



Universidad Zaragoza

Trabajo Fin de Grado

Reducción de ruido mediante estadísticas en render
estacionario y transitorio de Monte Carlo

A Statistical Denoiser for Steady and Transient Monte
Carlo rendering

Autor

Álvaro López Berga

Director y Codirector

Diego Gutierrez Pérez
Óscar Pueyo Ciudad

Agradecimientos

Me gustaría expresar mi agradecimiento a mi codirector, Óscar Pueyo, por su constante disposición para resolver mis dudas y por el seguimiento cercano mediante reuniones semanales, que han sido fundamentales para el desarrollo de este trabajo.

Así mismo, deseo agradecer a mi director, Diego Gutiérrez, por su orientación y apoyo, especialmente en la elaboración de la memoria. Esta parte del proyecto ha supuesto un gran reto para mí, ya que me resulta complejo plasmar mis ideas con claridad, y su ayuda ha sido decisiva para superar esa dificultad.

Resumen

La Informática Gráfica es la disciplina que pretende simular el comportamiento de las cámaras fotográficas, generando imágenes por ordenador con efectos realistas de iluminación, materiales y geometría. Existen diversas técnicas de renderizado para generar estas imágenes, muchas de las cuales aplican una versión simplificada de las leyes físicas para reducir el coste computacional. Una de estas simplificaciones es asumir que la luz se propaga a velocidad infinita, ya que en las cámaras convencionales el tiempo de exposición es muchos órdenes de magnitud mayor que el tiempo que tarda la luz en propagarse. Recientes avances han dado lugar a sensores ultrarrápidos capaces de capturar la propagación de la luz al tener tiempos de exposición comparables a los de la velocidad de la luz. Dados estos avances surge el campo del renderizado transitorio, que ya no asume que la luz se propaga a velocidad infinita, cuyo objetivo es simular la propagación de la luz por la escena fotograma a fotograma, simulando estos sensores de captura.

Para simular el transporte de luz, tanto en estado estacionario como transitorio, se utilizan métodos de Monte Carlo, debido a la complejidad de las ecuaciones. Sin embargo, la naturaleza estocástica de Monte Carlo provoca que los resultados siempre tengan ruido. Este problema es aún mayor en renderizado transitorio: las muestras deben distribuirse no solo en el espacio, sino también a través del tiempo.

Para reducir este ruido, los métodos de renderizado profesionales incorporan técnicas de *denoising*, que combinan información de vecinos en el espacio para reducir la varianza. Sin embargo, estos métodos introducen sesgo, manifestándose como difuminado de detalles finos como reflejos, sombras o cáusticas. Los enfoques más recientes, basados en redes neuronales profundas, pueden producir resultados visualmente agradables, pero no garantizan convergencia a la solución correcta, pueden generar alucinaciones y requieren grandes conjuntos de datos de entrenamiento que son especialmente costosos de generar para renderizado transitorio.

Recientemente, se propuso un *framework* de *denoising* estadístico para renderizado Monte Carlo que, mediante el análisis de estadísticas recolectadas durante el proceso de renderizado, garantiza un compromiso casi óptimo entre sesgo y varianza. Este método asegura convergencia a la solución correcta sin necesidad de entrenar redes neuronales con grandes conjuntos de datos, que requieren una gran cantidad de tiempo de entrenamiento en GPU. En este trabajo implementamos dicho *framework* en Mitsuba 3, un *software* de renderizado orientado a la investigación.

La principal contribución de este trabajo es la extensión de este *framework* al dominio transitorio, incorporando *kernels* espacio-temporales que explotan la correlación espacio-temporal de la propagación de la luz. Además, desarrollamos un algoritmo eficiente en memoria para la recolección de estadísticas transitorias mediante reutilización de caminos. Los resultados demuestran que el método propuesto reduce efectivamente el ruido en renderizado transitorio preservando detalles finos, superando en fiabilidad y reducción de ruido a métodos basados en redes neuronales.

Nuestra extensión del *framework* de *denoising* al dominio transitorio no genera alucinaciones (a diferencia de los métodos basados en redes neuronales), garantiza convergencia del resultado *denoiseado* al aumentar el número de muestras, es eficiente, no requiere entrenamiento y maneja naturalmente medios participativos. Además, dado que las simulaciones de Monte Carlo son una técnica bien establecida y se han derivado numerosas estrategias de muestreo específicas para renderizado transitorio, esperamos que nuestro trabajo pueda aplicarse en muchos otros campos como detección heterodina óptica, imagen fuera de visión, microscopía de absorción, renderizado acústico o imagen térmica.

Los resultados de este trabajo se han utilizado para un artículo enviado a Eurographics 2026, una conferencia de referencia en el ámbito de los gráficos por computador en Europa.

Para ver los resultados en vídeo: [vídeo de resultados del denoiser estadístico transitorio](#).

Índice general

1. Introducción y objetivos	1
1.1. Contexto y motivación	1
1.2. Objetivos y Metodología	4
1.3. Organización del documento	5
2. Marco Teórico	6
2.1. Transporte de luz estacionario	6
2.1.1. Ecuación de render	6
2.1.2. Integración Monte Carlo	7
2.1.3. Path Tracing	8
2.2. Transporte de luz transitorio	9
2.2.1. Retos del transporte transitorio	10
2.2.2. Path reuse	10
3. Denoising	12
3.1. Métodos de denoising	13
3.1.1. Métodos clásicos	13
3.1.2. Métodos basados en aprendizaje	13
3.1.3. Enfoque estadístico	14
3.2. Denoising estadístico	14
4. Denoising estacionario	19
4.1. Recolección de estadísticas	19
4.2. Denoising con PyTorch	21
4.2.1. Entrada y vectorización	21
4.2.2. Joint Bilateral Filter	21
4.2.3. Membresías	22
4.2.4. Teselado	24
4.3. Resultados en estado estacionario	25
5. Denoising transitorio	27
5.1. Renderizado transitorio con estadísticas	28
5.2. Obtención eficiente de las estadísticas	28
5.3. Denoising estadístico	30
5.4. Resultados	32
5.4.1. Análisis de parámetros	33
5.4.2. Resultados volumétricos	35
6. Conclusiones y trabajo futuro	37
Anexos	45

Anexo A. Rendimiento	47
A.1. Rendimiento de la recolección de estadísticas	47
A.2. Rendimiento del denoising	47
A.3. Conclusiones sobre rendimiento	48
Anexo B. Gestión del Proyecto	49

Capítulo 1

Introducción y objetivos

1.1. Contexto y motivación

La informática gráfica pretende simular el transporte de luz que observamos en la vida real, reproduciendo los fenómenos físicos que ocurren en la naturaleza para generar imágenes sintéticas fieles a la realidad. Existen diversas aproximaciones para simular la realidad que se diferencian en su fidelidad a las leyes físicas: cuanto más fieles son a la realidad, mejores resultados producen, pero también requieren mayor tiempo computacional. Algunos de estos enfoques de realismo físico son, ordenados de mayor a menor precisión física: óptica cuántica, óptica ondulatoria y óptica geométrica. Esta última es una de las más utilizadas debido a su menor complejidad computacional y a que produce resultados comparables a los otros enfoques para la mayoría de aplicaciones prácticas.

En este ámbito nace el concepto de *rendering* o renderizado, donde se pretende simular las cámaras fotográficas que tenemos en la vida real. En renderizado trabajamos con escenas compuestas por figuras geométricas, materiales, fuentes de luz y un sensor o cámara virtual. El objetivo del renderizado es simular cómo funciona una cámara real para poder obtener imágenes realistas generadas por ordenador.

El renderizado se basa fundamentalmente en la simulación del transporte de luz. Simula cómo los rayos de luz se propagan en la vida real, rebotando con diferentes objetos hasta llegar a un sensor, que en el caso de la vida real serían nuestros ojos o una cámara fotográfica.

Cuando observamos una escena, cada punto del sensor (cada píxel en una cámara digital, o cada célula fotorreceptora en nuestros ojos) recibe luz que puede provenir de infinitas direcciones diferentes y haber seguido infinitos caminos posibles a través de la escena. Por ejemplo, la luz que llega a un único píxel puede venir directamente de una fuente de luz, puede haber rebotado una vez en una pared, puede haber rebotado múltiples veces en diferentes superficies, o puede haber atravesado materiales transparentes como el vidrio. Cada uno de estos caminos aporta una contribución diferente al color e intensidad final que percibimos en ese píxel.

Para calcular el valor correcto de cada píxel, sería necesario considerar todos estos infinitos caminos de luz posibles y sumar todas sus contribuciones. Este problema se conoce como la *ecuación de render*, y matemáticamente se expresa como una integral multidimensional: debemos integrar sobre toda la superficie del sensor (todos los píxeles), sobre todas las posibles direcciones desde las que puede llegar luz a cada píxel, y sobre todos los posibles caminos que esa luz puede haber seguido por la escena.

Esta integral no puede resolverse de manera analítica (es decir, no podemos escribir una fórmula cerrada que nos dé la solución exacta) en escenas con geometrías y materiales complejos. Por esta razón, se recurre a métodos de Monte Carlo para aproximar la solución. Estos métodos funcionan mediante muestreo estocástico: en lugar de calcular todos los infinitos caminos de luz posibles, se generan aleatoriamente un número finito de caminos de luz, y se calcula el promedio de sus contribuciones. Cuantas más muestras se generen, más precisa será la aproximación a la solución real, aunque esto también incrementa el tiempo de cómputo necesario.

En aplicaciones prácticas del renderizado se suele asumir que la luz se propaga a velocidad infinita. Esta aproximación es válida porque en las cámaras convencionales el tiempo de exposición es muchos

órdenes de magnitud mayor que el tiempo que tarda la luz en atravesar la escena, por lo que estas no pueden captar la propagación de la propia luz. Sin embargo, con el desarrollo de sensores ultrarrápidos capaces de capturar la luz con tiempos de exposición del orden de picosegundos o femtosegundos, surge el concepto de femto-fotografía (ver Figura 1.1), donde el sensor es capaz de registrar el transporte temporal de la luz.

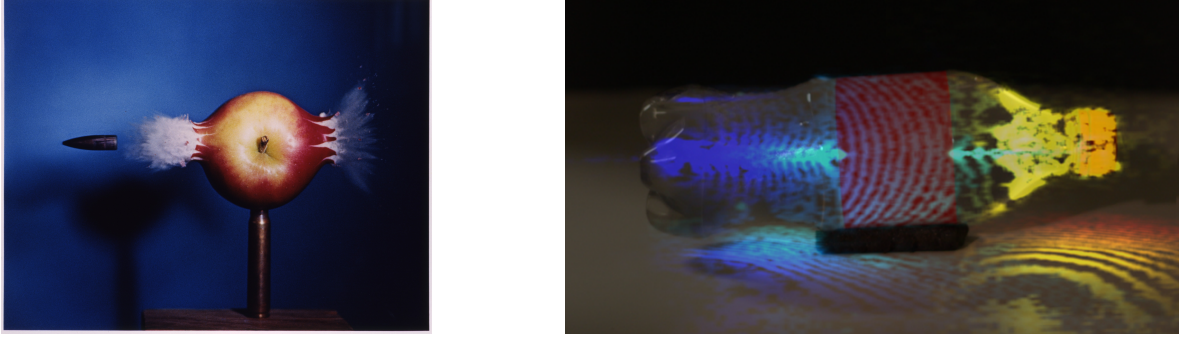


Figura 1.1: Femto-fotografía. A la izquierda, la icónica imagen de Harold Edgerton, donde una bala a 850 m/s atraviesa una manzana, equivalente a una tasa aproximada de captura de un millón de fotogramas por segundo ($\sim 10^6 \text{ fps}$). A la derecha, la técnica de femto-fotografía [35], realizada con un sistema capaz de registrar en torno a un billón de fotogramas por segundo ($\sim 10^{12} \text{ fps}$), permitiendo visualizar la propagación de la luz en tiempo real.

Con estos sensores nace el concepto de renderizado transitorio, donde ya no se asume que la luz se transporta instantáneamente, sino que se propaga a velocidad finita y los sensores tienen un tiempo de exposición lo suficientemente bajo como para capturar la propagación de la luz por la escena fotograma a fotograma. El renderizado transitorio permite capturar un video donde se puede observar la propagación de la luz. Para ello es necesario añadir una nueva dimensión temporal a la *ecuación de render*, así como los diferentes retardos temporales de la propagación de la luz.

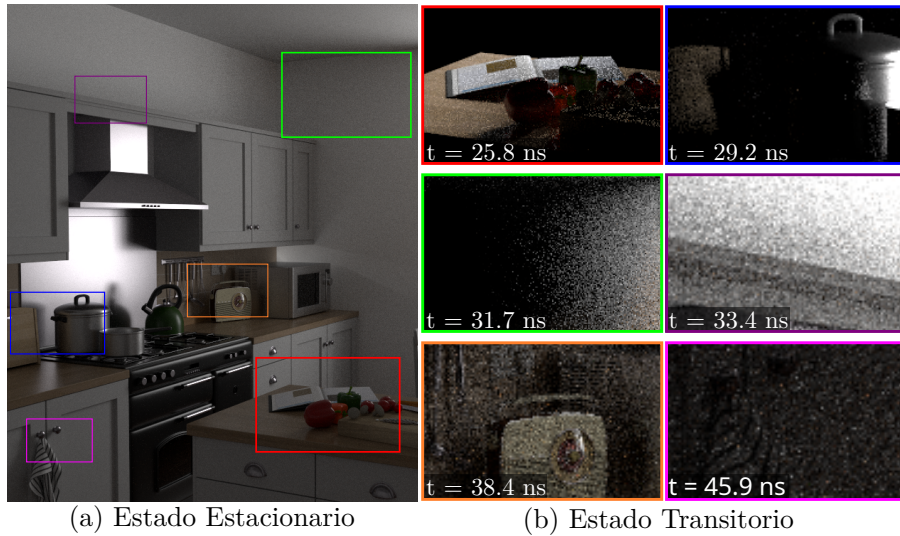


Figura 1.2: Diferencia de ruido entre renderizado estacionario y transitorio. Comparación de la misma escena renderizada con el mismo número de muestras de forma (a) estacionaria y (b) transitoria. Se observa que, a pesar de utilizar el mismo número de muestras, el renderizado transitorio presenta significativamente mayor nivel de ruido debido a la distribución de las muestras a través de la nueva dimensión temporal.

El problema de los métodos de Monte Carlo es que generan ruido en los resultados finales: cuantas menos muestras se utilicen, mayor será el ruido. Por esta razón, una de las soluciones clásicas en los *pipelines* de renderizado es aplicar un paso de *denoising* al final del proceso. Los *denoisers* analizan

los píxeles vecinos y los combinan para reducir el ruido, aprovechando la estructura espacial: píxeles cercanos en el espacio es probable que tengan valores similares.

En renderizado transitorio, este problema de ruido es aún mayor, ya que se necesita integrar en una dimensión adicional (la temporal), lo que distribuye las muestras a través del tiempo y genera resultados con niveles de ruido significativamente más elevados. En la Figura 1.2 se puede observar esta diferencia: la misma escena renderizada de forma estacionaria (Figura 1.2a) presenta mucho menos ruido que su versión transitoria (Figura 1.2b), a pesar de utilizar el mismo número de muestras totales. Esto se debe a que en el caso transitorio, las muestras se distribuyen a través de múltiples fotogramas temporales.

Como se ha mencionado, los *denoisers* combinan los píxeles vecinos, lo que reduce considerablemente la varianza pero puede añadir sesgo si se combinan píxeles con valores diferentes. Con el tiempo, este campo ha evolucionado y se han desarrollado *denoisers* cada vez más sofisticados.

Actualmente, la mayoría de los *denoisers* considerados estado del arte están basados en redes neuronales. Estos sistemas se entrenan con múltiples renderizados de diferentes escenas, utilizando renderizados con bajo número de muestras (ruidosos) como entrada y renderizados con alto número de muestras (prácticamente sin ruido) como salida deseada. De esta forma, la red neuronal aprende a transformar imágenes ruidosas en imágenes sin ruido. Estos sistemas son en la práctica muy útiles ya que pueden resolver un problema complejo, aunque lo hacen a través de modelos de gran complejidad con millones de parámetros cuyo funcionamiento interno resulta difícil de interpretar o explicar. Es decir, funcionan como una caja negra: sabemos qué entrada les damos y qué salida producen, pero no podemos entender exactamente cómo toman sus decisiones.

Sin embargo, estos enfoques presentan limitaciones importantes. En primer lugar, el proceso de entrenamiento es computacionalmente muy costoso, ya que requiere generar miles de pares de imágenes de diversas escenas. Más importante aún, estos métodos no garantizan resultados correctos: las redes neuronales pueden funcionar muy bien en escenas similares a las utilizadas durante el entrenamiento, pero cuando se enfrentan a escenas muy diferentes, pueden comenzar a alucinar o inventar detalles que no existen realmente en la escena, sin que podamos predecir o controlar cuándo ocurrirá esto. Este problema se agrava especialmente en renderizado transitorio, ya que la mayoría de estas redes están entrenadas únicamente para renderizado estacionario, y los datos de entrenamiento para renderizado transitorio son escasos y costosos de generar debido a la dimensión temporal adicional.

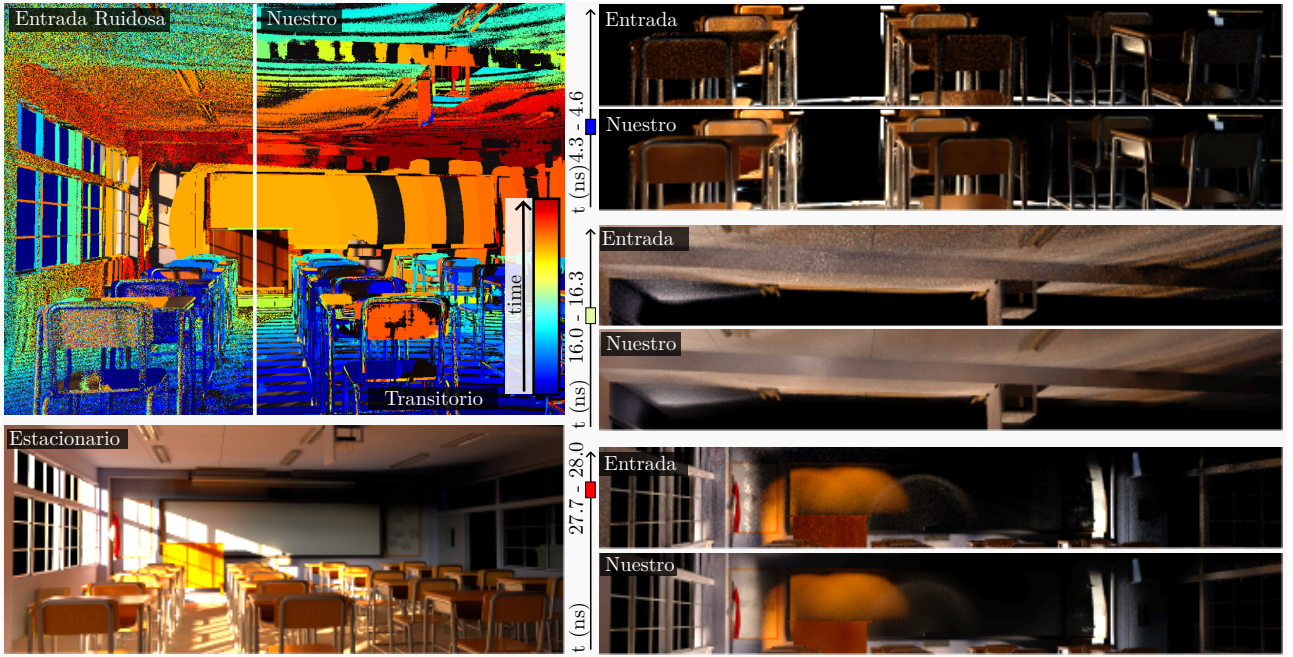


Figura 1.3: Denoising de renderizado transitorio. El renderizado transitorio simula la luz en movimiento a través de una escena (arriba izquierda). Sin embargo, la naturaleza estocástica del renderizado Monte Carlo, combinada con la dimensión temporal añadida, produce resultados extremadamente ruidosos (arriba izquierda, entrada ruidosa). Se realiza el *denoising* mediante el análisis espacio-temporal de las estadísticas del transporte de luz transitorio (arriba izquierda, resultado propuesto). El *denoiser* propuesto no requiere entrenamiento y logra una reducción notable del ruido preservando detalles espacio-temporales finos (derecha).

Para abordar estas limitaciones proponemos un *framework* de *denoising* transitorio (ver Figura 1.3) que no genera alucinaciones, garantiza convergencia del resultado *denoiseado* con el aumento del número de muestras, es eficiente y no requiere entrenamiento. Además, dado que las simulaciones de Monte Carlo son una técnica bien establecida y se han derivado numerosas estrategias de muestreo específicas para renderizado transitorio, esperamos que nuestro trabajo pueda aplicarse en muchos otros campos como detección heterodina óptica [20], imagen fuera de visión [27], microscopía de absorción [19], renderizado acústico [8] o imagen térmica [33].

Para ver los resultados del *denoiser* estadístico transitorio propuesto, puede consultar un vídeo comparando un render transitorio estándar con nuestro resultado en Google Drive¹.

1.2. Objetivos y Metodología

El objetivo de este trabajo es implementar el *denoiser* estadístico para renderizado Monte Carlo estacionario propuesto por Sakai et al. [30], quienes proponen un *denoiser* basado en estadísticas que resuelve los problemas de otros *denoisers*: no necesita datos de entrenamiento, asegura un compromiso óptimo entre sesgo y varianza, garantiza resultados correctos y converge a la solución correcta.

El objetivo principal, una vez implementado este *framework* en estado estacionario, es extenderlo al estado transitorio junto con el uso de *kernels* espacio-temporales. La realización de este trabajo se divide en varias etapas:

1. **Estudio y formación:** Estudio del renderizado transitorio [16], del *denoiser* estadístico [30] y de las tecnologías a utilizar, siendo estas principalmente Mitsuba 3 [15], mitransient [28] y PyTorch [24].
2. **Implementación en renderizado estacionario:**

¹<https://drive.google.com/file/d/1MNDdzgInd5YBHJlZCzm18ZRHmBEoWxxl/view?usp=sharing>

- a) Implementación en Mitsuba 3 del recolector de estadísticas.
- b) Implementación en PyTorch del *denoiser* (convoluciones, *kernels* y operaciones de filtrado).
- c) Evaluación del *denoiser* con diversas escenas y comparación con otros *denoisers* estado del arte.

3. Extensión a renderizado transitorio:

- a) Formulación del *denoiser* para renderizado transitorio.
- b) Extensión de mitransient para la recolección de estadísticas transitorias.
- c) Extensión del *denoiser* en PyTorch para trabajar con 3 dimensiones (espacio-temporal) en lugar de 2 dimensiones (espacial).
- d) Evaluación con diversas escenas y comparación con otros *denoisers* estado del arte.

4. Documentación: Redacción de la memoria del trabajo.

1.3. Organización del documento

El documento se estructura en seis Capítulos. El Capítulo 1 presenta el contexto, la motivación, los objetivos y la metodología del trabajo. El Capítulo 2 establece las bases teóricas sobre renderizado estacionario y renderizado transitorio, incluyendo los fundamentos de la ecuación de transporte de luz y los métodos de Monte Carlo. El Capítulo 3 establece las bases de *denoising*, revisa los últimos *denoisers* estado del arte y presenta la formulación del *denoiser* estadístico [30]. El Capítulo 4 desarrolla la implementación del *denoiser* en renderizado estacionario utilizando Mitsuba 3 [15] y PyTorch [24], y evalúa su rendimiento en diferentes escenas frente a otros *denoisers*. El Capítulo 5 extiende la formulación del *denoiser* estadístico al estado transitorio, detallando los problemas y soluciones abordados en dicha extensión. Se comparan los resultados con otros *denoisers* estado del arte en diferentes escenas, se realiza un análisis exhaustivo de los parámetros del *denoiser* y su influencia en el rendimiento, y se explora su funcionamiento en renderizado volumétrico transitorio. Finalmente, el Capítulo 6 resume las conclusiones del trabajo y propone líneas de investigación futuras.

Capítulo 2

Marco Teórico

En este capítulo se exploran los fundamentos del renderizado y el transporte de luz en estado estacionario (Sección 2.1), así como su extensión a la dimensión temporal en estado transitorio (Sección 2.2).

2.1. Transporte de luz estacionario

El renderizado tiene como objetivo simular el transporte de luz en una escena para generar imágenes sintéticas realistas. Este proceso se fundamenta en la *ecuación de render*, que modela matemáticamente cómo la luz interactúa con las superficies y materiales de la escena. En esta Sección se presenta la *ecuación de render* y su formulación mediante la integral de caminos, así como los métodos de Monte Carlo utilizados para resolverla numéricamente, en particular el algoritmo de *path tracing*.

2.1.1. Ecuación de render

El renderizado de imágenes se basa en la ecuación propuesta por Kajiya [17], conocida como la *ecuación de render*, que describe cómo la luz interactúa con las superficies en una escena. Esta ecuación, fundamentada en la óptica geométrica y en particular en la óptica de rayos, asume que la luz se propaga en línea recta, simplificación que resulta válida para la mayoría de aplicaciones prácticas de renderizado.

La *ecuación de render* describe la radiancia saliente $L_o(\mathbf{x}, \omega_o)$ (ver Figura 2.1) como

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\Omega} L_i(\mathbf{x}, \omega_i) f_r(\mathbf{x}, \omega_i, \omega_o) |\omega_i \cdot \mathbf{n}| d\omega_i, \quad (2.1)$$

donde $\mathbf{x}$ representa un punto en una superficie, ω_o es la dirección de salida, ω_i representa las posibles direcciones de incidencia de la luz, $\mathbf{n}$ es la normal de la superficie en $\mathbf{x}$, y Ω es el hemisferio centrado en $\mathbf{n}$, que representa todas las direcciones posibles desde las cuales puede llegar luz al punto $\mathbf{x}$.

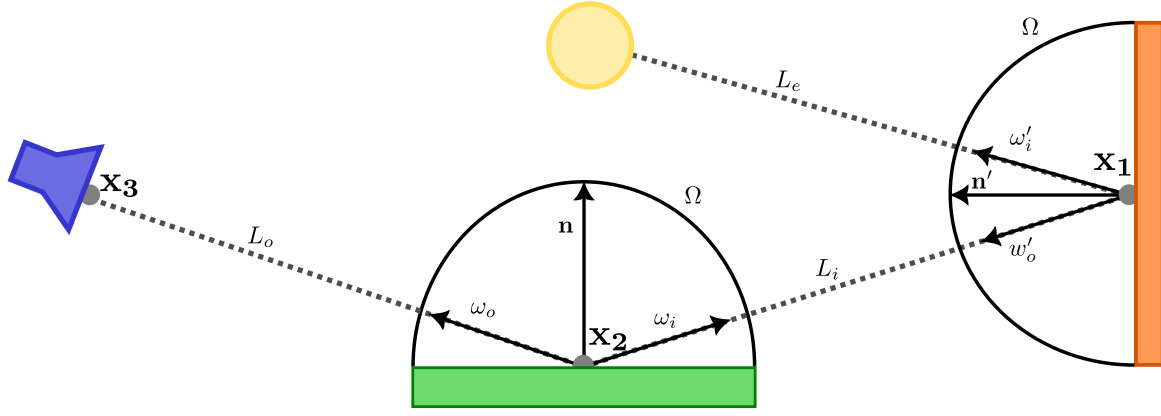


Figura 2.1: Ecuación de render. Representación visual de la *ecuación de render* de Kajiya, que describe la radiancia saliente $L_o(\mathbf{x}, \omega_o)$ en la dirección de la cámara. Esta radiancia es la suma de la radiancia emitida por el propio material, $L_e(\mathbf{x}, \omega_o)$, y de la radiancia incidente L_i integrada sobre todas las direcciones del hemisferio Ω alineado con la normal $\mathbf{n}$ en el punto $\mathbf{x}$. En la Figura se ilustra también el proceso de *path tracing*: el camino de luz realiza una única muestra por rebote y acumula contribuciones a través de las reflexiones sucesivas hasta alcanzar el sensor. En este ejemplo, el camino tiene tres rebotes: dos interacciones con materiales difusos y un último rebote en el sensor.

El primer término, $L_e(\mathbf{x}, \omega_o)$, representa la radiancia emitida directamente por el punto $\mathbf{x}$ en la dirección ω_o . Este término es no nulo únicamente si $\mathbf{x}$ pertenece a una fuente de luz. El otro término representa la luz reflejada, calculada como la integral de todas las contribuciones de luz incidente desde diferentes direcciones.

La radiancia incidente $L_i(\mathbf{x}, \omega_i)$ representa la luz que llega al punto $\mathbf{x}$ desde la dirección ω_i . El término $f_r(\mathbf{x}, \omega_i, \omega_o)$ es la función de distribución de reflectancia bidireccional (BRDF, *Bidirectional Reflectance Distribution Function*), que describe cómo se dispersa la luz incidente en $\mathbf{x}$ desde la dirección ω_i hacia la dirección ω_o , modelando las propiedades del material (color, reflectancia, etc.). El término geométrico $|\mathbf{n} \cdot \omega_i|$ (coseno del ángulo entre la normal y la dirección incidente) atenúa la intensidad en función del ángulo de incidencia: físicamente, una luz que incide perpendicularmente sobre una superficie aporta más energía por unidad de área que una luz que incide con un ángulo más oblicuo.

La *ecuación de render* presenta una recursividad inherente. Observemos que la radiancia incidente $L_i(\mathbf{x}, \omega_i)$ en un punto $\mathbf{x}$ es equivalente a la radiancia saliente de otro punto $\mathbf{x}'$ situado en la dirección $-\omega_i$

$$L_i(\mathbf{x}, \omega_i) = L_o(\mathbf{x}', -\omega_i), \quad (2.2)$$

donde $\mathbf{x}'$ es el punto más cercano intersectado por el rayo lanzado desde $\mathbf{x}$ en dirección ω_i . Por tanto, para calcular $L_i(\mathbf{x}, \omega_i)$, es necesario resolver nuevamente la *ecuación de render* en $\mathbf{x}'$, lo que genera una recursión que potencialmente involucra infinitos rebotes de luz en la escena. Esta recursividad, junto con la complejidad geométrica de las escenas y la variedad de materiales, hace que la ecuación no tenga solución analítica en casos prácticos. La asunción de propagación rectilínea de la luz es fundamental para poder calcular eficientemente estos puntos de intersección $\mathbf{x}'$ mediante trazado de rayos.

2.1.2. Integración Monte Carlo

Dado que la *ecuación de render* no puede resolverse analíticamente para escenas complejas, se recurre a métodos numéricos. Aunque existen diversos enfoques para resolver esta ecuación, los métodos basados en Monte Carlo se han convertido en los más utilizados.

El método de Monte Carlo permite aproximar integrales mediante muestreo estocástico. Para una integral general

$$I = \int_{\Omega} f(x) dx, \quad (2.3)$$

el estimador de Monte Carlo con N muestras viene dado por

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)}, \quad (2.4)$$

donde x_i son muestras generadas aleatoriamente según una función de densidad de probabilidad $p(x)$.

Aplicado a la *ecuación de render*, Monte Carlo se utiliza para aproximar la integral sobre el hemisferio Ω . En lugar de integrar sobre todas las direcciones posibles ω_i , se muestrean N direcciones aleatorias ω_i según alguna distribución de probabilidad $p(\omega_i)$, y se promedian sus contribuciones ponderadas como

$$L_o(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{i=1}^N \frac{L_i(\mathbf{x}, \omega_i) f_r(\mathbf{x}, \omega_i, \omega_o) |\omega_i \cdot \mathbf{n}|}{p(\omega_i)}. \quad (2.5)$$

La elección de $p(\omega_i)$ es crucial para la eficiencia del algoritmo. Una distribución común es el muestreo proporcional al coseno, donde la probabilidad de muestrear una dirección es proporcional al ángulo que forma con la normal de la superficie: direcciones más perpendiculares a la superficie tienen mayor probabilidad de ser muestreadas que direcciones más oblicuas.

El problema de ruido inherente a Monte Carlo surge precisamente de esta aproximación estocástica: con pocas muestras N , el estimador presenta alta varianza, manifestándose visualmente como ruido en la imagen renderizada. Conforme aumenta N , el ruido decrece, pero el tiempo de cómputo crece linealmente con N .

2.1.3. Path Tracing

La recursividad de la *ecuación de render* plantea un desafío computacional adicional. Una aproximación ingenua consistiría en aplicar Monte Carlo recursivamente en cada rebote, muestreando N direcciones en cada punto de intersección. Sin embargo, esto resulta en un crecimiento exponencial del número de rayos: N^k rayos para k rebotes, haciendo el método impráctico incluso para escenas simples.

Kajiya [17] propuso el algoritmo de *path tracing* (ver Figura 2.1) para resolver este problema. La idea fundamental es trazar un único camino aleatorio por muestra: desde la cámara se lanza un rayo que rebota en las superficies de la escena siguiendo la BRDF del material, hasta que alcanza una fuente de luz o es absorbido. En cada rebote, en lugar de muestrear múltiples direcciones, se muestrea una única dirección ω_i de forma estocástica. Esto transforma el crecimiento exponencial en lineal: para obtener N muestras independientes, se trazan N caminos completos, cada uno con crecimiento lineal en el número de rebotes.

Esta reformulación conduce a la conocida como integral de caminos (*path integral*) [34], que expresa la intensidad de luz I que llega a un píxel de la cámara como una integral sobre todos los posibles caminos de luz

$$I = \int_{\Omega} f(\bar{\mathbf{x}}) d\mu(\bar{\mathbf{x}}), \quad (2.6)$$

donde Ω representa el espacio de todos los caminos posibles desde la cámara, y $\bar{\mathbf{x}} = (x_0, x_1, \dots, x_k)$ representa un camino completo de luz con $k+1$ vértices. El vértice x_0 pertenece al sensor de la cámara, los vértices $x_1, \dots, x_{k-1}$ son puntos de intersección intermedios en las superficies de la escena, y x_k es el punto donde termina el camino, ya sea por intersección con una fuente de luz o por absorción.

La función $f(\bar{\mathbf{x}})$ representa la contribución de un camino $\bar{\mathbf{x}}$ a la intensidad del píxel y se expresa como

$$f(\bar{\mathbf{x}}) = L_e(x_k \rightarrow x_{k-1}) \left(\prod_{j=1}^{k-1} f_r(x_j) G(x_j \leftrightarrow x_{j+1}) \right) W_e(x_1 \rightarrow x_0), \quad (2.7)$$

donde $L_e(x_k \rightarrow x_{k-1})$ es la radiancia emitida por la fuente de luz en x_k hacia x_{k-1} , $f_r(x_j)$ es la BRDF en cada punto de rebote intermedio, $G(x_j \leftrightarrow x_{j+1})$ es el término geométrico que modela la atenuación

por distancia y ángulo entre vértices consecutivos, y $W_e(x_1 \rightarrow x_0)$ es la función de importancia del sensor que describe la sensibilidad de la cámara.

El estimador de Monte Carlo para la integral de caminos con N caminos muestreados viene dado por

$$\langle I \rangle = \frac{1}{N} \sum_{j=1}^N \frac{f(\bar{\mathbf{x}}_j)}{p(\bar{\mathbf{x}}_j)}, \quad (2.8)$$

donde $p(\bar{\mathbf{x}}_j)$ es la probabilidad de generar el camino $\bar{\mathbf{x}}_j$ mediante el proceso de muestreo estocástico. Cada camino se construye iterativamente: desde la cámara se lanza un rayo, en cada intersección se muestrea una dirección de rebote según la BRDF del material, y el proceso continúa hasta alcanzar una fuente de luz o hasta que el camino sea terminado estocásticamente mediante ruleta rusa.

Este enfoque, aunque conceptualmente simple, genera imágenes con ruido que decrece lentamente con el número de muestras. El *denoising* (ver Capítulo 3) surge como técnica esencial para obtener imágenes de calidad aceptable con un número razonable de muestras.

2.2. Transporte de luz transitorio

En la Sección anterior se ha presentado el método de *path tracing* para transporte de luz estacionario. Este método da por hecho que la luz se transporta a una velocidad infinita y, por tanto, la radianza final de la escena no depende del dominio temporal. La razón de esta asunción se debe a que la mayoría de sensores con los que trabajamos en la realidad no son lo suficientemente rápidos como para captar la relevancia del dominio temporal, ya que la luz se propaga a una velocidad muy superior a la que el sensor es capaz de capturar la radianza que le llega. Con el paso del tiempo se han desarrollado sensores en la vida real que son capaces de capturar la radianza en velocidades tan altas como la de la luz [35, 13] y a raíz de esto ha surgido el interés de explorar este caso también en el mundo de informática gráfica. Este trabajo se desarrolla a partir del framework de transporte de luz transitorio de Jarabo et al. [16], que establece la base teórica del transporte de luz en estado transitorio. Jarabo et al. expanden la *path integral* al dominio temporal redefiniendo la integral

$$I = \int_{\Omega} \int_{\Delta T} f(\bar{x}, \overline{\Delta t}) d\mu(\overline{\Delta t}) d\mu(\bar{x}), \quad (2.9)$$

donde $\overline{\Delta t} = \Delta t_0, \dots, \Delta t_k$ define la secuencia de retrasos de tiempo asociados a cada vértice del camino $\bar{x}$. Esta ecuación incorpora respecto a la Ecuación (2.6) el nuevo dominio temporal de integración, $\int_{\Delta T}$, así como la correspondiente componente temporal en la función de contribución del camino, $f(\bar{x}, \overline{\Delta t})$. La función de contribución del camino se define como la original, pero con la emisión L_e , el *path throughput* $\mathfrak{T}$, y la importancia del sensor W_e ahora dependiendo de la dimensión temporal

$$f(x, \Delta t) = L_e(x_0 \rightarrow x_1, \Delta t_k) \mathfrak{T}(x, \Delta t) W_e(x_{k-1} \rightarrow x_k, \Delta t_k). \quad (2.10)$$

Ahora el sensor temporal de importancia W_e no solo define características espaciales como la sensibilidad angular, también es capaz de definir la región temporal que estamos interesados en evaluar, por tanto podemos especificar un intervalo finito de tiempo a diferencia del transporte estacionario donde el intervalo de tiempo sería infinito. Así mismo el parámetro temporal del término de emisión L_e nos permite definir variaciones de este en el espacio temporal (p. ej., pulsos de distinta duración). El *path throughput* se redefine

$$T(x, \Delta t) = \left[\prod_{i=1}^{k-1} f_r(x_i, \Delta t_i) \right] \left[\prod_{i=0}^{k-1} G(x_i, x_{i+1}) V(x_i, x_{i+1}) \right]. \quad (2.11)$$

Dado que se asume que la geometría es estacionaria frente a la velocidad de la luz, el término geométrico $G(x_i, x_{i+1})$ y el término de visibilidad $V(x_i, x_{i+1})$ dependen únicamente de las coordenadas espaciales del camino. En cambio, el término de reflectancia del material (BSDF f_r) se extiende incluyendo un

parámetro de retraso temporal Δt_i para modelar el posible retraso en la dispersión de la luz en el vértice x_i . Este tipo de retardos pueden ocurrir por múltiples reflexiones internas en micro-geometría [39], desplazamientos de fase electromagnética en las ecuaciones de Fresnel [10, 29], o efectos de dispersión inelástica como la fluorescencia [40, 11].

Un camino transitorio de luz se define por las coordenadas espaciales y temporales de sus vértices. Las coordenadas temporales de cada vértice x_i son t_i^- , el momento exacto antes de que se produzca el evento de dispersión entre la luz y la materia, y t_i , el momento exacto después del evento de dispersión. Estas coordenadas temporales se pueden calcular teniendo en cuenta los retrasos de propagación $t(x_i \leftrightarrow x_{i+1})$, tiempo que tarda la luz en recorrer la distancia espacial entre dos vértices del camino, y los retrasos de dispersión Δt_i que produce un material en la luz

$$t_i^- = \sum_{j=0}^{i-1} (t(x_j \leftrightarrow x_{j+1}) + \Delta t_j), \quad t_i = t_i^- + \Delta t_i, \quad (2.12)$$

donde t_0 y t_k representan las coordenadas temporales de emisión y captura del camino. El tiempo de propagación entre vértices $t(x_0 \leftrightarrow x_k)$, dada la asunción de que el segmento entre ambos vértices del camino tenga un índice de refracción η constante es calculable mediante

$$t(x_i \leftrightarrow x_{i+1}) = \frac{\eta |x_i - x_{i+1}|}{c}, \quad (2.13)$$

con $c = 299,792,458 \text{ m s}^{-1}$, la velocidad de la luz en el vacío.

Como ocurre en el transporte de luz estacionario, la *path integral* no se puede resolver de manera analítica, por lo que se usa un estimador de Monte Carlo para aproximar numéricamente el resultado

$$\langle I \rangle = \frac{1}{n} \sum_{j=1}^n \frac{f(x_j, \Delta t_j)}{p(x_j, \Delta t_j)}, \quad (2.14)$$

ahora como se incorpora la dimensión transitoria, es necesario muestrear el espacio temporal.

2.2.1. Retos del transporte transitorio

La incorporación del dominio temporal en el transporte de luz introduce una serie de retos que complican la extensión directa de los algoritmos de *path tracing* empleados en el caso estacionario. El principal de ellos consiste en que las muestras caigan dentro de la ventana temporal de interés. Esto resulta especialmente difícil porque los retrasos temporales asociados a la propagación dependen tanto de la escena como del muestreo por importancia espacial, de modo que el tiempo total del camino t_k difícilmente coincide con el intervalo temporal Δt que se desea evaluar.

Para abordar este problema se han propuesto diferentes técnicas. Entre las más empleadas se encuentran: el uso de un *kernel* temporal para reconstruir la señal, lo que introduce cierto sesgo pero permite reducir el ruido, y la reutilización de caminos (*path reuse*). Además de estas técnicas se puede hacer posteriormente de forma complementaria el proceso de *denoising*, siendo esta la técnica en la que se centra este trabajo.

2.2.2. Path reuse

En transporte de luz transitorio, el término de importancia del sensor W_e permite definir explícitamente el *bin* temporal (ventana temporal) que se desea muestrear. A diferencia del caso estacionario, esto posibilita generar una secuencia de fotogramas correspondientes a *bins* temporales consecutivos, de manera análoga a cómo una cámara convencional graba un vídeo mediante exposiciones de duración τ . En el caso transitorio, cada fotograma abarca un *bin* temporal de tamaño τ , y el término de importancia del sensor se redefine para cada fotograma j como

$$W_e^{(j)}(x_{k-1} \rightarrow x_k, t_k) = \begin{cases} 1 & \text{si } t_0 + j \cdot \tau \leq t_k \leq t_0 + (j+1) \cdot \tau, \\ 0 & \text{en otro caso,} \end{cases} \quad (2.15)$$

donde t_0 representa el instante a partir del cual el sensor comienza a capturar información.

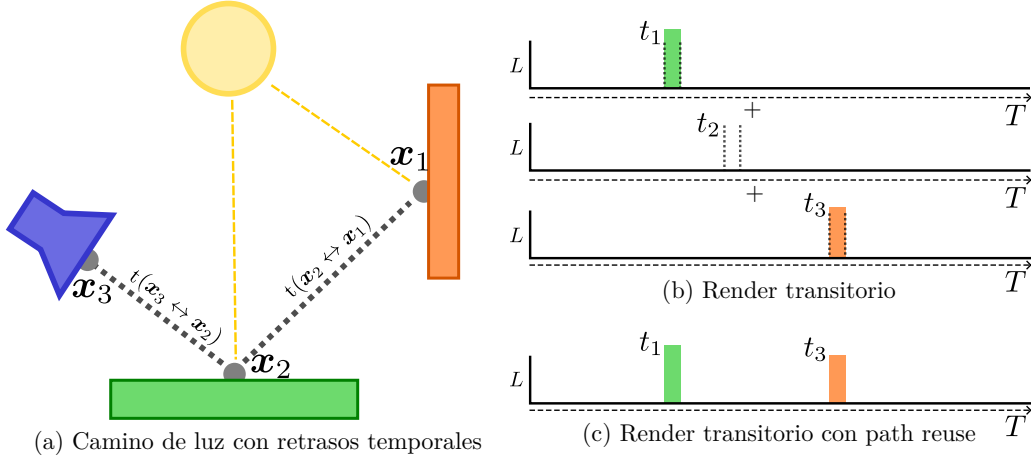


Figura 2.2: Renderizado transitorio y reutilización de caminos. (a) El renderizado transitorio tiene en cuenta los retardos temporales de la luz entre los vértices del camino x_i . (b) El renderizado transitorio convencional repite el renderizado T veces (número de *bins* temporales), controlando el sensor para cada bin temporal t_i , descartando los caminos de luz cuyo retardo temporal $\bar{t}$ no está contenido dentro del *bin* controlado, lo que implica repetir el proceso de renderizado T veces. (c) El renderizado transitorio con reutilización de caminos almacena la contribución L de cada camino de luz en el *bin* temporal t_i correspondiente a su retardo temporal $\bar{t}$, renderizando una sola vez para todo el dominio transitorio.

Si se aplicasen directamente las técnicas de muestreo del caso estacionario, en las que en cada rebote se asigna la contribución a un único *bin* temporal, sería necesario repetir el proceso de renderizado un número T de veces (una por cada *bin* temporal, ver Figura 2.2b). La técnica de *path reuse* resuelve este problema reutilizando cada camino para estimar simultáneamente todos los *bins* temporales de interés. De este modo, al construir un camino se calcula el tiempo de vuelo acumulado en cada rebote y se asigna la contribución al *bin* temporal correspondiente. Como resultado, cada camino de luz $\bar{x}$ contribuye a toda la dimensión temporal T (Figura 2.2c)

Capítulo 3

Denoising

Para resolver la ecuación de transporte de luz mediante la *Path Integral*, se recurre a estimadores de Monte Carlo. Este método constituye un estimador estocástico de integrales, por lo que los resultados obtenidos siempre presentan cierto nivel de ruido, conocido como varianza.

Para reducir este ruido es necesario aumentar el número de muestras, lo que conlleva un incremento considerable en el tiempo de ejecución. Uno de los principales inconvenientes de este enfoque es que la convergencia del algoritmo de Monte Carlo no es lineal. Al tratar de estimar la integral de una distribución, el error de la estimación viene dado por la desviación típica [25], expresada como

$$\text{Var}[\hat{I}_N] = \frac{\sigma^2}{N} \Rightarrow \text{RMSE}(\hat{I}_N) = \mathcal{O}\left(\frac{1}{\sqrt{N}}\right). \quad (3.1)$$

Debido a esta lenta convergencia hacia el valor esperado, muchos sistemas de renderizado emplean diversas técnicas para acelerar el proceso de convergencia, tales como el muestreo por importancia (*importance sampling*), el muestreo múltiple por importancia (*multiple importance sampling*) y el *denoising*, entre otras. Esta última técnica es la que se explorará en profundidad en este trabajo.

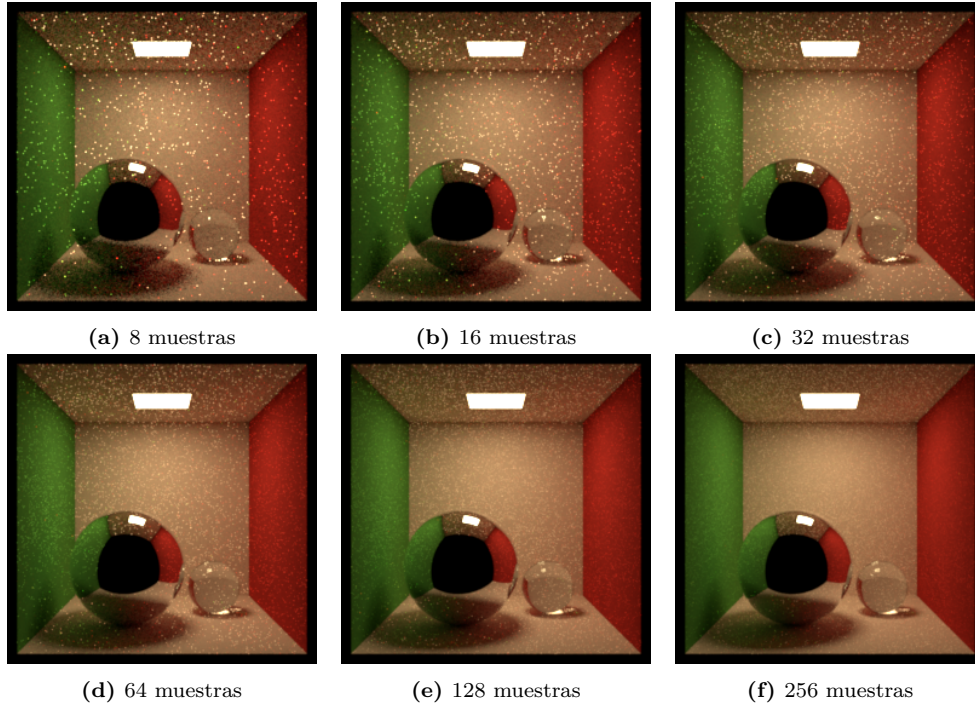


Figura 3.1: Convergencia del algoritmo de Monte Carlo con diferente número de muestras. Se observa la lenta convergencia del estimador y la presencia persistente de ruido (varianza) incluso con un número elevado de muestras. La mejora en calidad visual requiere incrementar significativamente el número de muestras, ilustrando la relación $\mathcal{O}(1/\sqrt{N})$ del error.

3.1. Métodos de denoising

El *Monte Carlo denoising* se basa fundamentalmente en el promedio ponderado de píxeles vecinos

$$\tilde{\theta}_j = \sum_i w_{ij} \hat{\theta}_i, \quad (3.2)$$

donde $\tilde{\theta}_j$ es el estimador *denoiseado* del píxel j , $\hat{\theta}_i$ son los estimadores ruidosos de los píxeles vecinos i , y w_{ij} son los pesos asignados a cada vecino. La clave de cualquier método de *denoising* reside precisamente en cómo se calculan estos pesos w_{ij} : pesos bien elegidos reducen la varianza sin introducir sesgo excesivo, mientras que pesos mal elegidos pueden difuminar detalles importantes de la imagen.

La idea del *Monte Carlo denoising* tuvo su origen en la comunidad de procesamiento de imágenes, donde el concepto principal consiste en el uso de filtros tradicionales de imagen. Estos filtros realizan el promedio de múltiples píxeles vecinos en el espacio imagen, reduciendo de esta manera la varianza. Este enfoque resulta muy efectivo en escenas que no presenten discontinuidades o bordes muy contrastados.

Sin embargo, en escenas con bordes claramente delimitados, estos filtros introducen un sesgo (*bias*) no deseado, que se manifiesta como un exceso de difuminado (*blurring*) en zonas donde el valor esperado presenta grandes diferencias (por ejemplo, en bordes, sombras, transiciones de iluminación, etc.).

3.1.1. Métodos clásicos

Conforme ha avanzado el campo del *Monte Carlo denoising*, se han desarrollado métodos más sofisticados para reducir el ruido sin introducir este sesgo indeseado. Los enfoques clásicos abarcan una amplia variedad de formulaciones, utilizando *kernels* de filtrado de imagen como filtros gaussianos, filtros bilaterales o medias no locales (*non-local means*) [26, 36], enfoques bayesianos [2], o técnicas basadas en ondículas [23, 7, 31].

Una de las técnicas más utilizadas son los *Joint Bilateral Filters* (JBF), que hacen uso de información adicional conocida como *G-Buffers*. Esta información se obtiene de la escena mediante el primer rebote de la luz y permanece constante durante el proceso de renderizado, incluyendo datos como el albedo (color de la superficie del material), las normales de superficie, la profundidad, etc. Muchos métodos dependen precisamente de estos *G-Buffers* junto con la varianza estimada para guiar el proceso de *denoising*.

Este filtro consigue evitar el difuminado en los bordes geométricos, pero al disponer únicamente de información del primer rebote, continúa difuminando materiales que tienen su mayor contribución a la radiancia final en rebotes posteriores, como pueden ser los materiales especulares. Tampoco es capaz de diferenciar adecuadamente efectos de iluminación complejos como sombras o cáusticas.

Otros enfoques clásicos se basan en distribuciones de histogramas [2, 32], aunque estos sufren de problemas de resolución que se agravan aún más al aplicarlos a renderizado transitorio debido a la dimensión temporal adicional. Los meta-*denoisers* combinan múltiples técnicas de *denoising* [22, 26, 42, 9], seleccionando cuáles utilizar para minimizar el error o mostrar robustez frente al ruido, aunque esto incrementa considerablemente los requisitos computacionales.

3.1.2. Métodos basados en aprendizaje

Las técnicas que actualmente representan el estado del arte están enfocadas en el uso de redes neuronales, como OptiX (NVIDIA OptiX AI-Accelerated Denoiser) [5] o OIDN (Intel Open Image Denoise) [1]. Estas redes utilizan los *G-Buffers* como datos auxiliares y la imagen ruidosa como entrada principal, produciendo como salida la misma imagen con una aproximación de bajo error respecto al *ground truth* (referencia calculada con un impráctico número de muestras).

Para entrenar estos modelos, es necesario generar grandes conjuntos de datos con imágenes ruidosas de entrada e imágenes de referencia sin ruido, lo que requiere enormes recursos computacionales. Este problema se agrava significativamente en el dominio espacio-temporal del renderizado transitorio.

Aunque estos enfoques pueden conseguir excelentes resultados y están más enfocados en producir resultados visualmente agradables, generalmente introducen sesgo y pueden generar alucinaciones en la salida *denoiseada*.

Debido a que estas técnicas dependen de datos de entrenamiento, los resultados pueden ser poco satisfactorios en escenas que difieran significativamente de aquellas utilizadas durante el proceso de entrenamiento. Además, en el caso del renderizado transitorio, los datos de entrenamiento son escasos y crear un conjunto de datos balanceado no es trivial. La creación de un conjunto de datos sin ruido para renderizado transitorio es computacionalmente extremadamente costosa debido a la alta varianza del transporte de luz indirecto con múltiples rebotes.

3.1.3. Enfoque estadístico

En lugar de depender de técnicas de aprendizaje, este trabajo se basa en el enfoque estadístico propuesto recientemente [30], que recolecta estadísticas en línea por píxel (momentos centrales) durante el renderizado, las cuales posteriormente se comparan y analizan para minimizar el ruido de forma que se garantiza convergencia a la solución correcta.

Sin embargo, adaptar ingenuamente este enfoque al renderizado transitorio resulta impracticable debido a sus excesivos requisitos de memoria. Este trabajo muestra cómo extender la recolección de estadísticas al dominio temporal y aprovechar las correlaciones espacio-temporales, manteniendo al mismo tiempo la eficiencia en memoria. El *denoiser* transitorio propuesto es independiente de la estrategia de muestreo utilizada, maneja tanto transporte de luz basado en superficies como volumétrico, y no depende de técnicas de aprendizaje computacionalmente costosas. Además, su naturaleza libre de entrenamiento y su versatilidad permiten que en el futuro pueda adaptarse para *denoisear* otras simulaciones de Monte Carlo como acústica resuelta en tiempo [8] o espectroscopía transitoria [19].

3.2. Denoising estadístico

En esta sección se introduce el *denoiser* estado del arte formulado por Sakai et al. [30], quienes presentan un novedoso algoritmo de *denoising* basado en el uso de estadísticas recopiladas durante el proceso de renderizado, utilizadas para conseguir un equilibrio óptimo entre el sesgo (*bias*) y la varianza.

Formalmente, el error cuadrático medio (*Mean Squared Error*, MSE) del estimador combinado $\tilde{\theta}$ puede descomponerse en términos del sesgo y la varianza

$$\text{MSE}(\tilde{\theta}_j, \theta_j) = \mathbb{E}[(\tilde{\theta}_j - \theta_j)^2] = \text{Var}(\tilde{\theta}_j) + \text{Bias}^2(\tilde{\theta}_j, \theta_j), \quad (3.3)$$

donde θ_j se refiere al valor real desconocido (*ground truth*) para el estimador $\hat{\theta}_j$. El estimador ruidoso $\hat{\theta}_j$ se considera no sesgado (*unbiased*), a diferencia de su contraparte filtrada $\tilde{\theta}_j$, que contiene cierto sesgo debido a la combinación con otros estimadores. La varianza y el sesgo de este estimador combinado vienen dados por

$$\text{Var}(\tilde{\theta}_j) = \sum_i w_{ij}^2 \text{Var}(\hat{\theta}_i(n_i)), \quad (3.4)$$

$$\text{Bias}(\tilde{\theta}_j, \theta_j) = \sum_i w_{ij} \text{Bias}(\hat{\theta}_i(n_i), \theta_j), \quad (3.5)$$

donde $\text{Bias}(\hat{\theta}_i, \theta_j) = \mathbb{E}[\hat{\theta}_i] - \theta_j$.

Una vez formulado el error MSE en términos de varianza y sesgo, sería conceptualmente sencillo encontrar un equilibrio óptimo entre ambos términos minimizando el error total

$$\{w_{ij}^*\} = \arg \min_{\{w_{ij}\}} \sum_j \text{MSE}(\tilde{\theta}_j, \theta_j). \quad (3.6)$$

Sin embargo, esto no es tan directo en la práctica, ya que los valores reales θ_j son desconocidos y por tanto es necesario trabajar con estimaciones ruidosas del sesgo y la varianza.

Una vez definido el problema de *denoising*, la propuesta de Sakai et al. [30] consiste en el uso de un filtro base ρ_{ij} que utiliza información *a priori* de bajo nivel (*G-Buffers*), junto con una función de membresía m_{ij} (*membership function*) calculada mediante estadísticas para definir qué estimadores se combinarán y obtener un equilibrio óptimo entre sesgo y varianza. Los pesos w_{ij} se calculan como

$$w_{ij} = \frac{\rho_{ij} m_{ij}}{\sum_k \rho_{ik} m_{ik}}. \quad (3.7)$$

Las razones para el uso de este filtro base ρ_{ij} son: (1) limitar el tamaño del *kernel*, consiguiendo así mejorar el rendimiento computacional, (2) aprovechar enfoques bien establecidos en reducción de varianza, y (3) incluir información *a priori* de bajo ruido disponible en el renderizador (*G-Buffers*). Este filtro base permite preservar características geométricas y de texturas, como bordes nítidos entre diferentes geometrías. Sin embargo, resulta ineficaz para otras características que solo se obtienen durante el proceso de muestreo, como las sombras y cáusticas.

Este filtro base es muy efectivo para la reducción de la varianza. El objetivo de las funciones de membresía m_{ij} es limitar el sesgo excluyendo la combinación entre estimadores que difieren significativamente. Consiguiendo así que el filtro diferencie características que se obtienen solo durante el proceso de muestreo, como las sombras y las cáusticas. Ambos filtros se complementan: el filtro base utiliza la información *a priori* con bajo error para diferenciar partes geométricas, mientras que el filtro de membresía consigue diferenciar zonas como las sombras y cáusticas para que no se combinen con otros estimadores, evitando así el exceso de difuminado.

Para el cálculo de las funciones de membresía m_{ij} se considera el problema de minimización del MSE (Ecuación (3.3)) para cada par de estimadores ruidosos $\hat{\theta}_i, \hat{\theta}_j$. Estos se combinan con un peso w : $\tilde{\theta}_i = w \hat{\theta}_i + (1 - w) \hat{\theta}_j$ y $\tilde{\theta}_j = w \hat{\theta}_j + (1 - w) \hat{\theta}_i$. Nótese que se fuerza la simetría en los pesos $w_{ij} = w_{ji} = (1 - w)$ con el objetivo de preservar la energía.

El sesgo introducido por la combinación de ambos estimadores es $\text{Bias}(\tilde{\theta}_i, \theta_i) = \mathbb{E}[\tilde{\theta}_i - \theta_i] = (w - 1) \theta_i + (1 - w) \theta_j$, y de forma análoga para $\text{Bias}(\tilde{\theta}_j, \theta_j)$. Minimizando la suma de los términos de error (Ecuación (3.3)), se obtiene la expresión para el peso óptimo, dada por

$$\begin{aligned} w^* = \arg \min_w & \left(w^2 \text{Var}(\hat{\theta}_i) + (1 - w)^2 \text{Var}(\hat{\theta}_j) \right. \\ & + [(w - 1) \theta_i + (1 - w) \theta_j]^2 \\ & + w^2 \text{Var}(\hat{\theta}_j) + (1 - w)^2 \text{Var}(\hat{\theta}_i) \\ & \left. + [(w - 1) \theta_j + (1 - w) \theta_i]^2 \right). \end{aligned} \quad (3.8)$$

Se obtiene el valor óptimo w igualando a cero la derivada con respecto al peso, resultando en

$$w^* = \frac{2(\theta_i - \theta_j)^2 + \text{Var}(\hat{\theta}_i) + \text{Var}(\hat{\theta}_j)}{2[(\theta_i - \theta_j)^2 + \text{Var}(\hat{\theta}_i) + \text{Var}(\hat{\theta}_j)]}. \quad (3.9)$$

Sin embargo, como se mencionó anteriormente, no conocemos los valores reales θ_i, θ_j , por lo que es necesario trabajar con estimaciones ruidosas, produciendo así resultados ruidosos para w^* . Consecuentemente, establecer $\rho_{ij} = 1$ y $m_{ij} = 1 - w^*$ no proporciona buenos resultados.

Por tanto, el enfoque de Sakai et al. [30] consiste en usar un filtro base ρ_{ij} para suavizar y definir la función de membresía binaria m_{ij} como

$$m_{ij} = \begin{cases} 1 & \text{si } (1 - w^*) > \gamma, \\ 0 & \text{en otro caso,} \end{cases} \quad (3.10)$$

donde γ es el umbral que determina cuán discriminativa es la función de membresía. De esta forma se utiliza el peso óptimo por pares como una prueba estadística, previniendo así la combinación de

estimadores si esto introduciría un sesgo muy grande relativo a la suma de sus varianzas. En el artículo también demuestran que esta prueba es equivalente al test de Welch, $t < \gamma_w$ con $\gamma_w = \sqrt{\frac{1}{2\gamma}} - 1$. Usando el valor crítico de la distribución t de Student $\gamma_w = t_{1-\alpha/2, v}$, con el nivel de significación $\alpha = 0,005$ y la cota superior para los grados de libertad $v = n_i + n_j - 2$, siendo n_i y n_j el número de muestras de los estimadores.

El test de Welch considera dos distribuciones normalizadas. Para relajar esta suposición de normalización se puede usar el intervalo de confianza propuesto por Curto [6], cuyas formulaciones consideran estadísticas de orden superior como la asimetría (*skewness*). Dado que en renderizado la distribución de muestras no está normalizada, ya que la mayoría de muestras son cero (camino que no llegan a una fuente de luz) y hay unas pocas que aportan la mayor parte de la estimación de la radiancia (camino que contribuyen significativamente), la distribución resultante presenta una cola derecha pronunciada (*skewness* > 0).

Sakai et al. encuentran que una combinación que proporciona buenos resultados es una corrección de la media para tener en cuenta la asimetría junto con una transformación de las muestras usando Box-Cox, definidas como

$$\text{Corrección de la media: } \theta_i \approx \mu_i + \frac{M_{3,i}}{6 \sigma_i^2 n_i} \quad (3.11)$$

$$\text{Transformación Box-Cox: } x'_k(\lambda) = \begin{cases} \log(x_k), & \text{si } \lambda = 0, \\ \frac{x_k^\lambda - 1}{\lambda}, & \text{en otro caso.} \end{cases} \quad (3.12)$$

Para el parámetro λ se considera que $\lambda = 0,5$ obtiene buenos resultados, ya que evita el logaritmo de cero (que no está definido) y consigue comprimir de forma efectiva los valores atípicos elevados sin comprimir los valores pequeños hacia $-\infty$. De esta forma, el artículo propone dos algoritmos principales del *denoiser* estadístico. Estos se realizan en dos etapas del renderizado: el primero (Algoritmo 1) se ejecuta durante la etapa de renderizado, donde, conforme el renderizador va muestreando, recopila las muestras y las estadísticas necesarias para su uso posterior en el test estadístico y la corrección de la media, con el objetivo de obtener un mejor estimador θ_i . El segundo algoritmo (Algoritmo 2) se realiza en la etapa de postprocesado, donde, usando las estadísticas y los *G-Buffers* obtenidos en la etapa de renderizado, se calculan y aplican los pesos formados por el filtro base ρ_{ij} y la función de membresía calculada mediante el test estadístico m_{ij} .

Algoritmo 1: Estadísticas en línea por píxel. Se calculan los momentos centrales de orden 1 a 3 por píxel a partir de muestras de Monte Carlo, aplicando la transformación de Box-Cox [3] y la actualización incremental de Welford [38].

```

1: def update_pixel_stats ( $x_k$ ):
2:   Inicializar:  $\{n_i, \mu_i, \hat{M}_{2,i}, \hat{M}_{3,i}\} = 0$ 
3:   for cada muestra de Monte Carlo  $x_k$  :
4:     # Transformación de Box-Cox (Ecuación (3.12))
5:      $x'_k = \text{Box-Cox}(x_k)$ 
6:     # Incrementar contador y actualizar media (Welford)
7:      $n_i \leftarrow n_i + 1$ 
8:      $\delta = x'_k - \mu_i$ 
9:      $\mu_i \leftarrow \mu_i + \delta/n_i$ 
10:    # Actualizar momentos centrales acumulados
11:     $\hat{M}_{2,i} \leftarrow \hat{M}_{2,i} + \delta(\delta - \delta/n_i)$ 
12:     $\hat{M}_{3,i} \leftarrow \hat{M}_{3,i} - 3(\delta/n_i)\hat{M}_{2,i} + \delta(\delta^2 - (\delta/n_i)^2)$ 
13:    # Calcular momentos y varianza final
14:     $M_{l,i} = \hat{M}_{l,i}/n_i$ 
15:     $\sigma_i^2 = \hat{M}_{2,i}/(n_i - 1)$ 

```

Algoritmo 2: Cálculo de los pesos w_{ij} en espacio imagen. Se calcula el peso final w_{ij} para cada par de píxeles, mediante la multiplicación del filtro base ρ_{ij} , y las funciones de membresía m_{ij} obtenidas a partir de las estadísticas

```

1: def compute_weights ( $\{n, \mu, \sigma^2, M_2, M_3, p\}, \gamma_w$ ):
2:   for cada píxel central  $i$  en paralelo :
3:     for cada píxel candidato  $j$  dentro del radio del filtro base :
4:       # Evaluar filtro base (Ecuación (3.13))
5:        $\rho_{ij} = \text{filtro\_base}(\rho_i, \rho_j)$ 
6:       # Calcular varianzas de las estimaciones
7:        $\text{Var}(\hat{\theta}_i) = \sigma_i^2/n_i$ 
8:        $\text{Var}(\hat{\theta}_j) = \sigma_j^2/n_j$ 
9:       # Corregir medias por asimetría (Ecuación (3.11), [6])
10:       $\theta_i \approx \mu_i + \frac{M_{3,i}}{6\sigma_i^2 n_i}$ 
11:       $\theta_j \approx \mu_j + \frac{M_{3,j}}{6\sigma_j^2 n_j}$ 
12:      # Calcular peso óptimo (Ecuación (3.9))
13:       $w_{ij}^* = \text{compute\_w\_opt}(\theta_i, \theta_j, \text{Var}(\hat{\theta}_i), \text{Var}(\hat{\theta}_j))$ 
14:      # Evaluar umbral estadístico  $t$ 
15:       $t = \sqrt{(2(1 - w_{ij}^*)^{-1} - 1)}$ 
16:       $m_{ij} = (t < \gamma_w) ? 1 : 0$ 
17:      # Combinar filtro base y función de membresía
18:       $w_{ij} = \rho_{ij} m_{ij}$ 
19:      # Normalizar pesos por píxel
20:       $w_{ij} \leftarrow w_{ij} / \sum_k w_{ik}$ 

```

El filtro base ρ_{ij} se calcula para cada par de píxeles

$$\rho_{ij} = \exp \left(-\frac{1}{2}(\rho_j - \rho_i)^T \Sigma^{-1} (\rho_j - \rho_i) \right), \quad (3.13)$$

donde ρ_i es la información *a priori* que tenemos del píxel i , siendo estas la posición en el espacio imagen (x_i, y_i) , el color del albedo (r_i, g_i, b_i) y las normales de la superficie de la geometría (n_{ix}, n_{iy}, n_{iz})

$$\rho_i = (x_i, y_i, r_i, g_i, b_i, n_{ix}, n_{iy}, n_{iz})^T$$

La matriz Σ representa la matriz de covarianza que controla la tasa de caída para cada dimensión de ρ_i . Para mejorar la eficiencia se limita el *kernel* a un pequeño vecindario con un radio constante para cada píxel:

$$\Sigma = \text{diag}(10, 10, 0,02, 0,02, 0,02, 0,1, 0,1, 0,1)$$

Cabe destacar que al renderizar se trabaja con una serie de canales, siendo estos por lo general RGB. Las funciones de membresía funcionan de manera que se evalúan para cada canal siguiendo el Algoritmo 2. El peso w_{ij} será cero a menos que todos los canales pasen el test estadístico, es decir, que para todos los canales $m_{ij} = 1$.

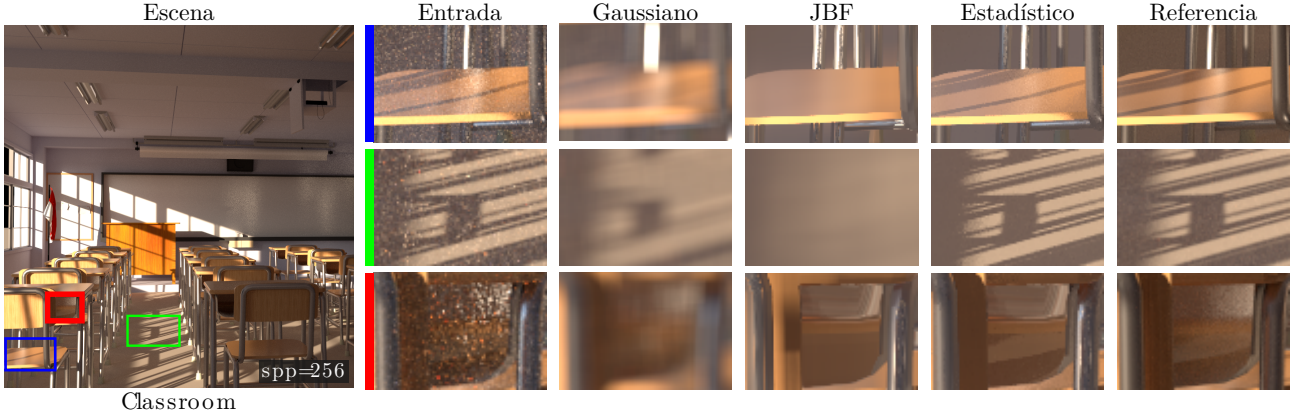


Figura 3.2: Comparación de diferentes métodos de denoising. En esta figura se puede ver una escena denoiseada con un filtro gaussiano, un *Joint Bilateral Filter*, el *denoiser* estadístico [30] y la imagen de referencia. Se puede observar que el filtro gaussiano reduce considerablemente la varianza, pero suaviza los bordes. El *Joint Bilateral Filter* mantiene los bordes provocados por la geometría, pero sigue suavizando efectos de iluminación como sombras y cáusticas, y los objetos especulares, ya que los *G-Buffers* solo contienen información del primer rebote. Mientras, el *denoiser* estadístico mantiene características como sombras, cáusticas y otros detalles importantes.

Capítulo 4

Denoising estacionario

Para la implementación del *denoiser* estadístico es necesario dividir el proceso en dos componentes principales que requieren tecnologías diferentes. Por un lado, se necesita un motor de renderizado que permita extender su funcionalidad para recolectar las estadísticas durante el proceso de renderizado (Sección 4.1). Por otro lado, se requiere una biblioteca eficiente para realizar las operaciones de *denoising* sobre las estadísticas recolectadas (Sección 4.2).

Para el primer componente se ha seleccionado Mitsuba 3 [15], un motor de renderizado orientado a la investigación que permite la extensión mediante plugins y ofrece capacidades de vectorización en CPU y GPU, gracias a que está construido sobre la librería Dr.Jit [14], un compilador *just-in-time* que compila el código a la arquitectura de bajo nivel que deseemos (CPU escalar, CPU vectorizada, GPU). Esta elección se justifica además porque posteriormente será necesario extender la implementación a renderizado transitorio utilizando el framework *mitransient* [28] (Capítulo 5), el cual está construido sobre Mitsuba 3.

Para el segundo componente se ha elegido PyTorch [24], una biblioteca de cálculo tensorial que permite implementar eficientemente las operaciones vectorizadas necesarias para el *denoising*, tales como la aplicación de *kernels* de filtrado y el cálculo de funciones de membresía sobre vecindarios de píxeles.

La comunicación entre ambos componentes se establece mediante tensores NumPy: Mitsuba 3 genera la imagen ruidosa, las estadísticas y los *G-Buffers*, y los transfiere a PyTorch en este formato compatible con ambos sistemas. PyTorch, utilizando esta entrada, aplica el *denoising* estadístico a la imagen.

4.1. Recolección de estadísticas

Para recolectar las estadísticas es necesario implementar un plugin que tenga acceso a las muestras de cada píxel. Con las muestras de cada píxel es posible calcular las estadísticas de la distribución de cada píxel, tales como la media μ , varianza σ^2 y los momentos centrales de segundo orden M_2 y del tercer orden M_3 . La forma más sencilla de conseguir esto es implementando un plugin de tipo Integrador (*Integrator*). Estos plugins definen el algoritmo de renderizado que se usa para calcular la integral (*ecuación de render*), como pueden ser *path tracing* o *photon mapping*.

Como únicamente se requiere acceso a las muestras, no es necesario implementar un nuevo algoritmo de renderizado completo. En su lugar, se implementa un *wrapper* al integrador real, es decir, el `StatisticalIntegrator` implementado llamará en todas sus funciones al integrador real definido por el usuario en el fichero `.xml` que define la escena.

Se modifica la función `sample()` ya que esta función tiene acceso a las muestras, estas se capturan y con ellas se calculan las estadísticas. Mitsuba 3 dispone de un modo simbólico que permite escribir código de forma iterativa que posteriormente, mediante Dr.Jit, se compila y vectoriza automáticamente para la arquitectura de bajo nivel que deseemos (CPU escalar, CPU vectorizada, GPU, etc.). Esto permitiría, en principio, implementar el algoritmo de Welford (Algoritmo 1) de forma iterativa y beneficiarse de la vectorización automática. Sin embargo, en modo simbólico no es posible acceder a variables que han

sido modificadas en iteraciones previas, como ocurre en la línea 6 del Algoritmo 1, donde el cálculo de μ requiere el valor de μ de la iteración anterior. Esta limitación impide implementar el algoritmo de Welford en modo simbólico.

Por esta razón, se opta por una implementación completamente vectorizada donde todas las muestras están disponibles simultáneamente, como se muestra en el Algoritmo 3. Al disponer de todas las muestras a la vez, la implementación resulta considerablemente más sencilla mediante operaciones vectorizadas. No obstante, esta decisión tiene la desventaja de limitar el código a arquitecturas vectorizadas (CPU vectorizada o GPU), perdiendo la capacidad de ejecutarse en modo escalar que proporcionaría el modo simbólico. En la práctica, esto no supone un problema significativo ya que las arquitecturas vectorizadas son las más utilizadas debido a su alto rendimiento.

Algoritmo 3: Estadísticas vectorizadas por píxel. Se calculan los momentos centrales, la varianza del estimador $Var(\hat{\theta})$ y se aproxima el estimador θ a partir de un conjunto de muestras $\mathbf{X}$, aplicando operaciones vectorizadas por píxel y canal.

```

1: def compute_pixel_stats_vectorized ( $\mathbf{X}$ ):
    # Entrada:  $\mathbf{X} \in \mathbb{R}^{h \times w \times c \times n}$ , donde  $n$  es el número de muestras por píxel
2:    $\mathbf{X}' \leftarrow \text{BoxCox}(\mathbf{X})$ 
    # Calcular media por píxel y canal
3:    $\mu \leftarrow \frac{1}{n} \sum_{i=1}^n X'_{[...,i]}$ 
    # Calcular desviaciones respecto a la media
4:    $\delta \leftarrow \mathbf{X}' - \mu$ 
    # Calcular momentos centrales de segundo y tercer orden
     $M_2 = \frac{1}{n-1} \sum_{i=1}^n \delta_{[...,i]}^2$ 
     $M_3 = \frac{1}{n} \sum_{i=1}^n \delta_{[...,i]}^3$ 
    # Corregir medias por asimetría (Ecuación (3.11))
5:    $\theta \approx \mu + \frac{M_3}{6\sigma^2 n}$ 
    # Calcular varianza de los estimadores
6:    $\text{Var}(\hat{\theta}) = \sigma^2/n$ 

```

Como se puede observar, no solo cambia la vectorización en el algoritmo, sino que también incluye el cálculo del estimador θ y la varianza $\text{Var}(\hat{\theta})$ en las dos últimas líneas. En el Algoritmo 1 se devuelven las estadísticas n , μ , σ^2 , M_2 y M_3 , siendo estas tres últimas las utilizadas para calcular θ y $\text{Var}(\hat{\theta})$ en el Algoritmo 2. Al realizar este cálculo en esta etapa y devolver como estadísticas únicamente n , θ y $\text{Var}(\hat{\theta})$, se reduce el número de estadísticas por píxel en dos (no se pierde información, las estadísticas finales que se usan son el estimador final θ y la varianza $\text{Var}(\hat{\theta})$ que derivan de las otras estadísticas), lo que supone un ahorro considerable de memoria al almacenar los tensores.

Este enfoque donde el cálculo de las estadísticas requiere todas las muestras vectorizadas podría causar problemas, que surgirían en casos en los que la memoria no fuese suficiente para almacenar todas las muestras vectorizadas simultáneamente. En tal escenario, Mitsuba 3 realiza varios pases vectorizando únicamente un subconjunto de muestras en cada iteración. En este caso, el algoritmo actual no está preparado y sería necesario implementar una versión más sofisticada basada en un algoritmo de Welford por lotes (*batched Welford*).

No obstante, esto no ha supuesto un inconveniente significativo en la práctica, ya que para que esta situación se presente es necesario un número muy elevado de muestras, y para obtener imágenes ruidosas precisamente se busca trabajar con un número reducido de muestras. Por ejemplo, para un renderizado de una imagen de 1280×720 píxeles con 1024 muestras por píxel, donde cada muestra es un `float32` (4 bytes), por lo que el uso de memoria total se puede estimar mediante

$$\text{Altura} \times \text{Anchura} \times \text{RGB} \times \text{Muestras} \times \text{Float} = 1280 \times 720 \times 3 \times 1024 \times 4 \text{ B} \approx 11 \text{ GB}. \quad (4.1)$$

El resultado es aproximadamente de 11 GB, por lo que todavía existe margen considerable antes de

alcanzar el límite que obligaría a usar procesamiento por lotes, asumiendo una máquina actualmente estándar con 32 GB de RAM.

4.2. Denoising con PyTorch

PyTorch [24] es una biblioteca de Python ampliamente utilizada en el ámbito de la Inteligencia Artificial. Esta biblioteca trabaja con tensores, estructuras de datos con un número arbitrario de dimensiones que permiten representar conceptos complejos. Por ejemplo, una imagen no puede representarse mediante una simple matriz bidimensional, ya que posee tres dimensiones: altura, anchura y canales de color (RGB). Estos tensores no solo permiten la representación de estos conceptos, sino que están implementados de forma eficiente para su paralelización y vectorización en arquitecturas GPU y CPU vectorizada, lo que los convierte en herramientas ideales para ámbitos como la Inteligencia Artificial y el Procesamiento de Imágenes. Este último es precisamente el ámbito del problema abordado en el Algoritmo 2, donde se dispone de una imagen ruidosa y se desea aplicar diferentes convoluciones (combinación de píxeles) con distintos filtros para obtener una imagen con menor nivel de ruido.

Como se mencionó en el Capítulo 3, el *denoiser* consta de dos *kernels* principales. El primero es el filtro base ρ_{ij} , un filtro basado en información *ground truth*, específicamente el *Joint Bilateral Filter* (ver Subsección 4.2.2), que permite distinguir entre diferentes objetos con distintos albedos y normales. El segundo es el filtro de membresías (ver Subsección 4.2.3) que, utilizando las estadísticas recopiladas en el Algoritmo 1, permite identificar qué píxeles añadirían excesivo sesgo frente a la reducción de varianza obtenida, logrando así un compromiso óptimo entre sesgo y varianza. Este filtro es capaz de distinguir fenómenos lumínicos como reflejos, sombras y cáusticas. Ambos filtros se combinan para obtener resultados de alta calidad.

4.2.1. Entrada y vectorización

PyTorch recibe como entrada la imagen ruidosa, las estadísticas y los *G-Buffers*, representados en forma de tensores con dimensiones (C, H, W) , donde C representa los canales (3 para RGB, 8 para los *G-Buffers*), H la altura y W la anchura de la imagen. Para poder aplicar las operaciones de forma vectorizada sobre vecindarios de píxeles, es necesario transformar estos tensores utilizando la función `Unfold` de PyTorch.

La función `Unfold(kernel_size, stride)` se utiliza para extraer ventanas deslizantes (*patches*) de un tensor de entrada. Se emplea con los siguientes parámetros:

- **kernel_size**: define el tamaño de la ventana a extraer. Si se desea un radio r en píxeles, el tamaño del *kernel* se calcula como $2 \cdot r + 1$. Por ejemplo, un radio de 20 píxeles corresponde a un **kernel_size** de 41, lo que genera $41^2 = 1681$ vecinos por píxel.
- **stride**: determina el desplazamiento de la ventana entre cada extracción. En este caso, se usa **stride=1** para generar un patch centrado en cada píxel.

Después de aplicar `Unfold()`, los tensores resultantes tienen dimensiones (C, N_v, H, W) , donde N_v indica el número de vecinos por píxel.

Esta representación permite vectorizar completamente las operaciones necesarias para calcular tanto los pesos del JBF como las membresías, evitando bucles iterativos en Python, que resultarían ineficientes.

4.2.2. Joint Bilateral Filter

El *Joint Bilateral Filter* (JBF) es un filtro similar al filtro gaussiano, pero no solo considera la distancia espacial entre píxeles para calcular los pesos, sino que incorpora información adicional. En imágenes reales capturadas por una cámara, normalmente solo se dispone de información espacial. Sin embargo, dado que las imágenes a *denoise* son generadas por un sistema de renderizado, se tiene

acceso a información adicional almacenada en los *G-Buffers*: albedo (color de la superficie del material) y las normales de la superficie del material. Para calcular los pesos del JBF, se utilizan los *G-Buffers* de los píxeles centrales y sus vecinos mediante la Ecuación (3.13) como

$$\rho = \exp \left(-\frac{1}{2}(\rho_v - \rho_c)^\top \Sigma^{-1} (\rho_v - \rho_c) \right), \quad (4.2)$$

donde ρ_v y ρ_c son los tensores de información a priori (*G-Buffers*) de los píxeles centrales y de los píxeles vecinos, respectivamente.

Para la implementación, primero se define la inversa de la matriz de covarianza Σ^{-1} , que determina la tasa de atenuación (*falloff*) de cada característica. Dado que esta matriz es constante, se define como un tensor de constantes para evitar calcular la inversa en cada ejecución

$$\Sigma^{-1} = \text{diag}(0,1, 0,1, 50, 50, 50, 10, 10, 10). \quad (4.3)$$

El cálculo principal a realizar dentro del argumento de la exponencial es

$$-\frac{1}{2}(\rho_v - \rho_c)^\top \Sigma^{-1} (\rho_v - \rho_c). \quad (4.4)$$

Dado que Σ^{-1} es una matriz diagonal, la multiplicación matricial se reduce a un producto elemento a elemento. Por lo tanto, los pesos ρ se pueden expresar directamente como

$$\rho = \exp \left(-\frac{1}{2} \sum_k \left[(\rho_v - \rho_c)^2 \odot \Sigma^{-1} \right]_k \right), \quad (4.5)$$

donde la suma recorre la dimensión de las características k .

Con esta simplificación, todas las operaciones intermedias se realizan elemento a elemento sobre tensores, permitiendo una vectorización eficiente mediante PyTorch, seguida de una reducción mediante suma en la dimensión de las características y la aplicación final de la exponencial.

El resultado final del cálculo del JBF es un tensor con dimensiones $(1, N_v, H, W)$, ya que el peso es el mismo para cada canal RGB.

4.2.3. Membresías

El cálculo de las membresías sigue la misma estrategia de vectorización que el JBF, trabajando con tensores de dimensiones (C, N_v, H, W) y produciendo una salida de dimensiones $(1, N_v, H, W)$.

El cálculo de las membresías se realiza como se describe en el Algoritmo 2. El proceso consta de varios pasos que se detallan a continuación:

Cálculo del valor crítico γ_w

El valor crítico γ_w es una constante que se puede calcular a priori mediante la distribución t de Student

$$\gamma_w = t_{1-\alpha/2, \nu}, \quad (4.6)$$

donde α es el nivel de significancia del test (pasado por el usuario, típicamente $\alpha = 0,005$) y $\nu = n_i + n_j - 2$ son los grados de libertad, siendo n_i y n_j el número de muestras de cada píxel. Dado que todos los píxeles tienen el mismo número de muestras, ν es constante para toda la imagen. Para calcular γ_w se utiliza la biblioteca SciPy [37] de Python.

Cálculo del peso óptimo w^*

Para cada par de píxeles, calculamos el peso óptimo w^* que minimiza el RMSE mediante la Ecuación (3.9). Nuestra implementación vectorizada en PyTorch calcula simultáneamente w^* para todos los pares de píxeles vecinos, definiéndolo como

$$w^* = \frac{2(\theta_c - \theta_v)^2 + \text{Var}(\hat{\theta}_c) + \text{Var}(\hat{\theta}_v)}{2 \left[(\theta_c - \theta_v)^2 + \text{Var}(\hat{\theta}_c) + \text{Var}(\hat{\theta}_v) \right]}, \quad (4.7)$$

donde θ_c y θ_v son tensores que contienen los estimadores del píxel central y sus vecinos respectivamente, y $\text{Var}(\hat{\theta}_c)$ y $\text{Var}(\hat{\theta}_v)$ son sus varianzas respectivas.

Esta expresión requiere un tratamiento especial de casos límite:

- Si el denominador es cero, esto implica que tanto la diferencia entre estimadores como la suma de varianzas son cero, resultando en una indeterminación $0/0$. En este caso se asigna $w^* = 0,5$, permitiendo la combinación de píxeles. Esta situación ocurre típicamente cuando un canal tiene valor cero para todas las muestras (por ejemplo, ausencia de componente roja), resultando en estimador y varianza nulos. Si ambos píxeles tienen varianza cero y el mismo estimador, significa que comparten el mismo color. La decisión de asignar $w^* = 0,5$ (permitir combinación) se basa en que los otros dos canales RGB podrían beneficiarse del *denoising*, ya que la similitud en este canal no aporta información negativa sobre la compatibilidad de los píxeles.
- Si la suma de varianzas es cero ($\text{Var}(\hat{\theta}_c) + \text{Var}(\hat{\theta}_v) \leq 0$) pero los estimadores difieren ($\theta_c \neq \theta_v$), se asigna $w^* = 1,0$. Esto indica que los píxeles tienen valores determinísticos pero diferentes, por lo que no deben combinarse para evitar introducir sesgo.

Cálculo del estadístico t

Con el peso óptimo w^* calculado, se obtiene el estadístico de prueba t para cada par de píxeles

$$t = \sqrt{\frac{1}{2(1 - w^*)}} - 1. \quad (4.8)$$

Cuando $w^* = 1$, esta expresión resulta en $t = \infty$, indicando que los píxeles son completamente diferentes y no deben combinarse.

Cálculo de las membresías

Finalmente, la membresía para cada vecino se determina comparando el estadístico t con el valor crítico γ_w . La condición de membresía se define como

$$m = \begin{cases} 1 & \text{si } (t < \gamma_w) \text{ para todos los canales RGB} \\ 0 & \text{en caso contrario.} \end{cases} \quad (4.9)$$

Es importante destacar que esta condición debe cumplirse para todos los canales RGB simultáneamente, es decir, basta con que un solo canal no cumpla el test estadístico para que la membresía sea 0. Adicionalmente, el píxel central siempre tiene membresía igual a 1.

El resultado final es un tensor de membresías con dimensiones $(1, N_v, H, W)$, donde cada elemento indica si el vecino correspondiente debe ser considerado (1) o excluido (0) en la convolución final.

4.2.4. Teselado

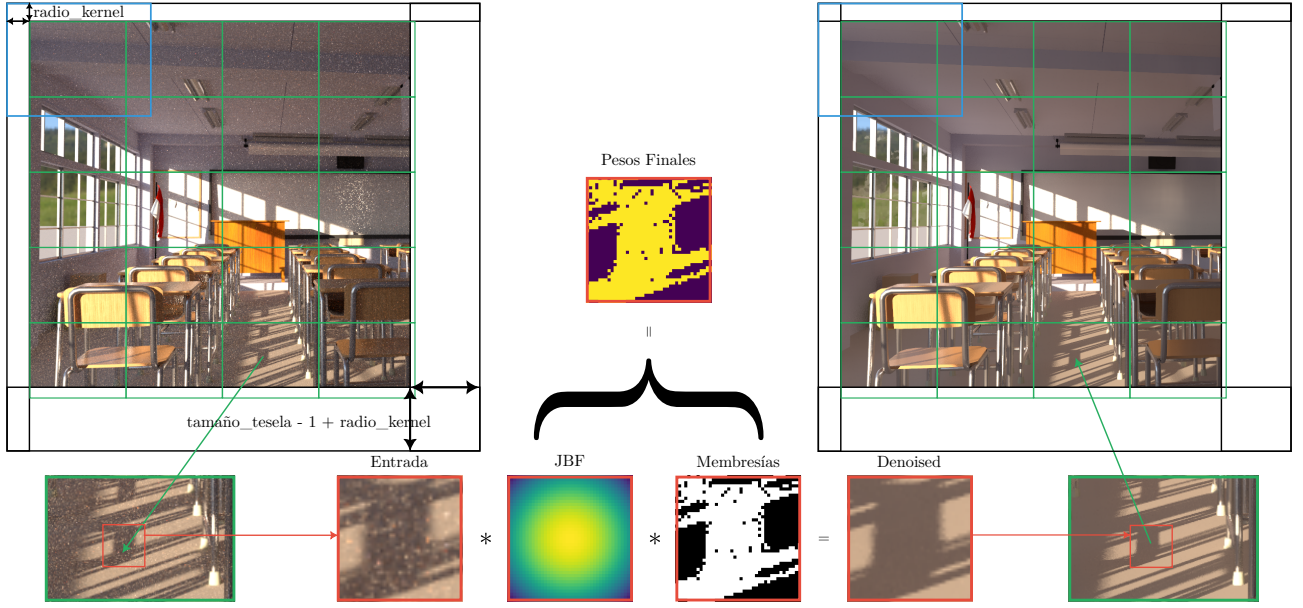


Figura 4.1: Denoising estacionario. En esta figura se muestra el proceso de *denoising* mediante teselado. Primero se aplica padding a la imagen original (recuadros negros), después se divide la imagen en teselas (recuadros azules y verdes). Para cada tesela se extraen los *kernels* de vecinos de cada píxel, se calculan los pesos del *Joint Bilateral Filter* (JBF) y las *membresías*, y se convolucionan con la imagen ruidosa para obtener una imagen sin ruido (recuadros rojos). En este caso se puede observar cómo en la zona de sombra del suelo el JBF no es capaz de distinguir qué zonas son correctas para combinar, ya que todos los píxeles pertenecen al mismo material con idénticas normales y albedo, funcionando únicamente como un *kernel* gaussiano. Por el contrario, las *membresías* logran diferenciar la parte sombreada, consiguiendo así un *denoising* donde no se difuminan las sombras.

Como se ha mencionado anteriormente, el proceso de *denoising* utiliza tensores donde se vectoriza completamente el procesamiento, resultando en tensores de 4 dimensiones (C, N_v, H, W). Para una imagen de 1280×720 píxeles con 3 canales RGB, usando un radio de *kernel* de 20 (1681 vecinos) y representación en punto flotante de 32 bits, el tamaño de un único tensor se puede calcular como

$$3 \times 1681 \times 1280 \times 720 \times 32 \text{ bits} \approx 18 \text{ GB.} \quad (4.10)$$

Dado que se trabaja con varios tensores de este tamaño simultáneamente (para el JBF, *membresías*, imagen ruidosa, etc.), no es viable vectorizar todo el procesamiento en una única pasada, ya que no es común disponer de equipos con GPU de más de 16 GB de VRAM o más de 32 GB de RAM.

La solución adoptada consiste en dividir la imagen en teselas (*tiles*) antes de procesarla. Es decir, se procesan partes de la imagen de forma iterativa mediante un bucle que recorre las diferentes teselas. Para generar estas teselas se utiliza nuevamente la función `Unfold()`.

Sin embargo, es importante tener en cuenta que si se desea procesar una tesela de 128×128 píxeles, es necesario extraer una tesela más grande de la imagen original. El tamaño real de la tesela extraída se calcula como

$$\text{tamaño_tesela_extraída} = \text{tamaño_tesela_procesada} + 2 \times \text{radio_kernel}, \quad (4.11)$$

donde *tamaño_tesela_procesada* es la dimensión de la tesela que se desea procesar y *radio_kernel* corresponde al radio del *kernel* utilizado para el filtrado.

Esto se debe a que los píxeles en los bordes de la tesela original necesitan acceso a sus vecinos para calcular correctamente los *kernels*. Por ejemplo, si se desea procesar una tesela de 128×128 con un radio de *kernel* de 20, es necesario extraer una tesela de 168×168 píxeles. Esto se puede observar en la

Figura 4.1, donde el recuadro azul representa la tesela grande extraída y el recuadro verde representa la tesela final que se procesa.

Además, es necesario aplicar *padding* a la imagen completa antes de realizar el teselado. Esto se debe a que la operación *unfold* genera patches centrados en cada píxel, por lo que sin *padding* los píxeles de los bordes de la imagen no se procesarían correctamente. Se aplica un *padding* de tamaño igual al radio del *kernel* en los bordes superior e izquierdo. Para los bordes derecho e inferior, el *padding* debe ser mayor, ya que no se conoce a priori en qué tesela caerán los últimos píxeles. El *padding* necesario se calcula como

$$\text{padding.derecho/inferior} = \text{tamaño_tesela} - 1 + \text{radio_kernel} \quad (4.12)$$

Siguiendo el ejemplo anterior, esto resultaría en un *padding* de $128 - 1 + 20 = 147$ píxeles para los bordes derecho e inferior.

4.3. Resultados en estado estacionario

Para la verificación del correcto funcionamiento del *denoiser* implementado, se han comparado con otros filtros de *denoising*. Inicialmente se pretendía realizar una comparación directa con los resultados presentados por Sakai et al. [30], sin embargo, al utilizar un sistema de renderizado diferente, las imágenes ruidosas generadas difieren ligeramente. Además, existen discrepancias en los *tone mappers* empleados y en las características de las escenas (texturas, materiales, etc.). Por tanto, para verificar el funcionamiento correcto del método propuesto, se compara con el filtro base sin membresías (JBF únicamente) y con un *denoiser* de referencia basado en redes neuronales, OptiX (NVIDIA OptiX AI Accelerated Denoiser), considerado como estado del arte en *denoising* en tiempo real.

En la Figura 4.2 se presentan los resultados obtenidos para tres escenas: BATHROOM, CLASSROOM y STAIRCASE, renderizadas con 32, 256 y 256 muestras por píxel (spp) respectivamente. La figura muestra varios *insets* (regiones ampliadas de la escena) que permiten observar con mayor detalle las diferencias entre los distintos filtros. Cada *inset* presenta cinco columnas: la imagen original con ruido, el resultado del JBF, el resultado de OptiX, el resultado del método propuesto y finalmente la imagen de referencia (*ground truth*), renderizada con 30k muestras por píxel para las tres escenas.

En la primera escena, BATHROOM, renderizada con 32 spp, se observa que las imágenes originales presentan un nivel de ruido considerable. El JBF logra reducir significativamente la varianza, obteniendo resultados visualmente agradables, especialmente en el primer *inset*. Sin embargo, un análisis más detallado revela que en el primer *inset* no consigue preservar adecuadamente el reflejo, mientras que en el segundo *inset*, al tratarse de un material especular que carece de información del primer rebote, no es capaz de mantener los reflejos. Tanto OptiX como el método propuesto obtienen buenos resultados en el primer *inset*, pero en el segundo y tercero se observa que OptiX presenta mayor nivel de ruido residual. Por el contrario, el método propuesto demuestra la efectividad de las membresías al conseguir preservar los reflejos en el primer y segundo *inset*.

En la segunda escena, CLASSROOM, se observa un comportamiento similar. En el primer y segundo *inset*, el JBF no es capaz de preservar adecuadamente las sombras y reflejos, difuminándolos, mientras que el método propuesto logra mantenerlos reduciendo simultáneamente la varianza. Es especialmente destacable el tercer *inset*, donde OptiX presenta un rendimiento notablemente deficiente con un nivel de ruido elevado. El JBF realiza un trabajo aceptable reduciendo la varianza y manteniendo las texturas de la madera, pero el método propuesto, combinando JBF y membresías, consigue reducir la varianza, mostrar las texturas de madera y conservar un reflejo que resulta difícil de observar en la imagen original ruidosa, pero que se aprecia claramente en la imagen de referencia.



Figura 4.2: Resultados del *denoising* estacionario. Se muestran tres escenas (BATHROOM, CLASSROOM, STAIRCASE) renderizadas con 32, 256 y 256 muestras por píxel (spp) respectivamente. Para cada escena se presentan diversos *insets* con cinco columnas: imagen original ruidosa, resultado del *Joint Bilateral Filter* (JBF), resultado de OptiX (NVIDIA OptiX AI-Accelerated Denoiser), resultado del método propuesto, e imagen de referencia (30k spp). Se utiliza el RMSE para comparar cada método con la referencia. Se observa cómo el JBF reduce efectivamente la varianza pero tiende a difuminar detalles como reflejos y sombras. OptiX, aunque basado en redes neuronales, presenta ruido residual en regiones complejas. El método propuesto, mediante la combinación de JBF y membresías estadísticas, logra reducir la varianza preservando simultáneamente detalles finos como reflejos especulares y sombras, evitando la introducción de sesgo en regiones con fenómenos lumínicos complejos.

Capítulo 5

Denoising transitorio

En el *render* transitorio simulamos la propagación de la luz en movimiento (Capítulo 2, Sección 2.2) mostramos cómo la integral de renderizado puede extenderse al dominio transitorio, lo que permite modelar fenómenos dependientes del tiempo, como la propagación de la luz a lo largo de este, y visualizar los resultados en forma de secuencias temporales o vídeos.

Por otro lado, en el Capítulo 3 se describió el método de *denoising* estadístico [30], el cual permite obtener resultados de alta calidad con bajo sesgo y varianza mediante la recolección de estadísticas durante el proceso de renderizado estacionario.

En este Capítulo realizamos la formulación teórica del *denoiser* estadístico extendido a la dimensión temporal. La Sección 5.1 presenta la extensión del *framework* estadístico al dominio transitorio mediante filtros espacio-temporales, implementada en *mitransient* [28] (ver Figura 5.1), una librería basada en Mitsuba 3 [15] que añade soporte para la simulación transitoria. En la Sección 5.2 se explora una forma eficiente de recolectar las estadísticas usando *mitransient*, aprovechando la reutilización de caminos para evitar problemas de memoria. Finalmente, en la Sección 5.3 se describe cómo, utilizando la formulación teórica y la recolección eficiente de estadísticas, se implementa el *denoising* estadístico transitorio mediante la combinación de funciones de membresía y filtros espacio-temporales. El proceso de *denoising* se realiza en PyTorch [24], de forma análoga a lo expuesto en el Capítulo 4.

Por último en la Sección 5.4 se muestra el rendimiento del *denoiser* estadístico transitorio propuesto en diferentes escenas frente a otros *denoisers*, también se hace un análisis exhaustivo de los diferentes parámetros del *denoiser* y cómo afecta a su rendimiento.

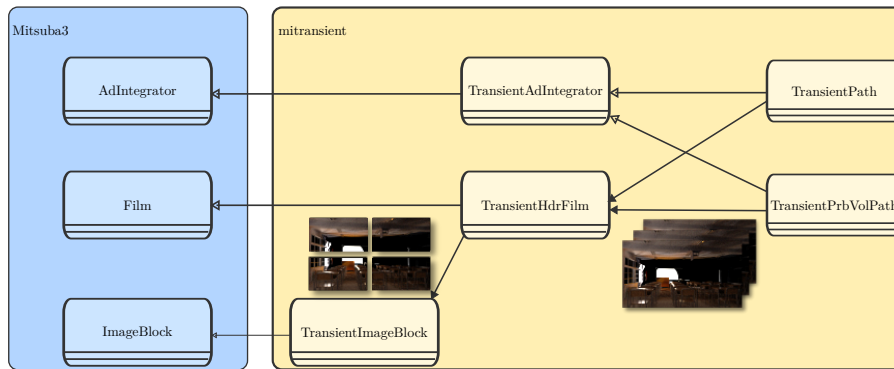


Figura 5.1: Diagrama de clases de *mitransient*. En este diagrama se muestra la relación entre las clases de *mitransient* y Mitsuba 3. *Mitrsient* proporciona dos integradores: *TransientPath* para *Path Tracing* transitorio con *Path Reuse*, y *TransientPrbVolPath* para medios participativos (materiales volumétricos como niebla o humo donde la luz interactúa con partículas en suspensión). Ambos integradores utilizan *TransientHdrFilm*, esta clase almacena el vídeo transitorio generado. A su vez, *TransientHdrFilm* utiliza *TransientImageBlock*, esta clase implementa la estructura de datos (tensores), que acumulan la radiancia en el *bin* correspondiente para cada rebote de luz. En este trabajo modificamos estas clases de la siguiente manera: (1) la función `sample` de los integradores según el Algoritmo 4, (2) *TransientHdrFilm* para calcular los estimadores θ y la varianza $Var(\hat{\theta})$ (Algoritmo 5), y (3) *TransientImageBlock* tiene los tensores X_1, X_2, X_3 que se acumulan usando `sync_accum` descritos en el Algoritmo 4.

5.1. Renderizado transitorio con estadísticas

Para extender el *denoising* estadístico al estado transitorio, se propone el uso de filtros espacio-temporales. Mientras que el *denoising* convencional promedia los vecinos cercanos únicamente en la dimensión espacial, un filtro espacio-temporal considera además los vecinos cercanos a través de la dimensión temporal, es decir, de *bins* temporales adyacentes. Para ello, se propone extender el *framework* de *denoising* estadístico [30] para tener en cuenta explícitamente la dimensión temporal.

Para obtener un compromiso casi óptimo entre sesgo y ruido, se busca obtener el estimador *denoised* $\tilde{\theta}_j^t$ (donde j representa el píxel y t el *bin* temporal) combinando estimadores ruidosos $\hat{\theta}_i^\tau$ espacio-temporalmente cercanos (donde i es el píxel y τ el *bin* temporal) mediante

$$\tilde{\theta}_j^t = \sum_{\tau} \sum_i w_{ij}^{\tau t} \hat{\theta}_i^\tau, \quad (5.1)$$

donde $w_{ij}^{\tau t}$ son los pesos asignados al estimador $\hat{\theta}_i^\tau$.

El *denoiser* estadístico debe cumplir una serie de propiedades clave: simetría, convergencia con el número de muestras, e identidad consigo mismo [30]. Además, la decisión de si un par de estimadores $(\hat{\theta}_i^\tau, \hat{\theta}_j^t)$ debe ser combinado o no solo puede depender de las estadísticas de dicho par de estimadores. Por último, las estadísticas deben ser calculadas de forma en línea (*online*) para prevenir requisitos prohibitivos de memoria y rendimiento.

Los filtros clásicos como un filtro gaussiano 3D o un *Joint Bilateral Filter* no analizan la entrada de estimadores y utilizan pesos predefinidos independientes de las muestras. En cambio, adaptamos los estimadores estadísticos de estado estacionario al dominio temporal para *denoise* imágenes controladas temporalmente. Los estimadores transitorios se obtienen a partir de la media, los momentos centrales M_l^t (donde l es el orden del momento), y la asimetría o *skewness* ($M_3/M_2^{3/2}$). Los momentos centrales [18] pueden calcularse como

$$M_l^t = \frac{1}{n} \sum_j^n \left(X_j^t - \bar{X}^t \right)^l, \quad (5.2)$$

donde X_j^t representa cada muestra j del *bin* temporal t , y $\bar{X}^t$ es la media muestral del *bin* temporal t .

Desafortunadamente, la recolección de estos estimadores $\hat{\theta}_j^t$ de forma ingenua es impráctica debido a las elevadas restricciones de memoria. En la siguiente Sección se muestra cómo recolectarlas de forma eficiente para renderizado transitorio.

5.2. Obtención eficiente de las estadísticas

Recordemos que para renderizar eficientemente el transporte de luz en estado transitorio es necesario reutilizar caminos (*path reuse*, ver Figura 2.2). En lugar de renderizar cada *bin* temporal de forma independiente, lo que resultaría en repetir el proceso de renderizado T veces (una vez por cada *bin* temporal, ver Figura 2.2b), *path reuse* estima simultáneamente toda la dimensión temporal de una sola vez, almacenando la contribución de cada camino de luz en el *bin* temporal correspondiente según el retardo temporal del camino. Como resultado, cada camino de luz $\mathbf{x}$ contribuye a todo el dominio temporal T (ver Figura 2.2c).

En el software *mitransient*, se vectorizan todos los caminos de luz simultáneamente. Por ejemplo, si tenemos una cámara de 1024×1024 píxeles y deseamos 1024 muestras por píxel, se vectorizarían 1024^3 caminos de luz. Para cada camino de luz se iteran los rebotes, calculando en cada rebote la intensidad I de la luz y el tiempo de vuelo del camino, y volcando la intensidad I en el *bin* temporal correspondiente.

Una aproximación ingenua consistiría en almacenar todas las contribuciones de luz de cada camino por *bin* temporal, de forma similar a la solución aplicada en la Capítulo 4, donde se guardan todas las muestras individuales vectorizadas para calcular posteriormente la media y los momentos centrales de forma sencilla. Sin embargo, al añadir la dimensión temporal, esta estrategia presenta problemas críticos de memoria. Considerando una escena con las siguientes dimensiones: $H = W = T = \text{spp} = 1024$, donde

cada contribución de luz utiliza un `float32`, esto resultaría en un uso impráctico de memoria de 4 TB únicamente para almacenar las contribuciones de luz.

Dado que no es posible guardar cada muestra individual por limitaciones de memoria, la siguiente alternativa sería utilizar un algoritmo en línea como el algoritmo de Welford [38]. Sin embargo, como se mencionó anteriormente, este tampoco es viable en Mitsuba 3. Esto se debe a la evaluación concurrente de múltiples muestras en un *framework* tensorizado [14], donde el número de caminos que actualizan simultáneamente las estadísticas de un determinado *bin* temporal es desconocido. Esto requeriría mantener una dimensión temporal para obtener las estadísticas por lotes antes de volcarlas a los valores transitorios finales, conduciendo nuevamente a requisitos de memoria imprácticos.

La única solución viable es el cálculo de la media y los momentos centrales mediante el almacenamiento de las sumas acumuladas de las muestras. Para ello, durante el renderizado se acumulan X_1, X_2, X_3 por píxel y *bin* temporal como

$$X_1 = \sum_{j=1}^n x_j, \quad X_2 = \sum_{j=1}^n x_j^2, \quad X_3 = \sum_{j=1}^n x_j^3, \quad (5.3)$$

donde x_j representa cada muestra transformada. Con estas sumas acumuladas podemos calcular posteriormente las estadísticas necesarias,

$$\mu = \frac{X_1}{n} \quad (5.4)$$

$$\sigma^2 = \frac{n}{n-1} \left(\frac{X_2}{n} - \mu^2 \right), \quad (5.5)$$

$$M_3 = \frac{X_3}{n} - 3\mu\sigma^2 - \mu^3, \quad (5.6)$$

que permiten obtener el estimador final θ_j^t y la varianza $\text{Var}(\hat{\theta}_j^t)$ (ver Algoritmo 5).

Usar este método genera otro desafío, dado que para las estadísticas se realizan dos tipos de transformaciones no lineales, es decir, sumar las contribuciones y luego realizar las transformaciones sobre esta suma no es equivalente a hacer la transformación a la muestra del píxel y *bin* temporal y luego sumarlas. Estas transformaciones son: la primera es una transformación de normalización de las muestras Box-Cox [3] o Yeo-Johnson [41]. Es importante normalizar las muestras, ya que las distribuciones de un *render* suelen ser asimétricas hacia la derecha (*right-skewed*), con la mayoría de los valores en 0 y unas pocas muestras con contribuciones muy grandes. La segunda transformación son las funciones cuadrática y cúbica para los momentos centrales.

Además, los rebotes cercanos ($\mathbf{x}_{k-1}, \mathbf{x}_k, \mathbf{x}_{k+1}$) usualmente contribuyen al mismo *bin* temporal, por lo que es necesario sumar dichas contribuciones que pertenezcan al mismo *bin* temporal en cada camino antes de aplicarles las transformaciones no lineales y acumular las contribuciones en (X_1, X_2, X_3) .

Para solucionar esto, se acumulan las contribuciones del camino de luz $\bar{\mathbf{x}}$ en un único valor escalar L mientras contribuya al mismo *bin* temporal, descrito en el Algoritmo 4. Una vez que el retardo temporal $\bar{t}$ del vértice del camino de luz $\mathbf{x}_k$ es mayor que el ancho del *bin* temporal actual (cuando el *bin* temporal cambia), se vuelca la contribución L en el valor transitorio final y se actualizan sus estadísticas. De esta forma, únicamente se mantiene un valor por cada camino de luz $\bar{\mathbf{x}}$, reduciendo por un factor de T el consumo de memoria. Siguiendo el ejemplo anterior, esto se reduciría por un factor de 1024, resultando en un uso de memoria significativamente más práctico de 4 GB.

Algoritmo 4: Recolección de estadísticas. En la función **sample** de un integrador Monte Carlo, se utiliza un único valor escalar L por muestra, acumulando en él la contribución del camino hasta que el retardo temporal excede el ancho del *bin* temporal actual h . Entonces, se vuelcan los datos en los valores transitorios finales mediante la función **update_stats**, que sincroniza los volcados paralelos mediante la función **sync_accum**.

```

1: def sample (...):
2:      $L = 0$ ,  $\mathfrak{T} = 1$ 
3:      $t_{\text{last\_bin}} = -1$ ,  $\bar{t} = 0$ 
4:     while camino activo :
5:         surface_it = ray_intersect(...)
6:         # Retardo de propagación entre vértices del camino
7:          $\bar{t} += \text{surface\_it.tof}$ 
8:         ...
9:         # Estimación del siguiente evento
10:        dir_sample = emitter.sample(surface_it)
11:         $t_{\text{bin}} = \lfloor (\bar{t} + \text{dir\_sample.tof} - t_{\text{start}}) / h \rfloor$ 
12:        if  $t_{\text{bin}} \neq t_{\text{last\_bin}}$  :
13:            update_stats( $L$ ,  $t_{\text{last\_bin}}$ , ...)
14:             $L = 0$ ,  $t_{\text{last\_bin}} = t_{\text{bin}}$ 
15:             $L += \mathfrak{T} \cdot \text{dir\_sample}.L$ 
16:            ...
17:        update_stats( $L$ ,  $t_{\text{last\_bin}}$ , ...)

14: def update_stats ( $L$ ,  $t_{\text{bin}}$ ,  $xy$ ):
15:      $v = \text{transform}(L)$ 
16:      $\text{non\_zero} += \text{sync\_accum}(1, t_{\text{bin}}, xy)$ 
17:      $(X_1 X_2 X_3) += \text{sync\_accum}((v v^2 v^3), t_{\text{bin}}, xy)$ 

```

Algoritmo 5: Cálculo del estimador θ . Se calculan el estimador final θ y la varianza $\text{Var}(\hat{\theta})$ a partir de los momentos centrales (μ, σ^2, M_3) de las estadísticas X_1, X_2, X_3 , las muestras por píxel (spp) y el conteo de muestras no nulas (non_zero).

```

1: def compute_estimators ( $X_1, X_2, X_3$ , spp, non_zero):
2:      $z = \text{transform}(0)$ 
3:     # Rellenado con ceros transformados
4:      $(X_1, X_2, X_3) += (\text{spp} - \text{non\_zero}) (z, z^2, z^3)$ 
5:      $\mu = X_1 / n$ 
6:      $\sigma^2 = \frac{n}{n-1} (X_2 / n - \mu^2)$ 
7:      $M_3 = X_3 / n - 3 \mu \sigma^2 - \mu^3$ 
8:     # Corrección de medias por asimetría [6]
9:      $\theta \approx \mu + M_3 / (6 \sigma^2 n)$ 
10:     $\text{Var}(\hat{\theta}) = \sigma^2 / n$ 

```

5.3. Denoising estadístico

Dada la formulación estadística para renderizado transitorio (Sección 5.1) y el algoritmo eficiente para recolectar estadísticas (Sección 5.2), ahora se presenta cómo combinarlos para el *denoising* en estado transitorio.

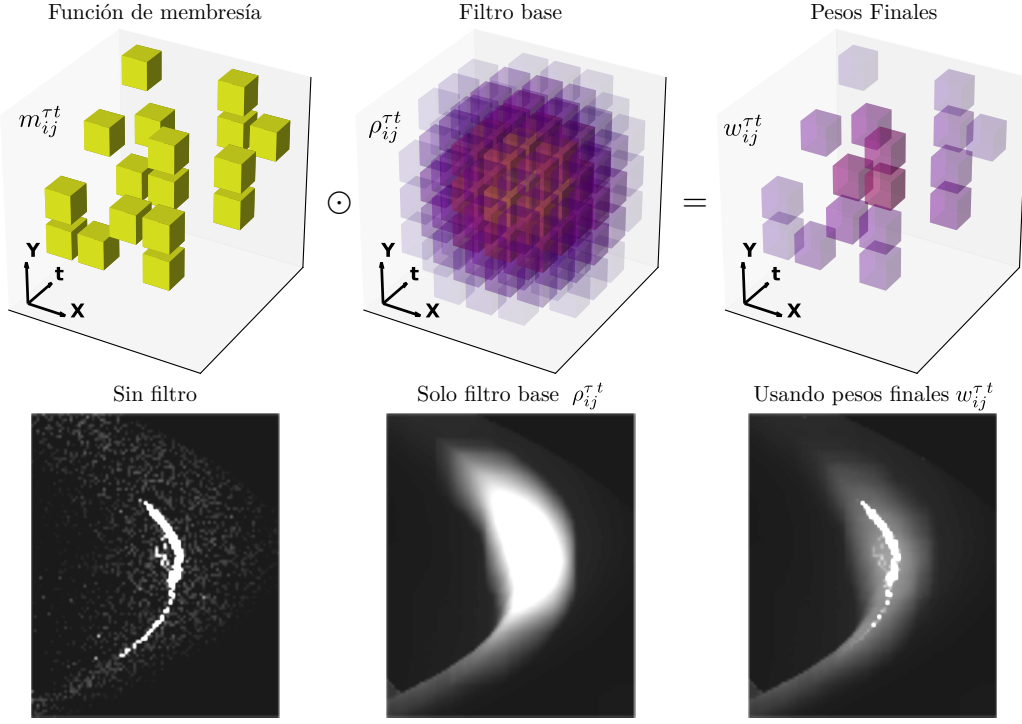


Figura 5.2: Denoising espacio-temporal. Se aprovecha la correlación espacio-temporal del transporte de luz transitorio para el *denoising*. El *denoiser* está compuesto por una función de membresía $m_{ij}^{\tau t}$ que decide qué píxeles transitorios deben combinarse, y un filtro base $\rho_{ij}^{\tau t}$, cuya combinación produce los pesos finales $w_{ij}^{\tau t}$. Mientras que el renderizado original es ruidoso y el filtro base añade una cantidad significativa de sesgo, los pesos propuestos logran un compromiso casi óptimo entre sesgo y ruido.

En renderizado transitorio, dado que las correlaciones surgen en el dominio espacio-temporal, se busca obtener los pesos $w_{ij}^{\tau t}$ que logren un compromiso entre sesgo y varianza en dicho espacio de mayor dimensionalidad. Teniendo en cuenta la dimensión temporal, los pesos óptimos w^* que minimizan el error cuadrático medio (MSE) se obtienen como

$$w^* = \frac{2(\theta_i^\tau - \theta_j^t)^2 + \text{Var}(\hat{\theta}_i^\tau) + \text{Var}(\hat{\theta}_j^t)}{2\left((\theta_i^\tau - \theta_j^t)^2 + \text{Var}(\hat{\theta}_i^\tau) + \text{Var}(\hat{\theta}_j^t)\right)}. \quad (5.7)$$

Sin embargo, los estimadores son ruidosos, lo que produce pesos ruidosos, y utilizarlos directamente introduciría sesgo adicional. Este sesgo puede limitarse mediante la imposición de funciones de membresía binarias

$$m_{ij}^{\tau t} = \begin{cases} 1 & \text{si } (1 - w^*) > \gamma \\ 0 & \text{en caso contrario,} \end{cases} \quad (5.8)$$

donde el umbral γ representa la discriminatividad de $m_{ij}^{\tau t}$, de forma similar a un test t de Welch. Las funciones de membresía $m_{ij}^{\tau t}$ discriminan adaptativamente qué estimadores $\hat{\theta}_j^t$ deben combinarse. Dos estimadores $(\hat{\theta}_j^t, \hat{\theta}_i^\tau)$ se combinan únicamente cuando la diferencia entre sus medias (sesgo introducido) es pequeña en relación con la suma de sus varianzas (ruido presente).

Para mejorar la calidad de imagen, se combinan de nuevo las funciones de membresía $m_{ij}^{\tau t}$ con filtros espacio-temporales existentes $\rho_{ij}^{\tau t}$, como el *Joint Bilateral Filter* (JBF), que utiliza características de bajo nivel (*low-level features*) de los *G-Buffers*, tales como la normal geométrica o el albedo de textura para filtrar vecinos, al cual para analizar la correlación espacio-temporal se añade un radio temporal. Los *G-Buffers* se replican a través de la dimensión temporal, ya que sus valores permanecen constantes para cada píxel espacial. Finalmente, para calcular los pesos finales $w_{ij}^{\tau t}$ de la Ecuación (5.1), se combinan las funciones de membresía $m_{ij}^{\tau t}$ con el filtro espacio-temporal $\rho_{ij}^{\tau t}$ mediante

$$w_{ij}^{\tau t} = \frac{\rho_{ij}^{\tau t} m_{ij}^{\tau t}}{\sum_{\tau} \sum_i \rho_{ij}^{\tau t} m_{ij}^{\tau t}}, \quad (5.9)$$

cuyo resultado se ilustra en la Figura 5.2.

Medios participativos. Para evaluar el *denoiser* en escenas volumétricas, implementamos el Algoritmo 4 en el integrador `TransientPrbVolPath` de `mitransient` (ver Figura 5.1), que proporciona soporte para medios participativos.

En este contexto, el *denoiser* requiere una adaptación: el filtro base $\rho_{ij}^{\tau,t}$ se reemplaza por un filtro gaussiano 3D, confiando exclusivamente en las funciones de membresía $m_{ij}^{\tau,t}$ para discriminar qué estimadores combinar. Esta modificación es necesaria porque el JBF, diseñado para superficies, discriminaría erróneamente los vecinos basándose en características de los *G-Buffers* que no son apropiadas para medios participativos.

5.4. Resultados

En esta Sección se muestra el rendimiento del *denoiser* transitorio estadístico en diversas escenas. Como se ha mencionado anteriormente, el algoritmo de recolección de estadísticas está implementado en `mitransient`, una biblioteca de renderizado transitorio desarrollada sobre los *frameworks* `Dr.Jit` y `Mitsuba 3`. El *denoiser* transitorio estadístico está implementado en `PyTorch` [24] de forma análoga a lo descrito en la Sección 4.2, donde la modificación más importante es que tanto el teselado como la obtención de los vecinos se realiza a través de 3 dimensiones (dos espaciales y una temporal) en lugar de solo 2 dimensiones.

Los resultados se comparan con dos enfoques diferentes: el *Joint Bilateral Filter* (JBF) utilizado como filtro base $\rho_{ij}^{\tau,t}$ en nuestro método (para verificar la importancia de la función de membresía $m_{ij}^{\tau,t}$), y `OptiX` (NVIDIA `OptiX` AI-Accelerated Denoiser), un *denoiser* basado en aprendizaje profundo considerado estado del arte. Se utiliza el *denoising* temporal tratando los datos transitorios como un vídeo, lo que mejora el rendimiento de `OptiX`.

La Figura 5.3 muestra los resultados comparando los distintos métodos previos contra nuestro método con una imagen de referencia computada con 30k muestras por píxel (spp). El JBF suaviza excesivamente la mayoría de los detalles finos de la imagen, ya que combina píxeles transitorios basándose en sus normales y albedos, ignorando la iluminación real. En contraste, nuestro método discrimina qué píxeles transitorios deben combinarse basándose en las estadísticas transitorias, que incorporan información de iluminación y, por tanto, preservan mejor los detalles finos.

`OptiX` también falla al reconstruir algunos detalles finos, llegando incluso a alucinar otros, como puede observarse en el jarrón y los reflejos en la escena `BATHROOM` (fila superior), la sombra en la escena `KITCHEN` o el papel de pared detrás de las lámparas en `STAIRCASE`. Nuestro *denoiser* es más fiable ya que no requiere un conjunto de datos para entrenamiento, utilizando únicamente las estadísticas transitorias recolectadas durante el proceso de renderizado.

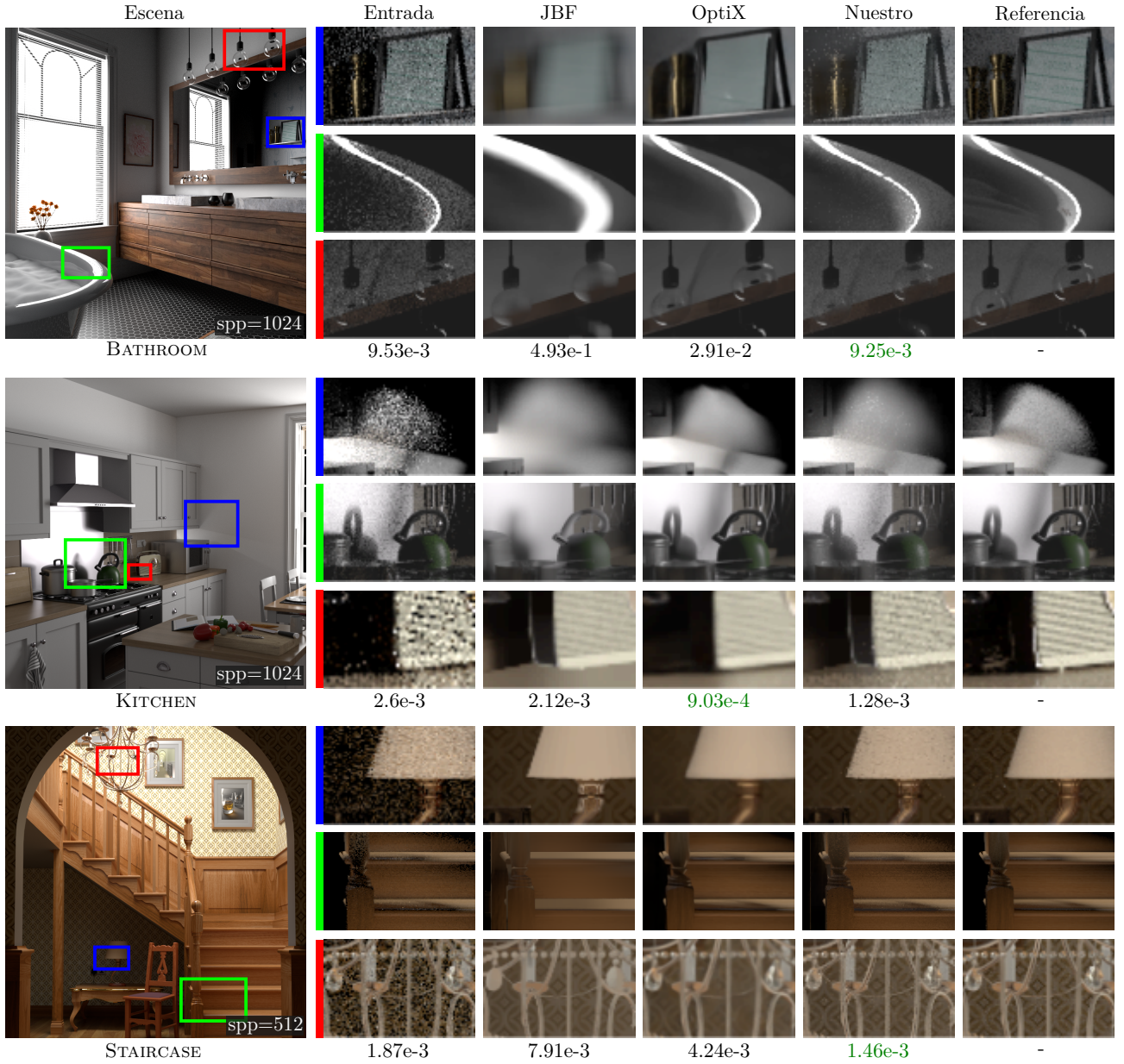


Figura 5.3: Comparación de resultados. Se compara el *denoiser* transitorio espacio-temporal basado en estadísticas con un *Joint Bilateral Filter* (JBF) (el filtro base utilizado en nuestro método) y OptiX (NVIDIA OptiX AI-Accelerated Denoiser). Se utiliza el RMSE para comparar cada método con la referencia, y cada *inset* muestra un *bin* temporal diferente. El JBF difumina la mayoría de los detalles de la imagen, llegando a eliminar muchas características de la escena. OptiX busca obtener imágenes visualmente más agradables a costa de introducir sesgo (véanse, por ejemplo, el reflejo del espejo en BATHROOM, la sombra en KITCHEN o el papel de pared en STAIRCASE), además de alucinar detalles (como la discontinuidad de la cáustica). El *inset* de la bañera está renderizado con 128 spp. Para ver los resultados de estas escenas en vídeo: [vídeo de resultados del denoiser estadístico transitorio](#).

5.4.1. Análisis de parámetros

Se analizan los componentes y propiedades del *denoiser* transitorio estadístico propuesto. En concreto, se analiza la influencia de las transformaciones de muestras y los parámetros estadísticos (Figura 5.4), la influencia del radio espacio-temporal (Figura 5.5), y la convergencia respecto al número de muestras por píxel (Figura 5.6).

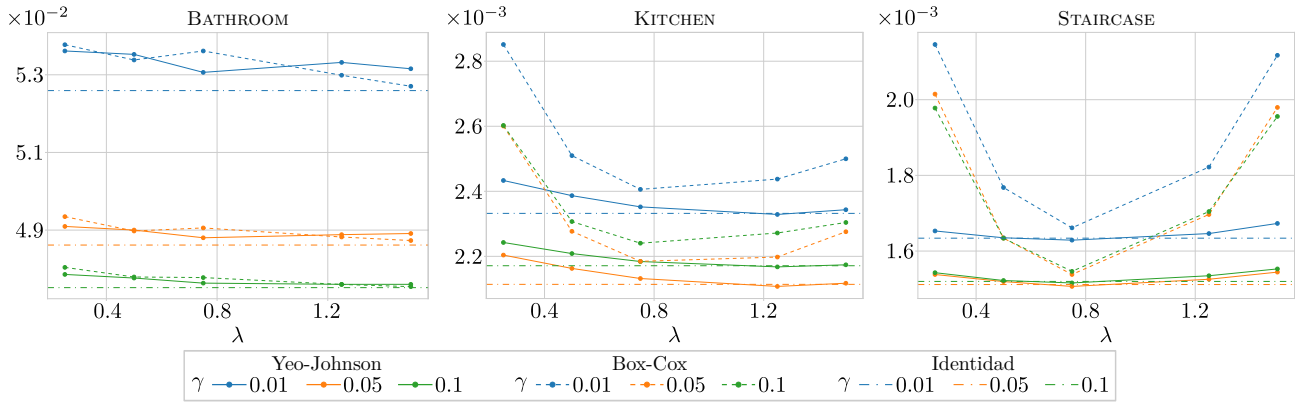


Figura 5.4: Error variando parámetros estadísticos. Se analiza para las escenas KITCHEN, BATHROOM y STAIRCASE la influencia de variar los parámetros y transformaciones de muestras en el error cuadrático medio (RMSE): se varía la función de transformación de muestras entre las transformaciones Identidad, Box-Cox [3] y Yeo-Johnson [41] con diferentes parámetros λ (eje x), y umbral de membresía γ . En renderizado transitorio, dado que la distribución de muestras es diferente a la del renderizado estacionario, la mejor función de transformación de muestras por no es Box-Cox sino la transformación Identidad o Yeo-Johnson con γ entre 0,05 y 0,1.

Transformaciones de muestras. En la Figura 5.4 se analiza el rendimiento de las transformaciones Identidad, Box-Cox [3] y Yeo-Johnson [41], variando el parámetro de transformación $\lambda \in [0,25, 1,50]$. También se analiza la influencia del umbral $\gamma = \{0,01, 0,05, 0,1\}$. Para este análisis se fija el radio espacial a 5 píxeles y el radio temporal a 1 *bin* temporal.

La distribución de muestras depende en gran medida de las características de la escena. Se ha observado que, para renderizado transitorio, la transformación Identidad y la transformación Yeo-Johnson con un γ entre 0.05 y 0.1 obtienen de forma consistente el mejor rendimiento en las escenas analizadas. Esto difiere del *denoising* estadístico estacionario [30], donde la transformación Box-Cox parecía ser la mejor opción.

Esta diferencia puede deberse a que la mayoría de las contribuciones de luz que llegan al sensor en renderizado transitorio son cercanas a cero. La transformación Box-Cox aplica directamente una transformación de potencia a x

$$\text{Box-Cox } (x, \lambda) = \frac{x^\lambda - 1}{\lambda}, \quad \lambda \neq 0, x > 0, \quad (5.10)$$

lo que provoca una transformación desproporcionada con valores cercanos a 0. En contraste, la transformación Identidad no realiza ninguna transformación, mientras que la transformación Yeo-Johnson evita dicha distorsión desplazando el valor x antes de realizar la transformación de potencia, definida como

$$\text{Yeo-Johnson } (x, \lambda) = \frac{(x+1)^\lambda - 1}{\lambda}, \quad \lambda \neq 0, x \geq 0. \quad (5.11)$$

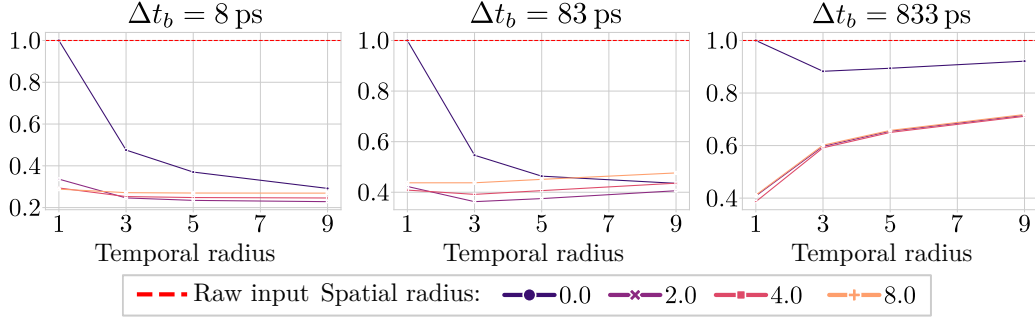


Figura 5.5: Análisis del radio temporal y anchura de bin h . Se muestra la evolución del RMSE variando el radio temporal del *kernel* para diferentes radios espaciales. Aprovechar la correlación espacio-temporal es fundamental para reducir el error en *bins* temporales estrechos ($h = 8$ ps). A medida que la anchura del *bin* aumenta ($h = 83$ ps), la correlación espacio-temporal pierde relevancia en favor de la correlación puramente espacial. Con *bins* temporales anchos ($h = 833$ ps), solo se necesita la correlación espacial, ya que los renderizados transitorios presentan una correlación espacial similar a los renderizados estacionarios y una correlación temporal muy débil. Evaluado en la escena KITCHEN.

Denoising espacio-temporal. En la Figura 5.5 se analiza la influencia de los *kernels* espacio-temporales (diferentes radios espaciales y temporales) sobre el RMSE para diferentes anchuras de *bin* temporal h . Se observa que explotar la correlación espacio-temporal del transporte de luz es crucial cuando h es pequeño. Sin embargo, conforme aumenta h , la correlación temporal pierde relevancia respecto a la espacial, ya que conforme aumenta h el renderizado se aproxima más a uno estacionario. Para este análisis se han usado renderizados de 512 spp, la transformación Identidad y un umbral $\gamma = 0,05$.

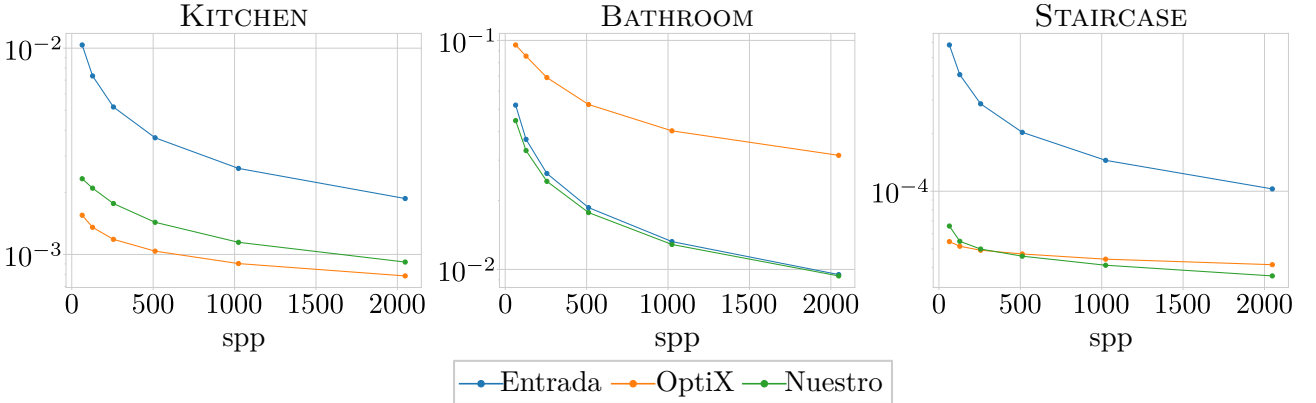


Figura 5.6: Convergencia con muestras por píxel. Se evalúa la convergencia del método que proponemos frente a OptiX, variando el número de muestras por píxel en las tres escenas. El método propuesto converge con el aumento del número de muestras. Sin embargo, mientras que OptiX no es fiable para mejorar el resultado – incluso aumenta el error en la escena BATHROOM –, el método propuesto mejora consistentemente el RMSE. Esto se debe al sesgo introducido por OptiX, mientras que el método propuesto utiliza únicamente las estadísticas presentes en el renderizado para lograr un compromiso entre sesgo y ruido.

Convergencia. En la Figura 5.6 se analiza la convergencia del método propuesto en las escenas KITCHEN, BATHROOM y STAIRCASE, y se compara con OptiX. El método propuesto mejora consistentemente el error cuadrático medio (RMSE), mientras que el rendimiento de OptiX depende en gran medida de la escena, llegando incluso a tener un rendimiento significativamente peor que la entrada ruidosa original (véase la escena BATHROOM). Esto se debe probablemente a que OptiX alucina en áreas con detalles muy finos, como se mostró en la Figura 5.3, añadiendo así sesgo considerable.

5.4.2. Resultados volumétricos

La formulación estadística para *denoising* en renderizado transitorio se generaliza de forma natural a renderizado volumétrico transitorio con medios participativos. Es importante destacar que el

renderizado de medios participativos requiere un integrador diferente al utilizado para el trazado de caminos estándar. Específicamente, es necesario usar el integrador `TransientPrbVolPath` en lugar de `TransientPath` (ver Figura 5.1).

En la Figura 5.7 se muestra el rendimiento del método propuesto para las escenas BANNER y DISNEY CLOUD, comparándolo frente al filtro base $\rho_{ij}^{\tau, t}$ utilizado en renderizado volumétrico, un filtro gaussiano 3D. Se puede observar cómo, mientras que el filtro gaussiano 3D suaviza excesivamente la imagen, el método propuesto preserva mejor los detalles estructurales finos.

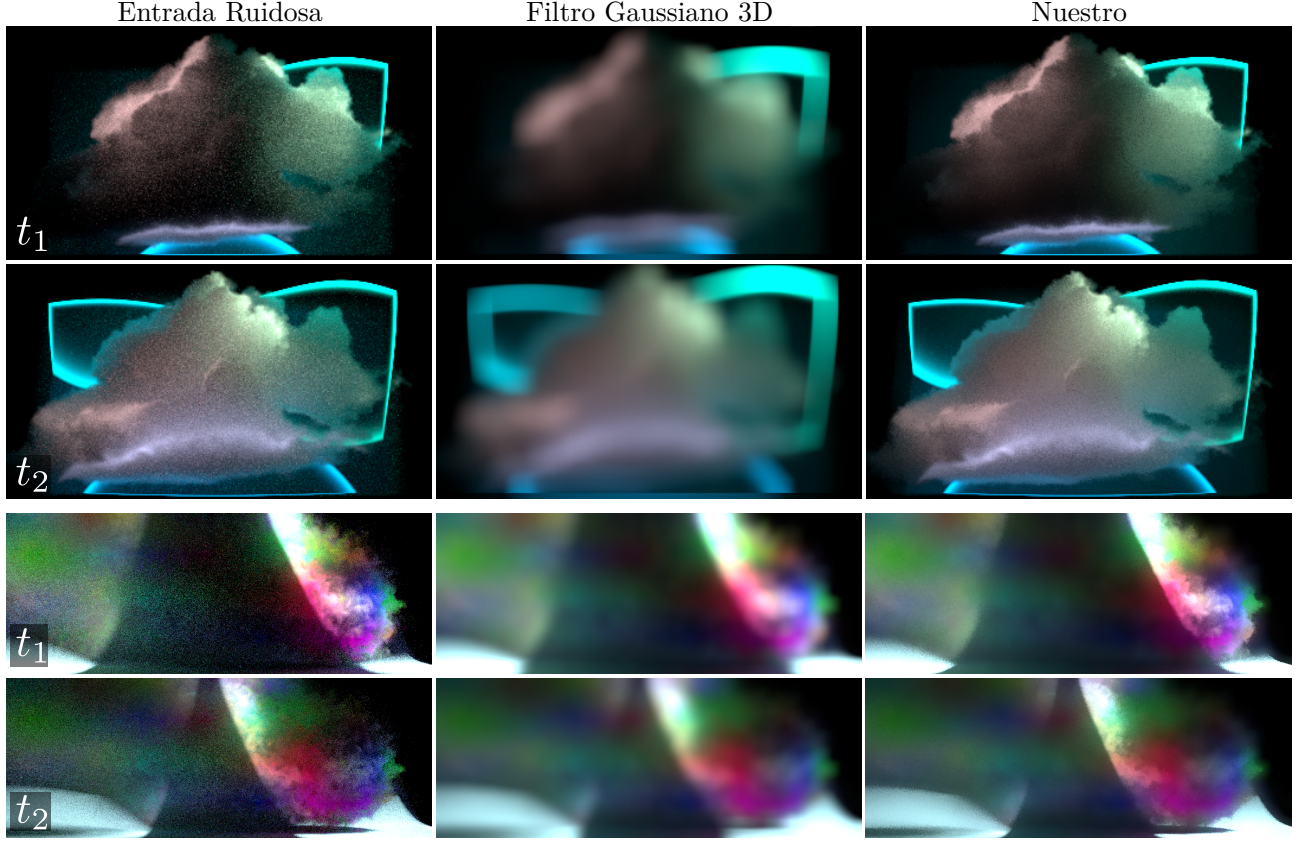


Figura 5.7: Denoising de medios participativos. Se compara el método propuesto con el filtro gaussiano 3D base en el volumen DISNEY CLOUD con tres luces (rosa, verde, azul) y la escena BANNER con iluminación únicamente lateral, renderizadas con 60k y 1024 spp, respectivamente. Mientras que el filtro gaussiano simplemente difumina la imagen, el método propuesto mejora la calidad de imagen discriminando qué *bins* temporales deben combinarse para preservar detalles espacio-temporales finos mientras se reduce el ruido en regiones de alto nivel de ruido.

Capítulo 6

Conclusiones y trabajo futuro

El transporte de luz transitorio tiende a producir resultados ruidosos debido a que las mismas muestras en estado estacionario se reparten entre todo el dominio temporal. Este trabajo extiende las técnicas basadas en estadísticas para *denoising* de simulaciones de Monte Carlo al dominio transitorio, ofreciendo una solución eficiente que minimiza el ruido mientras que preserva los detalles finos de la propagación de la luz a lo largo del tiempo.

Existen diferentes direcciones para mejorar el rendimiento de este método. Confiar en tests estadísticos de estimadores ruidosos ofrece una solución eficiente sin depender de datos de entrenamiento, aunque el ruido de estos estimadores puede afectar al rendimiento de dichos tests estadísticos. Una posible solución podría ser un enfoque híbrido, donde el filtro espacio-temporal base utilizado, como el *Joint Bilateral Filter*, sea reemplazado por una red neuronal específica para *denoising* de renderizados transitorios. Esta red, combinada con las funciones de membresía, podría obtener resultados excelentes: el filtro base se basaría en datos de entrenamiento, mientras que las funciones de membresía se fundamentarían en tests estadísticos sólidos que permiten obtener resultados correctos y que convergen a la solución adecuada.

Otro enfoque sería utilizar directamente estas estadísticas como entrada en redes neuronales como puede ser OptiX (NVIDIA OptiX AI-Accelerated Denoiser) [5] o OIDN (Intel Open Image Denoise) [1]. Normalmente las redes neuronales entrenadas para *denoising* generalmente toman la imagen ruidosa e información de bajo nivel (*low-level*) del renderizado, lo que les proporciona una base más sólida al disponer de información real. Por ejemplo, muchas utilizan los *G-Buffers*, empleando las normales y albedos de los materiales de la escena para obtener más información de bajo nivel y conseguir mejores resultados. A esta entrada se le podrían añadir las propias estadísticas extraídas del renderizado y evaluar qué resultados consigue una red neuronal con toda esta información.

Más allá de estas posibles mejoras, nuestro trabajo puede contribuir a la ayuda del *denoising* de simulaciones temporales en diferentes dominios, donde la precisión es preferible a resultados visualmente agradables, pero posiblemente sesgados. El método propuesto, al no depender de técnicas de aprendizaje automático y garantizar una convergencia a la solución correcta, podría aplicarse en ámbitos donde no es posible generar una gran cantidad de datos de entrenamiento como son: simulaciones acústicas [8], imagen térmica [33] o espectroscopía transitoria [19], así como a dominios que comparten similitudes con el control temporal, como simulaciones de detección heterodina óptica [20] o imagen fuera de visión [27].

Bibliografía

- [1] Attila T. Áfra. Intel® Open Image Denoise, 2025. <https://www.openimagedenoise.org>.
- [2] Malik Boughida and Tamy Boubekeur. Bayesian collaborative denoising for monte carlo rendering. *Computer Graphics Forum*, 36(4):137–153, 2017. doi: <https://doi.org/10.1111/cgf.13231>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13231>.
- [3] G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)*, 26(2):211–243, 12 2018. ISSN 0035-9246. doi: [10.1111/j.2517-6161.1964.tb00553.x](https://doi.org/10.1111/j.2517-6161.1964.tb00553.x). URL <https://doi.org/10.1111/j.2517-6161.1964.tb00553.x>.
- [4] G. Bradski. The OpenCV Library. *Dr. Dobbs's Journal of Software Tools*, 2000.
- [5] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4), July 2017. ISSN 0730-0301. doi: [10.1145/3072959.3073601](https://doi.org/10.1145/3072959.3073601). URL <https://doi.org/10.1145/3072959.3073601>.
- [6] José Dias Curto. Confidence intervals for means and variances of nonnormal distributions. *Communications in Statistics - Simulation and Computation*, 52(9):4414–4430, 2023. doi: [10.1080/03610918.2021.1963448](https://doi.org/10.1080/03610918.2021.1963448). URL <https://doi.org/10.1080/03610918.2021.1963448>.
- [7] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch. Edge-Avoiding À-TrousWavelet Transform for fast Global Illumination Filtering. In Michael Doggett, Samuli Laine, and Warren Hunt, editors, *High Performance Graphics*. The Eurographics Association, 2010. ISBN 978-3-905674-26-2. doi: [/10.2312/EGGH/HPG10/067-075](https://doi.org/10.2312/EGGH/HPG10/067-075).
- [8] Ugo Finnendahl, Markus Worchel, Tobias Jüterbock, Daniel Wujecki, Fabian Brinkmann, Stefan Weinzierl, and Marc Alexa. Differentiable geometric acoustic path tracing using time-resolved path replay backpropagation. *ACM Trans. Graph.*, 44(4), July 2025. ISSN 0730-0301. doi: [10.1145/3730900](https://doi.org/10.1145/3730900). URL <https://doi.org/10.1145/3730900>.
- [9] Arthur Firmino, Jeppe Revall Frisvad, and Henrik Wann Jensen. Progressive denoising of monte carlo rendered images. *Computer Graphics Forum*, 41(2):1–11, 2022. doi: <https://doi.org/10.1111/cgf.14454>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14454>.
- [10] {Jay S.} Gondek, {Gary W.} Meyer, and {Jonathan G. } Newman. Wavelength dependent reflectance functions. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994*, Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, pages 213–220. Association for Computing Machinery, Inc, July 1994. doi: [10.1145/192161.192202](https://doi.org/10.1145/192161.192202). Publisher Copyright: © ACM 1994.; 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994 ; Conference date: 24-07-1994 Through 29-07-1994.
- [11] Diego Gutierrez, Francisco Seron, Adolfo Muñoz, and Oscar Anson. Visualizing underwater ocean optics. *Computer Graphics Forum (Proc. of EUROGRAPHICS)*, 27(2):547–556, 2008.

- [12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL <https://doi.org/10.1038/s41586-020-2649-2>.
- [13] Felix Heide, Matthias B. Hullin, James Gregson, and Wolfgang Heidrich. Low-budget transient imaging using photonic mixer devices. *ACM Trans. Graph.*, 32(4), July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461945. URL <https://doi.org/10.1145/2461912.2461945>.
- [14] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.jit: a just-in-time compiler for differentiable rendering. *ACM Trans. Graph.*, 41(4), July 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530099. URL <https://doi.org/10.1145/3528223.3530099>.
- [15] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. Mitsuba 3 renderer, 2022. <https://mitsuba-renderer.org>.
- [16] Adrian Jarabo, Julio Marco, Adolfo Munoz, Raul Buisan, Wojciech Jarosz, and Diego Gutierrez. A framework for transient rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 33(6), November 2014. doi: 10/gfznb8.
- [17] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15902. URL <https://doi.org/10.1145/15886.15902>.
- [18] J.F. Kenney and E.S. Keeping. *Mathematics of Statistics*. Number parte 2. Van Nostrand, 1951. URL <https://books.google.es/books?id=AH4GxAEACAAJ>.
- [19] Juhyeon Kim, Joshua Multhaup, Mahima Sneha, and Adithya Pediredla. Efficient time sampling strategy for transient absorption spectroscopy. In *2024 IEEE International Conference on Computational Photography (ICCP)*, pages 1–12, 2024. doi: 10.1109/ICCP61108.2024.10644518.
- [20] Juhyeon Kim, Craig Benko, Magnus Wrenninge, Ryusuke Villemin, Zeb Barber, Wojciech Jarosz, and Adithya Pediredla. A Monte Carlo rendering framework for simulating optical heterodyne detection. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 44(4), August 2025. doi: 10/px43.
- [21] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM ’15*, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL <https://doi.org/10.1145/2833157.2833162>.
- [22] Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. Sure-based optimization for adaptive sampling and reconstruction. *ACM Trans. Graph.*, 31(6), November 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366213. URL <https://doi.org/10.1145/2366145.2366213>.
- [23] Ryan S. Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive wavelet rendering. *ACM Trans. Graph.*, 28(5):1–12, December 2009. ISSN 0730-0301. doi: 10.1145/1618452.1618486. URL <https://doi.org/10.1145/1618452.1618486>.
- [24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit

- Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [25] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. MIT Press, 4th edition, 2023.
- [26] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive rendering with non-local means filtering. *ACM Trans. Graph.*, 31(6), November 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366214. URL <https://doi.org/10.1145/2366145.2366214>.
- [27] Diego Royo, Jorge García, Adolfo Muñoz, and Adrian Jarabo. Non-line-of-sight transient rendering. *Computers & Graphics*, 2022. ISSN 0097-8493. doi: <https://doi.org/10.1016/j.cag.2022.07.003>. URL <https://www.sciencedirect.com/science/article/pii/S0097849322001200>.
- [28] Diego Royo, Miguel Crespo, and Jorge Garcia-Pueyo. mitransient. <https://github.com/diegoroyo/mitransient>, 2023.
- [29] Iman Sadeghi, Adolfo Munoz, Philip Laven, Wojciech Jarosz, Francisco Seron, Diego Gutierrez, and Henrik Wann Jensen. Physically-based simulation of rainbows. *ACM Trans. Graph.*, 31(1), February 2012. ISSN 0730-0301. doi: 10.1145/2077341.2077344. URL <https://doi.org/10.1145/2077341.2077344>.
- [30] Hiroyuki Sakai, Christian Freude, Thomas Auzinger, David Hahn, and Michael Wimmer. A statistical approach to monte carlo denoising. In *SIGGRAPH Asia 2024 Conference Papers*, SA '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400711312. doi: 10.1145/3680528.3687591. URL <https://doi.org/10.1145/3680528.3687591>.
- [31] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics, HPG '17*, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351010. doi: 10.1145/3105762.3105770. URL <https://doi.org/10.1145/3105762.3105770>.
- [32] Pradeep Sen and Soheil Darabi. On filtering the noise from the random parameters in monte carlo rendering. *ACM Trans. Graph.*, 31(3), May 2012. ISSN 0730-0301. doi: 10.1145/2167076.2167083. URL <https://doi.org/10.1145/2167076.2167083>.
- [33] Kenichiro Tanaka, Nobuhiro Ikeya, Tsuyoshi Takatani, Takuya Funatomi, Hiroyuki Kubo, and Yasuhiro Mukaigawa. Time-resolved light transport decomposition for thermal photometric stereo. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 4804–4813, June 2018.
- [34] E. Veach and Stanford University. Computer Science Department. *Robust Monte Carlo Methods for Light Transport Simulation*. Number n.º 1610 in Report (Stanford University. Computer Science Department). Stanford University, Department of Computer Science, 1998. URL <https://books.google.es/books?id=SaZEAQAIAAJ>.
- [35] Andreas Velten, Di Wu, Adrian Jarabo, Belen Masia, Christopher Barsi, Chinmaya Joshi, Everett Lawson, Mounqi Bawendi, Diego Gutierrez, and Ramesh Raskar. Femto-photography: capturing and visualizing the propagation of light. *ACM Trans. Graph.*, 32(4), July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461928. URL <https://doi.org/10.1145/2461912.2461928>.

- [36] D. Vicini, D. Adler, J. Novák, F. Rousselle, and B. Burley. Denoising deep monte carlo renderings. *Computer Graphics Forum*, 38(1):316–327, 2019. doi: <https://doi.org/10.1111/cgf.13533>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13533>.
- [37] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.
- [38] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi: 10.1080/00401706.1962.10490022. URL <https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022>.
- [39] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. *SIGGRAPH Comput. Graph.*, 26(2):255–264, July 1992. ISSN 0097-8930. doi: 10.1145/142920.134075. URL <https://doi.org/10.1145/142920.134075>.
- [40] Alexander Wilkie, Robert F. Tobler, and Werner Purgathofer. Combined rendering of polarization and fluorescence effects. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques 2001*, pages 197–204, Vienna, 2001. Springer Vienna. ISBN 978-3-7091-6242-2.
- [41] In-Kwon Yeo and Richard A. Johnson. A new family of power transformations to improve normality or symmetry. *Biometrika*, 87(4):954–959, 2000. ISSN 00063444, 14643510. URL <http://www.jstor.org/stable/2673623>.
- [42] Changwen Zheng and Yu Liu. Removing monte carlo noise with compressed sensing and feature information. In *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2018) - GRAPP*, pages 145–153. INSTICC, SciTePress, 2018. ISBN 978-989-758-287-5. doi: 10.5220/0006671601450153.

Lista de Figuras

1.1. Femto-fotografía	2
1.2. Diferencia de ruido entre renderizado estacionario y transitorio.	2
1.3. Denoising de renderizado transitorio.	4
2.1. Ecuación de render.	7
2.2. Renderizado transitorio y reutilización de caminos.	11
3.1. Convergencia del algoritmo de Monte Carlo con diferente número de muestras.	12
3.2. Comparación de diferentes métodos de denoising.	18
4.1. Denoising estacionario.	24
4.2. Resultados del denoising estacionario.	26
5.1. Diagrama de clases de mitransient.	27
5.2. Denoising espacio-temporal.	31
5.3. Comparación de resultados.	33
5.4. Error variando parámetros estadísticos.	34
5.5. Análisis del radio temporal y anchura de bin temporal h	35
5.6. Convergencia con muestras por píxel.	35
5.7. Denoising de medios participativos.	36
B.1. Diagrama de Gantt del proyecto	50

Lista de Tablas

A.1. Tiempos de ejecución del denoising	48
B.1. Horas dedicadas al proyecto	49

Anexos

Anexo A

Rendimiento

En este Anexo se analiza el rendimiento del *denoiser* implementado en PyTorch en diferentes arquitecturas (CPU vectorizada y GPU) y se compara con implementaciones alternativas utilizando otras bibliotecas como NumPy [12] y Numba [21]. También se discute el rendimiento teórico alcanzable basándose en la implementación de más bajo nivel de Sakai et al. [30].

Anexo A.1

Rendimiento de la recolección de estadísticas

La recolección de estadísticas en renderizado transitorio introduce una sobrecarga en el tiempo de ejecución. Esta sobrecarga se debe principalmente a las sincronizaciones necesarias al volcar las estadísticas acumuladas en los valores transitorios finales mediante la función `sync_accum` (ver Algoritmo 4).

En las pruebas realizadas con renderizado volumétrico transitorio (1024 muestras por píxel (spp), 100 bins temporales) en una CPU AMD Ryzen 7 9800X3D 4.7/5.2 GHz y una GPU MSI GeForce RTX 5070 Ti, esta sobrecarga representa aproximadamente 0,03 ms por píxel. Para una escena típica, esto supone un incremento marginal del tiempo total de renderizado. Por ejemplo, para la escena DISNEY CLOUD con resolución $360 \times 960 \times 100$ que tarda aproximadamente 15 minutos en renderizarse con 60k spp, la sobrecarga de recolección de estadísticas representa menos del 1 % del tiempo total.

Anexo A.2

Rendimiento del denoising

Sakai et al. [30] implementan su *denoiser* estacionario utilizando OpenCV [4] con aceleración CUDA directa, logrando tiempos de ejecución muy eficientes mediante *kernels* optimizados de bajo nivel. Para su *denoiser* estacionario con radio espacial de 20 píxeles, consiguen tiempos del orden de 28 ms por fotograma para resoluciones típicas.

Extrapolando al dominio transitorio, el tiempo esperado para *denoise* una secuencia completa sería proporcional al número de bins temporales. Para una resolución de $H \times W \times T = 360 \times 960 \times 100$ con radio espacial 20 y radio temporal 1, el tiempo teórico estimado sería aproximadamente 1,125 s. Este tiempo representa menos del 1 % del tiempo de renderizado transitorio (15 minutos para 60k spp), demostrando la viabilidad práctica de incorporar *denoising* en el pipeline.

Inicialmente, el *denoiser* fue implementado utilizando Numba y NumPy. Numba compila el código Python a código máquina optimizado y aprovecha la vectorización de la CPU, lo que proporciona un rendimiento considerablemente superior al de Python puro. Sin embargo, los tiempos de ejecución resultantes seguían siendo elevados para su uso práctico.

Posteriormente, se implementó el *denoiser* en PyTorch, biblioteca que permite la ejecución eficiente tanto en CPU como en GPU mediante operaciones tensoriales altamente optimizadas. La implementación en PyTorch GPU mostró una mejora considerable respecto a las alternativas anteriores.

La Tabla A.1 presenta los tiempos de ejecución para el *denoising* de una imagen estacionaria de resolución 1280×720 píxeles con diferentes radios de *kernel*, comparando las tres implementaciones: Numba, PyTorch CPU y PyTorch GPU. Se observa que PyTorch GPU ofrece aceleraciones significativas respecto a Numba (del orden de $50\times$ o más) y de aproximadamente $6\times$ respecto a PyTorch CPU.

Configuración	Numba	PyTorch CPU	PyTorch GPU
1280×720 , radio 5	30.60 s	3.24 s	0.54 s
1280×720 , radio 10	100.23 s	13.19 s	0.95 s
1280×720 , radio 20	222.33 s	51.80 s	2.51 s

Cuadro A.1: Tiempos de ejecución del denoising En esta Tabla se muestran los tiempos de ejecución del *denoiser* estacionario para una imagen de 1280×720 píxeles con diferentes radios de *kernel*. Las mediciones se realizaron en una CPU AMD Ryzen 7 9800X3D y una GPU MSI GeForce RTX 5070 Ti.

Es importante destacar que estos tiempos corresponden al *denoising* de un único fotograma estacionario. Para renderizado transitorio con T bins temporales, el tiempo esperado sería aproximadamente T veces mayor. Por ejemplo, para un vídeo transitorio de 100 fotogramas con radio espacial 20 y radio temporal 1, el tiempo de *denoising* en PyTorch GPU sería aproximadamente $100 \times 2,51 = 251$ segundos (~ 4.2 minutos).

Anexo A.3

Conclusiones sobre rendimiento

El *denoiser* implementado en PyTorch GPU ofrece un compromiso razonable entre rendimiento y facilidad de implementación. Aunque el rendimiento actual (~ 2.5 s por fotograma para resolución 1280×720 con radio 20) dista del tiempo teórico óptimo alcanzable con una implementación de bajo nivel como la de Sakai et al. (~ 28 ms por fotograma), sigue siendo suficiente para aplicaciones de investigación y permite procesar escenas de tamaño moderado en tiempos razonables.

Una implementación futura que utilice *kernels* CUDA personalizados, similar a la aproximación de Sakai et al. con OpenCV CUDA, podría reducir los tiempos de ejecución en aproximadamente dos órdenes de magnitud, acercándose al rendimiento teórico óptimo. No obstante, incluso con la implementación actual, la sobrecarga introducida por el *denoising* (minutos) es despreciable comparada con los tiempos de renderizado transitorio (decenas de minutos a horas), validando la viabilidad práctica del método propuesto.

Anexo B

Gestión del Proyecto

El proyecto se ha desarrollado a lo largo de 2025, desde principios de marzo hasta finales de noviembre. Como se ha mencionado anteriormente, se organiza en varias fases: una primera dedicada a Estudio y Formación, seguida de la implementación en *render* estacionario, luego la extensión al *render* transitorio, y finalmente la redacción de la memoria junto con la colaboración en el *paper*.

En la Tabla B.1 se muestran las horas dedicadas a cada tarea, mientras que el diagrama de Gantt en la Figura B.1 permite visualizar la distribución temporal de las distintas fases del proyecto.

Tarea	Tiempo dedicado (h)
Estudio y Formación	50:30
Reuniones	27:15
Estacionario	92:45
Transitorio	134:00
Memoria	60:30
Paper	30:00
Total	395:00

Cuadro B.1: Horas dedicadas al proyecto. En esta tabla se pueden ver las horas dedicadas a cada tarea del proyecto.

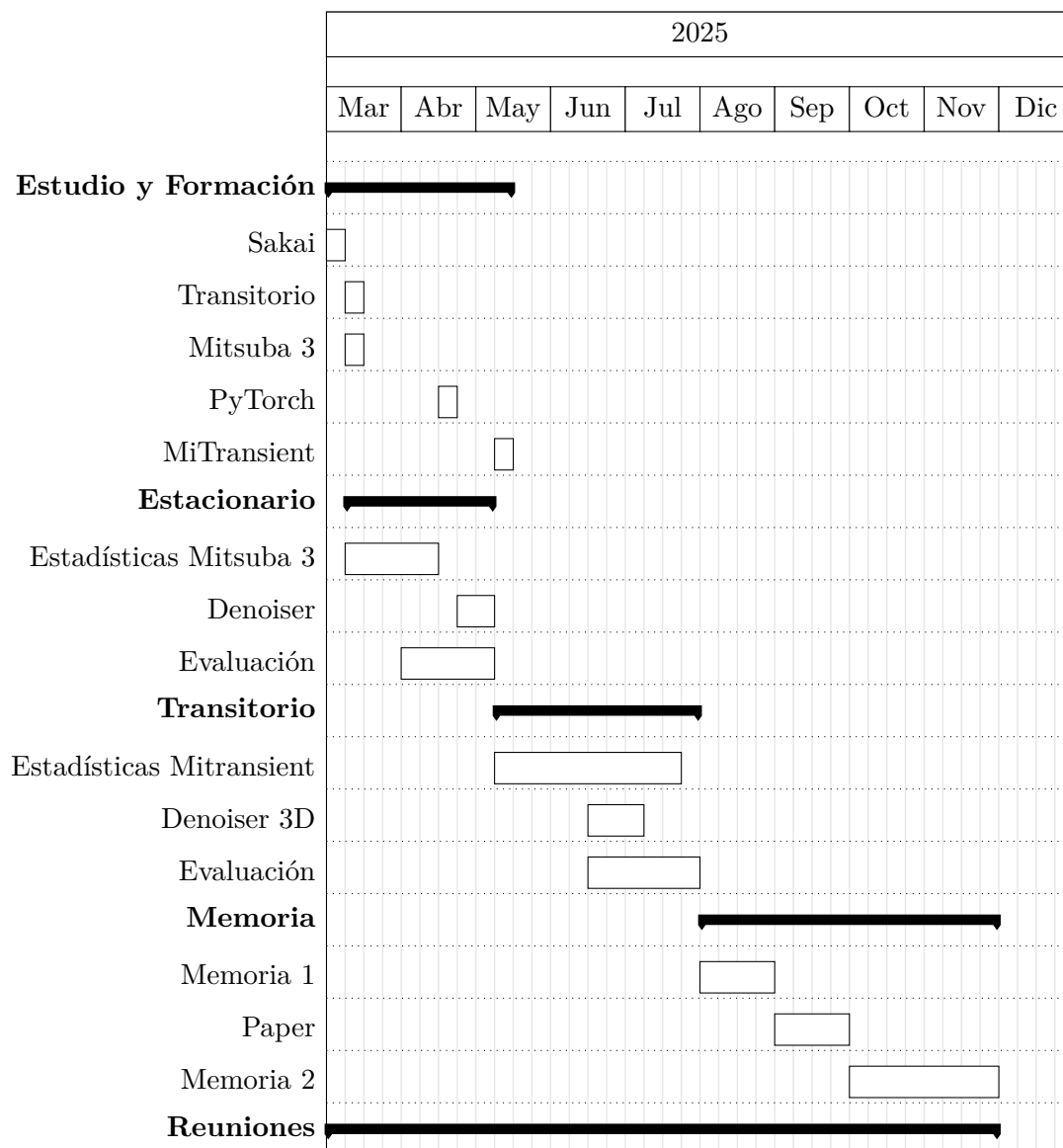


Figura B.1: Diagrama de Gantt del proyecto.