Universidad Zaragoza
1542

Trabajo Fin de Grado

Ingeniería Informática

$\frac{\partial}{\partial t}$ Mitsuba 2

Transporte de luz transitorio en Mitsuba 2

Transient rendering in Mitsuba 2

Autor

Jorge García Pueyo

Director

Adrián Jarabo

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021

$\frac{\partial}{\partial t}$ Mitsuba 2: Transporte de luz transitorio en Mitsuba 2

RESUMEN

Una de las asunciones más establecidas en informática gráfica y visión por computador es considerar que la velocidad de la luz es infinita. La introducción de la *femto-fotografía* (captura de escenas a exposiciones efectivas de picosegundos) inauguró el campo de la *imagen transitoria*, donde se aprovecha la información codificada en el dominio temporal del transporte de luz para resolver problemas como la estimación de profundidad o visión de escenas ocluidas a través de esquinas. Estas técnicas de captura de imagen en estado *transitorio* requieren nuevas herramientas que permitan simular el transporte de luz a escalas comparables con la velocidad de la luz, eliminando las asunción de velocidad de la luz infinita. Uno de los mayores retos de la simulación de luz en estado transitorio es el drástico aumento del tiempo de ejecución de estos algoritmos.

El objetivo de este trabajo consiste en generalizar el software de simulación de transporte *Mitsuba 2* al estado transitorio. Mitsuba 2 es un renderizador estacionario basado en física, que incluye soporte para imagen multiespectral y polarización, así como ejecución vectorizada y en GPU. Generalizarlo a estado transitorio permite desarrollar un renderizador transitorio eficiente, que compense el dramático aumento del tiempo de convergencia de los algoritmos de simulación de luz en estado transitorio. Además, está adaptado para su uso como *render diferenciable*, permitiendo calcular la derivada de una imagen sintética respecto a cualquiera de sus parámetros.

Este proyecto implementa las bases de la simulación de la luz en estado transitorio en Mitsuba 2. Para ello, implementamos el algoritmo de *path tracing* en estado transitorio sobre Mitsuba 2, incluyendo soporte para render espectral y polarizado y su ejecución en CPU de manera vectorizada. Finalmente, se usa el renderizador implementado para analizar efectos de la propagación de la luz solo visibles al considerar la velocidad de la luz finita. Este trabajo sienta las bases para el desarrollo de futuras extensiones a nuevos algoritmos de transporte de luz transitorios, nuevos modelos de interacción luz-materia resueltos en tiempo y mejores algoritmos de reconstrucción en el dominio temporal. Además, su modalidad diferenciable puede ser muy útil para nuevos problemas inversos que hagan uso de la imagen transitoria, como p.ej. ver a través de esquinas.

Índice

Capítulo 1

Introducción y objetivos

La luz, y más en concreto los fotones que forman el campo eléctrico que conocemos como luz, es el elemento más rápido del universo. Sin embargo, y pese a que marca un límite superior en términos de velocidad de propagación, su velocidad es finita. La naturaleza finita de la velocidad de la luz se conoce desde tiempos remotos (Antigua Grecia), y durante los siglos se realizaron numerosos experimentos para medirla (es particularmente célebre el experimento de Galileo Galilei a principios del siglo XVII). No fue hasta 1676 cuando el astrónomo Ole Roemer probó que la luz viaja a una velocidad finita estudiando la duración de los eclipses de las lunas de Júpiter [1]. Hoy día es bien sabido que la velocidad de la luz es $c = 299{,}792{,}458$ m s^{-1}, y la caracterización de la misma es fundamental es astrofísica, electromagenetismo o mecánica cuántica.

La informática gráfica, y en concreto el *render* basado en física, trata de simular el transporte de luz y cómo ésta interactúa con la materia, desarrollando modelos de sensores, emisores y materiales para la simulación. La mayoría de métodos de simulación se basan en cómo la luz es captada por cámaras estándar o el ojo humano, para las cuáles la velocidad de la luz es tan grande que, en comparación con los tiempos de exposición de las cámaras, es considerada infinita. Por esta razón, en el campo de la informática gráfica y la visión por computador, existe la asunción general de que la velocidad de la luz es infinita, de modo que la luz se propaga de manera instantánea en la escena. Este tipo de simulaciones es denominado simulación de transporte de luz en *estado estacionario*.

Imagen transitoria En 2013, Velten *et al.* presentaron el sistema que llamaron *femto-fotografía* [2], en el que combinando láseres pulsados, sensores ultra-rápidos y computación, lograron capturar escenas macróscópicas a exposiciones efectivas de picosegundos (10^{-12} s). A esa resolución temporal, en cada fotograma la luz viaja

alrededor de 0.3 mm, permitiendo por primera vez capturar la propagación de la luz en escenas macróscópicas (ver Figura 1.1). Este sistema inauguró el campo de la *imagen transitoria* (*transient imaging*) [3] que, aprovechando la información codificada en el dominio temporal del transporte de luz, ha permitido resolver problemas clásicos de visión por computador como estimación de profundidad [4], captura de reflectancias [5], separación de componentes de iluminación [6], *bare-sensor imaging* [7], o visión de escenas ocluidas a través de esquinas [8, 9, 10, 11] o a través de medios turbios [12, 13].

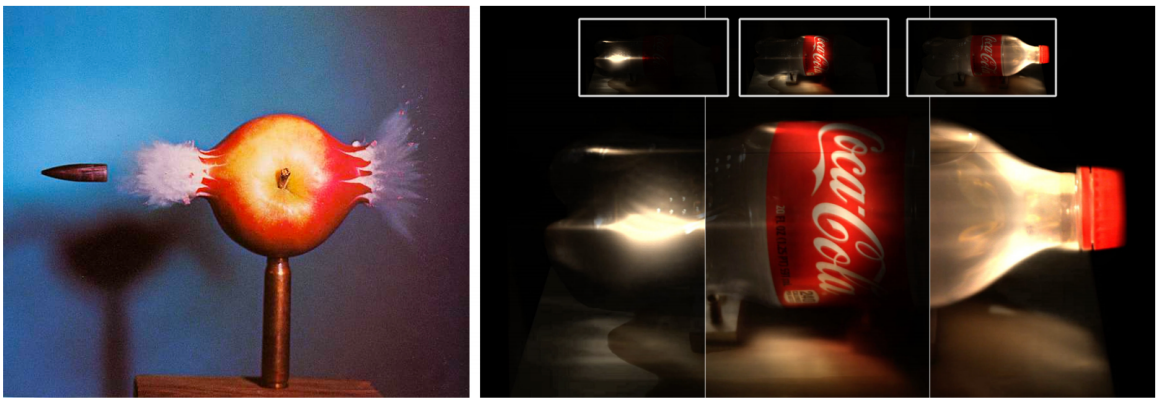


Figura 1.1: Izquierda: En 1964, Harold Edgerton fotografió una bala disparada a 850 m/s a través de una manzana con una exposición de $4 \cdot 10^{-6}$ segundos. **Derecha:** Mediante el uso de iluminación y sensores ultrarápidos, la *femto-fotografía* [2] consigue una exposición efectiva de 10^{-12} segundos, permitiendo capturar la luz en movimiento a través de escenas macroscópicas.

Esta nueva familia de técnicas de captura y reconstrucción de imagen en estado *transitorio* requiere de nuevas herramientas computacionales que permitan simular el transporte de luz a escalas comparables con la velocidad de la luz. Para ello, es necesario eliminar la asunción de la velocidad infinita de la luz, teniendo en cuenta el dominio temporal a la hora de simular el transporte de luz. Este tipo de simulaciones es denominado simulación de luz en *estado transitorio*.

Simulación del transporte de luz El campo de la simulación del transporte de luz basado en física es particularmente maduro, con su teoría datando de principios de los 80 [14]. Así, durante 40 años se han desarrollado numerosos algoritmos de una elevada sofisticación, basados en procesos estocásticos basados en métodos de Monte Carlo [15]. Sin embargo, el campo ha evolucionado de la base del *estado estacionario del transporte de luz* y la inclusión del dominio temporal trae consigo nuevos retos, incluyendo la necesidad de: 1) Nuevos modelos de transporte de luz y de interacción luz-materia; 2) Nuevas técnicas de muestreo y reconstrucción orientadas al dominio temporal; y 3) Nuevas implementaciones que resuelvan los problemas específicos del transporte de luz transitorio.

Objetivo Este proyecto se centra en el tercer punto, y tiene como objetivo desarrollar una implementación de un simulador de transporte de luz transitorio adaptado al dominio de aplicación del mismo (computación científica). Este campo tiene dos requisitos fundamentales: a) ser eficiente y adaptable a su uso en sistemas heterogéneos (CPU, GPU); y b) ser general y preciso en términos de los efectos de transporte de luz incluidos en la simulación.

Para ello, este trabajo desarrolla una extensión del sistema de simulación (renderizador) de transporte de luz ***Mitsuba 2*** **[16] a estado transitorio**. Mitsuba 2 es un renderizador estacionario basado en física (ver Sección 3 para detalles), que incluye soporte para imagen multiespectral y polarización, así como ejecución vectorizada y en GPU. Además, está adaptado para uso dentro de lo que se conoce como *render diferenciable* [17], que permite calcular la derivada de una imagen sintética respecto a cualquiera de sus parámetros, y que es una herramienta fundamental para aplicaciones de render inverso.

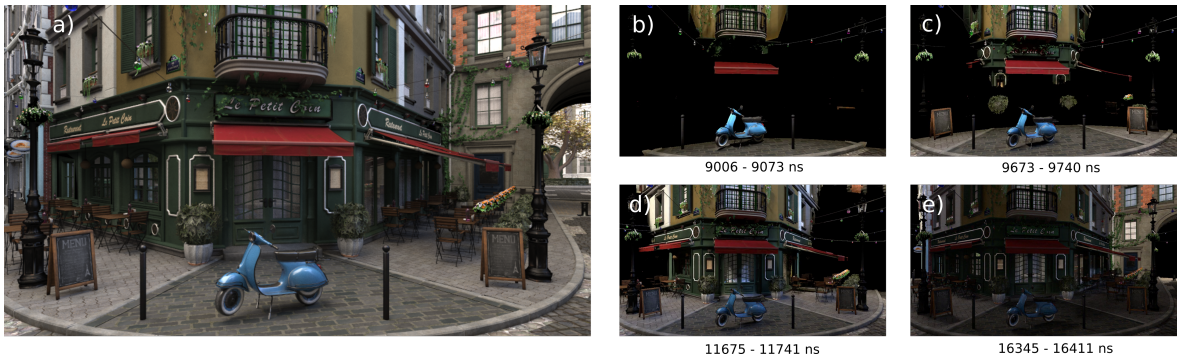


Figura 1.2: La implementación del renderizador transitorio de este trabajo permite visualizar la evolución de la luz en una escena en el dominio temporal, incluso en escenas de alta complejidad. La escena cuenta con una fuente de luz en forma de luz de área que se encuentra en el cielo. La figura muestra el render de la escena en estado estacionario (a) y versiones de la escena en estado transitorio (b-e) en diferentes instantes temporales. Ver la Sección 5.1 para más resultados.

Contribuciones La contribución principal del proyecto es la implementación del algoritmo de *path tracing* (ver Sección 2.1) en estado transitorio, dentro del entorno de Mitsuba 2, así como el desarrollo de los sensores virtuales necesarios para reconstruir de forma eficiente el transporte de luz resuelto en tiempo. Nuestra implementación es eficiente, y permite simular de forma eficiente el transporte de luz transitorio en escenas de elevada complejidad (Figura 1.2). Este trabajo sienta las bases para el desarrollo de futuras extensiones a nuevos algoritmos de transporte de luz transitorios, nuevos modelos de interacción luz-materia resueltos en tiempo y mejores algoritmos de reconstrucción en el dominio temporal. Además, su modalidad diferenciable puede ser muy útil para nuevos problemas inversos que hagan uso de la imagen transitoria,

como p.ej. ver a través de esquinas.

Los vídeos renderizados correspondientes a las escenas que se presentan en esta memoria están disponibles en la siguiente dirección[1]. El código del renderizador transitorio implementado es *open-source* y está disponible en la siguiente dirección [2].

[1] https://drive.google.com/drive/folders/1t5HhctzdceUQbcMNFzw47J6ZdEH9VPJg?usp=sharing
[2] https://github.com/jgarciapueyo/mitsuba2-transient/tree/transient

Capítulo 2

Conocimiento Previo

En este capítulo introducimos brevemente la teoría de transporte de luz sobre la que construye este proyecto. Para ello, distinguimos entre dos modalidades de transporte de luz: estacionario (Sección 2.1), que asume que la velocidad de la luz es infinita, y transitorio (Sección 2.2), que tiene en cuenta la velocidad finita de la luz y el dominio temporal del transporte de luz[1]. Finalmente, describimos brevemente el trabajo relacionado en el campo de *render* transitorio (Sección 2.3).

2.1. Transporte de luz estacionario

El renderizado basado en física [15] tiene como objetivo simular el transporte de luz y la interacción luz-materia a partir de principios físicos sólidos. Su uso es ubicuo en el día a día, en virtualmente todas las áreas que requieran del uso de imágenes realistas, como entretenimiento (películas y videojuegos), arquitectura, diseño de producto, marketing o publicidad [18, 19]. Además de los ámbitos clásicos del uso de informática gráfica, con el incremento en sofisticación de los efectos de transporte de luz simulados, su uso además se ha extendido a ámbitos de computación científica [20, 21], y se ha convertido en una herramienta clave en el desarrollo de técnicas de aprendizaje automático profundo (*deep learning*) permitiendo la generación sintética de datos para aprendizaje.

Los principios físicos utilizados para la simulación del transporte de luz existen en una jerarquía dependiendo las simplificaciones que tienen en cuenta (de más detallado a más simple): óptica cuántica, óptica ondulatoria y óptica geométrica. En general, en informática gráfica la simulación del transporte de luz se limita a la óptica geométrica

[1]Nótese que estos términos son usados para describir la evolución de cualquier sistema. El estado estacionario se define como el estado del sistema en el cual las propiedades del sistema no cambian respecto del tiempo, es decir, el sistema ha convergido ($\frac{\partial propiedades}{\partial t} = 0$). El estado transitorio se define como el estado entre el comienzo y el estado estacionario.

o radiométrica, que es suficiente para la mayoría de aplicaciones.

La radiancia es la unidad básica de energía en óptica geométrica. Dado un punto $\mathbf{x}$ y una dirección ω, la radiancia $L(\mathbf{x}, \omega)$ describe cuánta luz incide en el diferencial de área dA alrededor del punto $\mathbf{x}$, desde el cono de direcciones (ángulo sólido) diferencial $d\Omega$ alrededor de ω. La radiancia se mide en $\mathrm{Wm}^{-2}\mathrm{sr}^{-1}$.

La ecuación de render

La *ecuación de render* [22] describe el transporte de luz en superficies desde un punto de vista radiativo (es decir, asumiendo óptica de rayos). Es la ecuación fundamental dentro del campo de la generación de imágenes sintéticas, derivada de principios físicos de radiometría [15]. Esta ecuación describe la radiancia saliente $L_o(\mathbf{x}, \omega_o)$ de un punto $\mathbf{x}$ en una dirección ω_o como la suma entre la radiancia emitida por el propio objeto $L_e(\mathbf{x}, \omega_o)$ y la radiancia reflejada $L_r(\mathbf{x}, \omega_o)$, donde $L_r(\mathbf{x}, \omega_o)$ se define como la integral de la radiancia incidente $L_i(\mathbf{x}, \omega_i)$ reflejada por el propio objeto (ver Figura 2.1):

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \overbrace{\int_{S^2} L_i(\mathbf{x}, \omega_i) f_r(\mathbf{x}, \omega_o, \omega_i) |\mathbf{n}_\mathbf{x} \cdot \omega_i| d\omega_i}^{L_r(\mathbf{x}, \omega_o)}, \tag{2.1}$$

donde la función $f_r(\mathbf{x}, w_o, w_i)$ es la función de distribución de dispersión bidireccional (*bidirectional scattering distribution function*, o BSDF). La BSDF caracteriza la apariencia del material en $\mathbf{x}$: Intuitivamente, la BSDF explica cómo se dispersa la luz incidente en $\mathbf{x}$ desde la dirección ω_i en la dirección ω_o. El término $|\mathbf{n}_\mathbf{x} \cdot \omega_i|$ modela la atenuación geométrica, con el operador "$\cdot$." el producto escalar entre la dirección incidente ω_i y $\mathbf{n}_\mathbf{x}$ la normal de la superficie en $\mathbf{x}$. El dominio de integración S^2 es la esfera unitaria centrada en $\mathbf{n}_\mathbf{x}$. Nótese que la radiancia incidente $L_i(\mathbf{x}, \omega_i)$ define una relación de recursividad en la Ecuación (2.1), tal que

$$L_i(\mathbf{x}, \omega_i) = L_o(\mathbf{x}_t, \omega_i)\, G(\mathbf{x} \leftrightarrow \mathbf{x}_t) V(\mathbf{x} \leftrightarrow \mathbf{x}_t) \tag{2.2}$$

donde $\mathbf{x}_t = \mathbf{x} - t\,\omega_i$ es un punto en una superficie a distancia t de $\mathbf{x}$, $G(\mathbf{x} \leftrightarrow \mathbf{x}_t) = t^{-2}\,|\mathbf{n}_{\mathbf{x}_t} \cdot \omega_i|$ es la atenuación geométrica entre $\mathbf{x}$ y $\mathbf{x}_t$, y $V(\mathbf{x} \leftrightarrow \mathbf{x}_t)$ es la función de visibilidad entre los mismos.

Una importante asunción es que la luz se propaga con trayectoria rectilínea, siguiendo el Principio de Fermat, por el cual la luz se propaga siguiendo el camino de menor coste. Sin embargo, cambios en la densidad del medio pueden producir trayectorias curvas. No obstante, en la mayoría de escenas esta curvatura es despreciable, salvo en casos como por ejemplo espejismos. Asumir propagación lineal

nos permite computar el término de visibilidad $V(\mathbf{x} \leftrightarrow \mathbf{x}_t)$ utilizando trazado de rayos [23][2].

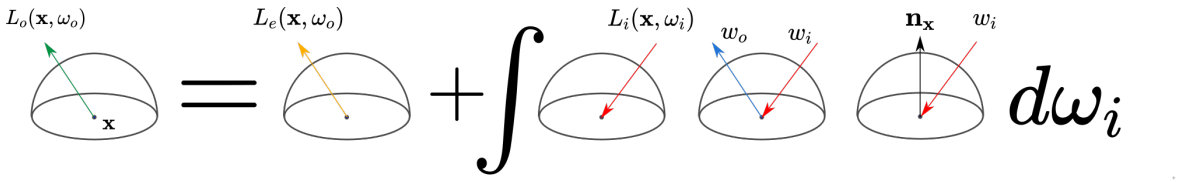


Figura 2.1: Explicación gráfica de la ecuación de render. La radiancia saliente $L_o(\mathbf{x}, \omega_o)$ (rayo verde) es igual a la radiancia emitida por el propio objeto en esa dirección $L_e(\mathbf{x}, \omega_o)$ (rayo amarillo) más la radiancia reflejada. La radiancia reflejada es igual a la integral de toda la radiancia incidente (rayo rojo) reflejada por el propio objeto, modelado a través de la *BSDF*. El dominio de la integral es la semiesfera unitaria centrada alrededor de $\mathbf{n_x}$.

Para calcular la energía I_j capturada por un pixel j de una imagen, se integra la radiancia $L_i(\mathbf{x}_\mathrm{s}, \omega)$ incidente en la superficie del pixel $A_\mathrm{s}(j)$, pesada por el filtro de reconstrucción de la imagen $W_e^{(j)}$ a través de la ecuación de medida (*measurement equation* [24]) como

$$I_j = \int_{A_\mathrm{s}(j)} \int_{S^2} W_e^{(j)}(\mathbf{x}_\mathrm{s}, \omega) L_i(\mathbf{x}_\mathrm{s}, \omega) |\mathbf{n}_{\mathbf{x}_\mathrm{s}} \cdot \omega| d\omega \, dA(\mathbf{x}_\mathrm{s}), \tag{2.3}$$

donde $\mathbf{x}_\mathrm{s}$ es un punto en el sensor virtual, y $W_e^{(j)}(\mathbf{x}_\mathrm{s}, \omega)$ modela la importancia del sensor del pixel j para la radiancia incidente en $\mathbf{x}_\mathrm{s}$ desde la dirección ω.

Métodos de Monte Carlo

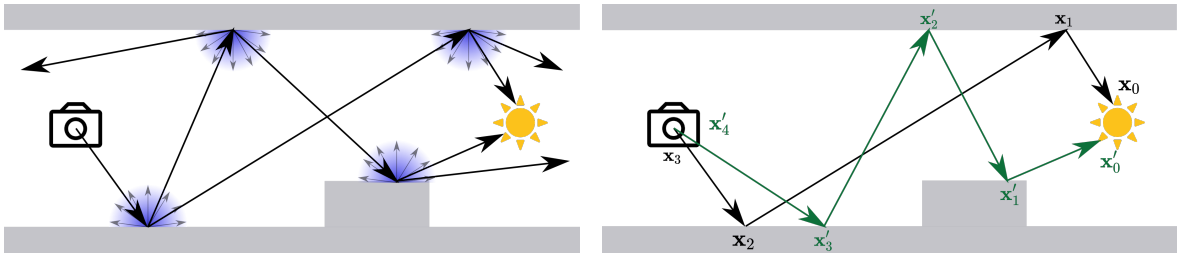
Aunque la ecuación de render (2.1) parece sencilla *a priori*, es imposible resolverla analíticamente para un escenario general por la complejidad derivada de la geometría y apariencia de los materiales de la escena a renderizar. Por esta razón, es necesario utilizar métodos numéricos de integración para resolverla, siendo los más comunes por su generalidad los métodos de Monte Carlo.

Los métodos de Monte Carlo son una familia de algoritmos estocásticos que aproximan el valor esperado $\langle F \rangle$ de un sistema complejo definido por una función d-dimensional $f(x)$ como la media de un conjunto de N evaluaciones aleatorias del sistema. Así, el estimador de Monte Carlo de la ecuación de render es

$$L_o(\mathbf{x}, \omega_o) \approx \langle L_o \rangle = L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{j=1}^{N} \frac{L_i(\mathbf{x}, \omega_j) f_r(\mathbf{x}, \omega_o, \omega_j) |\mathbf{n}_\mathbf{x} \cdot \omega_i|}{p(\omega_j)}, \tag{2.4}$$

[2]Trazado de rayos, o *ray tracing*, calcula la intersección en el punto $\mathbf{x}_t$ de un rayo con la geometría de la escena. Para ello, resuelve el sistema definido mediante igualar la ecuación de un segmento con el modelo implícito de la geometría de la escena.

donde la integral sobre la esfera de direcciones en la Ecuación (2.1) se transforma en la media de N evaluaciones del integrando para las direcciones ω_j muestreadas aleatoriamente con probabilidad definida por la distribución de probabilidad (*probability distribution function*, o pdf) $p(\omega_j)$.



(a) *Distributed ray tracing* **(b)** *Path tracing*

Figura 2.2: Comparación entre los algoritmos *distributed ray tracing* y *path tracing*. En *distributed ray tracing*, en cada intersección se trazan N rayos, lo que implica un crecimiento exponencial. Por claridad, en este diagrama sólo se representamos dos nuevos rayos en cada intersección. En *path tracing*, en cada intersección se traza un solo rayo, creando un camino desde el sensor hasta la fuente de luz. En este diagrama hay dos caminos ($\bar{\mathbf{x}}$ y $\bar{\mathbf{x}}'$).

Path Tracing

La forma más trivial de resolver la recursividad en Ecuación (2.1) es aplicar recursivamente una serie de estimadores de Monte Carlo, de modo que el término $L_i(\mathbf{x}, \omega_i)$ en la Ecuación (2.4) es a su vez una estimación estocástica de la radiancia incidente [14]. Sin embargo, esta aproximación tiene un crecimiento exponencial con el número de intersecciones (ver Figura 2.2). Para resolver este problema, Kajiya [22] introduce el conocido algoritmo de *path tracing*. El algoritmo de *path tracing* consiste en trazar caminos aleatorios (*random walks*) completos desde la cámara hasta que los caminos *mueren* por ser absorbidos, o llegan a una fuente de luz en la escena (ver Figura 2.2b).

Posteriormente, Veach [24] introduce una formulación basado en *path integral*[3], que reformula la recursividad de la ecuación de render como una integral de todos los caminos lumínicos que llegan al sensor, calculando la radiancia I_j en el pixel j como

$$I_j = \int_{\Omega} f^{(j)}(\bar{\mathbf{x}})\, d\mu(\bar{\mathbf{x}}), \tag{2.5}$$

donde Ω es el dominio de todos los caminos posibles en la escena, $\bar{\mathbf{x}} = (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_k) \in \Omega$ es un camino de longitud $k \in [0, \infty)$, donde $\mathbf{x}_j$ es un vértice de dicho camino en el que se produce una interacción luz-materia. El vértice $\mathbf{x}_0$ se encuentra en una fuente de luz y el vértice $\mathbf{x}_k$ se encuentra en el sensor de la cámara. La función $f^{(j)}(\bar{\mathbf{x}})$ es la

[3]No confundir con la formulación basado en *path integrals* de la mecánica cuántica en la que se integra sobre todos los posibles estados de un sistema cuántico.

contribución de un camino $\bar{\mathbf{x}}$, definida como

$$f^{(j)}(\bar{\mathbf{x}}) = L_e(\mathbf{x}_0 \to \mathbf{x}_1)\mathfrak{T}(\bar{\mathbf{x}})W_e^{(j)}(\mathbf{x}_{k-1} \to \mathbf{x}_k), \tag{2.6}$$

con $L_e(\mathbf{x}_0 \to \mathbf{x}_1)$ la radiancia emitida desde $\mathbf{x}_0$ en la fuente de luz hacia $\mathbf{x}_1$, $W_e^{(j)}(\mathbf{x}_{k-1} \to \mathbf{x}_k)$ es la importancia del sensor en $\mathbf{x}_k$ (ver Ecuación 2.3), y $\mathfrak{T}(\bar{\mathbf{x}})$ es el *path throughput*, que modela la pérdida de energía debido a las $k-1$ interacciones luz-materia en los vértices internos del camino $\bar{\mathbf{x}}$, y se define como

$$\mathfrak{T}(\bar{\mathbf{x}}) = \prod_{i=1}^{k-1} f_r(\mathbf{x}_{i-1} \to \mathbf{x}_i \to \mathbf{x}_{i+1}) \prod_{i=0}^{k-1} G(x_i \to x_{i+1})V(x_i \to x_{i+1}), \tag{2.7}$$

con $G(x_i \to x_{i+1})$ y $V(x_i \to x_{i+1})$ los términos geométricos y de visibilidad, respectivamente (ver Ecuación (2.2)).

La *path integral*, de manera similar a la Ecuación de render, debe ser evaluada numéricamente, usando el estimador de Monte Carlo sobre N caminos

$$\langle I_j \rangle = \frac{1}{N} \sum_{j=1}^{N} \frac{f^{(j)}(\bar{\mathbf{x}})}{p(\bar{\mathbf{x}})}. \tag{2.8}$$

Path tracing es fácilmente implementable como un algoritmo iterativo, y es extremadamente paralelizable, ya que cada camino es una muestra independiente de los otros. El Anexo A presenta el código de la implementación básica del algoritmo de *path tracing* en C++.

El algoritmo de *path tracing* y la *paht integral* forman la base de los algoritmos de simulación del transporte de luz por su simplicidad y habilidad para producir imágenes realistas. También serán la base de este trabajo para implementar la simulación del transporte de luz transitorio.

2.2. Transporte de luz transitorio

En la sección anterior, se ha explicado el transporte de luz estacionario a través de la ecuación de render. El transporte de luz en estado estacionario se refiere a considerar que la velocidad de la luz es infinita y, por tanto, la distribución de la radiancia de la escena no depende de la dimensión temporal, sino que solo depende de la dimensión espacial ($\partial L/\partial t = 0$). Esta es una asunción razonable teniendo en cuenta que la mayoría de sensores convencionales (desde las cámaras fotográficas hasta nuestros ojos) son muy lentos el comparación con la velocidad de la luz.

Este trabajo se basa en el transporte de luz en estado transitorio, donde se deja de asumir que la velocidad de la luz es infinita.

Pese a que la simulación de transporte de luz en estado transitorio tiene una larga trayectoria en electromagnetismo computacional, hasta recientemente no fue explorado en los campos de gráficos y visión por computador. En este trabajo, se utiliza el marco de trabajo de Jarabo *et al.* [25], que establece una base teórica para el transporte de luz en estado *transitorio* dentro de un marco de simulación usando métodos de Monte Carlo, extendiendo la *path integral* en el dominio temporal. Así, el valor del pixel j se calcula como una integral sobre el espacio de caminos Ω y sobre el espacio de retrasos temporales ΔT

$$I_j = \int_{\Omega} \int_{\Delta T} f^{(j)}(\bar{\mathbf{x}}, \bar{\Delta} t) d\mu(\bar{\Delta} t) d\mu(\bar{\mathbf{x}}), \tag{2.9}$$

donde $\bar{\Delta} t = \Delta t_0, ..., \Delta t_k$ define los retrasos temporales asociados a cada vértice del camino $\bar{\mathbf{x}}$. El principal cambio respecto a la Ecuación (2.6) es la inclusión de la componente temporal en la contribución del camino $f(\bar{\mathbf{x}}, \bar{\Delta} t)$, de modo que la emisión L_e, la importancia en el sensor $W_e^{(j)}$, y el *path throughput* $\mathfrak{T}$ ahora dependen del tiempo:

$$f^{(j)}(\bar{\mathbf{x}}, \bar{\Delta} t) = L_e(\mathbf{x}_0 \to \mathbf{x}_1, \Delta t_0) \mathfrak{T}(\bar{\mathbf{x}}, \bar{\Delta} t) W_e^{(j)}(\mathbf{x}_{k-1} \to \mathbf{x}_k, \Delta t_k). \tag{2.10}$$

De este modo, el *path throughput* transitorio se define como su análogo estacionario (2.7), extendiéndolo al dominio temporal como

$$\mathfrak{T}(\bar{\mathbf{x}}, \bar{\Delta} t) = \prod_{i=1}^{k-1} f_r(\mathbf{x}_{i-1} \to \mathbf{x}_i \to \mathbf{x}_{i+1}, \Delta t_i) \prod_{i=0}^{k-1} G(x_i \to x_{i+1}) V(x_i \to x_{i+1}). \tag{2.11}$$

Dado que asumimos que la geometría es estacionaria, ya que comparado con la velocidad de la luz el movimiento de elementos macroscópicos es casi nulo, el término geométrico y la relación de visibilidad solo dependen de las coordenadas espaciales. No obstante, la *BSDF* introduce un retraso temporal parametrizado por Δt_i, debido a los potenciales retrasos temporales durante el evento de dispersión de la luz en cada vértice $\mathbf{x}_i$.

Un camino transitorio de luz se define por las coordenadas espaciales y las coordenadas temporales de cada vértice. Las coordenadas temporales en cada vértice $\mathbf{x}_i$ son t_i^-, que es el tiempo justo antes de que se produzca el evento de dispersión de la luz. El tiempo total del camino t_k se calcula sumando los tipos de retrasos temporales debidos a la separación espacial de los vértices denominados *retrasos de propagación* $t(\mathbf{x}_i \to \mathbf{x}_{i+1})$ y los debidos a los eventos de dispersión en los vértices del camino (p. ej. efectos de dispersión inelásticos de un material fluorescente que tarda en reemitir la

luz un tiempo) denominados *retrasos de dispersión* Δt_i (Ver Figura 2.3). Los primeros están directamente relacionados con la distancia entre vértices, y se calculan

$$t(\mathbf{x}_i \rightarrow \mathbf{x}_{i+1}) = \frac{\eta \left| \mathbf{x}_i - \mathbf{x}_{i+1} \right|}{c}, \tag{2.12}$$

con $c = 299,792,458$ m s^{-1} la velocidad de la luz en el vacío, y η el ratio entre la velocidad de la luz en el medio y c (el índice de refracción del medio). Por ejemplo, la velocidad de la luz en el agua es 224,844,000 m s^{-1}, y su índice de refracción es $\eta = 1{,}3$.

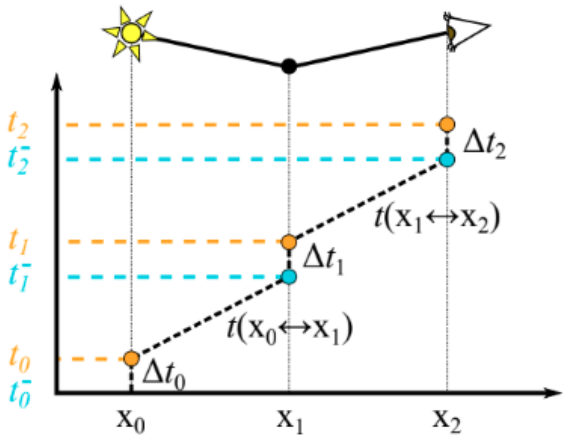


Figura 2.3: Diagrama espacio-temporal de un camino de luz transitorio. La luz es emitida desde el vértice $\mathbf{x}_0$ en el instante t_0^- y llega al vértice $\mathbf{x}_1$ en el tiempo $\Delta t_0 + t(\mathbf{x}_0 \rightarrow \mathbf{x}_1)$. Fuente: Jarabo *et al.* [25].

La introducción del dominio temporal trae nuevos retos como el incremento del tiempo de convergencia de extensiones directas de los algoritmos estacionarios al dominio temporal. La razón es que es casi imposible muestrear un camino transitorio cuya duración t_k esté dentro del rango de tiempos asociados a la ventana temporal que se está muestreando en cada momento. Para solventar estos nuevos retos, se trabaja en dos áreas; por un lado nuevas estrategias de muestreo para mejorar la distribución temporal de las muestras haciendo uso de *path reuse*, por otro lado nuevas técnicas de estimación que permitan reconstruir la señal con mayor calidad a partir de las muestras originales.

La dificultad de mejorar la distribución temporal de las muestras se debe a que es difícil estimar la duración del camino porque aunque es fácil muestrear los retrasos temporales de dispersión, mientras las posiciones espaciales sean determinadas por técnicas estacionarias, no se podrá controlar los retrasos temporales de propagación y, en consecuencia, el tiempo total del camino.

2.3. Simulación del transporte de luz transitorio: Trabajo Relacionado

El término *transient rendering*, como se ha comentado en la Sección 2.2 , fue introducido por Smith *et al.* [26], quien propuso una extensión de la ecuación de render añadiendo el retraso temporal debido a la propagación de la luz. A partir de este trabajo, otros autores han propuesto otras variaciones transitorias a las técnicas de renderizado basadas en Monte Carlo [27, 28, 29, 30, 31, 32].

Entre ellas se encuentra la base de este trabajo, en el cual Jarabo *et al.* [25] presentó un marco de trabajo práctico introduciendo la *transient path integral*, además de una técnica de reconstrucción eficiente y una nueva estrategia de muestreo para controlar la longitud de los caminos en medios participativos. Este trabajo fue extendido para incluir efectos como polarización o fluorescencia [33].

Pediredla *et al.* [34], introdujo conexiones elipsoidales para renderizado transitorio, usando vértices adicionales que controlan la longitud del camino en superfices en vez de medios particpativos. Otros autores han centrado su trabajo en aumentar la eficiencia al simular el transporte de luz transitorio usando modelos más simples [35, 36, 37, 38] o aplicando estimación de densidad en medios participativos [39].

Todos estos trabajos se centran en generar imágenes simulando el transporte de la luz en una escena. El área de renderizado diferenciable [17] añade las derivadas a la simulación del transporte de luz. Puede ser usado para problemas de renderizado inverso donde se pretende estimar los atributos de la escena (geometría, luz, materiales) dada la imagen final de la simulación. Este nuevo enfoque puede ser también aplicado a simulación de transporte de luz transitorio, creando algoritmos de renderizado diferenciables en estado transitorio.

Capítulo 3

Mitsuba 2

Mitsuba 2 [16] es un sistema de renderizado basado en física orientado a investigación escrito en C++17. El objetivo de Mitsuba 2 es la versatilidad, permitiendo la construcción de renderizadores orientado a distintos dominios de aplicación, centrados en aplicaciones de carácter científico. Para ello ofrece:

- Diferentes niveles de **precisión en el transporte de luz**, incluyendo desde radiancia monocromática hasta multiespectral, o estado de polarización de la luz utilizando la formulación de Stokes [33].
- Diferentes **modos de renderizado**, incluyendo el modo clásico para problemas de simulación de apariencia, o renderizado de las derivativas de la escena para su uso en problemas inversos de optimización.
- Diferentes **modos de compilación**, dependiendo del sistema de ejecución objetivo, incluyendo código altamente vectorizado (CPU) o masivamente paralelo (GPU).

Además, Mitsuba 2 presenta un notable soporte por parte de la comunidad, incluyendo plugins para el software de modelado 3D *Blender* que permite definir fácilmente las escenas a renderizar, así como una excelente conectividad con Python para su uso mediante *scripting*.

3.1. ¿Por qué Mitsuba 2?

Para la implementación de un renderizador transitorio, extenderemos un renderizador ya existente porque la creación de un nuevo renderizador se ha descartado por la complejidad y gran inversión de tiempo requerida. Uno de los mayores desafíos de la simulación del transporte de luz transitorio es la convergencia más lenta de los resultados por el nuevo dominio temporal a muestrear. De esta forma,

el criterio más importante para la elección será la eficiencia del renderizador, que debe ser ejecutable en CPU de manera vectorizada y en GPU. Además, debe poder simular escenas complejas, tanto en términos geométricos como de apariencia. Se tendrá en cuenta características adicionales del transporte de luz como la renderización espectral, la polarización, o render diferenciable.

Existen multitud de sistemas de renderizado *open source*, cada uno de ellos implementado con diferentes objetivos. Desde el punto de vista de este trabajo, donde se pretende simular el transporte de luz transitorio, se puede dividir los sistemas de renderización en dos tipos: sistemas de renderizado genéricos y sistemas de renderización de especializados (ad-hoc).

Los sistemas de renderizado genéricos tienen como objetivo la facilidad de extensión incluyendo por defecto una serie de luces, sensores y modelos de materiales reutilizables, así como uno o varios algoritmos de transporte de luz, siendo *path tracing* el algoritmo estándar. Entre los sistemas de renderización genéricos considerados como base del proyecto, aparte de Mitsuba 2, destacan:

- **PBRT v4** [40]: Asociado a la cuarta versión del libro *Physical Based Rendering* [15], es un renderizador académico, con una estructura similar a Mitsuba 2. Usa clases abstractas para definir la interfaz de los componentes del renderizador más importantes, mientras que las funcionalidades son implementada por instancias concretas de esos componentes. Soporta renderizado espectral, así como una gran variedad de *BSDF*, materiales y métodos de muestreo. Se puede ejecutar en CPU y GPU.

- **Nvidia Falcor** [41]: Orientado a investigación, es un renderizador en tiempo real en GPU, que implementa *path tracing* en forma de fases (*render phases*) escritos usando *Slang* [42] (un lenguaje de *shading* para GPU).

Por otro lado, nos encontramos sistemas de renderizado implementados para usos específicos, normalmente relacionado con un nuevo avance experimental en la simulación del transporte de luz. Entre ellos, podemos destacar:

- **dtrt** [43]: Renderizador diferenciable que calcula la derivada de una imagen respecto de los parámetros de la escena (p.ej. la geometría o los materiales en la escena). No soporta renderización espectral, tiene una variedad limitada de materiales, fuentes de luz y cámaras, y está implementado en CPU.

- **bunny-killer** [25]: Renderizador transitorio en el que se basa este trabajo. Soporta renderización espectral y polarización. Solo se puede ejecutar en CPU. En

la idea inicial, se consideró modificar la implementación original para su ejecución en GPU.

	Mitsuba 2	**PBRTv4**	**Falcor**	**dtrt**	**bunny-killer**
Transitorio					✓
Espectral	✓	✓			✓
Polarización	✓				✓
Diferenciable	✓			✓	
Materiales Complejos	✓	✓	slang		
Extensibilidad	Fácil	Fácil	Fácil	Difícil	Medio
Plataforma	CPU vectorizado + GPU	CPU + GPU	GPU	CPU	CPU

Tabla 3.1: Comparación de las características de cada sistema de renderizado considerado como base del nuevo renderizador transitorio.

Comparación y justificación En la Tabla 3.1 comparamos las características de todos los sistemas considerados en este trabajo. En ella podemos observar que Mitsuba 2 es el renderizador más completo de los cinco, haciéndolo el candidato más adecuado para su extensión al dominio transitorio.

En primer lugar, Mitsuba 2, permite ser usado en problemas de renderización espectral, polarización y render diferenciable. Esta última característica, render diferenciable, es muy interesante para aplicarla a problemas de *renderizado* inverso como se ha comentado en la Sección 2.3. El renderizador dtrt también es diferenciable, pero está implementado con ese objetivo en concreto y falla en las otras aplicaciones del transporte de luz, el tratamiento de escenas con materiales complejos y la plataforma de ejecución (sólo CPU).

En segundo lugar, Mitsuba 2 se puede ejecutar tanto en CPU, CPU vectorizado y GPU. Nvidia Falcor también tiene la opción de ejecución en GPU, pero se descartó por su falta de aplicación para simular características mas complejas del transporte de luz ya que está orientado a aplicaciones de renderizado en tiempo real. PBRTv4 también cuenta con la opción de ejecución en GPU, pero está orientado a ser un sistema de renderizado educacional, por lo que no cuenta con características más complejas como polarización o renderizado diferenciable. La ejecución en CPU de manera vectorizada y en GPU es una gran ventaja porque a medida que aumenta la complejidad de los

algoritmos transitorios, también aumentan los tiempos de renderizado que pueden llegar a ser prohibitivos.

En tercer lugar, y de manera análoga a todos los renderizadores genéricos, su diseño en forma de interfaces y plugins hace que sea fácilmente extensible. Bunny-Killer tiene una arquitectura similar, aunque no igual a Mitsuba 2, que hace que su extensión a otras aplicaciones tenga una dificultad media. Sin embargo, la extensión para su ejecución en GPU hubiera requerido un cambio importante en la arquitectura.

3.2. Diseño interno de Mitsuba 2

Mitsuba 2 consigue ser aplicable a distintos dominios de aplicación y plataformas a través de lo que llaman *variantes*. En tiempo de compilación, Mitsuba 2 transforma la aritmética, estructuras de datos y flujo de ejecución dependiendo de las variantes elegidas. Por ejemplo, la variante más básica es un renderizador escalar, ejecutable en CPU y que representa la luz con los valores RGB. Sin embargo, existen variantes que permiten ejecutar Mitsuba 2 en GPU y donde toda la renderización es diferenciable (es decir, en lugar de calcular radiancia, calculan su derivada respecto de algún parámetro en la escena).

Para ser aplicable a distintos dominios y poder transformar las estructuras de datos subyacentes, todo el sistema de renderización Mitsuba 2 aprovecha la (meta)programación genérica. Más específicamente, cada variante debe describir cuál es la representación de la radiancia (parámetro `Spectrum`) y la representación numérica usada en la simulación (parámetro `Float`). Éstos son tipos los básicos de un renderizador, a partir de los cuáles se crean los otros tipos (puntos, vectores, rayos, matrices, etc.).

Para una compilación y ejecución en plataformas diferentes (CPU, CPU vectorizado y/o GPU) transparente al compilador, Mitsuba 2 construye sus estructuras de

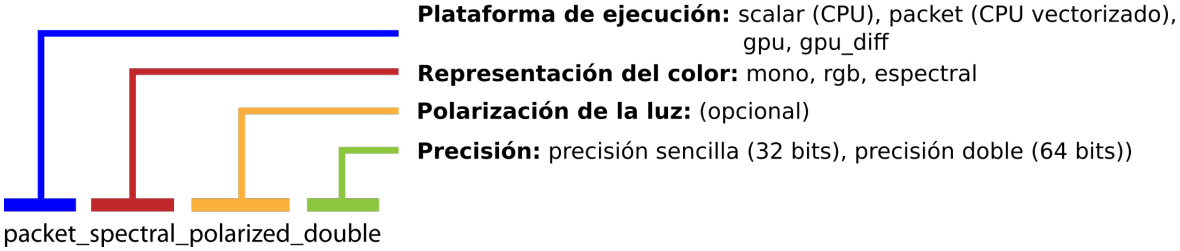


Figura 3.1: Diagrama con las opciones disponibles para cada variante de Mitsuba 2. Ejemplo de una variante que se ejecuta de manera vectorizada sobre CPU representando la luz de forma espectral y polarizada y con precisión doble. Fuente: Mitsuba 2 [16]

datos sobre la biblioteca genérica de programación vectorial **Enoki**, que describimos brevemente en las siguientes líneas. El uso de Enoki permite implementar una sóla vez un algoritmo, y que automáticamente se pueda compilar en múltiples plataformas de muy distintas características, sin necesidad de añadir código adicional. Sin embargo, requiere de una implementación más cuidadosa que en el caso de código escalar normal.

3.2.1. Enoki

Enoki [44] es una librería de C++ que habilita la transformación automática de código para una plataforma objetivo específica, incluyendo vectorización explícita en paquetes, o paralelismo masivo. También permite el cálculo de gradientes de forma transparente para una versión del algoritmo que contenga diferenciación automática. Enoki transforma el código del algoritmo a instrucciones SIMD (*Single Instruction Multiple Data*) de manera eficiente soportando arquitecturas como Intel (AVX512, AVX2, AVX y SSE4.2) o NVIDIA CUDA a traves de un compilador *just in time* (JIT).

La pieza fundamental de Enoki es `enoki::Array`, un contenedor genérico con un tamaño fijo que puede ser compilado para ser ejecutado en cualquiera de las plataformas presentadas anteriormente. La diferencia con la clase de la librería estándar de C++ `std::array` es que Enoki envía todas las operaciones a los elementos del contenedor (ver Figura 3.2). De esta manera, utilizando Enoki y definiendo de manera genérica el tipo de datos de los algoritmos (de forma similar a como se define la representación del color y la representación numérica en Mitsuba 2), podemos vectorizar cualquier algoritmo como en el ejemplo de la Figura 3.3).

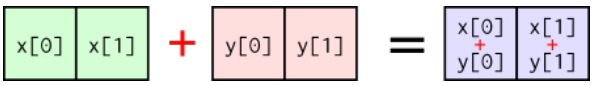


Figura 3.2: Ejemplo del envío de las operaciones a los elementos de *enoki::Array* para vectorizar la operación de suma.

Las comparaciones con tipos de datos de Enoki son aplicadas, en general, elemento a elemento y producen una máscara que representa el resultado de la comparación. Las máscaras habilitan el desarrollo de un algoritmo sin declaraciones del tipo `if` como se puede ver en el ejemplo 3.3b donde el primér parámetro de la función `enoki::select(.., .., ..)` es una comparación (`x <= 0.0031308f`) que produce una máscara.

Además, Enoki ofrece listas dinámicas (*enoki::DynamicArray*) en CPU y la opción de transformar listas de estructuras de datos (*Array of Structures*) en estructuras de

datos de listas (*Structure of Arrays*) con un mejor disposición para su vectorización.

```
float srgb_gamma(float x) {
    if (x <= 0.0031308f)
        return x * 12.92f;
    else
        return std::pow(x * 1.055f, 1.f / 2.4f) - 0.055f;
}

float input = 2.5;
float output = srgb_gamma(input);
```

(a) Versión del algoritmo no vectorizado. Se trata un solo valor en cada llamada a la función.

```
template <typename Value> Value srgb_gamma(Value x) {
    return enoki::select(
        x <= 0.0031308f,
        x * 12.92f,
        enoki::pow(x * 1.055f, 1.f / 2.4f) - 0.055f
    );
}

using Color3f = enoki::Array<float, 3>;
Color3f input(2.5, 3.4, 6.3);
Color3f output = srgb_gamma(input);
```

(b) Versión del algoritmo vectorizado. El tipo de los parámetros de entrada y salida son genéricos. En este ejemplo, se llama a la función con 3 valores que representan los valores RGB.

Figura 3.3: Ejemplo de vectorización de un algoritmo que calcula el valor SRGB aplicando una función gamma.

3.2.2. Arquitectura de plugins

Mitsuba 2 es un sistema de renderizado modular basado en plugins, haciendo que su extensión para ser aplicado a otros dominios de aplicación sea sencillo. Para ello, Mitsuba 2 está dividido 3 librerías básicas de apoyo para la creación de nuevos renderizadores: `libcore` implementa funcionalidades relacionadas con I/O, estructuras de datos básicas y carga de plugins, `librender` contiene abstracciones de los componentes necesarios para representar una escena y su renderizado y `libpython` que contiene los elementos básicos para crear *bindings* de Mitsuba 2 en Python.

Los plugins son clases concretas que implementan la funcionalidad siguiendo la interfaz definida por las clases abstractas en la librería `librender`. Por ejemplo, dada la interfaz general de una fuente de luz, Mitsuba 2 contiene plugins para luces puntuales, luces de área o luces de entorno (ver Figura 3.4). Los plugins son compilados como librerías dinámicas que son cargadas por el programa principal según los componentes de la escena a renderizar.

Los componentes básicos de un renderizador, y por lo tanto de Mitsuba 2, son los siguientes (junto al componente se nombra la clase abstracta correspondiente en Mitsuba 2):

- Sensores (`Sensor`): encargados, junto con las películas (`Film`), de procesar las medidas de radiancia de la escena para un píxel concreto y almacenarla.
- Emisores (`Emitter`): son las fuentes de luz. Existen de varios tipos: los emisores que se localizan en la escena (luz puntual), los emisores que se asignan a una geometría para convertirla en una fuente de luz (luz de área) y los emisores que rodean la escena a una distancia infinita (emisor del entorno).
- Geometrías (`Shape`): definen las superficies que marcan la separación entre diferentes materiales, por ejemplo una geometría podría describir la separación entre el aire y un objeto sólido como una roca.
- *BSDF* (`Shape`): modelos de dispersión de la luz que describen la manera en la que la luz interacciona con las geometrías de la escena.
- Integradores (`Integrator`): nombre que reciben las diferentes técnicas de renderizado y que calculan el resultado de la integral. Cada tipo de integrador representa una forma de resolver la ecuación de transporte de luz. Por ejemplo, el integrador de *path tracing*.

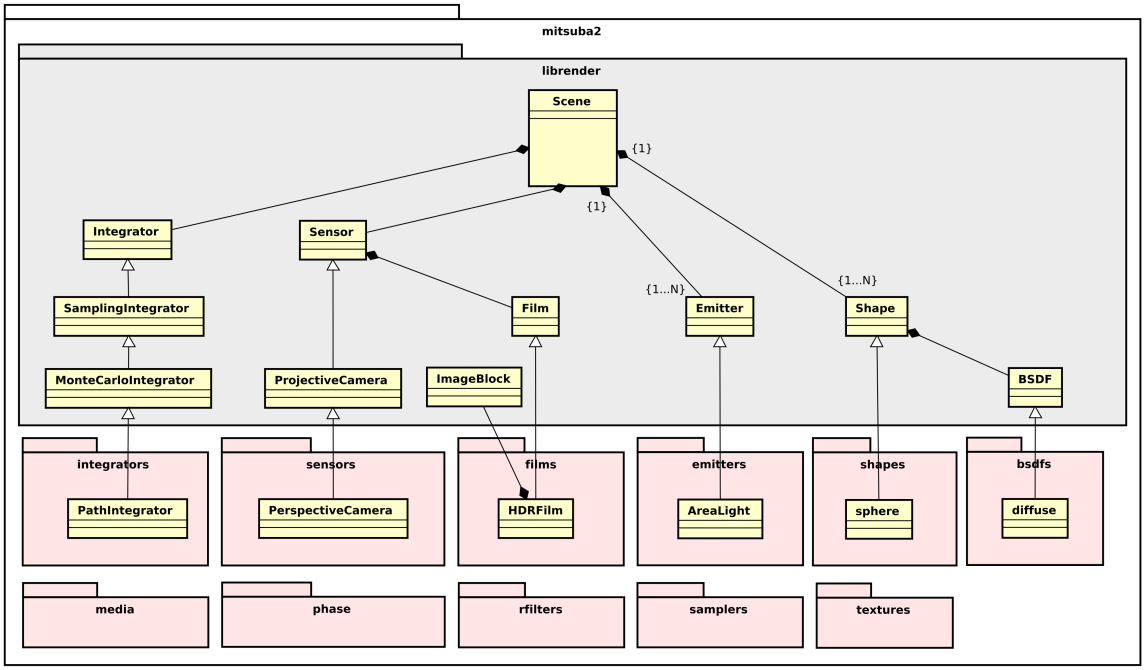


Figura 3.4: Diagrama de clases de Mitsuba 2 simplificado. El módulo `librender` contiene interfaces de los componentes de un sistema de renderizado. Los plugins se encuentran dentro de los módulos rosas como clases concretas que implementan la funcionalidad siguiendo la interfaz de las clases abstractas definidas en el módulo `librender`. Existen varios plugins de cada clase abstracta, pero se ha incluido un solo ejemplo para lós módulos más importantes por claridad. Se ha omitido los módulos `libcore` y `libpython` por su uso ubicuo en el resto del proyecto.

Capítulo 4

$\frac{\partial}{\partial t}$ Mitsuba 2: Implementación de un renderizador transitorio

En este capítulo presentamos la implementación de un renderizador transitorio en Mitsuba 2. En primer lugar, definimos las funcionalidades que debe tener un renderizador, de forma general, para poder soportar la simulación del transporte de luz transitorio (Sección 4.1) y explicamos el algoritmo de *path tracing* en estado transitorio (Sección 4.2). En segundo lugar, describimos cómo se han implementado el transporte de luz transitorio en Mitsuba 2 (Sección 4.3).

4.1. Funcionalidades de un renderizador transitorio

El objetivo de un renderizador transitorio es simular la radiancia en la escena a lo largo de la dimensión temporal. Siguiendo el marco teórico presentado en la Sección 2.2 donde se describe la integral de caminos transitoria (*transient path integral*), en un renderizador transitorio todos los componentes del renderizador deben tener en cuenta la dimensión temporal. Entre estos componentes destacan los sensores, las luces, los materiales y los integradores.

El integrador transitorio debe asociar a la radiancia calculada por la función de contribución de un camino transitorio $\bar{\mathbf{x}}$ (ecuación 2.10) el instante temporal correspondiente al tiempo total del camino t_k. Los sensores también pueden definir una sensibilidad temporal, normalmente un intervalo finito de tiempo (análogo al tiempo de exposición en fotografía) y las luces pueden variar su emisión a lo largo del tiempo. Además, ciertos materiales que modelen efectos más complejos del transporte de luz como la fluorescencia pueden introducir retrasos temporales (*scattering delays*) cuando la luz rebote en dichos materiales, aumentando el tiempo de ese camino. Finalmente, hay que considerar el tiempo de propagación, para lo cual debemos tener en cuenta que la velocidad de la luz es diferente dependiendo del medio en el que se encuentra. Para un cálculo físicamente correcto del tiempo total del camino, el renderizador transitorio

debe guardar el índice de refracción del medio en el que se encuentra el camino en cada momento.

4.2. Transient Path Tracing

El algoritmo de *path tracing* consiste en trazar caminos completos desde la cámara hasta las fuentes de luz. Para ello, cada vez que un rayo incidente intersecta una superficie, produciendo una interacción, un solo rayo saliente es muestreado, hasta intersectar una fuente de luz. Para elevar el algoritmo de *path tracing* al estado transitorio, es necesario modificarlo ligeramente. A lo largo de la creación del camino, se debe almacenar el tiempo del camino. Para ello, se calcula el tiempo de propagación entre vértices $t(\mathbf{x}_i \rightarrow \mathbf{x}_{i+1})$, teniendo en cuenta el índice de refracción del medio en el que se encuentran estos dos vértices, ya que el tiempo depende de la velocidad a la que viaja la luz en ese medio. Además, al evaluar la *BSDF* del material de la intersección, hay que comprobar si introduce un retraso de dispersión (*scattering delay*).

La Figura 4.1 muestra el código en C++ de una versión básica del algoritmo de *path tracing* transitorio que crea un camino de luz iterativamente y devuelve la radiancia y el tiempo asociado a dicho camino.

4.3. Diseño interno de Mitsuba 2 Transitorio

4.3.1. Decisiones de diseño

Extender Mitsuba 2 al estado transitorio requiere la creación y/o extensión de algunos componentes. Previamente, se tomaron una serie de decisiones de diseño que han guiado la implementación de estos nuevos componentes.

- Mitsuba 2 debe ser compatible con otras extensiones tras la modificación para su aplicación al transporte de luz en estado transitorio. La implementación de los nuevos componentes se hará de forma similar a la actual, donde se describe la interfaz del nuevo componente en el módulo `librender` y se implementa una clase concreta con la funcionalidad en forma de plugin, intentando modificar los componentes ya existentes lo menos posible.

- Se evaluarán varias ventanas temporales a la vez asignando la radiancia de un camino a la ventana temporal que le corresponda según el tiempo de dicho camino. Esto es similar a la técnica de reuso de caminos (*path reuse*) en la dimensión espacial que existe en los algoritmos de *photon mapping* o *light tracing*. De esta

```
std::tuple<Radiance, Time>
createPath(Scene &scene, Ray original_ray) {
    // Radiance is discretized to 3 values (RGB)
    Radiance throughput = [1.f, 1.f, 1.f];
    Radiance result = [0.f, 0.f, 0.f];
    Ray ray = original_ray;

    // Variables for Transient Rendering
    Time time = 0.f;
    float ior = scene.defaultIOR();

    // First Intersection
    SurfaceInteraction si = scene.intersect(ray);

    while(true) {
        // Add propagation time using the distance between the
        // origin of the ray and the intersection. Correct that
        // time with the ior of the medium
        time += si.time_of_flight(ray) * ior;

        // If intersection is emitter, evaluate the outgoing
        // radiance from the emitter and finish path
        if(si.material.type == EMITTER) {
            result = throughput * si.material.emitter->eval(ray);
            break;
        }

        // If length of path is greater than maximum length
        // allowed, finish path
        if(length > max_length)
           break;

        // Intersection is a surface with a material not emissive.
        // Sample new outgoing ray and evaluate interaction (BSDF).
        [outgoing_ray, bsdf_val] = si.material.bsdf->sample(ray);
        // Scale the throughput by the BSDF of the current
        // intersection
        throughput *= bsdf_val.value;
        // Add scattering delay due to the properties of the material
        // of the intersection
        time += bsdf_val.scattering_delay;
        // Update IOR with the one of the medium where the outgoing
        // ray is
        ior = bsdf_val.ior_medium_outgoing_ray;

        // Continue intersecting other surfaces creating
        // the path until reaching an emitter
        ray = outgoing_ray;
        si = scene.intersect(ray);
    }
    return {result, time};
}
```

Figura 4.1: Código C++ de una versión simplificada del algoritmo de *path tracing* transitorio.

forma, se almacenará la radiancia en la escena a lo largo del tiempo en una estructura de datos que permita reconstruir la evolución de la radiancia en 3 dimensiones: dos dimensiones espaciales (x-y) y la dimension temporal (t). Si se evaluase una sola ventana temporal, aquellos caminos cuyo tiempo no estuviese dentro de esa ventana temporal serían desechados y para ver la evolución de la radiancia en el tiempo sería necesario ejecutar el renderizador varias veces con ventanas temporales contiguas.

- Se asume que el retraso introducido por la dispersión es marginal, es decir, $\Delta t_i = 0$ en todas las interacciones. La fuentes de luz producirán un pulso de luz delta al comienzo ($t = 0$).

- Asumimos que la importancia del sensor en el dominio temporal está definida como un filtro de caja en el intervalo del frame.

- Se utilizará la técnica de estimación de densidad del histograma para la dimensión temporal, donde cada barra del histograma representa una ventana temporal y la anchura de la barra representa la exposición de la ventana temporal.

Las decisiones de diseño van orientadas a mostrar la implementación un renderizador transitorio funcional y sencillo sobre Mitsuba 2; extensible a futuras mejoras más complejas como materiales que introduzcan retrasos de dispersión, mejores técnicas de estimación de densidad temporal o fuentes de luz y sensores con otros perfiles temporales.

4.3.2. Decisiones de implementación

El diagrama de clases de Mitsuba 2 visto en el capítulo anterior (ver Figura 3.4) muestra la estructura de clases y, también, la estructura del fichero que describe la escena a renderizar. La escena es el elemento raíz y contiene un integrador, un sensor con una película, una serie de emisores y una serie de geometrías.

Para mantener compatibilidad con las escenas ya existentes y la extensibilidad de Mitsuba 2, el integrador transitorio (*Transient Integrator*) y todas sus subclases heredan de la clase base `Integrator` siguiendo un esquema similar a los integradores existentes en Mitsuba 2 (ver Figura 4.2). Las películas de ráfaga (*Streak Film*), que almacenan la radiancia en varias ventanas temporales, heredan de la clase base `Film` por el mismo motivo.

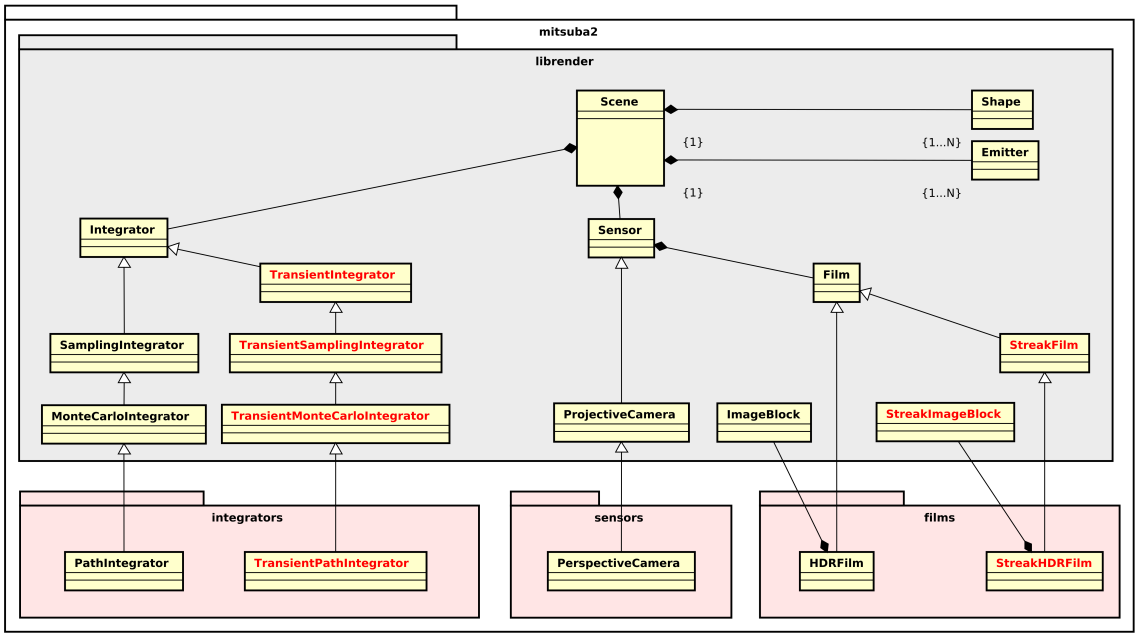


Figura 4.2: Diagrama de clases de Mitsuba 2 Transitorio con las componentes creados para la simulación del transporte de luz transitorio en rojo. Se han omitido aquellas clases y módulos que no han sido modificados.

Streak Film La decisión de implementación más importante fue la estructura de datos utilizada por la película de ráfaga (`StreakHDRFilm`). La película de Mitsuba 2 (`Film`) utiliza la clase `ImageBlock` que almacena la radiancia en dos dimensiones, más una dimensión implícita porque la radiancia es almacenada en C canales (p. ej. $C = 1$ si la luz es monocroma o $C = 3$ si es RGB). La modificación de `ImageBlock` se descartó por romper con la decisión de diseño de extensibilidad. Tras un análisis, se propusieron tres alternativas: usar un vector de la librería estándar de C++ que almacenase T instancias de la clase `ImageBlock` (una instancia asociada a cada ventana temporal), usar un sólo *image block* con $C * T$ canales (de forma similar a la implementación de un renderizador transitorio de Pediredla *et al.* [34]) o implementar una nueva estructura de datos `StreakImageBlock`. La primera opción presentaba problemas de implementación por métodos existentes en la clase `ImageBlock` y la segunda alternativa mezcla los conceptos de representación de radiancia discretizada a C valores y la dimensión temporal, por lo que se decidió usar la tercera alternativa. La opción elegida es flexible al ser una clase aislada implementada con un solo objetivo y, además, implementaría la estructura de datos usando en todo momento la librería Enoki.

Distancia óptica A la hora de calcular el tiempo del camino de luz, se ha decidido utilizar la medida denominada distancia óptica, que es el tiempo del camino (segundos) multiplicado por la velocidad de la luz en el vacío (metros/segundo). La razón de esta

decisión se debe a que si una escena define sus geometrías en escalas pequeñas (por ejemplo a escala real donde una unidad en el renderizador equivale a un metro en la vida real), el tiempo de los caminos de esta escena va a ser extremadamente pequeño, del orden de 10^{-9}, provocando errores de *underflow*. Por ello, en el cálculo del tiempo asociado a los caminos de luz en todo el renderizador transitorio implementado en Mitsuba 2 se ha utilizado la distancia óptica. Para recuperar el tiempo real del camino a partir de la distancia óptica es suficiente con dividir por la velocidad de la luz en el vacío.

Capítulo 5

Resultados

En este capítulo mostramos los resultados de la extensión del renderizador Mitsuba 2 al estado transitorio implementada en este trabajo. Por un lado, comentamos escenas renderizadas con diferentes niveles de complejidad y algunos efectos que la simulación de la luz en estado transitorio permite visualizar (Sección 5.1). Por otro lado, realizamos una comparación de rendimiento de Mitsuba 2 en estado transitorio (Sección 5.2). Todas las escenas han sido renderizadas en un ordenador con el sistema operativo Xubuntu 18.04 con dos procesadores Intel Xeon Gold 6140 (18 cores y 36 hilos cada uno) y 256 GB de memoria RAM. El Anexo B incluye datos de todas las escenas, incluyendo su complejidad geométrica, resolución, y coste de la simulación.

5.1. Análisis de efectos de la luz en estado transitorio

Hemos utilizado el renderizador transitorio implementado para visualizar la propagación de la luz con distintas geometrías y tipos de fuentes de luz. En todas las escenas, las fuentes de luz producen un pulso delta de luz al comienzo ($t = 0$). En las dos primeras escenas (Figura 5.1 y Figura 5.2), compuestas por una *Cornell Box* con dos cubos de materiales difusos, podemos apreciar fácilmente la separación entre la iluminación directa[1] y la luz indirecta. La luz directa se corresponde al primer frente de onda. Se puede apreciar como la luz se propaga en forma de elipse, siendo el sensor y la fuente de luz los dos focos. En los últimos fotogramas de ambas figuras (Figura 5.1e y Figura 5.2e), se aprecia el efecto de *color bleeding* en la pared del fondo debido a la luz indirecta que ha rebotado primero en las paredes laterales roja o verde.

[1]La luz directa es aquella que va directamente desde la fuente de luz hasta el punto iluminado y la luz indirecta aquella que proviene del rebote de la luz en otra superficie.

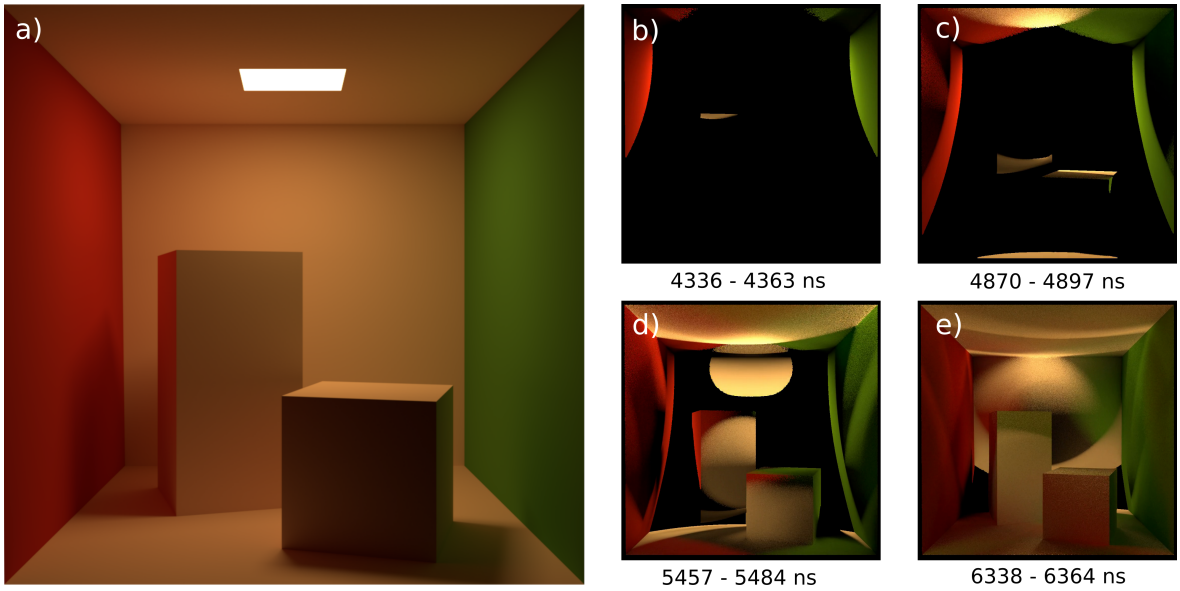


Figura 5.1: *Cornell Box* con luz de área que produce un pulso delta de luz al comienzo ($t = 0$). La imagen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal.

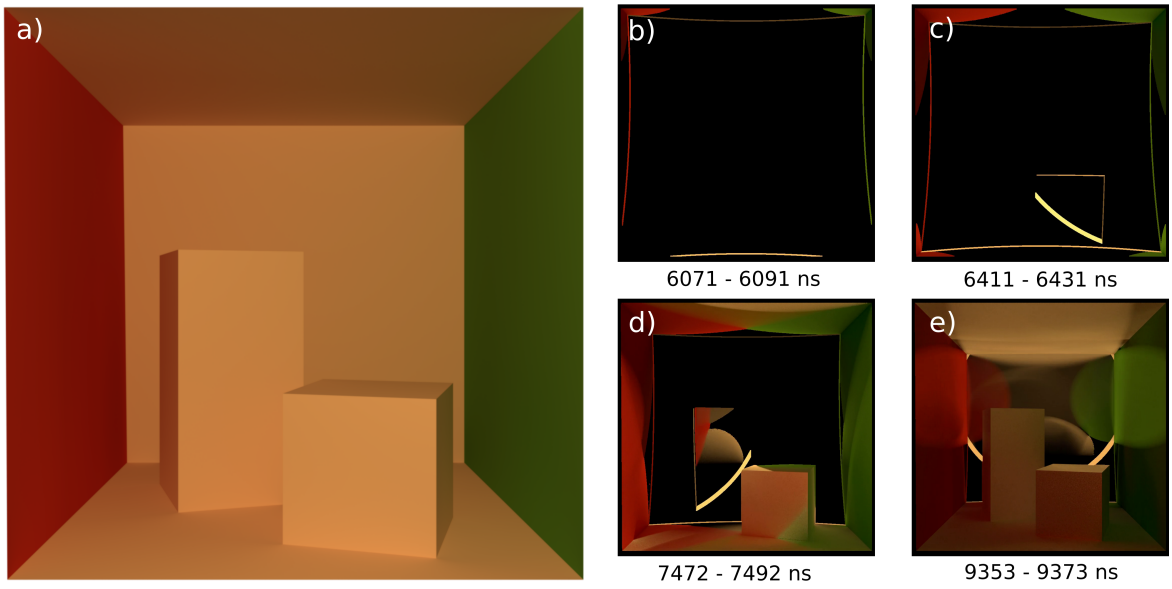


Figura 5.2: *Cornell Box* con luz puntual en la posición del sensor que produce un pulso delta de luz al comienzo ($t = 0$). La imagen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal.

La Figura 5.3 es una *Cornell Box* con tres esferas de materiales dieléctricos y conductores. Los materiales diélectricos son modelados por las ecuaciones de Fresnell produciendo efectos de reflexión y refracción en las esferas. La simulación de la luz en estado transitorio permite diferenciar estos dos efectos, principalmente en la esfera de la izquierda. En las Figuras 5.3b y 5.3c se aprecia el efecto de reflexión en la esfera izquierda ya que estos caminos de luz son más cortos y llegan antes a la cámara. En la Figura 5.3e, se puede apreciar el efecto de refracción en la esfera izquierda, viéndose la esquina del fondo de la escena a través de la esfera.

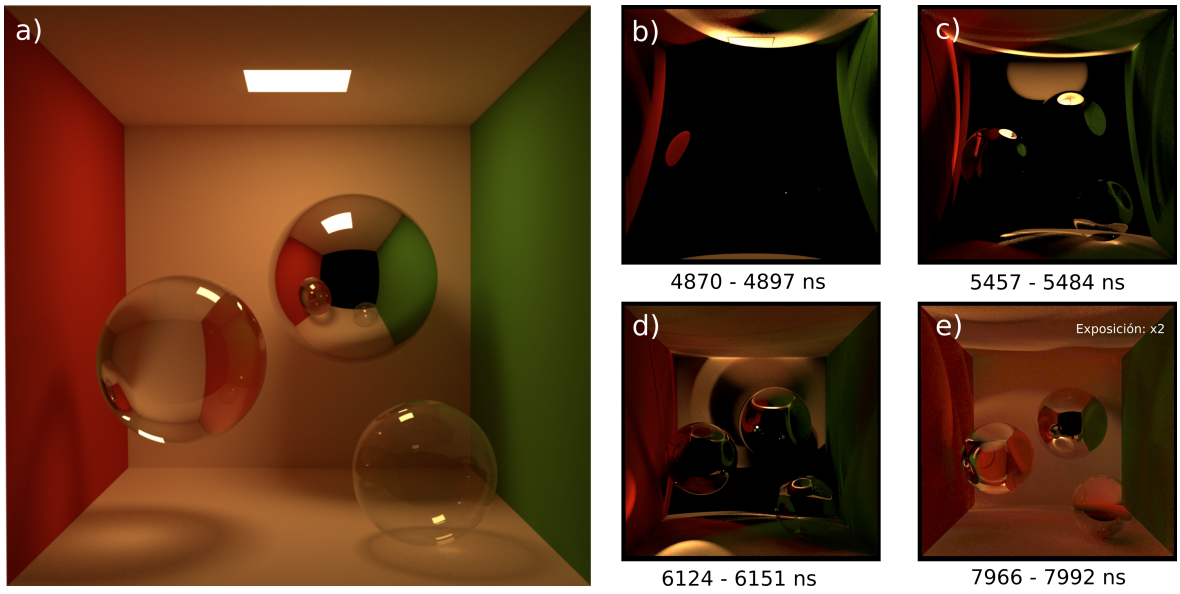


Figura 5.3: *Cornell Box* con luz de área que produce un pulso delta de luz al comienzo ($t = 0$). La esfera de la izquierda es un dieléctrico (diamante), la esfera del medio es un conductor (aluminio) y la esfera de la derecha es un dieléctrico fino. La imágen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal. Para una mejor visualización del efecto de refracción, la exposición de la imagen (e) se ha duplicado en comparación con las imágenes (b-d).

Los materiales dieléctricos producen cáusticas, que son patrones de luz producidos por la reflexión y refracción de rayos de luz reflejados o refractados por una superficie (p. ej. los patrones de luz en el fondo de la piscina). En la imagen 5.3a se puede ver la cáustica en el suelo y pared roja producida por la esfera izquierda y en la imagen 5.3d se puede ver la creación de esta cáustica. La Figura 5.4 presenta varios fotogramas que permiten ver mejor la creación de la cáustica.

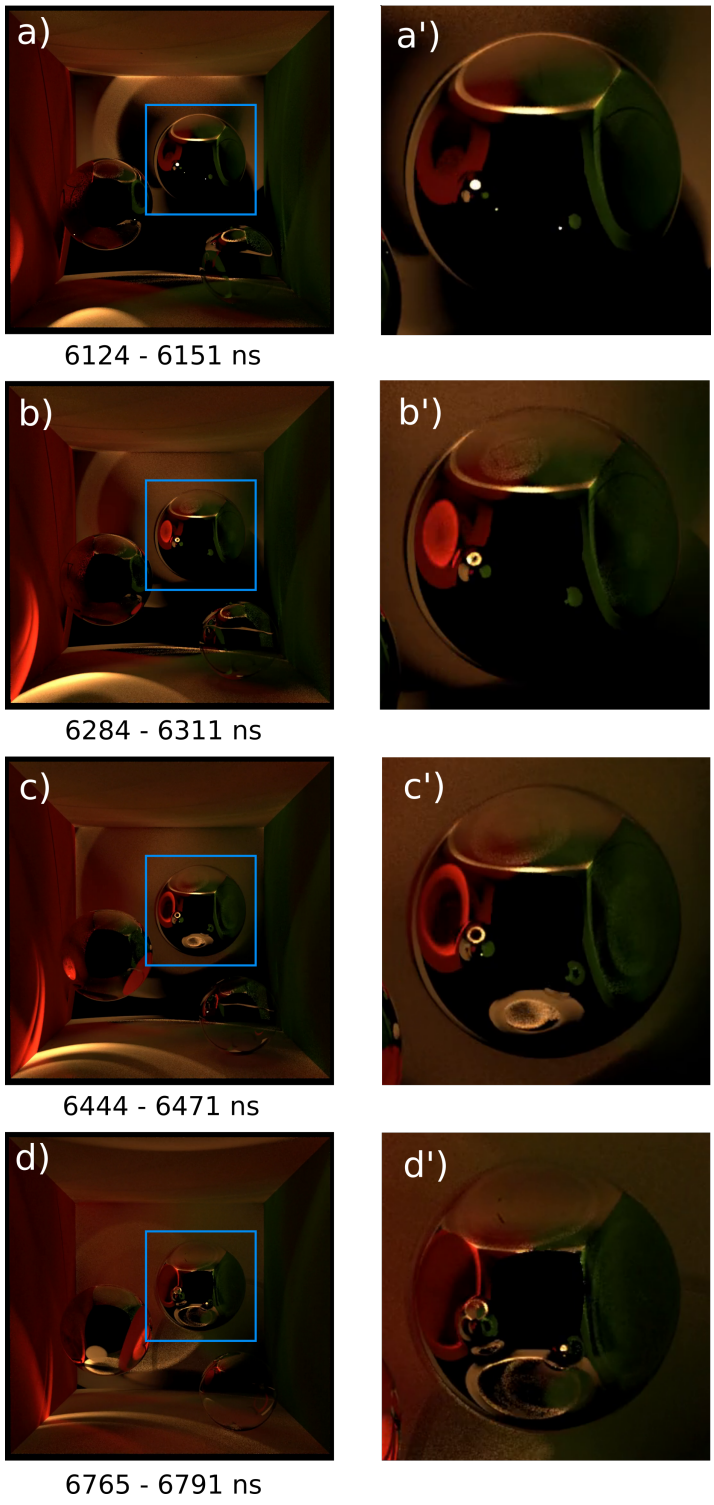


Figura 5.4: Fotogramas correspondientes a la Figura 5.3 para mostrar la creación de la cáustica causada por la esfera de la izquierda en la pared roja y el suelo. Las imágenes de la segunda columna son una ampliación del reflejo de la esfera conductora.

Hemos renderizado una escena teniendo en cuenta la polarización de la luz, representada mediante los vectores de Stokes[2], para mostrar que esta aplicación sigue siendo funcional en la versión transitoria de Mitsuba 2. La Figura 5.5 muestra la evolución de la polarización de la luz en la escena.

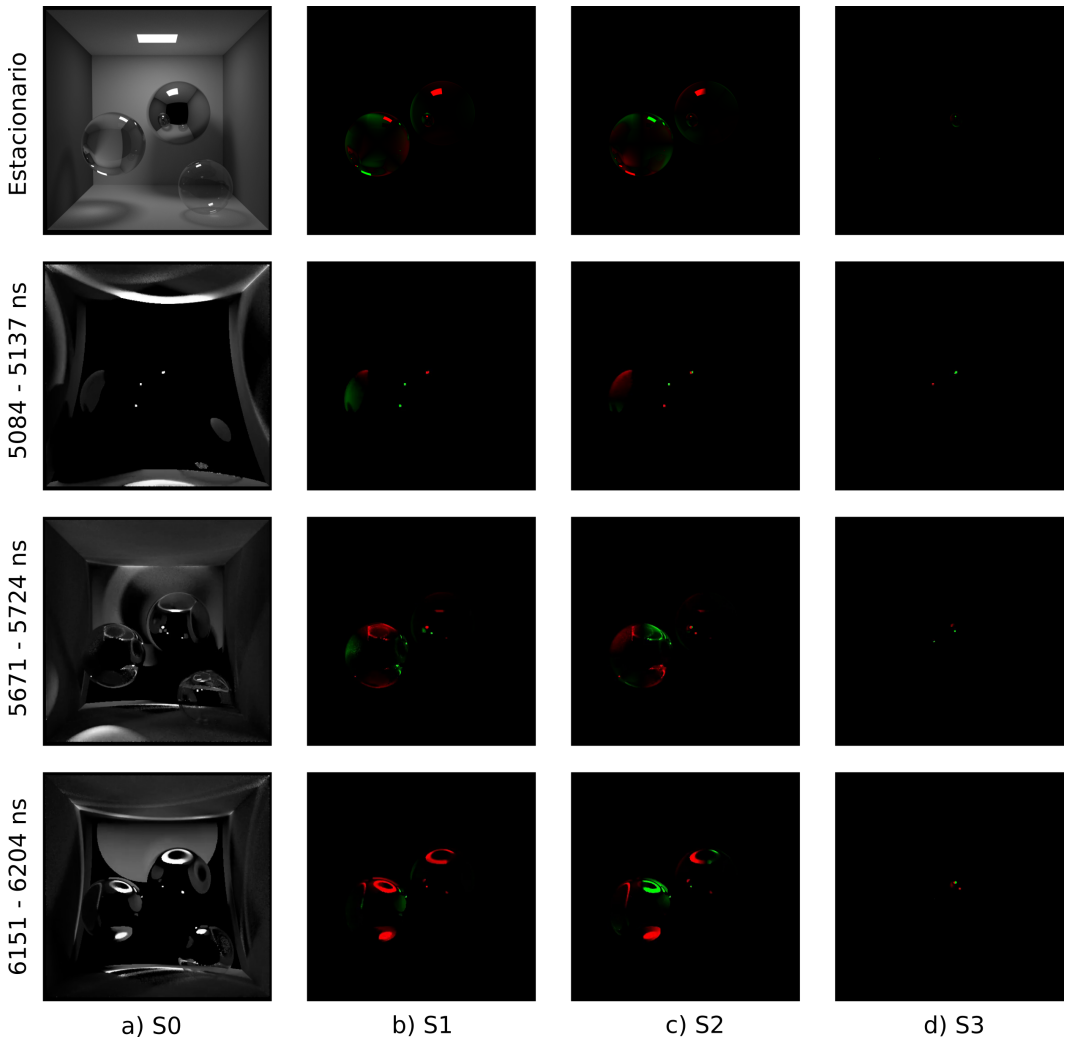


Figura 5.5: *Cornell Box* con luz de área que produce un pulso delta de luz al comienzo ($t = 0$). La esfera de la izquierda es un dieléctrico (diamante), la esfera del medio es un conductor (aluminio) y la esfera de la derecha es un dieléctrico fino. Polarización de la luz en estado estacionario y evolución de la polarización en el dominio temporal. La polarización está representada mediante los vectores de Stokes donde la la componente $S0$ muestra la radiancia, la componente $S1$ la polarización horizontal y vertical, la componente $S2$ la polarización diagonal y la componente $S3$ la polarización circular.

Además, hemos renderizado escenas más complejas, tanto por el número de geometrías como por los materiales. Las Figuras 5.6, 5.7 y 5.8 muestran escenas realistas del interior de un apartamento. La figura del baño (ver Figura 5.6) es especialmente interesante porque muestra cómo la formación del reflejo en el espejo se

[2]Los vectores de Stoke son una forma compacta de representar el estado medio de polarización de la luz, mediante un vector de cuatro componentes: El primero de ellos codifica la intensidad (radiancia), el segundo y tercero diferentes grados de polarización lineal, y el cuarto componente la polarización circular. Referimos a otras fuentes [33] para más detalles.

forma después de que la habitación sea iluminada porque se necesita que la luz rebote en el espejo y, por tanto, los caminos de luz son más largos.

Finalmente, la Figura 5.9 muestra una escena exterior de gran complejidad geométrica, demostrando que el renderizador transitorio implementado supone una mejora en cuanto al manejo de escenas complejas en comparación con el trabajo de Jarabo *et al.* [25].

Figura 5.6: *Contemporary Bathroom* [45]. Contiene una fuente de luz de área en el rectángulo de la ventana que produce un pulso delta de luz al comienzo ($t = 0$). La imagen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal. Estos fotogramas muestran la creación del reflejo en el espejo.

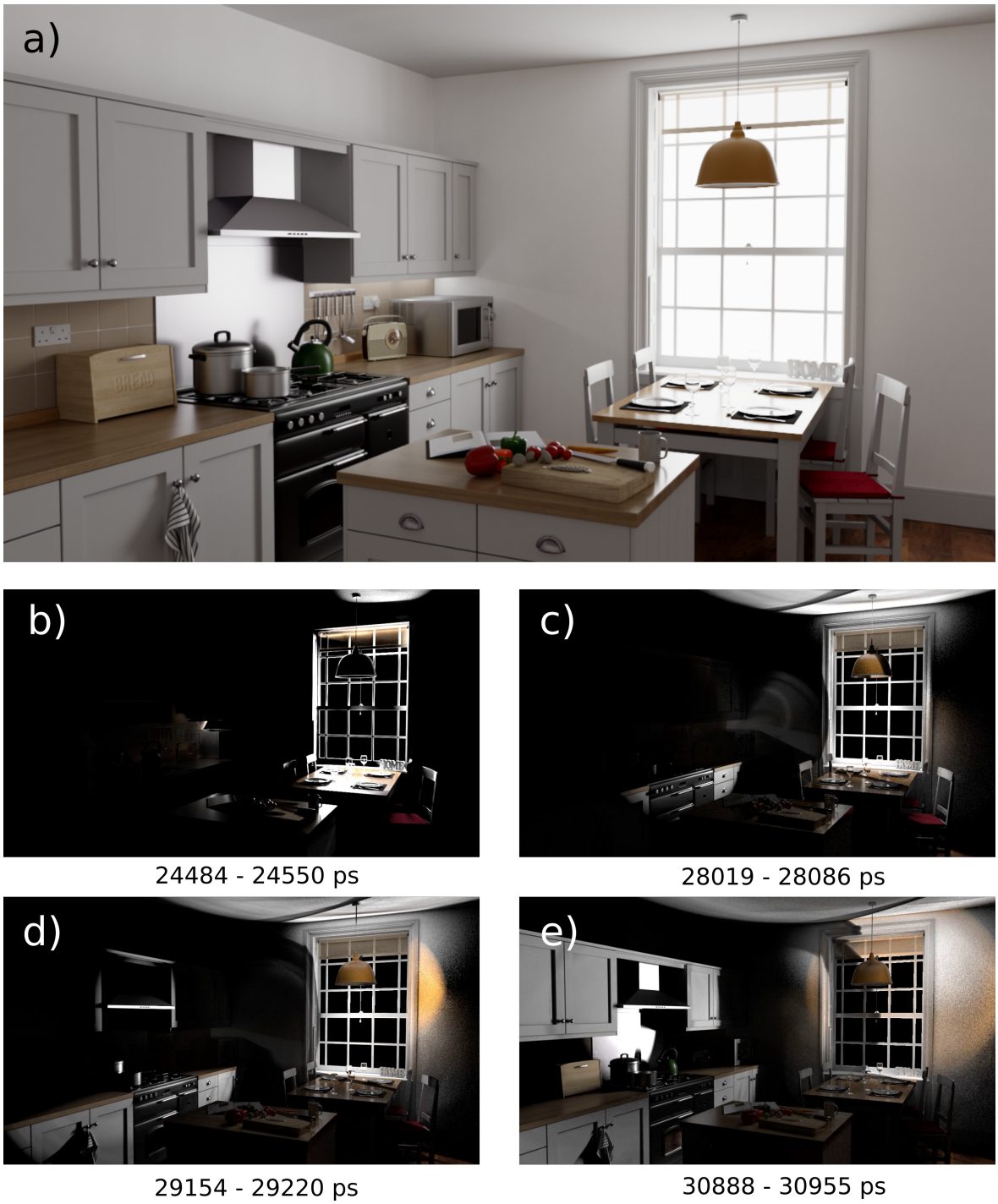


Figura 5.7: *Country Kitchen* [45]. Contiene una fuente de luz de área en el rectángulo de la ventana y una fuente de luz de área en la bombilla dentro de la lámpara naranja. Ambas fuentes de luz producen un pulso delta de luz al comienzo ($t = 0$). La imagen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal.

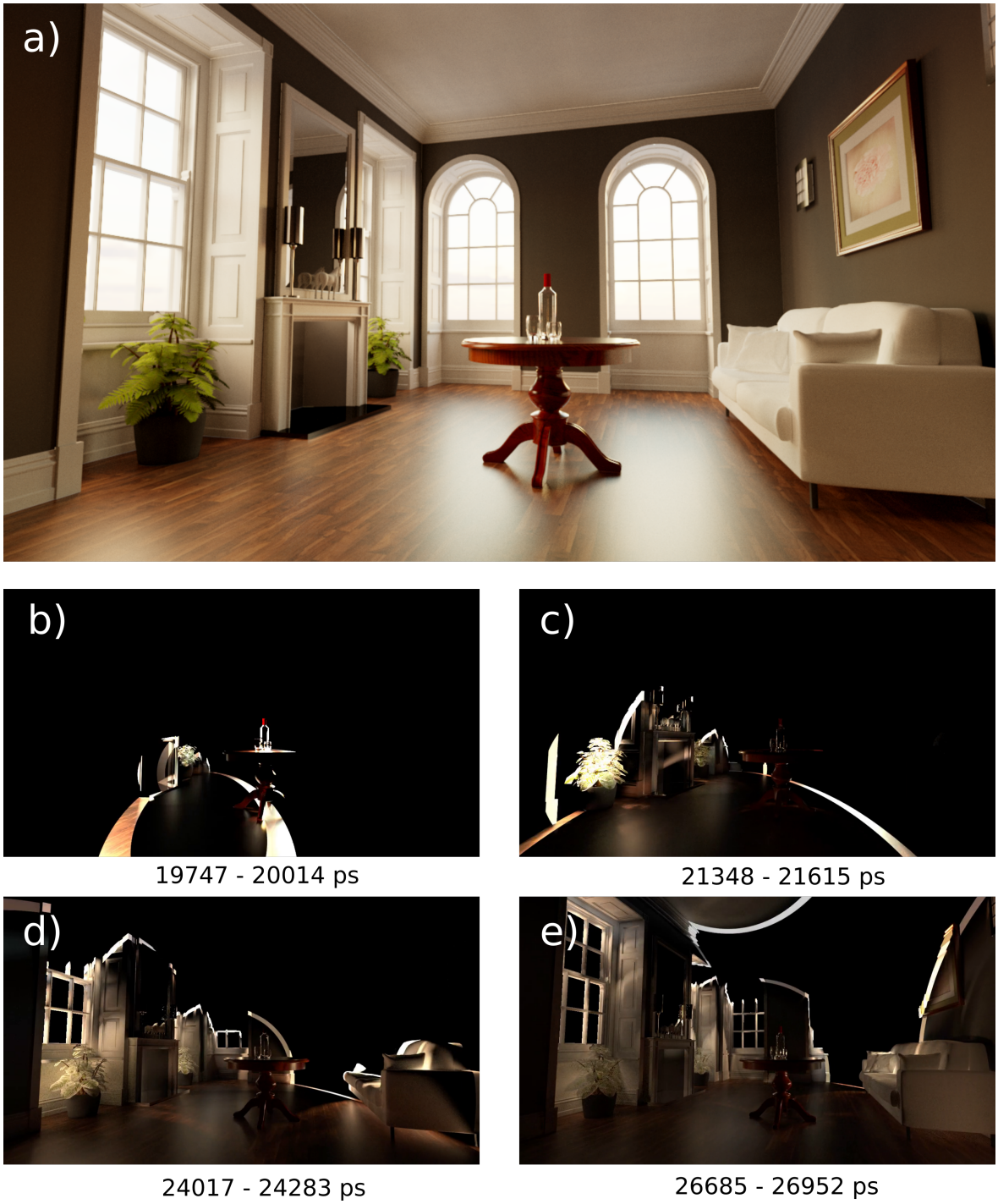


Figura 5.8: *The Grey & White Room* [45]. Contiene una fuente de luz de ambiente que ilumina la habitación a través de las ventanas que produce un pulso delta de luz al comienzo ($t = 0$). La imagen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal.

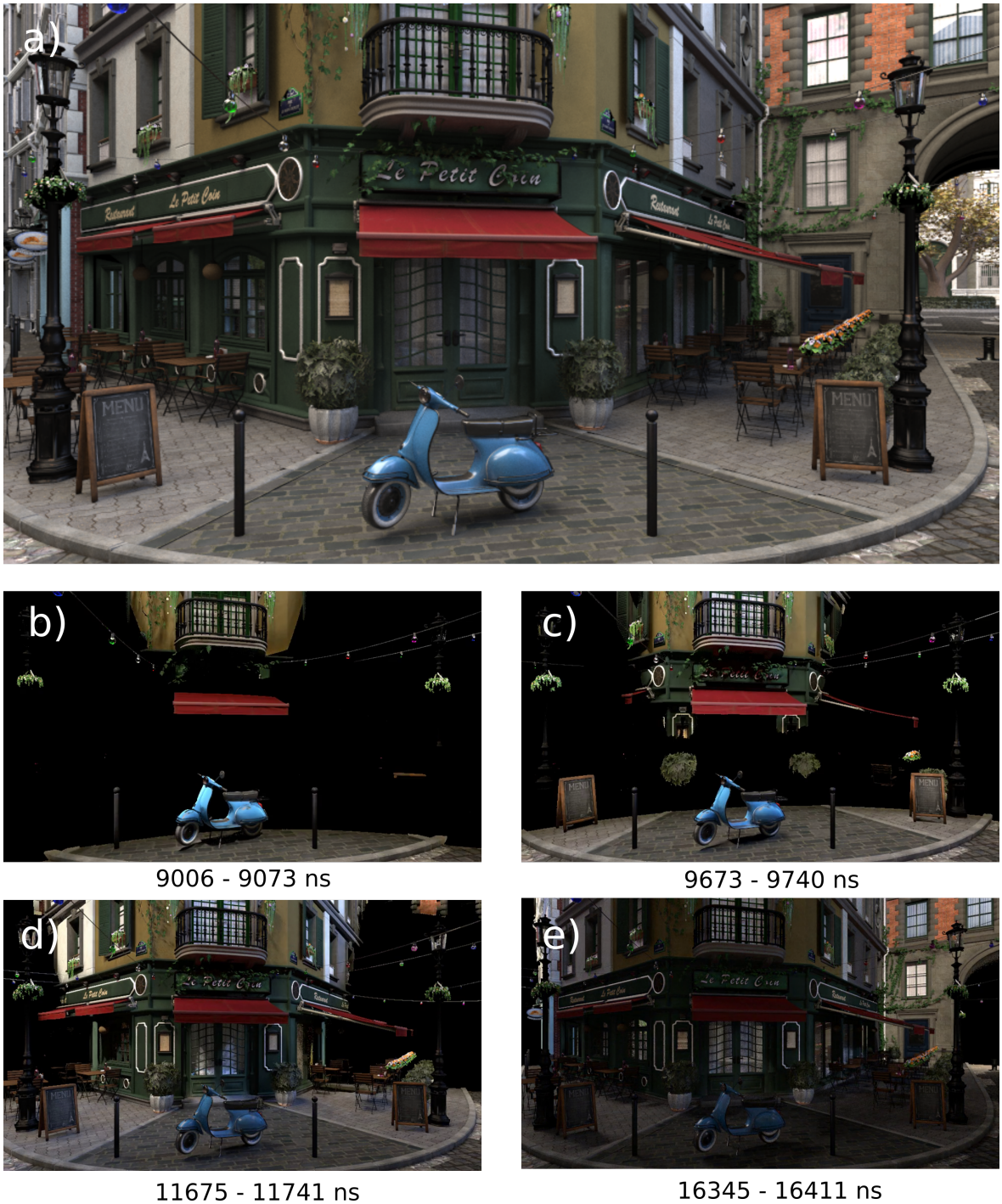


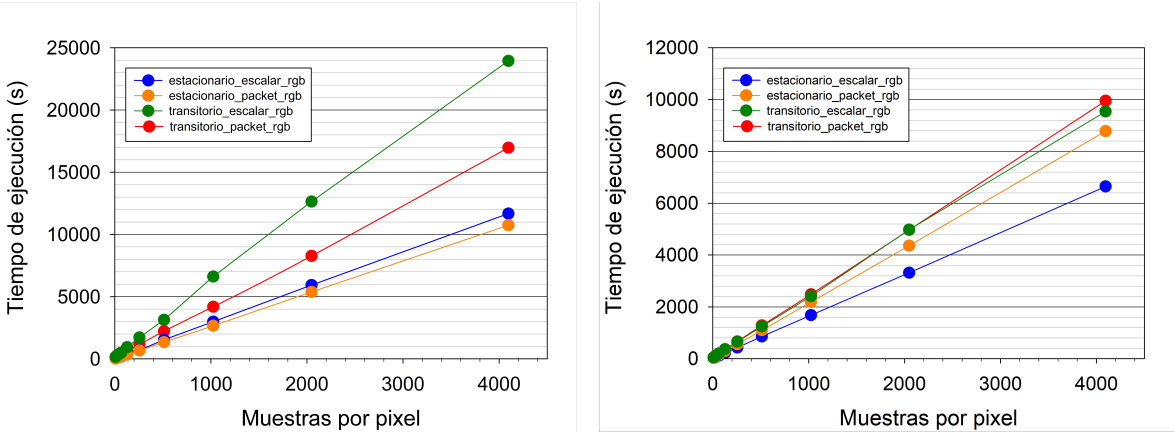
Figura 5.9: *Bistro* [46]. Contiene una fuente de luz de área en el cielo. La imagen (a) es un *render* en estado estacionario. Las imágenes (b-e) son fotogramas de la película de ráfaga que muestran la evolución de la radiancia en el dominio temporal.

5.2. Análisis de rendimiento

Uno de los aspectos más importantes de un renderizador transitorio es su eficiencia porque la simulación del transporte de luz en modo transitorio introduce una nueva dimensión a muestrear que incrementa el tiempo de convergencia de los algoritmos. Esto implica que el número de muestras a utilizar para conseguir un resultado aceptable es mayor.

La Figura 5.10 muestra el coste introducido por el algoritmo de *path tracing* transitorio respecto de la versión estacionaria del algoritmo en las escenas de la *Cornell Box* y de *Contemporary Bathroom*. En la escena de *Cornell Box*, la versión transitoria introduce un *overhead* x2.23 en la versión escalar y x1.81 en la versión vectorizada. En la escena de *Contemporary Bathroom*, la versión transitoria introduce un *overhead* x1.5 en la versión escalar y x1.2 en la versión vectorizada. Esto indica que el *overhead* introducido por la versión transitoria depende de la escena (ambas escenas evalúan el mismo número de ventanas temporales) y esto puede deberse al número y longitud de los caminos que acaban contribuyendo a la radiancia y que dependen de los elementos de la escena. También es interesante ver cómo la versión vectorizada de los algoritmos es más rápida en la escena de *Cornell Box* mientras que es más lenta en la escena de *Contemporary Bathroom*. Este puede deberse a que la implementación del algoritmo transitorio y las modificaciones hechas a Mitsuba 2 para soportar el transporte de luz transitorio no consiguen aprovechar la vectorización que proporciona Mitsuba 2. Por ejemplo, cuánto más diferentes son los caminos de luz creados, más ineficiente es el algoritmo porque el procesador debe cambiar entre el modo vectorizado y el modo no vectorizado. Existen versiones del algoritmo de *path tracing* (*coherent path tracing* [47] [48]) en las que los caminos creados son similares, mejorando la eficiencia de las versiones vectorizadas.

Finalmente, se ha comparado la eficiencia entre la implementación del algoritmo de *path tracing* transitorio en Mitsuba 2 y la implementación realizada por Jarabo *et al.* [25] (bunny-killer) que era la base de este trabajo. La Figura 5.11 presenta la complejidad temporal de ambas implementaciones respecto del número de muestras por píxel utilizadas. En esta figura podemos ver cómo la implementación en Mitsuba 2 es más rápida que la implementación por Jarabo *et al.*, ya que la complejidad de bunny-killer es cuadrática debido a decisiones de implementación mientras que el tiempo de ejecución del renderizador implementado en este trabajo crece linealmente respecto del número de muestras por píxel.



(a) *Cornell Box* (Figura 5.1) **(b)** *Contemporary Bathroom* (Figura 5.6)

Figura 5.10: Comparación del tiempo de ejecución (segundos) respecto del número de muestras por píxel para la versión estacionaria y transitoria de *path tracing* de manera no vectorizada y vectorizada.

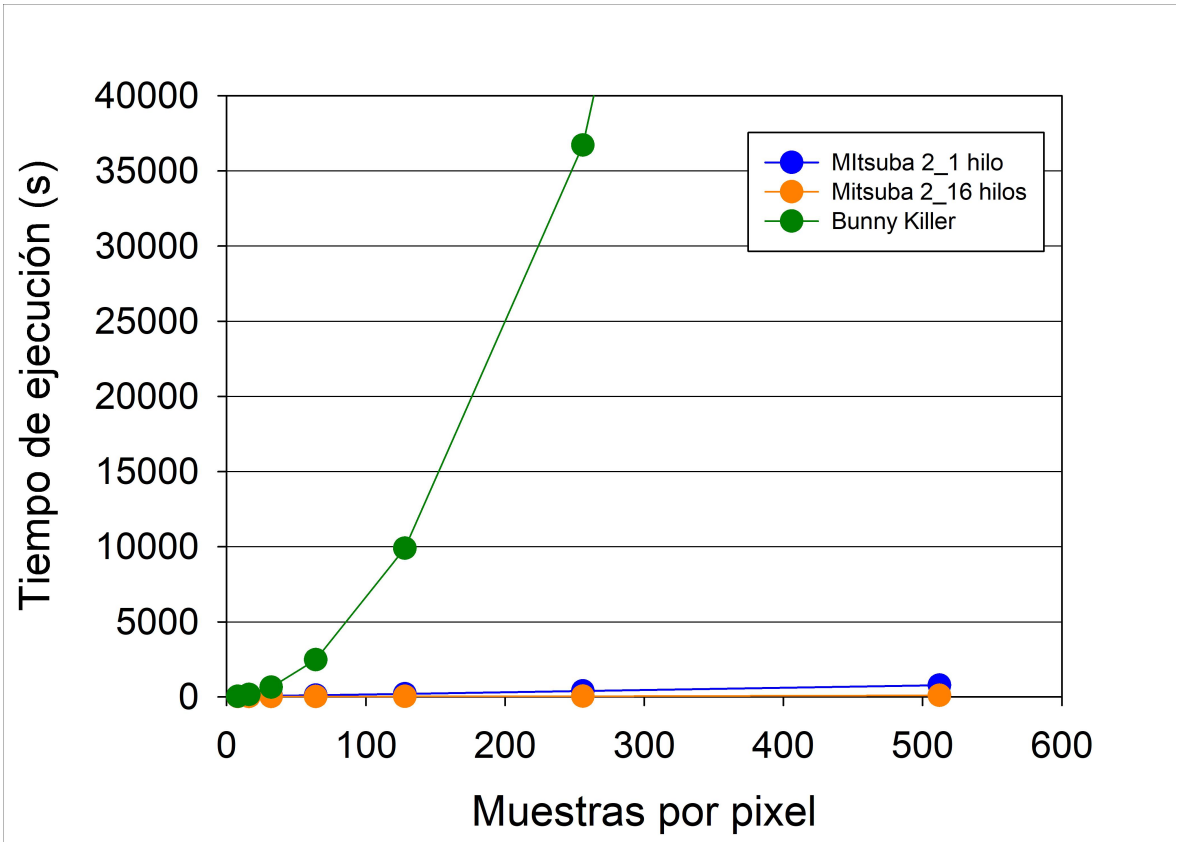


Figura 5.11: Comparación del tiempo de ejecución (segudos) respecto del número de muestras por píxel del renderizador transitorio presentado por Jarabo *et al.* [25] (bunny-killer) y la implementación de este trabajo en Mitsuba 2. El renderizador bunny-killer solo permite su ejecución en un hilo.

Capítulo 6

Conclusiones

En este trabajo se ha extendido el software de simulación de transporte de luz basado en física Mitsuba 2 para simular el transporte de luz en estado transitorio, dejando de considerar la velocidad de la luz infinita. Partiendo del marco de trabajo teórico establecido por Jarabo *et al.* [25] para el transporte de luz en estado transitorio, se han definido las funcionalidades básicas de un renderizador transitorio. Posteriormente, se ha implementado el algoritmo de *path tracing* en estado transitorio sobre Mitsuba 2. Esta implementación soporta la simulación de la luz polarizada y multiespectral, además de su ejecución en CPU de manera escalar y vectorizada. Finalmente, se ha usado el renderizador transitorio implementado para analizar efectos de la propagación de la luz solo visibles al considerar la velocidad de la luz finita como la separación entre la luz directa e indirecta, la separación de los efectos de reflexión y refracción en materiales dieléctricos o la creación del reflejo en un cristal.

La realización de este proyecto, desde la comprensión de la parte teórica del transporte de luz transitorio a la implementación en Mitsuba 2, me ha resultado un reto muy interesante. Esta dualidad de la parte teórica y de implementación es lo que más me ha gustado ya que no hay mejor manera de entender algo que implementarlo. Elegí este tema para el Trabajo de Fin de Grado porque el renderizado basado en física es un área que me atrae porque permite describir un fenómeno físico tan complicado como el transporte de luz a través de las matemáticas. Además, este proyecto me ha permitido aplicar el conocimiento técnico adquirido durante la carrera en áreas como estadística y matemáticas, ingeniería del software para la comprensión y descripción de un sistema complejo como es un renderizador, la programación de sistemas concurrentes a la hora de paralelizar el algoritmo de *path tracing* y, como no, el área de informática gráfica.

Considero que ha sido un reto complejo, especialmente al principio, cuando tuve

que comprender las bases teóricas del transporte de luz transitorio porque cada vez que leía de nuevo uno de los artículos de referencia veía una nueva información que no había relacionado antes con los conocimientos que ya tenía. Además, Mitsuba 2, por sus particularidades como la (meta)programación genérica de los algoritmos para su ejecución en CPU y GPU o los *bindings* con Python tiene una curva de trabajo bastante grande, que se incrementa por la reducida documentación que existe, llegando a ser nula en algunas partes de Mitsuba 2. Las reuniones con Adrián y sus explicaciones me han ayudado bastante porque incluso cuando un concepto era sencillo, el comentarlo en voz alta me ayudaba a comprenderlo mejor.

En retrospectiva, un punto a mejorar sería la creación de un prototipo funcional de renderizador transitorio antes, en vez de centrarme en que lo que desarrollase fuera la opción correcta y final. Esto me habría permitido ver el sistema completo en funcionamiento y creo que me hubiera ayudado a ver mucho antes dificultades que han aparecido posteriormente. Creo que esta opción es mejor aunque la primera versión desarrollada fuera desechada totalmente y tuviese que implementarse de nuevo.

Trabajo Futuro: Este trabajo ha implementado las bases de la simulación de la luz en estado transitorio en Mitsuba 2. De esta forma, se podría usar este trabajo como punto de partida para extender otros algoritmos de simulación de la luz como *bidirectional path tracing* o *photon mapping* al estado transitorio en Mitsuba 2, cambiando solamente el integrador. Además, los materiales utilizados en este trabajo no introducen retrasos de dispersión, pero la implementación de este tipo de materiales, de una manera sencilla por cómo es el código actual, permitiría la simulación de otros efectos del transporte de la luz interesantes como la fluorescencia.

Más allá de implementaciones relativamente directas de algoritmos estacionarios, la versatiliad de Mitsuba 2 y la diversidad de técnicas de guiado de caminos (*path guiding*) implementados sobre el renderizador, puede dar pie a mejores técnicas de muestreo en el dominio temporal.

Finalmente, otra dirección interesante a tener en cuenta es el desarrollo de un renderizador diferenciable transitorio a partir de este trabajo gracias a la biblioteca Enoki sobre la que está implementada Mitsuba 2, la cuál permite la ejecución del renderizador sobre GPU y de manera autodiferenciable. Esto tiene un enorme potencial en el campo de la imagen transitoria, de cara a resolver problemas inversos de forma eficiente, incluyendo la reconstrucción de escenas ocluidas. Un ejemplo de este potencial

se muestra en el reciente trabajo de Yi *et al.* [49], en el que paralelamente a este trabajo desarrollan un renderizador transitorio diferenciable, que demuestran reconstruyendo por primera vez escenas a través de dos esquinas.

Capítulo 7

Bibliografía

[1] Carl B Boyer. Early estimates of the velocity of light. *Isis*, 33(1):24–40, 1941.

[2] Andreas Velten, Di Wu, Adrian Jarabo, Belen Masia, Christopher Barsi, Chinmaya Joshi, Everett Lawson, Moungi Bawendi, Diego Gutierrez, and Ramesh Raskar. Femto-photography: capturing and visualizing the propagation of light. *ACM Trans. Graph.*, 32(4):1–8, 2013.

[3] Adrian Jarabo, Belen Masia, Julio Marco, and Diego Gutierrez. Recent advances in transient imaging: A computer graphics and vision perspective. *Visual Informatics*, 1(1), 2017.

[4] Julio Marco, Quercus Hernandez, Adolfo Muñoz, Yue Dong, Adrian Jarabo, Min Kim, Xin Tong, and Diego Gutierrez. Deeptof: Off-the-shelf real-time correction of multipath interference in time-of-flight imaging. *ACM Transactions on Graphics (SIGGRAPH Asia 2017)*, 36(6), 2017.

[5] Nikhil Naik, Shuang Zhao, Andreas Velten, Ramesh Raskar, and Kavita Bala. Single view reflectance capture using multiplexed scattering and time-of-flight imaging. *ACM Trans. Graph.*, 30, 2011.

[6] Di Wu, Andreas Velten, Matthew O'Toole, Belen Masia, Amit Agrawal, Qionghai Dai, and Ramesh Raskar. Decomposing global light transport using time of flight imaging. *IEEE International Conference on Computer Vision (ICCV)*, 107(2), 2014.

[7] Di Wu, Gordon Wetzstein, Christopher Barsi, Thomas Willwacher, Matthew O'Toole, Nikhil Naik, Qionghai Dai, Kyros Kutulakos, and Ramesh Raskar. Frequency analysis of transient light transport with applications in bare sensor imaging. In *European Conference on Computer Vision (ECCV)*, 2012.

[8] Andreas Velten, Thomas Willwacher, Otkrist Gupta, Ashok Veeraraghavan, Moungi G. Bawendi, and Ramesh Raskar. Recovering three-dimensional shape around a corner using ultrafast time-of-flight imaging. *Nature Communications*, (3), 2012.

[9] Victor Arellano, Diego Gutierrez, and Adrian Jarabo. Fast back-projection for non-line of sight reconstruction. *Optics Express*, to appear, 2017.

[10] Matthew O'Toole, David B Lindell, and Gordon Wetzstein. Confocal non-line-of-sight imaging based on the light-cone transform. *Nature*, 555(7696):338, 2018.

[11] Xiaochun Liu, Ibón Guillén, Marco La Manna, Ji Hyun Nam, Syed Azer Reza, Toan Huu Le, Adrian Jarabo, Diego Gutierrez, and Andreas Velten. Non-line-of-sight imaging using phasor fields virtual wave optics. *Nature*, 2019.

[12] Rihui Wu, Adrian Jarabo, Jinli Suo, Feng Dai, Yongdong Zhang, Qionghai Dai, and Diego Gutierrez. Adaptive polarization-difference transient imaging for depth estimation in scattering media. *Optics Letters*, 43(6), 2018.

[13] David B Lindell and Gordon Wetzstein. Three-dimensional imaging through scattering media based on confocal diffuse tomography. *Nature communications*, 11(1):1–8, 2020.

[14] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[15] Greg Hymphreys Matt Pharr, Wenzel Jakob. *Physically Based Rendering: From Theory to Implementation*.

[16] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), December 2019.

[17] Shuang Zhao, Wenzel Jakob, and Tzu-Mao Li. Physics-based differentiable rendering: From theory to implementation. In *ACM SIGGRAPH 2020 Courses*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Per H Christensen and Wojciech Jarosz. The path to path-traced movies. *Foundations and Trends® in Computer Graphics and Vision*, 10(2):103–175, 2016.

[19] Jaroslav Křivánek, Christophe Chevallier, Vladimir Koylazov, Ondřej Karlíik, Henrik Wann Jensen, and Thomas Ludwig. Realistic rendering in architecture and product visualization. In *ACM SIGGRAPH 2018 Courses*, pages 1–5. 2018.

[20] Chen Bar, Marina Alterman, Ioannis Gkioulekas, and Anat Levin. A monte carlo framework for rendering speckle statistics in scattering media. *ACM Trans. Graph.*, 38(4):1–22, 2019.

[21] Adithya Pediredla, Yasin Karimi Chalmiani, Matteo Giuseppe Scopelliti, Maysamreza Chamanzar, Srinivasa Narasimhan, and Ioannis Gkioulekas. Path tracing estimators for refractive radiative transfer. *ACM Trans. Graph.*, 39(6):1–15, 2020.

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

[23] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of SIGGRAPH*, page 14, 1979.

[24] Eric Veach. *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.

[25] Adrian Jarabo, Julio Marco, Adolfo Munoz, Raul Buisan, Wojciech Jarosz, and Diego Gutierrez. A framework for transient rendering. *ACM Trans. Graph.*, 33(6):1–10, 2014.

[26] Adam Smith, James Skorupski, and James Davis. Transient rendering. Technical Report UCSC-SOE-08-26, School of Engineering, University of California, Santa Cruz, 2008.

[27] Adrian Jarabo. Femto-photography: Visualizing light in motion. Master's thesis, Universidad de Zaragoza, 2012.

[28] Jaroslav Krivánek, Iliyan Georgiev, Anton Kaplanyan, and Juan Canada. Recent advances in light transport simulation: Theory and practice. In *ACM SIGGRAPH 2013 Courses*, 2013.

[29] Phil Pitts, Arrigo Benedetti, Malcolm Slaney, and Phil Chou. Time of flight tracer. Technical report, Microsoft, 2014.

[30] Marco Ament, Christoph Bergmann, and Daniel Weiskopf. Refractive radiative transfer equation. *ACM Trans. Graph.*, 33(2), 2014.

[31] Matthew O'Toole, Felix Heide, Lei Xiao, Matthias B. Hullin, Wolfgang Heidrich, and Kiriakos N. Kutulakos. Temporal frequency probing for 5D transient analysis of global light transport. *ACM Trans. Graph.*, 33(4), 2014.

[32] A Adam, C Dann, O Yair, S Mazor, and S Nowozin. Bayesian time-of-flight for realtime shape, illumination and albedo. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2016.

[33] Adrian Jarabo and Victor Arellano. Bidirectional rendering of vector light transport. *Computer Graphics Forum*, 37(6):96–105, 2018.

[34] Adithya Pediredla, Ashok Veeraraghavan, and Ioannis Gkioulekas. Ellipsoidal path connections for time-gated rendering. *ACM Trans. Graph.*, 38(4):1–12, 2019.

[35] Julian Iseringhausen and Matthias B Hullin. Non-line-of-sight reconstruction using efficient transient rendering. *ACM Trans. Graph.*, 39(1):1–14, 2020.

[36] Chia-Yin Tsai, Aswin C Sankaranarayanan, and Ioannis Gkioulekas. Beyond volumetric albedo–a surface optimization framework for non-line-of-sight imaging. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1545–1555, 2019.

[37] Xian Pan, Victor Arellano, and Adrian Jarabo. Transient instant radiosity for efficient time-resolved global illumination. *Computers & Graphics*, 83:107–113, 2019.

[38] Wenzheng Chen, Fangyin Wei, Kiriakos N. Kutulakos, Szymon Rusinkiewicz, and Felix Heide. Learned feature embeddings for non-line-of-sight imaging and recognition. *ACM Trans. Graph.*, 39(6), 2020.

[39] Julio Marco, Ibón Guillén, Wojciech Jarosz, Diego Gutierrez, and Adrian Jarabo. Progressive transient photon beams. *Computer Graphics Forum*, 2019.

[40] Greg Hymphreys Matt Pharr, Wenzel Jakob. Pbrt v4. https://github.com/mmp/pbrt-v4, 2021.

[41] Simon Kallweit, Petrik Clarberg, Craig Kolb, Kai-Hwa Yao, Theresa Foley, Lifan Wu, Lucy Chen, Tomas Akenine-Moller, Chris Wyman, Cyril Crassin, and Nir Benty. The falcor rendering framework, 08 2021.

[42] Yong He, Kayvon Fatahalian, and Tim Foley. Slang: Language mechanisms for extensible real-time shading systems. *ACM Trans. Graph.*, 37(4):141:1–141:13, July 2018.

[43] Cheng Zhang, Lifan Wu, Changxi Zheng, Ioannis Gkioulekas, Ravi Ramamoorthi, and Shuang Zhao. A differential theory of radiative transfer. *ACM Trans. Graph.*, 38(6):227:1–227:16, 2019.

[44] Wenzel Jakob. Enoki: structured vectorization and differentiation on modern processor architectures, 2019. https://github.com/mitsuba-renderer/enoki.

[45] Benedikt Bitterli. Rendering resources, 2016. https://benedikt-bitterli.me/resources/.

[46] Amazon Lumberyard. Amazon lumberyard bistro, open research content archive (orca), July 2017. `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`.

[47] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 101–108, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[48] Iman Sadeghi, Bin Chen, and Henrik Wann Jensen. Coherent path tracing. *Journal of Graphics, GPU & Game Tools.*, 14(2):33–43, January 2009.

[49] Shinyoung Yi, Donggun Kim, Kiseok Choi, Adrian Jarabo, Diego Gutierrez, and Min H. Kim. Differentiable transient rendering. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2021)*, 40(6), 2021.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Código del algoritmo de *path tracing*

El algoritmo de *path tracing* consiste en crear N caminos y promediar su contribución siguiendo el estimador de Monte Carlo de la integral de caminos (ecuación 2.8). Para una mejor comprensión del algoritmo, la Figura A.1 presenta una versión básica del algoritmo de *path tracing* en estado estacionario.

La aproximación básica de *path tracing* tiene varios problemas. Por un lado, un camino que no interseccione con ninguna fuente de luz antes del límite máximo de intersecciones no contribuirá al valor del píxel. Puesto que es probable que un camino no acabe interseccionando una fuente de luz si estas tienen un área pequeña, se pueden hacer conexiones deterministas entre una intersección y las fuentes de luz (*next event estimation* [15]). Además, que la condición de terminación sea un número fijo de rebotes no es óptimo ya que deberá ser ajustado manualmente para cada escena. En general, a mayor número de intersecciones, la radiancia asociada a dicho camino será menor porque la luz es dispersada en cada intersección. Existe un método llamado ruleta rusa [15] para terminar con aquellos caminos que hayan dispersado mucho la luz del emisor y apenas aporten radiancia al pixel. Existen otras mejoras más avanzadas como el muestreo múltiple por importancia (*multiple importance sampling*, o *MIS*) introducida por Veach [24].

```
void render(Scene &scene) {
    Film film = scene.getFilm();

    for (int x = 0; x < film.width; ++x) {
        for (int y = 0; y < film.height; ++y) {
            Radiance radianceEstimatorMC = [0.f, 0.f, 0.f];
            for (int s = 0; s < n; ++n) {
                Ray original_ray = film.createRay(x, y);
                radianceEstimatorMC += createPath(scene, original_ray)
            }
            film[x][y] = radianceEstimatorMC / n;
        }
    }
}

Radiance createPath(Scene &scene, Ray original_ray) {
    // Radiance is discretized to 3 values (RGB)
    Radiance throughput = [1.f, 1.f, 1.f];
    Radiance result = [0.f, 0.f, 0.f];
    Ray ray = original_ray;

    // First Intersection
    SurfaceInteraction si = scene.intersect(ray);

    while(true) {

        // If intersection is emitter, evaluate the outgoing
        // radiance from the emitter and finish path
        if(si.material.type == EMITTER) {
            result = throughput * si.material.emitter->eval(ray);
            break;
        }

        // If length of path is greater than maximum length
        // allowed, finish path
        if(length > max_length)
           break;

        // Intersection is a surface with a material not emissive.
        // Sample new outgoing ray and evaluate interaction (BSDF).
        [outgoing_ray, bsdf_val] = si.material.bsdf->sample(ray);
        // Scale the throughput by the BSDF of the current
        // intersection
        throughput *= bsdf_val.value;

        // Continue intersecting other surfaces creating
        // the path until reaching an emitter
        ray = outgoing_ray;
        si = scene.intersect(ray);
    }
    return result;
}
```

Figura A.1: Código en C++ de la versión básica del algoritmo de *path tracing* en estado estacionario.

Anexos B

Resultados

B.1. Parámetros de las escenas renderizadas

Las escenas renderizadas (excepto la Figura 5.9) están disponibles en el formato de escena de Mitsuba 2 en el repositorio: *https://github.com/jgarciapueyo/mitsuba2-transient-scenes*. En este anexo, presentamos los parámetros con los que las escenas presentadas en el Capítulo 5 han sido renderizadas.

La Tabla B.1 contiene los parámetros con los que se han renderizado las imágenes correspondientes al transporte de luz en estado estacionario (imágenes (a) de las figuras).

	Resolución	SPP	Máx. rebotes	Modo	Tonemapper	# Caras	# Vértices
Fig. 5.1	1024x1024	16384	10	scalar_rgb	Logarítmico	38	76
Fig. 5.2	1024x1024	16384	10	scalar_rgb	Logarítmico	36	72
Fig. 5.3	1024x1024	16384	7	scalar_rgb	Logarítmico	38	76
Fig. 5.5	1024x1024	16384	7	scalar_rgb	Logarítmico	38	76
Fig. 5.6	1280x720	16384	15	scalar_rgb	Logarítmico	592186	304264
Fig. 5.7	1280x720	16384	17	scalar_rgb	Logarítmico	1443509	744051
Fig. 5.8	1280x720	16384	65	scalar_rgb	Logarítmico	143163	105056
Fig. 5.9	960x540	16384	24	scalar_rgb	Logarítmico	3836098	3614035

Tabla B.1: Parámetros usados en el renderizado estacionario de las escenas.

La Tabla B.2 contiene los parámetros con los que se han renderizado las imágenes correspondientes al transporte de luz en estado transitorio. La exposición temporal se refiere a la exposición temporal de cada ventana temporal. A la hora de comparar el número de muestras usadas entre varias escenas, este debe normalizarse por el número de ventanas temporales. Se ha aplicado un tonemapper logarítmico calculando el valor máximo y mínimo de manera global a lo largo de todas las ventanas temporales.

	Resolución	Ventanas temporales (VT)	Exposición temporal	Offset	Muestras por píxel (MPP)	MPP / VT	Máx. rebotes	Modo
Fig. 5.1	1024x1024	400	8	500	51200	128	10	scalar_rgb
Fig. 5.2	1024x1024	400	6	1100	51200	128	10	scalar_rgb
Fig. 5.3	1024x1024	400	8	500	102400	256	7	scalar_rgb
Fig. 5.4	1024x1024	400	8	500	102400	256	7	scalar_rgb
Fig. 5.5	1024x1024	200	16	500	51200	256	7	scalar_rgb
Fig. 5.6	1280x720	400	0.08	0	204800	512	15	scalar_rgb
Fig. 5.7	1280x720	400	0.02	5	204800	512	8	scalar_rgb
Fig. 5.8	1280x720	400	0.08	0	204800	512	12	scalar_rgb
Fig. 5.9	960x540	400	20	2100	102400	256	12	scalar_rgb

Tabla B.2: Parámetros usados en el renderizado transitorio de las escenas.

B.2. Datos de rendimiento

Las Tablas B.3 y B.4 muestran el tiempo de ejecución del renderizador Mitsuba 2 respecto del número de muestras por píxel en versión estacionaria y transitoria para las escenas de *Cornell Box* y *Contemporary Bathroom* respectivamente. Estos datos corresponden a las Figuras 5.10a y 5.10b. Para estas pruebas de rendimiento, la escena de *Cornell Box* se ha renderizado con unas dimensiones de 1024×1024 píxeles y con un número máximo de rebotes igual a 10. La escena de *Contemporary Bathroom* se ha renderizado con unas dimensiones de 1024×1024 píxeles y con un número máximo de rebotes igual a 15.

Muestras por píxel	Tiempo ejecución estacionario scalar_rgb	Tiempo ejecución estacionario packet_rgb	Tiempo ejecución transitorio scalar_rgb	Tiempo ejecución transitorio packet_rgb	Overhead transitorio scalar_rgb	Overhead transitorio packet_rgb
8	31	20	161	136	5.24	6.80
16	44	51	224	182	5.06	3.60
32	97	84	330	259	3.40	3.08
64	178	200	491	390	2.76	1.95
128	415	338	933	648	2.25	1.92
256	702	675	1717	1156	2.44	1.71
512	1534	1344	3137	2222	2.04	1.65
1024	2989	2663	6616	4189	2.21	1.57
2048	5933	5378	12644	8273	2.13	1.54
4096	11674	10741	23946	16969	2.05	1.58

Tabla B.3: Escena *Cornell Box*. Tiempo de ejecución (segundos) respecto del número de muestras por píxel y overhead introducido por la versión transitoria. Datos correspondientes a la Figura 5.10a.

La Tabla B.5 muestra el tiempo de ejecución para la escena *Cornell Box* del renderizador transitorio implementado por este trabajo en comparación con el renderizador transitorio implementado por Jarabo *et al.* [25] (bunny-killer), utilizado como referencia a lo largo del proyecto. El renderizador bunny-killer solo tiene la opción

Muestras por píxel	Tiempo ejecución estacionario scalar_rgb	Tiempo ejecución estacionario packet_rgb	Tiempo ejecución transitorio scalar_rgb	Tiempo ejecución transitorio packet_rgb	Overhead transitorio scalar_rgb	Overhead transitorio packet_rgb
8	31	31	47	46	1.48	1.47
16	38	50	69	73	1.77	1.43
32	70	83	115	116	1.64	1.39
64	121	153	201	198	1.65	1.28
128	228	295	367	355	1.60	1.20
256	433	558	654	665	1.51	1.19
512	860	1100	1246	1285	1.44	1.16
1024	1680	2188	2409	2485	1.43	1.13
2048	3319	4360	4979	4970	1.50	1.13
4096	6647	8781	9538	9952	1.43	1.13

Tabla B.4: Escena *Contemporary Bathroom*. Tiempo de ejecución (segundos) respecto del número de muestras por píxel y overhead introducido por la versión transitoria. Datos correspondientes a la Figura 5.10b.

de ejecutarse en un hilo. Para estas pruebas de rendimiento, la escena de *Cornell Box* se ha renderizado con unas dimensiones de 600×450 píxeles y con un número máximo de rebotes igual a 10.

Muestras por píxel	Mitsuba 2 (1 hilo)	Mitsuba 2 (16 hilos)	bunny-killer (1 hilo)
8	26	25	44
16	40	31	159
32	84	27	645
64	119	23	2474
128	208	40	9900
256	397	55	36720
512	786	104	144300

Tabla B.5: Escena *Cornell Box*. Tiempo de ejecución (segundos) respecto del número de muestras por píxel del renderizador transitorio implementado en Mitsuba 2 por este trabajo y el renderizador transitorio implementado por Jarabo *et al.* [25] (bunny-killer). Datos correspondientes a la Figura 5.11.

Anexos C

Planificación temporal

La realización de este proyecto se planificó para realizarse desde la finalización de las clases en Mayo hasta la presentación del mismo en Septiembre. Inicialmente, se definieron las fases del trabajo a realizar de la siguiente manera:

1. Documentación sobre los algoritmos de render transitorio
2. Análisis del renderizador Mitsuba 2
3. Implementación de simulador de transporte de luz transitorio en modo escalar
4. Extensión al modo vectorial y prueba sobre GPU
5. Evaluación, pruebas y documentación

En la práctica, este trabajo se ha desarrollado a tiempo completo en las fechas establecidas, complementando el trabajo realizado con una reunión semanal con el director del trabajo, Adrián Jarabo. Sin embargo, y como suele pasar en el desarrollo de software, algunas fases del trabajo (principalmente de implementación) han sido desarrolladas iterativamente como se puede comprobar en el cronograma. Siguiendo la planificación, las primeras semanas fueron dedicadas a la documentación sobre algoritmos de render transitorio y a analizar el código de Mitsuba 2 y Enoki. Después comenzó la implementación de los integradores transitorios modificando ligeramente Mitsuba 2 para conseguir una primera versión muy simple. Posteriormente se implementaron los plugins de la película de ráfaga para poder obtener la imagen transitoria de la escena en una sola ejecución del renderizador. Este proceso duró varias semanas en las que, de forma iterativa, se refactorizó el código y se implementó una *pipeline* para el postprocesamiento de las imágenes transitorias que produce el renderizador. Durante las siguientes semanas se solucionaron algunos problemas para que el algoritmo de *Path Tracing* transitorio funcionase de manera polarizada y vectorizada. Finalmente, se prepararon las escenas adaptándolas a la sintaxis de Mitsuba 2 y se realizó la memoria.

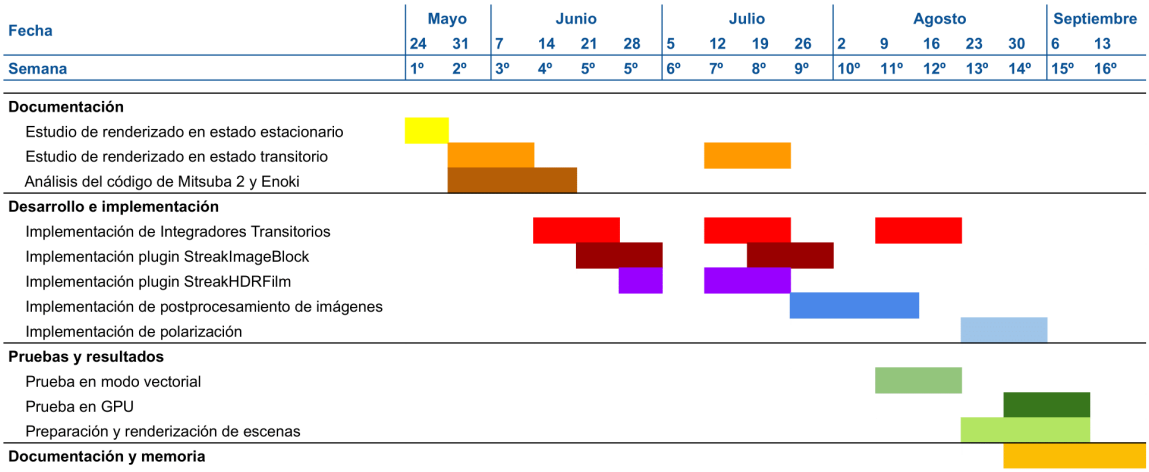


Figura C.1: Cronograma de las fases del trabajo.