Universidad de la República
Facultad de Ingeniería

Renderizado en tiempo real acelerado mediante hardware de ray tracing

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Manuel Machado de Benedetti

Tutores

Eduardo Fernández
José Pedro Aguerre

Tribunal

Ernesto Dufrechou
Federico Rivero
Libertad Tansini

Montevideo
jueves 7 octubre, 2021

Agradecimientos

Quisiera agradecer en primer lugar a mis tutores José Pedro Aguerre y Eduardo Fernández por haberme presentado en este proyecto, por sus aportes y excelente disposición a lo largo del mismo.

También me gustaría agradecer a mi familia que me apoyó y motivó durante toda la carrera, en especial a mi madre, mi padre y mi hermana, a quienes va dedicado este proyecto.

Resumen

A finales de 2018 la empresa NVIDIA especializada en el desarrollo de unidades de procesamiento gráfico, lanzó al mercado un nuevo producto para computadoras personales. La arquitectura «Turing» utilizada en sus nuevas tarjetas gráficas RTX, facilita la implementación y mejora sustancialmente el rendimiento de algoritmos basados en la técnica de ray tracing.

Las técnicas de ray tracing son óptimas para la simulación del comportamiento de la luz y cómo llega a nuestros ojos, siendo éste el objetivo más perseguido en computación gráfica para renderizado 3D. Sin embargo, teniendo en cuenta las exigencias de la industria de videojuegos en cuanto a resoluciones de pantalla y cantidad de cuadros por segundo, estas técnicas son virtualmente imposibles de ejecutar únicamente con el hardware específico de aceleración de ray tracing añadido en la arquitectura Turing.

Lo que hace viable al ray tracing en esta nueva generación de tarjetas gráficas es el complemento de otros componentes o funcionalidades de la GPU. Por un lado, el procesamiento de imágenes en tiempo real por redes neuronales con la ayuda de Tensor Cores. Y por otro lado, su implementación estratégica en combinación con otras técnicas clásicas de renderizado, que han sido utilizadas por décadas en el hardware de las GPU y permiten mayores optimizaciones.

En este proyecto de grado se estudian en profundidad las posibilidades del nuevo hardware de NVIDIA para el desarrollo de ray tracing, así como sus ventajas reales en comparación con el desarrollo en las GPU de generaciones anteriores. También se estudia una de las aplicaciones que utiliza Inteligencia Artificial desarrollada por la misma empresa para el mejorado, denoising y aceleración de la generación de imágenes con ray tracing. Para ello se implementaron tres aplicaciones de ray tracing, que tienen distintos objetivos: ray tracing físicamente realista, ray tracing con denoising basado en IA y ray tracing en tiempo real enfocado para su uso en videojuegos. El código de fuente de este trabajo está publicado en *https://github.com/manueme/ray-tracing-research*.

Como requerimiento para el desarrollo de las aplicaciones que ponen a prueba el hardware estudiado, se debió realizar un análisis sobre la forma en la que se adaptaron las distintas bibliotecas gráficas que acceden al GPU. Se presentan así implementaciones con *Vulkan*, una biblioteca gráfica de bajo nivel, que hoy en día es parte de los nuevos estándares de la industria en los que debe estar capacitado un desarrollador para acceder y trabajar con las nuevas funcionalidades de trazado de rayos.

Palabras clave: *Ray Tracing, Path Tracing, Render Híbrido, Hardware Gráfi-*

co, Iluminación Global, Tiempo Real, NVIDIA RTX, Vulkan, Optix Denoiser

Tabla de contenidos

Capítulo 1

Introducción

El trabajo de representar el color y el brillo de un objeto en una imagen, se puede realizar mediante el cálculo de la interacción de la luz con la superficie de ese objeto. Una forma sencilla de interpretar la luz es abstraer el comportamiento de los fotones a rayos. Para ello se asume que los fotones viajan en línea recta y transportan energía, luego, a raíz de la interacción de estos rayos con la escena representada, se define el color de las distintas superficies visibles para el observador.

Los algoritmos basados en este enfoque pueden converger a resultados de calidad cinematográfica y/o físicamente realistas, pero tienen un alto costo computacional. Por esta razón son inviables para su uso en aplicaciones que requieren renderizado en altas cantidades de cuadros por segundo. Particularmente para videojuegos la técnica aplicada por defecto actualmente es la rasterización. Esta alternativa para renderizado en tiempo real, difiere del enfoque de trazado de rayos y hace que sea dificultoso desarrollar aplicaciones que lleguen a resultados con el mismo nivel de realismo.

Las tarjetas gráficas están diseñadas para ejecutar renderizado por rasterización desde que están presentes en computadoras personales. Sin embargo, en los últimos años ha cambiado la convención de que el renderizado en tiempo real debe implementarse exclusivamente con esta técnica. Esto se debe a que la empresa NVIDIA presentó una arquitectura para estos dispositivos capaz de ejecutar aplicaciones ray tracing con aceleración por hardware.

En términos de tarjetas gráficas para computadoras personales, las tarjetas que NVIDIA denomina RTX con arquitectura Turing, son las primeras con hardware dedicado para ray tracing comercialmente disponibles. La aceleración por hardware es implementada con los *RT Cores*, unidades de lógica no programable para detección de intersecciones de rayos en estructuras de geometría tridimensional. A pesar de ser el componente más novedoso de la arquitectura Turing, el hardware específico para ray tracing no es la única característica a destacar por sobre sus predecesoras. Existen aplicaciones que complementan sus algoritmos con procesamiento de imagen vía inteligencia artificial. Dentro de cada unidad de procesamiento se encuentran también *Tensor Cores*, unidades de cálculo especializadas para una suite de servicios impulsados por redes neuronales.

1.1. Objetivo del Proyecto

Como principal objetivo del proyecto se desea aprender sobre la arquitectura Turing, entender la motivación de los cambios realizados respecto a su predecesora Pascal, revisar los nuevos componentes en la arquitectura, y poner a prueba el nuevo conjunto de capacidades en los procesadores gráficos de NVIDIA.

Por otro lado, por la importancia de esta tecnología en la industria del software gráfico, interesa analizar el impacto de estas nuevas funcionalidades en el área. Para esto, se debe conocer de qué manera se adaptaron las distintas API gráficas que acceden al GPU, y cuáles son los nuevos estándares en los que debe estar capacitado un desarrollador para acceder y trabajar con las nuevas funcionalidades.

Los detalles de implementación a bajo nivel de los elementos clave de las tarjetas gráficas RTX como los RT Cores y los Tensor Cores, no son dados a conocer al público de manera oficial por parte NVIDIA. No obstante, sí está disponible una buena cantidad de información que permite entender el propósito y capacidad de cada elemento de la arquitectura Turing. Esta información es suficiente para un proyecto que enfocado en técnicas de ray tracing, ponga a prueba de forma práctica muchas de las afirmaciones planteadas por el fabricante. Para ello se deben implementar aplicaciones que evalúen el hardware bajo las exigencias de ray tracing físicamente realista, ray tracing con denoising basado en inteligencia artificial y ray tracing en tiempo real enfocado para su uso en videojuegos.

1.2. Organización del Documento

En el capítulo 2 se presenta el marco teórico tanto a nivel de algoritmos de trazado de rayos y rasterización, como a nivel de arquitectura. De esta forma se repasan los conceptos más importantes de las técnicas de ray tracing y rasterización que se utilizan en la parte práctica del proyecto.

El análisis del hardware es sólo una parte del alcance del cambio propuesto por NVIDIA con la arquitectura Turing, y para una industria tan grande como la de videojuegos es importante también un análisis a nivel de software. Como parte del marco teórico, en la sección 2.8 también se presentan las distintas formas de acceder al hardware, con foco en las nuevas funcionalidades de la arquitectura Turing.

En el capítulo 3 se presenta las primera de tres aplicaciones basadas en la teoría planteada en el capítulo 2, enfocadas en utilizar todas las cualidades del hardware utilizando las herramientas planteadas en la sección 2.8.

En el capítulo 4 se presenta la segunda aplicación del proyecto como complemento de la presentada en el capítulo 3. Esta segunda aplicación intenta contestar algunas de las preguntas que surgen durante la experimentación en el capítulo anterior y genera las primeras conclusiones respecto a la capacidad de Turing para renderizado físicamente realista.

El capítulo 5 presenta la tercera aplicación del proyecto. En este caso la implementación está enfocada en un nuevo concepto que la arquitectura Turing introduce en la computación gráfica, el renderizado híbrido. Se implementa por medio de una combinación de ray tracing y rasterización, una aplicación de renderizado en tiempo real que saca provecho de los beneficios de ambas técnicas.

Por último el capítulo 6 contiene las conclusiones y las posibilidades de trabajo futuro sobre el resultado del proyecto.

Capítulo 2

Marco Teórico

En este capítulo se presentan los conceptos más importantes de las técnicas de ray tracing y rasterización, junto con el hardware de las tarjetas gráficas capaces de ejecutar estos algoritmos de forma eficiente, poniendo énfasis en la arquitectura Turing con componentes especializados en ray tracing.

2.1. Ecuación de Transporte de Luz

Las interacciones de la luz con la materia son el fundamento de cualquier algoritmo de renderizado físicamente realista. La información de color en un píxel de una imagen renderizada es producto de la simulación de la interacción de la luz con la superficie representada y el medio por el que fue propagada antes de llegar a la cámara.

De forma general, la cantidad de luz que llega a la cámara desde un punto en una superficie está dada por la suma de la luz emitida por ese objeto y la cantidad de luz reflejada. Este valor corresponde a la *radiancia* de salida desde ese punto en dirección al observador. La radiancia se define como la densidad de flujo luminoso por unidad de área y unidad de ángulo sólido [1, Cap. 5]. Existe una formalización de esta idea utilizada para calcular la cantidad de luz que llega a la cámara descrita por Kajiya [2] como ecuación de transporte de luz (ecuación 2.1).

$$L_o(P, \omega_0) = L_e(P, \omega_0) + \int_\Omega f(P, \omega_i, \omega_o)\, L_i(P, \omega_i)\, |\cos \theta_i|\, d\omega_i \tag{2.1}$$

En la ecuación 2.1 se define que la radiancia de salida $L_o(P, \omega_0)$ desde un punto P en dirección ω_o es la radiancia emitida en ese punto en esa misma dirección ($L_e(P, \omega_0)$), más la radiancia incidente desde todas las direcciones de la esfera Ω alrededor del punto P. La multiplicación de la función $f(P, \omega_i, \omega_o)$ y el coseno del ángulo θ_i entre la normal de la superficie y ω_i, indica la proporción de luz que llega a p desde la dirección ω_i y sale por la dirección ω_o. La función f se denomina función de distribución de reflectancia bidireccional (BRDF) y se explica con mayor detalle en la sección 2.2.

El principio clave de la ecuación de transporte de luz es el balance de energía.

Cualquier cambio en la energía debe ser adjudicado a algún elemento del proceso simulado, de forma que se pueda hacer seguimiento de toda la energía del sistema. Se asume que la iluminación es un proceso lineal, y que se cumple el principio de conservación de energía únicamente con los elementos tenidos en cuenta en la simulación.

2.2. Función de Distribución de Reflectancia Bidireccional

Una función de distribución de reflectancia bidireccional (BRDF) determina cuánta luz es reflejada en una dirección cuando cierta cantidad de luz incide desde otra. Este cálculo es realizado en base a las propiedades de la superficie en ese punto. Por ejemplo, si el material tiene reflexión difusa se refleja poca luz y de forma relativamente uniforme en todas las direcciones, sin embargo si se trata de un espejo la luz tiende a ser reflejada en una sola dirección.

La formulación de la BRDF encapsula la definición del material en base a un conjunto de parámetros que no dependen del tipo de iluminación o algoritmo utilizado. Sin embargo la abstracción debe cumplir (en cierto grado) con los siguientes principios para comportarse de acuerdo a un material físicamente realista:

- **Reciprocidad de Helmholtz** – la dirección de entrada y salida pueden ser intercambiadas y la reflectancia resultante debe permanecer igual.
- **Conservación de energía** – la energía saliente más la absorbida por la superficie debe ser igual a la recibida.

Es importante mencionar que estas dos reglas no son seguidas a la perfección por todos los modelos BRDF. Existen modelos empíricos que no satisfacen estos requerimientos, como el caso del modelo *Phong* tratado más adelante en el informe. Aunque algunos modelos no sean físicamente realistas, igual se utilizan en pos de beneficios en rendimiento o facilidad de implementación dándole mejor control a los artistas a la hora de ajustar los parámetros que definen el material [3].

2.2.1. Implementación de una BRDF

A continuación se discuten algunos de los modelos más utilizados como BRDF. El modelo *Lambertiano*, *Phong* y *Oren-Nayar*.

Modelo Lambertiano de Reflexión Difusa

La función lambertiana es la más sencilla implementación de un BRDF y corresponde al término difuso de un material. Es decir, la función lambertiana asume que la luz incidente puede ser dispersada en todas las direcciones posibles del hemisferio alrededor del vector normal de forma uniforme.

Esta BRDF es una buena aproximación del comportamiento de muchos materiales en la vida real y es muy rápida de evaluar, de hecho, es la función utilizada por defecto en prácticamente cualquier motor de renderizado moderno [1].

[1] *Lambert* es el material por defecto en el motor de renderizado de código abierto Godot Engine [4]

La reflectancia lambertiana se basa en que el color difuso de un material es constante en todas las direcciones, este valor es usualmente referido como *albedo* (ρ). A su vez el *albedo* es multiplicado por $1/\pi$ a raíz de la integración de la función con el término coseno en todo el hemisferio, esto asegura la conservación de energía en la ecuación 2.1. De esta forma la función BRDF lambertiana se define en la ecuación 2.2 para una luz proveniente de una dirección ω_i dispersada en dirección ω_o.

$$f(\omega_i, \omega_o) = \frac{\rho}{\pi} \tag{2.2}$$

Modelo Phong de Reflexión Especular

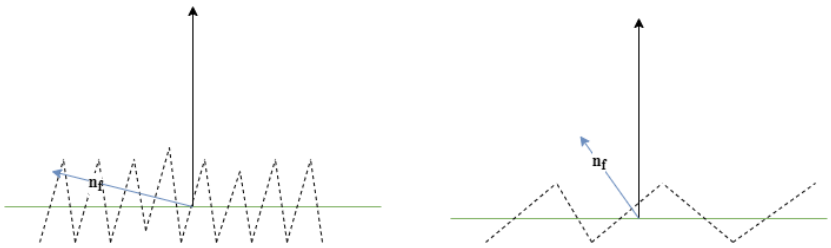
El modelo de reflexión *Phong* es comúnmente utilizado junto con el modelo lambertiano. *Phong* permite agregar de una forma muy eficiente brillos especulares y es utilizado principalmente en aplicaciones sencillas de renderizado en tiempo real.

$$f(L, V) = especularidad \times (R \cdot V)^{brillo} \tag{2.3}$$

La fórmula para el modelo *Phong* se define como la ecuación 2.3 para una luz proveniente de una dirección L dispersada en dirección V. El vector R es la dirección de L reflejada por la normal de la superficie y los valores de especularidad (factor entre 0 y 1) y brillo dependen del material a representar. De manera intuitiva el exponente controla el "ancho" que ocupa el brillo sobre la superficie.

Modelos de Micro Facetado

Muchos modelos de simulación de materiales están basados en la idea de que las superficies no son perfectamente lisas y están cubiertas por pequeñas irregularidades. La forma de representar estas irregularidades es por medio de *microfacetas*, que permiten modelar una superficie a través de la distribución de su orientación estadísticamente. La figura 2.1 muestra la comparación entre una superficie áspera y otra más suave.



(a) Superficie áspera, la distribución de las normales de las microfacetas está representada por el vector n_f

(b) Superficie suave, n_f se acerca más a la normal de la superficie de la geometría

Figura 2.1: Representación de una superficie simulada con microfacetado.

Los modelos basados en microfacetas funcionan modelando estadísticamente la forma en la que se dispersa la luz dada la distribución de los ángulos de las

irregularidades. El modelo Oren-Nayar presentado a continuación se basa en esta teoría.

Modelo Oren-Nayar

El modelo de reflectancia Oren-Nayar [5] se basa en que las superficies ásperas reflejan más luz que la calculada por el modelo lambertiano cuando la dirección del observador se acerca a la dirección de la luz incidente. Es así que los autores lo llaman una generalización del lambertiano.

A diferencia del modelo lambertiano, el modelo Oren-Nayar tiene en cuenta el término especular y no hay necesidad de utilizar el modelo *Phong*. El término especular es deducido únicamente a partir de un parámetro denotado σ que representa el microfacetado de la superficie. Técnicamente σ se define como la desviación estándar de la orientación del ángulo del microfacetado de la superficie.

$$
\begin{aligned}
f(\omega_i, \omega_o) &= \frac{\rho}{\pi}(A + B\,\text{máx}(0, \cos\varphi_i - \varphi_o)\sin\alpha\tan\beta) \\
\\
\alpha &= \text{máx}(\theta_i, \theta_o) \\
\beta &= \text{mín}(\theta_i, \theta_o) \\
A &= 1 - \frac{\sigma^2}{2(\sigma^2 + 0{,}33)} \\
B &= \frac{0{,}45\sigma^2}{\sigma^2 + 0{,}09}
\end{aligned}
\tag{2.4}
$$

El modelo Oren-Nayar está dado por la ecuación 2.4 donde σ está en radianes; θ_i y θ_o son los ángulos entre la normal de la superficie con ω_i y ω_o respectivamente; φ_i y φ_o son los ángulos de la proyección en azimut para ω_i y ω_o en el plano definido por la normal; y ρ es el *albedo* del material.

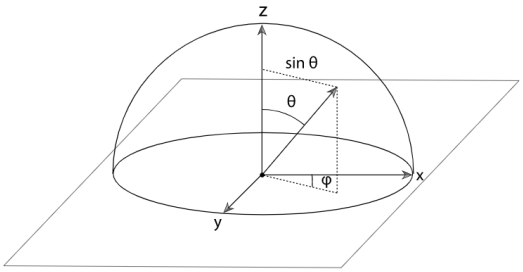


Figura 2.2: Descripción gráfica de los ángulos utilizados en las funciones BRDF [1]

2.2.2. Solución numérica de la ecuación de transporte de luz

La complejidad de los modelos físicos BRDF y la distribución de geometría de forma arbitraria en una escena conspira contra una posible solución numérica para la ecuación de transporte de luz. Sin embargo. gracias a la combinación de algoritmos de trazado de rayos e integraciones Monte Carlo, es posible manejar esta complejidad sin la necesidad de imponer restricciones de implementación.

Una forma de llegar a una solución numérica para la ecuación de transporte de luz, es hacer explícito el uso de una función de trazado de rayos. Esta expresión es fundamental para explicar el algoritmo path tracing implementado en el capítulo 3.

En primer lugar se reescribe la ecuación 2.1 como una integral del área, es decir, sobre todas las superficies de la escena en lugar de una integral de las direcciones de una esfera. Para esto se realizan las siguientes definiciones.

La radiancia saliente de un punto p' captada desde un punto p:

$$L(p' \to p) = L(p', \omega) \tag{2.5}$$

Si p' y p son mutuamente visibles y $\omega = \widehat{p - p'}$, la función BRDF en p' se puede representar como:

$$f(p'' \to p' \to p) = f(p, \omega_i, \omega_o), \tag{2.6}$$

donde $\omega_i = \widehat{p'' - p'}$ y $\omega_0 = \widehat{p - p'}$ (figura 2.3).

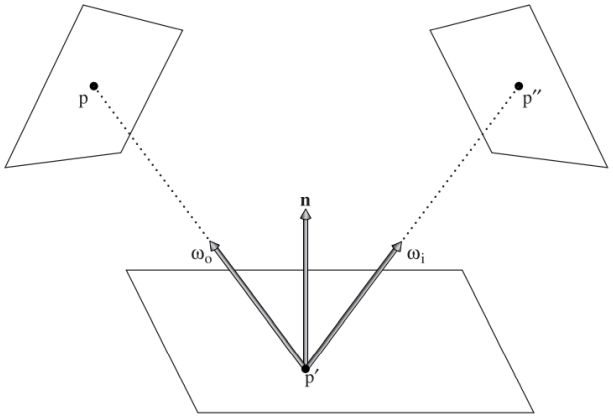


Figura 2.3: La ecuación de transporte de luz representada con tres puntos para convertir la integral al dominio de puntos de las superficies en la escena. [1]

Por último es necesario multiplicar el Jacobiano que relacione un ángulo sólido con un área para transformar la integral sobre dirección a integral sobre el área de la superficie cubierta. Esto es $|\cos\theta'|/r^2$, donde θ' corresponde al ángulo entre la normal de la superficie del punto p' y su dirección ω_i, y r a la distancia entre p y p'. Este cambio de variables se combina con el término $|\cos\theta|$ original de la ecuación de transporte de luz para el punto p, y una función binaria que evalúa la visibilidad $V(p \leftrightarrow p')$ entre los dos puntos en cuestión (1 si son mutuamente visibles o 0 en caso contrario). Luego el término geométrico se define como $G(p \leftrightarrow p')$, de la siguiente forma:

$$G(p \leftrightarrow p') = V(p \leftrightarrow p') \frac{|\cos\theta|\,|\cos\theta'|}{\|p - p'\|^2} \tag{2.7}$$

Utilizando estas definiciones se puede escribir la ecuación de transporte de luz para el conjunto A de todas las superficies de la escena:

$$L(p' \to p) = L_e(p' \to p) + \int_A f(p'' \to p' \to p)\, L(p'' \to p')\, G(p'' \leftrightarrow p')\, dA(p'') \quad (2.8)$$

Para resolver la ecuación 2.1 con un método Monte Carlo, hay que obtener un número de muestras a partir de una distribución de direcciones en una esfera y lanzar rayos para evaluar el integrando. En el caso de la ecuación 2.8, el procedimiento consiste en elegir un número de puntos en las superficies de la escena de acuerdo a una distribución sobre el área total y computar el integrando para cada uno, trazando rayos para evaluar el término $V(p \leftrightarrow p')$. Ambos métodos son utilizados en el capítulo 3 para la implementación de un algoritmo de trazado de rayos. El primero al momento de obtener iluminación indirecta incidente en un punto y el segundo al momento de obtener iluminación directa proveniente de luces de área representadas con geometría visible en la escena.

Integral de caminos

Partiendo de la forma integral sobre el área (ecuación 2.8), es posible deducir una formulación de transporte de luz basada en *integral de caminos*. La motivación de este nuevo planteo basado sobre un *espacio de caminos* es permitir que cualquier técnica que elija caminos de forma aleatoria (como *path tracing*) pueda ser transformada en un algoritmo de renderizado que computa una respuesta correcta dada un número suficiente de muestras.

Para ir de la integral de área a una suma de integrales de caminos involucrando múltiples recorridos de la luz de largo variable, se debe expandir la ecuación 2.8 basada en tres puntos (p, p' y p'') sustituyendo recursivamente el término $L(p'' \to p')$ del lado derecho de la ecuación:

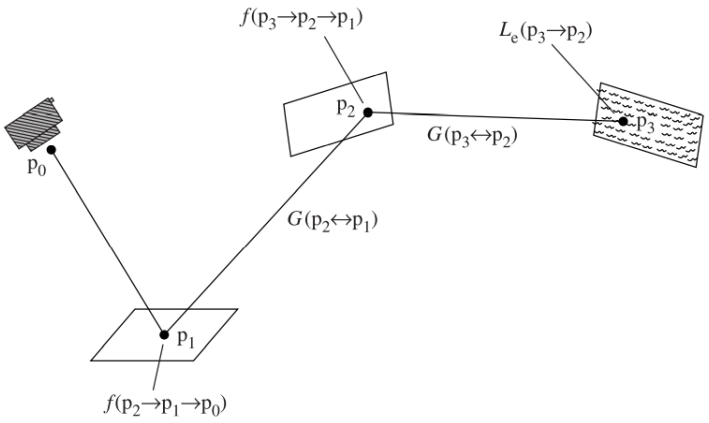


Figura 2.4: Contribución de un camino de profundidad dos partiendo del punto p_1 captada desde el punto p_0. Los componentes de la ecuación 2.9 son mostrados en la imagen. [1]

$$
\begin{aligned}
L(p_1 \to p_0) = {} & L_e(p_1 \to p_0) \\
& + \int_A L_e(p_2 \to p_1)\, f(p_2 \to p_1 \to p_0)\, G(p_2 \leftrightarrow p_1)\, dA(p_2) \\
& + \int_A \int_A L_e(p_3 \to p_2)\, f(p_3 \to p_2 \to p_1)\, G(p_3 \leftrightarrow p_2) \\
& \qquad f(p_2 \to p_1 \to p_0)\, G(p_2 \leftrightarrow p_1)\, dA(p_3)\, dA(p_2) \\
& + \ldots
\end{aligned}
\tag{2.9}
$$

En la ecuación 2.9 cada término del lado derecho representa un nuevo trazo de mayor profundidad en el camino. Para simplificar el planteo, se reescribe la ecuación de la siguiente forma:

$$
L(p_1 \to p_0) = \sum_{i=1}^{\infty} P(\bar{p_i}), \tag{2.10}
$$

$P(\bar{p_n})$ retorna la radiancia dispersada por el camino $\bar{p_n}$ con $n+1$ puntos:

$$
\begin{aligned}
P(\bar{p_n}) = {} & \underbrace{\int_A \int_A \cdots \int_A}_{\text{n - 1}} L_e(p_n \to p_{n-1}) \\
& \left(\prod_{i=1}^{n-1} f(p_{i+1} \to p_i \to p_{i-1})\, G(p_{i+1} \leftrightarrow p_i) \right) dA(p_2) \cdots dA(p_n).
\end{aligned}
\tag{2.11}
$$

Limitando la ecuación 2.10 para una profundidad máxima n definida, se puede computar la radiancia arribando a p_0 y pasando por p_1 utilizando una estimación Monte Carlo. Es así como la formulación presentada en esta sección es la base de *path tracing*, una de las técnicas de trazado de rayos más utilizadas.

El algoritmo *path tracing* se puede definir como la simulación del camino de un rayo de luz desde que se origina, hacia la cámara o el camino inverso desde la cámara hacia una fuente de luz. La generación masiva de rayos aproxima el comportamiento de la luz en la naturaleza y hace posible la simulación de efectos ópticos como cáusticas, reflexiones o refracciones.

2.3. Ray Tracing

Actualmente el término ray tracing es comúnmente utilizado como sinónimo de *path tracing*, siendo este método que definió James Kajiya [2] un estándar para la generación de imágenes fotorrealistas. Sin embargo existen numerosas formas de implementar algoritmos de trazado de rayos. La principal característica de estos algoritmos es la capacidad de simular el comportamiento de la luz teniendo en cuenta las interacciones con los objetos y sus materiales en un espacio tridimensional. Estos algoritmos tienen naturaleza recursiva y en cada interacción con una superficie se decide hacia donde continúan uno o más rayos.

2.3.1. Trazado Hacia Adelante

La forma más intuitiva de simular el comportamiento de la luz puede ser generando fotones desde la posición de las distintas fuentes de iluminación. Luego, a raíz de la interacción con cada superficie, se espera que esos fotones lleguen a la posición del observador. A este tipo de algoritmos se los caracteriza como de *trazado hacia adelante*.

Para renderizado en tiempo real este tipo de técnicas no se utilizan demasiado, porque la cantidad de fotones que se deben simular para generar una imagen de buena calidad, no escala en relación al tamaño de las escenas representadas. La figura 2.5 representa un típico caso en el que únicamente una pequeña porción de fotones emitidos por la fuente de luz terminan llegando a la cámara y por lo tanto contribuyendo en el resultado de la imagen. En este caso se estaría desperdiciando mucho poder de cómputo.

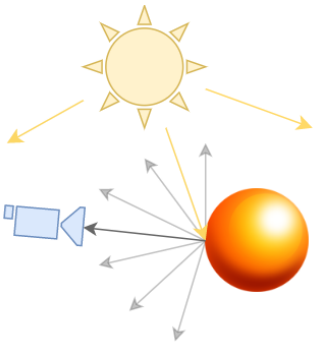


Figura 2.5: Trazado de rayos hacia adelante.

En la actualidad existen algunos algoritmos de trazado hacia adelante de gran utilidad para generar efectos relevantes en imágenes físicamente realistas, como cáusticas de luz con *photon mapping* o *bidireccional path tracing*. Pero estos algoritmos no dejan de tener un costo computacional muy alto y en general son aplicados de forma estocástica y estratificando la emisión de fotones de modo que sean de utilidad específica para algunos efectos en el resultado final del render [1, Cap. 16].

2.3.2. Trazado Hacia Atrás

En lugar de trazar rayos desde la fuente de luz hacia el observador, es posible revertir esta lógica y trazar rayos desde el observador hacia las diferentes fuentes de iluminación. Con las excepciones mencionadas en la sección anterior, esta es la forma de proceder para los algoritmos de trazado de rayos. El objetivo de este enfoque es invertir tiempo de cálculo únicamente en las interacciones de los rayos que llegan al observador.

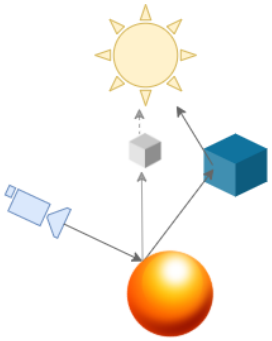


Figura 2.6: Trazado de rayos hacia atrás.

Los rayos en el caso de trazado hacia atrás parten de una cámara ubicada en algún lugar de un espacio tridimensional. Los primeros rayos lanzados desde la cámara son *rayos primarios*, y el objetivo de cada uno es obtener el color resultante para cada uno de los $W \times H$ píxeles correspondientes a la resolución de salida.

2.3.3. Estructuras de Aceleración

El concepto de rayo en ray tracing es representado con un punto en el espacio para su origen, una dirección y opcionalmente un largo máximo. Un rayo es entonces una semirrecta a la que se le obtienen puntos de intersección con todos los objetos de la escena, tal que el más cercano al origen representa una colisión. Obtener la colisión más cercana aplica para cualquier algoritmo de ray tracing y es una tarea que se debe realizar constantemente. Turner Whitted determinó durante las pruebas realizadas sobre su algoritmo (*Whitted ray tracing*) que para escenas complejas se podría invertir más del 95 % del tiempo de renderizado en cálculo de intersecciones [6].

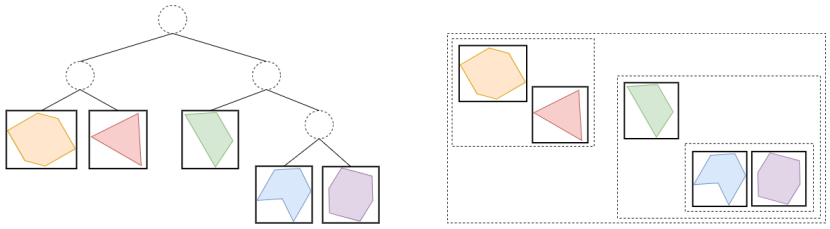
Un recorrido lineal por los triángulos de la escena, donde a priori no se conoce el orden en el que se encuentran en el espacio representado, podría implicar iterar sobre decenas de miles de elementos por rayo en cada cuadro renderizado. La herramienta usualmente utilizada para optimizar y mejorar el orden del cálculo de colisiones son las *estructuras de aceleración*. Una estructura de aceleración es una estructura de datos que agrupa en taxonomías y ordena la geometría en una escena con diversas estrategias para acelerar la consulta de una intersección entre entidades.

Jerarquías de volúmenes acotantes (BVH)

Un caso particular muy utilizado para renderizado en tiempo real son las jerarquías de volúmenes acotantes (*bounding volume hierarchies*). Un volumen acotante es un volumen relativamente sencillo, como una esfera o una caja que encierra una geometría o un grupo de objetos de mayor complejidad. Las cajas son usualmente implementadas como cajas alineadas respecto a los ejes de coordenada, y reciben el nombre de AABBs (*axis-aligned bounding boxes*), pero también existen variantes como OBBs orientadas respecto a un eje determinado.

Utilizando entidades como las AABBs las escenas se pueden organizar en estructuras de árboles finitarios tales que únicamente los nodos hoja contienen volúmenes con geometría. Pueden existir múltiples niveles de nodos encerrando

tanto nodos hoja como otros nodos intermedios en forma de volúmenes acotantes.



(a) Representación del árbol generado para un BVH

(b) Representación del BVH en el espacio

Figura 2.7: Jerarquía de volúmenes limitantes.

Supongamos que se pretende efectuar el chequeo de intersección de un rayo en una escena con un BVH. Si el rayo no se intersecta con el nodo raíz del BVH, entonces ya es posible afirmar que ese rayo no colisionará con ningún objeto. En caso contrario, el mismo chequeo se debe hacer para cada hijo directo del nodo raíz de manera recursiva hasta llegar a una hoja. Una vez que se llega a una hoja, se debe iterar por cada triángulo de la geometría correspondiente y de esa forma determinar si existe una intersección o no.

Típicamente es de interés para optimizar y simplificar la búsqueda de elementos en un árbol finitario, que tenga la forma de árbol *n-ario* y que además esté balanceado. Esto es, que todos sus nodos hoja tengan profundidad h o $h-1$, siendo posible afirmar para este árbol balanceado que $h = \log_n k$ donde k es la cantidad total de nodos del árbol. Luego $O(\log_n k)$ es el orden de complejidad de la búsqueda.

Árboles BSP

La estructura BVH es la mejor opción para cualquier tipo de escena sin importar la complejidad y forma de la geometría representada, por esta razón es la opción utilizada actualmente por el hardware de trazado de rayos. Sin embargo también existen otros tipos de estructuras basadas en la partición del espacio en sustitución de la utilización de volúmenes. A partir de divisiones del espacio tridimensional, se generan árboles BSP (*binary space partition*) que contienen la información de los objetos ubicados en cada partición.

Una ventaja de este tipo de estructuras sobre las BVH, es la posibilidad de realizar consultas basadas en el orden de aparición de los elementos en el espacio. En una BVH la detección de una intersección no asegura que sea la más cercana al origen. Para el caso de un algoritmo de trazado de rayos, detectar una primera colisión en un BVH requiere varias iteraciones.

2.4. Integración Monte Carlo

Las aplicaciones implementadas en el capítulo 3 y 4 deben computar la radiancia a lo largo de los caminos de los rayos de luz hacia la cámara. Para realizar esta tarea utilizando las ecuaciones integrales que describen el comportamiento de la luz, se debe recurrir a métodos de integración numérica Monte Carlo. Estos métodos utilizan la aleatoriedad para evaluar integrales con una tasa de convergencia que es independiente de la dimensionalidad del integrando. En esta sección, se revisan los

conceptos importantes de probabilidad y las bases para usar técnicas Monte Carlo con el fin de evaluar las integrales clave en las técnicas de renderizado utilizadas en este proyecto.

2.4.1. Introducción

La integración de Monte Carlo es un método que utiliza muestreo aleatorio para estimar los valores de integrales. Una propiedad muy importante de Monte Carlo es que solo se necesita capacidad de evaluar una $f(x)$ integrada en puntos arbitrarios del dominio para estimar el valor de su integral $\int f(x)\,dx$. Esta propiedad no solo hace que Monte Carlo sea fácil de implementar, sino que también hace que la técnica sea aplicable a una variedad de integrandos, incluidos los que contienen discontinuidades.

Muchas de las integrales que surgen en el renderizado son difíciles o imposibles de evaluar directamente. Por ejemplo, para calcular la cantidad de luz reflejada por una superficie en un punto de acuerdo con la ecuación 2.1, se debe integrar el producto de la radiancia incidente y el BRDF sobre la esfera unitaria.

La integración de Monte Carlo permite estimar la radiancia reflejada simplemente eligiendo un conjunto de direcciones sobre la esfera. Para cada dirección se calcula la radiancia incidente, multiplicando por el valor de BRDF y aplicando un término de ponderación. Los BRDF arbitrarios, las descripciones de las fuentes de luz y la geometría de la escena se manejan fácilmente; la evaluación de cada una de estas funciones en puntos arbitrarios es todo lo que se requiere.

La principal desventaja de Monte Carlo es que si se utilizan n muestras para estimar la integral, el algoritmo converge al resultado correcto a una tasa $O(n^{-1/2})$. En otras palabras, para reducir el error a la mitad, es necesario evaluar cuatro veces más muestras. Al renderizar, cada muestra generalmente requiere que se tracen uno o más rayos en el proceso de calcular el valor del integrando, algo computacionalmente costoso de soportar cuando se usa Monte Carlo para la síntesis de imágenes. En las imágenes, los artefactos del muestreo de Monte Carlo se manifiestan como ruido: los píxeles son aleatoriamente demasiado brillantes o demasiado oscuros.

2.4.2. Variables aleatorias continuas

Cuando se renderiza una imagen, las variables aleatorias discretas son menos comunes que las variables aleatorias continuas, se toman valores en rangos de dominios continuos (por ejemplo, los números reales, las direcciones en la esfera unitaria o las superficies de la geometría en la escena).

Una variable aleatoria particularmente importante es la *variable aleatoria uniforme canónica*, denotada ξ. Esta variable toma todos los valores en su dominio $[0, 1)$ con igual probabilidad. En particular ξ es importante por dos razones. Primero, es fácil generar una variable con esta distribución en software; la mayoría de las bibliotecas en tiempo de ejecución tienen un generador de números pseudoaleatorios que hace precisamente eso. En segundo lugar, es posible generar muestras a partir de distribuciones arbitrarias comenzando primero con variables aleatorias uniformes canónicas y aplicando una transformación apropiada.

Otro tipo de variable aleatoria continua puede variar sobre cierto rango de números reales con una densidad definida por una función arbitraria. La *función*

de densidad de probabilidad (PDF) formaliza esta idea y describe la probabilidad relativa de que una variable aleatoria tome un valor particular (ecuación 2.13).

La PDF $p(x)$ es la derivada de una función $P(x)$ de distribución acumulada (CDF) de la variable aleatoria. La CDF de una variable aleatoria es la probabilidad de que un valor de la distribución de la variable sea menor o igual a algún valor x (ecuación 2.12). Para el ejemplo de lanzar un dado, $P(2) = 1/3$, ya que dos de las seis posibilidades son menores o iguales que 2.

$$P(x) = Pr\{X \leq x\}, \tag{2.12}$$

$$p(x) = \frac{dP(x)}{dx} \tag{2.13}$$

2.4.3. Estimador Monte Carlo

Dada una PDF $p(x)$ arbitraria (distinta de cero) de las variables aleatorias X_i, el estimador de Monte Carlo básico que se aproxima al valor de una integral arbitraria, puede definirse como la ecuación 2.14:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{2.14}$$

El número de muestras N se puede elegir arbitrariamente, independientemente de la dimensión del integrando, por lo que el número de muestras tomadas en Monte Carlo es completamente independiente de la dimensionalidad de la integral. El error en el estimador de Monte Carlo disminuye a una tasa de $O(\sqrt{N})$ en el número de muestras tomadas.

2.4.4. Muestreo con Importancia

El muestreo de importancia es una técnica de reducción de la varianza, que aprovecha el hecho de que el estimador de Monte Carlo converge más rápidamente si las muestras se toman de una distribución $p(x)$ similar a la función $f(x)$ en el integrando. La idea básica es que al concentrar el trabajo donde el valor del integrando es relativamente alto, una estimación precisa se calcula de manera más eficiente.

Por ejemplo, al evaluar la ecuación de transporte de luz, ecuación 2.1. Si se muestrea aleatoriamente una dirección que es casi perpendicular a la normal de la superficie, el término del coseno será cercano a 0. Todo el gasto de evaluar el BRDF y trazar un rayo para encontrar la radiancia entrante en esa ubicación de muestra es un desperdicio, ya que la contribución al resultado final será minúscula. En general, es mejor si se toman muestras reduciendo la probabilidad de elegir direcciones cerca del horizonte. De manera más general, si se muestrean direcciones a partir de distribuciones que coinciden con otros factores del integrando, la eficiencia se mejora de forma similar.

Método de Malley (Muestreo de Hemisferio Ponderado por Coseno)

El método de Malley genera direcciones que tienen más probabilidades de estar cerca de la parte superior del hemisferio, donde el término coseno tiene un valor mayor que el de la parte inferior. La idea detrás del método de Malley es que si se eligen puntos uniformemente del disco unitario y luego generamos direcciones proyectando los puntos en el disco hasta el hemisferio por encima de él, la distribución de direcciones resultante tendrá una distribución de coseno.

El método de Malley es utilizado para la implementación de las aplicaciones del capítulo 3 y 4. Una particularidad de este método, es que el valor del PDF corresponde a: $\cos\theta/\pi$. Como el PDF es el divisor del estimador Montecarlo (ecuación 2.14), aplicar este muestreo resulta en la cancelación del termino $1/\pi$ de la ecuación 2.2 del modelo Lambertiano de reflexión difusa aplicada como BRDF.

2.5. Reducción de Ruido

La integración Monte Carlo de la iluminación indirecta conduce a imágenes muy ruidosas a bajas tasas de muestreo, de modo que gran parte de la energía se concentra en un pequeño subconjunto de caminos o píxeles. Particularmente para juegos y otras aplicaciones en tiempo real, la baja cantidad de muestras por píxel es una característica imposible de evitar. Desafortunadamente, incluso los trazadores de rayos más rápidos solo pueden trazar unos pocos rayos por píxel a 1080p y 30Hz. Si bien este número se duplica cada pocos años, la tendencia es (al menos parcialmente) contrarrestada por el movimiento hacia pantallas de mayor resolución y frecuencias de actualización más altas [7].

Existen diversas estrategias para reducir el ruido inherente a las representaciones de Monte Carlo. Por lo general, se enmarca el problema como una reconstrucción de la imagen final (en lugar de eliminar el ruido) a partir de las muestras del render, ya que, a tasas de muestreo prohibitivamente bajas, muchas áreas de la imagen tienen casi solo ruido. A modo de ejemplo para renderizado en tiempo real, una de las técnicas mas utilizadas es el filtro SVGF (filtrado guiado por varianza espacio-temporal) [8].

SVFG es un algoritmo de reconstrucción que genera una secuencia de imágenes temporalmente estable a partir de un solo camino de iluminación global trazado por píxel. Para manejar una entrada ruidosa, utiliza la acumulación temporal para aumentar el recuento de muestras efectivo y las estimaciones de varianza de luminancia espacio-temporal para impulsar un filtro que permite distinguir entre ruido y detalle a múltiples escalas utilizando la variación de luminancia local.

Por otro lado y particularmente para este proyecto, son de especial interés otro tipo de técnicas basadas de la utilización de redes neuronales, ya que uno de los objetivos es poner a prueba las nuevas capacidades de la arquitectura Turing en este rubro. La publicación «Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder» [7] describe una técnica de aprendizaje automático utilizada para reconstruir secuencia de imágenes renderizadas utilizando métodos Monte Carlo. Este trabajo es la base de la implementación de la herramienta OptiX Denoiser, utilizada en este proyecto para eliminación de ruido de iluminación global como producto de la implementación del algoritmo path tracing.

El denoiser de NVIDIA acelerado por IA se entrenó utilizando decenas de miles de imágenes renderizadas a partir de mil escenas en 3D. Los datos de entrenamiento se entregaron a una red neuronal similar a la descrita en la publicación. El resultado se incluye en el OptiX SDK que funciona en una gran cantidad de escenas y permite configurar distintos modos de ejecución con resultados que varían en calidad y rendimiento de ejecución. De la misma forma que el filtro SVFG, este denoiser ofrece la posibilidad de ejecutar un modo «temporal», que se retroalimenta con la salida del cuadro anterior e información de traslación de cada píxel en espacio de pantalla.

El alcance y resultados de OptiX Denoiser son discutidos en profundidad en el capítulo 4, junto con la implementación de una aplicación que aplica en cada cuadro estas funciones provistas por el SDK.

2.6. Rasterización

2.6.1. Introducción

Las técnicas de reducción de ruido descritas en la sección anterior son muy recientes en el mundo de la computación gráfica, sin ellas, los métodos de implementación de ray tracing más eficientes como los basados en integración Monte Carlo, son inviables cuando se tiene en cuenta la exigencia del renderizado en tiempo real. Es este el motivo por el cual la técnica usualmente utilizada en videojuegos no es ray tracing, sino la rasterización.

En esta sección se describe esta alternativa para renderizado, sus diferencias con el enfoque de ray tracing y las ventajas que provee su implementación dadas las características del hardware de la tarjeta gráfica convencional.

2.6.2. Comparación con Ray Tracing

En un algoritmo de trazado de rayos se recorre una imagen formada con píxeles, y para cada uno se lanza un rayo primario hacia la escena. Luego se itera sobre las primitivas y se determina cual se encuentra más cerca:

```
Para cada píxel:
    Para cada primitiva:
        ¿primitiva más cercana?
```

Pero si el bucle se plantea recorriendo inicialmente cada primitiva en lugar de los píxeles, se obtiene el algoritmo *z-buffering* en el que se basa el pipeline de rasterización. En este caso se calcula para cada primitiva su profundidad correspondiente en cada píxel de la imagen de salida:

```
Para cada primitiva:
    Para cada píxel:
        ¿primitiva más cercana?
```

Tomando en cuenta las estructuras de aceleración se podría plantear que el primero de los dos algoritmos es más eficiente y la rasterización sería el de “fuerza bruta”. Sin embargo, *z-buffering* es la solución aplicada de forma estándar para renderizado en tiempo real producto de diversas optimizaciones aplicadas con la ayuda del hardware gráfico.

El cálculo de un buffer de profundidad permite simplificación de geometría basada en distancia (*level-of-detail*) y la omisión de primitivas ocultas (*occlusion culling*). A su vez, las estructuras de aceleración también encuentran aplicabilidad en el pipeline de rasterización durante las etapas específicas de procesamiento de geometría del pipeline de rasterizado. Con la ayuda de un BVH es posible aplicar la técnica *hierarchical frustum culling*, que permite remover objetos que se encuentran fuera de cámara.

Muchas de las optimizaciones realizadas gracias a *z-buffering* no tienen sentido en algoritmos de trazado de rayos, por su naturaleza recursiva y la forma en la que se tienen en cuenta todos los objetos de una escena, visibles o no. Es esta la razón por la que el hardware gráfico históricamente ha sido optimizado para la rasterización, y el desarrollo de aplicaciones de renderizado en tiempo real como videojuegos ha evolucionado por décadas utilizando esta técnica.

2.6.3. Pipeline de Rasterización

Formalmente el concepto de rasterización es el proceso de llevar una imagen descrita en un formato gráfico vectorial a un espacio de píxeles como una imagen de mapa de bits o la pantalla de una computadora. A nivel de implementación, el rasterizado es parte un pipeline que inicia en el procesamiento de la geometría vértice por vértice y finaliza con el procesamiento de cada uno de los píxeles.

Una construcción básica de este pipeline consiste en cuatro etapas; la salida de la aplicación, procesamiento de geometría, rasterización y procesamiento de píxeles. Cada una de estas etapas puede contener su propio pipeline interno [9, Cap. 2].

Geometría

Para el procesamiento de geometría es importante mencionar algunas subetapas que están presentes en cualquier implementación:

- **Vertex** – Procesamiento de cada vértice en el espacio tridimensional.
- **Projection** – Cálculo del tipo de proyección geométrica.
- **Clipping** – Recorte de la geometría que no será utilizada.
- **Screen Mapping** – Mapeo de coordenadas al espacio de pantalla (resolución y ratio de aspecto).

También existen etapas opcionales como el teselado para optimizar la cantidad de triángulos o la generación automática de geometría, útil para la simulación de partículas con volumen a partir de unos pocos vértices.

Rasterizado

Una vez completada la etapa de geometría se continúa con la etapa específicamente dedicada al rasterizado. Aquí se trabaja sobre un subconjunto de triángulos, al que se les asignan fragmentos de píxel [2] con información de profundidad y la información procesada en la etapa de geometría.

[2]El concepto de fragmento suele confundirse con el concepto de píxel, el píxel es el resultado de procesar uno o más fragmentos generados en esta etapa.

El criterio en el que se asigna un fragmento a un triángulo puede depender del programador [3], pero típicamente se asignan únicamente los fragmentos cuyo centro se encuentra dentro del área del triángulo proyectada (figura 2.8). Si ningún fragmento fuera asignado a un triángulo, no sería procesado en la siguiente etapa y por lo tanto no sería parte del resultado final del pipeline.

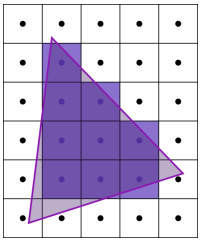


Figura 2.8: Proceso de rasterización de un triángulo.

Píxel

Con los fragmentos generados para toda geometría potencialmente visible, es necesario pasar por la última etapa del pipeline para calcular el valor resultante de cada píxel.

Algunas de las etapas antes mencionadas son ejecutadas por hardware dedicado no programable o son responsabilidad del controlador, y en contraste con la ganancia en rendimiento el programador pierde control sobre la lógica de ejecución. Son pocas las etapas programables en procesamiento de geometría y rasterizado, típicamente la proyección de vértices, y en tarjetas gráficas modernas, algunas etapas opcionales para generación de geometría extra.

Para el procesamiento de píxeles existe mayor libertad de utilizar lógica programable que determina qué hacer con cada fragmento. Con la información proveniente de la etapa de geometría normalmente el programador asigna texturas y calcula el sombreado en cada uno de los fragmentos para determinar el color exacto del material que es proyectado.

Una vez determinado el color (con información de transparencia) de cada fragmento, la información de profundidad que fue asignada en la etapa de rasterizado es necesaria para finalizar el último paso del pipeline. Aquí se realiza el mezclado de color (*blending*) con todos los fragmentos correspondientes al píxel. El programador elije sobre un conjunto de posibles funciones fijas de mezcla de color para que la unidad de operaciones de rasterizado [4] recorra y determine en orden $O(n)$ el color final del píxel, donde n es la cantidad de primitivas [5].

[3] Tanto Vulkan como *DirectX* proveen extensiones para activar "rasterizado conservativo", esto permite tener en cuenta toda al área del fragmento en lugar de solo el punto central para asignarlos a un triángulo.

[4] La unidad de operaciones de rasterizado es usualmente abreviada ROP o ROU dependiendo del fabricante de GPU

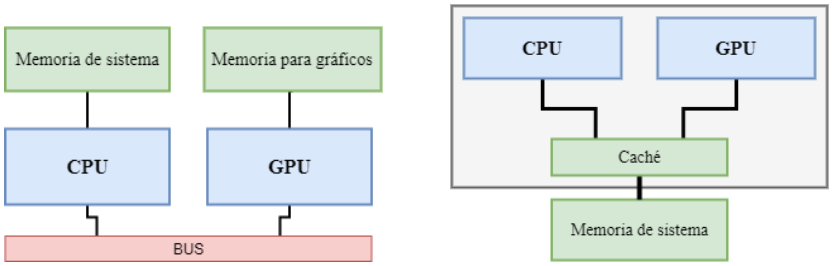
[5] Para casos de generación de anti-alias con super sampleo este proceso se puede repetir varias veces para cada píxel

2.7. Arquitectura del GPU

A continuación se analiza el hardware requerido para ejecutar los métodos planteados en las secciones anteriores, en particular se presentarán los nuevos componentes de la arquitectura Turing para aceleración de trazado de rayos e inteligencia artificial.

2.7.1. Memoria

La unidad de procesamiento gráfico (GPU) depende de la unidad de procesamiento central (CPU) para la ejecución de cualquier tipo de cómputo. Un sistema convencional con GPU puede implementar la interacción entre estas unidades de dos formas, discreta o integrada [10]. Los sistemas con GPU discreta separan la memoria de sistema, de la memoria de la tarjeta gráfica, donde la CPU y GPU acceden a cada una por separado respectivamente. En este caso el pasaje de información entre un subsistema y otro se realiza a través de un bus (típicamente PCI express). Por otro lado los sistemas con GPU integrada colocan ambos procesadores en un mismo subsistema comunicado con una memoria unificada (figura 2.9).



(a) Sistema con GPU discreta

(b) Sistema con CPU y GPU integrados

Figura 2.9: Comparación de sistemas integrado y discreto.

La cualidad principal de una GPU es la capacidad de ejecutar grandes cantidades de instrucciones de manera concurrente. El modelo de ejecución implementado (SIMT: *Single Instruction Multiple Threads*) explota esta característica ejecutando para cada instrucción múltiples hilos de ejecución. Como consecuencia de la gran cantidad de datos potencialmente requeridos para cada instrucción, se requiere de una buena gestión del ancho de banda de acceso a memoria. Si bien múltiples niveles de memoria y caché ayudan a mitigar esta necesidad, las GPU más potentes tienden a ser del tipo integradas, donde se utiliza una tecnología de memoria distinta a la utilizada para el CPU. Típicamente para el CPU se utiliza memoria con tecnología DDR que prioriza baja latencia y en el GPU tecnología GDDR que prioriza alto ancho de banda.

2.7.2. Unidades de Cómputo

En contraste con un CPU en el que gran parte del hardware está dedicado al control de ejecución, el elemento más importante de la arquitectura de un procesador gráfico son las unidades de cómputo, o en terminología de NVIDIA, *streaming multiprocessors* (SM). Una GPU moderna está compuesta por muchas unidades de cómputo, cada una con múltiples caché de instrucciones y de datos pero también

una pequeña memoria compartida a la que se puede acceder aplicando barreras de ejecución para sincronización entre potencialmente miles de hilos ejecutados en cada núcleo.

En la arquitectura Turing los SM son una versión reinventada de los SM utilizados en la arquitectura Pascal (predecesora a Turing para computadoras personales). Comparados con los SM de Pascal, son superiores en capacidad de cómputo [11] [12], sin embargo en teoría ofrecen mejoras en eficiencia que no están directamente relacionadas con la cantidad de ALU unificadas o CUDA cores en terminología de NVIDIA.

Muchas de las modificaciones aplicadas en Turing están inspiradas en el diseño de la arquitectura Volta también de NVIDIA. La particularidad de la arquitectura Volta es que fue destinada para ser utilizada en tarjetas gráficas de centros de cómputo masivo y no para uso doméstico, su característica principal es la capacidad para procesamiento de redes neuronales [13].

Unidades de Procesamiento de Enteros

En la arquitectura Pascal las instrucciones de trabajo procesadas por cada núcleo son una mezcla de operaciones complejas de adición y multiplicación de tipo punto flotante, con operaciones más simples de adición de números enteros. Las operaciones de números enteros son usualmente ejecutadas en aritmética de punteros para el cálculo de direcciones de memoria. En la arquitectura Volta y Turing se evita que las operaciones de punto flotante queden esperando por la finalización de este tipo de operaciones [13].

Cada bloque de ejecución dentro de las nuevas SM se divide en unidades de procesamiento de enteros de 32 bits (INT32) y unidades de procesamiento de punto flotante de 32 bits (FP32). De esta forma para cada ciclo de ejecución las unidades INT32 pueden ejecutar aritmética de punteros para obtener los datos de la próxima instrucción mientras se procesa en paralelo la actual con las unidades FP32.

Para cada unidad FP32 (núcleo CUDA) existe otra INT32 dentro del mismo bloque, y cada bloque de la SM (cuatro en Turing) tiene 16 unidades FP32 [6].

Tensor Cores

Los *Tensor Cores* son otra gran modificación en los SM de Turing también introducida en la arquitectura Volta. Sin embargo este tipo de unidades son una tecnología que se asemeja mucho a las unidades de procesamiento tensorial o TPU creadas por Google para utilización en sus centros de cómputo masivo. Las TPU no contienen ningún tipo de hardware para procesamiento gráfico, pero están optimizadas para aprendizaje automático con redes neuronales. En comparación con una unidad aritmético lógica de un núcleo de un GPU estándar, estas unidades fueron creadas para un mayor volumen de cálculo de menor precisión. Las TPU están especialmente diseñadas para multiplicación de matrices.

Para el caso de los *Tensor Cores* de NVIDIA, se especifica que cada uno es capaz de operar con matrices de tamaño 4×4 para ejecutar operaciones de la forma

[6] En NVIDIA Ampere (2020) [14], las unidades INT32 vuelven a ser multipropósito pudiendo ejecutar también operaciones FP32 cuando no son utilizadas, duplicando eventualmente la cantidad de núcleos CUDA de 64 a 128 por SM.

$D = A \times B + C$. En el caso de Volta [13] se limitó a precisión de 16 bits de punto flotante para multiplicación, con la posibilidad de sumar la matriz C con precisión 32 bits (figura 2.10). En la arquitectura Turing se agregaron los modos de precisión INT4, INT8, FP16 y FP32 tanto para suma como para multiplicación [12].

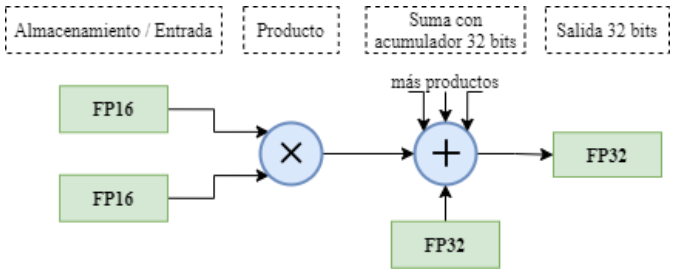


Figura 2.10: Multiplicación y adición (FMA) en un núcleo tensor de NVIDIA Volta [13].

En la práctica el controlador permite utilizar el hardware para multiplicar matrices de mayor tamaño y existen diversas interfaces de programación de aplicaciones como CUDA-C++ que permiten el acceso a esta funcionalidad. La GPU TU102 [12] de arquitectura Turing contiene ocho Tensor Cores por SM, cada uno puede realizar hasta 64 operaciones FP16 (64 FLOPS) de multiplicación y acumulación (FMA) por ciclo de reloj. Lo que corresponde a la cantidad de operaciones necesarias para la multiplicación de dos matrices de 4×4.

RT Cores

Los RT Cores son el hardware específico para aceleración de trazado de rayos en las GPU de NVIDIA, en la arquitectura Turing se encuentran dentro de cada unidad de procesamiento. Son unidades con lógica no programable para calcular eficientemente el recorrido de un rayo en una estructura de aceleración cargada en memoria, más específicamente una jerarquía de cajas como volúmenes acotantes (BVH). Los RT Cores pueden encargarse de ejecutar los tests de visibilidad de geometría requeridos por un hilo de ejecución en la SM.

El trazado de rayos sin aceleración de hardware requiere miles de instrucciones de software por rayo para chequear intersecciones con volúmenes limitantes. Es un proceso computacionalmente intensivo que hace que sea imposible hacerlo en GPU en tiempo real sin la aceleración del trazado de rayos basada en hardware (figura 2.11).

Cada RT Core incluye dos unidades especializadas. La primera realiza chequeos de intersección con los volúmenes limitantes y la segunda de intersección con triángulos. Cada SM simplemente solicita el lanzamiento de un rayo, el RT Core realiza las pruebas de recorrido sobre la BVH y devuelve una colisión si existe. Este proceso evita que el SM ejecute miles de instrucciones (figura 2.12).

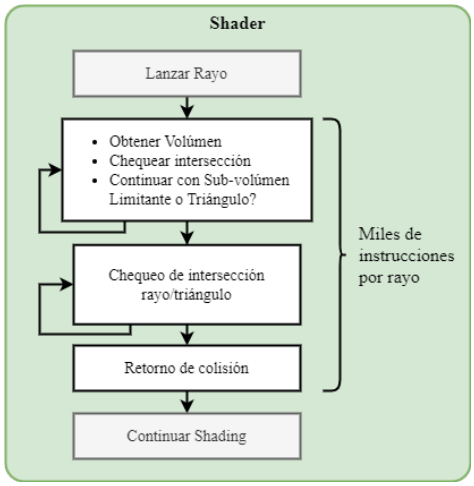


Figura 2.11: Trazado de rayos sin aceleración por hardware. Chequeo de intersecciones emulado por software.

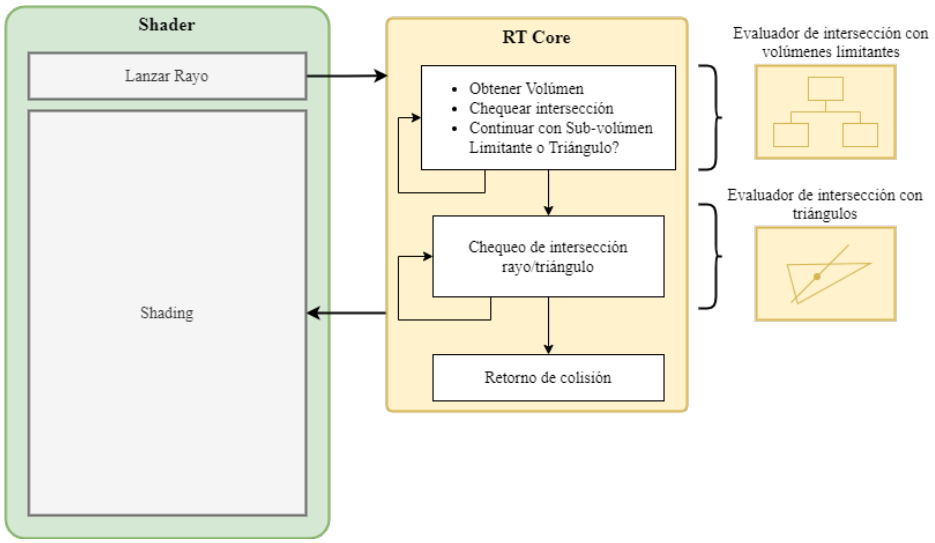


Figura 2.12: Ray tracing con aceleración por hardware (RT Cores) reemplazando la emulación por software representada en la figura 2.11 [12].

Las operaciones de trazado de rayos son ejecutadas utilizando un conjunto de nuevos shaders que se inicializan en un pipeline independiente al clásico pipeline de rasterizado. El nuevo pipeline interactúa directamente con las unidades de chequeo de intersecciones y obtiene para cada rayo una colisión (hit) o una pérdida (miss) reportada por el hardware. A partir de esta nueva lógica en el hardware, se construye un flujo como el de la figura 2.13.

Sobre el pipeline de trazado de rayos se define una lógica que puede variar dependiendo de la tecnología de desarrollo utilizada. En la sección 2.8 se analizan algunas formas de acceder a esta funcionalidad de la GPU. En particular para la interfaz de desarrollo de aplicaciones Vulkan, se presenta un pipeline lógico implementado por el controlador de la tarjeta gráfica. Vulkan expone en distintas

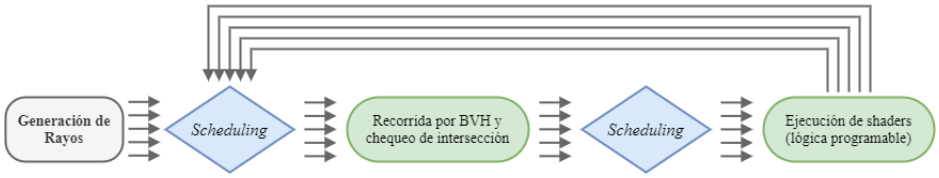


Figura 2.13: Pipeline de trazado de rayos a nivel de hardware en las tarjetas gráficas NVIDIA RTX [12].

etapas y de manera intuitiva para el programador, la generación y el chequeo de intersecciones de rayos en una escena.

Los RT Cores fueron recientemente mejorados y denominados *RT Cores* de segunda generación en la arquitectura Ampere [14]. AMD por su parte añade «Ray Accelerators (RA)», hardware de similares características en la segunda versión de su arquitectura RDNA™ [15].

2.8. Biblioteca Gráfica Vulkan

2.8.1. Introducción

Vulkan es una biblioteca gráfica desarrollada por el grupo Khronos que permite acceder a pipelines de GPU con funcionalidades gráficas y de cómputo. Su objetivo es alta eficiencia y agnosticismo entre plataformas y sistemas operativos incluyendo PCs, consolas, teléfonos y plataformas embebidas. A diferencia de OpenGL la biblioteca predecesora desarrollada también por Khronos, Vulkan está diseñada para funcionar muy cerca del hardware. Vulkan le da al programador control directo sobre los recursos de cómputo, con gran escalabilidad para procesamiento concurrente, los trabajos que comienzan en diferentes hilos se mantienen separados desde el momento que son creados hasta la ejecución.

Hoy en día existen diversas formas de acceder de forma agnóstica a las funcionalidades de cualquier GPU. La interfaces de programación de aplicaciones con alto nivel de abstracción como OpenCL o CUDA facilitan acceso al programador sin "acercarse" demasiado al hardware. Pero también existen interfaces específicas de programación gráfica como OpenGL, DirectX, Metal o la ya mencionada Vulkan con más bajos niveles de abstracción ofreciendo mejor control de funcionalidades específicas, como la aceleración de trazado de rayos.

En cuanto al desarrollo de aplicaciones de renderizado utilizando técnicas de trazado de rayos, cualquiera de estas herramientas puede ser útil, pero no muchas pueden acceder al nuevo hardware. Durante los primeros dos años desde el lanzamiento de Turing (2018), no existieron de manera oficial APIs multi-plataforma y agnósticas al hardware para acceder a funcionalidades de trazado de rayos. Únicamente para DirectX (de Windows) se generó la extensión DXR [16], pero con soporte únicamente en Windows. Con la excepción de DXR, no era posible desarrollar aplicaciones con esta funcionalidad que no fueran específicamente diseñadas para funcionar con productos de NVIDIA. Sin embargo, recientemente AMD también lanzó al mercado tarjetas gráficas con aceleración de trazado de rayos por hardware [15]. Esto provocó la necesidad de tener más herramientas para desarrollar aplicaciones que funcionen en el hardware de cualquiera de los dos fabricantes.

Vulkan fue otra de las primeras bibliotecas en implementar una extensión específica para pipeline de renderizado utilizando los *RT Cores* de NVIDIA. Pero esta API a diferencia de DirectX, funciona en una variedad de sistemas operativos, entre los que se incluye Linux. Vulkan es la única API que cuenta con una extensión (*VK_KHR_ray_tracing*) diseñada para que el controlador de cualquier distribuidor de hardware gráfico, pueda implementar un pipeline de trazado de rayos en cualquier sistema operativo. Por esa razón Vulkan es la herramienta utilizada en este proyecto, no solamente para evaluar la GPU de NVIDIA. Vulkan también ofrece la posibilidad de una comparación con hardware de otros vendedores como AMD o Intel, utilizando la misma implementación presentada en los capítulos 3 y 5.

2.8.2. Modelo de Ejecución

Una aplicación Vulkan debe acceder a las funcionalidades expuestas por los dispositivos físicos disponibles en el sistema. La API se encarga de traducir cada una de estas funcionalidades en colas de trabajo categorizadas en «familias», por ejemplo familia de cómputo, de gráficos o transferencia de datos. El soporte de cada familia por parte del dispositivo físico no define directamente la cantidad de colas de trabajo, siendo que cada una se puede encargar de interpretar comandos ejecutados para una o más familias. Las operaciones que se le asignan a una sola cola de trabajo son típicamente procesadas en orden, pero múltiples colas de trabajo pueden superponerse en paralelo.

Detrás de cada cola de trabajo el programador debe definir en uno o más buffers de comandos todas las operaciones que involucran, por ejemplo, un pase de render. En el ejemplo representado por la figura 2.14 se registran en un único buffer, comandos para enlazar recursos del pipeline de rasterizado. Se especifican shaders, vértices, materiales, la ventana de salida o una textura, y también la ejecución del proceso de dibujado antes de finalizar el pase de render. Dentro del mismo buffer también se podrían incluir comandos que no necesariamente son parte de un pase de render. Aunque también se pueden utilizar otras colas de trabajo para la ejecución exclusiva de shaders de cómputo general, comandos para copia o manipulación general de datos de texturas y buffers en la memoria de la GPU.

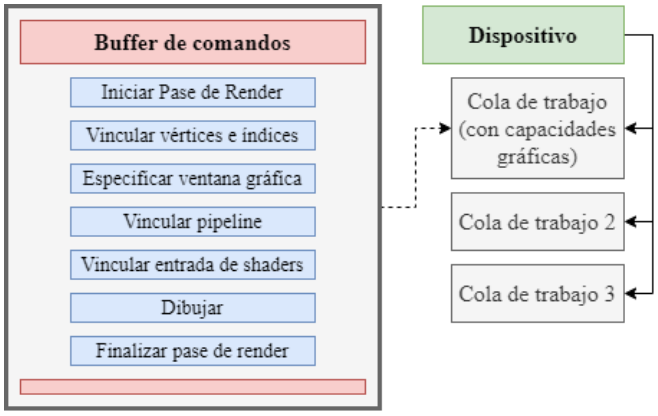


Figura 2.14: Buffer de comandos con un ejemplo de pase de render utilizando una cola de trabajo con capacidades gráficas.

La construcción de buffers de comandos es relativamente costosa, por lo que el procedimiento de Vulkan es capturar la secuencia y guardarla en memoria de

GPU para reutilizarla en cada ejecución. En este caso el programador simplemente hace referencia a los buffers que contienen las listas de comandos y se encarga de la sincronización entre los mismos, sin volver a especificarle al dispositivo el procedimiento. Esto es altamente eficiente en comparación con otras API más antiguas como OpenGL que no cuentan con este mecanismo.

Es importante mencionar que los comandos dentro de un buffer pueden tener dependencias ajenas al GPU, en este caso puede ser necesario destruir y reconstruir un buffer en plena ejecución de la aplicación. Continuando con el ejemplo de la figura 2.14 una reconstrucción del buffer de comandos debe realizarse al momento de cambiar el tamaño de la ventana de salida, muchos comandos dentro del pasaje de render dependen de la resolución.

2.8.3. Trazado de Rayos en Vulkan

El trazado de rayos en Vulkan tiene dos componentes principales, las estructuras de aceleración y un nuevo pipeline con cinco etapas programables. Las estructuras de aceleración contienen toda la geometría de la escena y permiten que el chequeo de intersecciones se realice eficientemente. Las cinco nuevas etapas del pipeline de trazado de rayos manejan la generación de rayos, las intersecciones o los rayos que se pierden sin colisionar.

Estructuras de Aceleración

Existen muchas formas de implementar una estructura de aceleración, la más común y la utilizada en las tarjetas gráficas de NVIDIA es una jerarquía de volúmenes acotantes (BVH) con cajas alineadas respecto a los ejes de coordenadas. Vulkan tiene mucho cuidado en no especificar la forma en la que se implementa esta estructura de datos a nivel de fabricante de hardware, esto pasa a ser responsabilidad del controlador. En cambio se provee un método de implementación opaco que el desarrollador debe definir y alojar en memoria. La implementación de Vulkan se compone de dos elementos, las estructuras de nivel superior y de nivel inferior, en inglés estructuras *top-level* y *bottom-level*.

La figura 2.15 representa la implementación de estructuras de aceleración provista por Vulkan. Las estructuras de aceleración de nivel inferior son las que contienen la geometría de la escena y las estructuras de nivel superior contienen múltiples instancias de las estructuras de nivel inferior. Cada una de las instancias dentro de la estructura de nivel superior tiene una matriz de transformación que define su posición en la escena. Múltiples instancias pueden representar a una misma geometría con distintas matrices de transformación en la misma escena.

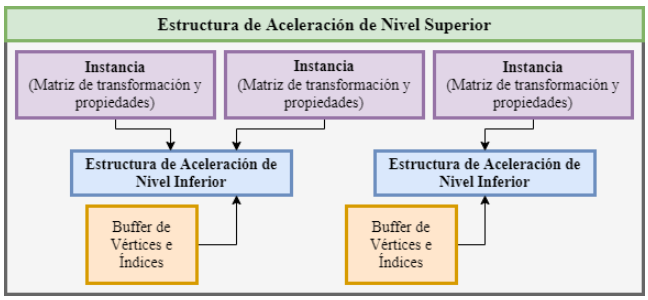


Figura 2.15: Jerarquía de una estructura de aceleración implementada en Vulkan.

Shaders para Trazado de Rayos

En la sección 2.6 se presentó un pipeline típico de rasterizado y se describieron algunas etapas a las que el programador puede acceder para personalizar el resultado. De la misma forma que existe un pipeline de rasterizado, las tarjetas gráficas con hardware para trazado de rayos cuentan con un flujo independiente que involucra el chequeo de intersecciones. A nivel de API, Vulkan especifica dos nuevos flujos para utilizar la extensión de trazado de rayos, el primero cuenta con cinco etapas (figura 2.16). Todas las etapas del primer pipeline de ray tracing son programables, siendo el controlador del hardware quien se encarga de interpretar cada una para utilizar el dispositivo de manera correcta [17].

En la figura 2.16 se identifican las cinco etapas programables en una representación del flujo de ejecución, donde cada una de ellas tiene un shader asociado:

- Ray generation (Generación de rayos)
- Any hit (Cualquier colisión)
- Intersection (Intersección)
- Closest hit (Colisión más cercana)
- Ray miss (Rayo perdido)

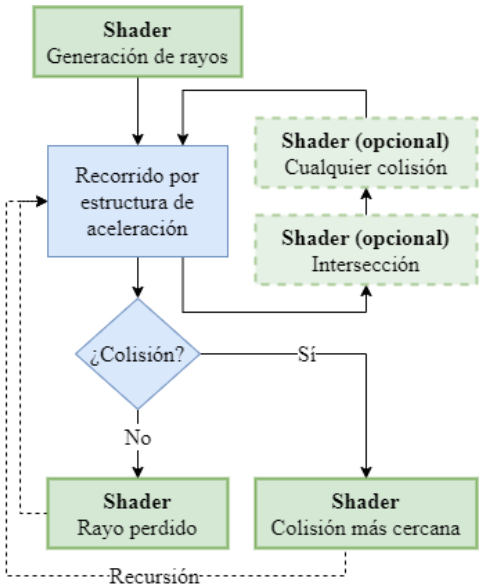


Figura 2.16: Flujo del pipeline de trazado de rayos en Vulkan.

Una posible implementación de este pipeline de ray tracing es descrita en profundidad en el capítulo 3 junto con una aplicación basada en ray tracing implementada utilizando Vulkan.

Como alternativa a la implementación del pipeline de cinco etapas, Vulkan ofrece *Ray Queries*, otra extensión que permite ejecutar un chequeo de intersecciones a través de una estructura de aceleración en cualquier shader. La extensión habilita consultas fuera del pipeline de trazado de rayos.

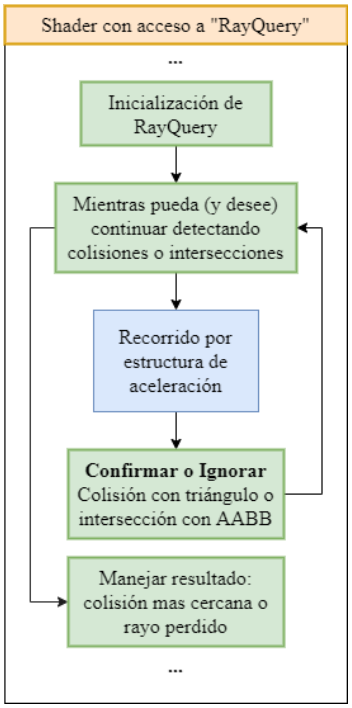


Figura 2.17: Flujo utilizando ray query como parte de la lógica de un shader en Vulkan.

La figura 2.17 muestra un flujo de ejecución utilizando ray query. El programador debe inicializar la extensión obteniendo la estructura de aceleración y definiendo el rayo a lanzar. Luego se itera sobre cada una de las intersecciones y colisiones más cercanas, hasta que alguna cumpla con una condición definida en el programa. La búsqueda se implementa como una estructura de control en el shader y una vez se sale de la misma, se obtiene la información correspondiente a la colisión confirmada o en su defecto un resultado vacío representando un rayo perdido.

La extensión permite continuar utilizando sin demasiadas modificaciones un shader de rasterización, aplicando de forma explícita trazado de rayos como complemento para generar efectos que de otra forma no serían posibles o muy difíciles de lograr. El caso de uso más evidente es el chequeo de intersección dentro de un shader de fragmentos en la etapa de procesamiento de píxeles. Esto sirve por ejemplo, para trazar rayos de sombra directamente hacia luces puntuales o de área.

Para el caso del nuevo pipeline lógico de cinco etapas para trazado de rayos, implementar esta técnica de renderizado híbrido no es tan trivial. De la misma forma que se permite ejecutar el pipeline de rasterizado en una cola de trabajo con capacidades gráficas, el de trazado de rayos tiene su propio punto de entrada como comando independiente. No obstante, esto implica que si el hardware lo permite, el trazado de rayos puede ser ejecutado de forma concurrente al de rasterizado, y es tarea del programador sincronizar y componer la salida ambos en un solo cuadro.

2.9. Aplicaciones y Trabajos Relacionados

En esta sección se presentan los casos de uso más populares en la actualidad para el marco teórico presentado en este capítulo. Por un lado, técnicas de ray tracing aplicadas en el mundo del software y casos de éxito utilizando la arquitectura Turing. Y por otro, publicaciones relacionadas que extienden o complementan el trabajo realizado en este proyecto.

2.9.1. Ray Tracing en Tiempo Real

La modificación de código abierto del juego «Quake II», realizada por Christoph Shied, fue el primer desarrollo de un motor de renderizado completamente implementado con ray tracing en tiempo real [18]. En un trabajo en conjunto con NVIDIA, el juego fue mejorado y utilizado como un ejemplo para comercializar el nuevo producto de NVIDIA.

Hasta ahora Quake II es el caso más conocido de renderizado 100 % potenciado por ray tracing, pero también se pueden encontrar otros trabajos basados en el código abierto de ese proyecto para implementar la misma funcionalidad. Tal es el caso del trabajo de Lipp [19] para una implementación similar en Quake III.

En cuanto a motores de renderizado en tiempo real, los motores de videojuegos más populares Unity y Unreal Engine proveen funcionalidades para acceder a ray tracing.

Ambos motores cuentan con una modalidad de ray tracing híbrido para agregar funcionalidades sobre el motor de rasterizado; se pueden implementar reflexiones, refracciones, iluminación global y oclusión ambiental.

Unreal Engine cuenta con una modalidad de path tracing para generar imágenes de referencia sobre escenas, y otra modalidad de render híbrido, donde las funcionalidades antes mencionadas funcionan junto con un algoritmo de remoción de ruido, que permite que los videojuegos sean enfocados para su uso en tiempo real. Por su parte, Unity parece enfocar el uso de ray tracing para producciones de calidad cinematográfica y no para su uso en tiempo real.

El lanzamiento de videojuegos comerciales con funcionalidades de ray tracing ha crecido en el último año por el lanzamiento de las consolas *Xbox Series X y S*, y *PlayStation 5*. La nueva generación de consolas cuenta con las tarjetas gráficas RDNA2 [15] con hardware dedicado para trazado de rayos.

Como videojuegos referentes en el área de ray tracing, se pueden mencionar *Battlefield V*, *Metro Exodus* y *Shadow of the Tomb Raider*, que fueron los primeros en agregar efectos especiales por encima de los motores de renderizado con rasterización. En el caso de Battlefield V se destacan reflexiones en tiempo real, en Metro Exodus se integró iluminación global y en Shadow of the Tomb Raider sombras de área dinámicas.

Por otro lado, fueron muy importantes para este proyecto algunos detalles de implementación del juego *Wolfenstein: Youngblood* que fueron publicados en forma de «mejores prácticas para renderizado híbrido». Esta publicación del *blog oficial de Vulkan* [20] sirvió como referencia para la implementación del pipeline híbrido del capítulo 5.

La información disponible sobre implementación de ray tracing híbrido en la web es escasa, siendo esta técnica una funcionalidad prácticamente nueva a partir del lanzamiento de Turing. Otro de los pocos artículos que se pueden encontrar como referencia en esta área esta basado en el proyecto PICA PICA de ray tracing en tiempo real, patrocinado por Electronic Arts [21]. El artículo es parte de del libro Ray Tracing GEMS. Allí se puede encontrar la publicación con la demostración del pipeline de renderizado, que logra, con una combinación de shaders de rasterización, cómputo general y ray tracing, efectos visuales de muy alta calidad en tiempo real [22].

2.9.2. Ray Tracing en Motores de Renderizado

La lista de software que utiliza ray tracing con el fin de renderizado físicamente realista es extensa [23], entre las herramientas más populares vale la pena destacar *Blender* [7], *Maya* [8], *3ds Max* [9] y *Cinema4D* [10]. Históricamente, la ejecución de los algoritmos de ray tracing es llevada adelante por este tipo de software vía CPU, y eran requeridos plug-ins como *Octane Renderer* [11] para utilizar la GPU como herramienta para acelerar el renderizado.

Recientemente, y probablemente gracias al lanzamiento de la arquitectura Turing, todos los programas mencionados soportan ray tracing con GPU. En particular, Blender tiene una integración con la herramienta de OptiX para remoción de ruido que permitió validar que la misma integración realizada en el capítulo 3 para la aplicación path tracing funcione correctamente.

2.9.3. Otras Publicaciones

La publicación «Examination of the Nvidia RTX» contiene una comparación de la ejecución de algoritmos de ray tracing en una tarjeta gráfica de modelo RTX2070 con arquitectura Turing, contra una tarjeta GTX1070 de generación previa (Pascal) [24]. Los autores mencionan una gran diferencia en términos de rendimiento, a favor de la GPU de arquitectura Turing, para la ejecución de este tipo de algoritmos.

Las ventajas de la implementación de estructuras de aceleración en el hardware de las RTX es evaluado en la publicación «RTX Beyond Ray Tracing» [25], donde se explora el uso de los RT Cores para localización de puntos en mallas tetraédricas. Los autores experimentan con comparaciones entre la implementación de estructuras de aceleración provista por CUDA, con consultas de intersección por software, contra las estructuras BVH implementadas a nivel de controlador, con consultas de intersección por hardware con y sin la presencia de RT Cores.

Existen también trabajos para explorar otros posibles casos de uso del hardware. En la publicación «GPU Accelerated Ray-tracing for Simulating Sound Propagation in Water» se utiliza OptiX para implementar una aplicación que simula la propagación de sonido en el agua [26]. Y en «Accelerating Force-Directed Graph Drawing with RT Cores» se utilizan los RT Cores para acelerar el cálculo de grafos con nodos afectados por distintos tipos de fuerzas físicas, tales que queden distribuidos de una forma óptima en el espacio [27].

[7] Blender: `https://www.blender.org/`
[8] Maya: `https://www.autodesk.com/products/maya`
[9] 3ds Max: `https://www.autodesk.com/products/3ds-max`
[10] Cinema4D: `https://www.maxon.net/en/cinema-4d/`
[11] Octane Renderer: `https://home.otoy.com/render/octane-render/`

Capítulo 3

Aplicación Path Tracing Monte Carlo

Para el proyecto se implementaron tres aplicaciones que ponen a prueba las nuevas características de la arquitectura Turing. Las tres aplicaciones están potenciadas por la tecnología de las GPU con arquitectura Turing y el acceso al hardware se implementa por medio de la biblioteca gráfica Vulkan. En este capítulo se presenta la primera de estas aplicaciones, que utiliza la técnica Path Tracing Monte Carlo y tiene como objetivo ray tracing físicamente realista potenciado por la GPU para obtener alto rendimiento.

3.1. Arquitectura de la Solución

Se presenta a continuación una representación de alto nivel de la solución implementada. Los componentes de la parte superior de la figura 3.1 son tres pipelines implementados y ejecutados en el dispositivo (GPU). El centro de la arquitectura es una aplicación Vulkan encargada de administrar los recursos y la sincronización con el hardware gráfico. Los números que figuran en las líneas que conectan cada uno de los pipelines de GPU a la aplicación (1, 2 y 3), representan el orden en el que se ejecutan. Por último se cuenta con una representación de la escena con todos los recursos a ser cargados y utilizados para la simulación.

3.1.1. Escena

Toda la información relevante para generar el espacio tridimensional es representada por distintas entidades dentro de una escena. Una escena debe contener al menos una cámara y un buffer de vértices e índices representando algún tipo de geometría. La geometría renderizada debe contener al menos un material asignado y cada material contiene una o más texturas.

A su vez, la representación de una geometría en la escena no tiene sentido si no aparece referenciada por una instancia, de esta forma pueden existir varias instancias para una misma geometría. Al momento de generar la estructura de aceleración también perteneciente a la escena, cada instancia se mapea uno a uno con las estructuras de aceleración de nivel inferior descritas en la sección 2.8.3.

La iluminación de la escena también es representada a través de una entidad. Una luz puede ser de tipo direccional para representar el sol, pero también puede

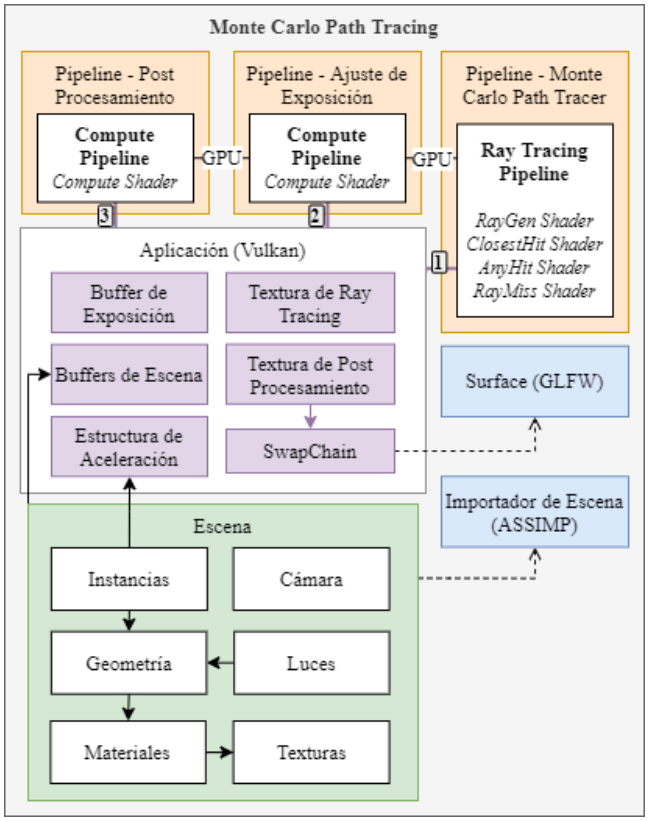


Figura 3.1: Arquitectrura de Sistema Path Tracing Monte Carlo.

instanciarse como geometría visible utilizando un tipo especial de material con color de emisión.

3.1.2. Path Tracer Monte Carlo

La lógica principal del algoritmo de trazado de rayos se encuentra en el bloque correspondiente. Este representa la implementación de un pipeline de trazado de rayos con sus etapas, generación de rayos (Ray Generation), colisión más cercana (Closest Hit), cualquier colisión (Any Hit), y rayo perdido (Ray Miss).

Este pipeline requiere referencias a todos los recursos definidos para la escena. En cada una de las etapas, se obtiene a partir de la estructura de aceleración, el triángulo con el que un rayo interactúa y la instancia a la que pertenece. Luego se puede obtener el material y las texturas correspondientes a la superficie representada.

3.1.3. Aplicación

La ejecución y sincronización de recursos se representa en este bloque. La textura de salida como producto del algoritmo de trazado de rayos, el valor de exposición y la textura de post procesamiento son compartidos entre los tres pipelines descritos anteriormente. Pero el momento de inicio de cada uno, no es determinado por el dispositivo gráfico, en cambio es la aplicación (host) ejecutada en CPU quien

envía la orden de ejecución de cada uno de estos procesos por medio de comandos. Para que sea posible implementar esta lógica del lado del host, existen mecanismos de sincronización provistos por Vulkan que permiten definir semáforos y barreras de ejecución para esperar en el bucle principal de hasta que termine cada pipeline de GPU.

En este caso la sincronización llevada a cabo por la aplicación es trivial, y los pipelines de GPU se ejecutan de forma secuencial. Sin embargo, es importante mencionar que no es un requerimiento, sino una consecuencia por la dependencia de los recursos de cada uno para con el siguiente.

Una vez finalizado todo el procesamiento en GPU, se debe dibujar la textura resultante del pipeline de post procesamiento. Para ello se debe asignar el valor de salida a otra textura con formato correcto, para ser presentada en la ventana del programa. La cadena de intercambio de imágenes (*Swap Chain*) es entidad que maneja este proceso de presentación, donde la imagen presentada en la ventana de salida, no es la misma que se está procesando en segundo plano.

3.1.4. Ajuste de Exposición y Post Procesamiento

La solución también cuenta con dos pipelines de cómputo general para post procesar la imagen resultante del proceso de trazado de rayos. El primero realiza el cálculo automático de exposición. Este proceso es fundamental para equilibrar el brillo de las escenas muy oscuras o muy claras. El segundo pipeline de cómputo general utiliza el valor de exposición calculado en el anterior, y aplica la operación de mapeo tonal sobre cada píxel generando la imagen final.

3.2. Tecnologías Utilizadas

A continuación se listan las tecnologías utilizadas para la implementación de las aplicaciones presentadas para este proyecto. Se prioriza la utilización de tecnologías de código abierto y multi-plataforma para trabajar tanto en Linux como en Windows.

- **Lenguaje de desarrollo** – C y C++ son los lenguajes principales utilizados para el desarrollo de la aplicación. Los programas de GPU son implementados utilizando el lenguaje GLSL que luego es transpilado con la herramienta ShaderC[1], previo a ser cargados en la memoria del dispositivo.
- **Estilo de código y convenciones de nombrado** – Para generar y mantener de forma automática el formato de código, y establecer convenciones de nombrado de elementos del programa, se utilizan dos archivos de configuración de *Clang Format* y *Clang Tidy*. Estas herramientas parte de Clang[2] son aceptadas por una gran cantidad de IDEs para desarrollo con C y C++.
- **Control de versionado** – Para control de versionado de código se utilizó un repositorio de GitHub[3]. El repositorio del proyecto está alojado y disponible con libre acceso en: `https://github.com/manueme/ray-tracing-research`.

[1] ShaderC: `https://github.com/google/shaderc`
[2] Clang: `https://clang.llvm.org/`
[3] GitHub: `https://github.com/`

- **LunarG Vulkan SDK**[4] – Vulkan SDK desarrollado por la empresa *LunarG*, es una colección de herramientas esenciales para facilitar el desarrollo y depuración de aplicaciones Vulkan.
- **GLFW**[5] – GLFW es una biblioteca de código abierto y multi-plataforma, para desarrollo de OpenGL o Vulkan en aplicaciones de escritorio. Provee una API simple para crear ventanas que reciben y emiten eventos manejados por la aplicación.
- **GLM**[6] – OpenGL Mathematics (GLM) es una biblioteca de utilidades de matemática para software gráfico basado en las especificaciones de GLSL.
- **Assimp**[7] – Assimp es una biblioteca de código abierto para importar modelos 3D con los formatos más utilizados en la industria. Assimp es utilizada para la carga de la geometría y materiales renderizados.
- **FreeImage**[8] – FreeImage es una biblioteca de código abierto que soporta la carga y manipulación de varios formatos de imagen como PNG, BMP, JPEG, TIFF. FreeImage se utiliza para cargar las texturas de los materiales pertenecientes a los modelos tridimensionales de la aplicación.

3.3. Implementación del Path Tracer

Dada la integral de la ecuación de renderizado 2.1, el objetivo es estimar el valor de la radiancia resultante de la intersección de un rayo en un punto p_1. Como cualquier algoritmo de trazado de rayos, la forma directa para su implementación es recursiva. Pero el stack de recursión que puede manejar cada unidad de procesamiento es limitado por sus niveles de memoria más cercanos y con poca capacidad. Una solución iterativa se ajusta mejor a trazado de rayos de caminos de cualquier profundidad implementado en GPU.

Una forma intuitiva de generar una solución iterativa es observando la ecuación 2.10, la solución generada calcula $L_(p_1 \rightarrow p_0)$ lanzando un rayo primario y sus consecuentes de forma iterativa en la etapa *Ray Generation* del pipeline de ray tracing de Vulkan. El término $P(\bar{p_i})$ de la ecuación se debe calcular en cada iteración de un bucle para cada rayo primario y el resultado de la suma es el resultado del color para el píxel en cuestión.

En la práctica, este procedimiento presenta dos problemas a resolver:

1. ¿Como se estima el valor de la suma de un número infinito de términos $P(\bar{p_i})$ en un tiempo de cómputo finito?.
2. Dado un $P(\bar{p_i})$ particular, ¿con qué criterio de generan uno o más caminos para computar el estimado Monte Carlo de la integral multidimensional?.

3.3.1. Ruleta Rusa

El primero de los problemas se puede resolver con la técnica *ruleta rusa*, que consiste en detener el muestreo de un camino cuando su contribución en el cálculo

[4]Vulkan SDK: `https://www.lunarg.com/vulkan-sdk/`
[5]GLFW: `https://github.com/glfw/glfw`
[6]GLM: `https://github.com/g-truc/glm`
[7]Assimp: `https://www.assimp.org/`
[8]FreeImage: `https://freeimage.sourceforge.io/`

es nula o muy baja. Esto es posible porque como consecuencia de la conservación de energía durante un trazado de caminos, aquellos con mayor profundidad tienden a tener menor influencia que caminos con menor profundidad. A partir de esta afirmación se puede asumir que detener un camino con cierta profundidad no aumenta de forma determinante el error del resultado.

La técnica de *ruleta rusa* se basa en esta asunción para detener con una probabilidad q_i determinada, el trazado del camino. A medida que la profundidad crece, la probabilidad de terminación también. A su vez para esta implementación, el valor de la probabilidad de terminación en cada interacción, es directamente proporcional a la radiancia aportada por el punto. Esto quiere decir que las superficies que contribuyen menos en el resultado final tienen menor probabilidad de generar nuevos rayos innecesariamente y las más iluminadas tendrán como resultado cálculos con mayor profundidad.

3.3.2. Distribución de Rayos

Para resolver el segundo punto hay que definir una función de distribución adecuada para la dirección de cada rayo lanzado y la cantidad de rayos a lanzar por interacción. Esto incluye en primer lugar, a los rayos primarios que nacen desde la posición del observador en dirección a un punto dentro de cada píxel de la cámara representada.

Generación de Semilla Semi-Aleatoria

Toda función de muestreo utilizada en esta solución se vale de una semilla con un valor semi-aleatorio independiente para cada muestra. Dentro de un programa ejecutado en GPU no existen herramientas para la generación de este tipo de valores aleatorios tan fácilmente como para un programa en CPU. La solución implementada para generarlos se basa en dos funciones auxiliares.

La primera función aplica *Tiny Encryption Algorithm* [28] y se utiliza para generar un número inicial basado en un hash de la coordenada del píxel, en dependencia el número de muestra actual. De esta forma para cada píxel renderizado y para cada cuadro se obtiene un número distinto.

La segunda función aplica un *Linear Congruential Generator* [29], en este caso a partir de un número previo que inicialmente es el hash generado por la función anterior, retorna otro aleatorio entre $[0, 2^{24})$. Luego, a partir de las funciones ***TEA*** y ***LCG*** se genera una tercer función ***RND*** que retorna un número semi-aleatorio entre 0 y 1 vital para el muestreo de rayos en el programa. El valor retornado por RND cumple la función de variable aleatoria uniforme canónica, definida en la sección 2.4.2.

Píxeles

Cada píxel que constituye una imagen, es un conjunto de muestras de puntos de una función discreta que mapea cada punto del plano de la imagen. Los píxeles son naturalmente definidos como un número entero positivo en una grilla. Por lo tanto se debe generar una función que a partir de una serie de muestras sobre un espacio de posiciones, representadas por números de punto flotante, obtenga el resultado para cada valor discreto correspondiente en la imagen.

Dado un píxel de la cámara representada para una escena, el primer paso para generar el rayo primario es obtener un punto dentro del espacio de posiciones representadas por punto flotante. Para el píxel (x_0, x_1), generando dos números r_1 y r_2 con la función *RND*, la muestra es $(x_0 + r_1 - 0{,}5, x_1 + r_2 - 0{,}5)$.

Muestreo de Fuentes de Luz

Una vez producida una intersección de un rayo con una superficie, el primer caso a tener en cuenta para generar un nuevo rayo es el de iluminación directa. El objetivo es determinar si a ese punto arriban rayos de forma directa, si este chequeo falla, se determina que este punto está "en sombra". Por este motivo los rayos lanzados con este fin son conocidos como *rayos de sombra*.

Para luces direccionales como el sol, el muestreo es trivial siendo que la dirección del rayo es la dirección opuesta a la luz.

El otro tipo de luz implementado es representado con geometría en la escena, ésta parte de un área definida por los triángulos y no de un punto o una orientación. El subconjunto de triángulos correspondiente a la luz, está definido como información a la que el programa puede acceder y el material correspondiente a cada elemento tiene información de intensidad y color.

Para el caso de luces de área, el muestreo se puede deducir a partir de la ecuación 2.8 de transporte de luz para un conjunto A de superficies de la escena. Para cada luz de área en la escena, el procedimiento para la generación de un nuevo rayo de sombra, es obtener de forma aleatoria un triángulo cualquiera y un punto dentro del área definida para ese triángulo.

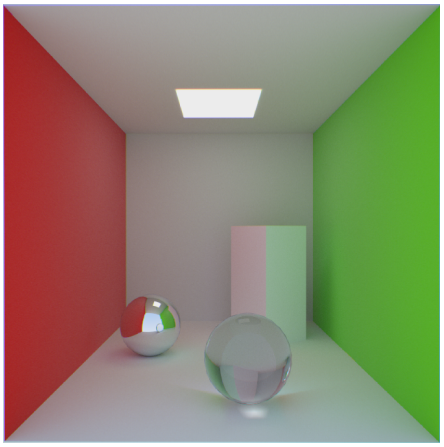


Figura 3.2: Ejemplo de luces de área con geometría, el plano blanco ubicado en el techo está compuesto por dos triángulos con un material con color de emisión configurado

Muestreo del Hemisferio

La iluminación indirecta es el último caso a tener en cuenta para generar nuevos rayos a partir de cada interacción con una superficie. La iluminación incidente en un punto o irradiancia, corresponde al cálculo para todas las direcciones del hemisferio. Sin embargo para cada camino trazado por el algoritmo, se traza un único rayo con una dirección cualquiera dentro de este hemisferio, esto se realiza

para calcular la irradiancia estocásticamente lanzando suficientes rayos con una distribución adecuada.

Generar direcciones aleatorias con una distribución uniforme es la forma más sencilla de muestrear el hemisferio de un punto para un camino generado por el algoritmo. Sin embargo, si se observa la ecuación transporte de luz 2.1, la función f correspondiente al término BRDF está ponderada por el término del coseno del ángulo de incidencia del rayo respecto a la normal. Esto quiere decir que la solución se puede optimizar generando una distribución también basada en el ángulo incidente, de modo que la mayoría de muestras sean aquellas con mayor contribución a la solución final.

El muestreo aplicado para la generación de rayos de iluminación indirecta es implementado con el Método de Malley [1, Cap. 13] para la generación de una distribución de puntos en el hemisferio ponderados por el coseno.

3.3.3. Pipeline de Ray Tracing

El algoritmo de trazado de rayos es implementado en su totalidad utilizando *shaders*, programas alojados en GPU. El pipeline cuenta con cinco etapas implementables, de las cuales dos son opcionales.

Para esta aplicación se implementaron las etapas *Ray Generation*, *Any Hit*, *Closest Hit* y *Ray Miss*.

Ray Generation

El bucle principal de cada rayo primario se inicia en el shader de generación de rayos, este bucle puede terminar si se da alguna de las siguientes tres condiciones:

- Terminación por *Ruleta Rusa*.
- Se alcanza el máximo nivel de profundidad configurado.
- No se detecta ninguna colisión, ray miss.

Para poder transportar información entre los diferentes shaders implementados para cada etapa, cada rayo lanzado contiene un objeto (*payload*) con una definición específica. La definición del *payload* de un rayo lanzado es la siguiente:

```
struct RayPayload {
    vec3 surfaceEmissive;
    vec3 surfaceRadiance;
    vec3 surfaceAttenuation;
    vec3 nextRayOrigin;
    vec3 nextRayDirection;
    uint seed;
    int rayType;
};
```

El *payload* de un rayo funciona tanto de entrada como de salida para una consulta. Los valores se cargan inicialmente en esta etapa y son leídos y modificados en las siguientes. Un atributo utilizado para lectura y escritura en todas las etapas, es la semilla generada inicialmente por *TEA*.

Los atributos más importantes de este objeto son los canales de emisión, radiancia y atenuación resultante de la colisión más cercana. El valor de atenuación corresponde al albedo del material, y es utilizado por el chequeo de *Ruleta Rusa*. Un rayo con atenuación muy baja incrementa la probabilidad de detener la generación de rayos para ese camino.

En el *payload* también se retorna la dirección y el origen del próximo rayo a lanzar, la colisión más cercana determina el valor de estos atributos junto con el tipo de rayo. El tipo de rayo puede ser de refracción, reflexión, difuso o rayo perdido.

Closest Hit

Esta etapa se ejecuta cuando se obtiene la colisión más cercana al origen de un rayo. Si el material de la superficie colisionada está definido como refractivo y/o reflectivo, el rayo puede ser reflejado o transmitido.

Para materiales con un índice de refracción definido, la probabilidad de que se genere una reflexión o una refracción en el camino calculado está determinada por las ecuaciones de Fresnel [1, Cap. 8]. Dado un índice de refracción y el ángulo de incidencia del rayo con la normal de la superficie, las ecuaciones de Fresnel especifican la reflectancia correspondiente para estos dos tipos de luz incidente en ese punto.

Las ecuaciones de Fresnel caracterizan los materiales en tres grupos. Materiales dieléctricos, conductores y semiconductores. Para simplificar el algoritmo y la definición de los materiales cargados para el render, se asume que todo material con índice de refracción es dieléctrico. Esto incluye los materiales más comunes como agua y vidrio utilizados para evaluar los resultados del algoritmo.

Si el rayo no fue reflejado o transmitido por el material, en esta etapa se genera una nueva dirección y un origen el cálculo de iluminación indirecta. Para esto se utiliza la función *RND* y la semilla incluida en el *payload*, tal como fue descrito en la sección 3.3.2. El nuevo rayo será lanzado nuevamente en Ray Generation una vez que esta etapa finalice.

En este caso también se deben lanzar rayos de sombra para cada una de las luces y calcular la radiancia. Esto corresponde al término BRDF de la ecuación de transporte de luz. También se obtiene la emisión del material, si está presente, y se asigna el albedo como valor de atenuación en el *payload* del rayo resultante.

Con el fin de generar una solución eficiente y sencilla se utiliza modelo Phong de reflexión especular presentado en la sección 2.2. El modelo Phong también facilita la carga de materiales siendo soportados por la biblioteca *Assimp*. También facilita definir y exportar modelos prácticamente de forma universal por cualquier editor de terceros como *Blender*[9].

Any Hit

La etapa *Any Hit* es la única de las opcionales implementadas. En esta etapa es posible descartar rayos antes de que sean considerados por *Closest Hit*, esto es de gran utilidad para ignorar materiales con texturas semitransparentes. El caso más

[9]Blender: `https://www.blender.org/`

común es la posibilidad de renderizar follaje a partir de geometría sencilla como un plano para cada hoja.

En esta etapa se ignoran aquellos rayos que detecten un material con una textura cuyo canal de transparencia es cero. Esto aplica tanto para rayos difusos como rayos de sombra. En este caso no hay cálculo de refracción porque el rayo atraviesa una superficie totalmente transparente.

Ray Miss

En Ray Miss se tienen en cuenta aquellos rayos que no detectan ninguna colisión. Aquí simplemente se calculan algunos valores por defecto para cargar en el *payload* del rayo. Estos valores corresponden al color del cielo que rodea la escena renderizada.

3.3.4. Exposición y Post Procesamiento

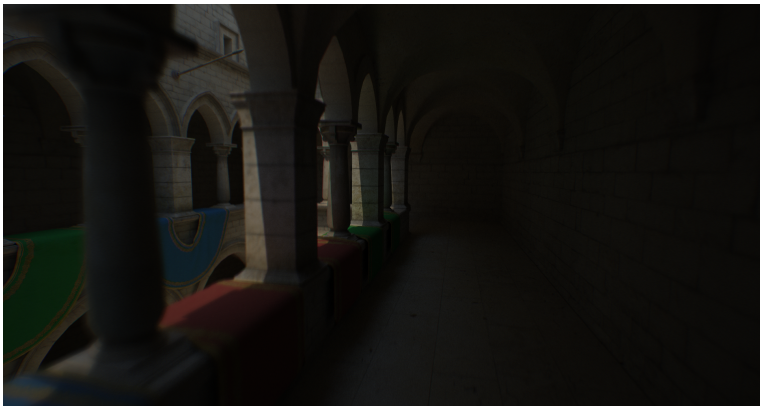
El color representado por la imagen generada en el algoritmo de trazado de rayos en algunos casos puede resultar oscura para escenas con poca iluminación. *Tone mapping* es el proceso de convertir los valores de radiancia de la escena a valores adecuados para una pantalla [9, Cap. 8]. El concepto de exposición es crítico para el *tone mapping*. En fotografía, la exposición refiere a controlar la cantidad de luz capturada por el sensor de una cámara. Sin embargo, en rendering, la exposición termina siendo una operación de escalado lineal aplicada sobre una imagen a ser aplicada durante el mapeo tonal.

La función del pipeline de exposición es leer el resultado del trazado de rayos y estimar a partir de la luminancia promedio de la imagen un valor de exposición. En esta implementación se pretende un valor de la exposición E igual a 1 cuando luminancia promedio L es 0,5, de esta forma se define $E = 0{,}5/L$, y se limita L entre 0,1 y 1,0 para evitar valores extremos con imágenes demasiado oscuras.

Durante el pipeline de post procesamiento y para cada píxel de la imagen se toma el valor de E actual y se multiplica el color C, luego al valor resultante se le aplica una corrección de gamma G estándar [10]. El resultado corresponde al valor de C' en la ecuación 3.1.

$$C' = (C * E)^{1/G} \tag{3.1}$$

[10] El valor de gamma utilizado es 2,2 y corresponde al valor promedio de la mayoría de los monitores del mercado [30]



(a) Resultado sin ajuste de exposición

(b) Resultado con ajuste de exposición

Figura 3.3: Comparación entre una imagen con ajuste de exposición y otra sin ajuste de exposición

El ajuste de exposición se realiza de forma automática pudiendo pasar de lugares oscuros a lugares de mucha iluminación en una misma escena sin necesidad de cambiar alguna configuración de renderizado.

3.4. Resultados

Se utilizaron dos escenas con diferentes características para probar las funcionalidades implementadas. El programa fue configurado para lanzar por defecto un rayo por píxel en cada cuadro renderizado, pero también se pueden asignar más muestras por píxel. El resultado de un cuadro se acumula con el siguiente si no hay ningún movimiento de cámara.

En base a pruebas realizadas sobre las escenas utilizadas, el largo máximo de un camino puede rondar entre diez y quince sin provocar cambios relevantes en el resultado final. Menos de diez rayos de profundidad tiende a oscurecer espacios con poca iluminación, y más de quince tiende a ser irrelevante producto de la terminación por ruleta rusa.

Todas las imágenes presentadas a continuación son generadas con caminos

de profundidad máxima diez, obteniendo así el mejor rendimiento sin afectar la iluminación en los resultados.

La primera de las escenas es la *Sponza*, éste modelo es comúnmente utilizado para realizar demostraciones de programas de renderizado. Ésta escena permite ilustrar principalmente cómo se comporta la iluminación global y el efecto de *color bleeding* cuando el color de una superficie iluminada influencia aquellas que la rodean. Las figuras 3.4, 3.5 y 3.6 son ejemplos de los resultados obtenidos.

Figura 3.4: Vista general el modelo de la sponza en una posición a la que no llega mucha iluminación.

Figura 3.5: Rincones oscuros del modelo de la sponza iluminados de forma indirecta.

Figura 3.6: La interacción de la luz con las banderas de distintos colores arroja color sobre las columnas de la sponza.

El segundo de los modelos tiene como propósito evaluar reflexiones y refrac-

ciones especulares. La piscina con agua comprueba como la selección de caminos para refracción y reflexión funcionan según lo deseado. Al cambiar la posición de la cámara el efecto de Fresnel provoca más rayos reflejados o transmitidos con dirección hacia el fondo de la piscina.

La escena también fue configurada con luces de área de varios polígonos. El canal de emisión rápidamente resalta el color blanco de la geometría, pero también se puede observar como son iluminadas las superficies que rodean la luz. Las figuras 3.7, 3.8 y 3.9 son ejemplos de los resultados obtenidos.

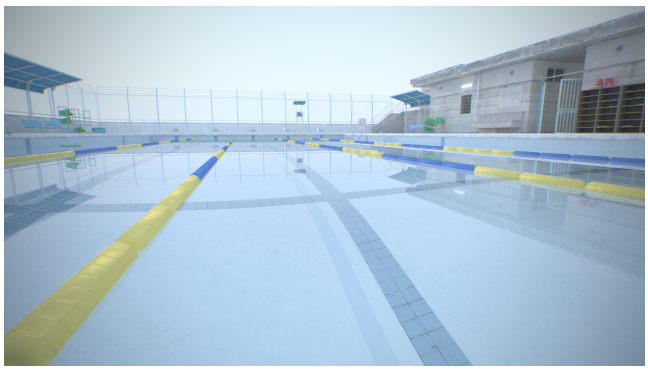


Figura 3.7: El índice de refracción del agua altera la dirección de la luz obedeciendo la Ley de Snell, la ecuación de Fresnel determina la cantidad de rayos reflejados y transmitidos.

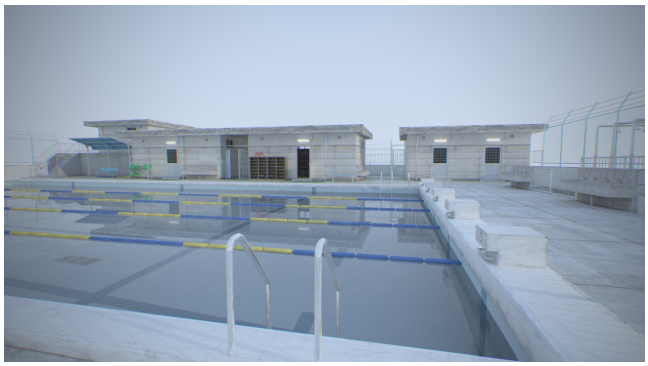


Figura 3.8: Las luces de área representadas con geometría también son reflejadas en el agua.

Figura 3.9: Escena interior con una luz de área en el techo.

3.5. Rendimiento

La GPU utilizada para este estudio es la *NVIDIA GeForce RTX 2060 SUPER*, correspondiente a la arquitectura Turing *TU106* cuyas características son detalladas en el anexo A.

Para obtener un mejor análisis del rendimiento de las aplicaciones se utilizó Nsight Graphics de NVIDIA [11]. Con esta herramienta se pueden obtener métricas de la ejecución de cada cuadro renderizado.

Figura 3.10: Escena utilizada para los estudios de rendimiento de las tres aplicaciones del proyecto.

Como caso de estudio para evaluar el rendimiento de las aplicaciones, se toma la traza de ejecución del GPU para renderizar la imagen 3.10 de la escena de la piscina. El algoritmo de trazado de rayos está configurado para caminos de profundidad máxima diez. Ambas aplicaciones ejecutan la misma escena utilizando la misma posición y configuración de cámara, y la resolución de salida es 1280x720 (resolución HD).

La posición de la cámara y escena elegida exige al algoritmo trazar rayos de reflexión, refracción y evaluar iluminación directa e indirecta para la luz del sol (direccional) y múltiples luces de área representadas con geometría.

3.5.1. Traza de GPU para Aplicación Path Tracer

La captura presentada es una muestra representativa de muchas muestras realizadas. El programa permite tomar de a grupos de 15 cuadros (figura 3.11), y en promedio los tiempos que toma cada operación se mantienen estables, con diferencias de no más de dos milisegundos en el total de duración de un cuadro.

La traza de un cuadro renderizado en la aplicación de path tracer (figura 3.12) contiene en detalle el tiempo invertido por cada buffer de comandos.

El primer buffer ejecutado es el que contiene la operación de trazado de rayos. En la captura tomada como ejemplo, esta operación cuesta 19.34ms.

[11] NVIDIA Nsight Graphics: `https://developer.nvidia.com/nsight-graphics`

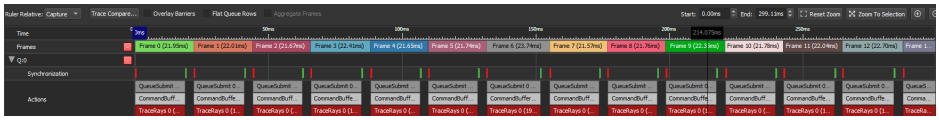


Figura 3.11: Traza GPU de 15 cuadros de la escena de la piscina.

El segundo buffer de comandos ejecutado contiene las operaciones correspondientes al post procesamiento. En la captura tomada como ejemplo, esta operación cuesta 2.60ms.

En total, este cuadro llevó 22.90ms, lo que da aproximadamente 44 cuadros por segundo a resolución HD. Lo importante a tomar en cuenta a la hora de analizar estos 44 cuadros por segundo, es que corresponden a una única muestra del path tracer. Un render en tiempo real no es viable si el programa no acumula el cálculo realizado cuadro tras cuadro, debido a que el resultado tiene ruido.

Sin embargo, la aplicación que genera la traza, también enlentece levemente el rendimiento de la aplicación. Las métricas de Nsight Graphics son buenas para analizar en detalle cada una de las etapas del render en cada cuadro, pero para un análisis de rendimiento real hay que basarse en la cantidad de cuadros por segundo (FPS) sin una aplicación de depuración enlazada.

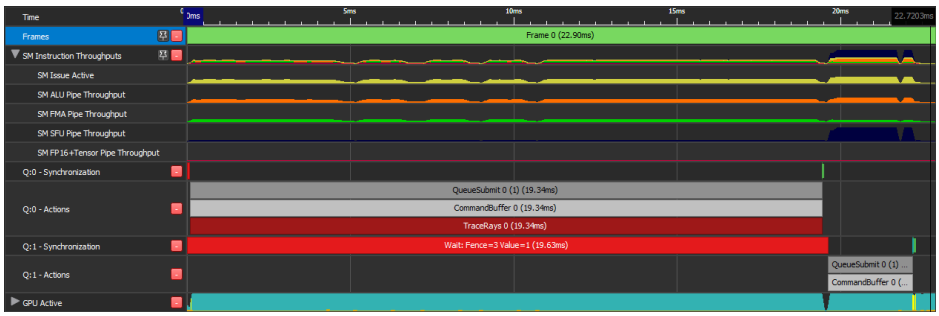


Figura 3.12: Traza de GPU de un cuadro de la escena de la piscina.

3.5.2. Rendimiento en Tiempo Real

A continuación se presenta un análisis de rendimiento basado en el FPS promedio para resoluciones HD (1280x720) y Full HD (1920x1080). Con cada resolución se obtiene la métrica en una posición de cámara exterior (imagen 3.10) y una interior (imagen 4.3), con largo máximo de los caminos de path tracer de profundidades máximas de 4, 6, 8 y 10. La tabla 3.1 contiene el resultado de las ejecuciones.

La profundidad mínima de 4 representa el caso trivial de un rayo que pasa por el agua de la piscina, rebota en el fondo y vuelve a salir hacia la cámara. Los resultados son visiblemente buenos con profundidad 4, pero para una escena interior con iluminación óptima podría interesar utilizar profundidades de largo mayor. Encontrar el largo óptimo de caminos es parte de un análisis cualitativo que no se realiza en este proyecto, sin embargo como parte del análisis de rendimiento, es interesante ver como se comporta el FPS en base a este parámetro.

El largo máximo de los caminos de path tracer es determinante para el tiempo de renderizado de cada cuadro. Para escenas interiores, en la mayoría de los casos

Resolución	Profundidad	Escena	FPS
HD	4	Exterior	86
HD	6	Exterior	71
HD	8	Exterior	64
HD	10	Exterior	59
HD	4	Interior	41
HD	6	Interior	29
HD	8	Interior	22
HD	10	Interior	18
Full HD	4	Exterior	41
Full HD	6	Exterior	33
Full HD	8	Exterior	29
Full HD	10	Exterior	27
Full HD	4	Interior	19
Full HD	6	Interior	13
Full HD	8	Interior	10
Full HD	10	Interior	8

Tabla 3.1: Rendimiento de la aplicación Path Tracing Monte Carlo. Una muestra por píxel, basada en profundidad máxima de caminos, tipo de escena y resolución de ventana.

los caminos serán de largo máximo debido a que no se pierden rayos en el infinito.

Reducir la profundidad máxima de un rayo y aumentar la cantidad de muestras por píxel es una opción para obtener mejor calidad en tiempo real. Con profundidad máxima 4, por momentos es posible obtener un rendimiento de hasta 90 cuadros por segundo en resolución HD para escenas exteriores. Con este rendimiento es posible lanzar hasta 3 muestras por píxel manteniendo apenas 30 cuadros por segundo. Sin embargo, aún con tres muestras por píxel, el resultado tiene ruido (figura 3.13).

Figura 3.13: Render de la piscina con 3 muestras por píxel y profundidad máxima 4, sin acumulación de cuadros anteriores.

Para una escena estática y sin movimientos de cámara, la ejecución del algo-

ritmo utilizando acumulación es extremadamente veloz. En cuestión de unos pocos segundos es posible trazar una gran cantidad de muestras por píxel generando imágenes de muy buena calidad. Pero cuando no se espera por una cantidad suficiente de muestras, los resultados no son buenos, en su lugar se genera una imagen con ruido (imagen 3.14).

Figura 3.14: Escena interior de la piscina con una baja cantidad de muestras (50 por píxel y profundidad máxima 10). Se puede observar como todavía hay ruido de color en la imagen.

Todo indica que la aplicación implementada sin herramientas para remover el ruido, no es capaz de lograr un render en tiempo real. Pero resta saber qué ocurre si se aplican éstas herramientas de denoising a cada cuadro renderizado, y qué capacidad tienen los Tensor Cores para potenciar este tipo de algoritmos. La aplicación presentada en el siguiente capítulo extiende el trabajo presentado a este punto del informe con una herramienta de denoising potenciada por inteligencia artificial.

Capítulo 4

Aplicación Path Tracing Monte Carlo con Denoiser

La segunda aplicación implementada y presentada en este capítulo, tiene como objetivo post procesar los resultados de la aplicación que utiliza Path Tracing Monte Carlo para ray tracing físicamente realista, con un algoritmo de denoising potenciado por inteligencia artificial.

La motivación de un denoiser es la mejora en el rendimiento, pero no necesariamente con el objetivo de renderizado en tiempo real. En términos generales, lo que se busca es la reducción de la mínima cantidad de muestras necesarias para eliminar completamente el ruido y generar un render de calidad aceptable para el usuario.

Para esta implementación, se utiliza la biblioteca *OptiX*[1] de NVIDIA que contiene un conjunto de herramientas para acelerar el desarrollo de algoritmos de trazado de rayos, entre ellas una para reducción de ruido. La herramienta *denoiser* de OptiX utiliza inteligencia artificial acelerada por los Tensor Cores de la GPU con el objetivo de reducir los tiempos de renderizado.

4.1. Arquitectura de la Solución

Para poder integrar el denoiser de OptiX a la aplicación de renderizado se necesitaron algunas modificaciones en la arquitectura, respecto a la aplicación Path Tracing Monte Carlo sin OptiX.

Lo primero a notar en la figura 4.1 es que aparece el denoiser como un nuevo pipeline entre el path tracer y el ajuste de exposición. El nuevo pipeline está diseñado para ser ejecutado de forma asincrónica desde la aplicación Vulkan, pero no forma parte de ninguno de los buffers de comandos definidos para el resto de los programas utilizando el pipeline de cómputo general del GPU. La biblioteca OptiX provee mecanismos para ejecutar el programa y herramientas para sincronizar por medio de semáforos de Vulkan el inicio y finalización de las tareas en GPU con el resto de los comandos.

[1] `https://developer.nvidia.com/optix`

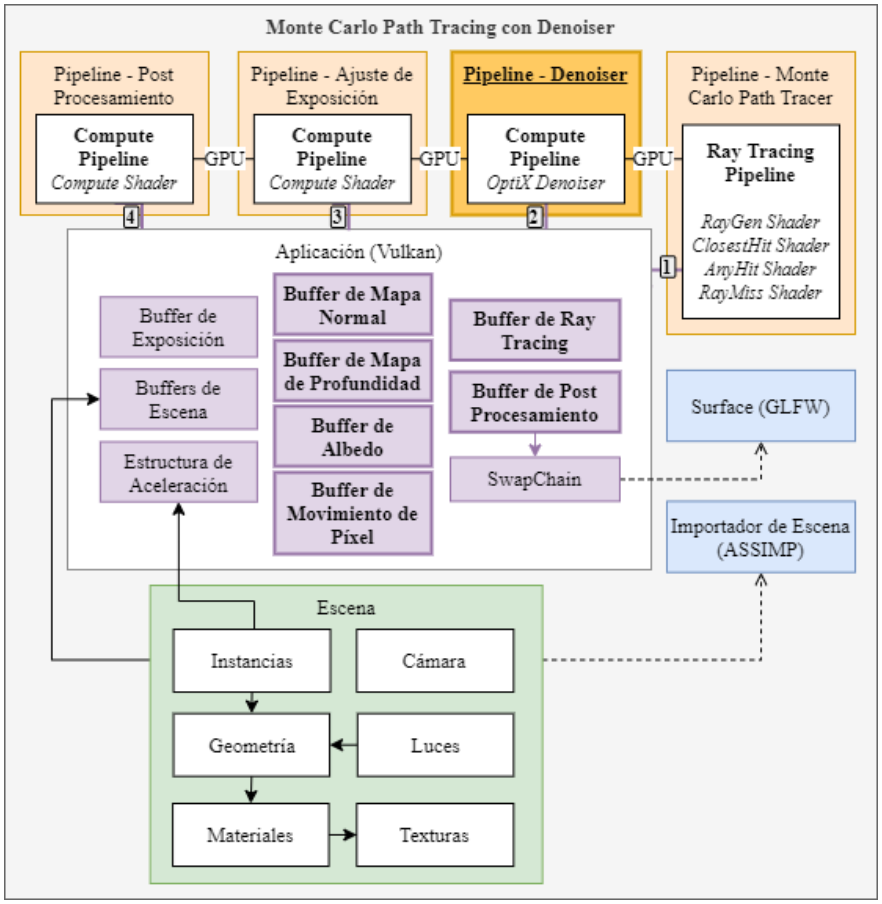


Figura 4.1: Arquitectura de Sistema Path Tracing Monte Carlo con OptiX Denoiser.

Al igual que en la aplicación anterior, los pipelines se terminan ejecutando de forma secuencial, aunque debido a la forma en la que trabaja Vulkan, esto no es un requerimiento del hardware, es responsabilidad de la aplicación la sincronización de la orden de ejecución de cada uno en GPU. A modo de ejemplo, el pipeline de cómputo general que ajusta el valor de exposición podría ser optimizado, ya que no tiene por qué ser ejecutado en cada cuadro renderizado y perfectamente podría procesarse de forma asíncrona.

Otra modificación importante en esta nueva aplicación, es la forma de representar las texturas en GPU. OptiX no entiende el concepto de textura soportado por Vulkan y en su lugar se deben utilizar buffers de datos. Tanto a nivel de aplicación como a nivel de shaders se realizaron modificaciones para tratar las texturas de entrada y salida como buffers.

El denoiser por defecto toma como entrada únicamente la información del cuadro actual a procesar, pero también puede ser configurado con un modo de ejecución «temporal» que se retroalimenta con la salida del cuadro anterior. Para utilizar el modo temporal, el denoiser requiere que se añada como entrada un mapa con el flujo de movimiento de cada píxel respecto al cuadro anterior, esto facilita el trabajo del algoritmo ante movimientos de cámara.

OptiX recibe como entrada tres texturas representadas como array de bytes en formato RGBA con 32 bits por canal y condicionalmente el cuarto buffer con información de movimiento en espacio de pantalla. La primera textura corresponde al albedo de las superficies visibles en pantalla. La segunda corresponde a la normal de la superficie, donde los primeros tres canales de la textura son x, y, y z del vector normal (normalizado) respectivamente. Y la tercera al resultado con ruido del algoritmo de path tracing. Para calcular el cuarto buffer con el flujo de movimiento por píxel, se aloja el estado anterior de la cámara con el fin de evaluar si existe una diferencia con el actual y se calcula en qué lugar sería renderizado cada punto en el cuadro anterior.

La salida de OptiX es un nuevo buffer representando la textura luego de aplicado el algoritmo de remoción de ruido. Esta textura se utiliza luego en los pipelines de exposición y post procesamiento.

4.2. Resultados

Para evaluar la calidad de los resultados del denoiser se comparan con un render de 50000 muestras por píxel, en una escena interior de la piscina. Esto permite comprobar la efectividad del denoiser para generar imágenes equivalentes a un render con una cantidad de muestras por píxel superior. La tabla 4.1 contiene la información del cálculo del error cuadrático medio con distintas cantidades de muestras por píxel, desde una imagen completamente negra (0 muestras), hasta llegar al máximo evaluado (50k muestras). Los valores corresponden al denoiser con y sin el modo temporal activado. La raíz del error cuadrático medio se calcula tomando cada imagen como un vector y calculando el error componente a componente.

MPP	**RMSE**	**RMSE Denoiser**	**RMSE Denoiser Temporal**
0	0.3869	N/A	N/A
1	0.3313	0.1658	0.1660
10	0.2096	0.0852	0.0850
50	0.1134	0.0196	0.0225
100	0.0723	0.0170	0.0165
500	0.0252	0.0161	0.0165
1000	0.0177	0.0159	0.0157
5000	0.0077	0.0155	0.0157
10000	0.0053	0.0154	0.0154
50000	0.0000	0.0153	0.0154

Tabla 4.1: Raíz del error cuadrático medio (RMSE) contra un render de 50k muestras por píxel (MPP) en la escena interior del modelo de la piscina

Durante las primeras iteraciones del algoritmo, cuando la cantidad de muestras es baja, los resultados no son muy buenos lógicamente debido a la falta de información de la imagen. Sin embargo, a pesar de no ser un buen resultado, aplicar el denoiser retorna una estimación de cómo se verá la imagen final (figura 4.3).

Una vez generada una cantidad suficiente de muestras, los resultados del denoiser en sus dos modalidades mejoran considerablemente. La cantidad de iteraciones que necesita el algoritmo para que el programa retorne imágenes como las

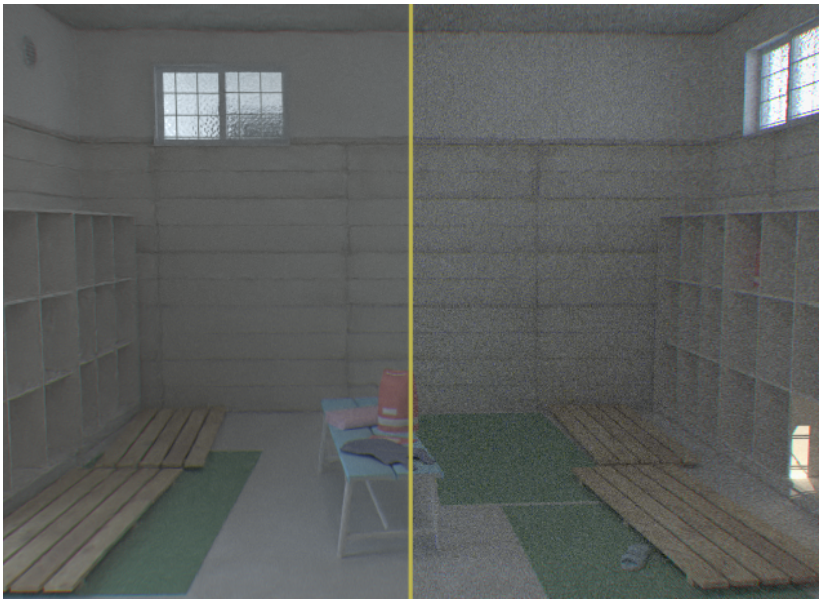


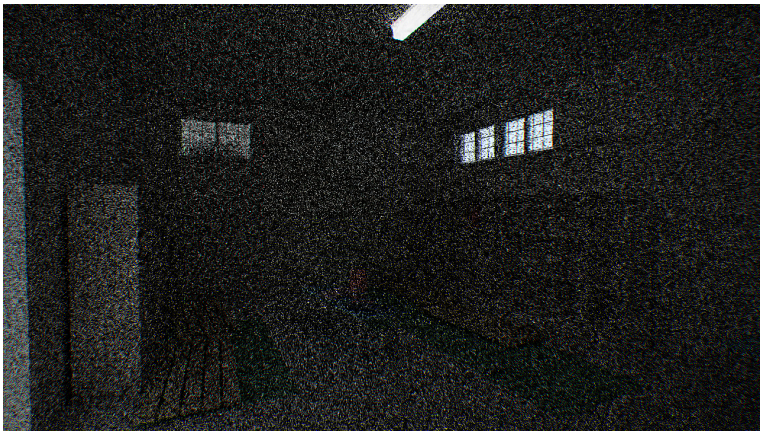
Figura 4.2: Imagen ampliada de un render de 500 muestras por píxel. La mitad izquierda tiene aplicado OptiX denoiser.

presentadas en la sección anterior son menores, sobre todo para escenas de poca iluminación (imagen 4.2). En particular para la escena estudiada, son necesarias alrededor de 5000 muestras por píxel para poder superar la calidad generada al aplicar el denoiser.

En cuanto al uso o no del modo temporal del denoiser, la diferencia entre uno y otro es insignificante. Esto ocurre porque en este caso la aplicación acumula muestras de la salida anterior sobre la actual, por lo tanto pierde sentido retroalimentar la entrada con el cuadro anterior. La ventaja del modo temporal aparece en caso contrario, cuando no es posible acumular muestras del lado del path tracer (trazando más rayos). En ese caso, a medida que pasan los cuadros renderizados, el denoiser se puede encargar de aumentar la calidad de la imagen utilizando resultados anteriores con una distribución de rayos distinta a la actual. Esto tiene un efecto similar a acumular las muestras por píxel. La tabla 4.2 muestra como sin acumular muestras cuadro tras cuadro, la retroalimentación mejora el error a medida que va pasando el tiempo.

Cuadros Renderizados	**RMSE Denoiser Temporal**
1	0.1656
4	0.1381
8	0.1179
12	0.1019
16	0.0794
20	0.0721

Tabla 4.2: RMSE contra un render de 50k muestras por píxel en la escena interior del modelo de la piscina utilizando el modo temporal de una sola muestra por píxel.



(a) Resultado de la primera iteración de path tracing (una sola muestra por píxel)

(b) Resultado de aplicar OptiX denoiser sobre la imagen (a)

Figura 4.3: OptiX denoiser aplicado a una imagen con una sola muestra por píxel

4.3. Rendimiento

4.3.1. Traza de GPU para Aplicación Path Tracer con OptiX Denoiser

De la misma forma que se capturó la ejecución del algoritmo path tracer, se realizó el mismo análisis para la aplicación que utiliza la herramienta OptiX Denoiser. La figura 4.4 contiene el resultado de la traza.

Las características del render son exactamente las mismas. Por esta razón los pipelines existentes en el análisis anterior tomaron prácticamente el mismo tiempo; 19.07ms contra 19.34ms y 2.29ms contra 2.60ms, respectivamente para ray tracing y post procesamiento, en esta nueva traza y la realizada para el algoritmo path tracing sin denoiser.

El cambio más notorio es el espacio vacío entre un pipeline y el otro en cuanto ejecución de buffer de comandos. Tal como fue mencionado anteriormente, OptiX no depende de ninguna cola de ejecución definida por Vulkan. En su lugar se proveen mecanismos de sincronización vía semáforos, que permiten poner en pausa la

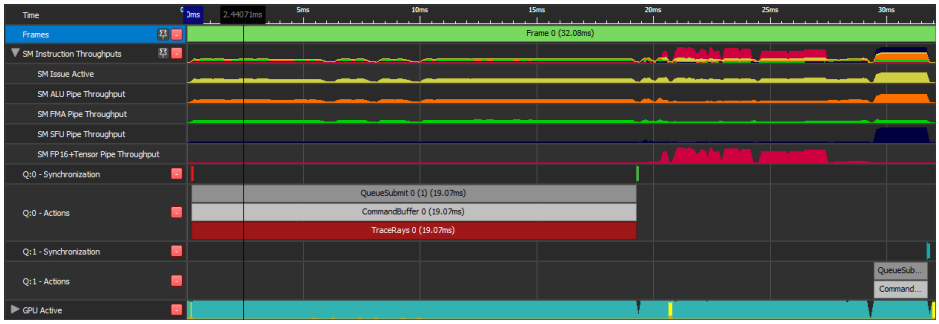


Figura 4.4: Traza de GPU de un cuadro de la escena de la piscina aplicando OptiX Denoiser.

ejecución de un nuevo buffer hasta que la operación de remoción de ruido termine.

Durante el tiempo que el denoiser se encuentra en ejecución, se puede observar un salto en la gráfica de ejecución de instrucciones FP16 para tensor cores. Esto confirma la utilización de este nuevo componente en la arquitectura Turing por parte del denoiser de OptiX.

En total, este cuadro llevó 32.08ms, lo que da aproximadamente 31 cuadros por segundo a resolución HD. Si bien esto corresponde a una sola muestra por píxel, el resultado tiene aplicado el denoiser de OptiX, otorgando la mejora de calidad analizada en la sección 4.2. Aplicar el denoiser tomó 10.72ms extra, un 33.4 % de la totalidad del procesamiento ante un 59.4 % invertido en path tracing y 7.1 % en post procesamiento. Solo resta analizar si con este costo extra es viable su uso en un render en tiempo real.

4.3.2. Rendimiento en Tiempo Real

A continuación se presenta un análisis de rendimiento basado en el FPS promedio para resoluciones HD (1280x720) y Full HD (1920x1080). Con cada resolución se obtiene la métrica en una posición de cámara exterior (imagen 3.10) y una interior (imagen 4.3), con largo máximo de los caminos de path tracer de profundidades máximas de 4, 6, 8 y 10. La tabla 4.3 contiene el resultado de las ejecuciones.

En ambientes cerrados el rendimiento puede bajar hasta 15 cuadros por segundo con esta misma configuración. Pero de la misma forma que se experimentó con el path tracer, bajando la profundidad máxima a 4, se alcanzan 30 cuadros por segundo sacrificando un poco de calidad en la iluminación.

Teniendo en cuenta este rendimiento, un render en tiempo real es viable sin la necesidad de acumular el cálculo realizado cuadro tras cuadro. El problema pasa a ser la calidad del resultado, que depende de la capacidad de la red neuronal a ajustarse a la información provista por el algoritmo. En base a los resultados de la sección 4.2, OptiX puede no producir resultados aceptables con una baja cantidad de muestras. A pesar de eso, como se puede observar en la tabla 4.2, la posibilidad de acumular muestras retroalimentando el algoritmo cuadros anteriores, logra errores relativamente bajos en comparación con un render de 50k muestras.

Más allá de la posibilidad de ejecutar un render en tiempo real, el principal caso de uso de OptiX es reducir la cantidad de muestras necesarias para generar resultados de buena calidad para la industria cinematográfica. Es posible mante-

Resolución	**Profundidad**	**Escena**	**FPS**
HD	4	Exterior	49
HD	6	Exterior	44
HD	8	Exterior	41
HD	10	Exterior	39
HD	**4**	**Interior**	**30**
HD	6	Interior	23
HD	8	Interior	18
HD	10	Interior	15
Full HD	4	Exterior	23
Full HD	6	Exterior	21
Full HD	8	Exterior	19
Full HD	10	Exterior	18
Full HD	4	Interior	14
Full HD	6	Interior	10
Full HD	8	Interior	8
Full HD	10	Interior	7

Tabla 4.3: Rendimiento de la aplicación path tracing con denoiser. Una muestra por píxel, basada en profundidad máxima de caminos, tipo de escena y resolución de ventana.

ner desactivado OptiX durante una determinada cantidad de muestras, obteniendo el rendimiento del capítulo 3, para luego aplicar la red neuronal. El denoiser le dará terminación a imágenes que podrían tomar una cantidad inviable de muestras. Teniendo en cuenta que un solo cuadro de 50k muestras puede tomar unos 45 minutos, aplicar el denoiser considerando un margen de error respecto a la imagen final, ahorraría una cantidad de tiempo sustancial para el renderizado de una animación.

Capítulo 5

Aplicación de Pipeline Híbrido

En contraste con la dificultad para la ejecución de renderizado en tiempo real de las dos aplicaciones presentadas con Monte Carlo Path Tracing, la tercera técnica de renderizado presentada en este capítulo, tiene un enfoque distinto y logra rendimiento superior en cantidad de cuadros por segundo. Se introduce otra forma de aplicar ray tracing, como complemento a las técnicas de renderizado actuales de rasterización.

La aplicación implementada ejecuta primero un pipeline clásico de rasterizado que obtiene las características de la geometría visible y sus materiales, y luego un pipeline de trazado de rayos que calcula la iluminación de la escena y enriquece el resultado agregando reflexiones, refracciones y sombras de luces direccionales. De esta forma se presenta un concepto relativamente nuevo que aparece con la arquitectura Turing en el mundo del renderizado en tiempo real: el pipeline híbrido.

Siendo los videojuegos la principal aplicación de las GPU y gran parte del negocio de NVIDIA, es importante analizar cuáles son las aplicaciones de la nueva arquitectura Turing para esta nueva forma de renderizado. Es por esto que la motivación de esta tercer aplicación implementada, es entender qué ventaja tiene y de qué manera se aplica ray tracing en software de estas características.

5.1. Introducción

Existe una limitante muy importante en aplicaciones logradas vía rasterización al simular las interacciones de la luz a nivel global de una escena tridimensional. Por la forma en la que se plantea el pipeline de rasterizado, la idea de realizar cálculos teniendo en cuenta todas las superficies con las que cada camino (path) de luz puede interactuar termina siendo poco práctica. Ya en la etapa de geometría se comienzan a descartar elementos de la escena que no van a ser tenidos en cuenta en el resultado final de la imagen. Y al llegar a la última etapa, se procesa independientemente cada píxel únicamente teniendo en cuenta los fragmentos proyectados en ese punto.

A continuación se presentan dos de los problemas de renderizado más comunes a resolver utilizando un pipeline de rasterizado; reflexiones y sombras. Esta aplicación utiliza trazado de rayos como complemento al rasterizado, para generar una composición final que obtiene lo mejor de ambas técnicas. El resultado es un pipeli-

ne híbrido que mantiene un rendimiento aceptable para renderizado en tiempo real y mejora la calidad de los resultados con rasterización utilizando técnicas de ray tracing. En este caso se deja de lado el cálculo más costoso para el ray tracing que es la iluminación difusa global, pero logra un resultado que se ajusta al comportamiento de la luz, es sencillo y a la vez eficiente para ser aplicado en un programa de renderizado en tiempo real.

5.1.1. Problemas de Rasterización

Reflexiones

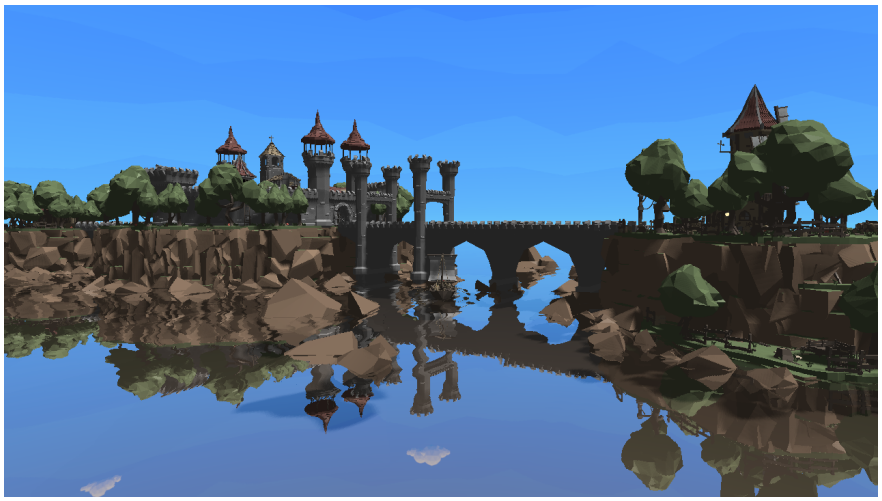


Figura 5.1: Ejemplo de render de reflexión con rasterizado.

Muchos de los efectos de iluminación logrados en las aplicaciones implementadas con rasterización son soluciones alternativas a lo que un algoritmo de trazado de rayos logra de forma sencilla. Un ejemplo para demostrar este problema son las reflexiones especulares.

En la imagen 5.1 se generó una escena aplicando una de las soluciones posibles para el problema de las reflexiones en una aplicación con rasterización. La aplicación se encarga de enviar para cada uno de los objetos en la escena, la información de materiales y los shaders que corresponden utilizar en el pipeline. La geometría que corresponde al agua no es más que un plano dentro del conjunto de objetos, utilizando un shader especial que utiliza una textura con el color reflejado. La dificultad de esta técnica radica en cómo obtener esta textura en una iteración previa al pasaje de render final. La solución aplicada en este ejemplo es reflejar temporalmente la posición de la cámara basándose en el plano que representa el agua, se renderiza la escena desde esa posición y se guarda el resultado como una textura.

Esta solución no es la más utilizada porque requiere renderizar dos veces todos los objetos, esto puede significar un golpe importante en el rendimiento de escenas complejas. La solución más utilizada es reflexiones en espacio de cámara. Esta solución es más eficiente ya que se proyecta el mismo resultado del render final como textura de reflejo. Se utiliza la información de normales y el buffer de profundidad calculado en el mismo pipeline de rasterizado. Si bien es más eficiente en compa-

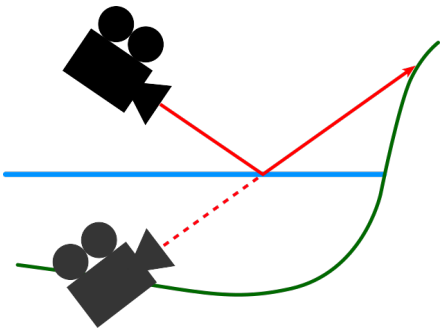


Figura 5.2: Técnica de render de reflexión con cámara auxiliar reflejada por debajo del nivel del agua.

ración con la solución anterior, se omiten objetos que no son directamente visibles por la cámara, y en algunos casos puede generarse un resultado no del todo realista.

Ninguno de los métodos anteriores soluciona reflexiones en tiempo real de forma realista para todos los casos, y la realidad es que no existe una forma perfecta y eficiente de generar este efecto en tiempo real con rasterizado. Se pueden utilizar combinaciones de los métodos anteriores dependiendo del caso, o simplemente utilizar texturas auxiliares pre-generadas como un *skybox*, que contiene el resultado de un render para todo el domo que rodea el objeto. Los reflejos con texturas auxiliares no son perfectos para todos los puntos de una superficie, y si no son generadas en tiempo real, se omiten objetos animados o cambios de iluminación de la escena.

Sombras

Al igual que con las reflexiones, la generación de sombras también requiere información auxiliar para determinar si en un punto la superficie está bajo una sombra o no. La solución más común es un *mapa de sombras*, típicamente representado con una textura bidimensional que contiene la información de profundidad de todos los elementos "visibles" por una luz. Para este efecto también se utiliza una cámara auxiliar, pero posicionada en lugar de la luz con un tipo de proyección que depende de las características de la misma. En el caso del sol se utiliza una proyección paralela, mientras que para una luz puntual se utiliza una proyección de 360 grados. Una vez determinadas las características de la cámara, la generación del mapa de sombras es parte del pipeline de rasterizado pero no implica volver a renderizar toda la escena. En la generación de mapa de sombras, sólo interesa la información de profundidad calculada en la etapa de rasterización para cada fragmento.

La dificultad de generar mapas de sobra radica en el tamaño o resolución de la textura, y el espacio que se quiere cubrir. Generar el mapa de sombras del sol en un espacio abierto puede ser muy exigente para algunas tarjetas gráficas. Un mapa de sombras demasiado grande deja de ser eficiente y ocupa espacio en memoria que podría ser utilizado con otro propósito.

Es difícil de generar sombras duras de buena calidad con un mapa de sombras, e imposible generar sombras de luces de área, ya que el mapa está basado en una proyección de cámara. Las sombras de fuentes de luz no puntuales o direccionales, deben ser pre-renderizadas en las texturas de los materiales, no existe una forma

de generarlas en tiempo real con rasterización.

5.1.2. Solución con Ray Tracing

Para la aplicación presentada en el capítulo 3, no existen los problemas para generar reflexiones y sombras que tienen las aplicaciones por rasterización. Pero tal como fue comprobado durante el estudio de rendimiento (sección 3.5), no es sencillo que esta aplicación funcione en tiempo real con alta resolución.

La solución presentada a continuación es un «pipeline híbrido», ya que implementa un pipeline clásico de rasterizado y un pipeline de trazado de rayos que enriquece la salida del primero agregando reflexiones, refracciones y sombras de luces direccionales.

Lo primero que se ejecuta para cada cuadro renderizado es el pipeline de rasterizado, donde se aloja la información requerida por el pipeline de trazado de rayos para que sea posible enriquecer la imagen agregando los nuevos efectos. Para esta implementación se necesitan cinco texturas de una resolución igual a la de salida del render. Para cada píxel se guarda la siguiente información:

- Vector normal.
- Si corresponde, porcentaje de reflexión y refracción e índice de refracción de la superficie.
- Profundidad desde la cámara.
- Características del material, brillo y especularidad para el cálculo del modelo Phong (ecuación 2.3).
- Albedo del material.

La lógica definida en el shader de generación de rayos en el pipeline de ray tracing, define si se debe trazar un rayo o no en base a la información antes obtenida, el resultado de cada rayo es añadido de forma recursiva en el resultado final de la imagen.

El origen del rayo es calculado a partir de la matriz de transformación y proyección de la cámara, y el mapa de profundidad. Notar que con este procedimiento se ahorra el trazado del rayo primario utilizando rasterización, es decir, no se ejecutarán $W \times H$ recorridos por la estructura de aceleración siendo W y H el ancho y alto de la resolución de salida definida.

La dirección del rayo en el caso de que el material refleje o transmita, se calcula en base al vector normal obtenido con el mapa de normales y el índice de refracción, también alojado en una textura calculada en el pipeline de rasterizado. Y la dirección del rayo para chequeo de sombras es simplemente la dirección del sol.

El caso de las luces de área no fue cubierto en esta solución. Las sombras de área generan ruido ya que se deben distribuir rayos en toda la superficie de la geometría que representa la luz. Este ruido debe ser removido de alguna forma. OptiX puede ser una solución, pero también existen otras herramientas dedicadas especialmente a quitar ruido de sombras en aplicaciones de renderizado en tiempo real [31].

5.2. Arquitectura de la Solución

Se presenta a continuación una representación de alto nivel de la solución implementada. De la misma forma que las otras dos aplicaciones, el centro de la arquitectura es una aplicación Vulkan encargada de administrar los recursos y la sincronización con el hardware gráfico. Se cuenta con las texturas de salida del pipeline de rasterización utilizados como entrada del pipeline de ray tracing, el mapa de normales, profundidad, refracción y reflexión, mapa de albedo y mapa de material con la información de especularidad.

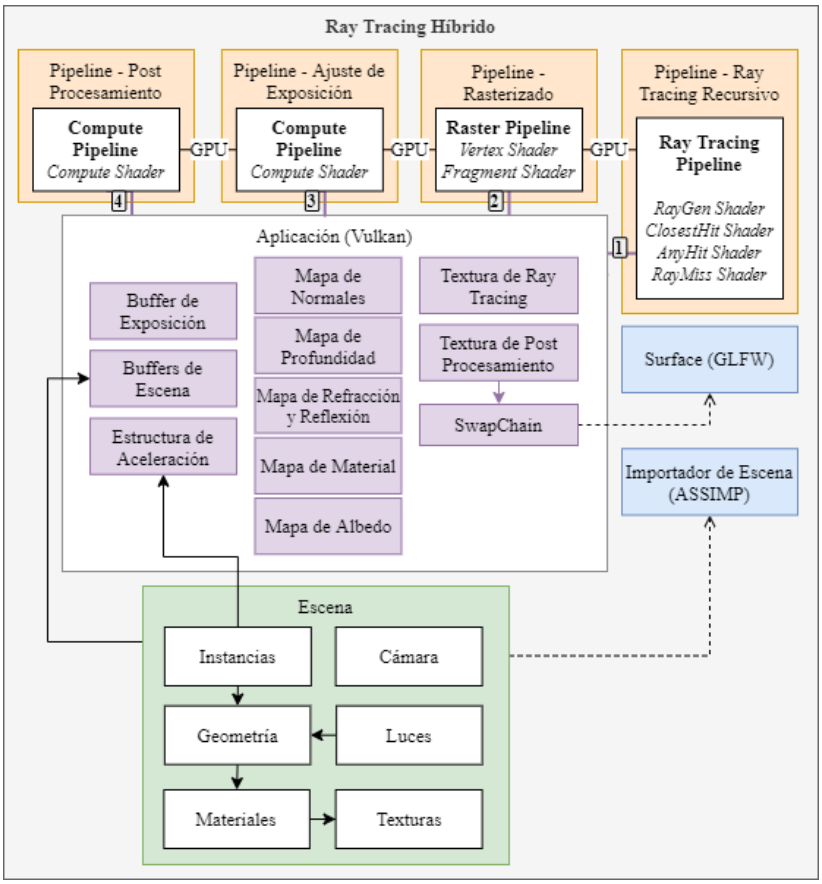


Figura 5.3: Arquitectura de sistema de ray tracing híbrido con rasterización.

5.2.1. Etapa de Rasterización

El shader correspondiente a la etapa de fragmentos del pipeline de rasterización, es el encargado de especificar cada uno de los píxeles de las texturas de entrada del siguiente pipeline. En términos de trazado de rayos, el cálculo realizado en este shader corresponde al rayo primario del algoritmo.

Una de las consideraciones especiales que debe hacer esta etapa, es ignorar las superficies que tienen texturas transparentes, por ejemplo para renderizar correctamente las rejas que rodean la piscina. Para esto la etapa de fragmentos ofrece la posibilidad de descartar un fragmento determinado. Dado que los fragmentos se calculan para todos los triángulos del z-buffer, descartar uno como consecuencia de

su transparencia, hace que el fragmento que termine asignando el color del píxel, sea el siguiente en orden de distancia a la cámara.

Usualmente la salida de un shader de fragmentos es una única textura que luego es presentada en la ventana de salida. Pero en este caso se aplica una estrategia comúnmente utilizada para «Deferred Shading» [9, Cap. 20], donde se definen varias texturas de salida, cada una representando los distintos mapas requeridos por la etapa de ray tracing. En Deferred Shading, estos mapas que contienen información de las superficies de la escena son denominados *G-buffers* (geometric buffers).

5.2.2. Etapa de Ray Tracing

El algoritmo de trazado de rayos fue modificado para adaptarse a una aplicación que no tiene el enfoque estocástico de path tracing. En este caso el algoritmo se parece un poco más a Whitted ray tracing [6], donde los rayos se ramifican en los casos de reflexión, refracción y rayos de sombra. El algoritmo es recursivo para reflexiones y refracciones, con una profundidad máxima configurable de igual manera que en la aplicación path tracing.

Los rayos de sombra se lanzan tanto en *Closest Hit* como en *Ray Generation*. Esto es porque en este caso la etapa *Ray Generation* corresponde a la primera colisión más cercana, cuyo cálculo no es necesario ya que se deduce directamente a partir del mapa de profundidad obtenido del pipeline de rasterización. Algo parecido ocurre con los rayos de reflexión y refracción, con la salvedad de que el resultado de las ecuaciones de Fresnel [1, Cap. 8], ya viene calculado en uno de los mapas de entrada.

5.2.3. Etapa de Post Procesamiento

Durante la etapa de rasterización no se efectúa ninguna técnica de antialiasing. Esto es porque el suavizado del valor de los píxeles de las texturas de salida, provoca efectos no deseados al momento de realizar la composición final con el color de los píxeles calculados trazando rayos. Aplicando antialiasing se obtienen valores incoherentes en la interpolación de bordes de algunas superficies, especialmente para los mapas vector normal y profundidad.

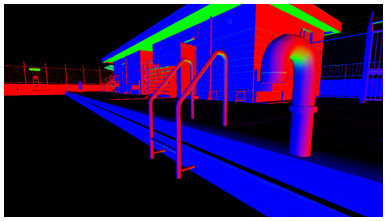
Por este motivo, los shaders de post procesamiento fueron levemente modificados con el fin de mejorar el efecto de aliasing de píxeles del renderizado. Un shader de cómputo general aplica un algoritmo de antialasing para post procesamiento llamado FXAA.

El suavizado aproximado rápido (FXAA) es un algoritmo de suavizado de espacio de pantalla creado por Timothy Lottes en NVIDIA [32].

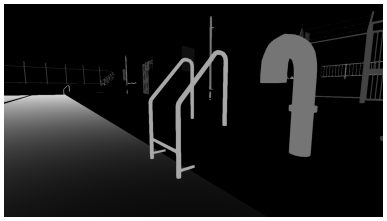
5.3. Resultados

Las reflexiones y refracciones se comportan de igual manera que en la aplicación path tracing. Este tipo de efectos es muy difícil de generar y pocas veces se ve en videojuegos y aplicaciones de render en tiempo real.

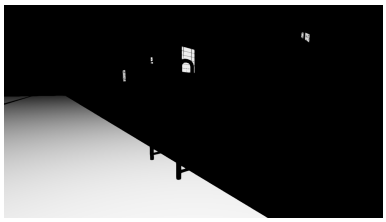
No se generó ningún tipo de iluminación indirecta ni sombras suaves de luces de área. Por esta razón los resultados no tienen ningún tipo de ruido y la calidad de la iluminación depende de las texturas de la escena. Para este tipo de aplicaciones,



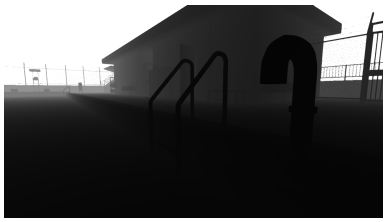
(a) Normales



(b) Reflexión

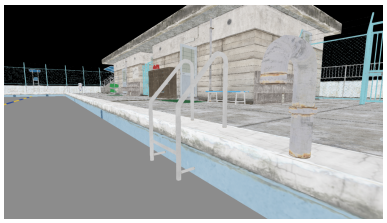


(c) Refracción

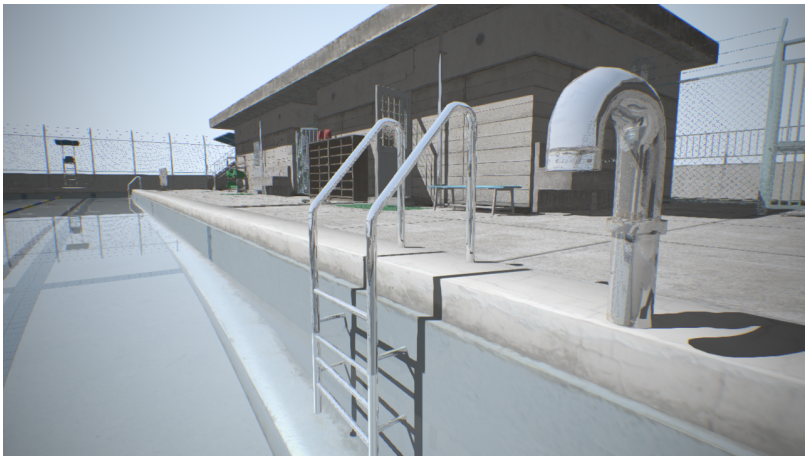


(d) Profundidad

(e) Especularidad



(f) Albedo



(g) Resultado

Figura 5.4: Composición del pipeline híbrido

estos efectos usualmente se calculan para toda la escena de forma previa, y se guardan en texturas en lugar de generarlos en cada cuadro.

La figura 5.4 contiene una descomposición del proceso de renderizado, las imágenes de salida de la etapa de rasterización y el resultado final aplicando ray tracing y post procesamiento.

5.4. Rendimiento

Al igual que en el estudio de rendimiento de las otras dos aplicaciones, la GPU utilizada para es la *NVIDIA GeForce RTX 2060 SUPER*, correspondiente a la arquitectura Turing *TU106* cuyas características son detalladas en el anexo A.

Se utilizó Nsight Graphics de NVIDIA para evaluar el rendimiento de la aplicación.

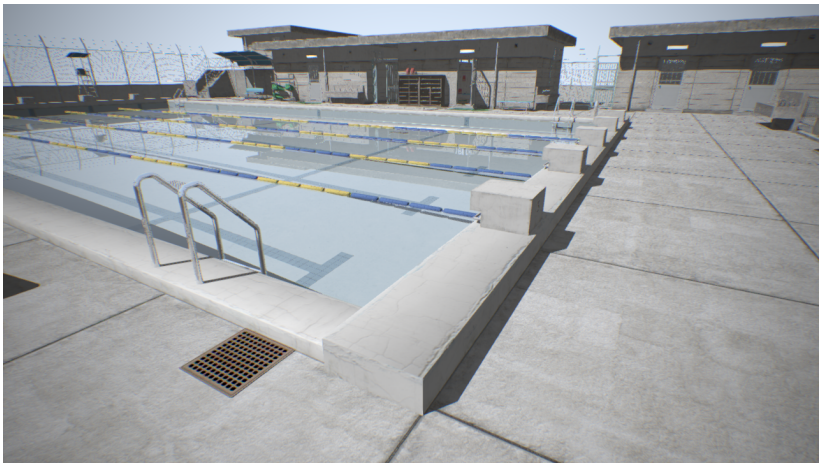


Figura 5.5: Escena utilizada para los estudios de rendimiento de la aplicación de render híbrido.

Como caso de estudio para evaluar el rendimiento de la aplicación, se toma la traza de ejecución del GPU para renderizar la imagen 5.5 de la escena de la piscina. Se ejecuta la misma escena que para las aplicaciones del capítulo 3 utilizando la misma posición y configuración de cámara, y la resolución de salida es 1280x720 (resolución HD).

La figura 5.6 contiene la traza de un cuadro renderizado. El primer buffer ejecutado es el que contiene la operación de rasterización seguida del trazado de rayos. En la captura tomada como ejemplo, esta operación toma 3.25ms, donde 0.03ms corresponden a una pequeña espera por sincronización de recursos, 0.61ms al pipeline de rasterización y 2.61ms al pipeline de ray tracing.

El segundo buffer de comandos contiene las operaciones correspondientes al post procesamiento. En la captura tomada como ejemplo, esta operación cuesta 2.91ms. Esta operación de post procesamiento es un poco más costosa que la analizada en las aplicaciones de ray tracing puro, ya que además del mapeo tonal, se ejecuta el algoritmo FXAA [32] como método de anti-aliasing.

En total, este cuadro llevó 6.78ms, lo que da aproximadamente 147 cuadros por segundo a resolución HD.

5.4.1. Rendimiento en Tiempo Real

A continuación se presenta un análisis de rendimiento basado en el FPS promedio para resoluciones HD (1280x720), Full HD (1920x1080). En contraste con los análisis de rendimiento de las aplicaciones path tracing, la profundidad de los rayos pasa a ser irrelevante ya que no se calcula iluminación global. La tabla 5.1

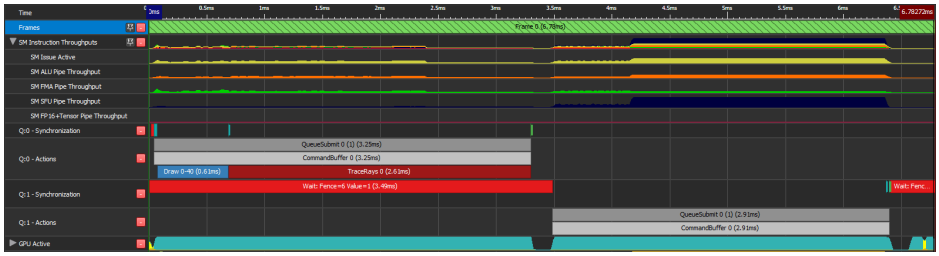


Figura 5.6: Traza de GPU de un cuadro de la escena de la piscina renderizado con la aplicación de pipeline híbrido.

contiene el resultado de las ejecuciones.

Resolución	**Escena**	**FPS**
HD	Exterior	182
HD	Interior	134
Full HD	Exterior	100
Full HD	Interior	90

Tabla 5.1: Rendimiento de la aplicación de ray tracing híbrido.

En resolución Full HD, la escena más exigente llega a 90 cuadros por segundo. Con estas resoluciones de pantalla, la aplicación perfectamente podría soportar en tiempo real nuevos efectos con trazado de rayos o más técnicas clásicas del pipeline de rasterización manteniéndose por encima de los 60 cuadros por segundo.

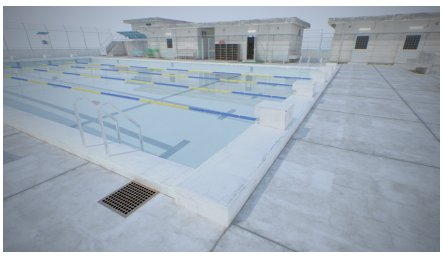
5.5. Comparación con las aplicaciones anteriores

La implementación presentada en este capítulo demuestra la capacidad del hardware de complementar las técnicas clásicas de renderizado con ray tracing. Sin embargo, la dificultad de implementación de un render con resultados de aspecto similar a una aplicación de ray tracing puro utilizando técnicas de rasterización, escapa del alcance del proyecto.

Los resultados del render siempre van a verse más reales si se tiene en cuenta iluminación global difusa con ray tracing (figura 5.7 (a) y (b)). Pero el análisis de rendimiento de la aplicación Path Tracing Monte Carlo, arroja que sin una herramienta de denoising, esto es inviable para renderizado en tiempo real. Por esta razón no es posible realizar una comparación justa entre los algoritmos de ray tracing puro aplicados en los capítulos 3 y 4, y esta implementación híbrida, tanto en el aspecto cualitativo como en el de rendimiento.

La segunda aplicación, presentada en el capítulo 4, utiliza OptiX denoiser para lograr un mejor rendimiento logrando imágenes de calidad similar a las de Path Tracing Monte Carlo (figura 5.7 (c) y (d)), pero las exigencias de los videojuegos modernos hacen que no sea suficiente. Actualmente se esperan resoluciones Full HD o 4K y hasta 120 cuadros por segundo. Es así que Path Tracing Monte Carlo sigue siendo una técnica para generar visualizaciones hiper realistas en tiempos apenas interactivos o fuera de linea para cine y televisión. Para el caso de los videojuegos aparece la herramienta del pipeline híbrido, que lejos de pretender

generar la misma calidad de resultados (figura 5.7 (e) y (f)), le da un propósito real al nuevo hardware de la tarjeta gráfica, ofreciendo potencial para mejorar sustancialmente efectos puntuales sobre las técnicas clásicas de renderizado por rasterización.

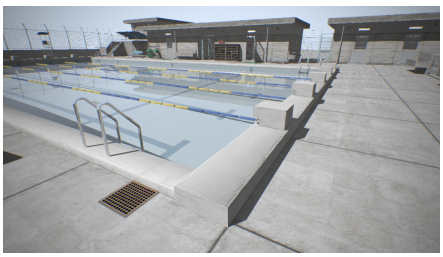


(a) Path tracer: Piscina — 1000 muestras por píxel — Profundidad 10 — Un solo cuadro 16.7 segundos

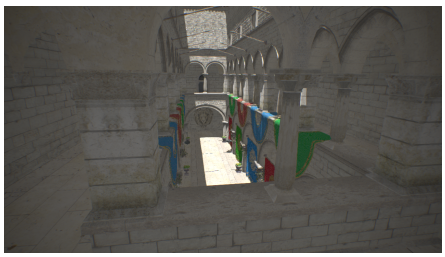
(b) Path tracer: Sponza — 1000 muestras por píxel — Profundidad 10 — Un solo cuadro 42.3 segundos

(c) Path tracer con denoiser: Piscina — Sólo una muestra por píxel — Profundidad 10 — 39 cuadros por segundo

(d) Path tracer con denoiser: Sponza — Sólo una muestra por píxel — Profundidad 10 — 20 cuadros por segundo



(e) Híbrido: Piscina — 182 cuadros por segundo



(f) Híbrido: Sponza — 250 cuadros por segundo

Figura 5.7: Comparación de renders generados por las tres aplicaciones del proyecto en resolución HD (1280x720)

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Se presentan a continuación las conclusiones que resultaron del trabajo realizado sobre la implementación de técnicas de ray tracing en GPU, utilizando las nuevas características de la arquitectura Turing y las API gráficas con capacidades para acceder a sus funcionalidades.

6.1.1. Acceso a Ray Tracing por GPU

El marco teórico de las técnicas de ray tracing y su comparación con rasterización, ayudó a entender las dificultades de su implementación para el hardware existente. En ese sentido, la principal motivación de las modificaciones en la arquitectura Turing, es facilitar la implementación de algoritmos de trazado de rayos en el hardware gráfico, que hasta el momento eran muy difíciles de lograr.

Hoy en día la utilización de GPU parece ser un estándar para el software de renderizado, ya que la funcionalidad está presente en las principales herramientas de producción 3D: Blender, Maya, 3ds Max y Cinema4D. Es por esto que independientemente del hardware específico agregado en Turing, el cambio más relevante es la aparición del nuevo pipeline de trazado de rayos a nivel de controlador. Las funcionalidades de trazado de rayos están disponibles para los desarrolladores y se facilita la implementación de aplicaciones como la presentada en el capítulo 3.

6.1.2. Complemento de Inteligencia Artificial

Los *RT cores* son los elementos que teóricamente hacen posible el buen rendimiento de una aplicación que utiliza ray tracing, pero los estudios realizados para la aplicación de ray tracing puro en el capítulo 3, arrojaron que en su máxima expresión, un render físicamente realista sigue siendo muy exigente para el nuevo hardware.

Al ver que en tiempo real, el pipeline de ray tracing lucha para generar una imagen sin ruido, aparecen los Tensor Cores como otro componente fundamental parte del conjunto de funcionalidades nuevas de Turing. Las capacidades de la GPU para aplicar inteligencia artificial no es sólo una funcionalidad extra debido

al crecimiento de popularidad del entrenamiento de redes neuronales con GPU. Los Tensor Cores son un complemento para procesamiento de imágenes, y con ellos se puede lograr, entre otras cosas, que se ejecuten de forma efectiva algoritmos que de otra forma no serían posibles de aplicar en tiempo real.

La utilización de redes neuronales para post procesamiento abre un universo de posibilidades, la remoción de ruido no es la única aplicación de los Tensor Cores. NVIDIA promociona redes neuronales de super resolución como DLSS [1], con el fin de aumentar la cantidad de cuadros por segundo renderizando en resoluciones bajas y aumentando el tamaño de la imagen de salida. Con el tiempo van a surgir herramientas para generar renderizado en tiempo real de características físicas potenciado por redes neuronales, OptiX Denoiser y NVIDIA DLSS son solo ejemplos de lo que se puede lograr con esta tecnología.

6.1.3. Pipeline Híbrido como Solución Definitiva

NVIDIA menciona en el whitepaper oficial del lanzamiento de la arquitectura, que «Turing introduce **ray tracing en tiempo real** logrando que una sola GPU renderice juegos 3D y modelos complejos visualmente realísticos, con sombras, reflexiones y refracciones físicamente precisas».

Lo cierto es que cuando mencionan ray tracing en tiempo real, no necesariamente se está hablando de Path Tracing Monte Carlo físicamente realista. Se trata más de funcionalidades específicas que pueden complementar un rendering en tiempo real, ya sea sombras, reflexiones o refracciones. Esto es, en esencia, rendering híbrido combinando rasterización y ray tracing.

Los RT Cores de Turing fueron creados para utilizase como complemento a las técnicas actuales de rasterización y no es relevante saber si sirven o no para ejecutar ray tracing como reemplazo del pipeline de rasterización. Lo que ocurre es que dejar de utilizar las técnicas clásicas de renderizado en pos de aplicaciones con ray tracing puro, actualmente no tiene sentido, ya que la velocidad del hardware para ejecutar rasterización es una gran ventaja. La implementación híbrida del capítulo 5, es un básico ejemplo del potencial de las nuevas funcionalidades en combinación con los avances actuales en el área de renderizado en tiempo real.

Las capacidades del hardware actual ya son muy buenas únicamente basándose en técnicas de rasterización. Pero la posibilidad de generar gráficos físicamente precisos con técnicas de ray tracing, hace que en la actualidad, el pipeline híbrido sea la solución definitiva para aplicaciones de renderizado en tiempo real utilizando una sola GPU de uso doméstico.

6.2. Trabajo Futuro

En esta sección se enumeran posibles mejoras que quedaron fuera del alcance del proyecto para las aplicaciones implementadas, así como también posibles aspectos del hardware, que no pudieron ser explorados con suficiente profundidad por falta de tiempo o acceso a los dispositivos.

[1] NVIDIA DLSS: `https://www.nvidia.com/es-la/geforce/technologies/dlss/`

6.2.1. Render Físico con Path Tracing Monte Carlo

Como función BRDF para la aplicación path tracing se utilizó el método Phong de reflexión especular. Este método provee facilidades para su implementación, tanto a nivel del algoritmo para renderizado, como al momento de definir materiales para su importación. La desventaja de Phong es que no se considera una función BRDF físicamente realista. Un modelo de microfacetado como Oren-Nayar se ajustaría mejor a un rendering con estas características.

Implementar un modelo BRDF físicamente realista también implica la definición de los materiales de la escena teniendo en cuenta las características de la superficie que representan. El trabajo de definir cada uno de estos materiales en una escena tan grande como la de la piscina estuvo fuera del alcance del proyecto. Sin embargo, el soporte de BRDF para materiales más avanzados aumentaría la calidad de los resultados considerablemente, y se ajustaría a los estándares de los programas de renderizado 3D utilizados comercialmente.

6.2.2. Deep Learning para Procesamiento de Imágenes en Tiempo Real

La herramienta OptiX Denoiser utilizada para la eliminación del ruido generado en la aplicación path tracing, no está diseñada para renderizado en tiempo real. Quedó comprobado que son malos los resultados logrados con la poca cantidad de muestras que se pueden lanzar por píxel, para una buena cantidad de cuadros por segundo.

En base al trabajo realizado sobre la aplicación path tracing sin aplicar OptiX, se podría comenzar un nuevo proyecto en el área de deep learning para procesamiento de imágenes, con el objetivo de generar una red neuronal con la capacidad de eliminar el ruido con los tensor cores.

6.2.3. Extensión del Pipeline Híbrido

En la aplicación implementada con pipeline híbrido, apenas explora el potencial de utilizar los pipelines de rasterizado y ray tracing de forma conjunta. La publicación PICA PICA del libro Ray Tracing GEMS [22] es una demostración de la capacidad del hardware y lo que se puede lograr utilizando ambos pipelines.

Con la ayuda del ray tracing se pueden agregar efectos de iluminación global, sombras de área o cáusticas, y a su vez es posible implementar otros efectos con el pipeline de rasterización, como oclusión ambiental y desenfoque de profundidad.

6.2.4. Estructuras de Aceleración Dinámicas

No se implementó soporte para animaciones de geometría, como consecuencia las escenas renderizadas son completamente estáticas y nunca son alteradas sus estructuras de aceleración. Las pruebas de rendimiento realizadas durante la parte práctica del proyecto, no tuvieron en cuenta los tiempos de reconstrucción de las estructuras de aceleración cuadro a cuadro.

La eficiencia en la reconstrucción de estructuras de aceleración depende de la habilidad del programador para agrupar adecuadamente estructuras bottom-level, de modo que no se reconstruyan innecesariamente elementos de la escena que no fueron modificados. Para videojuegos este aspecto es vital, pero se requiere un análisis de las capacidades del CPU para trabajar en conjunto con el GPU animando

la geometría y reconstruyendo las estructuras de aceleración sin afectar la calidad del render.

6.2.5. Comparaciones de Rendimiento con otras GPU

Generación Turing

Cada generación que lanza NVIDIA cuenta con una variedad de especificaciones para la misma arquitectura. En el caso de Turing en el anexo A se detallaron las especificaciones de los chips TU102, TU104 y TU106. Las pruebas de este proyecto fueron realizadas con una tarjeta gráfica TU106, que en términos de rendimiento, es el hardware con menor capacidad de los disponibles para la arquitectura Turing. Es posible en el futuro ejecutar nuevamente las aplicaciones en una tarjeta TU102 con el mejor hardware disponible de la generación.

Generación Pascal

Algunos modelos de la generación Pascal de NVIDIA soportan trazado de rayos sin contar con RT Cores. La eficiencia ganada con las nuevas unidades para trazado de rayos, no queda clara si no es posible compararla contra un hardware que no cuenta con ella.

Generación Ampere

La nueva generación de NVIDIA, cuenta con una nueva versión de los RT Cores que duplica la capacidad de calcular intersecciones [14]. Algo parecido ocurre con los tensor cores, que también se describen con nuevas capacidades. Sería interesante evaluar nuevamente los resultados del rendimiento de la aplicación con path tracing en tiempo real, si las capacidades de Ampere duplican a las de Turing.

Otros Fabricantes

AMD recientemente lanzó su propio hardware con aceleración para trazado de rayos [15], una de las ventajas de utilizar Vulkan es su portabilidad y su abstracción al hardware. De contar con el acceso a estos dispositivos sería posible una comparación de rendimiento con el hardware de NVIDIA.

Referencias

[1] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[2] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[3] Brent Burley. Physically based shading at disney. In *SIGGRAPH Practical Physically Based Shading in Film and Game Production course*, volume 2012, pages 1–7. vol. 2012, 2012.

[4] Godot Engine shading documentation. `https://docs.godotengine.org/en/3.0/tutorials/shading/shading_language.html#id1`, 2021.

[5] Michael Oren and Shree K Nayar. Generalization of lambert's reflectance model. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 239–246, 1994.

[6] Turner Whitted. An improved illumination model for shaded display. *SIGGRAPH Comput. Graph.*, 13(2):14, August 1979.

[7] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.

[8] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, pages 1–12. 2017.

[9] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition.* A. K. Peters, Ltd., USA, 4th edition, 2018.

[10] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[11] NVIDIA. Nvidia geforce gtx 1080 whitepaper. 2016.

[12] NVIDIA. Nvidia turing gpu architecture whitepaper. 2018.

[13] NVIDIA. Nvidia tesla v100 gpu architecture whitepaper. 2017.

[14] NVIDIA. Nvidia ampere ga102 architecture whitepaper. 2020.

[15] AMD. Arquitectura de gráficos amd rdna™ 2, 2020.

[16] Direct3D 12 ray tracing documentation. https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-raytracing, 2018.

[17] Daniel Koch. Ray tracing in vulkan, Dec 2020.

[18] Christoph Schied. Q2vkpt, Jan 2019.

[19] Lukas Lipp. *Real-Time Ray Tracing in Quake III*. PhD thesis, Wien, 2020.

[20] Vulkan ray tracing best practices for hybrid rendering. https://www.khronos.org/blog/vulkan-ray-tracing-best-practices-for-hybrid-rendering, Nov 2020.

[21] Project pica pica. https://www.ea.com/seed/news/seed-project-picapica, 2021.

[22] Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. Hybrid rendering for real-time ray tracing. In *Ray Tracing Gems*, pages 437–473. Springer, 2019.

[23] List of ray tracing software. https://en.wikipedia.org/wiki/List_of_ray_tracing_software, 2021.

[24] VV Sanzharov, AI Gorbonosov, VA Frolov, and AG Voloboy. Examination of the nvidia rtx. In *Proceedings of the 29th International Conference on Computer Graphics and Vision (GraphiCon 2019)*, volume 2485, page 7, 2019.

[25] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *High Performance Graphics (Short Papers)*, pages 7–13, 2019.

[26] Mattias Ulmstedt and Joacim Stålberg. Gpu accelerated ray-tracing for simulating sound propagation in water, 2019.

[27] Stefan Zellmann, Martin Weier, and Ingo Wald. Accelerating force-directed graph drawing with rt cores. In *2020 IEEE Visualization Conference (VIS)*, pages 96–100, 2020.

[28] Tiny encryption algorithm. https://es.wikipedia.org/wiki/Tiny_Encryption_Algorithm, 2021.

[29] Linear congruential generator. https://en.wikipedia.org/wiki/Linear_congruential_generator, 2021.

[30] Gamma correction. https://learnopengl.com/Advanced-Lighting/Gamma-Correction, 2021.

[31] Amd fidelityfx. https://www.amd.com/es/technologies/radeon-software-fidelityfx, 2021.

[32] Timothy Lottes. *FXAA*, Feb 2009.

Índice de tablas

Índice de figuras

Apéndice A

Especificaciones de las GPU Turing y Ampere

Característica	**GA102**
Arquitectura	Ampere
GPC	7
TPC	41
SMs	82
CUDA Cores / SM	128
CUDA Cores / GPU	10496
Tensor Cores / SM	4 (3ra Gen)
Tensor Cores / GPU	328 (3ra Gen)
RT Cores	82 (2da Gen)

Tabla A.1: Características de la tarjeta gráfica GA102 (RTX3090Ti) de arquitectura Ampere [14]

Característica	**TU102**
Arquitectura	Turing
GPC	6
TPC	36
SMs	72
CUDA Cores / SM	64
CUDA Cores / GPU	4608
Tensor Cores / SM	8 (2da Gen)
Tensor Cores / GPU	576 (2da Gen)
RT Cores	72 (1ra Gen)

Tabla A.2: Características de la tarjeta gráfica TU102 (RTX2080Ti) de arquitectura Turing [14]

Característica	**TU104**
Arquitectura	Turing
GPC	5 o 6
TPC	20
SMs	40
CUDA Cores / SM	64
CUDA Cores / GPU	2560
Tensor Cores / SM	8 (2da Gen)
Tensor Cores / GPU	320 (2da Gen)
RT Cores	40 (1ra Gen)

Tabla A.3: Características de la tarjeta gráfica TU104 (RTX2080 o RTX2070 FE) de arquitectura Turing [14]

Característica	**TU106**
Arquitectura	Turing
GPC	3
TPC	18
SMs	36
CUDA Cores / SM	64
CUDA Cores / GPU	2304
Tensor Cores / SM	8 (2da Gen)
Tensor Cores / GPU	288 (2da Gen)
RT Cores	36 (1ra Gen)

Tabla A.4: Características de la tarjeta gráfica TU106 (RTX2070 o RTX2060 SUPER) de arquitectura Turing [12]

Apéndice B

Cronograma de Trabajo

Aquí se presenta el cronograma de trabajo del proyecto iniciado en Abril 2020 y finalizado en Agosto 2021.

Abril 2020

- Investigación de arquitectura Turing.
- Planteo de las distintas posibilidades de implementación y herramientas disponibles para acceder a las funcionalidades necesarias para la parte práctica del proyecto.
- Comienza capacitación y entrenamiento de Vulkan API.

Mayo 2020

- Primera aplicación por rasterización con Vulkan.
- Carga y render de primer modelo 3D.
- Adquisición de la tarjeta gráfica utilizada en el proyecto.

Junio 2020

- Capacitación específica de la extensión provisional de Vulkan para ray tracing llamada *vk_nv_ray_tracing*.

Julio 2020

- Investigación con motivo de implementación de path tracing Montecarlo.
- Implementación de pipeline de ray tracing con Vulkan.

Agosto 2020

- Comienza implementación de path tracer Montecarlo.

Setiembre 2020

- Mejoras en la carga universal de modelos 3D con Assimp. Importación de luces y cámaras desde el archivo.
- Implementación de reflexiones con ray tracing.

Octubre 2020

- Implementación de refracciones con ray tracing.
- Implementación de lógica para rayos perdidos y la etapa Any Hit para objetos semi transparentes.

Noviembre 2020

- Se agrega opción para multi muestreo por píxel y acumulación de muestras cuadro a cuadro.

Diciembre 2020

- Implementación de luces de área por geometría.
- Comienza implementación del pipeline híbrido.

Enero 2021

- Refactorización de código por el lanzamiento oficial de vk_KHR_ray_tracing.
- Primera versión de path tracer Montecarlo.
- Comienza el trabajo de informe para el marco teórico.

Febrero 2021

- Finaliza implementación del pipeline híbrido.
- Se agrega el cálculo automático de exposición en post procesamiento.

Marzo 2021

- Integración de OptiX denoiser en la aplicación path tracing Montecarlo.

Abril 2021

- Cierre de implementación, solución de errores y limpieza de código.
- Análisis de rendimiento de aplicaciones.

Mayo 2021

- Versión inicial del informe.

Agosto 2021

- Versión final del informe.