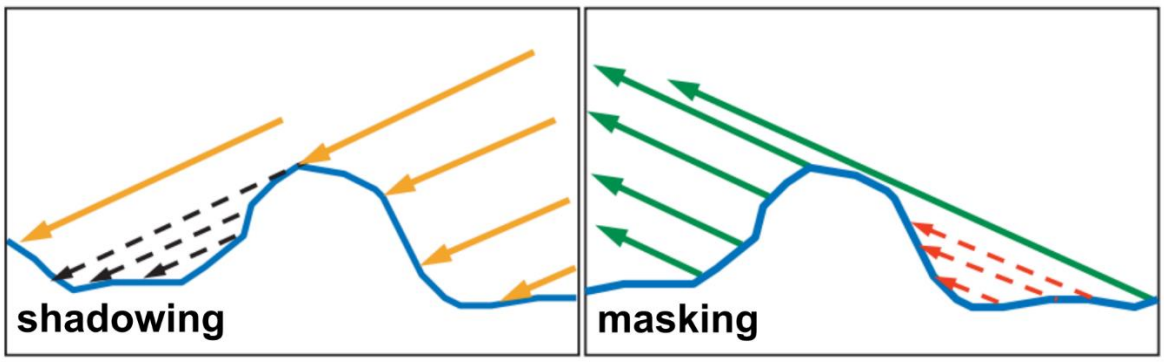
donde:
$m = media\ cuadratica\ de\ la\ pendiente\ de\ los\ microfacets,$
$\beta\ = angulo\ entre\ el\ vector\ normal\ N\ y\ el\ halfway\ vector\ H$

- $F(\omega_o)$, es el factor de Fresnel, donde se utiliza una versión aproximada de Schlick:

$$F(\omega_o) = F_0 + (1 - F_0)(1 - \cos\theta)^5$$
$$F_o(Reflexion\ en\ angulo\ 0) = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2$$

- $G(\omega_o, \omega_i)$, factor de atenuación geométrica. Este valor trata de definir el índice de luz ocluida por la propia superficie a causa de los facets.

$$G(\omega_o, \omega_i) = \min\left\{1, \frac{2(n \cdot \omega_h)(n \cdot \omega_o)}{(\omega_o \cdot \omega_h)}, \frac{2(n \cdot \omega_h)(n \cdot \omega_i)}{(\omega_o \cdot \omega_h)}\right\}$$



Images from “Real-Time Rendering, 3rd Edition”, A K Peters 2008

a.5 Cook-Torrance

La conferencia de SIGGRAPH es un evento annual que se lleva a cabo para presentar avances en investigación en términos de animación, técnicas de iluminación etc… En la presentación de 2013, Brian Karis (Epic Games) hacia una introducción al motor de shading que utiliza el Unreal Engine 4. Describiendo como integran el modelo de Cook-Torrance usando la técnica de Image-Based Lighting mediante el uso de muestras contrapesadas (Importance sampling).

Para el modelo de reflejos difusos se opta por el modelo tradicional lambertiano debido a su grado de rendimiento / calidad. El termino difuso del BRDF queda pues:

$$f_r(p, \omega_o, \omega_i) = \frac{albedo_{diff}}{\pi}$$

Para el modelo de reflejos especulares se utiliza el modelo Cook-Torrance que se deriva del modelo Torrance-Sparrow y es prácticamente idéntico:

$$f_r(p, \omega_o, \omega_i) = \frac{D(\omega_h)\, F_r(\omega_o)\, G(\omega_o, \omega_i)}{4\,(n \cdot v)\,(n \cdot l)}$$

Donde:

- $D(\omega_h)$, es de nuevo la "normal distribution function"(NDF) aproximada mediante el modelo GGX/Trowbridge-Reitz ya que ofrece un brillo con la decaída más larga a favor de un rendimiento aceptable:

$$D(\omega_h) = \frac{\alpha^2}{\pi((n \cdot h)^2\,(\alpha^2 - 1) + 1)^2}\,, \qquad \alpha = rugosidad^2;$$

- $F(\omega_o)$, el termino fresnel utiliza el modelo aproximado de Schlick pero con una modificación para aproximar el valor a un menor coste:

$$F(\omega_o, \omega_h) = F_0 + (1 - F_0)2^{(-5.55473(\omega_o \cdot \omega_h) - 6.98316)(\omega_o \cdot \omega_h)}$$

- $G(\omega_o, \omega_i)$, es el factor de atenuación geométrica. Aproximada esta vez mediante una modificación del modelo de Smith_GGX remapeando el valor de k:

$$G(\omega_o, \omega_i) = G_1(\omega_o) G_1(\omega_i)$$

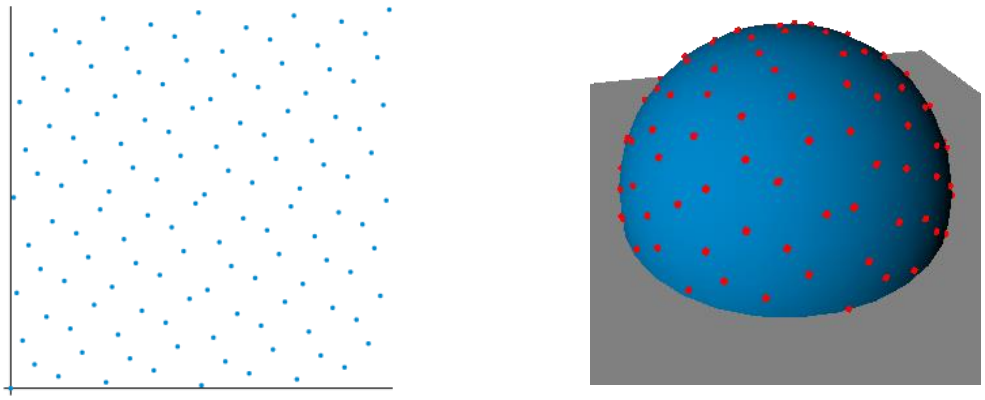
$$k = \frac{(Rugosidad + 1)^2}{8}\ (para\ luces\ puntuales), k = \frac{(Rugosidad)^2}{2}\,(IBL),$$

$$G_1(v) = \frac{(n \cdot v)}{(n \cdot v)(1 - k) + k}$$

a.6 Sampling

Para poder integrar la ecuación de la luz debemos solucionar primero la integración de la luz incidente a lo largo del punto p. El modo principal que tenemos es mediante la discretización de la integral mediante el uso de un sumatorio de muestras. Estas muestras deben aproximar todos los rayos de luz que inciden en la superficie.

Las muestras deben estar repartidas de forma equitativa en la semiesfera definida por la normal en el punto p. La secuencia Hammersley es una función que permite generar una serie de puntos a lo largo de la semiesfera superior repartidas de una forma relativamente equitativa (baja discrepancia):



128 muestras generadas con secuencia Hammersley

Esta secuencia es útil para mapear la contribución de la luz entorno a un punto en una superficie perfectamente difusa. Sin embargo es solo uno de los casos por los que deberíamos utilizar esta secuencia. **Importance Sampling** es una técnica estadística que se basa en la distribución de las muestras entorno a un punto de importancia. Este punto de importancia en el ámbito de los reflejos viene marcado por el reflejo especular y la importancia en este caso viene marcado por la rugosidad del material.

$$L_0(p,\omega_o) = L_e(p,\omega_o) + \int_{\Omega} L_i(\omega_i)\, f(p,\omega_o,\omega_i) \cos\theta_i \, d\omega_i \approx \frac{1}{N}\sum_{k=1}^{N} \frac{L_i(\omega_{i_k}) f(p,\omega_o,\omega_{i_k}) \cos\theta_{i_k}}{p(\omega_{i_k},\omega_o)}$$

Donde:

- donde k es la k-esima muestra de N muestras.
- $p(\omega_{i_k},\omega_o)$ es la función de densidad de probabilidad "probability density function"(pdf):

$$\int_{\Omega} p(\omega_i,\omega_o)\, d\omega_i \approx 1$$

1.6. Descripción de funcionalidades

Para evitar la equivocación vamos a analizar qué es lo que queremos que nuestro visualizador de modelos PBR haga. Diferentes casos de uso puede definir diferentes requisitos de los componentes del algoritmo.

a) Cargar diferentes escenarios.

Para nuestra implementación, el escenario requiere tener una serie de componentes básicos:

- Cámara/s
- Modelos a visualizar cada uno con su mesh, material y el shader que va a utilizar
- El skybox
- Luces puntuales

Dado que nuestro código estará hecho bajo JavaScript, la solución más rápida es crear un fichero JSON con la descripción de la escena que contenga todos estos parámetros. JavaScript tiene funcionalidades nativas para transformar cadenas de texto (que este correctamente codificado) a objetos de JavaScript que funcionan por clave valor del mismo modo que podríamos usar un XML. Por otro lado necesitaremos crear un parseador para este JSON, que detecte las diferentes propiedades y cree las diferentes instancias correctamente.

Si bien no existen propiamente dicho las “clases” como tal, JavaScript ofrece una aproximación con las que se pueden implementar algunos de los patrones de software similares. Para este caso decido utilizar un patrón de creación de tipo “Factory”, donde tengo una clase abstracta única que posee las propiedades comunes a todos los objetos.

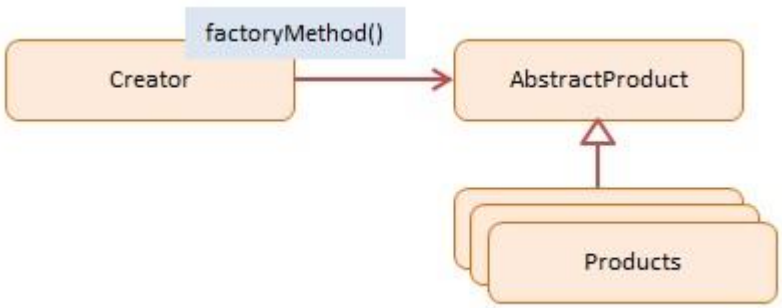


Diagrama del patrón de creación Factory

Básicamente la idea es tener una clase abstracta “NodoAbstracto” con las propiedades comunes a todos los objetos de una escena: id y posición. Luego implementar un constructor por cada tipo de instancia que pueda recibir, tener una instancia cámara, una instancia skybox, una instancia para los nodos que han de cargar una mesh etc… En cada constructor puedo definir las propiedades características de cada elemento: si necesita cargar una mesh, si tiene un método de renderizado especial, si utiliza materiales, etc.

b) Cargar modelos con componentes anidados.

Existen ocasiones en las que el modelo a visualizar contendrá definiciones de diferentes materiales, el sistema ha de ser capaz de cargar y gestionar todas las intstancias de materiales que utilice el modelo a visualizar.

El formato a la hora de cargar el modelo no es demasiado relevante. Este proyecto únicamente necesita de los arraybuffers del modelo con sus vértices, normales, coordenadas, índices etc… Lo que realmente es importante para este proyecto es enlazar las diferentes partes del modelo con su material respectivo.

Mi idea principal ha sido utilizar el formato wavefront “obj” con el que ya que he trabajado otras veces y aprovechar que tienen asociados ficheros mtl con la descripción del material. A pesar de que estos mtl poseen información limitada, hacer un pequeño ajuste manual es más rápido que ponerme a investigar cómo se implementa un parser para otro tipo de formatos. Si realmente es crucial el cambio de formato, se pueden encontrar ejemplos ya hechos en gitHub.

c) Visualizar los diferentes canales del modelo.

Ya que el objetivo del visualizador es puramente educativo, sería bueno incluir una pequeña interfaz gráfica con la que poder descomponer la fase de renderizado en sus diferentes etapas y/o los canales que utiliza. Dat.gui es una librería de JavaScript simple y fácil de utilizar con la que implementar pequeñas interfaces gráficas para este tipo de experimentos, con sliders, checkboxes, etc. Se puede añadir una lista desplegable con la que pasar un flag al shader y así poder escoger que partes mostrar.

d) Visualizar el modelo bajo diferentes influencias de luz.

Este requisito lo podemos cumplir de varias maneras o incluso de todas las posibles a la vez. La más sencilla es cargar diferentes environment map para una misma escena, algo que podríamos seleccionar con la misma GUI y ver el resultado al momento. Otra opción puede ser tener diferentes luces puntuales en la escena y poderlas mover manualmente o que rotasen alrededor del modelo. Y la tercera opción sería hacer rotar el environment map alrededor del modelo.

3. IMPLEMENTACIÓN

La mejor manera de llevar a cabo esta implementación es reconstruir a modo prueba y error la evolución que ha seguido, que problemas he encontrado y como solucionarlos.

3.1. Implementación del modelo Cook-Torrance

El modelo que he decidido implementar para este proyecto ha sido el modelo Cook-Torrance, un modelo simplificado del modelo Torrance-Sparrow. En el curso sobre PBR del Siggraph 2013, Brian Karis hizo una presentación detallada de cómo se implementa el algoritmo de PBR a tiempo real en el Unreal Engine 4. Esta descripción está llena de detalle y he optado por seguirla. Discutiremos pues como llevar a cabo dicha implementación.

Para la componente difusa se ha optado por el modelo lambertiano tradicional ya que, como comentábamos, el modelo Oren-Nayar para los reflejos difusos es bastante pesado.

```
//Lambert
vec3 diffTerm = baseColor / PI;
```

La componente especular sigue el modelo Cook-Torrance del BRDF:

```
//Cook Torrance BRDF
vec3 specTerm = D * F * G / ( 4.0 * NdotL * NdotV );
```

Donde:

- **D : Normal Distribution Function (NDF) Term (GGX_Smith)**

```
// Normal Distribution Function Term (GGX_Smith)
float NDF(float alpha, float NdotH){
    float alpha2 = alpha * alpha;
    float den = NdotH * NdotH * (alpha2 - 1.0) + 1.0;
    return alpha2 / (PI * den * den);
}
```

- **F : Fresnel Term (Schlick Optimized)**

```
// Fresnel Term : specular reflection at grazing angles
vec3 Fresnel_Schlick(vec3 F0, float LdotH){
    float power = (-5.55473 * LdotH - 6.98316) * LdotH;
    return F0 + (1.0 - F0) * pow(2.0,power);
    //return F0 + (1.0 - F0) * pow(1.0 - LdotH,5.0); //Schlik
}
```

- **G : Geometry Attenuation Term**

```
// Geometry Term : Geometry masking / shadowing due to microfacets
float GGX(float NdotV, float k){
    return NdotV / (NdotV * (1.0 - k) + k);
}

float G_Smith(float roughness, float NdotV, float NdotL){
    //float k = (roughness + 1.0)*(roughness + 1.0) / 8.0;point light
    float k = (roughness)*(roughness) / 2.0;//ibl
    return GGX(NdotL, k) * GGX(NdotV, k);
}
```

Para solventar la integral de la ecuación implementamos un sumatorio de N muestras que siga:

$$\frac{1}{N}\sum_{k=1}^{N}\frac{L_i(\omega_{i_k})f(p,\omega_o,\omega_{i_k})\cos\theta_{i_k}}{p(\omega_{i_k},\omega_o)}$$

Para cada muestra del sumatorio nos interesa encontrar el vector H, que simula la normal de un microfacet cualquiera. Cada muestra del sumatorio es generada mediante la secuencia **Hammersley**, la cual retorna el vector Xi, un vector 2 que mapea los ejes x,y entre [0,0] y [1,1].

```
vec2 Hammersley(const int index, const int numSamples){

    // Special fake Hammersley variant for WebGL and ESSL,
    // since WebGL and ESSL do not support uint and bit wise operations
    vec2 r = fract(vec2(float(index) * 5.3983,
                   float(int(int(2147483647.0) - index)) * 5.4427));
    r += dot(r.yx, r.xy + vec2(21.5351, 14.3137));

    return fract(vec2(float(index) / float(numSamples), (r.x * r.y) *
95.4337));
}
```

La función **ImporanceSampleGGX** recibe por entrada la muestra Xi la cual contiene un par de ángulos **phi y theta,** que son respectivamente los ángulos **azimut** y **altura** que indican un punto de la semiesfera superior. Esta función nos permite calcular una muestra **H** en espacio tangencial utilizando el valor $\boldsymbol{\alpha^2}$ para abrir o cerrar el arco de apertura de la altura de la muestra. Para poder trabajar con esta muestra debemos pues realizar primero un cambio de base usando la normal como referencia, el vector resultante se halla en espacio de mundo.

```
vec3 ImportanceSampleGGX( vec2 Xi, float Roughness, vec3 N ) {

    float a = Roughness * Roughness;

    float Phi = 2.0 * PI * Xi.x;

    float CosTheta = sqrt((1.0 - Xi.y) / ( 1.0 + (a*a - 1.0) * Xi.y ) );
    float SinTheta = sqrt( 1.0 - CosTheta * CosTheta );

    vec3 H;
    H.x = SinTheta * cos( Phi );
    H.y = SinTheta * sin( Phi );
    H.z = CosTheta;

    vec3 UpVector = abs(N.z) < 0.999999 ? vec3(0.0,0.0,1.0) :
vec3(1.0,0.0,0.0);
    vec3 TangentX = normalize( cross( UpVector, N ) );
    vec3 TangentY = cross( N, TangentX );

    // Tangent to world space
    return TangentX * H.x + TangentY * H.y + N * H.z;
}
```

Y para integrar el sumatorio de la ecuación de la luz, finalmente:

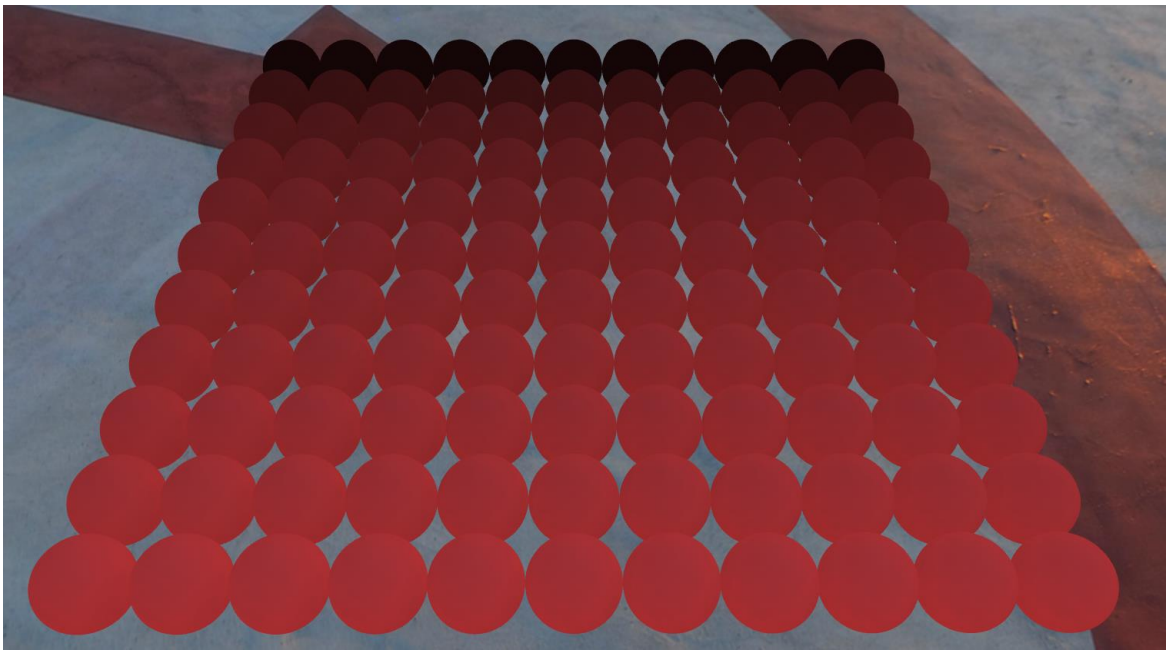
```
vec3 SpecularIBL(vec3 F0, float Roughness, vec3 N, vec3 V){

    float NdotV = max(0.0,dot(N,V));//292

    vec3 specularLighting = vec3(0.0);
    const int samples = 128;
    for(int i = 0; i < samples; i++){
        vec2 Xi = Hammersley( i, samples );
        vec3 H  = ImportanceSampleGGX( Xi, Roughness, N );
        vec3 L  =  2.0 * dot( V, H ) * H - V;//reflect(-V,H);

        float VdotH = max(0.0,dot(V,H));
        float NdotH = max(0.0,dot(N,H));
        float NdotL = max(0.0,dot(N,L));
        float LdotH = max(0.0,dot(L,H));

        if( NdotL > 0.0){
            vec3 Li = textureCubeLodEXT(u_map_env_texture, L,0.0).xyz;

            //Specular Term
            float D = NDF( Roughness * Roughness, NdotH );
            vec3  F = Fresnel_Schlick( F0, LdotH );
            float G = G_Smith( Roughness, NdotV, NdotL );

            vec3 f = D * F * G / (4.0 * NdotL * NdotV);
            float pdf = D * NdotH / (4.0 * VdotH);

            specularLighting += Li * f * NdotL / pdf;
        }
    }
    return specularLighting / float(samples);
}
```

Veamos algún resultado sobre esta implementación recordando algunas propiedades sobre los materiales metálicos y no metálicos.

	Color Base : R^3	**Reflectividad (F0) : R^3**
Materiales Dieléctricos	Variable: albedo	Constante: 0.04
Materiales Metálicos	Constante: [0,0,0]	Variable: albedo

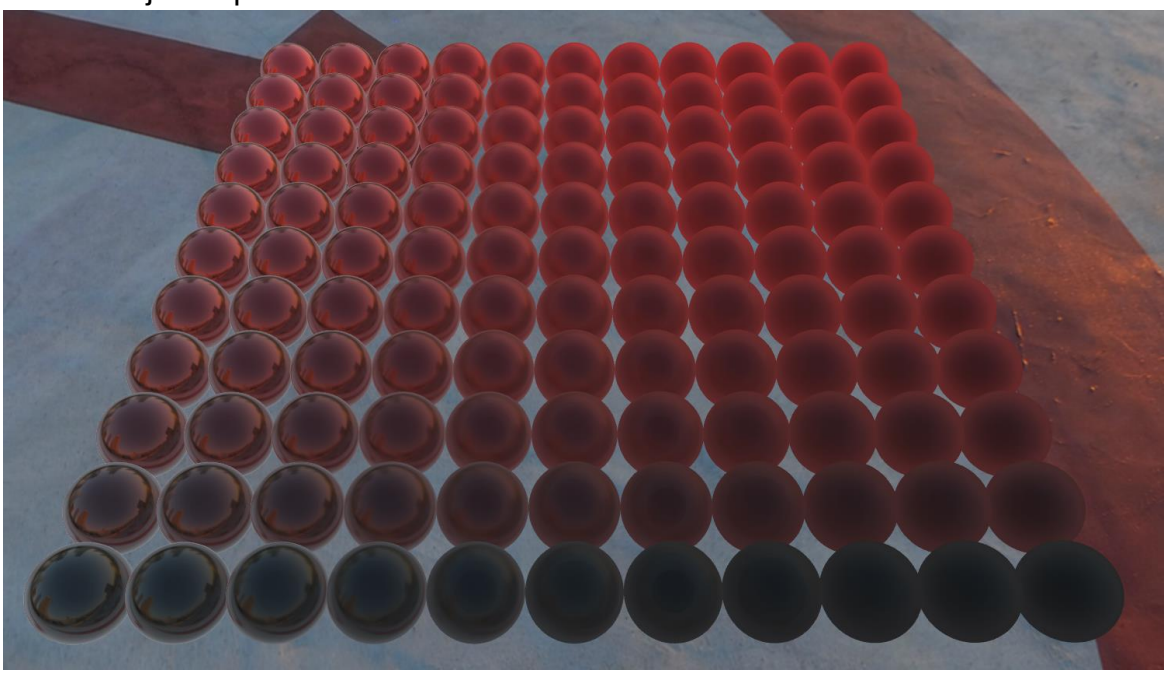
Las imágenes de a continuación representan la rugosidad ([0-1 en el eje horizontal]) y la metalicidad ([0-1] en el eje vertical) del material.

Solo Reflejos Difusos:



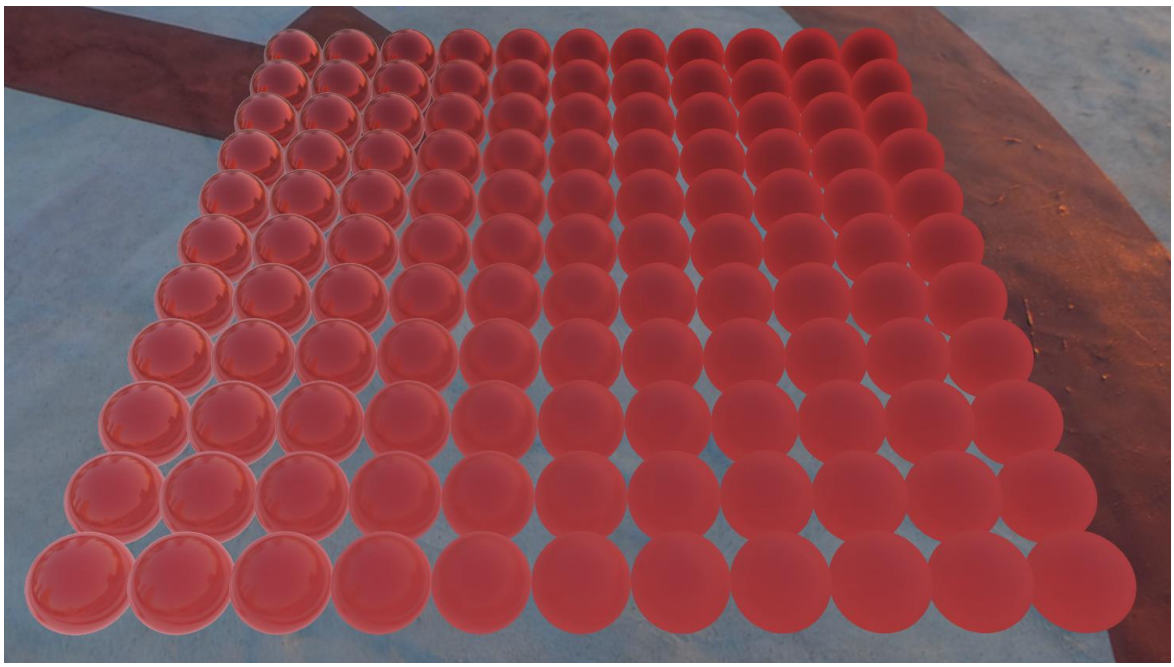
El color albedo disminuye a medida que incrementa la metalicidad del material

Solo Reflejos Especulares:



El valor F0 de la reflectividad en ángulo 0, varía en función de la metalicidad del material

Y el Resultado final:



Para esta implementación se ha hecho uso de un total de 128 muestras para calcular la componente especular del BRDF. A pesar de que el objetivo de este proyecto es puramente académico y no se pretende optimizar el algoritmo (hay gente mucho más profesional que yo dedicándose a eso), la tasa de muestreo del experimento roza los límites aceptables, entre 24 y 29 muestras por segundo evaluado con una tarjeta gráfica NVIDIA GTX 970.

Hay varias optimizaciones relativamente sencillas que podemos aplicar para obtener un par de muestras adicionales por segundo, sin embargo el paper de Brian Karis nos explica un modo optimizar por completo el algoritmo mediante un producto de sumatorios. Cada parte del sumatorio se puede precalcular alojando el valor en una “look up texture” (LUT).

$$\frac{1}{N}\sum_{k=1}^{N}\frac{L_i(\omega_{i_k})f(p,\omega_o,\omega_{i_k})\cos\theta_{i_k}}{p(\omega_{i_k},\omega_o)} \approx \left(\frac{1}{N}\sum_{k=1}^{N}L_i(\omega_{i_k})\right)\left(\frac{1}{N}\sum_{k=1}^{N}\frac{f(p,\omega_o,\omega_{i_k})\cos\theta_{i_k}}{p(\omega_{i_k},\omega_o)}\right)$$

El objetivo es alimentar el shader con estas texturas precalculadas y combinarlo con las propiedades del material. Veamos que texturas se pueden utilizar y cómo implementarlas en el proyecto.

$$\left(\frac{1}{N}\sum_{k=1}^{N} L_i(\omega_{i_k})\right) \quad \left(\frac{1}{N}\sum_{k=1}^{N} \frac{f(p,\omega_o,\omega_{i_k})\cos\theta_{i_k}}{p(\omega_{i_k},\omega_o)}\right)$$

La primera parte del producto es el sumatorio que hace referencia la luz incidente entorno al punto p en sentido ω_{i_k}. El objetivo es entonces utilizar la misma técnica que hemos utilizado previamente para pre-calcular los diferentes environment-map difuminados acordemente a los diferentes niveles de rugosidad que recoja la luz incidente del entorno. Luego podemos extraer la luz para el punto concreto de rugosidad interpolando entre las texturas más cercanas.

Para ello vamos a crear un método offline (fuera del render loop) que nos calcule una textura difuminada para un cierto nivel de rugosidad. En esta ocasión el modo rápido que tenemos para hacer esto es renderizar las seis caras del cubemap incluyendo este método

```
vec3 PrefilterEnvMap(vec3 R, float Roughness){
    vec3 N = R;
    vec3 V = R;
    vec3 PrefilteredColor = vec3(0.0);
    float TotalWeight = 0.0;
    const int NumSamples = 1024;

    for(int i = 0; i < NumSamples; i++){
        vec2 Xi = Hammersley( i, NumSamples );
        vec3 H  = ImportanceSampleGGX( Xi, Roughness, N );
        vec3 L  = 2.0 * dot( V, H ) * H - V;
        float NdotL = clamp( dot( N, L ), 0.0, 1.0 );
        if(NdotL > 0.0){
            PrefilteredColor += NdotL * Li(L,0.0);
            TotalWeight += NdotL;
        }
    }
    return PrefilteredColor / TotalWeight;
}
```

Cuando las tengamos todas las texturas calculadas podemos mirar de combinarlas en los diferentes niveles de mip-map de una única textura.

Uno de los grandes beneficios de hacer este tipo de precálculos offline es que el impacto únicamente afecta durante el tiempo de carga, previo a la fase de renderizado. Si bien un número alto de muestras puede llevar a un largo tiempo de carga, el tiempo posterior para leer esta Look Up Texture es ínfimo.

$$\left(\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{k=1}}^{\boldsymbol{N}}\boldsymbol{L_i}(\boldsymbol{\omega_{i_k}})\right)\quad\left(\frac{1}{N}\sum_{k=1}^{N}\frac{f(p,\omega_o,\omega_{i_k})\cos\theta_{i_k}}{p(\omega_{i_k},\omega_o)}\right)$$

La segunda parte del producto está relacionado con el BRDF. Hay que averiguar un modo en el que podamos desvincular las propiedades intrínsecas del material con un valor que sea pre-calculable. Primero procuremos simplificar la ecuación para ver con que nos quedamos.

$$\frac{1}{N}\sum_{k=1}^{N}\frac{\frac{D(\omega_h)\,F_r(\omega_o)\,G(\omega_o,\omega_i)}{4\,(n\cdot v)\,(n\cdot l)}(\mathrm{n}\cdot\omega_\mathrm{i})}{\frac{D(\omega_h)\,(\mathrm{n}\cdot\omega_\mathrm{h})}{(4\ *(\omega_\mathrm{o}\cdot\omega_\mathrm{h}))}}=\frac{1}{N}\sum_{k=1}^{N}\frac{\cancel{D(\omega_h)}\,\cancel{F_r}(\omega_o)\,G(\omega_o,\omega_i)\cancel{(\mathrm{n}\cdot\omega_\mathrm{i})}*\cancel{4}*(\omega_\mathrm{o}\cdot\omega_\mathrm{h}))}{\cancel{4}*\,(n\cdot\omega_o)\cancel{(n\cdot\omega_i)}\cancel{D(\omega_h)}\,(\mathrm{n}\cdot\omega_\mathrm{h})}$$

$$\boxed{\frac{1}{N}\sum_{k=1}^{N}\frac{F_r(\omega_o)\,G(\omega_o,\omega_i)(\omega_\mathrm{o}\cdot\omega_\mathrm{h})}{(n\cdot\omega_o)\,(\mathrm{n}\cdot\omega_\mathrm{h})}}$$

En el workflow que utilizamos, tenemos tres principales entradas: albedo, rugosidad, metalicidad. De estos tres valores extraemos el color base (reflectividad difusa), el valor F0 con la reflectividad especular para el término Fresnel y la rugosidad. El color base lo podemos descartar, no entra dentro de la ecuación, nos queda F0 (que utilizamos para el termino fresnel) y la rugosidad (geometry term).

Brian Karis describe que se puede extraer el termino F0 de la ecuación de fresnel, y mapear el valor de la rugosidad usando las uv de la textura donde la coordenada x puede mapear el valor del $\boldsymbol{\cos\theta}$ y el eje de las y mapear la **rugosidad**. Antes de dar nada por supuesto, primero veamos cómo podemos llegar a esta misma conclusión.

$$F_r(\omega_o)=F_0+(1-F_0)(1-\cos\theta)^5$$
$$F_r(\omega_o)=F_0-F_0(1-\cos\theta)^5+(1-\cos\theta)^5$$
$$F_r(\omega_o)=F_0(1-(1-\cos\theta)^5)+(1-\cos\theta)^5$$

$$\frac{1}{N}\sum_{k=1}^{N}\frac{(F_0(1-(1-\cos\theta)^5)+(1-\cos\theta)^5)G(\omega_o,\omega_i)(\omega_\mathrm{o}\cdot\omega_\mathrm{h})}{(n\cdot\omega_o)\,(\mathrm{n}\cdot\omega_\mathrm{h})}$$
$$\frac{1}{N}\sum_{k=1}^{N}\frac{F_0(1-(1-\cos\theta)^5)G(\omega_o,\omega_i)(\omega_\mathrm{o}\cdot\omega_\mathrm{h})}{(n\cdot\omega_o)\,(\mathrm{n}\cdot\omega_\mathrm{h})}+\frac{(1-\cos\theta)^5G(\omega_o,\omega_i)(\omega_\mathrm{o}\cdot\omega_\mathrm{h})}{(n\cdot\omega_o)\,(\mathrm{n}\cdot\omega_\mathrm{h})}$$

$$\boxed{\frac{1}{N}\sum_{k=1}^{N}F_0A+B}$$

Donde:

$$A=\frac{(1-(1-\cos\theta)^5)G(\omega_o,\omega_i)(\omega_\mathrm{o}\cdot\omega_\mathrm{h})}{(n\cdot\omega_o)\,(\mathrm{n}\cdot\omega_\mathrm{h})},B=\frac{(1-\cos\theta)^5G(\omega_o,\omega_i)(\omega_\mathrm{o}\cdot\omega_\mathrm{h})}{(n\cdot\omega_o)\,(\mathrm{n}\cdot\omega_\mathrm{h})}$$

Extrayendo $\boldsymbol{F_0}$ del sumatorio tenemos ya las dos partes(**A** y **B**) libres de cualquier implicación por parte del material, ya que el factor de atenuación ***k*** va ligado a la rugosidad (mapeada por el eje de ordenadas de la textura). Sin embargo aún necesitamos disponer de los vectores **V** y **N** para hallar el resto de incógnitas.

El BRDF depende directamente del par de vectores $\boldsymbol{\omega_o}$ y $\boldsymbol{\omega_i}$, pero realmente el valor del BRDF es constante (teniendo en cuenta el mismo material a lo largo de la superficie) para todo vector $\boldsymbol{\omega_o}$**'** donde $\boldsymbol{\cos\theta_o{}'} = \boldsymbol{\cos\theta_0}$. Si mapeamos el valor de coordenadas como el $\boldsymbol{\cos\theta_o{}'}$ podemos hallar un par de vectores **V** y **N** arbitrarios que cumplan:

$$\boldsymbol{\cos\theta_o{}'} = \frac{\boldsymbol{V}\cdot\boldsymbol{N}}{|\boldsymbol{V}|\cdot|\boldsymbol{N}|}$$

Podemos pues definir uno de los dos vectores y hallar el otro en función del primero. Para este caso vamos a definir $N = (0,0,1)$ y hallaremos V en función de **N** y $\boldsymbol{\cos\theta_o{}'}$**.**El vector **V** es necesario para determinar una muestra **L($\boldsymbol{\omega_i}$),** reflejando **V** desde un vector **H** arbitrario (vector bisectriz entre V y L que hallamos usando la secuencia Hammersley). Usando la identidad trigonométrica y el valor N que hemos definido podemos hallar el vector V calculando:

$$V = (\sqrt{1-\boldsymbol{\cos^2\theta}}\,, 0, \boldsymbol{\cos\theta_o})$$

$$Trigonometric\ identity\colon \sin^2\theta + \cos^2\theta = 1 \rightarrow \sin\theta = \sqrt{1-\cos^2\theta}$$

```
void main(){
    float Roughness = v_coord.y;
    float NdotV     = v_coord.x;
    vec3  N = vec3(0.0, 0.0, 1.0);
    vec3  V = vec3(sqrt(1.0 - NdotV * NdotV), 0.0, NdotV);

    float A = 0.0;
    float B = 0.0;
    const int NumSamples = SAMPLES;
    for(int i = 0; i < NumSamples; i++){
        vec2 Xi = Hammersley(i, NumSamples);
        vec3 H  = ImportanceSampleGGX(Xi, Roughness, N);
        vec3 L  = 2.0 * dot(V,H) * H - V; //Reflected vector

        float NdotL = clamp(L.z, 0.0, 1.0);
        float NdotH = clamp(H.z, 0.0, 1.0);
        float VdotH = clamp(dot(V, H), 0.0, 1.0);

        if(NdotL > 0.0){
            float G = G_Smith(Roughness, NdotV, NdotL);
            float G_Vis = G * VdotH / ( NdotH * NdotV );
            float Fc = pow(1.0 - VdotH, 5.0);

            A += (1.0 - Fc) * G_Vis;
            B += Fc * G_Vis;
        }
    }
    vec2 result = vec2(A,B)/ float(NumSamples);
    gl_FragColor = vec4(result, 0.0, 1.0);
}
```

Una vez tenemos las dos texturas LUT’s (look up texture) precalculadas, solo nos falta integrarlas en nuestro shader:

```
vec3 SPLIT_BRDF(vec3 R, Material mat, Vars vars){
    float NdotV = max(dot(vars.N, vars.V),1.e-4);

    //Diffuse Term
    vec3 diffTerm = Li(vars.N, 1.0) * mat.albedo * invPI;

    //Specular Term
    vec3 Li = textureCubeLodEXT(u_map_env_texture, R, mat.roughness *
10.0).xyz;
    vec3 BRDF = texture2D(u_brdf_texture, vec2(NdotV,mat.roughness)).xyz;
    vec3 specTerm = Li * (mat.F0 * BRDF.x + BRDF.y);

    return mat.emission + (diffTerm + specTerm) * mat.AO;
}
```

Los resultados son idénticos a cuando hacíamos las operaciones en el mismo shader, sin embargo ahora funciona a 60 fps (limitado por la tasa de refresco del monitor), algo que lo convierte en un candidato valido para render de PBR en la web.

Al final de este documento, en el anexo añadiré una serie de capturas realizadas con el algoritmo de PBR funcionando

3.2. Shaders adaptados a cada tipo de material

Tras haber cargado el SceneGraph (con los materiales necesarios). Recorremos cada uno de los nodos para ver que material tienen. Ya que el shader resultante depende directamente de los parámetros que hayan en el material disponibles. Nos interesa encontrar un identificador único por cada configuración del material. Cada configuración del material se basa en que parámetros se pasan por valor o por textura. Para ello he incluido dos métodos que ordenan las claves de un objeto, las pasa a un string y luego serializa el string mediante una clave md5.

```
function object2md5(config){
    if( Object.keys(config).length == 0)
        return '';
    var sortedConfig = sort(config);
    config = JSON.stringify(sortedConfig);
    if(typeof config != 'string')
        throw 'object2md5 : something went wrong';
    return md5(config);
}

function sort(object){
    if (typeof object != "object" || object instanceof Array)
        return object;
    var keys = Object.keys(object);
    keys.sort();

    var newObject = {};
    for (var i = 0; i < keys.length; i++){
        newObject[keys[i]] = sort(object[keys[i]])
    }
    return newObject;
}
```

Con este identificador único ya tengo la id para el shader con esa configuración. Para crear el shader, la librería de litegl ofrece un método para compilar el shader que le podemos pasar además del código del Vertex Shader y el Fragment Shader, un objeto con las macros que queramos incluir. Aprovechamos las claves del objeto serializado para activar / desactivar diferentes partes del código.

3.3. Librerías utilizadas

El proyecto hace uso de toda una serie de librerías en las que se ha delegado parte de la responsabilidad dado a su grado de optimización y rendimiento.

Litegl.js

Desarrollada por Javi Agenjo, **Litegl.js** es una librería que proporciona una interfaz minimalista y simplificada de WebGL. Litegl ofrece un conjunto de métodos simplificados para operaciones habituales como puede ser la carga de los arrayBuffers de una mesh, asignación de los programas shader, carga de los uniforms, tratamiento de las texturas…

Se ha hecho uso de la funcionalidad “drawto” de litegl. Este método nos permite hacer un volcado de un buffer en una textura. Dentro del contexto, drawto simplifica el proceso de generación de cubemaps, pudiendo pintar las diferentes caras del cubemap en una sola llamada simplificada.

Rendeer.js

Esta librería, también desarrollada por Javi Agenjo, está enfocada en el uso simple del scenegraph. Proporciona una serie de clases y métodos con las que identificar la escena: mediante nodos, la cámara y el renderer.

He utilizado esta librería para poder definir escenas completas mediante el uso de ficheros JSON, para luego poderlos parsear y convertir los diferentes elementos de la escena en una serie de nodos anidados, con sus materiales, shaders, texturas, etc…

glMath.js

GlMath es una librería que implementa las principales operaciones con vectores y matrices. Es de vital importancia para el proyecto y es una dependencia directa de litegl.

dat.gui.js

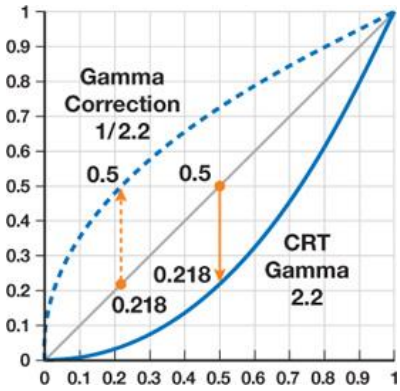
Esta librería permite implementar fácilmente pequeñas interfaces para este tipo de experimentos con los que poder modificar algunos parámetros mediante sliders, checkboxes, multichoice selectors etc

3.4. Problemas encontrados durante el desarrollo

Uno de los problemas principales que he tenido es que el modelo de PBR está diseñado para funcionar con un rango de valores muy ajustado, cuando el modelo no está correctamente implementado no es fácil apreciar el “porqué”. Es por ello que muchas veces los diferentes modelos pueden resultar complejos de implementar porque no muestran el resultado esperado y cualquier imprecisión puede ser la causa.

Un problema que me fue difícil de resolver, no por la complejidad de la solución sino porque tampoco es un tema que se debata es sobre la importancia de mantener lineal el valor que recibimos de entrada.

En el capítulo 24 de Gpu Gems 3 se discute sobre la importancia de mantener aplicar la corrección necesaria a las muestras debido a que estas suelen alojarse en memoria con una cierta corrección gamma determinada. La consecuencia de no realizar esta corrección gamma previa es que los colores se pueden ver distorsionados. El problema es que durante mucho tiempo asociaba este error a un fallo de algoritmo.



Corrección Gamma (Gpu Gems)

Otro GRAN problema que he tenido a la hora de desarrollar este proyecto es la falta de modelos 3D adaptados para PBR que se encuentren libres de derechos. Por suerte existe una herramienta muy interesada llamada Substance Painter 2 que tiene un recogido de materiales PBR con los que podemos pintar las texturas de cualquier modelo 3D.

A la hora de desarrollar el algoritmo en distintos dispositivos, hay que considerar que WebGL es algo muy reciente y las diferentes implementaciones entre dispositivos pueden diferir. Algunos dispositivos no aceptaban ciertas extensiones o simplemente no aceptaban las uniforms pasadas. Algunos de estos dispositivos no eran tan viejos (LGg2 de 2014) no conseguí que funcionase, mientras que en otros (IPad Air, Samsung Galaxy Tab) desactivando la extensión de LOD y modificando ciertas operaciones terminó funcionando.

3.5. Conclusión

La técnica de PBR puede ser aproximada de múltiples maneras, la implementada en este proyecto es solo una de ellas. Como ya compartieron con nosotros los alumnos de la universidad de Granada, el modelo Cook-Torrance es uno de entre 26 diferentes posibles que llegaron a catalogar cada uno con sus fuertes y sus debilidades.

Mediante el uso de texturas precalculadas se ha podido llegar a una solución plausible y que puede ser visualizada en unos muy interesantes 60 fps. Si bien estoy contento con el resultado del proyecto, existen muchos componentes que no he podido y me hubiese gustado poder implementar: refracción de la luz con el BTDF, reflexión interna del material con el BSSRDF, diferentes capas de materiales para combinar un material base y por ejemplo una capa de “polvo”.

WebGL es una tecnología que aún parece muy primeriza que aún no es estándar en todos los dispositivos e incluso entre navegadores de escritorio. Con las primeras versiones de desarrollo de WebGL 2.0 estamos viendo un motor mejorado, evolucionado, más próximo a OpenGL 3.1 ofreciendo gráficos que no difieren tanto de lo que se puede apreciar en aplicaciones gráficas o videojuegos de pocos años atrás.

WebGL, los gráficos tradicionales de OpenGL y JavaScript, un lenguaje de scripting con miles de ejemplos abiertos. La fusión de tecnologías ofrece una alternativa simple y ágil para el desarrollo de aplicaciones 3D.

Bibliografía

A Reflectance Model for Computer Graphics. (s.f.). Obtenido de Berkeley Institute: http://inst.eecs.berkeley.edu/~cs283/sp13/lectures/cookpaper.pdf

Alamia, M. (2013). *Physically Based Rendering*. Obtenido de CodingLabs.net: http://www.codinglabs.net/article_physically_based_rendering.aspx

Dammertz, H. (2012). *Hammersley Points on the Hemisphere.* Obtenido de holger.dammertz: http://holger.dammertz.org/stuff/notes_HammersleyOnHemisphere.html

dat.gui. (s.f.). Obtenido de GitHub: https://github.com/dataarts/dat.gui

doFactory.com. (2016). *JavaScript Design Patterns*. Obtenido de doFactory.com: http://www.dofactory.com/javascript/design-patterns

Humphreys, G., & Pharr, M. (s.f.). From Theory to Implementation. En *Physically Based Rendering.*

Jang, J. (2010). *Global illumination for PIXAR movie production*. Obtenido de Slideshader.net: http://www.slideshare.net/JaehyunJang2/individual-study2014-jaehyunjang

Jotatsu. (2006). *Primitivas en OpenGL*. Obtenido de Black-Byte.com: http://black-byte.com/tutorial/primitivas-en-opengl/

Khronos. (2009). *OpenGL ES Common Profile Specification 2.0.24 .* Obtenido de Khronos.org: https://www.khronos.org/registry/gles/specs/2.0/es_cm_spec_2.0.24.pdf

Maximum-Dev. (2014). *PBR Tutorial Series* . Obtenido de UnrealEngine.com: https://forums.unrealengine.com/showthread.php?52529-PBR-Tutorial-Series

McDermott, W. (s.f.). *The Comprehensive PBR Guide by Allegorithmic - vol. 1.* Obtenido de Allegorithmic.com: https://www.allegorithmic.com/system/files/software/download/build/PBR_Guide_Vol.1.pdf

McDermott, W. (s.f.). *the Comprehensive PBR Guide by Allegorithmic - vol. 2.* Obtenido de Allegorithmic.com: https://www.allegorithmic.com/system/files/software/download/build/PBR_Guide_Vol.2.pdf

Montes, R., & Ureña, C. (s.f.). *An Overview of BRDF Models.* Obtenido de Universidad de Granada: http://digibug.ugr.es/bitstream/10481/19751/1/rmontes_LSI-2012-001TR.pdf

Nvidia CGDC. (2014). *Next generation, TEGRA K1. (nvidia CGDC 2014)*. Obtenido de developer.nvidia.com: http://twvideo01.ubm-us.net/o1/vault/gdcchina15/slides/Li_Wenlei_PBR%20Implications_CN.pdf

Oren, M., & Nayar, S. (s.f.). *Generalization of Lambert's Reflectance Model.* Obtenido de Columbia Edu: http://www1.cs.columbia.edu/CAVE/publications/pdfs/Oren_SIGGRAPH94.pdf

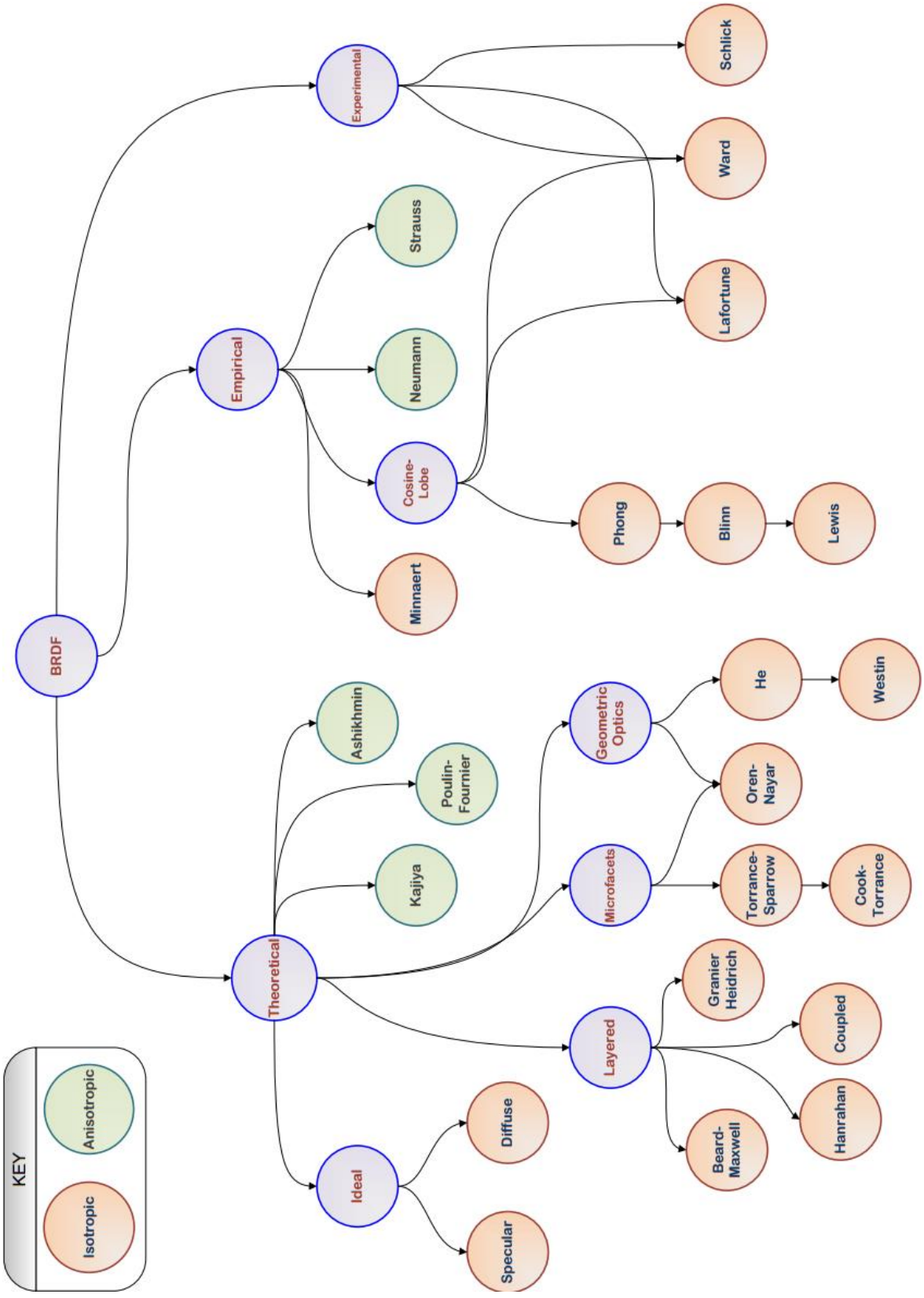
Ouwerkerk, J. (s.f.). *van Ouwerkerk's rewrite of the Oren Nayar BRDF*. Obtenido de Shaderjvo: http://shaderjvo.blogspot.com.es/2011/08/van-ouwerkerks-rewrite-of-oren-nayar.html

Oxeyeball. (2007). *Modelo de iluminacion para superficies rugosas Oren/Nayar*. Obtenido de Oxeyeball: https://translate.google.es/translate?hl=es&sl=zh-

CN&u=http://oxeyeball.blog.163.com/blog/static/33365958200781103044547/&prev=search
Rusell, J. (2015). *Basic Theory of Physically-Based Rendering*. Obtenido de Marmoset.co: http://www.marmoset.co/toolbag/learn/pbr-theory
thealmightyguru.com. (2016). *Galaga*. Obtenido de thealmightyguru.com: http://www.thealmightyguru.com/Games/Hacking/Wiki/index.php?title=Galaga
Thomas, M. (2014). *Physically Based Camera Rendering*. Obtenido de extremeistan: https://extremeistan.wordpress.com/2014/09/24/physically-based-camera-rendering/
tiansijie. (2014). *WebGLpbr*. Obtenido de Github.com: https://github.com/tiansijie/WebGLpbr/blob/master/index.html
Trent. (2015). *Physically Based Shading and Image Based Lighting*. Obtenido de Trent Reed: http://www.trentreed.net/blog/physically-based-shading-and-image-based-lighting/
Wikipedia. (2016). *Albedo*. Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Albedo
Wikipedia. (2016). *Bidirectional scattering distribution function*. Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Bidirectional_scattering_distribution_function
Wikipedia. (2016). *Blinn-Phong*. Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Blinn-Phong_shading_model
Wikipedia. (2016). *Faraday Efect*. Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Faraday_effect
Wikipedia. (2016). *Fresnel Equations*. Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Fresnel_equations
Wikipedia. (2016). *Light*. Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Light
Wikipedia. (2016). *Open.gl*. Obtenido de https://open.gl/depthstencils
Wikipedia. (2016). *WebGL*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/WebGL
Wilson, J. ". (2015). *Tutorial: Physically Based Rendering, And You Can Too!* Obtenido de Marmoset.co: http://www.marmoset.co/toolbag/learn/pbr-practice
Wilson, J. “. (2015). *Tutorial: PBR Texture Conversion*. Obtenido de Marmoset.co: http://www.marmoset.co/toolbag/learn/pbr-conversion

Anexo

- **Diagrama sobre los diferentes modelos para el BRDF: (UdG, *"An Overview of BRDF models"*, Montes & Ureña):**



- **Tabla sobre los diferentes modelos para el BRDF:**
  (UdG, *"An Overview of BRDF models"*, Montes & Ureña):

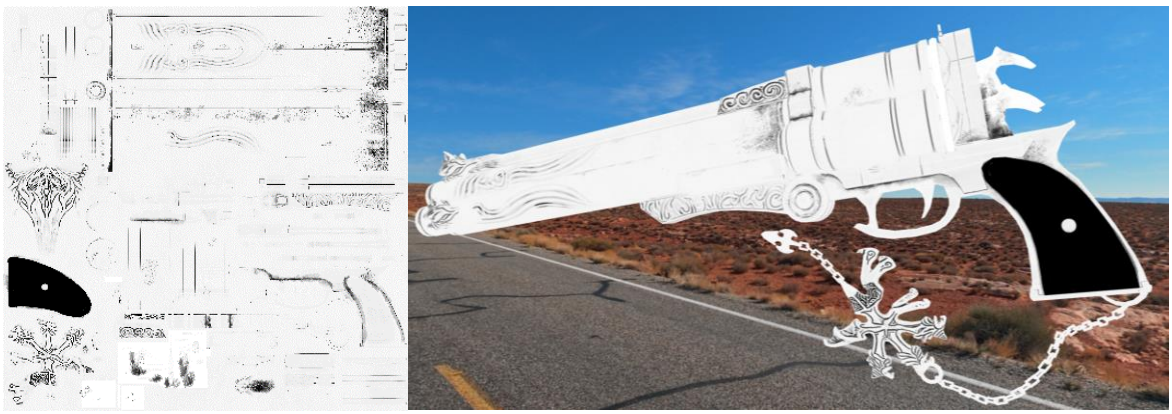
Models	*P*hysical	*P*lausible	*F*resnel Eq.	*A*nisotrotopic	*S*ampling	*Rel.Cost (cycles)*	*M*aterial *T*ype
*I*deal Specular	★	★	▼	▼	★	x	perfect specular
*I*deal Diffuse	★	★	▼	▼	★	x	perfect diffuse
*M*innaert	▼	···	▼	▼	▼	$5.35x$	Moon surf.
*T*orrance-*S*parrow	★	▼	★	★	▼	···	rough surf.
*B*eard-*M*axwell	★	▼	★	▼	▼	$397x$	painted surf.
*B*linn-*P*hong	▼	▼	▼	▼	★	$9.18x$	rough surf.
*C*ook-*T*orrance	★	★	★	▼	▼	$16.9x$	metal,plastic
*K*ajiya	★	▼	★	★	▼	···	metal,plastic
*P*oulin-*F*ournier	★	▼	▼	★	▼	$67x$	clothes
*S*trauss	▼	···	★	▼	▼	$14.88x$	metal,plastic
*H*e et al.	★	★	★	▼	▼	$120x$	metal
*W*ard	▼	▼	▼	★	★	$7.9x$	wood
*W*estin	★	···	★	★	▼	···	metal
*L*ewis	▼	★	▼	▼	★	$10.73x$	mats
*S*chlick	▼	★	★	★	▼	$26.95x$	heterogeneous
*H*anrahan	★	···	★	▼	▼	···	human skin
*O*ren-*N*ayar	★	★	▼	▼	★	$10.98x$	matte, dirty.
*N*eumann	▼	★	▼	★	★	···	metal,plastic
*L*afortune	▼	★	▼	★	★	$5.43x$	rough surf.
*C*oupled	★	★	★	▼	★	$17.65x$	polished surf.
*A*shikhmin-*S*hirley	★	▼	★	★	★	$79.6x$	polished surf.
*G*ranier-*H*eidrich	★	···	★	▼	▼	···	old-dirty metal

Table 1: *Brief summary of the properties exhibited by the reviewed BRDFs. Legend: (★) if the BRDF has this property; (▼) if the BRDF does not; (···) unknown value.*

- **Diferentes tipos de texturas y como se reflejan en el modelo:**

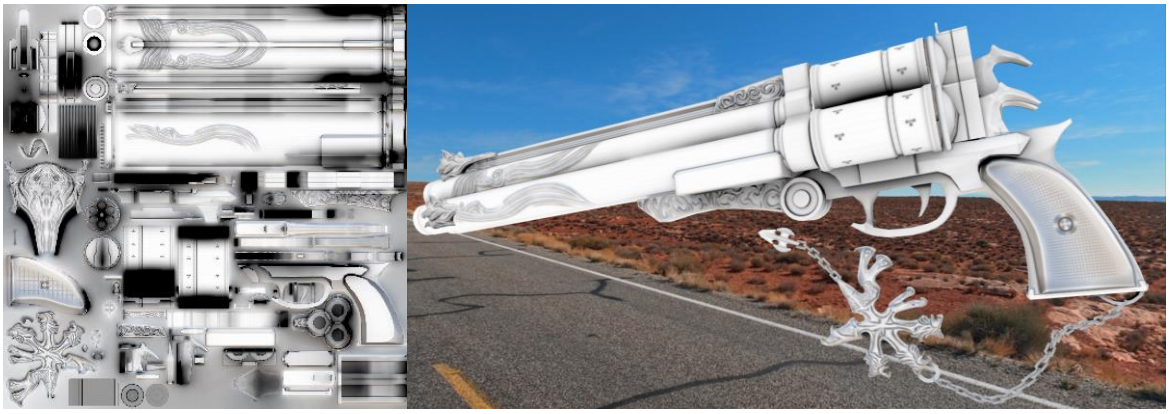
El mapa albedo define el color base del modelo.

El mapa de rugosidad define la irregularidad de la micro-superficie. Define que tan propensa es la luz a dispersarse cuando rebota sobre la superficie del material.



El mapa metálico define que partes del modelo son metálicas (material conductor) o no-metálicas (material dieléctrico).

Adicionalmente se utilizan en algunos modelos mapas adicionales que dan mayor detalle y un mejor acabado pero que no son imprescindibles: el mapa de normales, oclusión ambiental, altura, cavity etc…:

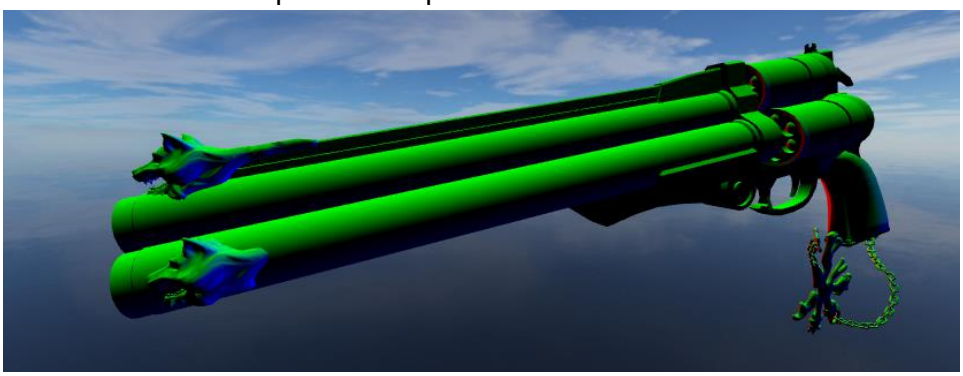


El mapa de oclusión ambiental representa las zonas del modelo donde la luz ambiental no alcanza debido a que el propio modelo la bloquea.

La siguiente textura caso es un tanto especial, el mapa de normales. Habitualmente cuando se generan las texturas, están pensadas para ser utilizadas en un único modelo. Al ir a buscar el texel obtenemos el nuevo valor directamente. Esto se le denomina espacio de objeto.
Sin embargo existe un segundo método de alojar las normales en la textura: el espacio tangencial. Las normales se almacenan teniendo en cuenta un sistema de coordenadas basado en la dirección de la normal y la tangente a la misma.El objetivo de este método es reaprovechar en diferentes modelos una misma textura. Si bien esta textura en concreto podría no tener mucho sentido en otro modelo distinto, una textura de ruido “perlin-noise” puede ser aplicada universalmente con este método.

El mapa de normales permite definir la rugosidad del material a nivel macroscópico

Este es el modelo antes de aplicar el mapa de normales:



- **Detalles de renderizado:**

Reflejos del entorno en los martillos dorados de la pistola,

Detalles de la rugosidad a nivel microscópico (roughness del material) y macroscópico (bump map)

El color del modelo se funde con la iluminación del entorno

Esfera espejo con grietas

- **Algunos enlaces de interés:**

Manifiesto WebGL:
https://www.khronos.org/registry/gles/specs/1.1/es_cm_spec_1.1.12.pdf

MERL BRDF Database:
http://www.merl.com/brdf/

Real-Time Polygonal-Light Shading with Linearly Transformed Cosines
https://eheitzresearch.wordpress.com/415-2/

Fresnel's Equations:

http://hyperphysics.phy-astr.gsu.edu/hbase/phyopt/freseq.html