



UNIVERSIDAD
DE GRANADA

Los fundamentos de Ray Tracing

Doble grado en ingeniería informática y matemáticas

asmilex.github.io/Raytracing

Presentado por: Andrés Millán Muñoz,

Tutorizado por: Carlos Ureña Almagro, María del Carmen Segovia García

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Facultad de Ciencias

April 23, 2022



Contents

Abstract	1
A brief overview	2
Dedicatoria	4
1. Introducción	5
1.1. Nota histórica	5
1.2. ¿Qué es ray tracing?	6
1.3. Objetivos del trabajo	8
1.4. Técnicas empleadas para la resolución	8
1.5. Principales fuentes consultadas	9
2. Las bases	11
2.1. Eligiendo direcciones	13
2.2. Intersecciones rayo - objeto	14
2.2.1. Superficies implícitas	15
2.2.2. Superficies paramétricas	16
2.2.3. Intersecciones con esferas	17
2.2.4. Intersecciones con triángulos	20
3. Transporte de luz	23
3.1. Unidades radiométricas básicas	23
3.1.1. Potencia	24
3.1.2. Irradiancia	24
3.1.3. Ángulos sólidos	26

3.1.4.	Intensidad radiante	30
3.1.5.	Radiancia	31
3.2.	Fotometría y radiometría	34
3.3.	Integrales radiométricas	34
3.3.1.	Una nueva expresión de la irradiancia y el flujo	34
3.3.2.	Integrando sobre área	35
3.4.	Dispersión de luz: las familias de funciones de distribución bidireccionales	36
3.4.1.	La función de distribución de reflectancia bidireccional (BRDF) .	37
3.4.2.	La función de distribución de transmitancia bidireccional (BTDF)	38
3.4.3.	Juntando la BRDF y la BTDF en La función de distribución de dispersión bidireccional	39
3.4.4.	Reflectancia hemisférica	40
3.4.5.	Reflejos	40
3.5.	La rendering equation	41
3.6.	Materiales	43
3.6.1.	Ecuaciones de fresnel, ley de Snell	44
4.	Integración de Monte Carlo	45
4.1.	Repaso de probabilidad	45
4.1.1.	Variables aleatorias discretas	46
4.1.2.	Variables aleatorias continuas	48
4.1.3.	Esperanza y varianza de una variable aleatoria	49
4.1.4.	Estimadores	52
4.2.	El estimador de Monte Carlo	53
4.3.	Escogiendo puntos aleatorios	56
4.3.1.	Método de la transformada inversa	57
4.3.2.	Método del rechazo	58
4.4.	Importance sampling	59
4.5.	Multiple importance sampling	60
5.	¡Construyamos un path tracer!	62
5.1.	Requisitos de <i>real time ray tracing</i>	62
5.1.1.	Arquitecturas de gráficas	63

5.1.2.	Frameworks y API de ray tracing en tiempo real	64
5.2.	Setup del proyecto	65
5.3.	Compilación	65
5.4.	Estructuras de aceleración	66
5.4.1.	Bottom-Level Acceleration Structure (BLAS)	67
5.4.2.	Top-Level Acceleration Structure (TLAS)	68
5.5.	La ray tracing pipeline	70
5.5.1.	Descriptores y conceptos básicos	70
5.5.2.	La Shader binding table	70
5.5.3.	Tipos de shaders	73
5.5.4.	Traspaso de información entre shaders	74
5.5.5.	Creación de la ray tracing pipeline	74
5.6.	Asmiray	75
5.7.	Transporte de luz en la práctica	75
5.7.1.	Materiales y objetos	75
5.8.	Fuentes de luz	76
5.8.1.	Point lights + spotlights	76
5.8.2.	Fuentes de área	77
6.	Análisis de rendimiento	79
7.	El presente y futuro de RT	80
7.1.	Denoising	80
7.2.	Filtering	80
7.3.	Offline renderers	81
7.4.	Importance Resampling	81
7.5.	ReSTIR	81
7.6.	Low discrepancy sampling	81
7.7.	La industria del videojuego	81
7.7.1.	Ray tracing híbrido	81
7.7.2.	Productos comerciales	81
7.7.3.	Unreal Engine 5	82
7.7.4.	La última generación de consolas	82

7.8. Posibles mejoras del trabajo	82
A. Metodología de trabajo	83
A.1. Influencias	83
A.2. Ciclos de desarrollo	84
A.3. Presupuesto	84
A.4. Arquitectura del software	85
A.5. Diseño	85
A.5.1. Bases del diseño	85
A.5.2. Tipografías	86
A.5.3. Paleta de colores	86
A.6. Flujo de trabajo y herramientas	88
A.6.1. Pandoc	88
A.6.2. Figma	89
A.6.3. Otros programas	90
A.7. Github	90
A.7.1. Integración continua con Github Actions y Github Pages	91
A.7.2. Issues y Github Projects	93
A.7.3. Estilo de commits	96
B. Glosario de términos	97
B.1. Notación	97
B.2. Radiometría	98
Bibliografía	100

Abstract

Este trabajo explorará las técnicas modernas de informática gráfica físicamente fieles basadas en *ray tracing* en tiempo real. Para ello, se usarán métodos de integración de Monte Carlo dado que disminuyen el tiempo necesario de cómputo.

Para conseguirlo, se ha diseñado un software basado en la interfaz de programación de aplicaciones gráficas Vulkan, usando como base un entorno de desarrollo de Nvidia conocido como nvpro-samples. El software implementa un motor gráfico basado en *path tracing*. Este motor será capaz de muestrear fuentes de iluminación de forma directa, lo que se conoce como *next-event estimation*. Para disminuir el tiempo de cómputo y hacerlo viable en tiempo real, se usarán técnicas de Monte Carlo para integrar radiancia. Se explorarán cómo afectan los diferentes métodos al ruido final de la imagen.

Este motor se comparará con una implementación puramente en CPU basada en el software desarrollado en los libros de (Shirley 2020a) “Ray Tracing in One Weekend series”. Se han estudiado las diferencias de tiempo entre una implementación y otra, sus ventajas y desventajas y el ruido de las imágenes producidas.

Palabras clave: raytracing, ray tracing, Monte Carlo, Monte Carlo integration, radiometry, path tracing, Vulkan.

A brief overview

TODO

Keywords: raytracing, ray tracing, Monte Carlo, Monte Carlo integration, radiometry, path tracing, Vulkan.

Dedicatoria

¡Parece que has llegado un poco pronto! Si lo has hecho voluntariamente, ¡muchas gracias! Este proyecto debería estar finalizado en verano de 2022. Mientras tanto, actualizaré poco a poco el contenido. Si quieres ir comprobando los progresos, puedes visitar [Asmilex/Ray-tracing](#) en Github para ver el estado del desarrollo.

Aun así, hay mucha gente que me ha ayudado a sacar este proyecto hacia delante.

Gracias, en primer lugar, a mi familia por permitirme acabar la carrera. A Cristina, Jorge, Jose OC, Lucas, Mari, Marina y Paula, Sergio por ayudarme con el contenido, feedback del desarrollo y guía de diseño.

1. Introducción

Este trabajo puede visualizarse en la web asmilex.github.io/Raytracing o en el PDF disponible en el repositorio del trabajo [Asmilex/Raytracing](#).

La página web contiene la versión más actualizada, además de recursos adicionales como vídeos.

1.1. Nota histórica

Ser capaces de capturar un momento.

Desde siempre, este ha sido uno de los sueños de la humanidad. La capacidad de retener lo que ven nuestros ojos comenzó con simples pinturas ruprestres. Con el tiempo, el arte evolucionó, así como la capacidad de retratar nuestra percepción con mayor fidelidad.

A inicios del siglo XVIII, se caputaron las primeras imágenes con una cámara gracias a Nicéphore Niépce. Sería una imagen primitiva, claro; pero era funcional. Gracias a la compañía Kodak, la fotografía se extendió al consumidor rápidamente sobre 1890. Más tarde llegaría la fotografía digital, la cual simplificaría muchos de los problemas de las cámaras tradicionales.

Hablando de digital. Los ordenadores personales modernos nacieron unos años más tarde. Los usuarios eran capaces de mostrar imágenes en pantalla, que cambiaban bajo demanda. Y, entonces, nos hicimos una pregunta...

¿Podríamos **simular la vida real** para mostrarla en pantalla?

Como era de esperar, esto es complicado de lograr. Para conseguirlo, hemos necesitado crear abstracciones de conceptos que nos resultan naturales, como objetos, luces y seres

vivos. “Cosas” que un ordenador no entiende, y sin embargo, para nosotros *funcionan*.

Así, nació la geometría, los puntos de luces, texturas, sombreados, y otros elementos de un escenario digital. Pero, por muchas abstracciones elegantes que tengamos, no nos basta. Necesitamos visualizarlas. Y como podemos imaginarnos, esto es un proceso costoso.

La **rasterización** es el proceso mediante el cual estos objetos tridimensionales se transforman en bidimensionales. Proyectando acordemente el entorno a una cámara, conseguimos colorear un pixel, de forma que represente lo que se ve en ese mundo.

TODO insertar imagen rasterización.

NOTE ¿quizás debería extender un poco más esta parte? Parece que se queda algo coja la explicación.

Aunque esta técnica es bastante eficiente en términos de computación y ha evolucionado mucho, rápidamente saturamos sus posibilidades. Conceptos como *shadow maps*, *baked lightning*, o *reflection cubemaps* intentan solventar lo que no es posible con rasterización: preguntarnos *qué es lo que se encuentra alrededor nuestra*.

En parte, nos olvidamos de la intuitiva realidad, para centrarnos en aquello computacionalmente viable.

Y, entonces, en 1960 el trazado de rayos con una simple idea intuitiva.

1.2. ¿Qué es ray tracing?

En resumidas cuentas, *ray tracing* (o trazado de rayos en español), se basa en disparar fotones en forma de rayo desde nuestra cámara digital y hacerlos rebotar en la escena.

De esta forma, simulamos cómo se comporta la luz. Al impactar en un objeto, sufre un cambio en su trayectoria. Este cambio origina nuevos rayos, que vuelven a dispersarse por la escena. Estos nuevos rayos dependerán de las propiedades del objeto con el que hayan impactado. Con el tiempo necesario, lo que veremos desde nuestra cámara será una representación fotorealista de lo que habita en ese universo.

Esta técnica, tan estúpidamente intuitiva, se ha hecho famosa por su simpleza y su elegancia. *Pues claro que la respuesta a “¿Cómo simulamos fielmente una imagen en un ordenador?” es “Representando la luz de forma realista”.*

Aunque, quizás intuitiva no sea la palabra. Podemos llamarla *natural*, eso sí. A fin de cuentas, fue a partir del siglo XVIII cuando empezamos a entender que podíamos capturar la luz. Nuestros antepasados tenían teorías, pero no podían explicar por qué *veíamos* el mundo.

Ahora sí que sabemos cómo funciona. Entendiendo el por qué lo hace nos permitirá programarlo. Y, resulta que funciona impresionantemente bien.

Atrás se quedan los *hacks* necesarios para rasterización. Los cubemaps no son esenciales para los reflejos, y no necesitamos cámaras virtuales para calcular sombras. Ray tracing permite simular fácilmente efectos como reflejos, refracción, desenfoque de movimiento, aberración cromática... Incluso fenómenos físicos propios de las partículas y las ondas.

Espera. Si tan bueno es, ¿por qué no lo usamos en todos lados?

Por desgracia, el elefante en la sala es el rendimiento. Como era de esperar, disparar rayos a diestro y siniestro es costoso. **Muy costoso.**

A diferencia del universo, nosotros no nos podemos permitir el lujo de usar fotones de tamaño infinitesimal y dispersiones casi infinitas. Nos pasaríamos una eternidad esperando. Y para ver una imagen en nuestra pantalla necesitaremos estar vivos, claro.

Debemos evitar la fuerza bruta. Dado que la idea es tan elegante, la respuesta no está en el “*qué*”, sino en el “*cómo*”. Si **disparamos y dispersamos rayos con cabeza** seremos capaces de obtener lo que buscamos en un tiempo razonable.

Hace unos años, al hablar de tiempo razonable, nos referiríamos a horas. Quizás días. Producir un *frame* podría suponer una cantidad de tiempo impensable para un ordenador de consumidor. Hoy en día también ocurre esto, claro está. Pero la tecnología evoluciona.

Podemos bajarlo a milisegundos.

Hemos entrado en la era del **real time ray tracing**.

1.3. Objetivos del trabajo

Los objetivos del trabajo iniciales son los siguientes:

- Análisis de los algoritmos modernos de visualización en 3D basados en métodos de Monte Carlo.
- Revisión de las técnicas de Monte Carlo, examinando puntos fuertes y débiles de cada una. Se busca minimizar el error en la reconstrucción de la imagen y minimizar el tiempo de ejecución.
- Implementación de dichos algoritmos en hardware gráfico moderno (GPUs) específicamente diseñado para aceleración de ray tracing.
- Diseño e implementación de un software de síntesis de imágenes realistas por path tracing y muestreo directo de fuentes de luz por GPU.
- Análisis del rendimiento del motor con respecto al tiempo de ejecución y calidad de imagen.
- Comparación del motor desarrollado con una implementación por CPU.
- Investigación de las técnicas modernas y sobre el futuro del área.

TODO: determinar si lo siguiente es cierto.

Afortunadamente, **se ha conseguido realizar exitosamente cada uno de los objetivos**. Esta memoria cubrirá todo el trabajo que ha sido necesario realizar para lograrlo.

1.4. Técnicas empleadas para la resolución

TODO: echarle un ojo a esto cuando termine el trabajo.

Además del antedicho algoritmo ray tracing y su versión más pura path tracing, se han empleado técnicas de Monte Carlo para calcular la luz resultante de un punto.

En particular, con respecto a la **matemática** empleada, estudiaremos diferentes formas de generar números aleatorios mediante distribuciones particulares, (*multiple*) *importance sampling*, next event estimation, ...

En un área híbrida se encuentra la **radiometría**. Dado que estamos tratando con transporte de luz, será esencial introducir los conceptos más importantes de la radiometría. Trataremos con algunos términos como irradiancia, ángulos sólidos, radiancia, funciones de distribuciones de reflectancia y transmitancia bidireccionales, etc.

Finalmente, la parte **informática** usará en la API gráfica Vulkan junto a un framework de Nvidia para acelerar la adopción de ray tracing en KHR. Veremos qué se necesita para implementar ray tracing en tiempo real, lo que nos llevará aprender sobre programación en Vulkan, las estructuras de aceleración de nivel alto y bajo (TLAS y BLAS), la Shader Binding Table, comunicación con CPU y GPU, etc.

Todo este programa estará alojado en Github. En el **apéndice**, aprenderemos cómo se ha usado la plataforma para integrar la documentación, el código fuente y los ciclos de desarrollo.

Como podemos ver, esta área relaciona íntimamente la matemática y la informática, con un poco de física de por medio.

1.5. Principales fuentes consultadas

Esencialmente, este trabajo ha sido posible gracias a los siguientes recursos:

- La serie de libros de *Ray Tracing* de Peter Shirley, conocidos como “Ray tracing In One Weekend Series” (Shirley 2020a), (Shirley 2020b), (Shirley 2020c). El motor desarrollado en estos libros es el que se utilizará para la comparación.
- Physically Based Rendering: From Theory to Implementation (3rd ed.) (Pharr, Jakob, and Humphreys 2016), considerado como el santo grial de la informática gráfica moderna.
- Ray Tracing Gems I y II (Haines and Akenine-Möller 2019), (Adam Marrs and Wald 2021), una colección de papers esenciales sobre ray tracing publicada por Nvidia.

Referencias

(Wikipedia: history of photography n.d.), (Wikipedia: Kodak n.d.), (Wikipedia: Computer n.d.), (Wikipedia: rendering (computer graphics) n.d.), (Caulfield n.d.), (tracing n.d.), (“Rendering” n.d.), (Haines and Akenine-Möller 2019)

- RT IOW series

2. Las bases

Empecemos por definir lo que es un rayo.

Un rayo es una función $P(t) = O + tD$, donde O es el origen, D la dirección, y $t \in \mathbb{R}$. Podemos considerarlo una interpolación entre dos puntos en el espacio, donde t controla la posición en la que nos encontramos.

Por ejemplo, si $t = 0$, obtendremos el origen. Si $t = 1$, obtendremos el punto correspondiente a la dirección. Usando valores negativos vamos *hacia atrás*.

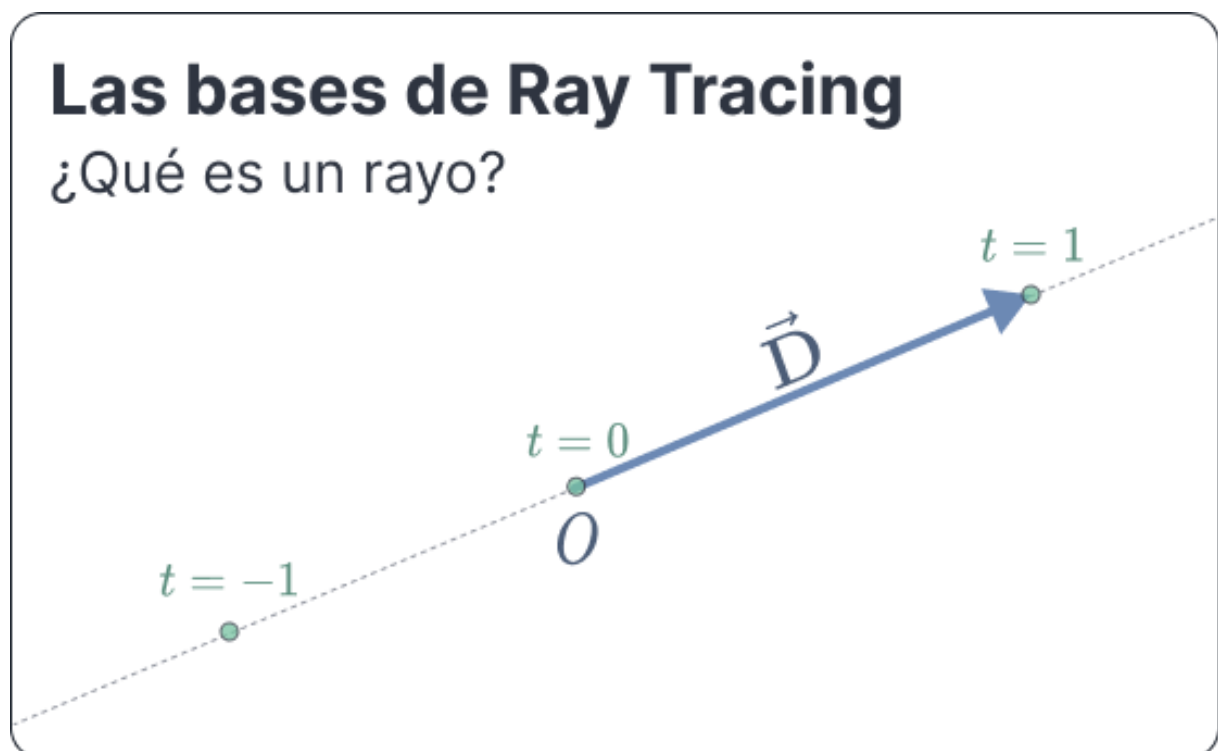


Figure 2.1.: El parámetro t nos permite controlar los puntos del rayo

Dado que estos puntos estarán generalmente en \mathbb{R}^3 , podemos escribirlo como

$$P(t) = (O_x, O_y, O_z) + t(D_x, D_y, D_z)$$

Estos rayos los *dispararemos* a través de una cámara virtual, que estará enfocando a la escena. De esta forma, los haremos rebotar con los objetos que se encuentren en el camino del rayo. A este proceso lo llamaremos **ray casting**.

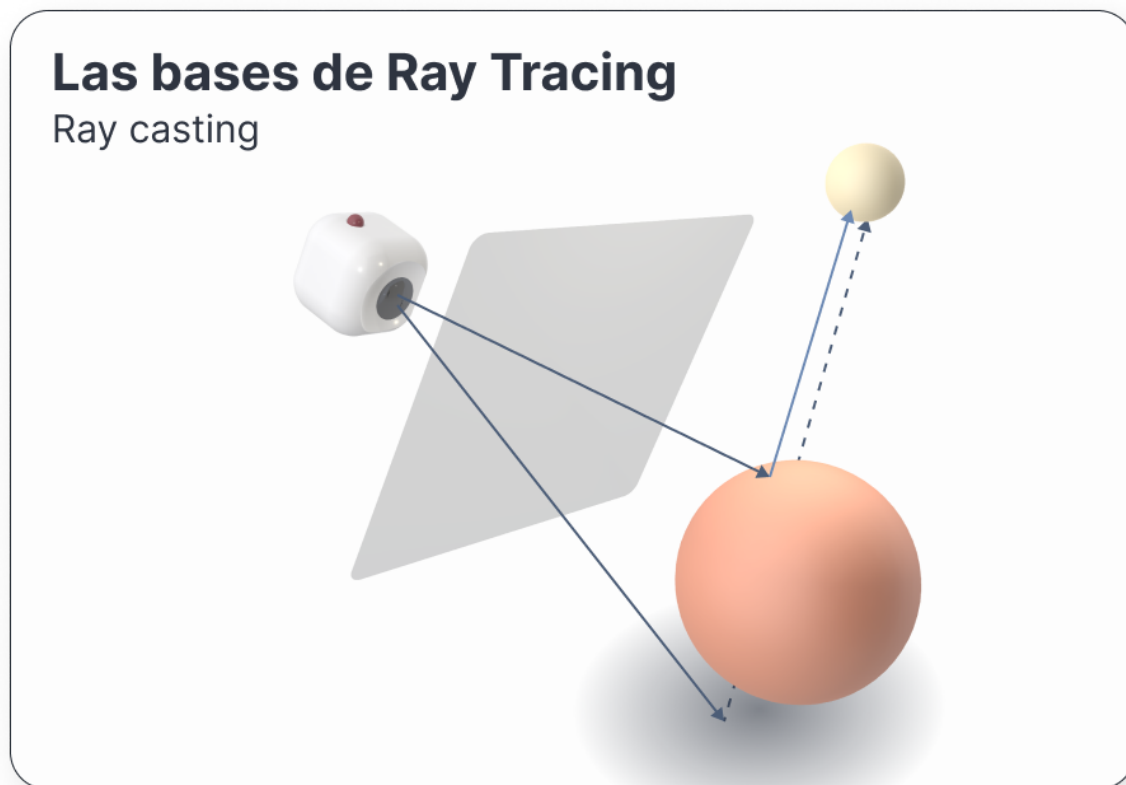


Figure 2.2.: Diagrama de ray casting

Generalmente, nos quedaremos con el primer objeto que nos encontremos en su camino. Aunque, a veces, nos interesará saber todos con los que se encuentre.

Cuando un rayo impacta con un objeto, adquirirá parte de las propiedades lumínicas del punto de impacto. Por ejemplo, cuánta luz proporciona la lámpara que tiene encima la esfera de la figura anterior.

Una vez recojamos la información que nos interese, aplicaremos otro raycast desde el nuevo punto de impacto, escogiendo una nueva dirección determinada. Esta dirección dependerá del tipo de material del objeto. Y, de hecho, algunos serán capaces de invocar varios rayos.

Por ejemplo, los espejos reflejan la luz casi de forma perfecta; mientras que otros elementos como el agua o el cristal reflejan y refractan luz, así que necesitaremos generar dos nuevos raycast.

Usando suficientes rayos obtendremos la imagen de la escena. A este proceso de **ray casting recursivo** es lo que se conoce como ray tracing.

Como este proceso puede continuar indefinidamente, tendremos que controlar la profundidad de la recursión. A mayor profundidad, mayor calidad de imagen; pero también, mayor tiempo de ejecución.

2.1. Eligiendo direcciones

Una de las partes más importantes de ray tracing, y a la que quizás dedicaremos más tiempo, es a la elección de la dirección.

Hay varios factores que entran en juego a la hora de decidir qué hacemos cuando impactamos con un nuevo objeto:

1. **¿Cómo es la superficie del material?** A mayor rugosidad, mayor aleatoriedad en la dirección. Por ejemplo, no es lo mismo el asfalto de una carretera que una lámina de aluminio impecable.
2. **¿Cómo de fiel es nuestra geometría?**
3. **¿Dónde se encuentran las luces en la escena?** Dependiendo de la posición, nos interesará muestrear la luz con mayor influencia.

Estas cuestiones las exploraremos a fondo en las siguientes secciones.

2.2. Intersecciones rayo - objeto

Como dijimos al principio del capítulo, representaremos un rayo como

$$\begin{aligned} P(t) &= (O_x, O_y, O_z) + t(D_x, D_y, D_z) = \\ &= (O_x + tD_x, O_y + tD_y, O_z + tD_z) \end{aligned}$$

Por ejemplo, tomando $O = (1, 3, 2)$, $D = (1, 2, 1)$:

- Para $t = 0$, $P(t) = (1, 3, 2)$.
- Para $t = 1$, $P(t) = (1, 3, 2) + (1, 2, 1) = (2, 5, 3)$.

Nos resultará especialmente útil limitar los valores que puede tomar t . Restringiremos los posibles puntos del dominio de forma que $t \in [t_{min}, t_{max})$, con $t_{min} < t_{max}$. En general, nos interesará separarnos de las superficies un pequeño pero no despreciable ε para evitar errores de redondeo.

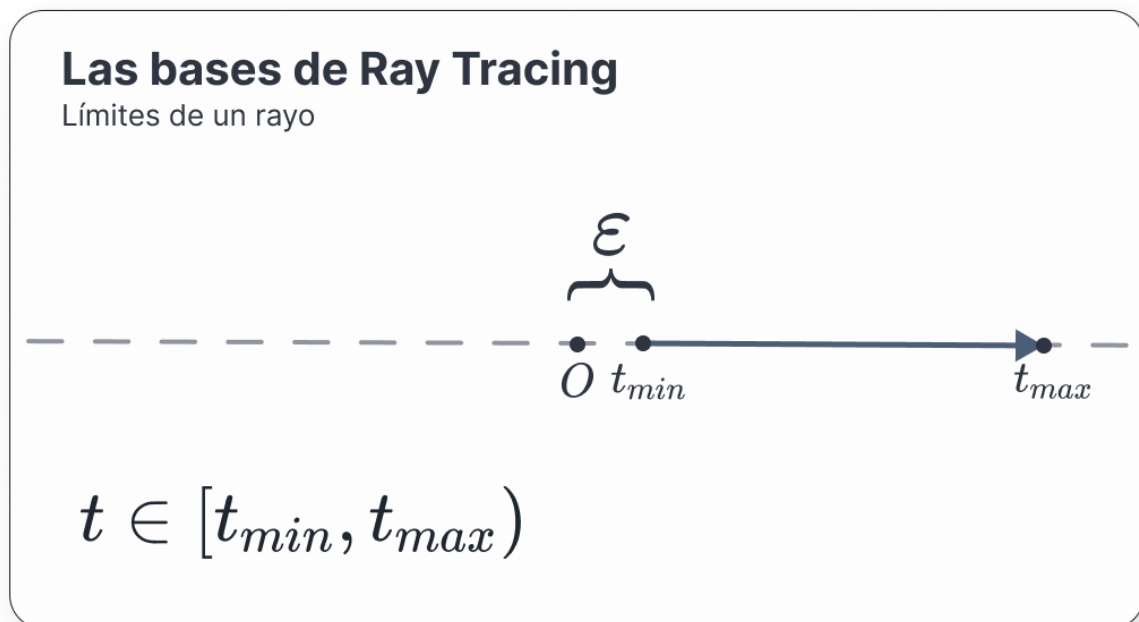


Figure 2.3.: Separarnos un poquito del origen evitará errores de coma flotante

Una de las principales cuestiones que debemos hacernos es saber cuándo un rayo impacta con una superficie. Lo definiremos analíticamente.

2.2.1. Superficies implícitas

Generalmente, cuando hablemos de superficies, nos referiremos superficies diferenciables ([Wikipedia: Differential geometry of surfaces n.d.](#)), pues nos interesará conocer el vector normal en cada punto.

Una superficie implícita es una superficie en un espacio euclidiano definida como

$$F(x, y, z) = 0$$

Esta ecuación implícita define una serie de puntos del espacio \mathbb{R}^3 que se encuentran en la superficie.

Por ejemplo, la esfera se define como $x^2 + y^2 + z^2 - 1 = 0$.

Consideremos una superficie S y un punto regular de ella P ; es decir, un punto tal que el gradiente de F en P no es 0. Se define el vector normal \mathbf{n} a la superficie en ese punto como

$$\mathbf{n} = \nabla F(P) = \left(\frac{\partial F(P)}{\partial x}, \frac{\partial F(P)}{\partial y}, \frac{\partial F(P)}{\partial z} \right)$$

TODO: dibujo de la normal a una superficie.

Dado un punto $Q \in \mathbb{R}^3$, queremos saber dónde interseca un rayo $P(t)$. Es decir, para qué t se cumple que $F(P(t)) = 0 \iff F(O + tD) = 0$.

Consideremos por ejemplo un plano, como en ([Shirley and Morley 2003](#)). Para ello, nos tomamos un punto Q_0 del plano y un vector normal a la superficie \mathbf{n} .

La ecuación implícita del plano será

$$F(Q) = (Q - Q_0) \cdot \mathbf{n} = 0$$

Si pinchamos nuestro rayo en la ecuación,

$$\begin{aligned} F(P(t)) &= (P(t) - Q_0) \cdot \mathbf{n} \\ &= (O + tD - Q_0) \cdot \mathbf{n} = 0 \end{aligned}$$

Resolviendo para t , esto se da si

$$\begin{aligned} O \cdot \mathbf{n} + tD \cdot \mathbf{n} - Q_0 \cdot \mathbf{n} &= 0 && \Leftrightarrow \\ tD \cdot \mathbf{n} &= Q_0 \cdot \mathbf{n} - O \cdot \mathbf{n} && \Leftrightarrow \\ t &= \frac{Q_0 \cdot \mathbf{n} - O \cdot \mathbf{n}}{D \cdot \mathbf{n}} \end{aligned}$$

Es decir, hemos obtenido el único valor de t para el cual el rayo toca la superficie.

Debemos tener en cuenta el caso para el cual $D \cdot \mathbf{n} = 0$. Esto solo se da si la dirección y el vector normal a la superficie son paralelos.

TODO: dibujo de dos rayos con un plano: uno corta a la superficie, mientras que el otro es paralelo.

2.2.2. Superficies paramétricas

Otra forma de definir una superficie en el espacio es mediante un subconjunto $D \subset \mathbb{R}^2$ y una serie de funciones, $f, g, h : D \rightarrow \mathbb{R}$, de forma que

$$(x, y, z) = (f(u, v), g(u, v), h(u, v))$$

En informática gráfica, hacemos algo similar cuando mapeamos una textura a una superficie. Se conoce como UV mapping

Demos un par de ejemplos de superficies paramétricas: - El grafo de una función $f : D \rightarrow \mathbb{R}$,

$$G(f) = \{(x, y, f(x, y)) \mid (x, y) \in D\}$$

define una superficie diferenciable siempre que f también lo sea. - Usando coordenadas esféricas (r, θ, ϕ) , podemos parametrizar la esfera como $(x, y, z) = (\cos \phi \sin \theta, \sin \phi \sin \theta, \cos \theta)$

TODO añadir imagen de coordenadas esféricas. U otro capítulo con coordenadas.

NOTE: estoy usando (radial, polar, azimuthal). θ corresponde con la apertura con respecto a la vertical

El vector normal \mathbf{n} a la superficie en un punto (u, v) del dominio viene dado por

$$\mathbf{n}(u, v) = \left(\frac{\partial f}{\partial u}, \frac{\partial g}{\partial u}, \frac{\partial h}{\partial u} \right) \times \left(\frac{\partial f}{\partial v}, \frac{\partial g}{\partial v}, \frac{\partial h}{\partial v} \right)$$

Encontrar el punto de intersección de una superficie paramétrica con un rayo es sencillo. Basta con encontrar aquellos puntos (u, v) y t para los que

$$O_x + tD_x = f(u, v)$$

$$O_y + tD_y = g(u, v)$$

$$O_z + tD_z = h(u, v)$$

Es posible que el rayo no impacte en ningún punto. En ese caso, el sistema de ecuaciones no tendría solución. Otra posibilidad es que intersequen en varios puntos.

2.2.3. Intersecciones con esferas

Estudiemos ahora cómo intersecan una esfera con nuestro rayo. Una esfera de centro C y radio r viene dada por aquellos puntos $P = (x, y, z)$ que cumplen

$$(P - C) \cdot (P - C) = r^2$$

Podemos reescribir esta ecuación en términos de sus coordenadas para obtener

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Veamos para qué valores de t de nuestro rayo se cumple esa ecuación:

$$\begin{aligned}(P(t) - C) \cdot (P(t) - C) &= r^2 \iff \\ (O + tD - C) \cdot (O + tD - C) &= r^2 \iff\end{aligned}$$

Aplicando las propiedades del producto escalar de la conmutatividad ($a \cdot b = b \cdot a$) y la distributiva ($a \cdot (b + c) = a \cdot b + a \cdot c$), podemos escribir

$$\begin{aligned}((O - C) + tD) \cdot ((O - C) + tD) &= r^2 \iff \\ (O - C)^2 + 2 \cdot (O - C) \cdot tD + (tD)^2 &= r^2 \iff \\ D^2 t^2 + 2D \cdot (O - C)t + (O - C)^2 - r^2 &= 0 \iff\end{aligned}$$

Así que tenemos una ecuación de segundo grado. Resolviéndola, nos salen nuestros puntos de intersección:

$$t = \frac{-D \cdot (O - C) \pm \sqrt{(D \cdot (O - C))^2 - 4(D^2)((O - C)^2 - r^2)}}{2D^2}$$

Debemos distinguir tres casos, atendiendo al valor que toma el discriminante $\Delta = (D \cdot (O - C))^2 - 4(D^2)((O - C)^2 - r^2)$:

1. Si $\Delta < 0$, $\sqrt{\Delta} \notin \mathbb{R}$, y el rayo no impacta con la esfera
2. Si $\Delta = 0$, el rayo impacta en un punto, que toma el valor $t = \frac{-D \cdot (O - C)}{2D \cdot D}$. Digamos que *pegaría* justo en el borde.
3. Si $\Delta > 0$, existen dos soluciones. En ese caso, el rayo atraviesa la esfera.

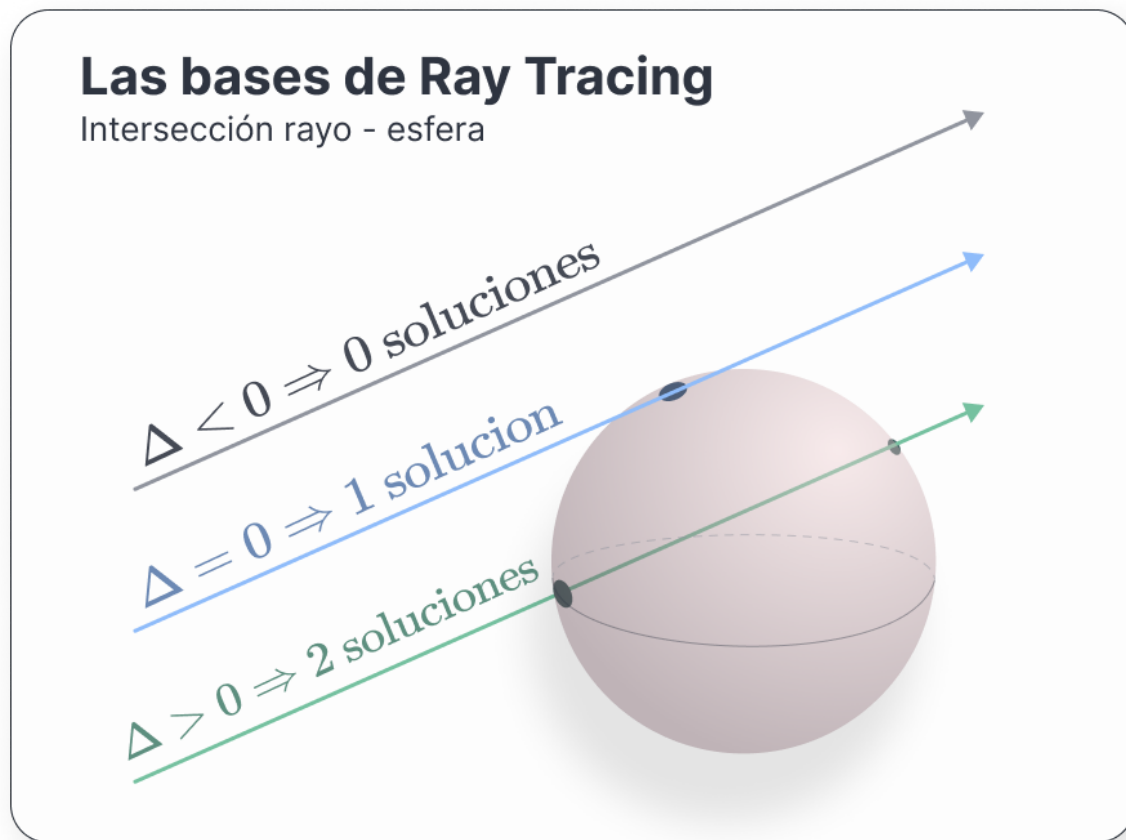


Figure 2.4.: Puntos de intersección con una esfera.

Para estos dos últimos, si consideramos t_0 cualquier solución válida, el vector normal resultante viene dado por

$$\mathbf{n} = 2(P(t_0) - C)$$

o, normalizando,

$$\hat{\mathbf{n}} = \frac{(P(t_0) - C)}{r}$$

2.2.4. Intersecciones con triángulos

Este tipo de intersecciones serán las más útiles en nuestro path tracer. Generalmente, nuestras geometrías estarán compuestas por mallas de triángulos, así que conocer dónde impacta nuestro rayo será clave. Empecemos por la base:

Un triángulo viene dado por tres puntos, A , B , y C ; correspondientes a sus vértices. Para evitar casos absurdos, supongamos que estos puntos son afinmente independientes; es decir, que no están alineados.

2.2.4.1. Coordenadas baricéntricas

Podemos describir los puntos contenidos en el plano que forman estos vertices mediante **coordenadas baricéntricas**. Este sistema de coordenadas expresa cada punto del plano como una combinación convexa de los vértices. Es decir, que para cada punto P del triángulo existen α , β y γ tales que $\alpha + \beta + \gamma = 1$ y

$$P = \alpha A + \beta B + \gamma C$$

TODO: triángulo con coordenadas baricéntricas.

Debemos destacar que existen dos grados de libertad debido a la restricción de que las coordenadas sumen 1.

Una propiedad de estas coordenadas que nos puede resultar útil es que un punto P está contenido en el triángulo si y solo si $0 < \alpha, \beta, \gamma < 1$.

Esta propiedad y la restricción de que sumen 1 nos da una cierta intuición de cómo funcionan. Podemos ver las coordenadas baricéntricas como la contribución de los vértices a un punto P . Por ejemplo, si $\alpha = 0$, eso significa que el punto viene dado por $\beta B + \gamma C$; es decir, una combinación lineal de B y C . Se encuentra en la recta que generan.

Por proponer otro ejemplo, si alguna de las coordenadas fuera mayor que 1, eso significaría que el punto estaría más allá del triángulo.

TODO: dibujo con explicación de cómo funciona (libreta Shinrin - Yoku)

2.2.4.2. Calculando la intersección

Podemos eliminar una de las variables escribiendo $\alpha = 1 - \beta - \gamma$, lo que nos dice

$$\begin{aligned} P &= (1 - \beta - \gamma)A + \beta B + \gamma C \\ &= A + (B - A)\beta + (C - A)\gamma \end{aligned}$$

bajo la restricción

$$\begin{aligned} \beta + \gamma &< 1 \\ 0 &< \beta \\ 0 &< \gamma \end{aligned} \tag{2.1}$$

Un rayo $P(t) = O + tD$ impactará en un punto del triángulo si se cumple

$$P(t) = O + tD = A + (B - A)\beta + (C - A)\gamma$$

cumpliendo [2.1]. Podemos expandir la ecuación anterior en sus coordenadas para obtener

$$\begin{aligned} O_x + tD_x &= A_x + (B_x - A_x)\beta + (C_x - A_x)\gamma \\ O_y + tD_y &= A_y + (B_y - A_y)\beta + (C_y - A_y)\gamma \\ O_z + tD_z &= A_z + (B_z - A_z)\beta + (C_z - A_z)\gamma \end{aligned}$$

Reordenamos:

$$\begin{aligned} (A_x - B_x)\beta + (A_x - C_x)\gamma + tD_x &= A_x - O_x \\ (A_y - B_y)\beta + (A_y - C_y)\gamma + tD_y &= A_y - O_y \\ (A_z - B_z)\beta + (A_z - C_z)\gamma + tD_z &= A_z - O_z \end{aligned}$$

Lo que nos permite escribir el sistema en forma de ecuación:

$$\begin{pmatrix} A_x - B_x & A_x - C_x & D_x \\ A_y - B_y & A_y - C_y & D_y \\ A_z - B_z & A_z - C_z & D_z \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = \begin{pmatrix} A_x - O_x \\ A_y - O_y \\ A_z - O_z \end{pmatrix}$$

Calcular rápidamente la solución a un sistema de ecuaciones lineales es un problema habitual. En (Shirley and Morley 2003) se utiliza la regla de Cramer para hacerlo, esperando que el compilador optimice las variables intermedias creadas. Nosotros no nos tendremos que preocupar de esto en particular, ya que el punto de impacto lo calculará la GPU gracias a las herramientas aportadas por KHR (The Khronos® Vulkan Working Group n.d.).

Para obtener el vector normal, podemos hacer el producto vectorial de dos vectores que se encuentren en el plano del triángulo. Como, por convención, los vértices se guardan en sentido antihorario visto desde fuera del objeto, entonces

$$\mathbf{n} = (B - A) \times (C - A)$$

Referencias

(Wikipedia: Implicit surface n.d.), (Wikipedia: Parametric surface n.d.), (Wikipedia: Barycentric coordinate system n.d.), (Wikipedia: Differential geometry of surfaces n.d.)

3. Transporte de luz

En este capítulo estudiaremos las bases de la radiometría. Esta área de la óptica nos proporcionará una serie de herramientas con las cuales podremos responder a la pregunta *cuánta luz existe en un punto*.

3.1. Unidades radiométricas básicas

Nota: cuando usemos un paréntesis tras una ecuación, dentro denotaremos sus unidades de medida.

Antes de comenzar a trabajar, necesitamos conocer *qué entendemos* por luz. Aunque hay muchas formas de trabajar con ella (a fin de cuentas, todavía seguimos discutiendo sobre *qué es exactamente la luz*¹), nosotros nos quedaremos con algunas pinceladas de la cuántica. Nos será suficiente quedarnos con la concepción de fotón. Una fuente de iluminación emite una serie de fotones. Estos fotones tienen(Shirley and Morley 2003) una posición, una dirección de propagación y una longitud de onda λ . Un fotón también tiene asociado una velocidad c que depende del índice de refracción del medio, n .

La unidad de medida de λ es el nanómetro (nm). También nos vendrá bien definir una frecuencia, f . Su utilidad viene del hecho de que, cuando la luz cambia de medio al propagarse, la frecuencia se mantiene constante.

$$f = \frac{c}{\lambda}$$

¹No entraremos en detalle sobre la naturaleza de la luz. Sin embargo, si te pica la curiosidad, hay muchos divulgadores como (QuantumFracture n.d.) que han tratado el tema con suficiente profundidad.

Un fotón tiene asociada una carga de energía, denotada por Q :

$$Q = hf = \frac{hc}{\lambda} (\text{J})$$

donde $h = 6.62607004 \times 10^{-34} \text{J} \cdot \text{s}$ es la constante de Plank y $c = 299792458 \text{m/s}$ la velocidad de la luz.

En realidad, **todas estas cantidades deberían tener un subíndice λ** , puesto que dependen de la longitud de onda. La energía de un fotón Q , por ejemplo, debería denotarse Q_λ . Sin embargo, en la literatura de informática gráfica, **se ha optado por omitirla**. ¡Tenlo en cuenta a partir de aquí!

3.1.1. Potencia

A partir de la energía anterior, podemos estimar *la tasa de producción de energía*. A esta tasa la llamaremos (Pharr, Jakob, and Humphreys 2016) **potencia**, o **flujo radiante** Φ . Esta medida nos resultará más útil que la energía total, puesto que nos permite estimar la energía en un instante:

$$\Phi = \lim_{\Delta t \rightarrow 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt} (\text{J/s})$$

Su unidad es julios por segundo, comúnmente denotado vatio (*watts*, W). También se utiliza el lumen. Podemos encontrar la energía total en un periodo de tiempo $[t_0, t_1]$ integrando el flujo radiante:

$$Q = \int_{t_0}^{t_1} \Phi(t) dt$$

3.1.2. Irradiancia

La **irradiancia** o **radiancia emitida** es el flujo radiante que recibe una superficie. Dada un área A , se define como

$$E = \frac{\Phi}{A} (\text{W/m}^2)$$

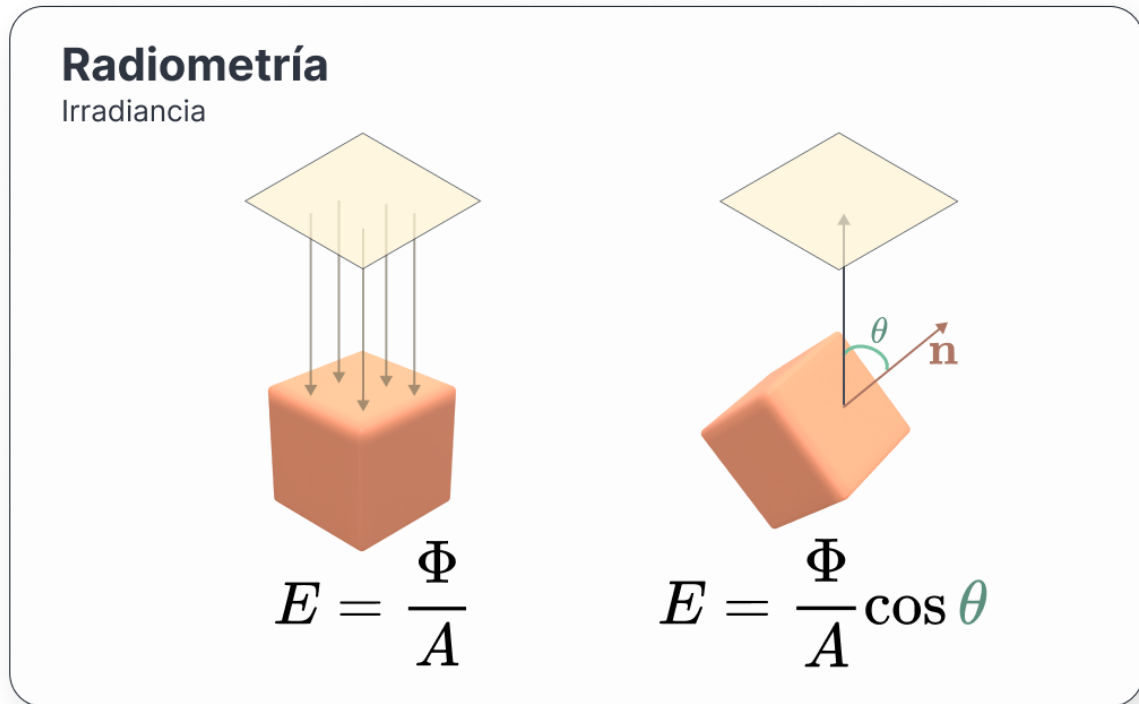


Figure 3.1.: La irradiancia es la potencia por metro cuadrado incidente en una superficie. Es proporcional al coseno del ángulo entre la dirección de la luz y la normal a la superficie.

Ahora que tenemos la potencia emitida en una cierta área, nos surge una pregunta: ¿y en un cierto punto p ? Tomando límites en la expresión anterior, encontramos la respuesta:

$$E(p) = \lim_{\Delta A \rightarrow 0} \frac{\Delta \Phi}{\Delta A} = \frac{d\Phi}{dA} (\text{W/m}^2)$$

De la misma manera que con la potencia, integrando $E(p)$ podemos obtener el flujo radi-

ante:

$$\Phi = \int_A E(p) dp$$

El principal problema de la irradiancia es que *no nos dice nada sobre las direcciones* desde las que ha llegado la luz.

3.1.3. Ángulos sólidos

Con estas tres unidades básicas, nos surge una pregunta muy natural: *¿cómo mido cuánta luz llega a una superficie?*

Para responder a esta pregunta, necesitaremos los **ángulos sólidos**. Son la extensión de los **ángulos planares**, en dos dimensiones.

Ilustremos el sentido de estos ángulos: imaginemos que tenemos un cierto objeto en dos dimensiones delante de nosotros, a una distancia desconocida. ¿Sabríamos cuál es su tamaño, solo con esta información? Es más, si entrara otro objeto en la escena, ¿podríamos distinguir cuál de ellos es más grande?

Parece difícil responder a estas preguntas. Sin embargo, sí que podemos determinar *cómo de grandes nos parecen* desde nuestro punto de vista. Para ello, describimos una circunferencia de radio r alrededor nuestra. Si trazamos un par de líneas desde nuestra posición a las partes más alejadas de este objeto, y las cortamos con nuestra circunferencia, obtendremos un par de puntos inscritos en ella. Pues bien, al arco que encapsulan dichos puntos le vamos a hacer corresponder un cierto ángulo: el ángulo planar.

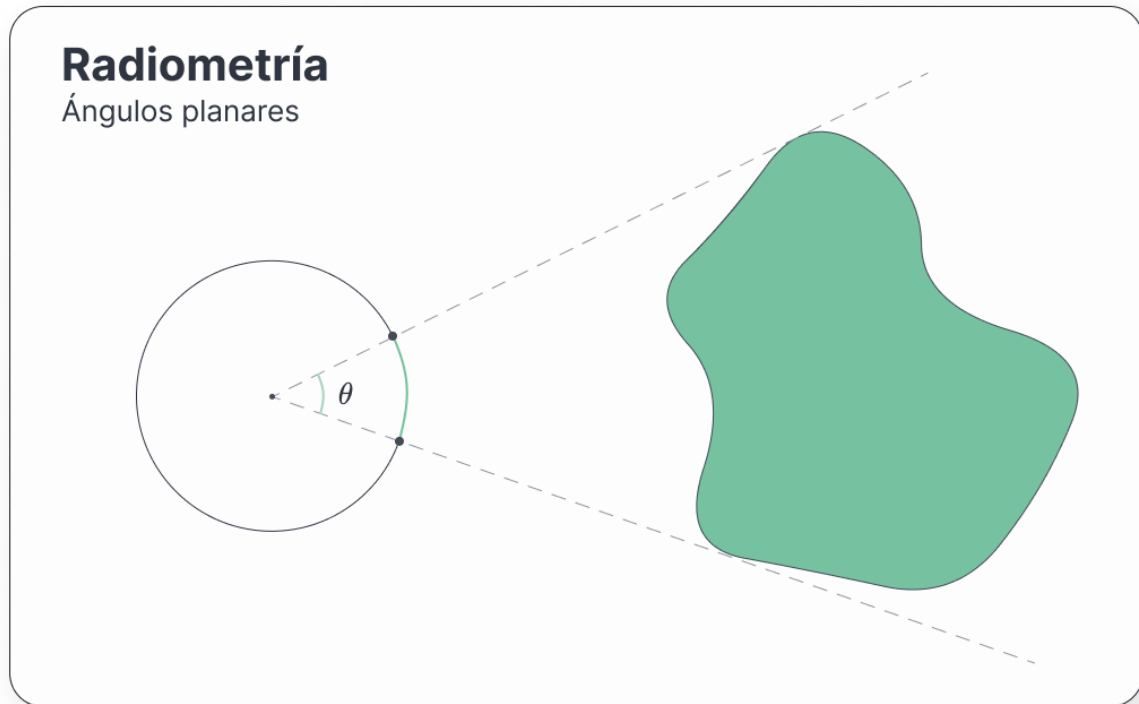


Figure 3.2.: La idea intuitiva de un ángulo planar

Llevando esta idea a las tres dimensiones es como conseguimos el concepto de **ángulo sólido**. Si en dos dimensiones teníamos una circunferencia, aquí tendremos una esfera. Cuando generemos las rectas proyectantes hacia el volumen, a diferencia de los ángulos planares, se inscribirá un área en la esfera. La razón entre dicha área A y el cuadrado del radio r nos dará un ángulo sólido:

$$\omega = \frac{A}{r^2}(\text{sr})$$

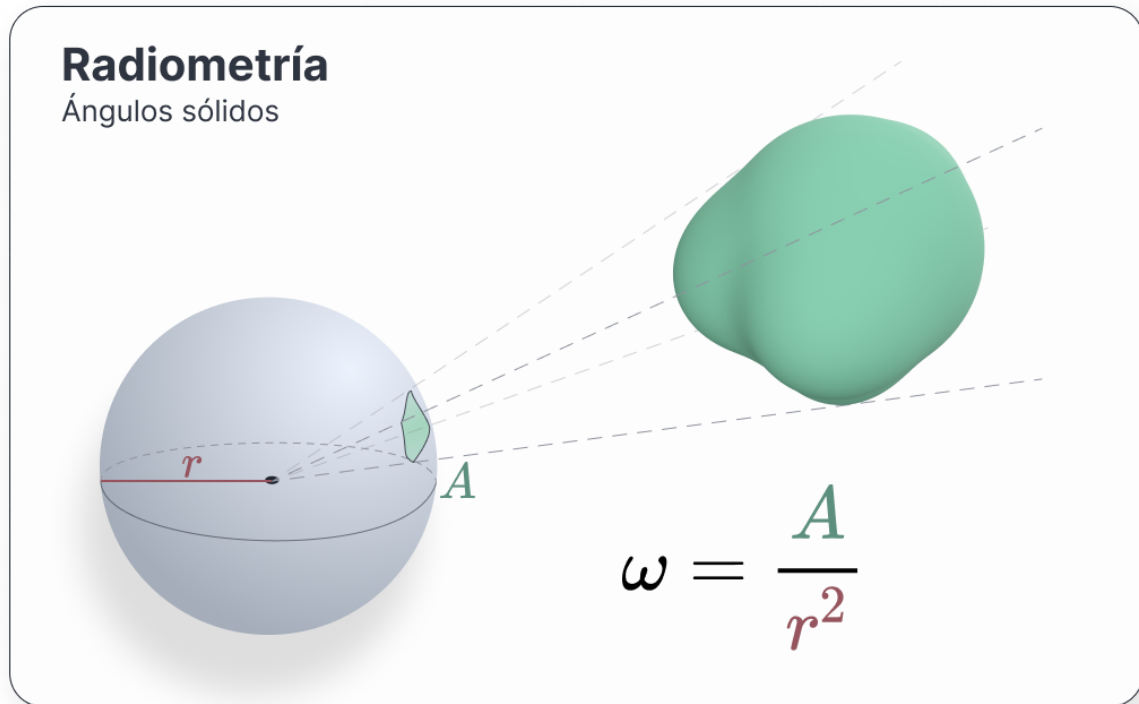


Figure 3.3.: Un ángulo sólido es la razón entre el área proyectada y el cuadrado del radio

Los denotaremos por ω . En física se suele usar Ω , pero aquí optaremos por la minúscula. Su unidad de medida es el estereorradián (sr). Se tiene que $\omega \in [0, 4\pi]$. Si 2π radianes corresponden a la circunferencia completa, para la esfera se tiene que 4π estereorradianes cubren toda la superficie de esta. Se tiene también que 2π sr cubren un hemisferio. Además, un estereorradián corresponde a una superficie con área r^2 : $1\text{sr} = \frac{r^2}{r^2}$.

De vez en cuando, usaremos ω **un vector dirección unitario en la esfera**.

THE SIZE OF THE PART OF EARTH'S SURFACE DIRECTLY UNDER VARIOUS SPACE OBJECTS

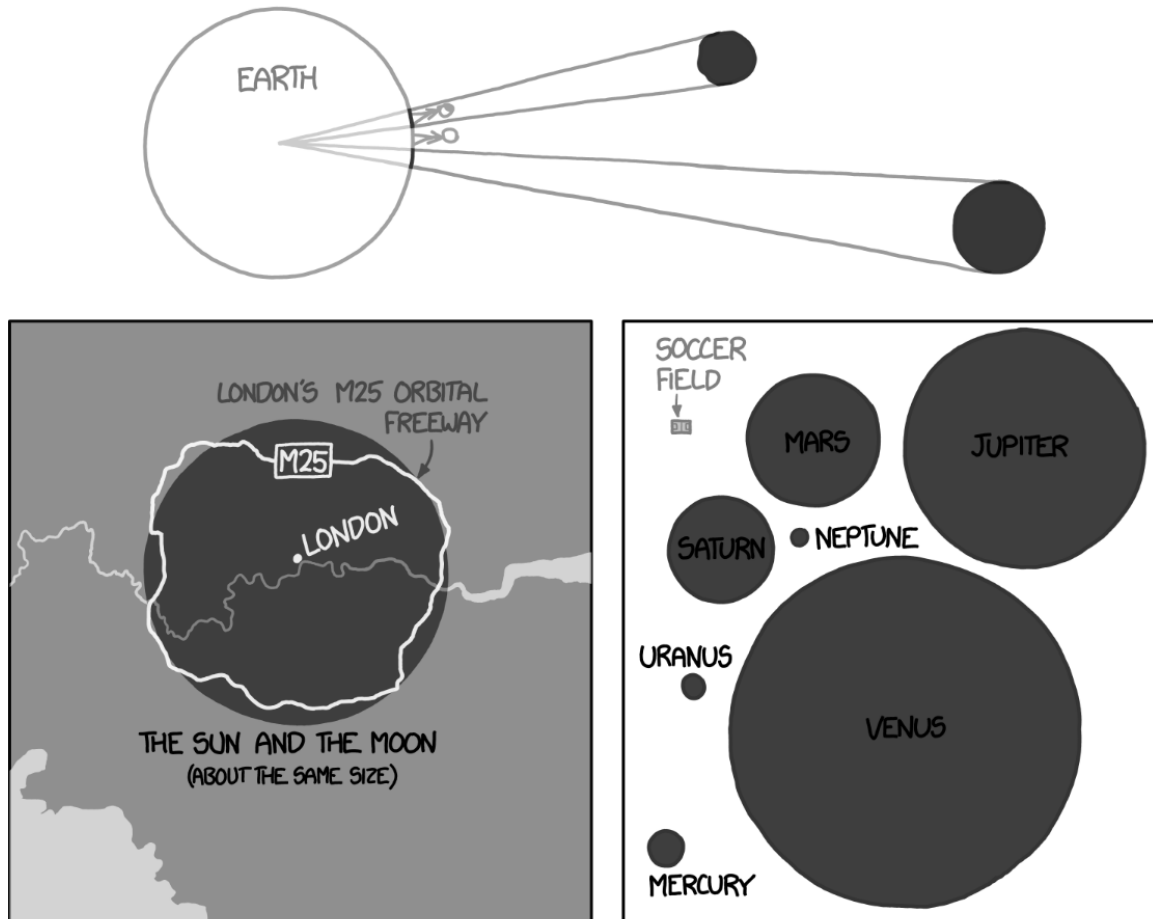


Figure 3.4.: Como de costumbre, hay un xkcd relevante. ([Fuente](#))

Usualmente emplearemos coordenadas esféricas cuando trabajemos con ellos, dado que resulta más cómodo.

$$\begin{cases} x = \sin \theta \cos \phi \\ y = \sin \theta \sin \phi \\ z = \cos \theta \end{cases}$$

A θ se le denomina ángulo polar, mientras que a ϕ se le llama acimut. Imaginémonos un punto en la esfera de radio r ubicado en una posición (r, θ, ϕ) . Queremos calcular un área chiquitita dA_h , de forma que el ángulo sólido asociado a dicha área debe ser $d\omega$. Así, $d\omega = \frac{dA_h}{r^2}$. Si proyectamos el área, obtenemos $d\theta$ y $d\phi$: pequeños cambios en los ángulos que nos generan nuestra pequeña área.

dA_h debe tener dos lados $lado_1$ y $lado_2$. Podemos hallar $lado_1$ si lo trasladamos al eje z de nuevo. Así, $lado_1 = r \sin \theta$. De la misma manera, $lado_2 = r d\theta$.

TODO: foto que explique todo esto, porque si no, no hay quien se entere. Quizás me sirva la de <https://cs184.eecs.berkeley.edu/public/sp22/lectures/lec-11-radiometry-and-photometry/lec-11-radiometry-and-photometry.pdf>, p.16 siempre que adapte ϕ .

Poniendo estos valores en $d\omega$:

$$\begin{aligned} d\omega &= \frac{dA_h}{r^2} = \frac{lado_1 lado_2}{r^2} = \\ &= \frac{r \sin \theta d\phi r d\theta}{r^2} = \\ &= \sin \theta d\theta d\phi \end{aligned} \tag{3.1}$$

¡Genial! Acabamos de añadir un recurso muy potente a nuestro inventario. Esta expresión nos permitirá convertir integrales sobre ángulos sólidos en integrales sobre ángulos esféricos.

3.1.4. Intensidad radiante

Los ángulos sólidos nos proporcionan una variedad de herramientas nuevas considerable. Gracias a ellos, podemos desarrollar algunos conceptos nuevos. Uno de ellos es la **intensidad radiante**.

Imaginémonos un pequeñito punto de luz encerrado en una esfera, el cual emite fotones en todas direcciones. Nos gustaría medir cuánta energía pasa por la esfera. Podríamos entonces definir

$$I = \frac{\Phi}{4\pi} (\text{W/sr})$$

Otra unidad de medida es el lumen por esterrradián, (lm/sr). La anterior definición mide cuántos fotones pasan por toda la esfera. ¿Qué ocurre si *cerramos* el ángulo, restringiéndonos así a un área muy pequeña de la esfera?

$$I = \lim_{\Delta\omega \rightarrow 0} \frac{\Delta\Phi}{\Delta\omega} = \frac{d\Phi}{d\omega}$$

De la misma manera que con los conceptos anteriores, podemos volver a la potencia integrando sobre un conjunto de direcciones:

$$\Phi = \int_{\Omega} I(\omega) d\omega$$

3.1.5. Radiancia

Finalmente, llegamos al concepto más importante. La **radiancia espectral** (o radiancia a secas²) es una extensión de la radiancia emitida teniendo en cuenta la dirección:

$$L(p, \omega) = \lim_{\Delta\omega \rightarrow 0} \frac{\Delta E_{\omega}(p)}{\Delta\omega} = \frac{dE_{\omega}(p)}{d\omega}$$

siendo $E_{\omega}(p)$ la radiancia emitida a la superficie perpendicular a ω .

TODO: foto como la de <https://cs184.eecs.berkeley.edu/public/sp22/lectures/lec-11-radiometry-and-photometry/lec-11-radiometry-and-photometry.pdf>, página 10.

Podemos dar otra expresión de la radiancia en términos del flujo:

$$L(p, \omega) = \frac{d^2\Phi(p, \omega)}{d\omega dA^{\perp}} = \frac{d^2\Phi(p, \omega)}{d\omega dA \cos \theta} \quad (3.2)$$

donde dA^{\perp} es el área proyectada por dA en una hipotética superficie perpendicular a ω :

²Recuerda que estamos omitiendo la longitud de onda λ .

TODO: figura similar a pbr figura 5.10 https://www.pbr-book.org/3ed-2018/Color_and_Radiometry/

Cuando un rayo impacta en una superficie, L puede tomar valores muy diferentes en un lado y otro de dicha superficie. Por ejemplo, si nos imaginamos un espejo, el valor un poco por encima y un poco por debajo de un punto del espejo es muy diferente. Para solucionarlo, podemos tomar límites para distinguir a ambos lados:

$$\begin{aligned} L^+(p, \omega) &= \lim_{t \rightarrow 0^+} L(p + t\mathbf{n}_p, \omega) \\ L^-(p, \omega) &= \lim_{t \rightarrow 0^-} L(p + t\mathbf{n}_p, \omega) \end{aligned} \quad (3.3)$$

donde \mathbf{n}_p es la normal en el punto p .

Otra forma de solucionarlo (y preferible, puesto que simplifica entender lo que ocurre) es distinguir entre la radiancia que llega a un punto –la incidente–, y la saliente.

La primera se llamará $L_i(p, \omega)$, mientras que la segunda será $L_o(p, \omega)$. Es importante destacar que ω apunta *hacia fuera* de la superficie. Quizás es contraintuitivo en L_i , puesto que $-\omega$ apunta *hacia* la superficie. Depende del autor se utiliza una concepción u otra.

Nota(ción): a L_o también se le conoce como la radiancia reflejada. Por eso, algunas veces aparece como L_r en algunas fuentes.

Utilizando esta notación y usando [3.3], podemos escribir L_i y L_o como

$$\begin{aligned} L_i(p, \omega) &= \begin{cases} L^+(p, -\omega) & \text{si } \omega \cdot \mathbf{n}_p > 0 \\ L^-(p, -\omega) & \text{si } \omega \cdot \mathbf{n}_p < 0 \end{cases} \\ L_o(p, \omega) &= \begin{cases} L^+(p, \omega) & \text{si } \omega \cdot \mathbf{n}_p > 0 \\ L^-(p, \omega) & \text{si } \omega \cdot \mathbf{n}_p < 0 \end{cases} \end{aligned}$$

Hacemos esta distinción porque, a fin de cuentas, necesitamos distinguir entre los fotones que llegan a la superficie y los que salen.

TODO: <https://cs184.eecs.berkeley.edu/public/sp22/lectures/lec-11-radiometry-and-photometry/lec-11-radiometry-and-photometry.pdf>, p.36

Una propiedad a tener en cuenta es que, si cogemos un punto p del espacio donde no existe ninguna superficie, $L_o(p, \omega) = L_i(p, -\omega) = L(p, \omega)$

La importancia de la radiancia se debe a un par de propiedades:

La primera de ellas es que, dado L , podemos calcular cualquier otra unidad básica mediante integración. Además, **su valor se mantiene constante en rayos que viajan en el vacío en línea recta** (Fabio Pellacini n.d.). Esto último hace que resulte muy natural usarla en un ray tracer.

Veamos por qué ocurre esto:

TODO: https://pellacini.di.uniroma1.it/teaching/graphics17b/lectures/12_pathtracing.pdf, página 18.

Consideremos dos superficies ortogonales entre sí, S_1 y S_2 separadas una distancia r . Debido a la conservación de la energía, cualquier fotón que salga de una superficie y se encuentre bajo el ángulo sólido de la otra debe llegar impactar en dicha superficie opuesta.

Por tanto:

$$d^2\Phi_1 = d^2\Phi_2$$

Sustituyendo en la expresión de la radiancia [3.2], y teniendo en cuenta que son ortogonales (lo que nos dice que $\cos \theta = 1$):

$$L_1 d\omega_1 dA_1 = L_2 d\omega_2 dA_2$$

Por construcción, podemos cambiar los ángulos sólidos:

$$L_1 \frac{dA_2}{r^2} dA_1 = L_2 \frac{dA_1}{r^2} dA_2$$

Lo que finalmente nos dice que $L_1 = L_2$, como queríamos ver.

3.2. Fotometría y radiometría

TODO: hablar sobre las diferencias. Hay información útil en 01_lights.pdf, p.43

3.3. Integrales radiométricas

En esta sección, vamos a explorar las nuevas herramientas que nos proporciona la radiancia. Veremos también cómo integrar ángulos sólidos, y cómo simplificar dichas integrales.

3.3.1. Una nueva expresión de la irradiancia y el flujo

Como dijimos al final de [la sección de la irradiancia](#), esta medida no tiene en cuenta las direcciones desde las que llegaba la luz. A diferencia de esta, la radiancia sí que las utiliza. Dado que una de las ventajas de la radiancia es que nos permite obtener el resto de medidas radiométricas, ¿por qué no desarrollamos una nueva expresión de la irradiancia?

Para obtener cuánta luz llega a un punto, debemos acumular la radiancia incidente que nos llega desde cualquier dirección.

TODO: dibujo como el de la libreta roja. Me lo mandé por Telegram, por si no lo encuentro

Dado un punto p que se encuentra en una superficie con normal \mathbf{n} en dicho punto, la irradiancia se puede expresar como

$$E(p, \mathbf{n}) = \int_{\Omega} L_i(p, \omega) |\cos \theta| d\omega \quad (3.4)$$

El término $\cos \theta$ aparece en la integral debido a la derivada del área proyectada, dA^\perp . θ es el ángulo entre la dirección ω y la normal \mathbf{n} .

Generalmente, la irradiancia se calcula únicamente en el hemisferio de direcciones asociado a la normal en el punto, $H^2(\mathbf{n})$.

Podemos eliminar el $\cos \theta$ de la integral mediante una pequeña transformación: proyectando el ángulo sólido sobre el disco alrededor del punto p con normal \mathbf{n} , obtenemos una expresión más sencilla: como $d\omega^\perp = |\cos \theta|d\omega$, entonces

$$E(p, \mathbf{n}) = \int_{H^2(\mathbf{n})} L_i(p, \omega) d\omega^\perp$$

Usando lo que aprendimos sobre la derivada de los ángulos sólidos [3.1], se puede reescribir la ecuación anterior como

$$E(p, \mathbf{n}) = \int_0^{2\pi} \int_0^{\pi/2} L_i(p, \theta, \phi) \cos \theta \sin \theta d\theta d\phi$$

Haciendo el mismo juego con el flujo emitido de un cierto objeto al hemisferio que encapsula la normal, conseguimos:

$$\begin{aligned} \Phi &= \int_A \int_{H^2(\mathbf{n})} L_o(p, \omega) \cos \theta d\omega dA = \\ &= \int_A \int_{H^2(\mathbf{n})} L_o(p, \omega) d\omega^\perp dA \end{aligned}$$

TODO: a lo mejor merece la pena hacer un ejemplo sobre los diferentes tipos de luz, como en <https://cs184.eecs.berkeley.edu/public/sp22/lectures/lec-11-radiometry-and-photometry/lec-11-radiometry-and-photometry.pdf> p.41? O a lo mejor un capítulo para hablar de luces en general.

3.3.2. Integrando sobre área

Una herramienta más que nos vendrá bien será la capacidad de convertir integrales sobre direcciones en integrales sobre área. Hemos hecho algo similar en las secciones anteriores, así que no perdemos nada por generalizarlo.

Considera un punto p sobre una superficie con normal en dicho punto \mathbf{n} . Supongamos que tenemos una pequeña área dA con normal \mathbf{n}_{dA} . Sea θ el ángulo entre \mathbf{n} y \mathbf{n}_{dA} , y r la

distancia entre p y dA .

Entonces, la relación entre la diferencial de un ángulo sólido y la de un área es

$$d\omega = \frac{dA \cos \theta}{r^2}$$

TODO: figura como la de pbr book 5.16.

Esto nos permite, por ejemplo, expandir algunas expresiones como la de la irradiancia [3.4] si partimos de un cuadrilátero dA :

$$\begin{aligned} E(p, \mathbf{n}) &= \int_{\Omega} L_i(p, \omega) |\cos \theta| d\omega = \\ &= \int_A L \cos \theta \frac{\cos \theta_o}{r^2} dA \end{aligned}$$

siendo θ_o el ángulo de la radiancia de salida de la superficie del cuadrilátero.

3.4. Dispersión de luz: las familias de funciones de distribución bidireccionales

Cuando una fuente de luz emite fotones hacia una superficie impactando en ella, ocurren un par de sucesos: parte de la luz se refleja en ella, saliendo disparada hacia alguna dirección; mientras que otra parte se absorbe.

En informática gráfica se consideran tres tipos principales de dispersión de luz: **dispersión en superficie** (*surface scattering*), **dispersión volumétrica** (*volumetric scattering*) y **dispersión bajo superficie** (*subsurface scattering*)

En este capítulo vamos a modelar la primera. Estudiaremos qué es lo que ocurre cuando los fotones alcanzan una superficie, en qué dirección se reflejan, y cómo cambia el comportamiento dependiendo de las propiedades del material.

3.4.1. La función de distribución de reflectancia bidireccional (BRDF)

La **función de distribución de reflectancia bidireccional** (en inglés, *bidirectional reflectance distribution function*, BRDF) describe cómo la luz se refleja en una superficie opaca. Se encarga de informarnos sobre cuánta radiancia sale en dirección ω_o debido a la radiancia incidente desde la dirección ω_i , partiendo de un punto p en una superficie con normal \mathbf{n} . Depende de la longitud de onda λ , pero, como de costumbre, la omitiremos.

Intuición: ¿cuál es la probabilidad de que, habiéndome llegado un fotón desde ω_i , me salga disparado hacia ω_o ?

TODO: esquema como el de pbr fig. 5.18, o como <https://pellacini.di.uniroma1.it/teaching/graphics17/> p.20

Si consideramos ω_i como un cono diferencial de direcciones, la irradiancia diferencial en p viene dada por

$$dE(p, \omega_i) = L_i(p, \omega_i) \cos \theta_i d\omega_i$$

Debido a esta irradiancia, una pequeña parte de radiancia saldrá en dirección ω_o , proporcional a la irradiancia:

$$dL_o(p, \omega_o) \propto dE(p, \omega_i)$$

Si lo ponemos en forma de cociente, sabremos exactamente cuál es la proporción de luz. A este cociente lo llamaremos $f_r(p, \omega_o \leftarrow \omega_i)$; la función de distribución de reflectancia bidireccional:

$$f_r(p, \omega_o \leftarrow \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos \theta_i d\omega_i} (1/\text{sr})$$

Nota(ción): dependiendo de la fuente que estés leyendo, es posible que te encuentres una integral algo diferente. Por ejemplo, en tanto en Wikipedia como en (Shirley and Morley 2003) se integra con respecto a los ángulos de salida ω_o , en vez de los incidentes.

Aquí, usaremos la notación de integrar con respecto a los incidentes, como se hace en (Pharr, Jakob, and Humphreys 2016).

Las BRDF físicamente realistas tienen un par de propiedades importantes:

1. **Reciprocidad:** para cualquier par de direcciones ω_i, ω_o , se tiene que $f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o \leftarrow \omega_i)$.
2. **Conservación de la energía:** La energía reflejada tiene que ser menor o igual que la incidente:

$$\int_{H^2(\mathbf{n})} f_r(p, \omega_o \leftarrow \omega_i) \cos \theta_i d\omega_i \leq 1$$

3.4.2. La función de distribución de transmitancia bidireccional (BTDF)

Si la BRDF describe cómo se refleja la luz, la *bidirectional transmittance distribution function* (abreviada BTDF) nos informará sobre la transmitancia; es decir, cómo se comporta la luz cuando *entra* en un medio. Generalmente serán dos caras de la misma moneda: cuando la luz impacta en una superficie, parte de ella, se reflejará, y otra parte se transmitirá.

Puedes imaginarte la BTDF como una función de reflectancia del hemisferio opuesto a donde se encuentra la normal de la superficie.

Denotaremos a la BTDF por

$$f_t(p, \omega_o \leftarrow \omega_i)$$

Al contrario que en la BRDF, ω_o y ω_i se encuentran en hemisferios diferentes.

3.4.3. Juntando la BRDF y la BTDF en La función de distribución de dispersión bidireccional

Convenientemente, podemos unir la BRDF y la BTDF en una sola expresión, llamada **la función de distribución de dispersión bidireccional** (*bidirectional scattering distribution function*, BSDF). A la BSDF la denotaremos por

$$f(p, \omega_o \leftarrow \omega_i)$$

Intuición: la BSDF son todas las posibles direcciones en las que puede salir disparada la luz.

Usando esta definición, podemos obtener

$$dL_o(p, \omega_o) = f(p, \omega_o \leftarrow \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i$$

Esto nos deja a punto de caramelo una nueva expresión de la radiancia en términos de la radiancia incidente en un punto p . Integrando la expresión anterior, obtenemos

$$L_o(p, \omega_o) = \int_{\mathbb{S}^2} f(p, \omega_o \leftarrow \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i \quad (3.5)$$

siendo \mathbb{S}^2 la esfera.

Esta forma de expresar la radiancia es muy importante. Generalmente se le suele llamar la *ecuación de dispersión* (*scattering equation*, en inglés). Dado que es una integral muy importante, seguramente tengamos que evaluarla repetidamente. ¡Los métodos de Monte Carlo nos vendrán de perlas! Más adelante hablaremos de ella.

Las BSDFs tienen unas propiedades interesantes:

- **Positividad:** como los fotones no se pueden reflejar “negativamente”, $f(p, \omega_o \leftarrow \omega_i) \geq 0$.
- **Reciprocidad de Helmholtz:** se puede invertir un rayo de luz: $f(p, \omega_o \leftarrow \omega_i) = f(p, \omega_i \leftarrow \omega_o)$.

- **Conservación de la energía:** todos los fotones que llegan a la superficie deben ser reflejados o absorbidos. Es decir, no se emite ningún fotón nuevo:

$$\int_{H^2(\mathbf{n})} f(p, \omega_o \leftarrow \omega_i) \cos \theta_i d\omega_i \leq 1 \quad \forall \omega_o$$

3.4.4. Reflectancia hemisférica

Puede ser útil tomar el comportamiento agregado de las BRDFs y las BTDFs y reducirlo un cierto valor que describa su comportamiento general de dispersión. Sería Algo así como un resumen de su distribución. Para conseguirlo, vamos a introducir dos nuevas funciones:

La **reflectancia hemisférica-direccional** (*hemispherical-directional reflectance*) describe la reflexión total sobre un hemisferio debida a una fuente de luz que proviene desde la dirección ω_o :

$$\rho_{hd}(\omega_o) = \int_{H^2(n)} f_r(p, \omega_o \leftarrow \omega_i) |\cos \theta_i| d\omega_i$$

Por otra parte, la **reflectancia hemisférica-hemisférica** (*hemispherical-hemispherical reflectance*) es un valor espectral que nos proporciona el ratio de luz incidente reflejada por una superficie, suponiendo que llega la misma luz desde todas direcciones:

$$\rho_{hh} = \frac{1}{\pi} \int_{H^2(n)} \int_{H^2(n)} f_r(p, \omega_o \leftarrow \omega_i) |\cos \theta_o \cos \theta_i| d\omega_o d\omega_i$$

3.4.5. Reflejos

Una vez hemos definido las funciones de distribución bidireccionales, debemos encargarnos de modelar el comportamiento explícitamente. Para ello, veamos cómo los materiales modifican las distribuciones.

En esencia, los reflejos se pueden clasificar en cuatro grandes tipos:

- **Difusos** (*Diffuse*): esparcen la luz en todas direcciones casi equiprobablemente. Por ejemplo, la tela y el papel son materiales difusos.
- **Especulares brillantes** (*Glossy specular*): la distribución de luz se asemeja a un cono. La chapa de un coche es un material especular brillante.
- **Especulares perfectos** (*Perfect specular*): en esencia, son espejos. El ángulo de salida de la luz es muy pequeño, por lo que reflejan casi a la perfección lo que les llega.
- **Retroreflectores** (*Retro reflective*): la luz se refleja en dirección contraria a la de llegada. Esto es lo que sucede a la luna.

Ten en cuenta que es muy difícil encontrar objetos físicos que imiten a la perfección un cierto modelo. Suelen recaer en un híbrido entre dos o más modelos.

Fijado un cierto modelo, la función de distribución de reflectancia, BRDF, puede ser **isotrópica** o **anisotrópica**. Los materiales isotrópicos mantienen las propiedades de reflectancia invariantes ante rotaciones; es decir, la distribución de luz es la misma en todas direcciones. Por el contrario, los anisotrópicos reflejan diferentes cantidades de luz dependiendo desde dónde los miremos. Los ejemplos más habituales de materiales anisotrópicos son las rocas y la madera.

3.5. La rendering equation

Y, finalmente, tras esta introducción de los principales conceptos radiométricos, llegamos a la ecuación más importante de todo este trabajo: la **rendering equation**; también llamada la **ecuación del transporte de luz**.

Nota(ción): esta vez no traduciré el concepto. Es cierto que afea un poco la escritura teniendo en cuenta que esto es un texto en castellano. Sin embargo, la otra opción es inventarme una traducción que nadie usa.

Antes de comenzar, volvamos a plantear de nuevo la situación: nos encontramos observando desde nuestra pantalla una escena virtual mediante la cámara. Queremos saber qué color tomará un pixel específico. Para conseguirlo, dispararemos rayos desde nuestro punto de vista hacia el entorno, haciendo que reboten en los objetos. Cuando un

rayo impacte en una superficie, adquirirá parte de las propiedades del material del objeto. Además, de este rayo surgirán otros nuevos (un rayo dispersado y otro refractado), que a su vez repetirán el proceso. La información que se obtiene a partir de estos caminos de rayos nos permitirá darle color al píxel.

La *rendering equation* se va a encargar de describir analíticamente cómo ocurre esto.

Un último concepto más: denotemos por $L_e(p, \omega_o)$ a **la radiancia producida por los materiales emisivos**. Por ejemplo, una luz emite radiancia por sí misma.

Bien, partamos de la ecuación de para la radiancia reflejada:

$$L_o(p, \omega_o) = \int_{H^2(\mathbf{n})} f(p, \omega_o \leftarrow \omega_i) L_i(p, \omega_i) \cos \theta_i d\omega_i$$

Vamos a buscar expresar la radiancia incidente en términos de la radiancia reflejada. Para ello, usamos la propiedad de que la radiancia a lo largo de un rayo no cambia.

Si a una superficie le llega un fotón desde alguna parte, debe ser porque “*alguien*” ha tenido que emitirlo. El fotón necesariamente ha llegado a partir de un rayo. La propiedad nos dice que la radiancia no ha podido cambiar en el camino.

Pues bien, consideremos una función $r : \mathbb{R}^3 \times \Omega \rightarrow \mathbb{R}^3$ tal que, dado un punto p y una dirección ω , devuelve el siguiente punto de impacto en una superficie. En esencia, es una función de *ray casting*.

TODO: foto como la de https://pellacini.di.uniroma1.it/teaching/graphics17b/lectures/12_pathtracing
p.29

Esta función nos permite expresar el punto anterior de la siguiente forma:

$$L_i(p, \omega) = L_o(r(p, \omega), -\omega)$$

Esto nos permite cambiar la expresión de L_i en la integral anterior:

$$L_o(p, \omega_o) = \int_{H^2(\mathbf{n})} f(p, \omega_o \leftarrow \omega_i) L_o(r(p, \omega_i), -\omega_i) \cos \theta_i d\omega_i$$

Finalmente, la radiancia total vendrá dada por la suma de la radiancia emitida y la reflejada:

$$L(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2(\mathbf{n})} f(p, \omega_o \leftarrow \omega_i) L_o(r(p, \omega_i), -\omega_i) \cos \theta_i d\omega_i \quad (3.6)$$

Y con esto, ¡hemos obtenido la *rendering equation*!

Si quieres ver gráficamente cómo funciona, te recomiendo pasarte por (Arnebäck n.d.). Es un vídeo muy intuitivo.

Si nos paramos a pensar, la ecuación de reflexión es muy similar a la de renderizado. Sin embargo, hay un par de matices que las hacen muy diferentes:

- La ecuación de reflexión describe cómo se comporta la luz reflejada en un cierto punto. Es decir, tiene un ámbito local. Además, para calcular la radiancia reflejada, se necesita conocer la radiancia incidente.
- La *rendering equation* calcula las condiciones globales de la luz. Además, no se conocen las radiancias de salida.

Este último matiz es importante. Para renderizar una imagen, se necesita calcular la radiancia de salida para aquellos puntos visibles desde nuestra cámara.

3.6. Materiales

TODO: <https://alain.xyz/blog/advances-in-material-models> este señor me acaba de solucionar la vida. Gracias por tanto.

TODO: WIP. Bastante WIP. Basado en O2_Rendering_Zsolnai_Ray_Tracing.

- La suma de las siguientes 3 componentes hacen el modelo de Phong:
 - Ambient: $I = K_\alpha I_\alpha$, con k_α el coeficiente ambiental del objeto, I_α la intensidad ambiental de la escena/fuente de luz
 - Diffuse (simplificada): $I = k_d (L \cdot N)$, k_d coeficiente difuso del objeto, L vector que apunta a la luz, N normal a la superficie

- Specular (simplificada): $I = k_s(V \cdot R)^n$, k_s especular, V vector apuntando a la cámara, R vector reflejado del rayo, $()^n$ shininess factor.
- $I = K_\alpha I_\alpha + I_i(k_d(L \cdot N) + k_s(V \cdot R)^n)$
- Aproximación muy bruta
- Con recursividad, $I = K_\alpha I_\alpha + I_i(k_d(L \cdot N) + k_s(V \cdot R)^n) + k_t I_t + k_r I_r$, k_t fresnel transmission coefficient, I_t intensity coming from the transmission direction, k_r fresnel reflection coefficient, I_r intensity coming from the reflection direction.

3.6.1. Ecuaciones de fresnel, ley de Snell

Referencias

(Pharr, Jakob, and Humphreys 2016), (Wikipedia: Radiometry n.d.), (StudySession n.d.), (Berkeley cs184 n.d., Radiometry & Photometry), (Wikipedia: Función de distribución de reflectancia bidireccional n.d.), (Wikipedia: Transmittance n.d.)

- <https://matmatch.com/learn/property/isotropy-anisotropy>
- https://pellacini.di.uniroma1.it/teaching/graphics17b/lectures/12_pathtracing.pdf

4. Integración de Monte Carlo

Como vimos en el capítulo anterior, la clave para conseguir una imagen en nuestro ray tracer es calcular la cantidad de luz en un punto de la escena. Para ello, necesitamos hallar la radiancia en dicha posición mediante la *rendering equation*. Sin embargo, es *muy* difícil resolverla; tanto computacional como analíticamente. Por ello, debemos atacar el problema desde otro punto de vista.

Las técnicas de Monte Carlo nos permitirán aproximar el valor que toman las integrales mediante una estimación. Utilizando muestreo aleatorio para evaluar puntos de una función, seremos capaces de obtener un resultado suficientemente bueno.

Una de las propiedades que hacen interesantes a este tipo de métodos es la **independencia del ratio de convergencia y la dimensionalidad del integrando**. Sin embargo, conseguir un mejor rendimiento tiene un precio a pagar. Dadas n muestras, la convergencia a la solución correcta tiene un orden de $\mathcal{O}(n^{-1/2}) = \mathcal{O}(\frac{1}{\sqrt{n}})$. Es decir, para reducir el error a la mitad, necesitaríamos 4 veces más muestras.

En este capítulo veremos los fundamentos de la integración de Monte Carlo, cómo muestrear distribuciones específicas y métodos para afinar el resultado final.

4.1. Repaso de probabilidad

Antes de comenzar a fondo, necesitaremos unas nociones de variable aleatoria para poder entender la integración de Monte Carlo, por lo que vamos a hacer un breve repaso.

Una **variable aleatoria** X (v.a.) es, esencialmente, una regla que asigna un valor numérico a cada posibilidad de proceso de azar. Formalmente, es una función definida en un espacio de probabilidad (Ω, \mathcal{A}, P) asociado a un experimento aleatorio:

$$X : \Omega \rightarrow \mathbb{R}$$

A Ω lo conocemos como espacio muestral (conjunto de todas las posibilidades), \mathcal{A} es una σ -álgebra de subconjuntos de Ω que refleja todas las posibilidades de eventos aleatorios, y P es una función probabilidad, que asigna a cada evento una probabilidad.

NOTE: no sé hasta qué punto debería meterme en la definición formal de variable aleatoria. Es una movida tremenda para poca cosa que necesitamos. De momento, voy con lo más interesante.

Una variable aleatoria X puede clasificarse atendiendo a cómo sea su rango $R_X = \{x \in \mathbb{R} \mid \exists \omega \in \Omega \text{ tal que } X(\omega) = x\}$: en discreta o continua.

4.1.1. Variables aleatorias discretas

Las v.a. discretas son aquellas cuyo rango es un conjunto discreto.

Para comprender mejor cómo funcionan, pongamos un ejemplo: Consideremos un experimento en el que lanzamos dos dados, anotando lo que sale en cada uno. Los posibles valores que toman serán

$$\begin{aligned} &\{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), \\ &\quad (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), \\ &\quad (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), \\ &\quad (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), \\ &\quad (5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), \\ &\quad (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6)\} \end{aligned}$$

Cada resultado tiene la misma probabilidad de ocurrir (claro está, si el dado no está truco). Como hay 36 posibilidades, la probabilidad de obtener un cierto valor es de $\frac{1}{36}$.

La v.a. X denotará la suma de los valores obtenidos en cada uno. Así, por ejemplo, si al lanzar los dados hemos obtenido $(1, 3)$, X tomará el valor 4. En total, X puede tomar

todos los valores comprendidos entre 2 y 12. Este sería el **espacio muestral**. Cada pareja no está asociada a un único valor de X . Por ejemplo, $(1, 2)$ suma lo mismo que $(2, 1)$. Esto nos lleva a preguntarnos... ¿Cuál es la probabilidad de que X adquiera un cierto valor?

La **función masa de probabilidad** nos permite conocer la probabilidad de que X tome un cierto valor x . Se denota por $P(X = x)$, aunque también usaremos $p_X(x)$ o directamente $p(x)$, cuando no haya lugar a dudas.

Nota(ción): Cuando X tenga una cierta función masa de probabilidad, escribiremos $X \sim p_X$

En este ejemplo, la probabilidad de que X tome el valor 4 es

$$\begin{aligned} P(X = 4) &= \sum \text{nº parejas que suman 4} \cdot \text{probabilidad de que salga la pareja} \\ &= 3 \cdot \frac{1}{36} = \frac{1}{12} \end{aligned}$$

Las parejas serían $(1, 3)$, $(2, 2)$ y $(3, 1)$.

Por definición, si el espacio muestral de X es $\Omega = \{x_1, \dots, x_n\}$, la función masa de probabilidad debe cumplir que

$$\sum_{i=1}^n P(X = x_i) = 1$$

Muchas veces nos interesará conocer la probabilidad de que X se quede por debajo de un cierto valor x (de hecho, podemos caracterizar distribuciones aleatorias gracias a esto). Para ello, usamos la **función de distribución**:

$$F_X(x) = P(X \leq x) = \sum_{\substack{k \in \Omega \\ k \leq x}} P(X = k)$$

Es una función continua por la derecha y monótona no decreciente. Además, se cumple que $0 \leq F_X \leq 1(x)$ y $\lim_{x \rightarrow -\infty} F_X = 0$, $\lim_{x \rightarrow \infty} F_X = 1$.

En nuestro ejemplo, si consideramos $x = 3$:

$$\begin{aligned}
F_X(x) &= \sum_{i=1}^3 P(X = i) = P(X = 1) + P(X = 2) + P(X = 3) \\
&= \frac{1}{36} + \frac{2}{36} + \frac{3}{36} = \frac{1}{12}
\end{aligned}$$

4.1.2. Variables aleatorias continuas

Este tipo de variables aleatorias tienen un rango no numerable; es decir, el conjunto de valores que puede tomar abarca un intervalo de números.

Un ejemplo podría ser la altura de una persona.

Si en las variables aleatorias discretas teníamos funciones masa de probabilidad, aquí definiremos las **funciones de densidad de probabilidad** (o simplemente, funciones de densidad). La idea es la misma: nos permite conocer la probabilidad de que nuestra variable aleatoria tome un cierto valor del espacio muestral.

Es importante mencionar que, aunque *la probabilidad de que la variable aleatoria tome un valor específico* es 0, ya que nos encontramos en un conjunto no numerable, sí que podemos calcular la probabilidad de que se encuentre entre dos valores. Por tanto, si la función de densidad es f_X , entonces

$$P(a \leq X \leq b) = \int_a^b f_X(x) dx$$

La función de densidad tiene dos características importantes:

1. f_X es no negativa; esto es, $f_X(x) \geq 0 \forall x \in \Omega$
2. f_X integra uno en todo el espacio muestral:

$$\int_{\Omega} f_X(x) = 1$$

Intuitivamente, podemos ver esta última propiedad como *si acumulamos todos los valores que puede tomar la variable aleatoria, la probabilidad de que te encuentres en el conjunto debe*

ser 1. Si tratamos con un conjunto de números reales, podemos escribir la integral como $\int_{-\infty}^{\infty} f_X(x) = 1$.

Una de las variables aleatorias que más juego nos darán en el futuro será la **v.a. con distribución uniforme en $[0, 1)$** . La denotaremos como ξ , y escribiremos $\xi \sim U([0, 1))$. La probabilidad de que ξ tome un valor es constante, por lo que podemos definir su función de densidad como

$$f(\xi) = \begin{cases} 1 & \text{si } \xi \in [0, 1) \\ 0 & \text{en otro caso.} \end{cases}$$

La probabilidad de ξ tome un valor entre dos elementos $a, b \in [0, 1)$ es

$$P(\xi \in [a, b]) = \int_a^b 1 dx = b - a$$

Como veremos más adelante, definiendo correctamente una función de densidad conseguiremos mejorar el rendimiento del path tracer.

La función de distribución $F_X(x)$ podemos definirla como:

$$F_X(x) = P(X \leq x) = \int_{-\infty}^x f_X(t) dt$$

Es decir, dado un x , ¿cuál sería la probabilidad de que X se quede por debajo de x ?

El Teorema Fundamental del Cálculo nos permite relacionar función de distribución y función de densidad directamente:

$$f_X(x) = \frac{dF_X(x)}{dx}$$

4.1.3. Esperanza y varianza de una variable aleatoria

La **esperanza de una variable aleatoria**, denotada $E[X]$, es una generalización de la media ponderada. Nos informa del *valor esperado* de dicha variable aleatoria.

En el caso de las variables discretas, se define como

$$E[X] = \sum_{x_i \in \Omega} x_i p_i$$

donde x_i son los posibles valores que puede tomar la v.a., y p_i la probabilidad asociada a cada uno de ellos; es decir, $p_i = P[X = x_i]$

Para una variable aleatoria continua real, la esperanza viene dada por

$$E[X] = \int_{-\infty}^{\infty} x f_X(x) dx$$

aunque, generalizando a una v.a. con espacio muestral Ω , la esperanza se puede generalizar como

$$E[X] = \int_{\Omega} x f_X(x) dx$$

Pongamos un par de ejemplos del cálculo de la esperanza. En el **ejemplo de las variables discretas**, la esperanza venía dada por

$$E[X] = \sum_{i=2}^{12} i \cdot P[X = i] = 2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + \dots + 12 \cdot \frac{1}{36} = 7$$

Para variables aleatorias uniformes en (a, b) (es decir, $X \sim U(a, b)$), la esperanza es

$$E[X] = \int_a^b x \cdot \frac{1}{b-a} dx = \frac{a+b}{2}$$

La esperanza tiene unas cuantas propiedades que nos resultarán muy útiles. Estas son:

- **Linealidad:**

- Si X, Y son dos v.a., $E[X + Y] = E[X] + E[Y]$
- Si a es una constante, X una v.a., entonces $E[aX] = aE[X]$
- Análogamente, para ciertas X_1, \dots, X_k , $E\left[\sum_{i=1}^k X_i\right] = \sum_{i=1}^k E[X_i]$

- Estas propiedades no necesitan que las variables aleatorias sean independientes. Este hecho será clave para las técnicas de Monte Carlo.
- La **Ley del estadístico inconsciente** (*Law of the unconscious statistician*, o LOTUS): dada una variable aleatoria X y una función medible g , la esperanza de $g(X)$ se puede calcular como

$$E[g(X)] = \int_{\Omega} g(x) f_X(x) dx$$

- La **Ley (fuerte) de los grandes números** nos dice que dada una muestra de n valores X_1, \dots, X_N de una variable aleatoria X con esperanza $E[X] = \mu$,

$$P \left[\lim_{N \rightarrow \infty} \frac{1}{n} \sum_{i=1}^N X_i = \mu \right] = 1$$

Usando que $\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i$, esta ley se suele escribir como

$$P \left[\lim_{N \rightarrow \infty} \bar{X}_N = \mu \right] = 1$$

Estas dos últimas propiedades resultarán claves en el desarrollo.

Será habitual encontrarnos con el problema de que no conocemos la distribución de una variable aleatoria Y . Sin embargo, si encontramos una transformación medible de una variable aleatoria X de forma que obtengamos Y (esto es, $\exists g$ función medible tal que $g(X) = Y$), entonces podemos calcular la esperanza de Y fácilmente. Esta propiedad hará que las variables aleatorias con distribución uniforme adquieran muchísima importancia. Generar números aleatorios en $[0, 1)$ es muy fácil, así **que obtendremos otras v.a.s a partir de ξ .**

Otra medida muy útil de una variable aleatoria es **la varianza**. Nos permitirá medir cómo de dispersa es la distribución con respecto a su media. La denotaremos como $Var[X]$, y se define como

$$Var[X] = E[(X - E[X])^2]$$

Si desarrollamos esta definición, podemos conseguir una expresión algo más agradable:

$$\begin{aligned}
 Var[X] &= E[(X - E[X])^2] = \\
 &= E[X^2 + E[X]^2 - 2XE[X]] = \\
 &= E[X^2] + E[X]^2 - 2E[X]E[X] = \\
 &= E[X^2] - E[X]^2
 \end{aligned}$$

Hemos usado que $E[E[X]] = E[X]$ y la linealidad de la esperanza.

Enunciemos un par de propiedades que tiene, similares a la de la esperanza:

- La varianza saca constantes al cuadrado: $Var[aX] = a^2 Var[X]$
- $Var[X + Y] = Var[X] + Var[Y] + 2Cov[X, Y]$, donde $Cov[X, Y]$ es la covarianza de X y Y .
 - En el caso en el que X e Y sean incorreladas (es decir, la covarianza es 0), $Var[X + Y] = Var[X] + Var[Y]$.

La varianza nos será útil a la hora de medir el error cometido por una estimación de Monte Carlo.

4.1.4. Estimadores

A veces, no podremos conocer de antemano el valor que toma un cierto parámetro de una distribución. Sin embargo, conocemos el tipo de distribución que nuestra variable aleatoria X sigue. Los estimadores nos proporcionarán una forma de calcular el posible valor de esos parámetros a partir de una muestra de X .

Sea X una variable aleatoria con distribución perteneciente a una familia de distribuciones paramétricas $X \sim F \in \{F(\theta) \mid \theta \in \Theta\}$. Θ es el conjunto de valores que puede tomar el parámetro. Buscamos una forma de determinar el valor de θ .

Diremos que $T(X_1, \dots, X_N)$ es **un estimador de θ** si T toma valores en Θ .

A los estimadores de un parámetro los solemos denotar con $\hat{\theta}$.

Como vemos, la definición no es muy restrictiva. Únicamente le estamos pidiendo a la función de la muestra que pueda tomar valores viables para la distribución.

Se dice que un estimador $T(X_1, \dots, X_N)$ es **insesgado** (o centrado en el parámetro θ) si

$$E[T(X_1, \dots, X_N)] = \theta \quad \forall \theta \in \Theta$$

Naturalmente, decimos que un estimador $T(X_1, \dots, X_N)$ está **sesgado** si $E[T(X_1, \dots, X_N)] \neq \theta$.

4.2. El estimador de Monte Carlo

Tras este breve repaso de probabilidad, estamos en condiciones de definir el estimador de Monte Carlo. Primero, vamos con su versión más sencilla.

Los estimadores de Monte Carlo nos permiten hallar la esperanza de una variable aleatoria, digamos, Y , sin necesidad de calcular explícitamente su valor. Para ello, tomamos unas cuantas muestras Y_1, \dots, Y_N que sigan la misma distribución que Y con media μ . Entonces, consideramos el estimador de μ (Owen 2013):

$$\hat{\mu}_N = \frac{1}{N} \sum_{i=1}^N Y_i \quad (4.1)$$

Haciendo la esperanza de este estimador, vemos que

$$\begin{aligned} E[\hat{\mu}_N] &= E \left[\frac{1}{N} \sum_{i=1}^N Y_i \right] = \frac{1}{N} E \left[\sum_{i=1}^N Y_i \right] \\ &= \frac{1}{N} \sum_{i=1}^N E[Y_i] = \frac{1}{N} \sum_{i=1}^N \mu = \\ &= \mu \end{aligned}$$

Por lo que el estimador es insesgado.

Generalmente nos encontraremos en la situación en la que $Y = f(X)$, donde X sigue una distribución con función de densidad $p_X(x)$, y $f : S \rightarrow \mathbb{R}$. En ese caso, sabemos que la esperanza de Y se puede calcular como

$$\mu = E[Y] = E[f(X)] = \int_S f(x)p_X(x)dx$$

Lo que estamos buscando es calcular $\int_S f(x)dx$. Entonces, ¿qué ocurre si intentamos compensar en [4.1] con la función de densidad?

$$\begin{aligned} E \left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)} \right] &= \frac{1}{N} \sum_{i=1}^N E \left[\frac{f(X_i)}{p_X(X_i)} \right] = \\ &= \frac{1}{N} \sum_{i=1}^N \left(\int_S \frac{f(x)}{p_X(x)} p_X(x) dx \right) = \\ &= \frac{1}{N} N \int_S f(x) dx = \\ &= \int_S f(x) dx \end{aligned}$$

¡Genial! Esto nos da una forma de calcular la integral de una función usando muestras de variables aleatorias con cierta distribución. Llamaremos al estimador de Monte Carlo

$$\hat{I}_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)} \quad (4.2)$$

Es importante mencionar que $p_X(x)$ debe ser distinto de 0 cuando f también lo sea.

Nota(ción): si te preguntas por qué lo llamamos \hat{I}_N , piensa que queremos calcular la integral $I = \int_S f(x)dx$. Para ello, usamos el estimador \hat{I} , y marcamos explícitamente que usamos N muestras.

Podemos particularizar el caso en el que nuestras muestras X_i sigan una distribución uniforme en $[a, b]$. Si eso ocurre, su función de densidad es $p_X(x) = \frac{1}{b-a}$, así que podemos simplificar un poco [4.2]:

$$\hat{I}_N = \frac{b-a}{N} \sum_{i=1}^N f(X_i)$$

Elegir correctamente la función de densidad p_X será clave. Si conseguimos escogerla debidamente, reduciremos en gran medida el error que genera el estimador. Esto es lo que se conoce como *importance sampling*.

Puesto que la varianza del estimador nos dará información sobre el error que genera, vamos a calcular $Var[\hat{I}_N]$. Para ello, usamos las propiedades que vimos en la *sección anterior*:

$$\begin{aligned} Var[\hat{I}_N] &= Var \left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)} \right] = \\ &= \frac{1}{N^2} Var \left[\sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)} \right] = \\ &= \frac{1}{N^2} N Var \left[\frac{f(X)}{p_X(X)} \right] = \\ &= \frac{1}{N} Var \left[\frac{f(X)}{p_X(X)} \right] \end{aligned} \tag{4.3}$$

es decir, la varianza del estimador es inversamente proporcional al número de muestras N .

La desviación estándar es

$$\sqrt{Var[\hat{I}_N]} = \frac{\sqrt{Var \left[\frac{f(X)}{p_X(X)} \right]}}{\sqrt{N}}$$

así que, como adelantamos al inicio del capítulo, la estimación tiene un error del orden $\mathcal{O}(N^{-1/2})$. Esto nos dice que, para reducir el error a la mitad, debemos tomar 4 veces más muestras.

Pongamos un ejemplo de estimador de Monte Carlo para una caja de dimensiones $[x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$. Si queremos estimar la integral de la función $f : \mathbb{R}^3 \rightarrow \mathbb{R}$

$$\int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{z_0}^{z_1} f(x, y, z) dx dy dz$$

mediante una variable aleatoria $X \sim U([x_0, x_1] \times [y_0, y_1] \times [z_0, z_1])$ con función de densidad $p(x, y, z) = \frac{1}{x_1 - x_0} \frac{1}{y_1 - y_0} \frac{1}{z_1 - z_0}$, tomamos el estimador

$$\hat{I}_N = \frac{1}{(x_1 - x_0) \cdot (y_1 - y_0) \cdot (z_1 - z_0)} \sum_{i=1}^N f(X_i)$$

Otro ejemplo clásico de estimador de Monte Carlo es calcular el valor de π . Se puede hallar integrando una función que valga 1 en el interior de la circunferencia de radio unidad y 0 en el exterior:

$$f = \begin{cases} 1 & \text{si } x^2 + y^2 \leq 1 \\ 0 & \text{en otro caso} \end{cases} \Rightarrow \pi = \int_{-1}^1 \int_{-1}^1 f(x, y) dx dy$$

Para usar el estimador de [4.2], necesitamos saber la probabilidad de obtener un punto dentro de la circunferencia.

Bien, consideremos que una circunferencia de radio r se encuentra inscrita en un cuadrado. El área de la circunferencia es πr^2 , mientras que la del cuadrado es $(2r)^2 = 4r^2$. Por tanto, la probabilidad de obtener un punto dentro de la circunferencia es $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$. Podemos tomar $p(x, y) = \frac{1}{4}$, de forma que

$$\pi \approx \frac{4}{N} \sum_{i=1}^N f(x_i, y_i), \text{ con } (x_i, y_i) \sim U([-1, 1] \times [-1, 1])$$

4.3. Escogiendo puntos aleatorios

Una de las partes clave del estimador de Monte Carlo [4.2] es saber escoger la función de densidad p_X correctamente. En esta sección, veremos algunos métodos para conseguir distribuciones específicas partiendo de funciones de densidad sencillas.

4.3.1. Método de la transformada inversa

En resumen: Para conseguir una muestra de una distribución específica F_X :

1. Generar un número aleatorio $\xi \sim U(0, 1)$.
2. Hallar la inversa de la función de distribución deseada F_X , denotada $F_X^{-1}(x)$.
3. Calcular $F_X^{-1}(\xi) = X$.

Este método nos permite conseguir muestras de cualquier distribución continua a partir de variables aleatorias uniformes, siempre que se conozca la inversa de la función de distribución.

Sea X una variable aleatoria con función de distribución F_X ¹. Queremos buscar una transformación $T : [0, 1] \rightarrow \mathbb{R}$ tal que $T(\xi) \stackrel{d}{=} X$, siendo ξ una v.a. uniformemente distribuida. Para que esto se cumpla, se debe dar

$$\begin{aligned} F_X(x) &= P[X < x] = \\ &= P[T(\xi) < x] = \\ &= P(\xi < T^{-1}(x)) = \\ &= T^{-1}(x) \end{aligned}$$

Este último paso se debe a que, como ξ es uniforme en $(0, 1)$, $P[\xi < x] = x$. Es decir, hemos obtenido que F_X es la inversa de T .

TODO: dibujo similar a [este: p.52](#)

Como ejemplo, vamos a muestrear la función $f(x) = x^2$, $x \in [0, 2]$.

Primero, normalizamos esta función para obtener una función de densidad $p_X(x)$. Es decir, buscamos $p_X(x) = cf(x)$ tal que

¹En su defecto, si tenemos una función de densidad f_X , podemos hallar la función de distribución haciendo $F_X(x) = P[X < x] = \int_{x_{\min}}^x f_X(t)dt$.

$$\begin{aligned}
1 &= \int_0^2 p_X(x) dx = \int_0^2 c f(x) dx = c \int_0^2 f(x) dx = \\
&= \frac{cx^3}{3} \Big|_0^2 = \frac{8c}{3} \\
&\Rightarrow c = \frac{3}{8} \\
&\Rightarrow p_X(x) = \frac{3x^2}{8}
\end{aligned}$$

A continuación, integramos la función de densidad para obtener la de distribución F_X :

$$F_X(x) = \int_0^x p_X(x) dx = \int_0^x \frac{3x^2}{8} = \frac{x^3}{8}$$

Solo nos queda conseguir la muestra. Para ello,

$$\begin{aligned}
\xi &= F_X(x) = \frac{x^3}{8} \quad \Leftrightarrow \\
x &= \sqrt[3]{8\xi}
\end{aligned}$$

Sacando un número aleatorio ξ , y pasándolo por la función obtenida, conseguimos un elemento con distribución $f(x)$.

4.3.2. Método del rechazo

En resumen: Para conseguir una muestra de una variable aleatoria X con función de densidad p_X :

1. Obtener una muestra y de Y , y otra ξ de $U(0, 1)$.
2. Comprobar si $\xi < \frac{p_X(y)}{Mp_Y(y)}$. Si es así, aceptarla. Si no, sacar otra muestra.

El método anterior presenta principalmente dos problemas:

1. No siempre es posible integrar una función para hallar su función de densidad.
2. La inversa de la función de distribución, F_X^{-1} no tiene por qué existir.

Como alternativa, podemos usar este método (en inglés, *rejection method*). Para ello, necesitamos una variable aleatoria Y con función de densidad $p_Y(y)$. El objetivo es conseguir una muestra de X con función de densidad $p_X(x)$.

La idea principal es aceptar una muestra de Y con probabilidad p_X/Mp_Y , con $1 < M < \infty$. En esencia, estamos jugando a los dardos: si la muestra de y que hemos obtenido se queda por debajo de la gráfica de la función $Mp_Y < p_X$, estaremos obteniendo una de p_X .

TODO dibujo de la gráfica $\frac{p_X(y)}{Mp_Y(y)}$.

¿Quizás haga falta una demostración también? No estoy satisfecho con este apartado ahora mismo. Necesita trabajo.

El algoritmo consiste en:

1. Obtener una muestra de Y , denotada y , y otra de $U(0, 1)$, llamada ξ .
2. Comprobar si $\xi < \frac{p_X(y)}{Mp_Y(y)}$.
 1. Si se cumple, se acepta y como muestra de p_X
 2. En caso contrario, se rechaza y y se vuelve al paso 1.

4.4. Importance sampling

Si recordamos la varianza del estimador de Monte Carlo [4.3],

$$\text{Var}[\hat{I}_N] = \frac{1}{N} \text{Var} \left[\frac{f(X)}{p_X(X)} \right]$$

podemos ver que depende de dos factores: el número de muestras N y la varianza de $\text{Var} \left[\frac{f(X)}{p_X(X)} \right]$. Aumentar el número de muestras haría que la varianza decrezca. Sin embargo, alcanzaríamos un punto de retornos reducidos. Por tanto, vamos a centrarnos ahora en el segundo término.

En esencia, la varianza de $\text{Var} \left[\frac{f(X)}{p_X(X)} \right]$ decrecerá cuanto más cercana sea la función de probabilidad p_X a la función $f(X)$.

Supongamos que f es proporcional a p_X . Esto es, existe un s tal que $f(x) = sp_X(x)$. Como p_X debe integrar uno, podemos calcular el valor de s :

$$\begin{aligned} \int_S p_X(x) dx &= \int_S s f(x) dx = 1 \quad \Leftrightarrow \\ s &= \frac{1}{\int_S f(x) dx} \end{aligned}$$

Y entonces, se tendría que

$$\begin{aligned} Var \left[\frac{f(X)}{p_X(X)} \right] &= Var \left[\frac{f(X)}{s f(X)} \right] = \\ &= Var \left[\frac{1}{s} \right] = \\ &= 0 \end{aligned}$$

En la práctica, esto es inviable. El problema que queremos resolver es calcular la integral de f . Y para sacar s , necesitaríamos el valor de la integral de f . ¡Estamos dando vueltas!

Por fortuna, hay algoritmos que son capaces de proporcionar la constante s sin necesidad de calcular la integral. Uno de los más conocidos es **Metropolis-Hastings**, el cual se basa en cadenas de Markov de Monte Carlo.

En este trabajo nos centraremos en buscar funciones de densidad p_X que se aproximen a f lo más fielmente posible, dentro del contexto del transporte de luz.

4.5. Multiple importance sampling

<https://graphics.stanford.edu/courses/cs348b-03/papers/veach-chapter9.pdf>

Referencias

(Shirley and Morley 2003), (Pharr, Jakob, and Humphreys 2016), (Owen 2013), (Berkeley cs184 n.d., Monte Carlo Integration), (Wikipedia: Rendering equation n.d.), (Wikipedia:

Variable aleatoria n.d.), (Wikipedia: Distribución de probabilidad n.d.), (Wikipedia: Función de probabilidad n.d.), (Wikipedia: Expected value n.d.), (Galvin n.d.), (Wikipedia: Probability density function n.d.), (Wikipedia: Estimador n.d.), (Wikipedia: Método de la transformada inversa n.d.), (Wikipedia: Rejection sampling n.d.),

- (*berkeley-cs184*) <https://cs184.eecs.berkeley.edu/public/sp22/lectures/lec-12-monte-carlo-integration/lec-12-monte-carlo-integration.pdf>
- Gems I, p.284.
- https://pellacini.di.uniroma1.it/teaching/graphics17b/lectures/12_pathtracing.pdf
- Apuntes de inferencia estadística (cómo cito este tipo de fuentes??)
- https://www.wikiwand.com/en/Metropolis%E2%80%93Hastings_algorithm

5. ¡Construyamos un path tracer!

Ahora que hemos introducido toda la teoría necesaria, es hora de ponernos manos a la obra. En este capítulo, vamos a escoger una serie de herramientas y haremos una pequeña implementación de un motor de path tracing en tiempo real.

La implementación estará basada en Vulkan, junto al pequeño framework de nvpro-samples. El motor mantendrá el mismo espíritu que la serie de (Shirley 2020a), Ray Tracing In One Weekend.

Le pondremos especial atención a los conceptos claves. Vulkan tiende a crear código muy verboso, por lo que se documentarán únicamente las partes más importantes.

5.1. Requisitos de *real time ray tracing*

Como es natural, el tiempo es una limitación enorme para cualquier programa en tiempo real. Mientras que en un *offline renderer* disponemos de un tiempo muy considerable por frame (hablamos de varios segundos), en un programa en tiempo real necesitamos que un frame salga en 16 milisegundos o menos. Este concepto se suele denominar *frame budget*: la cantidad de tiempo que disponemos para un frame.

Nota: cuando hablamos del tiempo disponible para un frame, solemos hablar en milisegundos (ms) o frames por segundo (FPS). Para que un motor vaya suficientemente fluido, necesitaremos que el motor corra a un mínimo de 30 FPS (que equivalen a 33 ms por frame). Hoy en día, debido al avance del área en campos como los videojuegos, el estándar se está convirtiendo en 60 FPS (16 ms/frame).

Las nociones anteriores no distinguen entre un motor en tiempo real y *offline*. Como es natural, necesitaremos introducir unos pocos conceptos más para llevarlo a tiempo real.

Además, existe una serie de requisitos hardware que debemos cumplir para que un motor en tiempo real con ray tracing funcione.

5.1.1. Arquitecturas de gráficas

NOTE: sería interesante enlazarlo con la sección de rendimiento.

TODO: Esta [página](#) es maravillosa *chef kiss*

El requisito más importante de todos es la gráfica. Para ser capaces de realizar cálculos de ray tracing en tiempo real, necesitaremos una arquitectura moderna con núcleos dedicados a este tipo de cálculos¹.

A día 17 de abril de 2022, para correr ray tracing en tiempo real, se necesita alguna de las siguientes tarjetas gráficas:

Arquitectura	Fabricante	Modelos de gráficas
Turing	Nvidia	RTX 2060, RTX 2060 Super, RTX 2070, RTX 2070 Super, RTX 2080, RTX 2080 Super, RTX 2080 Ti, RTX Titan
Ampere	Nvidia	RTX 3050, RTX 3060, RTX 3060 Ti, RTX 3070, RTX 3070 Ti, RTX 3080, RTX 3080 Ti, RTX 3090, RTX 3090 Ti
RDNA2 (Navi 2X, Big Navi)	AMD	RX 6400, RX 6500 XT, RX 6600, RX 6600 XT, RX 6700 XT, RX 6800, RX 6800 XT, RX 6900 XT
Arc Alchemist	Intel	<i>No revelado aún</i>

Solo se han incluido las gráficas de escritorio de consumidor.

Para este trabajo se ha utilizado una **RTX 2070 Super**. En el capítulo de análisis del rendimiento se hablará con mayor profundidad de este apartado.

¹Esto no es del todo cierto. Aunque generalmente suelen ser excepciones debido al coste computacional de RT en tiempo real, existen algunas implementaciones que son capaces de correrlo por software. Notablemente, el motor de Crytek, CryEngine, es capaz de mover ray tracing basado en hardware y en software ([Crytek n.d.](#))

5.1.2. Frameworks y API de ray tracing en tiempo real

Una vez hemos cumplido los requisitos de hardware, es hora de escoger los frameworks de trabajo.

Las API de gráficos están empezando a adaptarse a los requisitos del tiempo real, por lo que cambian frecuentemente. La mayoría adquirieron las directivas necesarias muy recientemente. Aun así, son lo suficientemente sólidas para que se pueda usar en aplicaciones empresariales de gran envergadura.

Esta es una lista de las API disponibles con capacidades de Ray Tracing disponibles para, al menos, la arquitectura Turing:

- Vulkan (los bindings de ray tracing se denominan KHR).
- Microsoft DirectX Ray Tracing (DXR), una extensión de DirectX 12.
- Nvidia OptiX.

De momento, no hay mucho donde elegir.

OptiX es la API más vieja de todas. Su primera versión salió en 2009, mientras que la última estable es de 2021. Tradicionalmente se ha usado para offline renderers, y no tiene un especial interés para este trabajo estando las otras dos disponibles.

Tanto DXR como Vulkan son los candidatos más sólidos. DXR salió en 2018, con la llegada de Turing. Es un par de años más reciente que Vulkan KHR. Cualquiera de las dos cumpliría su cometido de forma exitosa. Sin embargo, para este trabajo, **hemos escogido Vulkan** por los siguientes motivos:

- DirectX 12 está destinado principalmente a plataformas de Microsoft. Es decir, está pensado para sistemas operativos Windows 10 o mayor ².
- Vulkan está apoyado principalmente por AMD. Esto sigue las líneas de la su política de empresa de apoyar el código abierto. Además, resulta más sencillo exportarlo a otros sistemas operativos.

²Afortunadamente, esto tampoco es completamente cierto. La compañía desarrolladora y distribuidora de videojuegos Valve Corporation ha creado una pieza de software fascinante: [Proton](#). Proton utiliza Wine para emular software en Linux que solo puede correr en plataformas Windows. La versión 2.5 añadió soporte para traducción de bindings de DXR a KHR, lo que permite utilizar DirectX12 ray tracing en sistemas basados en Linux. El motivo de este software es expandir el mercado de videojuegos disponibles en su consola, la Steam Deck.

Ambas API se comportan de manera muy similar, y no existe una gran diferencia entre ellas; tanto en rendimiento como en complejidad de desarrollo. Actualmente el proyecto solo compila en Windows 10 o mayor, por lo que estos dos puntos no resultan especialmente relevantes para el trabajo.

5.2. Setup del proyecto

Un proyecto de Vulkan necesita una cantidad de código inicial considerable. Para acelerar este trámite y partir de una base más sólida, se ha decidido usar un pequeño framework de trabajo de Nvidia llamado [nvpro-samples](#).

Esta serie de repositorios contienen proyectos de ray tracing de Nvidia con fines didácticos. Nosotros usaremos [vk_raytracing_tutorial_KHR](#), pues ejemplifica cómo añadir ray tracing en tiempo real a un proyecto de Vulkan.

Nuestro repositorio utiliza los citados anteriormente para compilar su proyecto. El Makefile es una modificación del que se usa para ejecutar los ejemplos de Nvidia. Por defecto, ejecuta una aplicación muy simple que muestra un cubo mediante rasterización.

5.3. Compilación

Las dependencias necesarias para compilarlo son:

1. CMake.
2. Un driver de Nvidia compatible con la extensión [VK_KHR_ray_tracing_pipeline](#).
3. El SDK de Vulkan, versión 1.2.161 o mayor.

La parte inicial del desarrollo consiste en adaptar Vulkan para usar la extensión de ray tracing, extrayendo la información de la gráfica y cargando correspondientemente el dispositivo.

Para compilarlo, ejecuta los siguientes comandos:

```
1 git clone git@github.com:Asmilex/Raytracing.git
2 cd .\application\vulkan_ray_tracing\
```

```
3 mkdir build
4 cd build
5 cmake ..
```

Si todo funciona correctamente, debería generarse un binario en `./application/bin_x64/Debug` llamado `asmiray.exe`.

5.4. Estructuras de aceleración

El principal coste de ray tracing es el cálculo de las intersecciones con objetos; hasta un 95% del tiempo de ejecución total ([Scratchapixel n.d.](#)). Reducir el número de test de intersección es clave.

Las **estructuras de aceleración** son una forma de representar la geometría de la escena. Aunque hay varios tipos diferentes, en esencia, engloban a un objeto o varios en una estructura con la que resulta más eficiente hacer test de intersección. Son similares a los grafos de escena de un rasterizador.

Uno de los tipos más comunes es la **Bounding Volume Hierarchy (BVH)**. Fue una técnica desarrollada por Kay y Kajilla en 1986. En esencia, este método encierra un objeto en una caja (lo que se denomina una **bounding box**), de forma que el test de intersección principal se hace con la caja y no con la geometría. Si un rayo impacta en la *bounding box*, entonces se pasa a testear la geometría.

Se puede repetir esta idea repetidamente, de forma que agrupemos varias *bounding boxes*. Así, creamos una jerarquía de objetos –como si nodos de un árbol se trataran–. A esta jerarquía es a la que llamamos BVH.

Es importante crear buenas divisiones de los objetos en la BVH. Cuanto más compacta sea una BVH, más eficiente será el test de intersección.

Una forma habitual de crear la BVH es mediante la división del espacio en una rejilla. Esta técnica se llama **Axis-Aligned Bounding Box (AABB)**. Usualmente se usa el método del *slab* (también introducido por Kay y Kajilla). Se divide el espacio en una caja n-dimensional alineada con los ejes, de forma que podemos verla como $[x_0, x_1] \times [y_0, y_1] \times [z_0, z_1] \times \dots$. De esta forma, comprobar si un rayo impacta en una *bounding box*

es tan sencillo como comprobar que está dentro del intervalo. Este es el método que se ha usado en Ray Tracing in One Weekend.

Vulkan gestiona las estructuras de aceleración dividiéndolas en dos partes: **Top-Level Acceleration Structure** (TLAS) y **Bottom-Level Acceleration Structure** (BLAS).

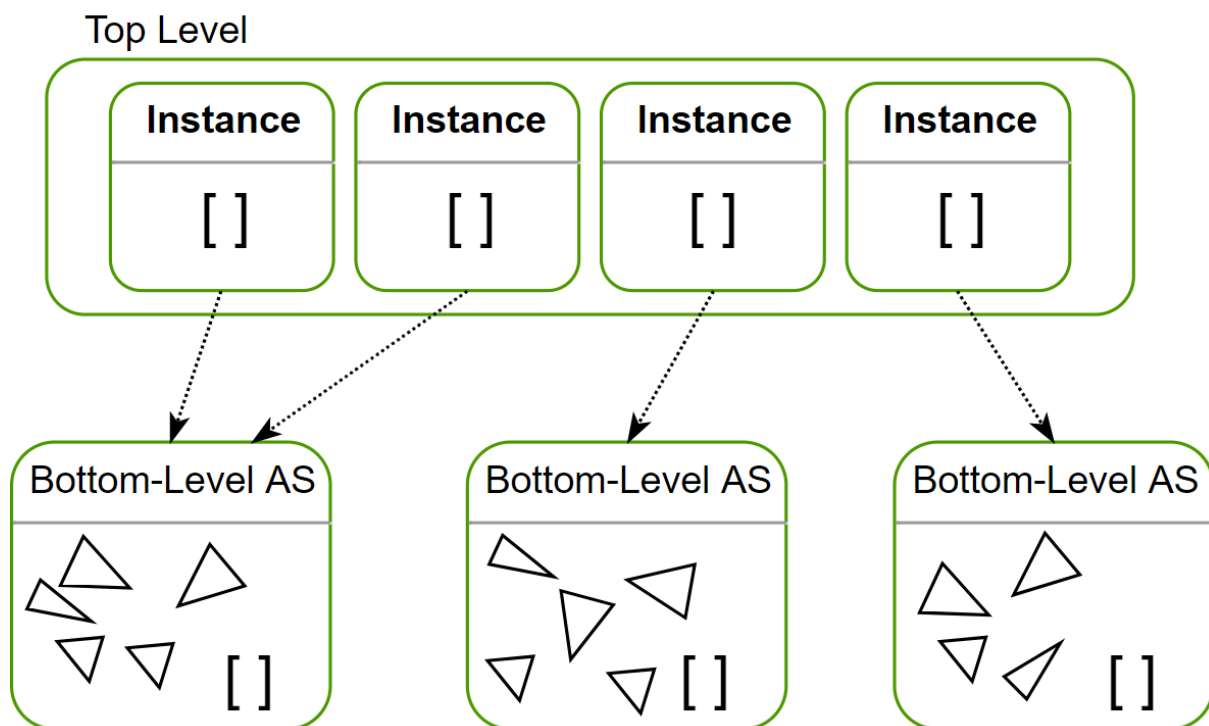


Figure 5.1.: La TLAS guarda información de las instancias de un objeto, así como una referencia a BLAS que contiene la geometría correspondiente. Fuente: Nvidia

TODO: Deberíamos cambiar esa foto por otra propia.

5.4.1. Bottom-Level Acceleration Structure (BLAS)

Las Bottom-Level Acceleration Structure almacenan la geometría de un objeto individual; esto es, los vértices y los índices de los triángulos, además de una AABB que la encapsula.

sula.

Pueden almacenar varios modelos, puesto que alojan uno o más buffers de vértices junto a sus matrices de transformación. Si un modelo se instancia varias veces *dentro de la misma BLAS*, la geometría se duplica. Esto se hace para mejorar el rendimiento.

Como regla general, cuantas menos BLAS, mejor.

El código correspondiente a la creación de la BLAS en el programa es el siguiente:

```
1 void HelloVulkan::createBottomLevelAS() {
2     // BLAS - guardar cada primitiva en una geometría
3
4     std::vector<nvk::RaytracingBuilderKHR::BlasInput> allBlas;
5     allBlas.reserve(m_objModel.size());
6
7     for (const auto& obj: m_objModel) {
8         auto blas = objectToVkGeometryKHR(obj);
9
10        // Podríamos añadir más geometrías en cada BLAS.
11        // De momento, solo una.
12        allBlas.emplace_back(blas);
13    }
14
15    m_rtBuilder.buildBlas(
16        allBlas,
17        VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR
18    );
19 }
```

5.4.2. Top-Level Acceleration Structure (TLAS)

Las Top-Level Acceleration Structures almacenan las instancias de los objetos, cada una con su matriz de transformación y referencia a la BLAS correspondiente.

Además, guardan información sobre el *shading*. Así, los shaders pueden relacionar la geometría intersecada y el material de dicho objeto. En esta última parte jugará un papel fundamental la **Shader Binding Table**.

En el programa hacemos lo siguiente para construir la TLAS:

```
1 void HelloVulkan::createTopLevelAS() {
2     std::vector<VkAccelerationStructureInstanceKHR> tlas;
3     tlas.reserve(m_instances.size());
4
5     for (const HelloVulkan::ObjInstance& inst: m_instances) {
6         VkAccelerationStructureInstanceKHR rayInst{};
7
8         // Posición de la instancia
9         rayInst.transform = nvvk::toTransformMatrixKHR(inst.transform);
10
11         rayInst.instanceCustomIndex = inst.objIndex;
12
13         // returns the acceleration structure device address of the blasId.
14         // The id correspond to the created BLAS in buildBlas.
15         rayInst.accelerationStructureReference = m_rtBuilder.
16             getBlasDeviceAddress(inst.objIndex);
17
18         rayInst.flags =
19             VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
20         rayInst.mask = 0xFF; // Solo registramos hit si rayMask & instance
21             .mask != 0
22         rayInst.instanceShaderBindingTableRecordOffset = 0; // Usaremos el
23             mismo hit group para todos los objetos
24
25         tlas.emplace_back(rayInst);
26     }
27
28     m_rtBuilder.buildTlas(
29         tlas,
30         VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR
31     );
32 }
```

5.5. La ray tracing pipeline

5.5.1. Descriptores y conceptos básicos

Primero, debemos introducir unas nociones básicas de Vulkan sobre cómo gestiona la información que se pasa a los shaders.

Un *resource descriptor* (usualmente lo abreviaremos como descriptor) es una forma de cargar recursos como buffers o imágenes para que la tarjeta gráfica los pueda utilizar; concretamente, los shaders. El *descriptor layout* especifica el tipo de recurso que va a ser accedido. Finalmente, el *descriptor set* determina el buffer o imagen que se va a asociar al descriptor. Este set es el que se utiliza en los **drawing commands**. Un **pipeline** es una secuencia de operaciones que reciben una geometría y sus texturas, y la transforma en unos pixels.

Si necesitas más información, todos estos conceptos aparecen desarrollados extensamente en ([Overvoorde n.d.](#))

Tradicionalmente, en rasterización se utiliza un descriptor set por tipo de material, y consecuentemente, un pipeline por cada tipo. En ray tracing esto no es posible, puesto que no se sabe qué material se va a usar: un rayo puede impactar *cualquier* material presente en la escena, lo cual invocaría un shader específico. Debido a esto, empaquetaremos todos los recursos en un único set de descriptores.

5.5.2. La Shader binding table

Para solucionar esto, vamos a crear la **Shader Binding Table** (SBT). Esta estructura permitirá cargar el shader correspondiente dependiendo de dónde impacte un rayo.

Para cargar esta estructura, se debe hacer lo siguiente:

1. Cargar y compilar cada shader en un `VkShaderModule`.
2. Juntar los cada `VkShaderModule` en un array `VkPipelineShaderStageCreateInfo`.
3. Crear un array de `VkRayTracingShaderGroupCreateInfoKHR`. Cada elemento se convertirá al final en una entrada de la Shader Binding Table.
4. Compilar los dos arrays anteriores más un pipeline layout para generar un `vkCreateRayTracingPipelineKHR`.

5. Conseguir los *handlers* de los shaders usando `vkGetRayTracingShaderGroupHandlesKHR`.
6. Alojarse un buffer con el bit `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` y copiar los *handlers*.

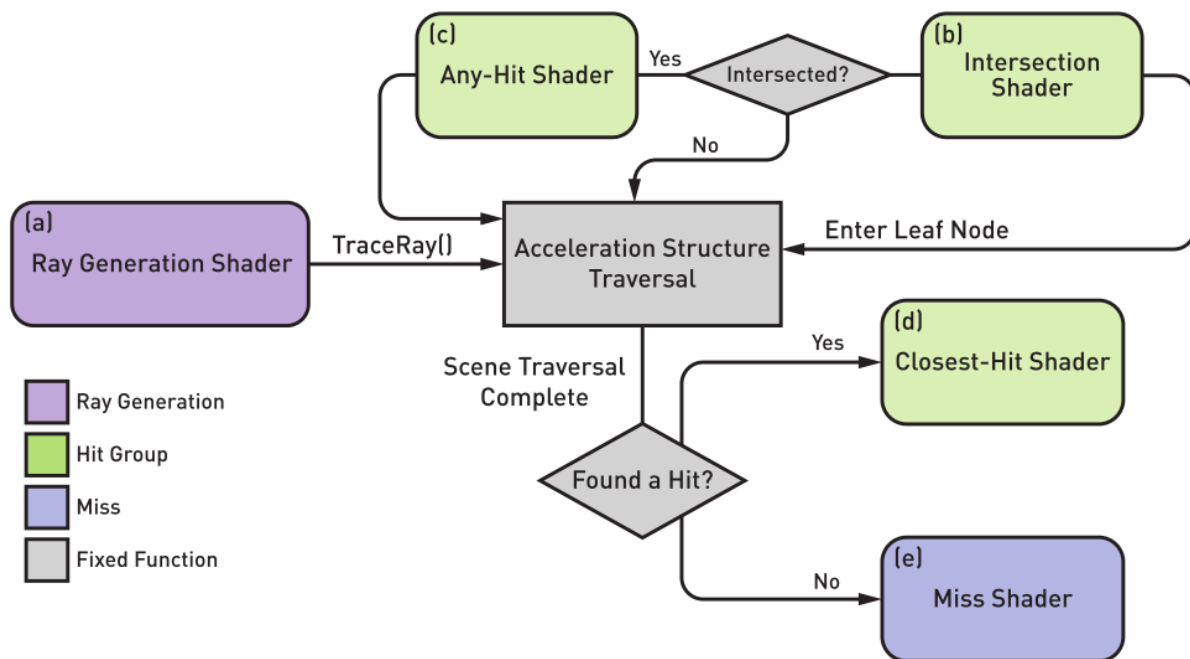


Figure 5.2.: La Shader Binding Table permite seleccionar un tipo de shader dependiendo del objeto en el que se impacte. Para ello, se genera un rayo desde el shader `raygen`, el cual viaja a través de la Acceleration Structure. Dependiendo de dónde impacte, se utiliza un `closest hit`, `any hit`, o `miss` shaders. Fuente: Nvidia

Cada entrada de la SBT contiene un handler y una serie de parámetros embebidos. A esto se le conoce como **Shader Record**. Estos records se clasifican en:

- **Ray generation record:** contiene el handler del ray generation shader.
- **Hit group record:** se encargan de los handlers del closest hit, anyhit (opcional), e intersection (opcional).

- **Miss group record:** se encarga del miss shader.
- **Callable group record.**

Una de las partes más difíciles de la SBT es saber cómo se relacionan record y geometría. Es decir, cuando un rayo impacta en una geometría, ¿a qué record de la SBT llamamos? Esto se determina mediante los parámetros de la instancia, la llamada a *trace rays*, y el orden de la geometría en la BLAS.

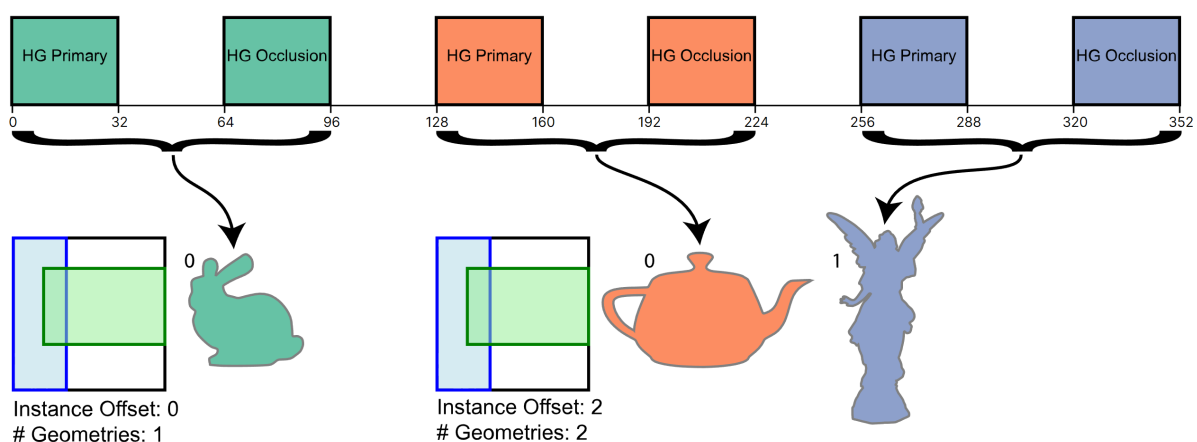


Figure 5.3.: Fuente: <https://www.willusher.io/>

5.5.2.1. Cálculo de la entrada de la SBT

El principal problema es el cálculo del índice en los hit groups.

Llamemos al índice de cada instancia de una geometría en la BLAS \mathbb{G}_{ID} . A cada instancia se le puede asignar un desplazamiento con respecto a la SBT (\mathbb{I}_{offset}) desde donde empieza la subtabla de hit groups.

TODO: esto se deja temporal de momento. No sé hasta qué punto me conviene poner todo esto. Lo veo importante, pero no sé si *tan* importante. El recurso que es-

toy usando es <https://www.willusher.io/graphics/2019/11/20/the-sbt-three-ways>, por si al final decidimos escribirlo.

5.5.3. Tipos de shaders

El pipeline soporta varios tipos de shaders diferentes que cubren la funcionalidad esencial de un ray tracer:

- **Ray generation shader:** es el punto de inicio del viaje de un rayo. Calcula punto de inicio y procesa el resultado final. Idealmente, solo se invocan rayos desde aquí. Se suele invocar
- **Closest hit shader:** este shader se ejecuta cuando un rayo impacta en una geometría por primera vez. Se pueden trazar rayos recursivamente desde aquí (por ejemplo, para calcular oclusión ambiental).
- **Any-hit shader:** similar al closest hit, pero invocado en cada intersección del camino del rayo que cumpla $t \in [t_{min}, t_{max})$. Es comúnmente utilizado en los cálculos de transparencias (*alpha-testing*).
- **Miss shader:** si el rayo no choca con ninguna geometría –pega con el infinito–, se ejecuta este shader. Normalmente, añade una pequeña contribución ambiental al rayo.
- **Intersection shader:** este shader es algo diferente al resto. Su función es calcular el punto de impacto de un rayo con una geometría. Por defecto se utiliza un test triángulo - rayo. En nuestro path tracer lo dejaremos por defecto, pero podríamos definir algún método como los que vimos en la sección [intersecciones rayo - objeto](#).

Este es el código de los shaders del path tracer: [Raygen](#), [Closest hit](#), [Miss](#), [Any-hit](#).

Existe otro tipo de shader adicional denominado **callable shader**. Este es un shader que se invoca desde otro shader. Por ejemplo, un shader de intersección puede invocar a un shader de oclusión. Otro ejemplo sería un closest hit que reemplaza un bloque if-else por un shader para hacer cálculos de iluminación. Este tipo de shaders no se han implementado en el path tracer, pero se podrían añadir con un poco de trabajo.

5.5.4. Traspaso de información entre shaders

En ray tracing, los shaders por sí solos no pueden realizar todos los cálculos necesarios. Por ello, necesitaremos enviar información de uno a otro. Tenemos diferentes mecanismos para conseguirlo:

El primero de ellos son las **push constants**. Estas son variables que se pueden enviar a los shaders (es decir, de CPU a GPU), pero que no se pueden modificar. Únicamente podemos mandar un pequeño número de variables, el cual se puede consultar mediante `VkPhysicalDeviceLimits.maxPushConstantSize`.

Nuestro path tracer tiene implementado actualmente (19 de abril de 2022) las siguientes constantes:

```
1 struct PushConstantRay {
2     vec4  clearColor;    // Color ambiental
3     vec3  lightPosition;
4     float lightIntensity;
5     int   lightType;
6     int   maxDepth;      // Cuántos rebotes máximos permitimos
7     int   nb_samples;    // Para antialiasing
8     int   frame;         // Para acumulación temporal
9 };
```

Las push constants son, como dice su nombre, constantes. ¿Y si queremos pasar información mutable entre shaders?

Para eso están los **payloads**. Específicamente, cada rayo puede llevar información adicional. Como una pequeña mochila. Esto resulta *muy* útil, por ejemplo, a la hora de calcular la radiancia de un camino. Se crean mediante la estructura `rayPayloadEXT`, y se reciben en otro shader mediante `rayPayloadInEXT`. Es importante controlar que el tamaño de la carga no sea excesivamente grande.

5.5.5. Creación de la ray tracing pipeline

El código de la creación de la pipeline lo encapsula la función `createRtPipeline()`, que se puede consultar [aquí](#)

5.6. Asmiray

5.7. Transporte de luz en la práctica

TODO: creo... que esto no debería ir aquí. Pero no quiero tampoco que el capítulo de radiometría sea un tocho impresionante.

Hemos llegado a una de las partes más importantes de este trabajo. Es el momento de poner en concordancia todo lo que hemos visto a lo largo de los capítulos anteriores.

Empecemos por la dispersión. ¿Recuerdas la ecuación de dispersión [3.5]? Podemos estimarla utilizando Monte Carlo:

$$L_o(p, \omega_o) = \int_{\mathbb{S}^2} f(p, \omega_o \leftarrow \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i$$

$$\approx \frac{1}{N} \sum_{j=1}^N \frac{f(p, \omega_o \leftarrow \omega_j) L_i(p, \omega_j) |\cos \theta_j|}{p(\omega_j)}$$

5.7.1. Materiales y objetos

NOTE: esto son notas para el Andrés del futuro. Sí, lo sé, está bastante claro solo con leerlo (□-□;)

Si quiero meter las BxDFs en los materiales tal y como tenía pensado (es decir, unas cuantas flags que me indiquen la BxDF que tengo que usar), tengo que...

1. Modificar `common/obj_loader.h/MaterialObj` para meterle las flags necesarias.
2. Modificar acordemente `shaders/host_device.h/WaveFronMaterial`.
3. Secuestrar `ObjLoader::loadModel()` para indicarle los parámetros nuevos.
4. (Creo que no hace falta tocar `HelloVk::loadModel()` de esta manera)
5. Toquetear los shaders para que me saque las flags.

CREO que de esta manera no me va a hacer falta tocar framebuffer. Simplemente, todo dependerá de mi material y ya.

Creo.

5.8. Fuentes de luz

TODO: point lights, area lights, ambient lights...

TODO: estas son notas muy puntuales (como las luces, jeje). Ya las revisaré más adelante.

TODO: 01_lights.pdf tiene información útil sobre muestreo directo de fuentes de luces.

La interfaz se encuentra en `host_device.h`. Describe cómo comunicarse con la GPU.

Ahora mismo, tenemos 3 constantes: tipo de luz:

```
1 vec3 lightPosition;    // (x, y, z)
2 float lightIntensity;  // (Intensidad)
3 int  lightType;        // (0 => point light, 1 => area light)
```

Sería interesante añadir algunas constantes para controlar el tamaño (radio, posición, normal para las de área...)

5.8.1. Point lights + spotlights

pbr-book, point lights: *“Strictly speaking, it is incorrect to describe the light arriving at a point due to a point light source using units of radiance. Radiant intensity is instead the proper unit for describing emission from a point light source, as explained in Section 5.4. In the light source interfaces here, however, we will abuse terminology and use `Sample_Li()` methods to report the illumination arriving at a point for all types of light sources, dividing radiant intensity by the squared distance to the point p to convert units. Section 14.2 revisits the details of this issue in its discussion of how delta distributions affect evaluation of the integral in the scattering equation. In the end, the correctness of the computation does not suffer from this fudge, and it makes the implementation of light transport algorithms more straightforward by not requiring them to use different interfaces for different types of lights.”*

```
1 // https://github.com/mmp/pbrt-v3/blob/master/src/lights/point.h
2 // https://github.com/mmp/pbrt-v3/blob/master/src/lights/point.cpp
```

```
3
4 Spectrum sample_light(interaccion, vec2 u, vec3 wi, float pdf,
    visibility_tester) {
5     wi = normalize(posicion_luz - interaccion.p);
6     pdf = 1.f;
7     // testeo de visibilidad. Opcional, I guess.
8
9     return intensidad / distancia_al_cuadrado(posicion_luz, interaccion.p)
        ;
10 }
```

La potencia total emitida por la luz puede calcularse integrando la intensidad desprendida en toda su superficie. Asumiendo la intensidad constante:

$$\Phi = \int_{\mathbb{S}^2} I d\omega = I \int_{\mathbb{S}^2} d\omega = 4\pi I$$

Las spotlights son variaciones de las point lights iluminando en un cono.

5.8.2. Fuentes de área

Para simplificar la implementación, podemos asumir que son rectangulares.

Nos van a hacer falta técnicas de Monte Carlo para solucionar el problema de calcular integrales a lo largo de su superficie.

Primero, lo mejor es asumir un cuadrado, y después, extender la interfaz para meter otras formas (es decir, rectángulos. Porque lo otro sería mucha parafernalia innecesaria).

[Código fuente](#)

Referencias

- <https://github.com/dannyfritz/awesome-ray-tracing>
- <https://www.wikiwand.com/en/Radeon>
- https://www.wikiwand.com/en/List_of_Nvidia_graphics_processing_units#/GeForce_30_series

- <https://www.eurogamer.net/digitalfoundry-2021-the-big-intel-interview-how-intel-alchemist-gpus-and-xess-upscaling-will-change-the-market>
- <https://www.intel.com/content/www/us/en/products/docs/arc-discrete-graphics/overview.htm>
- <https://www.khronos.org/registry/vulkan/specs/1.2-khr-extensions/html/chap1.html>
- <https://www.wikiwand.com/en/OptiX>
- https://www.wikiwand.com/en/DirectX_Raytracing
- https://www.wikiwand.com/es/Valve_Corporation
- https://www.phoronix.com/scan.php?page=news_item&px=VKD3D-Proton-2.5
- <https://github.com/ValveSoftware/Proton>
- https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/
- <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure>
- <https://www.khronos.org/blog/vulkan-ray-tracing-best-practices-for-hybrid-rendering>
- <https://raytracing.github.io/books/RayTracingTheNextWeek.html#boundingvolumehierarchies>
- https://vulkan-tutorial.com/en/Uniform_buffers/Descriptor_layout_and_buffer
- Ray tracing gems II, p.241.
- <https://www.willusher.io/graphics/2019/11/20/the-sbt-three-ways>

6. Análisis de rendimiento

TODO: para completar esta parte, necesitamos ambas implementaciones (en CPU y GPU) listas. Hasta entonces, esto se queda vacío. NOTE: Podría preguntarle a Kako que tire benchmark en su 2060, y a Manu con su 3060 (¿ti?)

Referencias

7. El presente y futuro de RT

TODO: de momento, se queda como está. Es un capítulo bastante fácil de escribir, así que en unos tres días como muchísimo podría estar todo listo.

7.1. Denoising

<https://alain.xyz/blog/ray-tracing-denoising>

7.2. Filtering

<https://alain.xyz/blog/ray-tracing-filtering>

7.3. Offline renderers

7.4. Importance Resampling

7.5. ReSTIR

7.6. Low discrepancy sampling

7.7. La industria del videojuego

7.7.1. Ray tracing híbrido

<https://www.khronos.org/blog/vulkan-ray-tracing-best-practices-for-hybrid-rendering>

7.7.2. Productos comerciales

- Control
 - Híbrido
 - <https://alain.xyz/blog/frame-analysis-control>
 - <https://www.youtube.com/watch?v=blbu0g9DAGA>
- Minecraft RTX
 - Path tracing
 - <https://alain.xyz/blog/frame-analysis-minecraftrtx>
 - https://www.youtube.com/watch?v=s_eeWr622Ss
 - https://www.youtube.com/watch?v=TVtSsJf86_Y
- Cyberpunk 2077
 - Híbrido
- Quake II RTX

- Path tracing
- Doom RTX
 - Path tracing
- Metro Exodus

7.7.3. Unreal Engine 5

7.7.4. La última generación de consolas

7.8. Posibles mejoras del trabajo

- Test Driven Development
 - *White furnace test* (01_lights.pdf, p.61)
- Debugging <https://alain.xyz/blog/graphics-debugging>
- Materiales
 - Physically based materials
 - Diferentes tipos de materiales
 - Subsurface scattering
- HDR
- Cámara
 - Diferentes tipos de cámaras
 - Focal lenght, depth of field, motion blur

Referencias

A. Metodología de trabajo

Cualquier proyecto de una envergadura considerable necesita ser planificado con antelación. En este capítulo vamos a hablar de cómo se ha realizado este trabajo: mostraremos las herramientas usadas, los ciclos de desarrollo, integración entre documentación y path tracer, y otras influencias que han afectado al producto final.

A.1. Influencias

Antes de comenzar con la labor, primero uno se debe hacer una simple pregunta:

“Y esto, ¿por qué me importa?”

Dar una respuesta contundente a este tipo de cuestiones nunca es fácil. Sin embargo, sí que puedo proporcionar motivos por los que he querido escribir sobre ray tracing.

Una de las principales influencias ha sido [Digital Foundry](#). Este grupo de divulgación se dedica al estudio de las técnicas utilizadas en el mundo de los videojuegos. El inicio de la era del ray tracing en tiempo real les llevó a dedicar una serie de vídeos y artículos a esta tecnología, y a las diferentes maneras en las que se ha implementado. Se puede ver un ejemplo en ([Digital Foundry n.d.](#)).

Dado que esta área combina tanto informática, matemáticas y una visión artística, ¿por qué no explorarlo a fondo?

Ahora que se ha decidido el tema, es hora de ver cómo atacarlo.

Soy un fiel creyente del aprendizaje mediante el juego. Páginas como [Explorable Explanations](#), el [blog de Bartosz Ciechanowski](#), el proyecto [The napkin](#) o el divulgador

3Blue1Brown repercuten inevitablemente en la manera en la que te planteas cómo comunicar textos científicos. Por ello, aunque esto a fin de cuentas es un trabajo de fin de grado de una carrera, quería ver hasta dónde era capaz de llevarlo.

Otro punto importante es la *manera* de escribir. No me gusta especialmente la escritura formal. Prefiero ser distendido. Por suerte, parece que el mundo científico se está volviendo más informal (*Nature n.d.*), así que no soy el único que aprueba esta tendencia. Además, la estructura clásica de un escrito matemático de “teorema, lema, demostración, corolario” no me agrada especialmente. He intentado preservar su estructura, pero sin ser tan explícito. Estos dos puntos, en conjunto, suponen un balance entre formalidad y distensión difícil de mantener.

A.2. Ciclos de desarrollo

Este proyecto está compuesto por 2 grandes pilares: documentación—lo que estás leyendo, ya sea en PDF o en la web— y software.

La metodología que se ha seguido es, en esencia, una versión de Agile muy laxa (*Beck et al. 2001*).

Para empezar, se implementaron los tres libros de Shirley de la “serie In One Weekend”: In One Weekend (*Shirley 2020a*), The Next Week (*Shirley 2020b*), y The Rest of your Life (*Shirley 2020c*).

Tras esto, comenzó a *desarrollarse* el motor por GPU. Cuando se consiguió una base sólida (que se puede ver en *este issue del repositorio*), se empezó a alternar entre escritura de documentación y desarrollo del software. A fin de cuentas, no tiene sentido implementar algo que no se conoce.

Para apoyar el desarrollo, se ha utilizado *Github*. Más adelante hablaremos de cómo esta plataforma ha facilitado el trabajo.

A.3. Presupuesto

TODO: ahora mismo, no tengo ni idea de cómo empezar esto.

A.4. Arquitectura del software

TODO: especificaciones de los requerimientos y metodología de desarrollo, así como los planos del proyecto que contendrán las historias de usuario o casos de uso, diagrama conceptual, de iteración, de diseño, esquema arquitectónico y bocetos de las interfaces de usuario. Se describirán las estructuras de datos no fundamentales y algoritmos no triviales

(Odio los diagramas de clases. Me gustaría evitar como sea incluirlos.)

A.5. Diseño

TODO: hablar de paleta de colores, tipografía...

El diseño juega un papel fundamental en este proyecto. Todos los elementos visuales han sido escogidos con cuidado, de forma que se preserve la estética.

Se ha creado **un diseño que preserve el equilibrio entre la profesionalidad y la distensión**.

A.5.1. Bases del diseño

Para la documentación en versión PDF, usamos como base la *template* [Eisvogel](#). Esta es una elegante plantilla fácil de usar para LaTeX. Uno de sus puntos fuertes es la personalización, la cual aprovecharemos para darle un toque diferente.

La web utiliza como base el estilo generado por Pandoc, el microframework de css [Bamboo](#) y unas modificaciones personales.

A.5.2. Tipografías

Un apartado al que se le debe prestar especial énfasis es a la combinación de tipografías. A fin de cuentas, esto es un libro; así que escoger un tipo de letra correcto facilitará al lector comprender los conceptos. Puede parecer trivial a priori, pero es importante.

Para este trabajo, se han escogido las siguientes tipografías:

- **Crimson Pro**: una tipografía serif clara, legible y contemporánea. Funciona muy bien en densidades más bajas, como 11pt. Es ideal para la versión en PDF. Además, liga estupendamente con Source Sans Pro, utilizada para los títulos en la plantilla Eisvogel.
- **Fraunces**: de lejos, la fuente más interesante de todo este proyecto. Es una soft-serif *old style*, pensada para títulos y similares (lo que se conoce como *display*). Es usada en los títulos de la web. Una de sus propiedades más curiosas es que modifica activamente los glifos dependiendo del valor del *optical size axis*, el peso y similares. Recomiendo echarle un ojo a su [repositorio de Github](#).
- **Rubik**: La elección de Rubik es peculiar. Por sí sola, no casa con el proyecto. Sin embargo, combinada con Fraunces, proporcionan un punto de elegancia y familiaridad a la web. Su principal fuerte es la facilidad para la comprensión lectora en pantallas, algo que buscamos para la página web.
- **Julia Mono**: monoespaciada, pensada para computación científica. Llevo usándola bastante tiempo, y combia bien con Crimson Pro.
- **Jetbrains Mono**: otra tipografía monoespaciada open source muy sólida, producida por la compañía JetBrains. Se utiliza en la web para los bloques de código.

Todas estas fuentes permiten un uso no comercial gratuito.

TODO: Añadir imagen comparativa con las fuentes

A.5.3. Paleta de colores

A fin de mantener consistencia, se ha creado una paleta de colores específica.

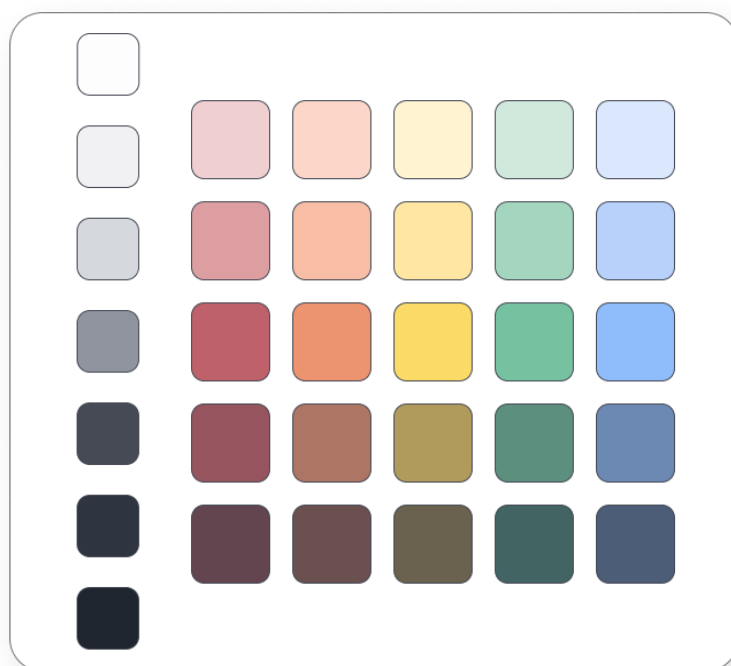


Figure A.1.: La paleta de colores del proyecto

El principal objetivo es **transmitir tranquilidad**, pero a la misma vez, **profesionalidad**. De nuevo, buscamos la idea de profesionalidad distendida que ya hemos repetido un par de veces.

Partiendo del rojo que traía Eisvogel (lo que para nosotros sería el rojo primario), se han creado el resto. En principio, con 5 tonalidades diferentes nos basta. Todas ellas vienen acompañadas de sus respectivas variaciones oscuras, muy oscuras, claras y muy claras. Corresponderían a los `color-100`, `color-300`, `color-500`, `color-700`, `color-900` que estamos acostumbrados en diseño web. Para la escala de grises, se han escogido 7 colores en vez de 9. Son más que suficientes para lo que necesitamos. Puedes encontrar las definiciones en el [fichero de estilos](#).

Todos los colores que puedes ver en este documento se han extraído de la paleta. ¡La consistencia es clave!

A.6. Flujo de trabajo y herramientas

Encontrar una herramienta que se adapte a un *workflow* es complicado. Aunque hay muchos programas maravillosos, debemos hacerlos funcionar en conjunto. En este apartado, vamos a describir cuáles son las que hemos usado.

Principalmente destacan tres de ellas: **Github**, **Pandoc** y **Figma**. La primera tendrá **su propia sección**, así que hablaremos de las otras.

TODO: foto del workflow.

A.6.1. Pandoc

Pandoc es una estupendísima de conversión de documentos. Se puede usar para convertir un tipo de archivo a otro. En este caso, se usa para convertir una serie de ficheros Markdown (los capítulos) a un fichero HTML (la web) y a PDF. Su punto más fuerte es que permite escribir LaTeX de forma simplificada, como si se tratara de *sugar syntax*. Combina la simplicidad de Markdown y la correctitud de LaTeX.

Su funcionamiento en este proyecto es el siguiente: Primero, recoge los capítulos que se encuentra en `docs/chapters`, usando una serie de cabeceras en YAML que especifican ciertos parámetros (como autor, fecha, título, etc.), así como scripts de Lua. Estas cabeceras se encuentran en `docs/headers`. En particular:

1. `meta.md` recoge los parámetros base del trabajo.
2. `pdf.md` y `web.md` contienen algunas definiciones específicas de sus respectivos formatos. Por ejemplo, el YAML del PDF asigna las variables disponibles de la plantilla Eisvogel; mientras que para la web se incluyen las referencias a algunas bibliotecas de Javascript necesarias o los estilos (`docs/headers/style.css`, usando como base `Bamboo.css`).
3. `math.md` contiene las definiciones de LaTeX.
4. Se utilizan algunos filtros específicos de Lua para simplificar la escritura. En específico, `standard-code.lua` formatea correctamente los bloques de código para la web.

Un fichero Makefile (`docs/Makefile`) contiene varias órdenes para generar ambos formatos. Tienen varios parámetros adicionales de por sí, como puede ser la bibliografía

(docs/chapters/bibliography.bib).

A.6.2. Figma

Figma es otro de esos programas que te hace preguntarte por qué es gratis. Es una aplicación en la web usada para diseño gráfico. Es muy potente, intuitiva, y genera unos resultados muy buenos en poco tiempo. Todos los diseños de este trabajo se han hecho con esta herramienta.

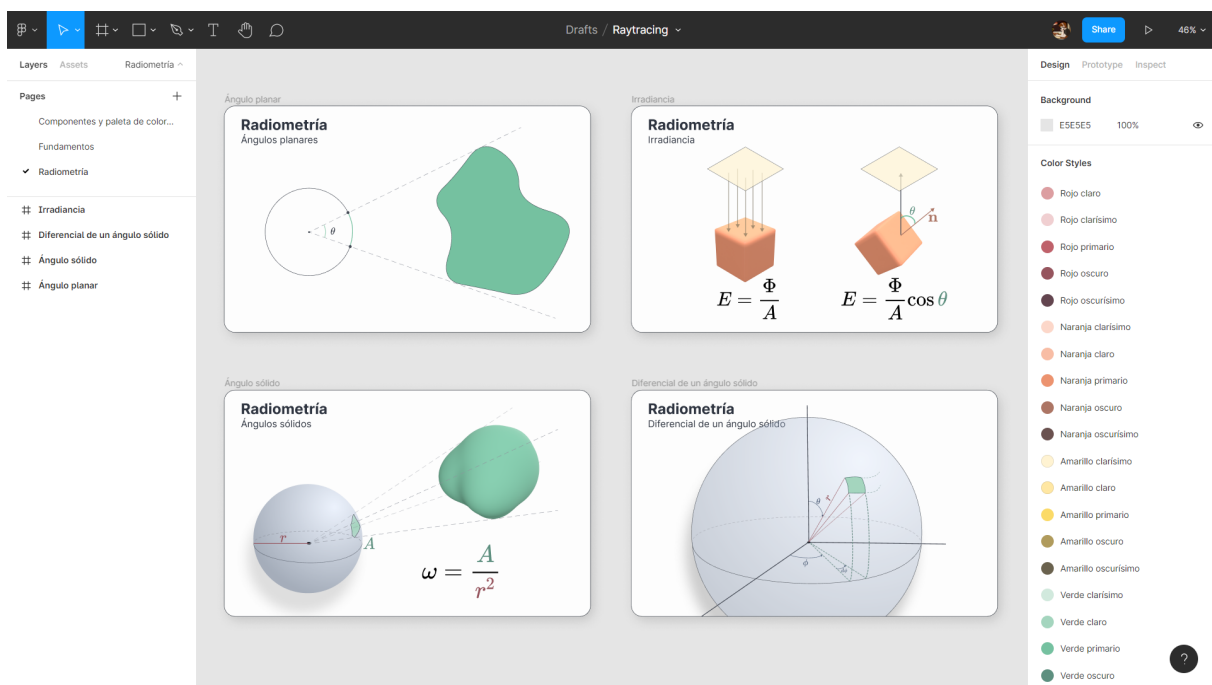


Figure A.2.: Tablón principal del proyecto de Figma, a día 15 de abril de 2022

Una de las características más útiles es poder exportar rápidamente la imagen. Esto permite hacer cambios rápidos y registrarlos en el repositorio fácilmente. Además, permite instalar plugins. Uno de ellos ha resultado especialmente útil: **Latex Complete**. Esto nos permite incrustar código LaTeX en el documento en forma de SVG.

A.6.3. Otros programas

Como es normal, hay muchos otros programas que han intervenido en el desarrollo. Estos son algunos de ellos:

- El editor por excelencia [VSCode](#). Ha facilitado en gran medida el desarrollo de la aplicación y la documentación. En particular, se ha usado una extensión denominada [Trigger task on save](#) que compila la documentación HTML automáticamente al guardar un fichero. ¡Muy útil y rápido!
- [Vectary](#) para hacer los diseños en 3D fácilmente. Permite exportar una escena rápidamente a png para editarla en Figma.
- Como veremos más adelante, la documentación se compila en el repositorio usando un contenedor de [Docker](#).
- Cualquier proyecto informático debería usar [git](#). Este no es una excepción.

A.7. Github

La página [Github](#) ha alojado prácticamente todo el contenido del trabajo; desde el programa, hasta la documentación online. El repositorio se puede consultar en [Github.com/Asmilex/Raytracing](https://github.com/Asmilex/Raytracing).

Se ha escogido Github en vez de sus competidores por los siguientes motivos:

1. Llevo usándola toda la carrera. Es mi página de hosting de repositorios favorita.
2. Los repositorios de Nvidia se encontraban en Github, por lo que resulta más fácil sincronizarlos.
3. La documentación se puede desplegar usando Github Pages.
4. Las Github Actions son particularmente cómodas y sencillas de usar.

Entremos en detalle en algunos de los puntos anteriores:

A.7.1. Integración continua con Github Actions y Github Pages

Cuando hablamos de **integración continua**, nos referimos a ciertos programas que corren en un repositorio y se encargan de hacer ciertas transformaciones al código, de forma que este se prepare para su presentación final. En esencia, automatizan algunas tareas habituales de un desarrollo de software.

En este trabajo lo usaremos para compilar la documentación. De esta forma, no necesitamos lidiar con “proyecto final”, “proyecto final definitivo”, “proyecto final final v2”, etc. Simplemente, cuando registremos un cambio en los ficheros Markdown (lo que se conoce en git como un `commit`), y lo subamos a Github (acción de `push`), se ejecutará un denominado `Action` que operará sobre nuestros archivos.

Tendremos dos tipos de `Actions`: uno que se encarga de compilar la web, y otro el PDF. En esencia, operan de la siguiente manera:

1. Comprueba si se ha modificado algún fichero `.md` en el último commit subido. Si no es el caso, para.
2. Si sí se ha modificado, accede a la carpeta del repositorio y compila la documentación mediante `pandoc`.
 1. La web se genera en `docs/index.html`. Publica la web a Github Pages.
 2. El PDF se crea en `docs/TFG.pdf`
3. Commitea los archivos y termina.

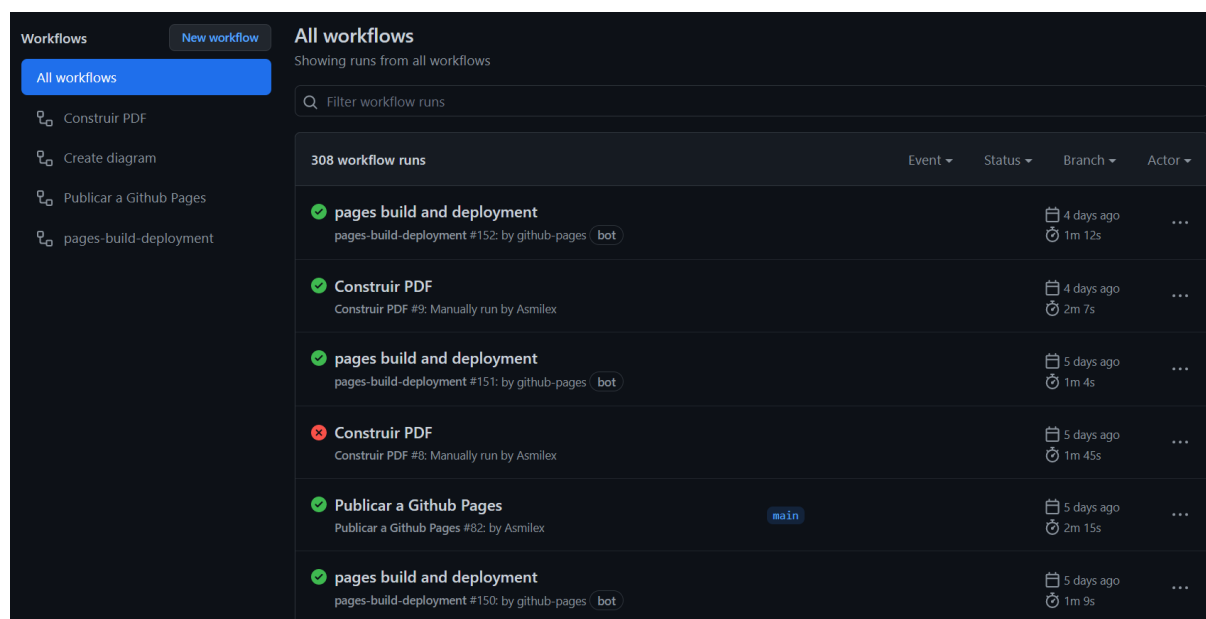


Figure A.3.: La pestaña de Github Actions permite controlar con facilidad el resultado de un workflow y cuánto tarda en ejecutarse

El workflow de la web corre automáticamente, mientras que para generar el PDF hace falta activación manual. Aunque no es *del todo* correcto almacenar ficheros binarios en un repositorio de git, no me resulta molesto personalmente. Así que, cuando considero que es el momento oportuno, lo hago manualmente. Además, también se activa por cada *release* que se crea.

Volviendo a la web, Github permite alojar páginas web para un repositorio. Activando el parámetro correcto en las opciones del repositorio, y configurándolo debidamente, conseguimos que lea el archivo `index.html` generado por el Action y lo despliegue. Esto es potentísimo: con solo editar una línea de código y subir los cambios, conseguimos que la web se actualice al instante.

Para generar los archivos nos hace falta una distribución de LaTeX, Pandoc, y todas las dependencias (como filtros). Como no encontré ningún contenedor que sirviera mi propósito, decidí crear uno. Se encuentra en el [repositorio de Dockerhub](#). Esta imagen está basada en [dockershelf/latex:full](#). Por desgracia, es *mu*y pesada para ser un

contenedor. Desafortunadamente, una instalación de LaTeX ocupa una cantidad de espacio considerable; y para compilar el PDF necesitamos una muy completa, por lo que debemos lidiar con este *overhead*. Puedes encontrar el Dockerfile [aquí](#).

A.7.2. Issues y Github Projects

Las tareas pendientes se gestionan mediante issues. Cada vez que se tenga un objetivo particular para el desarrollo, se anota un issue. Cuando se genere un commit que avance dicha tarea, se etiqueta con el número correspondiente al issue. De esta forma, todas las confirmaciones relacionadas con la tarea quedan recogidas en la página web. Puedes ver un ejemplo en el [issue número 22](#).

Esto permite una gestión muy eficiente de los principales problemas y objetivos pendientes de la aplicación.

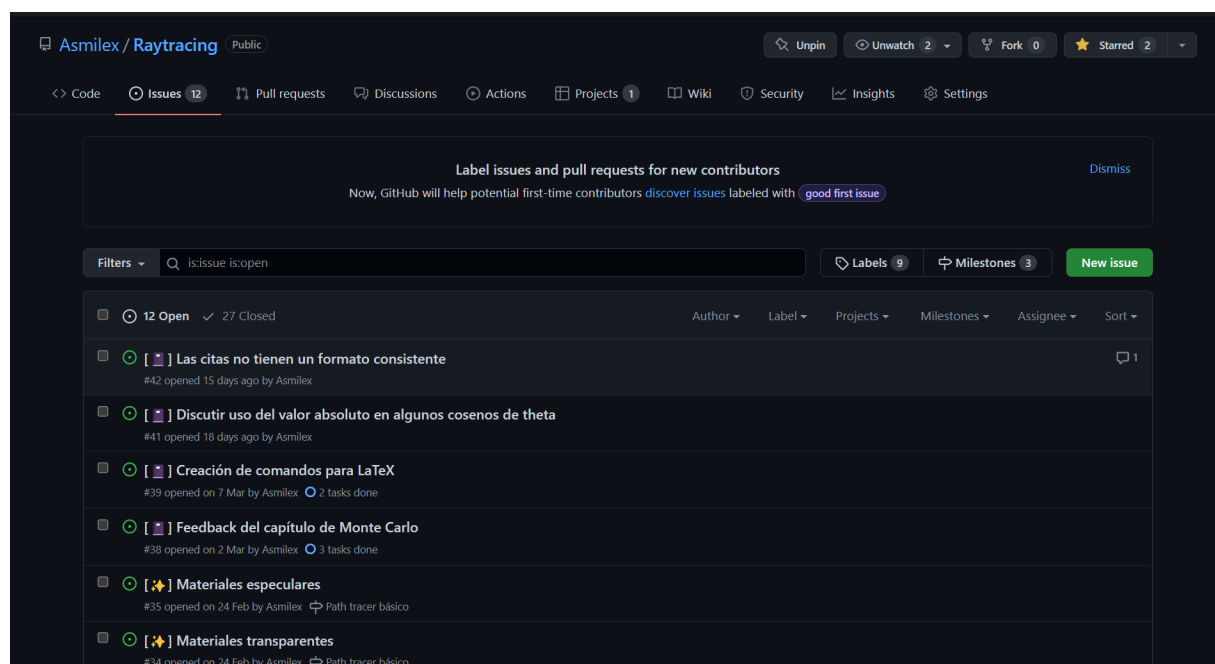


Figure A.4.: Pestaña de issues, día 16 de abril de 2022

Los issues se agrupan en *milestones*, o productos mínimamente viables. Estos issues suelen estar relacionados con algún apartado importante del desarrollo.

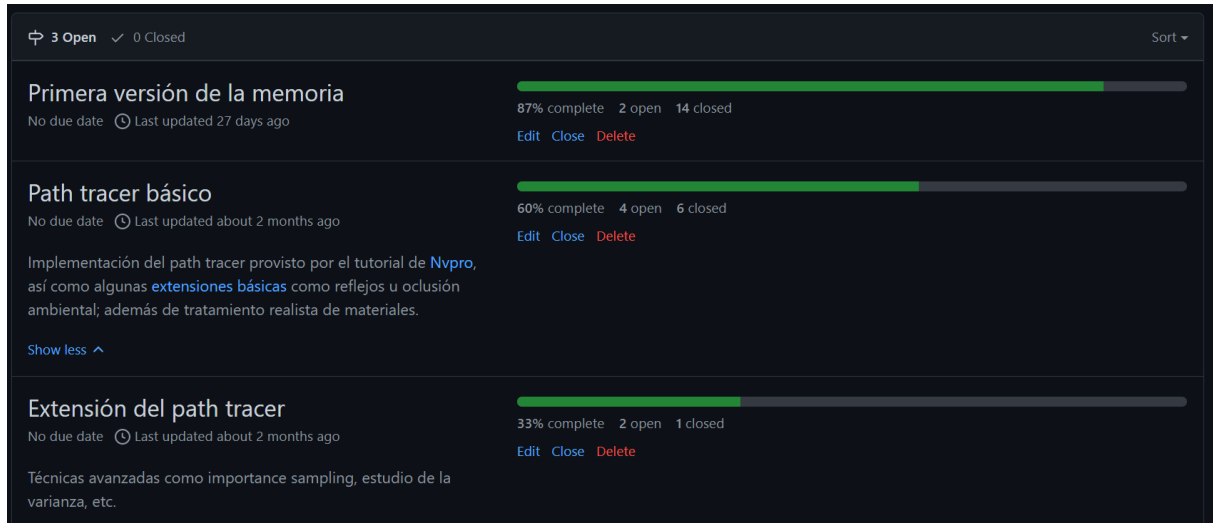
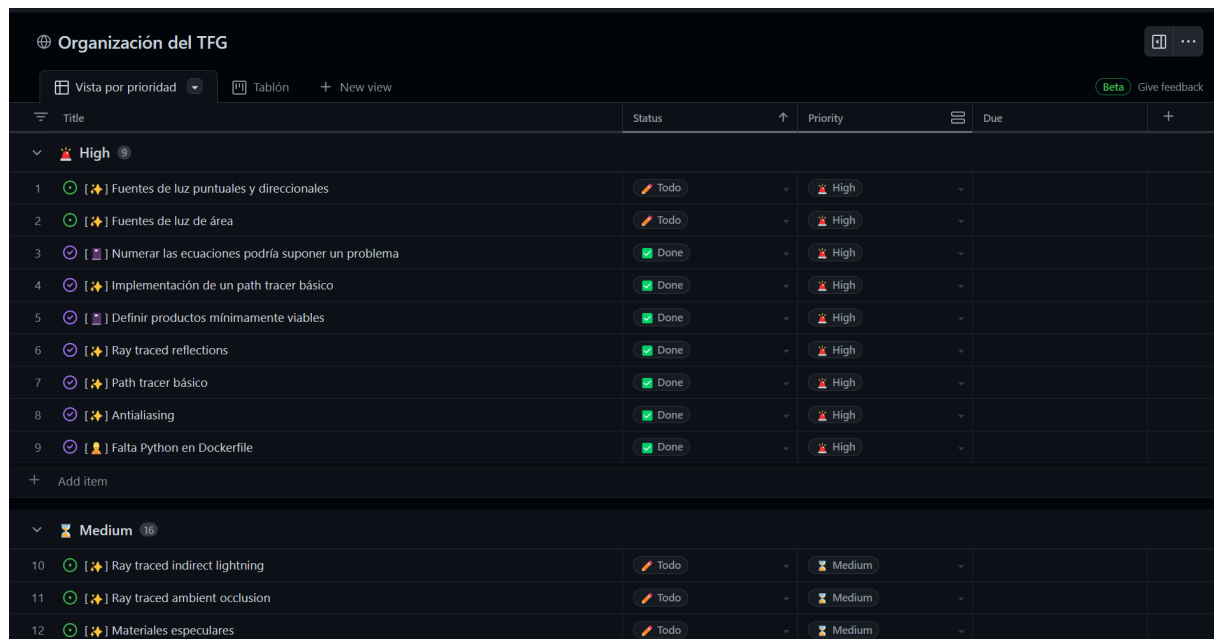


Figure A.5.: Los *milestones* agrupan una serie de issues relacionados con un punto clave del desarrollo

De esta forma, podemos ver todo lo que queda pendiente para la fecha de entrega.

Para añadir mayor granularidad a la gestión de tareas y proporcionar una vista informativa, se utiliza Github Projects. En esencia, esta aplicación es un acompañante del repositorio estilo Asana.



Title	Status	Priority	Due
High			
1 [🔥] Fuentes de luz puntuales y direccionales	Todo	High	
2 [🔥] Fuentes de luz de área	Todo	High	
3 [🔥] Numerar las ecuaciones podría suponer un problema	Done	High	
4 [🔥] Implementación de un path tracer básico	Done	High	
5 [🔥] Definir productos mínimamente viables	Done	High	
6 [🔥] Ray traced reflections	Done	High	
7 [🔥] Path tracer básico	Done	High	
8 [🔥] Antialiasing	Done	High	
9 [🔥] Falta Python en Dockerfile	Done	High	
Medium			
10 [🔥] Ray traced indirect lightning	Todo	Medium	
11 [🔥] Ray traced ambient occlusion	Todo	Medium	
12 [🔥] Materiales especulares	Todo	Medium	

Figure A.6.: Projects agrupa los issues y les asigna prioridades

Una de las alternativas que se planteó al inicio fue [Linear](#), una aplicación de gestión de issues similar a Projects. Sin embargo, la conveniencia de tener Projects integrado en Github supuso un punto a favor para este gestor. De todas formas, el equipo de desarrollo se compone de una persona, así que no hace falta complicar excesivamente el workflow.

El desarrollo general de la documentación no ha seguido este sistema de issues, pues está sujeta a cambios constantes y cada commit está marcado con `[:notebook:]`. No obstante, ciertos problemas relacionados con ella, como puede ser el formato de entrega, sí que quedan recogidos como un issue.

Finalmente, cuando se produce un cambio significativo en la aplicación (como puede ser una refactorización, una implementación considerablemente más compleja...) se genera una nueva rama. Cuando se ha cumplido el objetivo, se *mergea* la rama con la principal `main` mediante un *pull request*. Esto proporciona un mecanismo de robustez ante cambios complejos.

A.7.3. Estilo de commits

Una de los detalles que has podido apreciar si has entrado al repositorio es un estilo de commit un tanto inusual. Aunque parece un detalle de lo más insustancial, añadir emojis a los mensajes de commits añade un toque particular al repositorio, y permite identificar rápidamente el tipo de cambio.

Cada uno tiene un significado particular. En esta tabla se recogen sus significados:







Tipo de commit	Emoji	Cómo se escribe rápidamente
Documentación		<code>:notebook:</code>
Archivo de configuración		<code>:wrench:</code>
Integración continua		<code>:construction_worker:</code>
Commit de Actions		<code>:robot:</code>
Quitar archivos		<code>:fire:</code>
Nuevas características		<code>:sparkles:</code>
Test		<code>:alembic:</code>
Refactorización		<code>:recycle:</code>
Bugfix		<code>:bug:</code>

Figure A.7.: Los emojis permiten reconocer el objetivo de cada commit. Esta tabla recoge el significado de cada uno

Referencias

(Digital Foundry n.d.), (Nature n.d.), (Beck et al. 2001), (Merelo n.d.)

B. Glosario de términos

It's dangerous to go alone, take this.

Tener en mente *todos* los conceptos y sus expresiones que aparecen en un libro como este es prácticamente imposible. Tampoco hay necesidad de ello, realmente. ¡Vaya desperdicio de cabeza! Por eso, aquí tienes recopilada una lista con todos los elementos importantes y un enlace a sus secciones correspondientes.

B.1. Notación

Concepto	Notación
Puntos	Letras mayúsculas: P, Q, \dots
Escalares	Letras minúsculas: a, b, c, k, \dots
Vectores	Letras minúsculas en negrita: $\mathbf{v}, \mathbf{w}, \mathbf{n}, \dots$. Si están normalizados, se les pone gorrito (por ejemplo, $\hat{\mathbf{n}}$)
Matrices	Letras mayúsculas en negrita: \mathbf{M} . Por columnas.
Producto escalar	$\mathbf{v} \cdot \mathbf{w}$. Si es el producto escalar de un vector consigo mismo, a veces pondremos \mathbf{v}^2
Producto vectorial	$\mathbf{v} \times \mathbf{w}$
Variables aleatorias	X, Y . ξ representa una variable aleatoria con distribución uniforme en $[0, 1)$.

B.2. Radiometría

Concepto	Expresiones
Ángulo sólido, derivada [3.1]	$\omega = \frac{A}{r^2}$ $d\omega = \sin \theta \, d\theta \, d\phi$
Hemisferio de direcciones alrededor de un vector	$H^2(\mathbf{n})$
Carga de energía	$Q = hf = \frac{hc}{\lambda}$
Flujo radiante, potencia	$\Phi = \frac{dQ}{dt}$ $\Phi = \int_A \int_{H^2(\mathbf{n})} L_o(p, \omega) d\omega^\perp dA$
Irradiancia, radiancia emitida	$E = \frac{\Phi}{A}$ $E(p) = \frac{d\Phi}{dA}$ $E(p, \mathbf{n}) = \int_{\Omega} L_i(p, \omega) \cos \theta d\omega$ $E(p, \mathbf{n}) = \int_0^{2\pi} \int_0^{\pi/2} L_i(p, \theta, \phi) \cos \theta \sin \theta \, d\theta \, d\phi$ $E(p, \mathbf{n}) = \int_A L \cos \theta \frac{\cos \theta_o}{r^2} dA$
Intensidad radiante	$I = \frac{d\Phi}{d\omega}$

Concepto	Expresiones
Radiancia	$L(p, \omega) = \frac{dE_\omega(p)}{d\omega}$ $L(p, \omega) = \frac{d^2\Phi(p, \omega)}{d\omega dA^\perp} = \frac{d^2\Phi(p, \omega)}{d\omega dA \cos \theta}$ $L^+(p, \omega) = \lim_{t \rightarrow 0^+} L(p + t\mathbf{n}_p, \omega)$ $L^-(p, \omega) = \lim_{t \rightarrow 0^-} L(p + t\mathbf{n}_p, \omega)$
Radiancia incidente	$L_i(p, \omega) = \begin{cases} L^+(p, -\omega) & \text{si } \omega \cdot \mathbf{n}_p > 0 \\ L^-(p, -\omega) & \text{si } \omega \cdot \mathbf{n}_p < 0 \end{cases}$
Radiancia reflejada, radiancia de salida	$L_o(p, \omega) = \begin{cases} L^+(p, \omega) & \text{si } \omega \cdot \mathbf{n}_p > 0 \\ L^-(p, \omega) & \text{si } \omega \cdot \mathbf{n}_p < 0 \end{cases}$
Ecuación de dispersión	$L_o(p, \omega_o) = \int_{\mathbb{S}^2} f(p, \omega_o \leftarrow \omega_i) L_i(p, \omega_i) \cos \theta_i d\omega_i$
BRDF	$f_r(p, \omega_o \leftarrow \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos \theta_i d\omega_i}$
BTDF	$f_t(p, \omega_o \leftarrow \omega_i)$
BSDF	$f(p, \omega_o \leftarrow \omega_i)$

Bibliografía

- Adam Marrs, Peter Shirley, and Ingo Wald, eds. 2021. “Ray Tracing Gems II.” Apress. 2021. <http://raytracinggems.com/rtg2>.
- Arnebäck. n.d. “An Explanation of the Rendering Equation.” Accessed April 9, 2022. https://www.youtube.com/watch?v=eo_MTI-d28s.
- Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, et al. 2001. “Manifesto for Agile Software Development.” <http://www.agilemanifesto.org/>.
- Berkeley cs184. n.d. “Monte Carlo Integration Cs184/284a.” Accessed March 20, 2022. <https://cs184.eecs.berkeley.edu/sp22>.
- Caulfield, Brian. n.d. “What’s the Difference Between Ray Tracing, Rasterization?” Accessed April 22, 2022. <https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/>.
- Crytek. n.d. “Crysis Remastered Brings Ray Tracing to Current-Gen Consoles.” Accessed April 17, 2022. <https://www.cryengine.com/news/view/crysis-remastered-brings-ray-tracing-to-current-gen-consoles>.
- Digital Foundry. n.d. “Cyberpunk 2077 PC: What Does Ray Tracing Deliver... And Is It Worth It?” Accessed April 10, 2022. <https://www.youtube.com/watch?v=6bqA8F6B6NQ>.
- Fabio Pellacini, Steve Marschner. n.d. “Fundamentals of Computer Graphics.” Accessed April 9, 2022. <https://pellacini.di.uniroma1.it/teaching/graphics17b/>.
- Galvin. n.d. “Random Variables.” Accessed March 20, 2022. https://www3.nd.edu/~dgalvin1/10120/10120_S16/Topic17_8p4_Galvin_class.pdf.

- Haines, Eric, and Tomas Akenine-Möller, eds. 2019. "Ray Tracing Gems." Apress. 2019. <http://raytracinggems.com>.
- Merelo. n.d. "Infraestructura Virtual." Accessed April 16, 2022. http://jj.github.io/IV/documentos/temas/Integracion_continua.
- Nature. n.d. "Scientific Language Is Becoming More Informal." Accessed April 10, 2022. <https://doi.org/10.1038/539140a>.
- Overvoorde, Alexander. n.d. "Introduction - Vulkan Tutorial." Accessed April 18, 2022. <https://vulkan-tutorial.com/>.
- Owen, Art B. 2013. *Monte Carlo Theory, Methods and Examples*. <https://artowen.su.domains/mc/>.
- Pharr, Matt, Wenzel Jakob, and Greg Humphreys. 2016. "Physically Based Rendering: From Theory to Implementation (3rd Ed.)." San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. November 2016. <https://www.pbr-book.org/3ed-2018/contents>.
- QuantumFracture. n.d. "Ya, En Serio, ¿Qué Es La Luz?" Accessed April 22, 2022. <https://www.youtube.com/watch?v=DkcEAz09Buo>.
- "Rendering." n.d. Accessed March 20, 2022. <https://sciencebehindpixar.org/pipeline/rendering#:~:text=They%20said%20it%20takes%20at,to%20render%20that%20many%20frames>.
- Scratchapixel. n.d. "Learn Computer Graphics from Scratch!" Accessed April 17, 2022. <https://www.scratchapixel.com/index.php?redirect>.
- Shirley, Peter. 2020a. "Ray Tracing in One Weekend." 2020. <https://raytracing.github.io/books/RayTracingInOneWeekend.html>.
- . 2020b. "Ray Tracing: The Next Week." 2020. <https://raytracing.github.io/books/RayTracingTheNextWeek.html>.
- . 2020c. "Ray Tracing: The Rest of Your Life." 2020. <https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html>.
- Shirley, Peter, and R. Keith Morley. 2003. *Realistic Ray Tracing*. 2nd ed. USA: A. K. Peters, Ltd. <https://www.taylorfrancis.com/books/mono/10.1201/9780429294891/realistic->

[ray-tracing-peter-shirley-keith-morley](#).

StudySession. n.d. “Solid Angle Derivation & Intuition.” Accessed April 22, 2022. <https://www.youtube.com/watch?v=WtKsgBEIPWA>.

The Khronos® Vulkan Working Group. n.d. “Vulkan® 1.2.210 - KHR Extensions: 33. Ray Intersection.” Accessed April 1, 2022. <https://www.khronos.org/registry/vulkan/specs/1.2-khr-extensions/html/chap33.html#ray-intersection-candidate-determination>.

tracing, Wikipedia: Ray. n.d. “Ray Tracing (Graphics).” Accessed April 22, 2022. [https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics)).

Wikipedia: Barycentric coordinate system. n.d. “Barycentric Coordinate System.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Barycentric_coordinate_system.

Wikipedia: Computer. n.d. “Computer.” Accessed April 22, 2022. <https://en.wikipedia.org/wiki/Computer>.

Wikipedia: Differential geometry of surfaces. n.d. “Differential Geometry of Surfaces.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Differential_geometry_of_surfaces.

Wikipedia: Distribución de probabilidad. n.d. “Distribución de Probabilidad.” Accessed April 22, 2022. https://es.wikipedia.org/wiki/Distribuci%C3%B3n_de_probabilidad.

Wikipedia: Estimador. n.d. “Estimador.” Accessed April 22, 2022. <https://es.wikipedia.org/wiki/Estimador?oldformat=true>.

Wikipedia: Expected value. n.d. “Expected Value.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Expected_value.

Wikipedia: Función de distribución de reflectancia bidireccional. n.d. “Función de Distribución de Reflectancia Bidireccional.” Accessed April 22, 2022. https://es.wikipedia.org/wiki/Funci%C3%B3n_de_distribuci%C3%B3n_de_reflectancia_bidireccional.

- Wikipedia: Función de probabilidad. n.d. “Función de Probabilidad.” Accessed April 22, 2022. https://es.wikipedia.org/wiki/Funci%C3%B3n_de_probabilidad.
- Wikipedia: history of photography. n.d. “History of Photography.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/History_of_photography.
- Wikipedia: Implicit surface. n.d. “Implicit Surface.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Implicit_surface.
- Wikipedia: Kodak. n.d. “Kodak.” Accessed April 22, 2022. <https://es.wikipedia.org/wiki/Kodak>.
- Wikipedia: Método de la transformada inversa. n.d. “Método de La Transformada Inversa.” Accessed April 22, 2022. https://es.wikipedia.org/wiki/M%C3%A9todo_de_la_transformada_inversa.
- Wikipedia: Parametric surface. n.d. “Parametric Surface.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Parametric_surface.
- Wikipedia: Probability density function. n.d. “Probability Density Function.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Probability_density_function.
- Wikipedia: Radiometry. n.d. “Radiometry.” Accessed March 20, 2022. <https://en.wikipedia.org/wiki/Radiometry>.
- Wikipedia: Rejection sampling. n.d. “Rejection Sampling.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Rejection_sampling.
- Wikipedia: rendering (computer graphics). n.d. “Rendering (Computer Graphics).” Accessed April 22, 2022. [https://en.wikipedia.org/wiki/Rendering_\(computer_graphics\)](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)).
- Wikipedia: Rendering equation. n.d. “Rendering Equation.” Accessed April 22, 2022. https://en.wikipedia.org/wiki/Rendering_equation.
- Wikipedia: Transmittance. n.d. “Transmittance.” Accessed April 22, 2022. <https://en.wikipedia.org/wiki/Transmittance>.
- Wikipedia: Variable aleatoria. n.d. “Variable Aleatoria.” Accessed April 22, 2022. https://es.wikipedia.org/wiki/Variable_aleatoria.