Universidad Internacional de la Rioja (UNIR)

Escuela Superior de Ingeniería y Tecnología

Máster Ingeniería Matemática y Computación

Métodos de Montecarlo para corregir autointersecciones en ray tracing

Trabajo Fin de Máster

Presentado por: Enrique Cardero Riveiro

Dirigido por: Raudys Rafael Capdevila Brown

Ciudad: Ourense

Fecha: 22 de julio de 2021

Índice de Contenidos

Índice de Ilustraciones

Índice de Tablas

Resumen

El propósito principal de este trabajo es introducir una nueva aproximación para resolver el problema de las autointersecciones en ray tracing mediante métodos de Montecarlo. Los métodos más avanzados para tratar estos problemas, provocados por la falta de precisión al calcular puntos de intersección, son eficientes pero dependen de la escena, lo que puede causar igualmente artefactos en la imagen final renderizada.

Mediante la introducción de nuevos algoritmos basados en la generación de números aleatorios, podemos desarrollar nuevos métodos capaces de adaptarse a la escena por sí mismos, obteniendo como resultado una mayor flexibilidad a la hora de ajustar los parámetros e imágenes con un mejor aspecto.

Introducimos dos algoritmos que implementan este nuevo enfoque del problema de la autointersección: uno de ellos basado en la generación de números uniformemente distribuidos, y otro basado en la generación de valores de una distribución normal. Los resultados muestran que, con pocas iteraciones de cualquiera de estos métodos de Montecarlo, podemos renderizar una imagen con menos puntos de autointersección de los que conseguiríamos utilizando la técnica más popular hoy en día para este propósito.

Palabras Clave: ray tracing, autointersección, Montecarlo, renderizado, Python.

Abstract

This work's main purpose is to introduce a new Monte Carlo approach for solving the self-intersection problem in ray tracing. State-of-the-art methods dealing with these issues, caused by the lack of precision when computing intersection points, are efficient but scene-dependent, which can still lead to some artifacts on the resulting rendered image.

By introducing new algorithms based on the generation of random numbers, we can develop new methods capable of adapting to the scene by themselves, which can result in more flexibility when tweaking parameters and better-looking frames.

We introduce two algorithms implementing this new approach to the self-intersection problem: one of them based on the generation of uniformly distributed numbers, and the other based on generating values of a normal distribution. Results show that, with a few iterations of any of these Monte Carlo methods, we can render a picture with less self-intersection points than we would get by using today's most popular technique for this purpose.

Keywords: ray tracing, self-intersection, Monte Carlo, rendering, Python.

Capítulo 1

Introducción

La **computación gráfica** es una disciplina científica dedicada a la generación, manipulación y almacenamiento de datos gráficos digitales, incluyendo imágenes estáticas (tanto en 2D como en 3D), gráficos animados e imágenes interactivas (Salomon, 2011). A la hora de generar estas imágenes por ordenador, se han puesto en práctica multitud de técnicas y algoritmos, cada uno con sus ventajas e inconvenientes.

Un término fundamental dentro del campo de los gráficos por ordenador es el **renderizado** (en inglés, *rendering*), que hace referencia al proceso por el cual se representa sobre una salida bidimensional (por ejemplo, una pantalla) una escena formada por objetos sólidos tridimensionales (Salomon, 2011). Según los objetivos del renderizado, podemos distinguir entre dos grandes categorías:

- Se habla de **prerrenderizado** (*offline rendering*) cuando el proceso de generación de la imagen puede ser muy largo (del orden de varias horas) debido al uso de algoritmos que priorizan la calidad de la imagen final frente al tiempo de renderizado. Estas técnicas se suelen usar para imágenes estáticas o películas, donde se busca conseguir el máximo grado de realismo.

- Por otro lado, el **renderizado en tiempo real** (*online rendering*) prioriza el rendimiento, de manera que las imágenes (llamadas *fotogramas* en este contexto) se generen en pocos milisegundos, a una frecuencia suficiente como para transmitir sensación de movimiento y fluidez. Estas técnicas son la base de los videojuegos, donde cada fotograma se debe crear en respuesta casi instantánea a las acciones del jugador.

Una de las múltiples técnicas de renderizado que existen es el **ray tracing**. El origen

de la idea de este algoritmo, según Hofmann (1990), se atribuye al matemático y pintor renacentista Albrecht Dürer, en el siglo XVI. En uno de sus libros, muestra la imagen de un artista pintando el retrato de una persona. Para hacerlo de forma precisa, sitúa una rejilla justo en el plano de proyección, y la utiliza como referencia para copiar lo que ve en otra rejilla idéntica situada en su lienzo. Así, la primera rejilla sería el equivalente a una pantalla dividida en píxeles, y la idea de trazar una línea de visión a través de cada píxel es el fundamento del algoritmo.

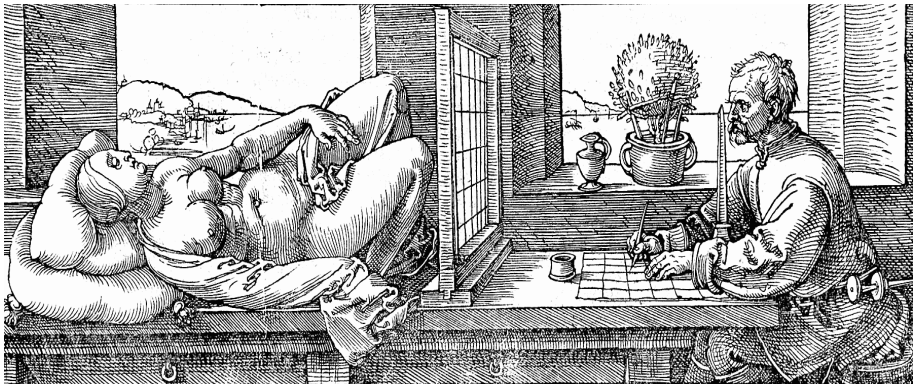

Figura 1.1: Ilustración de la idea del ray tracing. Fuente: *Underweysung der Messung* (Albrecht Dürer).

Según Freniere y Tourtellott (1997), la primera referencia documentada sobre el uso de ray tracing para la síntesis de imágenes se atribuye a Arthur Appel en su artículo *Some techniques for shading machine renderings of solids*, publicado en 1968. En él, el autor trata de determinar la visibilidad de los puntos de la escena trazando rectas entre el punto, la fuente de luz y el observador, lo que podremos identificar con el concepto de **rayo**, que introduciremos en una sección posterior de este trabajo. En dicho artículo se destaca el alto coste computacional de estos cálculos, que el autor describe como la búsqueda de una correspondencia punto a punto entre la escena y el plano sobre el que se proyecta.

Este coste computacional tan elevado pone de manifiesto una necesidad de aumentar la eficiencia del algoritmo de ray tracing, sin renunciar por ello a las imágenes fotorrealistas que este es capaz de generar. Un recurso muy frecuente para tratar problemas con cálculos muy complejos consiste en el uso de **métodos de Montecarlo**, que se basan en la generación de números aleatorios para hallar soluciones aproximadas de un cierto problema. Algunas de las aplicaciones de estos métodos sobre el algoritmo de ray tracing

tienen que ver con el cálculo de integrales, técnicas de muestreo o toma de decisiones (por ejemplo, la dirección en la que se genera un nuevo rayo).

La aplicación de estos métodos ha demostrado su eficacia para tratar algunos de los problemas ligados al uso de este algoritmo. Sin embargo, tras una consulta de los estudios existentes en relación con el problema que se tratará en este trabajo, todavía no se han aplicado métodos de Montecarlo en este campo. En particular, se explicará el problema de la **autointersección**, vinculado a los errores de precisión cometidos durante la ejecución del algoritmo de ray tracing, y cuya consecuencia es la aparición de imperfecciones en la imagen final.

1.1. Justificación

El algoritmo de ray tracing es un método de renderizado en auge, especialmente en los últimos años, con la introducción de técnicas que permiten su ejecución incluso en tiempo real. Por ello, se espera que tenga una presencia cada vez mayor, acompañada de avances en cuanto a la eficiencia y la calidad del algoritmo.

Uno de estos problemas que pueden afectar negativamente a la calidad final es la presencia de autointersecciones, como consecuencia de la precisión limitada en los cálculos. Varios autores han tratado de buscar vías alternativas para solventar estas limitaciones y, aunque algunas funcionan muy bien en la práctica, todavía pueden ser problemáticas en ciertas situaciones, por lo que existe un cierto margen de mejora en este campo. Wächter y Binder (2019), además de introducir una posible nueva solución a este problema, presentan también un ilustrativo estado del arte que recopila algunas de las técnicas ya estudiadas por otros investigadores, y que ha sido la principal fuente de inspiración de este trabajo.

El propósito de esta memoria es dar un pequeño paso en estas líneas de investigación, tratando el problema de las autointersecciones desde el punto de vista de los métodos de Montecarlo, y abriendo una posible línea de investigación futura para tratar de mejorar la calidad final de las imágenes.

1.2. Planteamiento del problema y objetivos del trabajo

El problema central que trataremos en este trabajo es el de la autointersección. Brevemente, podemos resumirlo como el error cometido al calcular intersecciones entre rectas

y objetos sólidos. Hasta ahora, se han propuesto métodos deterministas para tratar de corregir los cálculos. Frente a esto, se propone en este trabajo el uso de métodos estocásticos.

En particular, introducimos este nuevo enfoque mostrando dos métodos de Montecarlo para afrontar el problema: el primero de ellos basado en la generación de números aleatorios uniformes, y el segundo utilizando valores de una distribución normal. Se explicarán formalmente ambos métodos, se hará una implementación de los mismos en un ray tracer escrito en Python, y se pondrán a prueba renderizando dos escenas y comparando los resultados con los que se obtendrían con el método usual para corregir autointersecciones.

1.3. Estructura de la memoria

Esta memoria se dividirá en dos bloques principales.

El primero de ellos abarca todo el Capítulo 2. En él, se dedicará la Sección 2.1 a explicar fundamentos teóricos relacionados con el ray tracing y el problema de la autointersección que queremos tratar, mientras que la Sección 2.2 recopilará las soluciones utilizadas habitualmente para tratar este mismo problema.

El Capítulo 3 separa los dos bloques principales, y el él se desglosan los objetivos generales y específicos del trabajo, antes de pasar al desarrollo central.

El segundo bloque de esta memoria abarca el Capítulo 4 y el Capítulo 5. En el primero de ellos se desarrolla paso a paso la contribución, presentando la nueva solución propuesta para tratar el problema de las autointersecciones y estudiando su rendimiento a la hora de renderizar dos escenas de ejemplo. Finalmente, en el segundo capítulo de este bloque se recogen las conclusiones extraídas de los resultados del capítulo anterior, explicando también aquellos puntos cuyo estudio queda abierto para futuras investigaciones.

Capítulo 2

Contexto y Estado del Arte

2.1. Fundamentos teóricos

2.1.1. El algoritmo de ray tracing

El **trazado de rayos** (en inglés, *ray tracing*) es un algoritmo que permite generar imágenes con un alto nivel de fotorrealismo simulando el recorrido de los rayos de luz en una escena. Como su nombre indica, el elemento principal en este método será el **rayo**, que se define del siguiente modo:

$$\vec{q}(t) = o + t\vec{d} \tag{2.1}$$

donde o es el origen, $\vec{d}$ un vector unitario en la dirección del rayo, y t el parámetro que nos da la distancia a lo largo del rayo (Akenine-Möller et al., 2018). La interpretación geométrica de t como la distancia se debe a que hemos definido $\vec{d}$ tal que $||\vec{d}|| = 1$. Es fácil de ver que un rayo no es más que un vector situado en un punto del espacio tridimensional que llamamos origen.

Otro elemento importante para poder explicar el algoritmo es la **escena**. Se trata de una estructura de datos que describe geométricamente todos los objetos que se encuentran dentro de la imagen, incluyendo también una cámara que representa al observador (Moncada et al., p. 1). Para cada uno de los objetos en la escena se describen también sus correspondientes propiedades (refracción, reflexión, etc.), que nos permitirán estudiar la trayectoria de los rayos de luz que inciden sobre ellos.

El propósito de los diferentes métodos de renderizado es generar una **imagen** bidimensional, correspondiente a la visión que tendría un espectador de la escena a través de una

pantalla dividida en **píxeles**. El color final de cada píxel queda determinado por la luz que pasa a través de él, ya sea directamente o reflejada por objetos de la escena.

Una primera e intuitiva aproximación a este problema consiste en trazar los rayos con origen en las fuentes de luz, calcular los rebotes de los rayos contra los diferentes objetos de la escena, y colorear cada píxel en función de los rayos que lleguen a él. Sin embargo, el ray tracing lo aborda desde una perspectiva distinta: la mayoría de los rayos que podemos generar partiendo de las fuentes de luz no llegarán al observador; por lo tanto, podemos prescindir de sus cálculos, ya que no aportan información útil. En el algoritmo de ray tracing:

- Se genera un **rayo primario** con origen en el observador y dirección apuntando hacia un píxel de la pantalla.
- Se calcula el punto de intersección del rayo primario con el objeto (opaco) de la escena más cercano al observador.
- Se genera un **rayo de sombra** con origen en el punto de intersección calculado y en dirección a una fuente de luz.

  Si el rayo de sombra corta a un objeto que está más cerca que la fuente de luz, significa que el punto de intersección sobre el objeto no recibe luz de dicha fuente.

  Si el rayo de sombra no encuentra obstáculos en el camino a la fuente de luz, entonces el punto de intersección está iluminado por dicha fuente.

De esta forma, se computan únicamente aquellos rayos cuya trayectoria afecta directamente a la visión del observador, evitando cálculos innecesarios.

La Figura 2.1 muestra un esquema con los elementos descritos hasta ahora, y donde se muestra el funcionamiento del algoritmo:

2.1.2. El problema de la autointersección

Como se ha visto en la sección anterior, el algoritmo de ray tracing se basa en ideas intuitivas y es sencillo de entender e implementar. Sin embargo, en la práctica surgen varios problemas que es necesario tratar. Aunque el primero del que se suele hablar es el alto coste computacional del método, este trabajo se centrará en el problema de la autointersección, que describiremos en esta sección.

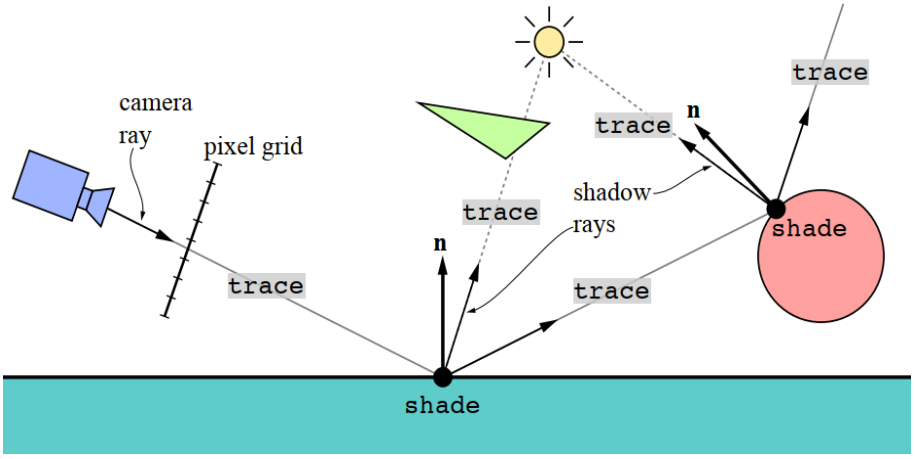

Figura 2.1: Representación de los elementos fundamentales que nos permiten describir el algoritmo de ray tracing. Fuente: Akenine-Möller et al., 2018.

Buena parte del coste computacional se debe al cálculo de las intersecciones de los rayos con los objetos de la escena. Además, dichos cálculos no están exentos de errores, que pueden provocar la aparición de **artefactos** (del inglés *artifacts*) en la imagen final. Esto es lo que conocemos como el error de truncamiento, y se debe a que un ordenador solo puede trabajar con un conjunto finito de números.

Como consecuencia de este hecho, el punto de intersección que calcula el algoritmo de ray tracing no tiene por qué coincidir exactamente con el punto de intersección real. Sean, respectivamente, $\hat{X}$ y X los dos puntos mencionados, y supongamos que $\hat{X}$ está situado dentro de un objeto de la escena. Si generamos ahora un rayo de sombra con origen en $\hat{X}$ y en dirección a una fuente de luz que ilumina X, se ve inmediatamente que el rayo de sombra corta de nuevo al objeto en cuestión, de manera que el algoritmo interpreta erróneamente que hay un obstáculo entre el objeto y la fuente de luz. La consecuencia de este hecho es que el punto X, que en realidad está iluminado, se representará erróneamente como un punto de sombra en la imagen final (lo que se conoce en inglés como *surface acne*).

Por su naturaleza, este problema recibe el nombre de **autointersección**, y será el que trataremos fundamentalmente en este trabajo.

Dado que la causa es la precisión finita de los cálculos, conviene identificar en qué situaciones puede aparecer este problema. El estándar más extendido hoy en día es IEEE 754 (aritmética en punto flotante), donde la distancia entre números consecutivos representados por ordenador aumenta con la escala. Esto significa que los errores de cálculo serán más frecuentes cuando trabajemos con números muy grandes; en nuestro caso, en-

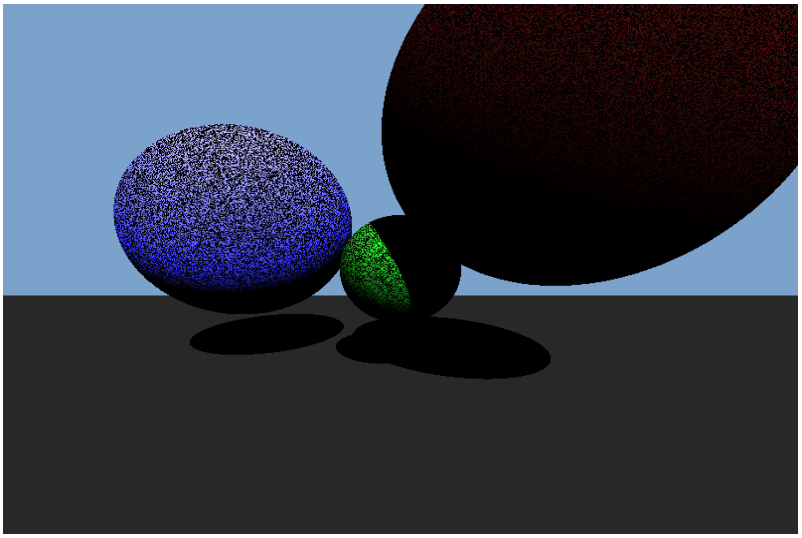

Figura 2.2: Ejemplo de *surface acne* en la superficie iluminada de las esferas. Fuente: Heisler, B. (2017).

contraremos este problema al trabajar con las distancias del origen a ciertos puntos de intersección, y el tamaño de algunos objetos.

2.1.3. Representación de números en un ordenador

Como las autointersecciones son consecuencia de la precisión finita de los ordenadores, conviene entender cómo se representan internamente los números en un procesador, ya que encontramos aquí la causa del problema que queremos afrontar.

Existen diferentes formas en que los ordenadores trabajan con números, pero las más habituales son las aritméticas de punto fijo y de punto flotante, así que nos centraremos en estos dos tipos.

Es sabido que los ordenadores trabajan con una representación binaria de los números. Supongamos un número representado por 8 dígitos binarios (*bits*):

$$b_4 b_3 b_2 b_1 b_0 b_{-1} b_{-2} b_{-3}$$

Como $b_i = \{0, 1\}, i \in \{-3, -2, ..., 4\}$, es inmediato que el total de combinaciones posibles es 2^8. En general, podemos decir que con N dígitos binarios somos capaces de representar un subconjunto de $\mathbb{Z}$ de tamaño 2^N. Obsérvese que estamos trabajando en todo momento en un contexto de números enteros.

Para extender estas representaciones a un subconjunto de $\mathbb{Q}$ (es decir, representar

números con decimales), basta con determinar la posición del punto decimal. Es aquí donde entran en juego las dos representaciones que vamos a explicar.

La **aritmética en punto fijo**, como su nombre indica, determina una posición estática para el punto decimal, de modo que todos los números representados en el sistema tienen siempre el mismo número de dígitos a cada lado del punto. Esto se puede conseguir aplicando un cierto factor de escalado a un número entero (Heinly et al., 2009).

Una de las ventajas de esta representación numérica es que todos los números se encuentran igualmente distribuidos. Así, la precisión que se tiene cerca del origen se tiene también en cualquier lugar del espacio, esto es, la precisión es un invariante por traslaciones si trabajamos con aritmética de punto fijo. Además, los cambios de escala se pueden hacer con un simple desplazamiento de bits, logrando una mayor eficiencia en este tipo de cálculos. Por otro lado, el uso de este sistema de representación exige un compromiso entre el rango de números representables y la precisión (Heinly et al., 2009). En efecto: si queremos ganar decimales de precisión, necesariamente tenemos que reducir el número de bits de la parte entera, por lo que la amplitud del intervalo de números representables será menor.

Como alternativa surge la **aritmética en punto flotante**, donde el punto decimal ya no ocupa una posición fija. La representación de un número real en este formato viene dada por una terna (signo, exponente, parte significativa); dado $r \in \mathbb{R}$ y una base b (en binario, $b = 2$), entonces (Institute of Electrical and Electronics Engineers, 2008):

$$r = (-1)^{signo} \times b^{exponente} \times parte\ significativa$$

Además de estos números, también hay representaciones para $\pm\infty$, por lo que la estructura subyacente a esta representación numérica es en realidad $\mathbb{R} \cup \{\pm\infty\}$ (Institute of Electrical and Electronics Engineers, 2008).

La aritmética en punto flotante sigue unas normas establecidas por el Institute of Electrical and Electronics Engineers (IEEE). Actualmente, el estándar utilizado es IEEE 754, según el cual un número en punto flotante codificado en 32 bits utiliza un bit para el signo (s), 8 para el exponente (e) y 23 para la parte significativa (m). Con esta estructura, es fácil ver que los exponentes están entre 0 ($= 2^0 - 1$) y 255 ($= 2^8 - 1$), todos no negativos. Por lo tanto, para poder representar exponentes negativos, calculamos el exponente verdadero e_b a partir de e como (Pharr et al., 2017):

$$e_b = e - 127$$

Debido a esta representación numérica, se tiene que el espacio entre números en punto flotante sucesivos en el intervalo $[2^e, 2^{e+1})$ toma el valor 2^{e-23} (Pharr et al., 2017). De aquí se sigue que la distribución de los números en punto flotante es uniforme en un intervalo de potencias de 2 consecutivas, pero no lo es en todo el rango de representación; de hecho, a medida que aumenta el exponente e, aumenta también la distancia entre números consecutivos.

Esta es la causa del problema de la autointersección: si la distancia entre números en punto flotante consecutivos es muy grande, entonces existe un amplio rango de números reales no representables en punto flotante, lo que implica un inevitable error de truncamiento al efectuar cálculos con dichos números. En nuestro caso, los cálculos problemáticos aparecen al operar con grandes distancias para calcular intersecciones, de manera que estos errores de cálculo se traducen en puntos indebidamente sombreados en la imagen final.

2.1.4. Algunos ejemplos de cálculo de intersecciones

En adelanto de las implementaciones en Python que mostraremos en el Capítulo 4 del trabajo, dedicaremos esta sección a explicar los métodos que usaremos para el cálculo de intersecciones. En particular, desarrollaremos un código que nos permita representar esferas, planos, triángulos y rectángulos dando sus correspondientes atributos.

Intersección rayo-esfera

Se trata de uno de los casos más intuitivos de cálculo de intersecciones. Tomaremos como referencia las explicaciones dadas por Rueckert (2002).

Como sabemos, una esfera queda determinada dados su centro (cuyo vector de posición denotaremos $\vec{p_s}$) y su radio r. Entonces, un punto q se encuentra en la esfera si verifica la ecuación:

$$|\vec{q} - \vec{p_s}|^2 - r^2 = 0 \tag{2.2}$$

donde $\vec{q}$ denota el vector de posición del punto q.

Supongamos un rayo de ecuación $p_0 + \mu\vec{d}$. Calcular los puntos de intersección del rayo con la esfera es lo mismo que decir que buscamos aquellos puntos del rayo que están en la

esfera y, por lo tanto, verifican la ecuación anterior. Si sustituimos la expresión del rayo en la ecuación de la esfera, obtenemos:

$$|\vec{p_0} + \mu\vec{d} - \vec{p_s}|^2 - r^2 = 0 \tag{2.3}$$

donde $\vec{p_0}$ es el vector de posición del origen del rayo.

Si resolvemos la ecuación anterior para μ, pueden darse los siguientes casos:

- Si no tiene solución, el rayo no corta a la esfera.
- Si tiene dos soluciones μ_1 y μ_2 (supongamos que $\mu_1 < \mu_2$), entonces μ_1 nos da el punto de entrada del rayo en la esfera; por lo tanto, es el valor que nos interesa tomar para calcular el punto de intersección.
- Si obtenemos soluciones negativas, significa que los puntos de intersección están por detrás del origen del rayo. Si se trata de un rayo primario, entendemos que la esfera está por detrás del observador y, por lo tanto, no nos interesa ese punto de intersección.

Intersección rayo-plano

El segundo objeto cuya intersección queremos describir es el plano. Es importante, ya que estos cálculos se pueden utilizar para la representación de cualquier figura plana, como explicaremos más adelante. Nuevamente, tomaremos como referencia las explicaciones de Rueckert (2002).

La idea fundamental para este cálculo es que todo vector $\vec{q}$ contenido en el plano es, por definición, ortogonal al vector normal $\vec{n}$ y, por lo tanto, se cumple que $\vec{q} \cdot \vec{n} = 0$. Igual que antes, si sustituimos la expresión de un rayo en la ecuación del plano obtenemos lo siguiente:

$$((\vec{p_0} + \mu\vec{d}) - \vec{p_1}) \cdot \vec{n} = 0 \tag{2.4}$$

donde p_1 es un punto arbitrario del plano. Si resolvemos la ecuación anterior para μ, obtendremos el punto del rayo que está en el plano; por lo tanto, es inmediato que el vector $(\vec{p_0} + \mu\vec{d}) - \vec{p_1}$ está en el plano y la expresión anterior tiene sentido.

Intersección con un polígono plano convexo

El cálculo de la intersección de un rayo y un plano se puede extender a la intersección del rayo con cualquier polígono plano convexo. Esto se puede efectuar en dos pasos: primero se calcula el punto de intersección con el plano que contiene al polígono, y después se comprueba si el punto de intersección está dentro del polígono.

Para hacer esta última comprobación es muy importante el orden en que definimos los vértices del polígono, ya que esto afectará a la orientación de los productos vectoriales (incluyendo el cálculo del vector normal como producto vectorial de dos lados consecutivos). Por defecto, definiremos siempre los vértices en sentido antihorario.

Introducimos ahora el método explicado en Scratchapixel (s. f.). Supongamos que q es el punto de intersección del rayo con el plano que contiene al polígono. Sean v_0 y v_1 dos vértices consecutivos del polígono definidos en sentido antihorario, como hemos indicado anteriormente; por lo tanto, el vector $v_0\vec{v}_1$ se corresponde con un lado. El test consiste en calcular el producto vectorial $v_0\vec{v}_1 \times v_0\vec{q}$, de donde se pueden sacar las siguientes conclusiones:

- Si el producto vectorial nos da un vector con el mismo sentido que el normal del plano, entonces el punto está dentro del polígono. Esto se puede determinar con el signo del producto interior $(v_0\vec{v}_1 \times v_0\vec{q}) \cdot \vec{n}$.
- En caso contrario, el punto estará fuera del polígono y, por lo tanto, el rayo no lo corta.

Si interpretamos geométricamente estos resultados, lo que ocurre es que, tal y como hemos definido los vértices, el vector $v_0\vec{q}$ se obtiene girando el lado $v_0\vec{v}_1$ en sentido antihorario si, y solo si, el punto q está en el interior del polígono.

Los cálculos descritos se repiten para cada uno de los lados del polígono. Por lo tanto, este método nos servirá para la implementación tanto de los triángulos como de los rectángulos, variando únicamente el número de lados para los que hay que efectuar el test.

2.2. Soluciones habituales al problema

Existen diferentes formas de minimizar el problema de la autointersección, aunque ninguna de ellas es perfecta, y el problema puede seguir presente en ciertas escenas. Expondremos los métodos más habituales hoy en día.

2.2.1. Identificación de primitivas

El alto coste computacional del ray tracing deriva fundamentalmente del cálculo de intersecciones. Para simplificar este problema, lo habitual es representar los objetos de la escena a través de figuras geométricas más simples, que llamamos **primitivas**. Este proceso es el que se conoce como **teselado** (en inglés, *tessellation*). De esta forma, se puede reducir la búsqueda de algoritmos eficientes de cálculo de intersecciones a aquellas en las que intervienen las primitivas. El ejemplo más destacado de primitiva es el triángulo, el polígono más utilizado para el renderizado en tiempo real.

Como resultado del teselado, en la escena tenemos objetos representados como conjuntos de polígonos (triángulos), y el algoritmo de ray tracing busca la intersección de los rayos con alguno de dichos triángulos. Además, cada primitiva cuenta con una ID, y de este hecho surge el método que presentamos a continuación para prevenir autointersecciones.

La idea de este método consiste en descartar una intersección si las IDs de las dos primitivas coinciden, lo que sería un claro indicio de autointersección. Sin embargo, este método presenta dos problemas claros en la práctica (Wächter y Binder, 2019):

- Si generamos un rayo de sombra en un ángulo rasante, puede ocurrir que la distancia al mismo objeto no supere el valor mínimo de la aritmética en punto flotante (es decir, se considera que la distancia es cero); además, dado que nos hemos desplazado sobre la superficie, el nuevo punto se encontrará en otra primitiva, por lo que las IDs no coinciden y la intersección se da por válida.
- Si el origen del rayo de sombra está muy cerca del borde de una primitiva, es posible que al generarlo corte a la primitiva adyacente, con lo que la intersección se dará por válida.

La Figura 2.3 ilustra estos dos casos de error.

2.2.2. Ray tracing en aritmética de punto fijo

Entre las soluciones propuestas para evitar autointersecciones, casi siempre está presente un valor que llamamos **epsilon** (ε), que representa una distancia que desplazamos un punto de intersección calculado en una cierta dirección, con el fin de evitar que el rayo de sombra con origen en ese punto corte a la misma superficie.

Con esto en mente, algunos autores trataron de implementar software de ray tracing en sistemas que utilizan aritmética de punto fijo. Uno de los problemas más recurrentes

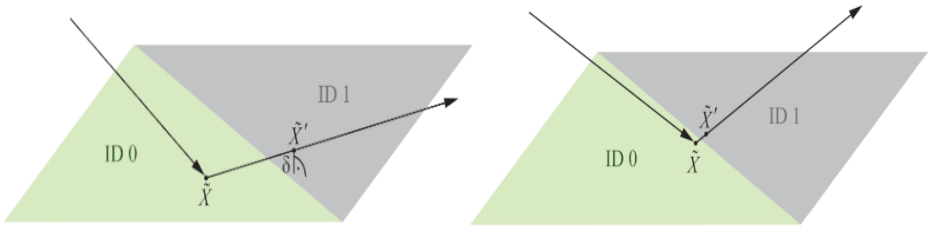

Figura 2.3: Problemas con la exclusión por ID de primitivas. Fuente: Wächter y Binder, 2019.

en el contexto de las autointersecciones es la elección del mejor valor de épsilon, que habitualmente depende en gran medida de la escena con la que se esté trabajando. En efecto, los objetos más distantes están sujetos a mayores errores de cálculo, por lo que el valor de épsilon ha de ser mayor; por contrapartida, un valor demasiado grande provocará incoherencias en objetos cercanos, como sombras separadas del objeto que las produce.

En la Figura 2.4 se muestra un ejemplo de este hecho. En la imagen de la izquierda, un valor de $\varepsilon = 10^{-5}$ es suficiente para producir una imagen correcta; por el contrario, en la imagen de la derecha se usa un valor de $\varepsilon = 10^{-1}$, que es demasiado grande para la escena en cuestión, provocando desplazamientos en sombras y reflejos. El desplazamiento de las sombras se ve claramente en la sombra de la esfera roja, que se sale del marco de la imagen. Además, en la esfera verde se aprecia que, lo que antes era el hemisferio del lado opuesto a la fuente de luz, ahora tiene una tonalidad incorrecta, debida al desplazamiento de los reflejos. Observando atentamente, se puede ver este mismo defecto visual en las otras dos esferas.

Como hemos indicado anteriormente, la causa subyacente a este hecho es la distribución de los números en punto flotante, ya que la distancia entre números consecutivos aumenta a medida que lo hace su valor absoluto. Frente a esto, la aritmética en punto fijo se puede entender como una malla de nodos equidistantes, por lo que la resolución de una distancia medida sobre un rayo es la misma que la de los vértices de los objetos, evitando así la dependencia de la escena para fijar ε. Con estos razonamientos, Hanika y Keller (2007)

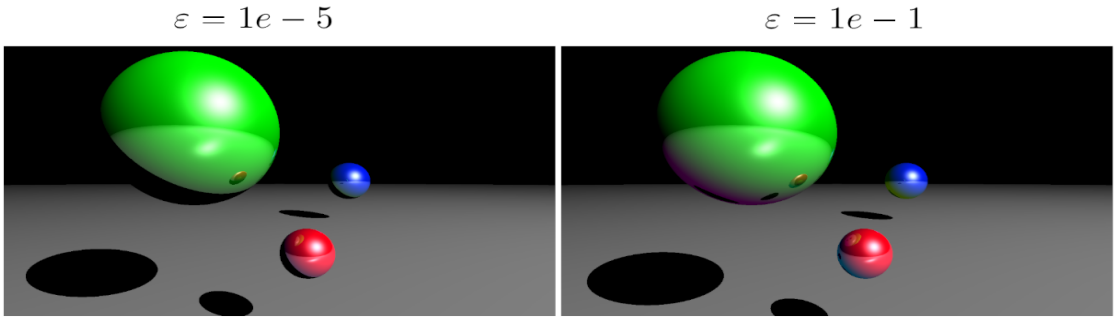

Figura 2.4: Efectos de distintos valores de ε en una misma escena. Fuente: elaboración propia.

proponen una implementación del algoritmo de ray tracing para sistemas que trabajan con aritmética entera, y que permite tomar en el peor de los casos un valor de $\varepsilon = 2$.

Posteriormente, Heinly et al. (2009) desarrollaron una versión adaptada del algoritmo de ray tracing para sistemas que no trabajan con el estándar IEEE 754 para aritmética en punto flotante, que es el más extendido hoy en día. En su lugar, existen todavía procesadores más especializados que no utilizan internamente esta representación numérica, por lo que se trata de una alternativa a tener en cuenta para trasladar el ray tracing a estos sistemas. Nuevamente, en este método el valor de ε no depende de la escena, aunque frente al valor de $\varepsilon = 2$ dado por Hanika y Keller, los ejemplos de Heinly et al. exigieron tomar al menos $\varepsilon = 6$.

2.2.3. Cálculo de intersecciones en el espacio de objetos

Se trata de una alternativa propuesta por Dammertz y Keller (2006) para corregir los errores de precisión en el cálculo de intersecciones. Por ello, trata de corregir también las autointersecciones, aunque no se limita a resolver este problema.

El enfoque tradicional consiste en calcular a qué distancia se encuentra el punto de intersección del origen del rayo. Cuando esta distancia es muy grande, la aritmética en punto flotante de los ordenadores no es capaz de calcularla exactamente, lo que se traduce en artefactos en la imagen final. Como alternativa, se propone calcular los puntos de intersección directamente sobre el espacio de objetos, evitando las grandes distancias que puede alcanzar un rayo.

La idea del método consiste en aproximar la posición del punto de intersección dando un

intervalo tridimensional que lo contenga. Es decir, dado el punto de intersección verdadero X, el algoritmo busca un bloque $B = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3] \subset \mathbb{R}^3$ tal que $X \in B$. Esencialmente, estamos buscando una caja alineada con los ejes que contenga a X.

El algoritmo comienza con una caja determinada por los extremos del objeto. A partir de dicha caja inicial, comienza un proceso de subdivisión que se repite hasta que no hay cambios entre iterados consecutivos. Esto ocurrirá cuando no existan números en punto flotante para efectuar la subdivisión. Para realizar este proceso se utiliza el valor de parámetro 0,5, es decir, dividimos los intervalos originales por la mitad. Esto asegura que no existan fisuras entre los bloques resultantes de una subdivisión, ya que multiplicar por 0,5 equivale a restar una unidad al exponente del número en punto flotante, evitando posibles errores de truncamiento (excepto en el caso de un *underflow* numérico).

Para evitar el problema de la autointersección, lo que se hace es tomar como punto de origen del rayo de sombra la esquina más alejada en la dirección del vector normal de la superficie, como se muestra en la Figura 2.5. De todas formas, no es un método infalible: sigue siendo necesario un tamaño mínimo del bloque para poder evitar autointersecciones cuando el rayo de sombra está muy cerca de ser tangente, especialmente en geometrías no convexas.

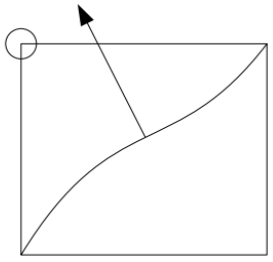

Figura 2.5: Elección del origen del rayo de sombra. Fuente: Dammertz y Keller, 2006.

2.2.4. Desplazamiento de la posición del observador

Este enfoque planteado por Kim et al. (2016) parte del hecho de que la aritmética en punto flotante es más imprecisa cuanto más lejos se está del origen. Por lo tanto, la idea de este método consiste en desplazar la cámara al origen, y aplicar la misma traslación a los objetos de la escena. De esta forma, los cálculos se harán con números más cercanos a cero, y por tanto, la precisión en punto flotante es mayor.

En la práctica, existen dos casos en los que este método puede ser problemático:

- Si la cámara ya se encuentra muy cerca del origen, la ventaja del método es prácticamente nula. Si existe autointersección en un objeto lejano, el algoritmo no es capaz de solucionarlo.

- Si un objeto está muy cerca del origen y la cámara muy lejos, al efectuar el desplazamiento, el objeto se alejará del origen, y el problema de autointersección puede persistir.

2.2.5. Cota inferior de la distancia

Las autointersecciones se producen cuando el rayo de sombra corta a la misma superficie. En tal caso, lo habitual es que la distancia del origen del rayo de sombra al punto de corte con la superficie tome un valor muy pequeño. Partiendo de esta idea, podemos sugerir una cota inferior (t_{min}) para la distancia medida sobre el rayo de sombra. De esta forma, si el punto de corte se encuentra a una distancia menor que t_{min} del origen del rayo, consideraremos que es un punto de autointersección, y lo obviamos en nuestros cálculos.

A pesar de que esta solución es sencilla y eficiente (no incrementa el coste computacional), el valor de t_{min} depende en buena parte de la escena con la que se está trabajando. Esto es especialmente problemático cuando el rayo de sombra se genera en un ángulo rasante, o cuando se salta una intersección válida por tratar de respetar la distancia mínima definida (Wächter y Binder, 2019). Estos dos casos de error se ilustran en la Figura 2.6.

La dependencia de t_{min} de la escena la describen Pharr et al. (2017) del siguiente modo:

- Cuando el rayo de sombra es casi tangente a la superficie, se necesita un valor de t_{min} grande para evitar una intersección incorrecta, producida porque la distancia a la superficie puede seguir siendo 0 en puntos muy alejados del origen del rayo de sombra.

- Un valor de t_{min} excesivamente grande puede provocar saltos de intersecciones válidas, perdiendo detalles finos en sombras y reflejos.

2.2.6. Corrección por desplazamiento

Este método consiste en desplazar ligeramente el punto de intersección calculado para que aproxime mejor a la intersección verdadera. Para ello, es necesario responder dos

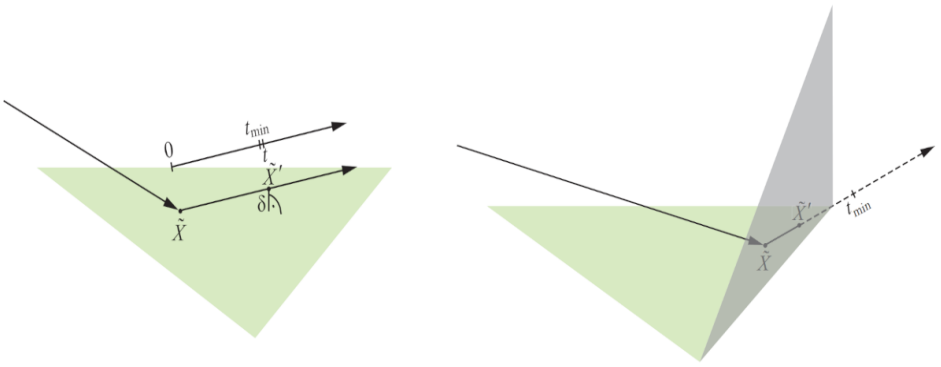

Figura 2.6: Problemas al limitar la distancia sobre el rayo de sombra. Fuente: Wächter y Binder, 2019.

cuestiones: a lo largo de qué vector lo desplazamos, y qué valor (*offset*) usamos para medir la distancia.

Dirección de desplazamiento

Como lo que queremos es evitar que un rayo de sombra corte de nuevo a la superficie, tiene sentido pensar en un vector orientado hacia el exterior del objeto. Wächter y Binder (2019) exponen los siguientes casos:

- Desplazamiento en la dirección del rayo primario. En efecto, una primera idea consiste en retroceder en la dirección del rayo que se interseca con la superficie. Aunque intuitivo, presenta los mismos problemas que otros métodos.

- Desplazamiento en la dirección del vector normal de sombreado (*shading normal*). Un modelo poliédrico para representar un objeto tridimensional solamente proporciona los vectores normales en los vértices, pero esta información se necesita en todo punto del modelo para representar las propiedades especulares del objeto; por ello, Phong (1975) propone una aproximación del vector normal en cada punto de la superficie que se calcula mediante interpolación. Este vector es el que llamamos *shading normal* y no coincide necesariamente con el normal geométrico.

- Desplazamiento en la dirección del vector normal geométrico: como este vector es ortogonal a la superficie por definición, entonces será el que nos permita tomar un *offset* menor, por lo que es la elección más apropiada. De hecho, este se trata del método más utilizado hoy en día para la corrección de autointersecciones.

Elección del *offset*

Como ya se ha visto en secciones anteriores, el valor del *offset* depende directamente de la distancia a la que se encuentre el punto de intersección del origen del rayo. Cuanto mayor sea esa distancia, mayor será el error cometido por la falta de precisión; por lo tanto, el desplazamiento requerido para corregir la posición del punto de intersección será mayor.

Indirectamente, la elección de este *offset* es uno de los principales problemas que surgen al aplicar cualquiera de las soluciones expuestas hasta ahora. Al ser un valor dependiente de la escena, es imposible hacer una elección robusta, que sea efectiva para una escena arbitraria. Frente a esta situación, Wächter y Binder (2019) trataron de dar una respuesta empírica a esta cuestión.

El experimento consiste en generar 10 millones de triángulos aleatorios, cuyos lados miden entre 2^{-16} y 2^{22}, y estudiar las diferencias que existen entre el punto de intersección calculado y el punto de intersección real en cada caso. Con estos valores obtenidos empíricamente, se han calculado las distancias medias y las distancias máximas, para posteriormente representarlas en función de la distancia al origen, como se puede ver en la Figura 2.7.

Para que el offset elegido sea suficientemente robusto, este ha de ser al menos tan grande como el error máximo cometido durante el experimento; en caso contrario, podría ocurrir que la corrección del punto de intersección calculado no sea suficiente para evitar la autointersección.

Se trata de un método que funciona muy bien en la práctica, pero también existen situaciones en las que da problemas. Al igual que ocurría al acotar inferiormente la distancia sobre el rayo de sombra, la elección de un offset grande para intersecciones muy alejadas del origen puede dar problemas con detalles geométricos muy finos, por ejemplo, pequeñas hendiduras. El punto de intersección desplazado podría situarse de nuevo detrás de la superficie; de esta forma, se produce autointersección en un nuevo punto, como muestra la Figura 2.8.

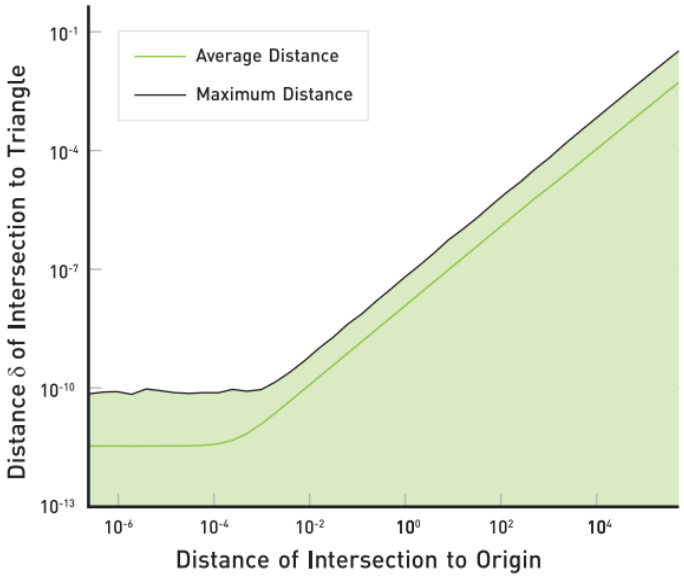

Figura 2.7: Error medio y máximo en función de la distancia al origen. Fuente: Wächter y Binder, 2019.

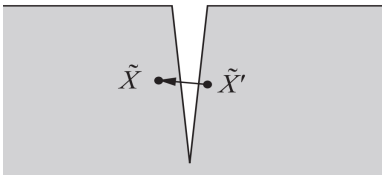

Figura 2.8: Pequeña hendidura en la que se produce una nueva autointersección. Fuente: Wächter y Binder, 2019.

Capítulo 3

Objetivos

3.1. Objetivo general

Desarrollar un método de Montecarlo que logre reducir el número de autointersecciones con respecto a los métodos utilizados en la actualidad.

3.2. Objetivos específicos

- Descripción teórica del método, basado en la generación de números aleatorios de una distribución uniforme.
- Implementación de un ray tracer sencillo en Python que permita aplicar el método usual de desplazamiento en la dirección del vector normal mediante una distancia fija.
- Modificación del ray tracer anterior para añadir la posibilidad de aplicar el nuevo método de Montecarlo como variante.
- Comparativa visual entre el método original y la variante de Montecarlo para renderizar dos escenas.
- Estudio del tiempo de renderizado del método de Montecarlo en función del número de iteraciones.
- Desarrollo de una variante del método de Montecarlo que sustituya el generador de valores uniformes por un generador de valores aleatorios según una distribución normal.

- Implementación de la nueva variante del método de Montecarlo en el ray tracer.
- Comparativa visual entre el método original y esta nueva variante del método de Montecarlo.
- Estudio del tiempo de renderizado del nuevo método de Montecarlo en función del número de iteraciones.

Capítulo 4

Desarrollo del trabajo

4.1. Idea del método a desarrollar: caso uniforme

El propósito de este trabajo es desarrollar un método de Montecarlo que sea capaz de afrontar el problema de la autointersección. Actualmente, el método más utilizado en la práctica de entre todos los presentados en la Sección 2.2 es el desplazamiento en la dirección del vector normal geométrico de la superficie. Por lo tanto, tomaremos como referencia un ray tracer que implemente dicho método, para desarrollar a partir de este la variante de Montecarlo.

Se ha hablado de que uno de los pasos más importantes es la elección del valor de épsilon (ε), ya que este depende de la escena que se quiere renderizar. Cuanto mayor sea la distancia del observador al punto de intersección, mayor será el error cometido, y se requerirá un valor de ε más grande; a su vez, un ε excesivamente grande provocará efectos no deseados en la imagen final.

Nuestro nuevo método permitirá una mayor versatilidad en cuanto a la elección de ε, y al mismo tiempo mejorar los resultados de la imagen final. Para ello, tomaremos un ε que corrija la mayor cantidad posible de autointersecciones (por ejemplo, los máximos obtenidos empíricamente por Wächter y Binder, Figura 2.7). Posteriormente, para evitar una corrección excesiva del punto de autointersección, o problemas como el descrito en la Figura 2.8, lo que haremos será generar valores aleatorios de una distribución uniforme continua en el intervalo $[0, 1]$, que jugarán el papel de factor de escalado de ε. De esta forma, mediante la generación de estos valores aleatorios (Montecarlo) podremos hacer una elección de ε más adecuada a la escena, sin necesidad de buscar un valor apropiado de forma tan exhaustiva.

Los pasos a seguir en este nuevo método son los siguientes:

- Se fija un valor inicial de ε, que denotamos ε_0, que no sea excesivamente grande. Se corresponde con el valor que utilizaríamos con el método habitual de desplazamiento por el vector normal.
- Si llamamos q_0 al punto de intersección calculado inicialmente, definimos $q = q_0 + \varepsilon_0 \vec{n}$, donde $\vec{n}$ es el vector normal de la superficie.
- Se genera de un rayo de sombra con origen en q y se comprueba si existe algún objeto de la escena en su trayectoria hacia la fuente de luz.

  Si no hay ningún objeto en el camino, no es necesario aplicar el método de Montecarlo y el proceso de renderizado continúa con normalidad.

  En caso contrario, supondremos que se trata de un problema de autointersección, y efectuamos el test de Montecarlo para comprobarlo.
- Se fija un valor ε_1 tal que $\varepsilon_1 \geqslant \varepsilon_0$. A continuación, generamos n valores aleatorios de una distribución uniforme continua en $[0, 1]$, que denotamos $r_1, r_2, ..., r_n$. Supongamos que están en orden creciente ($r_1 < r_2 < ... < r_n$).
- A partir de los números aleatorios generados, definimos las distancias $\Delta_k = r_k \varepsilon_1,\ k = 1, ..., n$.
- A partir de las distancias anteriores, definimos los puntos $q_k = q_0 + \Delta_k \vec{n},\ k = 1, ..., n$.
- Comenzando por $k = 1$ (bucle), generamos un rayo de sombra con origen en q_k.

  Si el rayo de sombra encuentra un objeto de la escena en su trayectoria hacia la fuente de luz, pasamos a la siguiente iteración, en la que generaremos un rayo de sombra con origen en q_{k+1}.

  En caso contrario, consideramos que q_k es nuestra mejor aproximación al punto de intersección real.

4.2. Implementación de un ray tracer básico orientado a objetos en Python

El primer paso de este trabajo consiste en implementar en Python un ray tracer básico que usaremos como referencia para desarrollar el nuevo algoritmo. Tomamos como base

el código desarrollado por Aflak (2020), que nos permite renderizar una escena sencilla cuyos objetos son esferas. En este caso, el código utiliza el desplazamiento en la dirección del vector normal con un *offset* fijo (10^{-5}) para evitar las autointersecciones; como hemos dicho, este es el método más usado habitualmente. Se adjunta el código completo en el Apéndice A.

Lo primero que haremos será reescribir el código para facilitar la adición de nuevos tipos de objetos a la escena y la edición de sus propiedades. Para ello, seguiremos un paradigma de programación orientada a objetos, donde cada elemento de la escena se corresponderá con una clase, que recoge aquellas características que nos permiten definirlo.

4.2.1. Clases de objetos

Surface

Para evitar sobrecargar en exceso las clases, definimos primero una superclase de la que se heredarán los atributos comunes, que son aquellos relacionados con las propiedades de los materiales para calcular la iluminación (*ambient*, *diffuse*, *specular*, *shininess* y *reflection*). También definimos dos métodos abstractos que cada clase deberá implementar: un método que devuelva el vector normal del objeto en un cierto punto (*normal*), y un método que devuelva la distancia del origen del rayo al punto en que corta al objeto, si lo corta (*intersect*). De esta forma, al llamar al método desde una instancia de una cierta clase, se ejecutará el código correspondiente a dicha clase, lo que permitirá utilizar una nomenclatura única para simplificar la escritura del código principal del ray tracer. A esta superclase la llamaremos *Surface*, y la definimos como se muestra a continuación.

```
1 import numpy as np
2 from abc import ABC, abstractmethod #Para definir la clase
     abstracta
3
4 class Surface(ABC):
5
6     def __init__(self,ambient,diffuse,specular,shininess,reflection
         ):
7         self.ambient = ambient
8         self.diffuse = diffuse
9         self.specular = specular
```

```
10          self.shininess = shininess
11          self.reflection = reflection
12
13      @abstractmethod
14      def intersect(self):
15          pass
16
17      @abstractmethod
18      def normal(self):
19          pass
```

Ahora, para definir un nuevo tipo de objeto, bastará con crear una clase que herede de *Surface*, añadir los atributos propios del objeto, e implementar los métodos *intersect* y *normal*.

Sphere

El primer objeto de la escena que implementaremos será una esfera, ya que se trata del caso más intuitivo. En primer lugar, los atributos que permiten definir geométricamente la esfera son el centro (*center*) y el radio (*radius*). Para calcular el vector normal en un punto, basta con tomar la diferencia entre los vectores de posición del punto y de la esfera. Finalmente, para implementar el cálculo del punto de intersección aprovechamos la función *sphere_intersect* definida por Aflak (2020), que se basa en la misma idea expuesta en la Subsección 2.1.4. El código correspondiente a la clase *Sphere* se muestra a continuación.

```
1  import numpy as np
2  from Surface import *
3
4  class Sphere(Surface):
5
6      def __init__(self,center,radius,ambient,diffuse,specular,
           shininess,reflection):
7          Surface.__init__(self,ambient,diffuse,specular,shininess,
               reflection)
8          self.center = center
9          self.radius = radius
10
```

```
11      def intersect(self, ray_origin, ray_direction):
12          b = 2 * np.dot(ray_direction, ray_origin - self.center)
13          c = np.linalg.norm(ray_origin - self.center) ** 2 - self.
                radius ** 2
14          delta = b ** 2 - 4 * c
15          if delta > 0:
16              t1 = (-b + np.sqrt(delta)) / 2
17              t2 = (-b - np.sqrt(delta)) / 2
18              if t1 > 0 and t2 > 0:
19                  return min(t1, t2)
20          return None
21
22      def normal(self,point):
23          return (point - self.center)/np.linalg.norm(point - self.
                center)
```

Plane

Implementaremos también una clase correspondiente a un plano. El interés de esta clase se encuentra fundamentalmente en el cálculo del punto de intersección, ya que es una operación que utilizaremos para la representación de cualquier polígono convexo, como es el caso de los triángulos y rectángulos que mostraremos a continuación.

Los atributos fundamentales que nos permiten definir un plano son un punto y el vector normal. Para implementar el cálculo de la intersección, basta con escribir la expresión resultante de despejar μ en la ecuación (2.4). A continuación se muestra el código correspondiente a la clase *Plane*.

```
1   import numpy as np
2   from Surface import *
3
4   class Plane(Surface):
5
6       def __init__(self,point,normal,ambient,diffuse,specular,
            shininess,reflection):
7           Surface.__init__(self,ambient,diffuse,specular,shininess,
                reflection)
```

```
8           self.point = point
9           self.normal_vec = normal / np.linalg.norm(normal)
10
11      def intersect(self, ray_origin, ray_direction):
12          mu = -np.dot(ray_origin - self.point,self.normal_vec) / np.
                dot(ray_direction,self.normal_vec)
13          if mu>0:
14              return mu
15          else:
16              return None
17
18      def normal(self,point):
19          return self.normal_vec
```

Quad

Explicamos ahora la implementación de la clase correspondiente a los rectángulos, que nos servirá para definir, por ejemplo, los cuadrados que constituyen cada una de las caras de un cubo. Además, veremos que la implementación de la clase *Triangle* se sigue inmediatamente eliminando uno de los vértices, sin afectar a los procesos de cálculo.

Como se ha explicado en la Subsección 2.1.4, el primer paso para calcular la intersección con un polígono plano convexo consiste en considerar únicamente el plano que lo contiene. Una vez hecho esto, se comprueba que el punto calculado está en el interior del polígono.

El test de intersección es el mismo para cada lado del polígono, cambiando únicamente la correspondiente nomenclatura para adaptarla al mayor o menor número de lados (en este caso, se efectúa la comprobación para los cuatro lados). El código resultante de la clase *Quad* se muestra a continuación.

```
1   import numpy as np
2   from Surface import *
3
4   class Quad(Surface):
5
6       def __init__(self,vertex0,vertex1,vertex2,vertex3,ambient,
            diffuse,specular,shininess,reflection):
7           Surface.__init__(self,ambient,diffuse,specular,shininess,
```

```
            reflection)
8           self.vertex0 = vertex0
9           self.vertex1 = vertex1
10          self.vertex2 = vertex2
11          self.vertex3 = vertex3
12          self.normal_vec = np.cross(vertex2-vertex1,vertex0-vertex1)
                / np.linalg.norm(np.cross(vertex2-vertex1,vertex0-
            vertex1))
13
14      def intersect(self,ray_origin,ray_direction):
15
16          mu = -np.dot(ray_origin - self.vertex0,self.normal_vec) /
            np.dot(ray_direction,self.normal_vec)
17          if mu>0:
18              p = ray_origin + mu*ray_direction #Punto de
                    interseccion
19
20              #Comprobamos que el punto esta dentro del poligono
21
22              #Lado 0
23              side0 = self.vertex1 - self.vertex0
24              vp0 = p - self.vertex0
25              N = np.cross(side0,vp0)
26              if(np.dot(N,self.normal_vec)<0):
27                  return None
28
29              #Lado 1
30              side1 = self.vertex2 - self.vertex1
31              vp1 = p - self.vertex1
32              N = np.cross(side1,vp1)
33              if(np.dot(N,self.normal_vec)<0):
34                  return None
35
36              #Lado 2
37              side2 = self.vertex3 - self.vertex2
38              vp2 = p - self.vertex2
39              N = np.cross(side2,vp2)
```

```
40              if(np.dot(N,self.normal_vec)<0):
41                  return None
42
43              #Lado 3
44              side3 = self.vertex0 - self.vertex3
45              vp3 = p - self.vertex3
46              N = np.cross(side3,vp3)
47              if(np.dot(N,self.normal_vec)<0):
48                  return None
49
50              return mu
51          else:
52              return None
53
54
55      def normal(self,point):
56          return self.normal_vec
```

Aunque en este caso sería recomendable incluir una comprobación de que los vértices dados son coplanarios, omitiremos esta prueba para acortar el código. Nuestro propósito es poder definir escenas para poner a prueba el método de Montecarlo, por lo que la introducción de datos correctos hace que esta comprobación sea prescindible.

Triangle

Los triángulos, como ya hemos explicado, son fundamentales para el proceso de teselado. La capacidad de representar triángulos nos permitirá desarrollar aproximaciones de figuras geométricas muy complejas, y de ahí su importancia.

Un triángulo queda determinado dando sus tres vértices (*vertex0*, *vertex1*, *vertex2*). A diferencia de lo que ocurre con polígonos más complejos, no es necesario hacer ninguna comprobación adicional, ya que tres puntos en el espacio son siempre coplanarios.

A partir de la clase *Quad*, la implementación de esta clase es inmediata; es suficiente con eliminar la comprobación hecha para el cuarto lado, que en este caso ya no lo tenemos, así como eliminar las referencias al cuarto vértice. Se adjunta el código de esta clase en el anexo.

Light

Aunque la fuente de luz no es un elemento tan importante en nuestro estudio, crearemos una clase independiente para uniformizar el código del programa principal. A continuación, se adjunta el código correspondiente a la clase *Light*.

```
class Light:

    def __init__(self,position,ambient,diffuse,specular):
        self.position = position
        self.ambient = ambient
        self.diffuse = diffuse
        self.specular = specular
```

4.2.2. Programa principal del ray tracer

Dedicamos esta sección a describir brevemente el funcionamiento del ray tracer, basado en el código desarrollado por Aflak (2020) al que le hemos introducido las modificaciones necesarias para trabajar con un paradigma de programación orientada a objetos.

Podemos dividir todo el código en dos secciones. En la primera se definen algunas de las funciones necesarias para los cálculos:

- *normalize*: devuelve un vector normalizado (es decir, dividido entre su norma para obtener un vector unitario con la misma dirección y sentido).
- *reflected*: recibe como argumentos de entrada un vector y un eje, y devuelve el resultado de reflejar el vector de entrada con respecto al eje especificado.
- *nearest_intersected_object*: es una de las funciones más importantes del ray tracer. Recibe como argumentos de entrada una lista con los objetos que constituyen la escena, y un rayo (su origen y su dirección). La función busca entre todos los objetos de la escena aquel al que el rayo corta primero, si lo corta. Como resultado, nos devuelve tanto el objeto intersecado como la distancia del punto de intersección al origen del rayo. Cabe destacar que, si el vector director del rayo es unitario, podemos conocer las coordenadas del punto de intersección simplemente tomando $q = O + \mu\vec{d}$, donde q es la intersección, O el origen del rayo, μ la distancia que devuelve la función y $\vec{d}$ la dirección normalizada del rayo.

Además de las funciones que acabamos de introducir, se definen también otros elementos importantes:

- *width*, *height*: parámetros de la resolución de la imagen de salida.
- *max_depth*: grado de recursión del cálculo. Si tomamos $max\_depth = 1$, solo se calculan los rayos primarios. Por defecto, fijaremos $max\_depth = 3$ porque es suficiente para generar buenos reflejos.
- *camera*: el punto en el que se encuentra el observador de la escena. Por defecto, consideraremos que el observador está situado en el punto $(0, 0, 1)$.
- *ratio*, *screen*: la pantalla a través de la cual el observador ve la escena. Se divide en una cantidad de píxeles definida mediante los parámetros width y height.
- *light*: la fuente de luz de la escena. Consideraremos que es una fuente puntual, por lo que basta con dar un punto para determinar su posición. Supondremos también que es una luz blanca.

La segunda sección del código es un bucle que determina el color de cada píxel de la imagen final. Por defecto, un píxel será de color negro ($color = (0, 0, 0)$) hasta que se vayan sumando las contribuciones de los rayos reflejados por cada objeto que pasen por el píxel en cuestión. Se implementa el modelo de iluminación de Blinn-Phong.

Hasta aquí hemos presentado los elementos fundamentales que constituyen el ray tracer desarrollado por Aflak (2020), y que tomaremos como punto de partida. A continuación, presentamos a grandes rasgos los cambios que hemos hecho sobre el código original para adaptarlo a un paradigma de programación orientada a objetos:

- En el programa original se representan únicamente esferas. Cada esfera se define mediante un diccionario de Python, donde las claves son las que identifican el atributo que se está definiendo (por ejemplo, la clave *center* almacena las coordenadas del centro de la esfera). Así, se define un vector *objects* que en cada componente contiene un diccionario, correspondiente a una esfera. En su lugar, nuestro ray tracer conservará el vector *objects*, pero ahora cada componente es una nueva instancia de la clase correspondiente al objeto que queremos representar.
- Al estar usando clases externas, tenemos que importarlas al principio del código.

- Tanto en la función *nearest_intersected_object* como en los bucles, sustituimos todas las claves de diccionarios por las referencias a los correspondientes atributos de cada objeto.

- En particular, en la función *nearest_intersected_object*, el cambio anterior va ligado al uso del polimorfismo. En vez de identificar el centro y el radio de una esfera mediante la correspondiente clave del diccionario como en el programa original, lo que tenemos ahora es una llamada al método *intersect*, que recordemos que nos devuelve el valor necesario para calcular el punto de intersección sustituyendo el valor de salida (μ) en la ecuación de un rayo. Como ahora tenemos distintos tipos de objetos, y cada uno implementa el método *intersect* (introducido en la superclase *Surface*) de una forma distinta, podemos utilizar una única llamada para todo tipo de objetos: *o.intersect*, donde *o* es cualquier objeto de la escena (equivalentemente, cualquier componente del vector *objects*).

- Mediante la librería *time* se introducen los elementos necesarios para medir el tiempo de ejecución del programa. Esto será útil para las medidas de rendimiento que estudiaremos en el Sección 4.5.

- Antes de definir el vector de objetos, definimos matrices de rotación (*giro_x*, *giro_y*, *giro_z*), vectores de traslación (*loc*) y factores de escalado (*scale*) que nos permitirán modificar algunas escenas con más facilidad. En particular, los utilizaremos para definir la escena mostrada en la Sección 4.4.2.

El código del programa principal, a excepción del contenido del vector *objects*, se muestra en el anexo. En la Sección 4.4 mostraremos ejemplos de escenas renderizadas mediante la ejecución de dicho código; junto a cada escena, se adjuntarán también las líneas de código que nos permitirán obtener las imágenes mostradas, simplemente introduciendo el código correspondiente en la definición del vector *objects*, dentro del programa principal.

4.3. Implementación del método de Montecarlo

En la Sección 4.1 se ha introducido paso a paso el método que queremos desarrollar. En esta sección, trataremos su implementación de forma más detallada, y explicaremos las ideas que se han tenido en cuenta para desarrollarlo.

En el ray tracer propuesto por Aflak (2020) se define una pantalla de resolución configurable (*width* × *height*), de manera que en cada iteración del bucle se determina el color de un píxel. Cada color se define en el modelo RGB (siglas de *red*, *green*, *blue*) mediante un vector de tres componentes, donde cada componente indica la contribución de cada color primario. La escala utilizada habitualmente para estos valores numéricos va del 0 al 255, aunque nosotros trabajaremos con una escala normalizada (de 0 a 1).

Lo que ocurre cuando se produce una autointersección es que se representan como sombreados puntos que en realidad deberían estar iluminados (es decir, la salida de color para ciertos píxeles es, erróneamente, el vector $(0, 0, 0)$, correspondiente al color negro). Como hemos explicado anteriormente, esto se debe a los errores cometidos por la falta de precisión numérica. Como consecuencia, los puntos de intersección calculados por el algoritmo no se correspondan con los reales.

Así, la idea subyacente al método de Montecarlo que queremos implementar consiste en generar puntos aleatorios en un entorno del punto de intersección calculado. De esta forma, podremos conocer un poco mejor la geometría de las regiones próximas a la intersección real, y en función de los resultados obtenidos, establecer un criterio que decida si la luz alcanza o no al punto en cuestión, para obtener resultados más cercanos a la realidad.

Como punto de partida para la implementación, utilizaremos el método de desplazamiento en la dirección del vector normal de la superficie, multiplicado por un *offset* fijo que requiere un ajuste para adaptarse a la escena renderizada. Es el método más utilizado por ser fiable en la mayoría de situaciones y fácil de implementar, pero hay casos en los que es difícil encontrar el valor adecuado para una escena concreta, o simplemente no es posible encontrar un valor que ofrezca resultados satisfactorios para todos los objetos presentes en la imagen final.

Supongamos entonces que partimos de este método con un *offset* previamente fijado (por ejemplo, $\varepsilon_0 = 10^{-5}$), con el que renderizamos una escena donde se producen autointersecciones claras en el objeto más lejano, como se muestra en la Figura 4.1. Las esferas azul y verde se encuentran cerca del origen, por lo que los cálculos de sus intersecciones no son problemáticos. Sin embargo, la esfera roja es de gran tamaño y se encuentra a una distancia muy grande del origen (del orden de 10^{10}), lo que propicia la aparición del *surface acne* del que hemos hablado en la Sección 2.1.2.

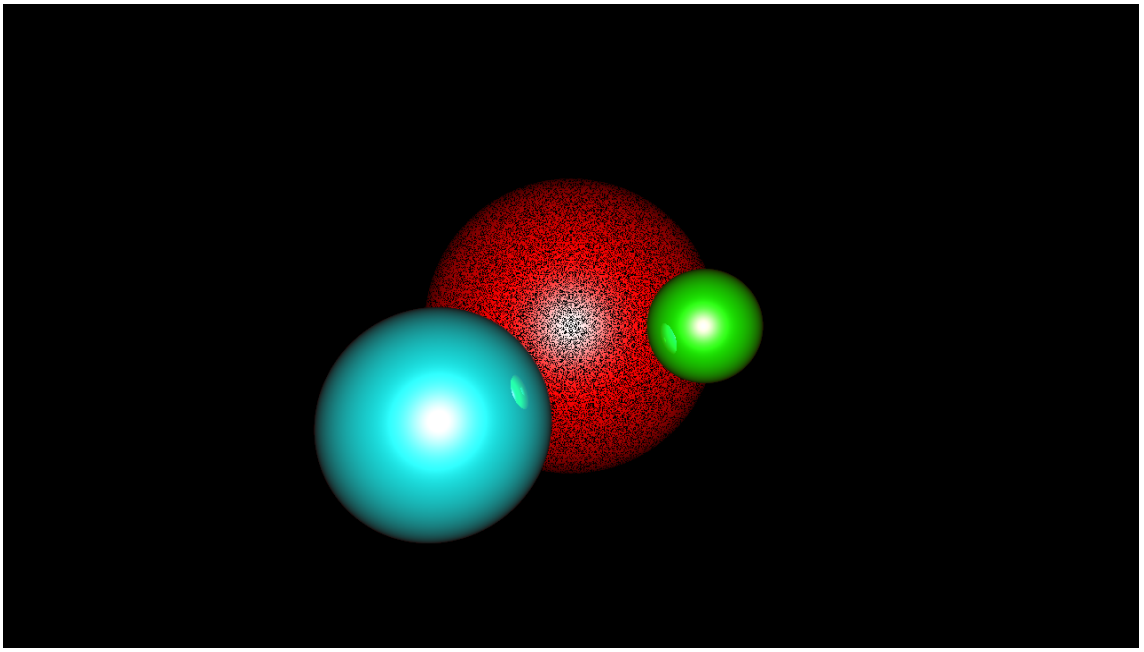

Figura 4.1: Esfera distante con problemas de autointersección. Fuente: Elaboración propia.

Lo que ocurre es que el error cometido en el cálculo es superior al valor de ε_0 que hemos considerado; por contrapartida, y como hemos explicado anteriormente, aumentar ε_0 en exceso provocaría desplazamientos en las sombras y reflejos de otros objetos en la escena, por lo que no es una solución recomendable.

En su lugar, lo que vamos a intentar es encontrar una distancia de desplazamiento que se adapte mejor a cada punto de intersección, y que llamaremos Δ. Sea r un valor aleatorio de una distribución uniforme continua en el intervalo $[0, 1]$, entonces definimos:

$$\Delta = r\varepsilon_1 \tag{4.1}$$

donde ε_1 será una cota superior de la distancia de desplazamiento tal que $\varepsilon_1 \geqslant \varepsilon_0$. El hecho de tomar ε_1 mayor que ε_0 hace posible que Δ sea mayor o menor que ε_0, por lo que generar varios valores distintos de Δ (generando distintos números aleatorios) nos permitirá estudiar puntos más alejados o más cercanos al punto de intersección original que con el método de partida.

Por lo tanto, nuestro método de Montecarlo consistirá en generar n valores aleatorios r_k, donde n es un parámetro configurable del programa. Como resultado, obtenemos también n distancias, que denotamos $\Delta_k = r_k\varepsilon_1$ o, equivalentemente, n puntos p_k de la forma:

$$p_k = q + \Delta_k \vec{n} \tag{4.2}$$

donde q es el punto de intersección calculado inicialmente, sin corrección, y $\vec{n}$ es el vector normal geométrico de la superficie.

Una vez generados estos puntos, lo que haremos será tomarlos como origen de los rayos de sombra, para comprobar si encuentran algún objeto de la escena en su trayectoria. Es necesario definir también los criterios que utilizaremos para elegir el mejor de esos puntos (el que se acerque más al punto de intersección real y, por tanto, ofrezca una imagen final más libre de autointersecciones).

4.3.1. Criterios de elección del mejor punto

Si recordamos la causa de las autointersecciones, en general ocurre que el punto de intersección calculado cae por debajo de la superficie, de forma que al generar el rayo de sombra, corta al mismo objeto. Entonces, lo que haremos será desplazar el punto calculado en la dirección del vector normal. Para hacerlo progresivamente, almacenaremos los números aleatorios generados en un vector, y una vez generados todos, los ordenamos en orden creciente. De esta forma, saltaremos de punto en punto hasta alcanzar el más apropiado.

Para identificar el punto correcto, tenemos que buscar el primero de ellos que no encuentre obstáculos en su trayectoria, ya que eso significa que es el punto más cercano a la superficie real pero sin estar por detrás de esta.

Relacionado con lo anterior, hay que tener en cuenta una condición importante que caracteriza aquellos puntos donde se produce una autointersección: si se genera un rayo de sombra con origen en dichos puntos, existe un objeto de la escena que se encuentra en la trayectoria entre el objeto intersecado y la fuente de luz. Por lo tanto, definiremos en nuestro programa una variable lógica (*is_shadowed*) que servirá para identificar aquellos puntos que son susceptibles a tener autointersecciones. De esta forma, el método de Montecarlo se aplicará solamente a aquellos puntos que verifiquen la condición *is_shadowed=True*, reduciendo también el coste computacional del programa.

Entre los puntos que verifican la condición anterior, encontraremos dos situaciones:

- Si un punto está correctamente sombreado, no a causa de una autointersección, nuestro método de Montecarlo no encontrará ningún punto p_k tal que *is_shadowed=False*.

- Si un punto debe estar iluminado pero no lo está a causa de una autointersección, se intentará buscar el p_k más cercano al punto de intersección calculado inicialmente, q, tal que *is_shadowed=False*.

4.3.2. Implementación en Python

Ahora que hemos explicado el funcionamiento del método, ya podemos transcribirlo a código de Python. Se trata de una idea conceptualmente sencilla, y que por lo tanto no ofrece grandes complicaciones en cuanto a su programación. Se muestra a continuación el código resultante:

```
#METODO DE MONTECARLO
iteraciones = 3 #Numero de iteraciones del metodo de Montecarlo
nalea = 0

while is_shadowed and nalea<iteraciones:

    #Genero numeros aleatorios
    aleatorios = np.random.uniform(size=iteraciones, )
    np.sort(aleatorios)

    shifted_point = intersection + aleatorios[nalea] * 1e-3 *
        normal_to_surface #offsetting
    intersection_to_light = normalize(light.position -
        shifted_point) #Direccion del rayo de sombra

    _, min_distance = nearest_intersected_object(objects,
        shifted_point, intersection_to_light)
    intersection_to_light_distance = np.linalg.norm(light.position
        - intersection)

    is_shadowed = min_distance < intersection_to_light_distance
    nalea+=1
```

Como observación, trabajamos con un generador de valores aleatorios uniformes en $(0, 1)$, aunque también se pueden generar directamente valores en $(0, \varepsilon_1)$ utilizando el argumento de entrada $high = \varepsilon_1$. En tal caso, se eliminaría la multiplicación por ε_1 en la

línea de definición de *shifted_point*.

4.4. Comparativa visual de los métodos

En las secciones anteriores de este trabajo hemos implementado dos ray tracers en un paradigma orientado a objetos: uno de ellos utilizando el método habitual de desplazamiento en la dirección del vector normal con un *offset* fijo, basado en el código de Aflak (2020), y otro con el procedimiento de Montecarlo que acabamos de presentar. En esta sección, mostraremos los resultados de renderizar algunas imágenes con cada uno de estos dos métodos, estableciendo una comparativa visual y destacando algunos aspectos importantes que se puedan observar.

4.4.1. Render 1

Comenzaremos con la misma imagen de la Figura 4.1, que presentaba autointersecciones en la esfera de color rojo. El siguiente bloque de código nos permitirá generar dicha escena si lo pegamos dentro del vector *objects* del programa principal.

```
#light = Light(np.array([0, 1, 0]), np.array([1, 1, 1]), np.array
    ([1, 1, 1]), np.array([1, 1, 1]))

Sphere(np.array([0, 0, -2*1e10]), 0.5*1e10, np.array([0.1, 0, 0]),
    np.array([1, 0, 0]), np.array([1, 1, 1]), 100, 0.5),
Sphere(np.array([0.7, 0, -2]), 0.3, np.array([0.1, 0, 0]), np.array
    ([0, 1, 0]), np.array([1, 1, 1]), 100, 0.5),
Sphere(np.array([-0.7, -0.5, -2]), 0.6, np.array([0.1, 0, 0]), np.
    array([0, 1, 1]), np.array([1, 1, 1]), 100, 0.5),
```

Para el ray tracer normal, tomaremos un offset fijo de 10^{-5}, mientras que para nuestro método de Montecarlo tomaremos inicialmente $\varepsilon_0 = 10^{-5}$, $\varepsilon_1 = 10^{-4}$ y $n = 3$. Los resultados se muestran en la Figura 4.2 y la Figura 4.3.

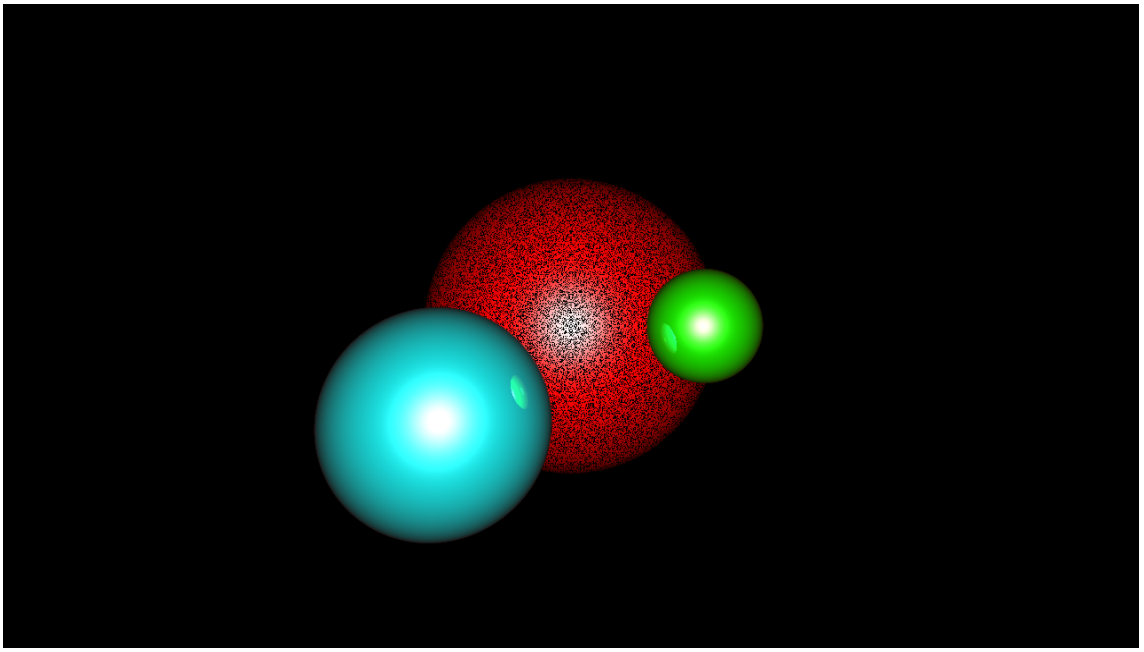

Figura 4.2: Ray tracer original, $\varepsilon = 1e-5$. Fuente: Elaboración propia.

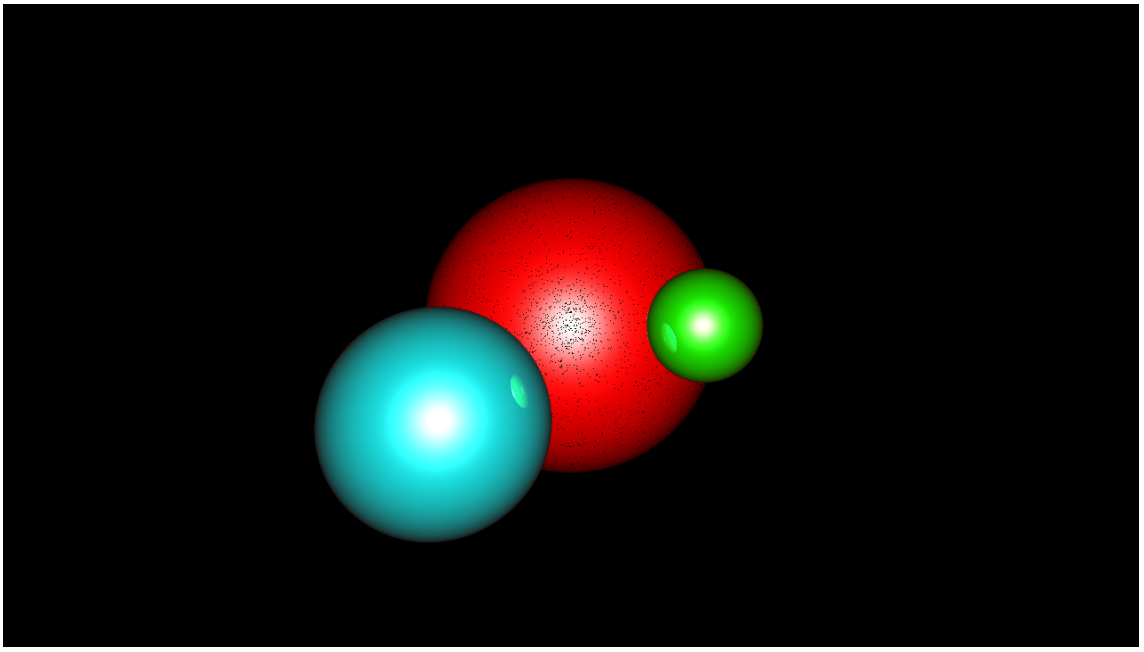

Figura 4.3: Ray tracer de Montecarlo, $n = 3,\ \varepsilon_0 = 1e-5,\ \varepsilon_1 = 1e-4$. Fuente: Elaboración propia.

Se observa que el ray tracer de Montecarlo ofrece una imagen final con un número de

autointersecciones muy reducido. Además, en un examen más exhaustivo de la imagen no se encuentran efectos inesperados en las dos esferas al frente ni en sus reflejos.

Un hecho a tener en cuenta es que, a diferencia del ray tracer de partida, ahora estamos trabajando con un método estocástico, por lo que el resultado será distinto en cada ejecución. De esta forma, si ejecutamos de nuevo el código que nos ha permitido generar la Figura 4.2, obtendremos exactamente el mismo resultado, con los mismos puntos de autointersección, al tratarse de un método determinista. Por otro lado, si hacemos lo mismo con la escena de la Figura 4.3, se obtendrá una imagen ligeramente distinta.

Para continuar con las pruebas de nuestro algoritmo, renderizaremos la escena con el ray tracer de Montecarlo, pero tomando esta vez $\varepsilon_1 = 10^{-3}$. Al hacer este cambio, estamos permitiendo que Δ tome valores mayores que en el caso anterior, esperando así que el número de autointersecciones sea menor. La imagen resultante es la que se muestra en la Figura 4.4.

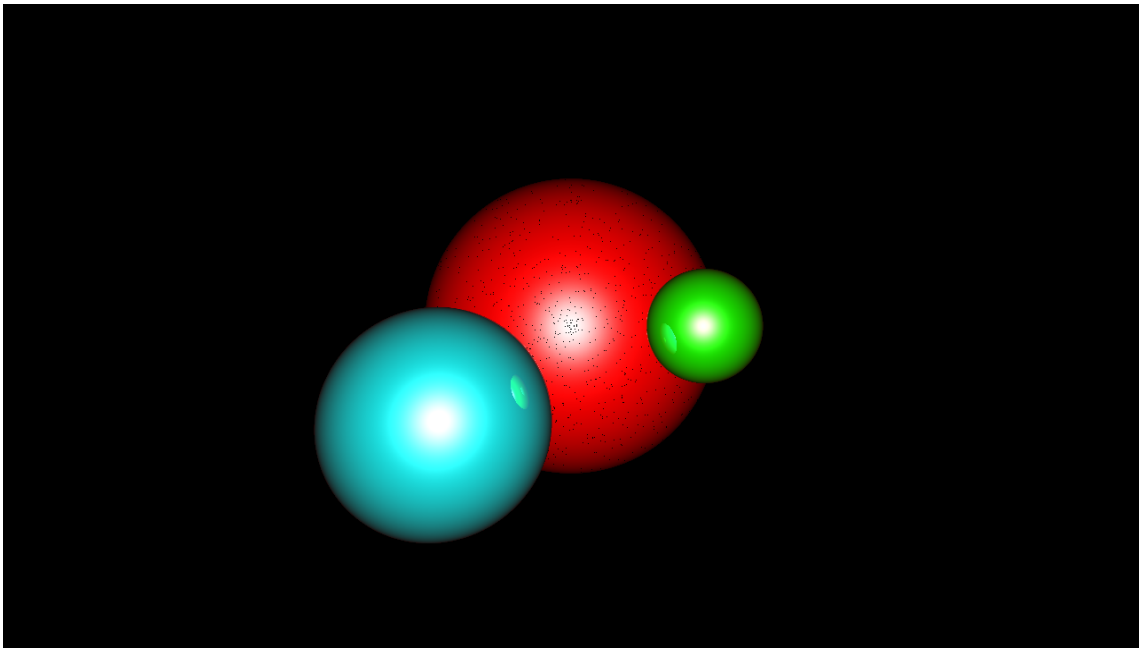

Figura 4.4: Ray tracer de Montecarlo, $\varepsilon_0 = 1e-5,\ \varepsilon_1 = 1e-3,\ n = 3$. Fuente: Elaboración propia.

Como se esperaba, el número de autointersecciones es menor, ya que esta vez hemos permitido la generación de valores aleatorios más grandes capaces de corregir errores mayores. Una observación exhaustiva de la imagen no muestra defectos visuales en los demás

objetos de la escena ni en los reflejos.

Probamos ahora tomando $\varepsilon_1 = 10^{-1}$, que es un valor relativamente grande. Se espera que el número de autointersecciones sea todavía menor, pero queremos ver si la elección de un valor grande tiene un impacto negativo en la imagen final. El resultado se muestra en la Figura 4.5.

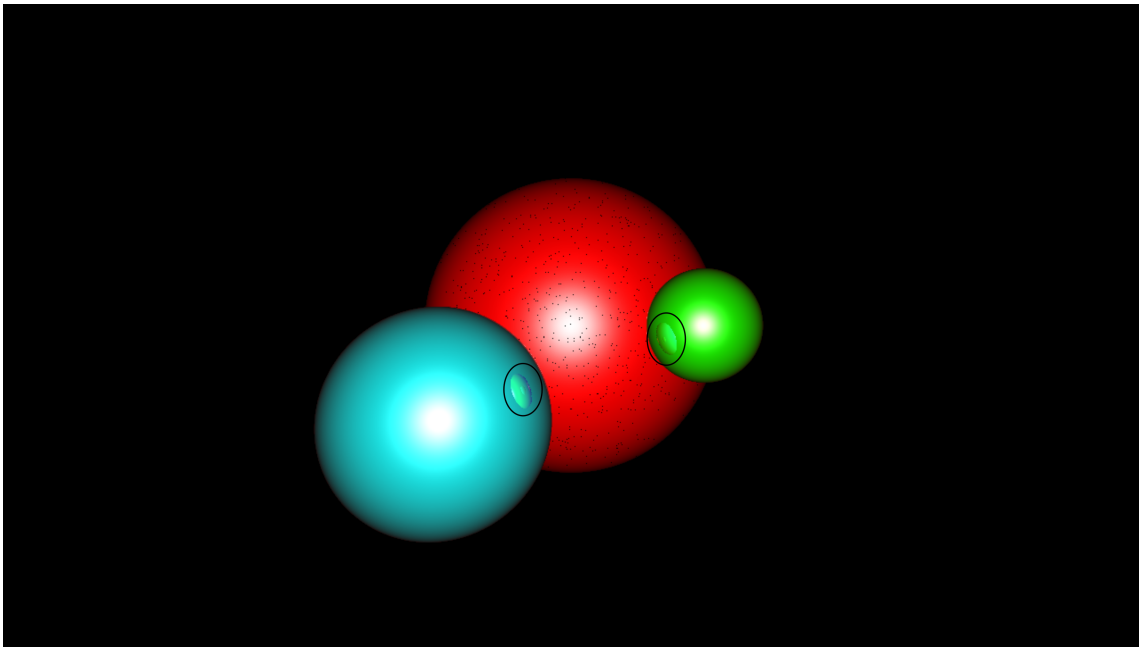

Figura 4.5: Ray tracer de Montecarlo, $\varepsilon_0 = 1e-5,\ \varepsilon_1 = 1e-1,\ n = 3$. Fuente: Elaboración propia.

Nuevamente, se reduce la cantidad de autointersecciones en la esfera roja. Sin embargo, observando cuidadosamente la imagen, se pueden apreciar contornos extraños en los reflejos de las esferas azul y verde. Concluimos, por tanto, que existe también un compromiso entre el número de autointersecciones y los efectos sobre el resto de la escena a la hora de elegir un valor de ε_1 más o menos grande.

A pesar de la limitación que acabamos de encontrar, los resultados para esta escena siguen siendo mejores que los que se obtendrían con el método usual. Aún escogiendo un valor grande como $\varepsilon = 10^{-1}$, los resultados con el método usual siguen mostrando muchas autointersecciones, además de las incorrecciones en los reflejos cercanos, como se muestra en la Figura 4.6.

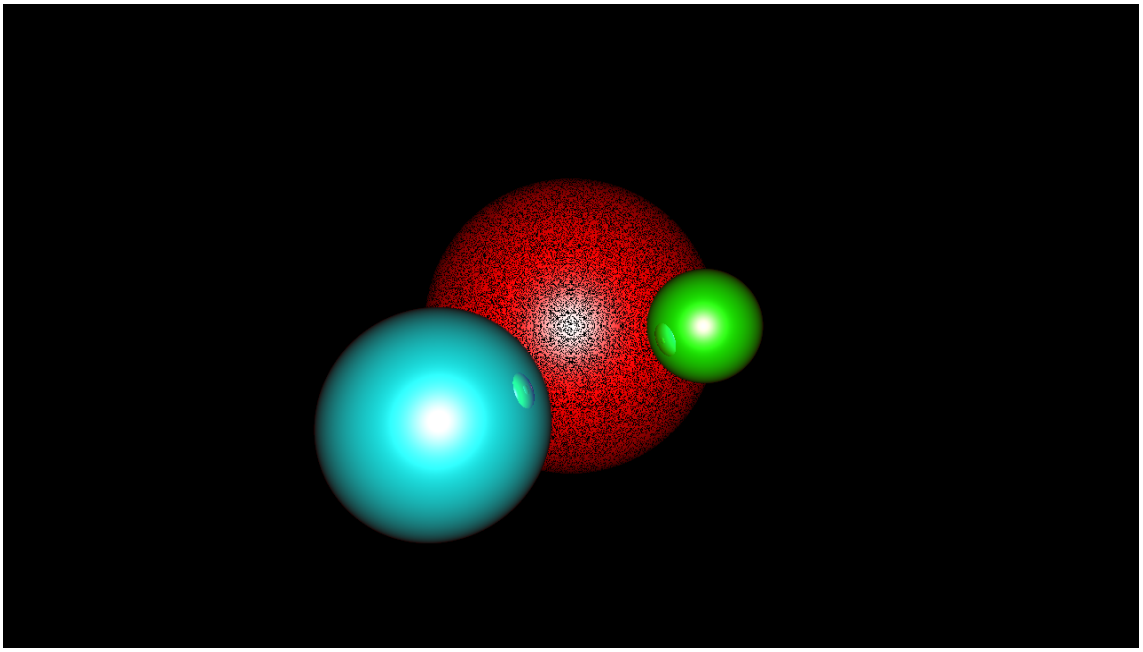

Figura 4.6: Ray tracer original, $\varepsilon_0 = 1e - 1$. Fuente: Elaboración propia.

Como se puede observar, la elección de un *offset* tan grande provoca desplazamientos notables en reflejos y sombras cercanos, y además no es capaz de tratar adecuadamente las autointersecciones de la esfera roja. Por ello, la principal ventaja del método expuesto en este trabajo es que permite corregir autointersecciones a gran distancia del origen minimizando el impacto de las correcciones sobre sombras y reflejos cercanos al observador.

Nos quedaremos entonces con el valor de $\varepsilon_1 = 10^{-3}$, que no provocaba defectos en los reflejos. Veremos ahora qué ocurre cuando aumentamos la cantidad de números aleatorios generados para nuestro experimento de Montecarlo ($n = 10$). Lógicamente, aumenta el coste computacional, pero esto lo veremos en la siguiente sección; por ahora, nos centramos únicamente en la calidad de la imagen resultante (Figura 4.7).

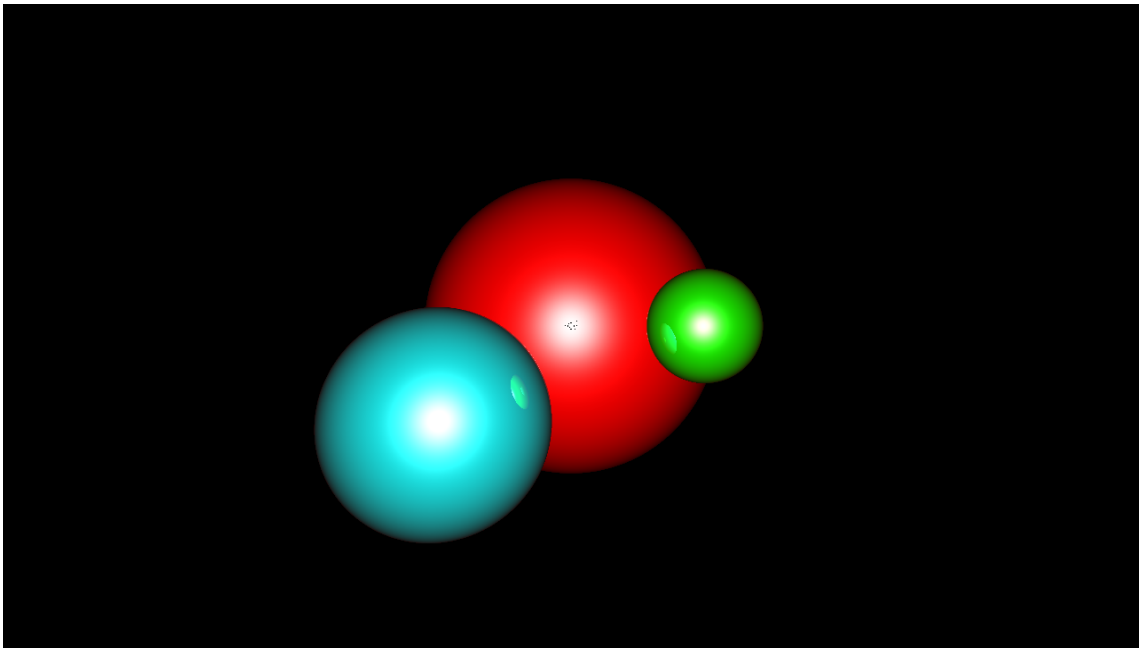

Figura 4.7: Ray tracer de Montecarlo, $\varepsilon_0 = 1e{-}5,\ \varepsilon_1 = 1e{-}3,\ n = 10$. Fuente: Elaboración propia.

4.4.2. Render 2

Completaremos las pruebas de la calidad del método generando una escena inspirada en la caja de Cornell (*Cornell box*). Desde su introducción por parte de Goral et al. en 1984, se ha convertido en un test muy popular para poner a prueba la precisión de un algoritmo de renderizado. Lo que haremos ahora será modificar los parámetros de nuestro método y ver el impacto que tienen estas variaciones en la imagen final. Recordemos que, por lo visto en la sección anterior, la elección de un *offset* inicial excesivamente grande provoca defectos en reflejos y sombras cercanos, aunque en esta escena no tendremos objetos cerca de la cámara y, por lo tanto, este efecto no será tan notorio.

Comenzamos mostrando la imagen inicial, renderizada con el ray tracer original tomando un valor de $\varepsilon = 10^{-3}$, que no producía defectos en reflejos y sombras cercanos. Como nuestra intención es estudiar las autointersecciones, hemos definido la caja de Cornell a una escala lo suficientemente grande como para que se presente este problema de precisión en los cálculos. Así, la imagen resultante se muestra en la Figura 4.8.

En este caso, podemos probar también los efectos de utilizar un *offset* grande con el ray tracer original, ya que no afectará tanto a la calidad de la imagen final como en el ejemplo

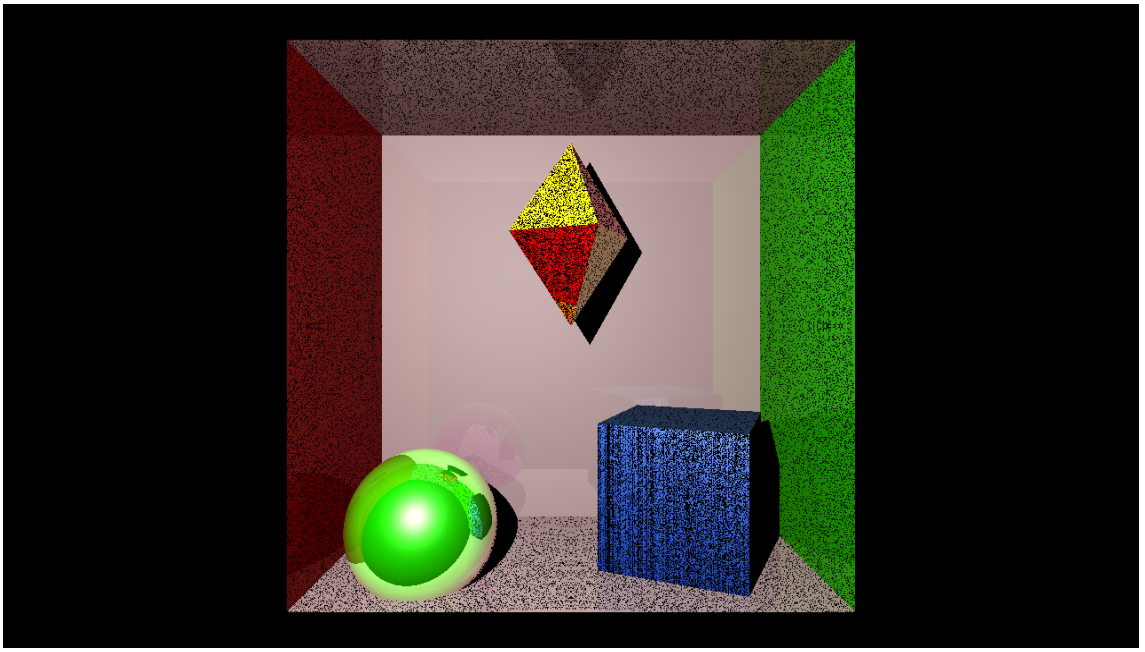

Figura 4.8: Ray tracer original, $\varepsilon = 1e-3$. Fuente: Elaboración propia.

que hemos visto en la sección anterior. Por lo tanto, si tomamos un valor de $\varepsilon = 0{,}5$, el resultado es el que se muestra en la Figura 4.9.

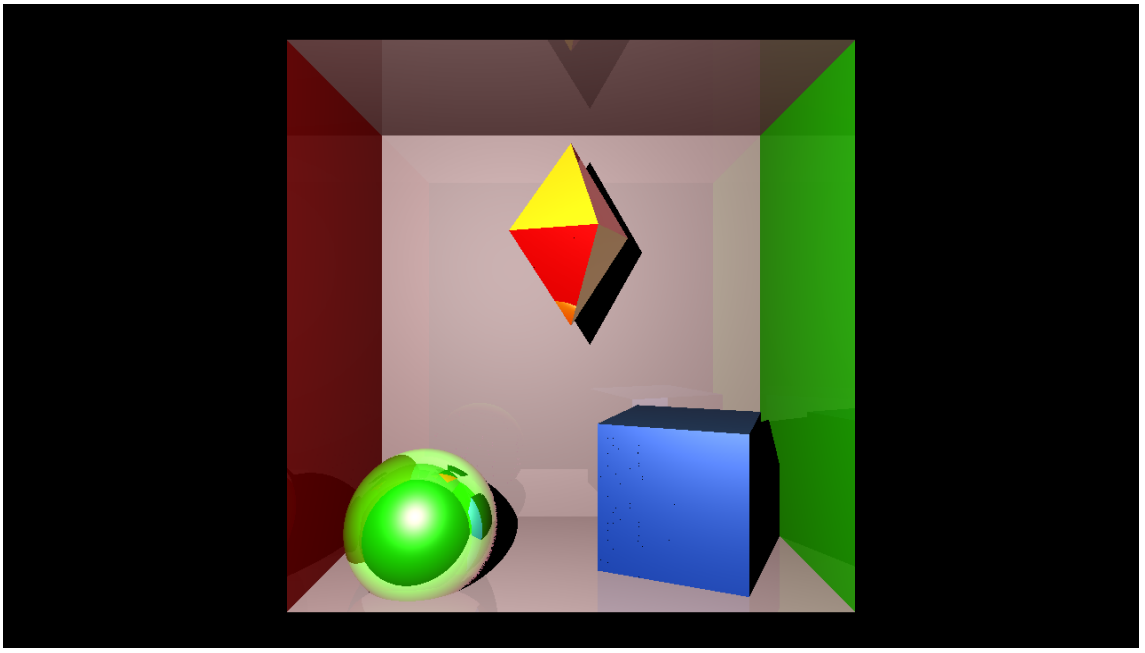

Figura 4.9: Ray tracer original, $\varepsilon = 5e-1$. Fuente: Elaboración propia.

El ray tracer original presenta aquí un muy buen funcionamiento, aunque todavía no es capaz de resolver todas las autointersecciones de la escena (se pueden apreciar en la cara frontal del cubo azul).

Pasamos ahora a comprobar el desempeño del método de Montecarlo descrito. Como la idea consiste en poder tomar valores de ε_1 relativamente grandes sin que esto perjudique excesivamente al resto de la escena, podemos comenzar probando con el valor de $\varepsilon_1 = 0{,}5$ que acabamos de utilizar, y ver qué tal funciona el método. Para la primera aproximación de la imagen tomamos $\varepsilon_0 = 10^{-3}$, el mismo valor utilizado para generar la Figura 4.8. Tras 3 iteraciones, la imagen resultante se muestra en la Figura 4.10.

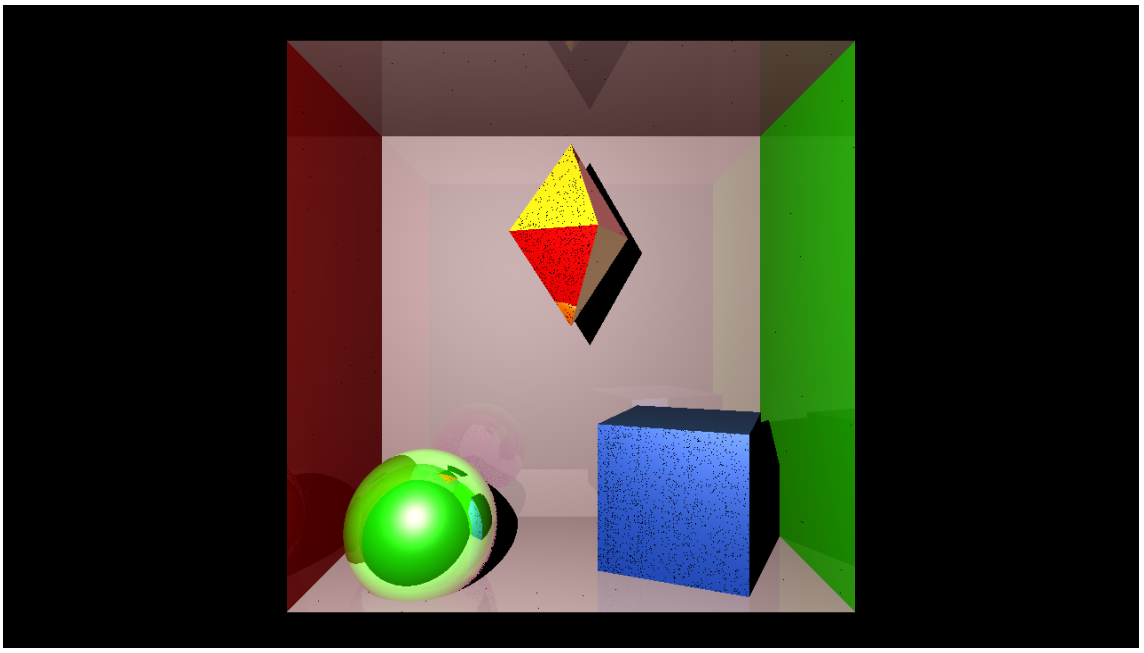

Figura 4.10: Ray tracer de Montecarlo, $\varepsilon_0 = 1e{-}3,\ \varepsilon_1 = 5e{-}1,\ n = 3$. Fuente: Elaboración propia.

La reducción en el número de autointersecciones con respecto a la imagen de referencia, en este caso la Figura 4.8, es notoria. Sin embargo, conviene hacer una observación sobre el impacto de este método en el resto de la escena. En las paredes de la caja de Cornell podemos observar los reflejos de los objetos de la escena en mayor o menor medida. Al estar trabajando con objetos a una escala muy grande, los reflejos que se ven en la imagen son muy distantes, a diferencia de lo que veíamos en la escena renderizada en la sección anterior. Mientras que el método de Montecarlo no mostraba grandes defectos en los reflejos

cercanos, en la Figura 4.10 sí que podemos ver colores extraños en el reflejo de la esfera sobre la pared trasera de la caja. Este mismo defecto está también presente en la Figura 4.8, donde no se utilizaba el método de Montecarlo, pero sí un valor de ε pequeño.

Continuando con nuestro estudio, podemos mejorar la eliminación de autointersecciones aumentando el número de iteraciones del método. Si probamos con $n = 10$, conservando los valores anteriores de ε_0 y ε_1, obtenemos como resultado la Figura 4.11.

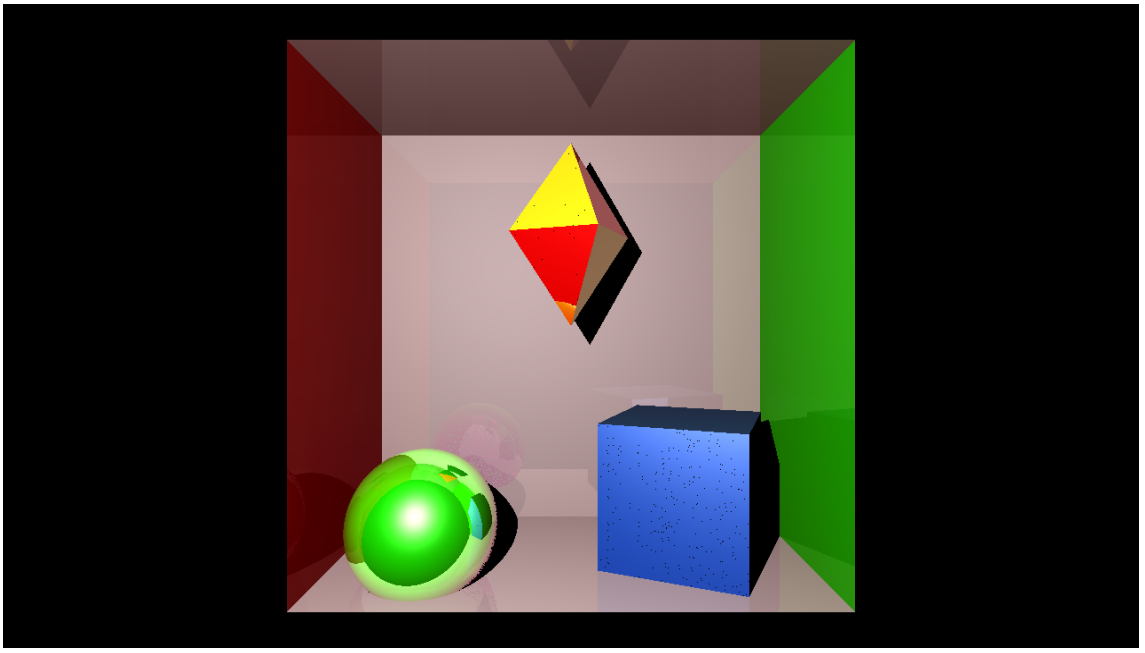

Figura 4.11: Ray tracer de Montecarlo, $\varepsilon_0 = 1e-3$, $\varepsilon_1 = 5e-1$, $n = 10$. Fuente: Elaboración propia.

Se consigue finalmente una imagen con un número de autointersecciones muy reducido con respecto a la imagen de referencia (Figura 4.8), sin necesidad de corregir la posición de todos los puntos de intersección computados inicialmente.

4.5. Comparativa de rendimiento

Aunque el método de Montecarlo se ha desarrollado principalmente enfocado a mejorar el problema de las autointersecciones en el aspecto visual, también es interesante establecer una comparativa de rendimiento entre los dos métodos presentados hasta ahora para estimar el coste adicional del método de Montecarlo.

En esta sección, lo que haremos será renderizar varias veces la escena mostrada en la

Figura 4.1 y medir el tiempo que tarda el algoritmo en renderizar la imagen completa. Repetiremos la prueba para distinto número n de iteraciones, donde $n = 0$ se corresponde con el tiempo que se tarda en renderizar la imagen con el método usual (desplazamiento en la dirección del vector normal mediante una distancia previamente fijada).

Para la realización de las pruebas, se ha utilizado un equipo con las siguientes características:

- Procesador: Intel Core i5-6600K (OC 4.3 GHz).
- Placa base: MSI Z170A Krait Gaming 3X.
- RAM: GSkill Ripjaws V Red DDR4 2400 PC4 19200 16GB 2x8GB CL15.
- GPU: Nvidia GeForce GTX 1060 6Gb.
- SSD (S.O.[1]): Kingston HyperX Fury 128Gb SATA 3.
- HDD: Samsung M3 Portable 1Tb, Seagate Expansion Portable 4Tb.
- Software:

  Python 3.9.

  IDE: Visual Studio Code 1.58.0.

Hay que tener en cuenta que este proceso de renderizado recae en el procesador utilizado, y no en la tarjeta gráfica. Por ello, como trabajo futuro, sería interesante trasladar la idea presentada en este trabajo a un entorno donde se aproveche el rendimiento extra que puede ofrecer una GPU para el renderizado de imágenes.

La resolución utilizada en este caso es 600x400. Los parámetros tomados para el método son $\varepsilon_0 = 10^{-5},\ \varepsilon_1 = 10^{-3}$.

En la siguiente tabla se recogen algunos datos sobre las mediciones obtenidas. En particular, para cada número de iteraciones del método de Montecarlo, se ha renderizado la misma imagen diez veces, midiendo el tiempo (en segundos) de cada iteración. Para cada número de iteraciones (i.e., para cada valor de n considerado), nos quedaremos con el mínimo, el máximo, y la media de los correspondientes diez tiempos de renderizado,

[1] S.O: Sistema Operativo.

con los que elaboraremos una representación gráfica que nos permita ver cómo aumenta el tiempo de renderizado en función de n.

Iteraciones	Mínimo	Media	Máximo
$n = 0$	16,533546924591064	16,740593481063843	16,849934339523315
$n = 50$	17,175065040588379	17,441552448272706	17,530115604400635
$n = 100$	17,569011449813843	17,711868953704833	17,924062490463257
$n = 150$	17,757507324218750	17,885040783882140	18,095568180084229
$n = 200$	17,945006608963013	18,118329882621765	18,251708269119263
$n = 250$	18,269139289855957	18,458486533164979	18,592274665832520
$n = 300$	18,539416074752808	18,683304643630983	18,806701183319092
$n = 350$	18,808695554733276	18,923762655258180	19,115273952484131
$n = 400$	19,029104232788086	19,229239344596863	19,400114774703979
$n = 450$	19,372189044952393	19,568870139122009	19,696344614028931
$n = 500$	19,826972723007202	19,915269994735716	20,037410497665405
$n = 550$	20,032423257827759	20,239146757125855	20,482220411300659
$n = 600$	20,442327022552490	20,564965438842773	20,751500129699707
$n = 650$	20,807350635528564	20,982527351379396	21,147441387176514
$n = 700$	21,291056871414185	21,364660191535950	21,446640968322754
$n = 750$	21,415724039077759	21,656736683845519	21,775760889053345
$n = 800$	21,831611871719360	22,037361192703248	22,351221561431885
$n = 850$	22,411061525344849	22,527659749984743	22,607536077499390
$n = 900$	22,873824119567871	22,994201970100402	23,151081800460815
$n = 950$	23,368500709533691	23,531365370750429	23,725546121597290
$n = 1000$	23,826276540756226	23,947409486770631	24,153401613235474

Tabla 4.1: Tiempos de renderizado con el método uniforme (en segundos).

A continuación, la Figura 4.12 muestra la representación gráfica de los valores mínimos, medios y máximos para cada valor de n considerado.

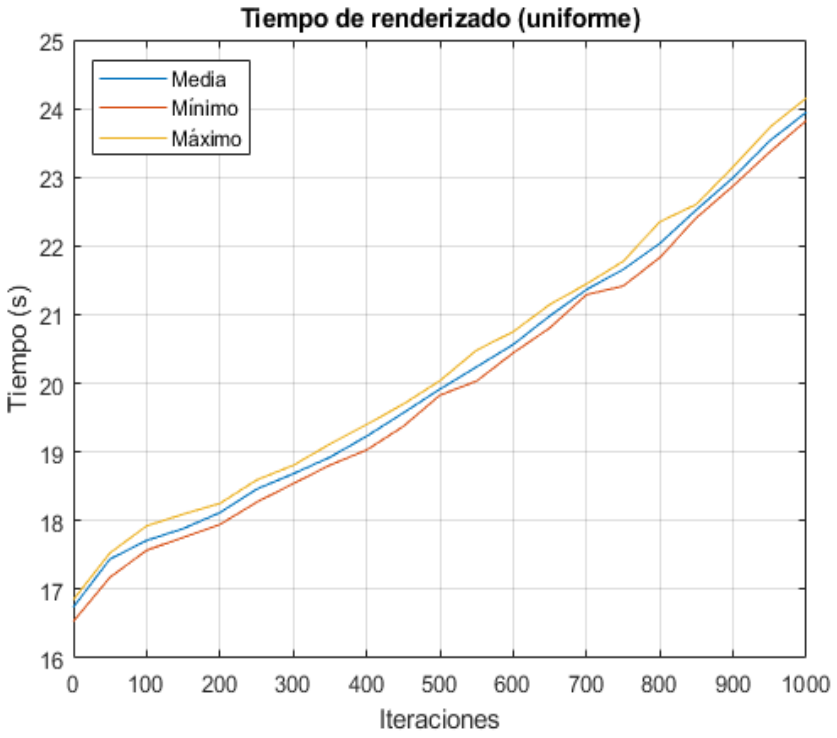

Figura 4.12: Gráfica de rendimiento del método uniforme. Fuente: elaboración propia.

Lógicamente, al aumentar el número de iteraciones del método de Montecarlo, aumenta también el coste computacional, por lo que el tiempo de renderizado es mayor. Además, se puede ver que el coste aumenta más rápidamente a medida que aumenta n; esto se debe a que el método, aunque eficaz, no es capaz de resolver todas las autointersecciones porque se limita a un cierto rango de valores de ε que en la práctica no es capaz de afrontar los errores cometidos de mayor magnitud. La consecuencia de esto es que el método de Montecarlo se ejecuta exactamente n veces para esos puntos, mientras que para otros puntos donde el error es pequeño el método puede concluir con una cantidad mucho menor de comprobaciones.

4.6. Otra variante: números aleatorios de una normal

Hasta este punto, hemos desarrollado un ray tracer orientado a objetos en Python que implementa un método de Montecarlo para mejorar la calidad de la imagen final. Por defecto, hemos utilizado un generador de números aleatorios uniformes, pero podemos generar nuevas variantes del método planteando el uso de distribuciones distintas. En este caso, lo que haremos será proponer ahora el uso de números aleatorios de una **distribución**

normal, y ver qué tal se comporta esta nueva variante del método.

4.6.1. Idea general de la nueva variante

En el caso uniforme, el planteamiento del método se basaba en establecer una distancia máxima para desplazar el punto de intersección computado en la dirección del vector normal de la superficie ($\vec{n}$). Esta distancia máxima es lo que hemos llamado hasta ahora ε_1, y el experimento de Montecarlo consistía en buscar mejores aproximaciones del punto de intersección real a una distancia menor que ε_1 de la intersección calculada inicialmente (q). Al ser una cota que verifica $\varepsilon_1 \geqslant \varepsilon_0$, esto implica que los puntos resultantes del desplazamiento podrían estar a una distancia de q mayor o menor que ε_0.

Para incluir ahora una distribución normal en nuestro problema, es necesario replantearlo y adaptarlo a los nuevos parámetros. Mientras que en el caso anterior bastaba con definir la cota máxima de la distancia y generar un valor aleatorio uniforme en el intervalo $(0, \varepsilon_1)$, en este caso tendremos que determinar la media (μ) y la desviación típica (σ) de la distribución normal utilizada.

Lógicamente, como resultado de esta nueva generación de números aleatorios, obtendremos valores mayores y menores que la media μ, y cuya distancia a la media será mayor cuanto mayor sea la desviación típica σ elegida. De esta forma, no tenemos tanta flexibilidad con los parámetros configurables del método como en el caso de la distribución uniforme, ya que antes nos limitábamos a establecer una cota superior de las distancias generadas; sin embargo, en este caso, necesariamente μ debe ser una buena aproximación del error real cometido en el punto, y esa misma necesidad de adecuar el parámetro a la escena se extiende a la elección de σ. Así, como paso previo a la ejecución de este algoritmo, dedicaremos la siguiente sección a comentar muy brevemente la elección de los parámetros adecuados para cada escena.

Desde otra perspectiva, podemos pensar que una buena elección de μ y σ permitirá alcanzar rápidamente mejores aproximaciones de un punto de intersección real; mientras que con el generador de valores de una distribución uniforme estamos asignando la misma probabilidad a todos los puntos del intervalo $(0, \varepsilon_1)$, con un generador de valores de una normal estaremos concentrando los números aleatorios en regiones más cercanas a la media, y por lo tanto más cercanas a la solución de nuestro problema. De todas formas, y aunque el enfoque principal de este trabajo consiste en eliminar autointersecciones, la cuestión del rendimiento del método también es importante, y dedicaremos una pequeña sección a

estudiar el impacto del método en el tiempo que se tarda en renderizar una determinada escena con respecto al ray tracer original, sin aplicar ningún método de Montecarlo.

Desarrollamos ahora la idea de este método desde un punto de vista más formal. Llamamos $r_k,\ k = 1, ..., n$, a n valores aleatorios generados según una distribución normal de media μ y varianza σ. De esta forma, el método de Montecarlo que queremos desarrollar ahora consiste en estudiar la visibilidad de ciertos puntos definidos como sigue:

$$p_k = q + r_k\vec{n} \tag{4.3}$$

Obsérvese que, sin consideraciones adicionales, no existe ninguna restricción que impida la aparición de valores aleatorios negativos. Sea r_i para cierto $i \in \{1, ..., n\}$ un valor aleatorio de una distribución normal $N(\mu, \sigma)$ tal que $r_i < 0$. Entonces, se tiene que el punto $p_i = q + r_i\vec{n}$ es el resultado de desplazar q hacia el interior del objeto de la escena (ya que hemos tomado $\vec{n}$ como el vector normal exterior por defecto). Esta situación no nos interesa, así que lo que haremos es tomar el valor absoluto del valor aleatorio generado; es decir, definimos:

$$p_k = q + |r_k|\vec{n}, \tag{4.4}$$

y esta definición será válida en cualquier caso para aplicar nuestro nuevo método.

Igual que hacíamos con el método de Montecarlo con números aleatorios uniformes, lo que haremos ahora es ordenar los puntos p_k en orden creciente según su distancia a q; equivalentemente, consideramos que los números aleatorios r_k verifican la desigualdad $r_k \leqslant r_{k+1},\ k = 1, ..., n-1$. Haciendo esto, podemos establecer como criterio de parada la aparición del primer punto tal que, si generamos un rayo de sombra con origen en dicho punto, no corta a ningún objeto de la escena en su trayectoria hacia la fuente de luz.

4.6.2. Elección de μ y σ

Antes de poner a prueba el algoritmo de renderizado, debemos hacer una elección más cuidadosa de los parámetros μ y σ , a diferencia de lo que ocurría con la generación de valores aleatorios uniformes para nuestro experimento de Montecarlo.

Partiendo de una idea general del error cometido en las dos escenas que vamos a renderizar (las tres esferas y la caja de Cornell), podemos definir la media μ como un valor próximo a dicho error, de manera que solo nos falta por determinar el valor de σ o,

equivalentemente, el grado de dispersión de los valores aleatorios generados. Tenemos que tener en cuenta que:

- Si el valor de σ es demasiado pequeño, entonces todos los valores estarán muy próximos a la media, por lo que nuestro método diferirá muy poco del resultado obtenido con un *offset* fijo en el ray tracer original.
- Si el valor de σ es excesivamente grande, el rango de estudio de nuestro experimento de Montecarlo será muy disperso, por lo que se requeriría un número de iteraciones muy grande para alcanzar una solución satisfactoria. Además, un valor de σ grande también provocará la aparición de valores negativos, que en algunos casos podrían aumentar la cantidad de valores r_k tales que $|r_k| < \mu$, y aumentar el tiempo que se tarda en alcanzar la solución del problema si el error cometido es mayor que μ.

Como preliminares de las pruebas de renderizado, utilizaremos Python para generar 1000 valores aleatorios con unos ciertos parámetros, y estudiar los valores máximos y mínimos obtenidos para poder elegir μ y σ.

En el ejemplo de las esferas vimos con el generador de valores aleatorios uniformes que, si establecemos el máximo en $\varepsilon_1 = 10^{-3}$, se obtiene una imagen bastante limpia (Figura 4.4), así que podemos comenzar tomando $\mu = 10^{-3}$ y buscar ahora una buena elección de σ.

σ	Mínimo	Máximo
10^{-5}	0,0009642716529243515	0,0010394450472457507
10^{-3}	$1{,}3599500555611706e-06$	0,004180864660642991
10^{-1}	$7{,}362576515667274e-05$	0,3075887400843039
1	0,0004982044139689941	3,5155883019256704

Tabla 4.2: Valores máximos y mínimos con $\mu = 10^{-3}$.

A la vista de los resultados obtenidos, podemos extraer las siguientes conclusiones:

- Para $\sigma = 10^{-5}$, el rango de valores es muy pequeño, y el resultado final no sería el mejor que podemos conseguir con este método.
- Para $\sigma \in 10^{-3}, 10^{-1}$, parece que tenemos un rango de valores adecuado, que nos proporciona puntos a una cierta distancia de la media.

- Para $\sigma = 1$ aparecen valores muy grandes que pueden ser problemáticos; el rango en este caso es demasiado amplio.

Con estas consideraciones, ya tenemos un punto de partida para poder poner a prueba el algoritmo en la próxima sección.

Realizamos un estudio similar para la escena de la caja de Cornell, donde podríamos tomar un valor entre 0,1 y 0,5. Como un cambio en la media no es más que un cambio de localización, la tabla anterior nos sirve también como estudio de la dispersión en este caso, de manera que los valores de $\sigma = 10^{-3}$ y $\sigma = 10^{-1}$ parecen apropiados para llevarlos a la práctica.

4.6.3. Implementación del método: caso normal

Al tratarse solamente de un cambio en la distribución utilizada, el bloque de código necesario para implementar el método es muy similar al utilizado en el caso uniforme, modificando únicamente la línea de generación de los valores aleatorios (recordando tomar el valor absoluto para evitar la aparición de valores negativos) y la línea de definición del punto de intersección desplazado (*shifted_point*), en caso de que hayamos utilizado un generador de valores uniformes en el intervalo $(0, 1)$ para el método anterior. El código resultante es el siguiente:

```
1
2  iteraciones = 3 #Numero de iteraciones del metodo de Montecarlo
3  nalea = 0
4
5  while is_shadowed and nalea<iteraciones:
6
7      aleatorios = np.abs(np.random.normal(loc = 3e-1, scale = 1e-1,
           size = iteraciones))
8      np.sort(aleatorios)
9
10     shifted_point = intersection + aleatorios[nalea] *
           normal_to_surface
11     intersection_to_light = normalize(light.position -
           shifted_point)
12
```

```
13      _, min_distance = nearest_intersected_object(objects,
            shifted_point, intersection_to_light)
14      intersection_to_light_distance = np.linalg.norm(light.position
            - intersection)
15
16      is_shadowed = min_distance < intersection_to_light_distance
17      nalea+=1
```

4.6.4. Comparativa visual

Aprovechando el experimento de la sección anterior, ponemos a prueba el nuevo algoritmo renderizando las mismas dos escenas que se han mostrado para el caso del generador de valores aleatorios uniformes.

Render 1

Partimos de la escena de las tres esferas, renderizada con 0 iteraciones de Montecarlo y un offset $\varepsilon_0 = 10^{-3}$, que nos permite evitar imprecisiones en los reflejos de las esferas. El resultado se muestra en la Figura 4.13.

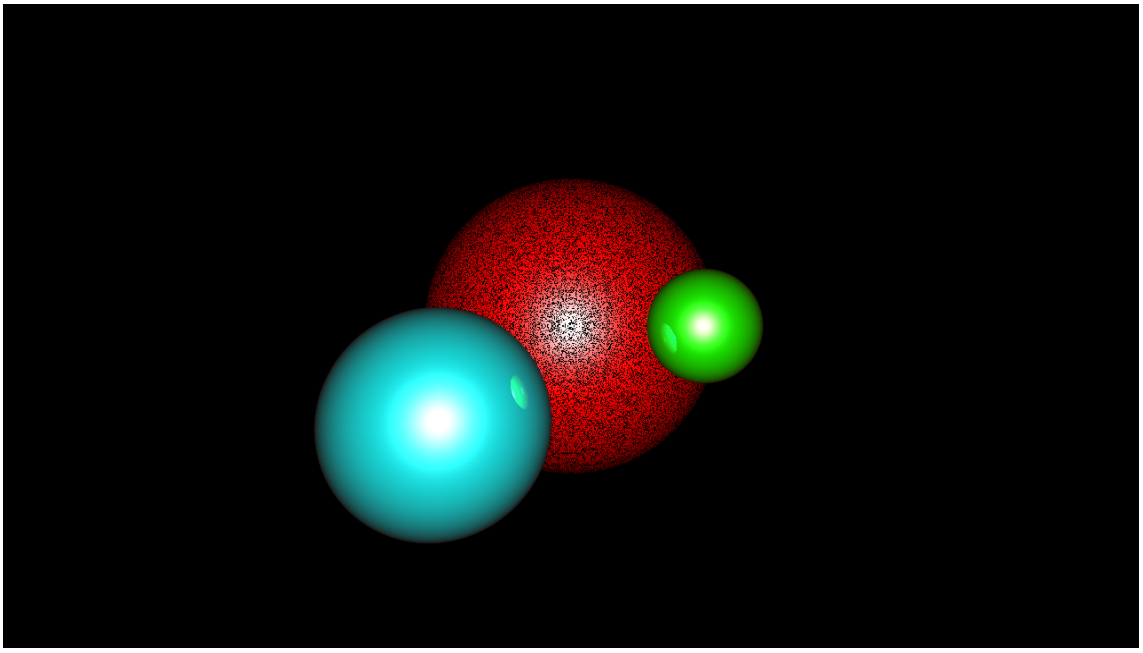

Figura 4.13: Ray tracer original, $\varepsilon = 1e-3$. Fuente: Elaboración propia.

Esta será nuestra imagen de referencia. Ahora, lo que haremos será estudiar el comportamiento del nuevo método de Montecarlo en relación con la figura anterior. Aplicando lo que hemos visto sobre la elección de μ y σ en una sección previa, comenzamos generando tres valores de una distribución normal $N(10^{-3}, 10^{-3})$, y el resultado es el que se muestra en la Figura 4.14.

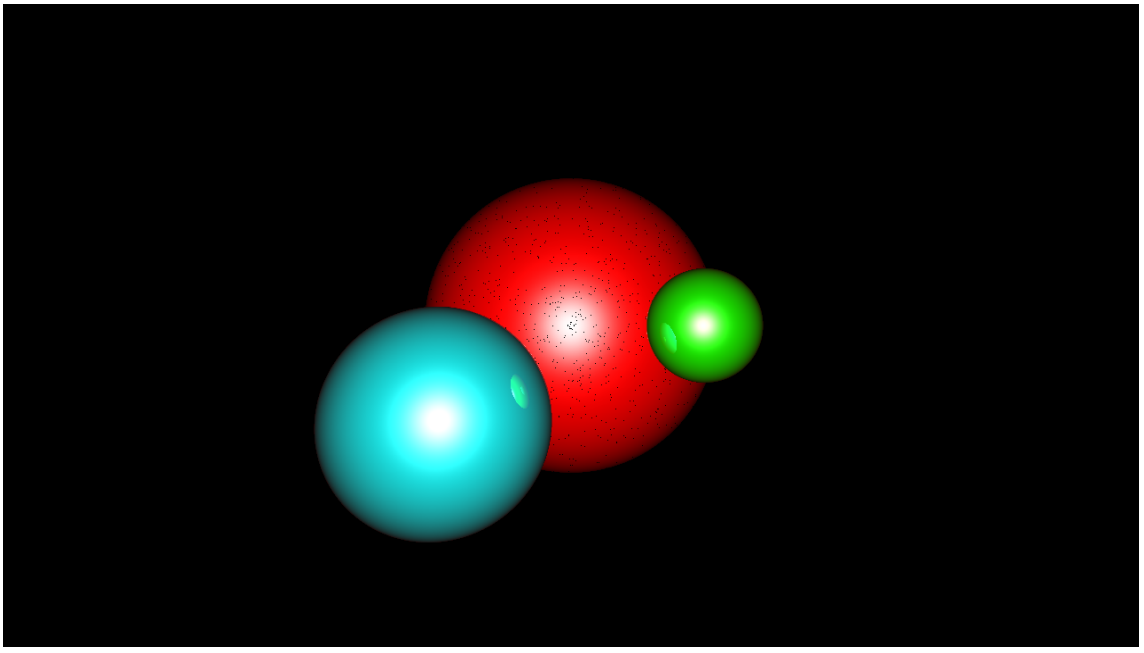

Figura 4.14: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 1e-3, \sigma = 1e-3, n = 3$. Fuente: Elaboración propia.

Con esta primera aplicación del método, ya se consigue una imagen final mucho más limpia que la que habíamos tomado como referencia. Tampoco se aprecian efectos sobre los reflejos de la escena, por lo que la elección de los parámetros de la distribución normal parece adecuada para seguir con nuestro estudio.

Lógicamente, aumentar el número de iteraciones a $n = 10$ nos devolverá un resultado mejor, como se puede ver en la Figura 4.15.

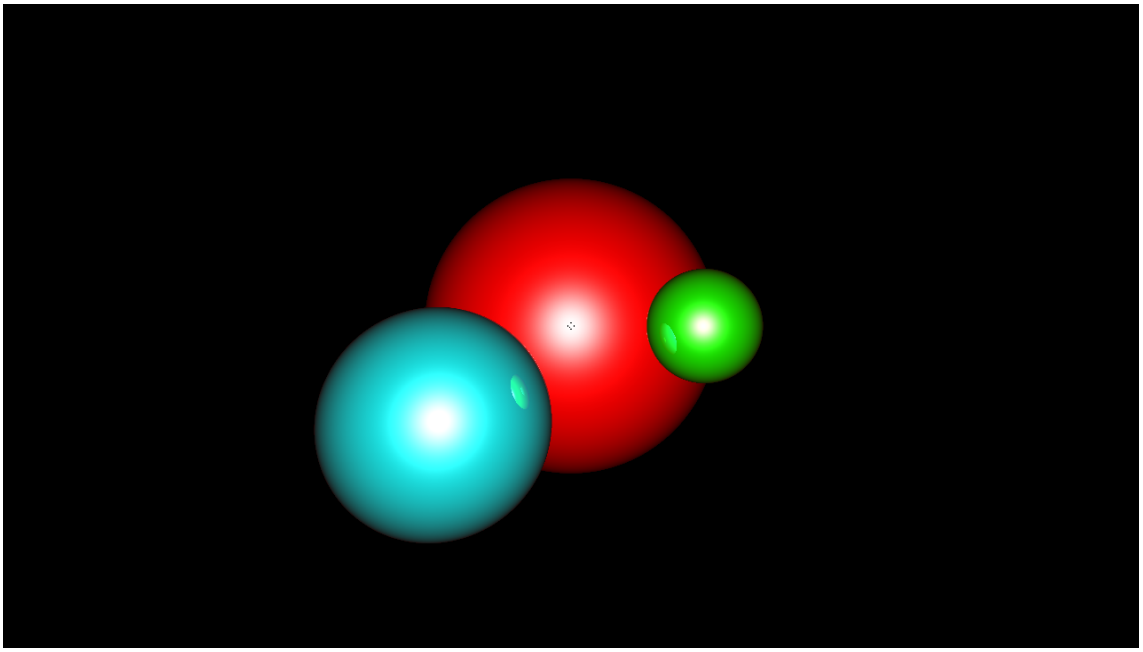

Figura 4.15: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 1e-3, \sigma = 1e-3, n = 10$. Fuente: Elaboración propia.

En este caso, las iteraciones extra han permitido corregir casi todas las autointersecciones de la esfera roja, salvo algunos puntos situados en la región central, que probablemente presenten un error demasiado grande como para resolverlo con este número de iteraciones. También es posible que se requiera una pequeña modificación de los parámetros μ y σ para obtener valores aleatorios que permitan mejorar aún más la imagen, sin aumentar en exceso el número de iteraciones del método.

En definitiva, los resultados obtenidos con $n = 10$ iteraciones han sido bastante satisfactorios, así que ahora fijamos este valor y aumentaremos la desviación típica tomando $\sigma = 10^{-1}$, lo que implica la aparición de valores más alejados de la media. Veamos qué ocurre en la Figura 4.16.

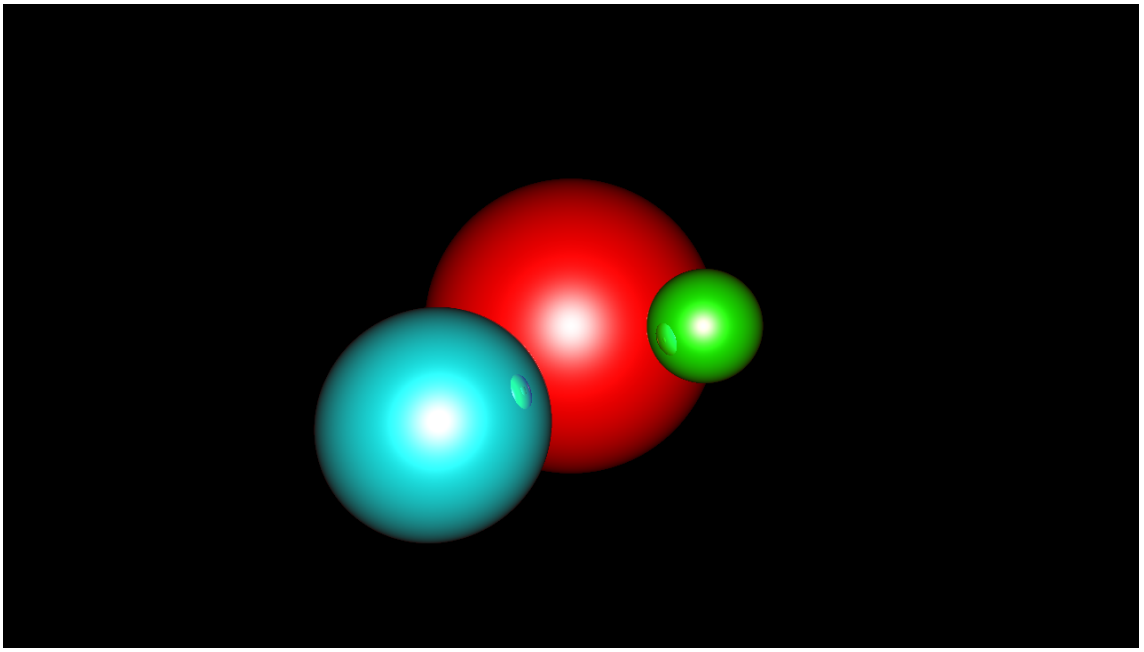

Figura 4.16: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 1e-3, \sigma = 1e-1, n = 10$. Fuente: Elaboración propia.

Así, el método demuestra ser también muy eficaz a la hora de eliminar autointersecciones en objetos distantes. Por otro lado, comienzan a aparecer defectos visuales en reflejos, por lo que puede que ese valor de la desviación típica sea demasiado elevado para tratar esta escena. Al igual que en el caso del método de Montecarlo con valores aleatorios uniformes, existe también un compromiso entre la elección de los correspondientes parámetros del método (ε_1 en el caso uniforme, μ y σ en el caso normal) y el efecto que tiene el algoritmo sobre los reflejos de la escena.

Render 2

De forma análoga a lo visto con el método de Montecarlo uniforme, renderizamos ahora nuestra escena basada en la caja de Cornell, teniendo en cuenta nuevamente los razonamientos acerca de la elección de valores de μ y σ adecuados.

Podríamos considerar $\varepsilon = 0{,}3$ como un buen punto de partida para renderizar la escena. Aunque vimos que un *offset* fijo de $\varepsilon = 0{,}5$ nos da un resultado muy bueno para esta escena (Figura 4.9), en esta sección queremos probar cómo se comporta el método de Montecarlo con distribución normal en igualdad de condiciones, así que buscamos una

imagen de referencia que presente suficientes autointersecciones para apreciar la capacidad de mejora del algoritmo. El resultado, sin aplicar el método de Montecarlo, es el que se muestra en la Figura 4.17, que tomaremos como referencia.

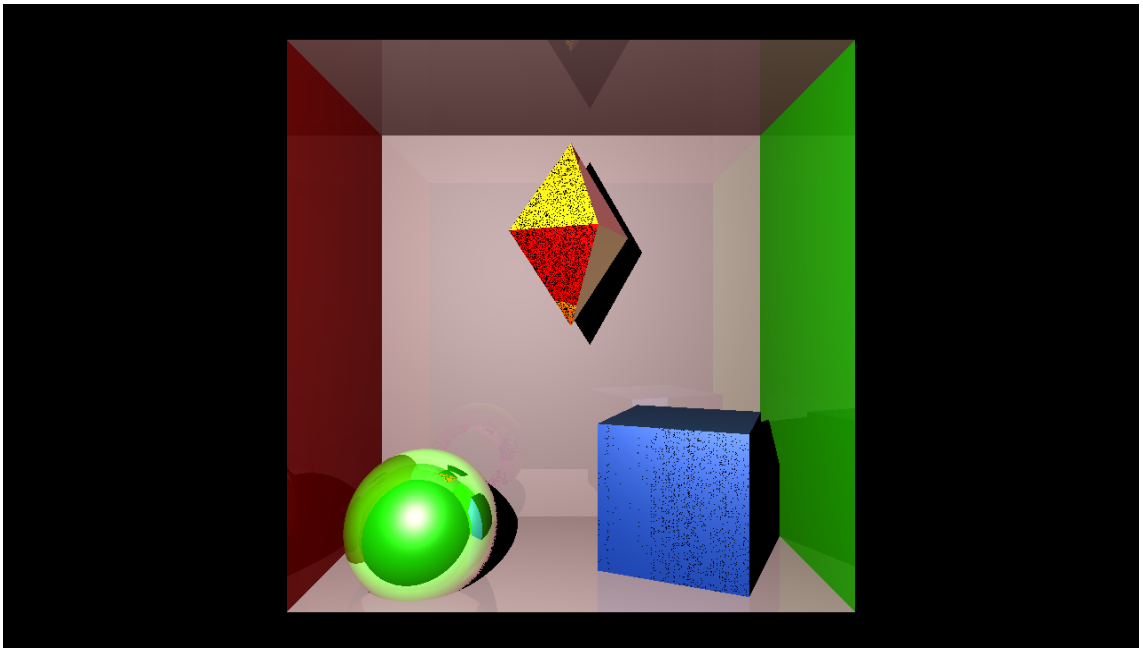

Figura 4.17: Ray tracer original, $\varepsilon_0 = 3e - 1$. Fuente: Elaboración propia.

Este valor fijo de ε corrige parte de las autointersecciones, pero todavía quedan bastantes puntos que no es capaz de tratar adecuadamente. Veamos cómo actúa el nuevo algoritmo para corregir esta imagen. Comenzamos nuevamente con 3 iteraciones del método, y generamos valores aleatorios de una distribución normal $N(0{,}3, 10^{-3})$, obteniendo como resultado la Figura 4.18.

La imagen resultante apenas mejora a la que habíamos tomado como referencia. Por lo tanto, antes de aumentar el número de iteraciones, lo que haremos será aumentar la desviación típica tomando $\sigma = 10^{-1}$, para comprobar que se obtiene un resultado mejor y que realmente tiene sentido seguir iterando con los mismos parámetros. Así, obtenemos la imagen que se muestra en la Figura 4.19.

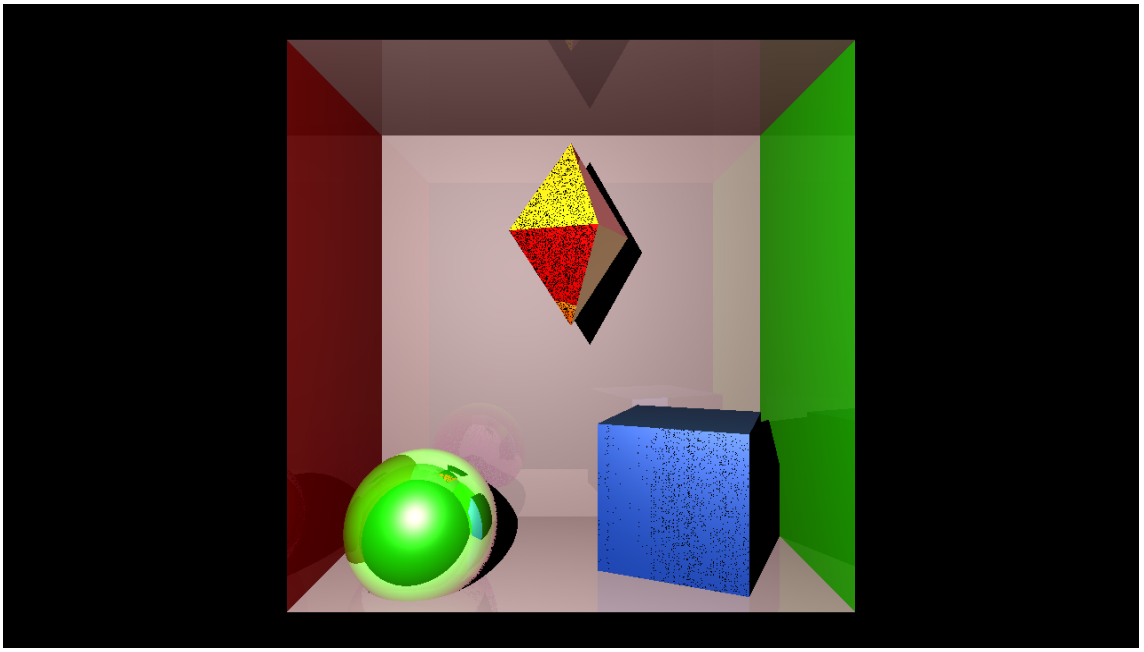

Figura 4.18: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 3e-1, \sigma = 1e-3, n = 3$. Fuente: Elaboración propia.

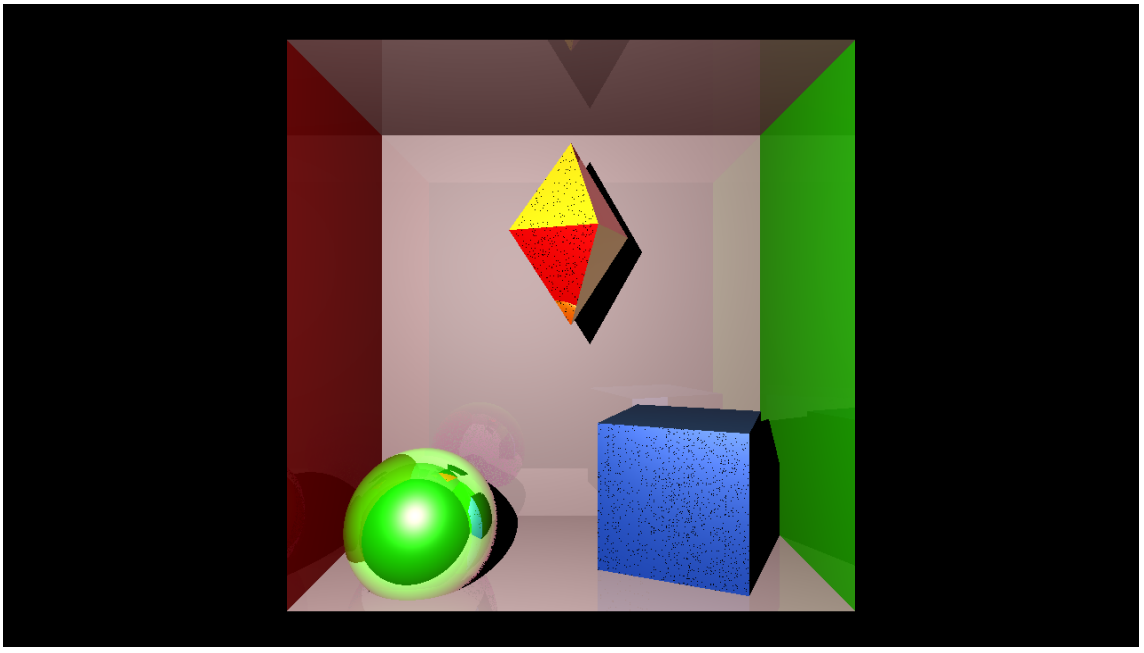

Figura 4.19: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 3e-1, \sigma = 1e-1, n = 3$. Fuente: Elaboración propia.

Esta vez se aprecia mejor la reducción en el número de autointersecciones, especialmente en el octaedro. Mejoramos ahora el resultado aumentando el número de iteraciones a $n = 10$ (Figura 4.20):

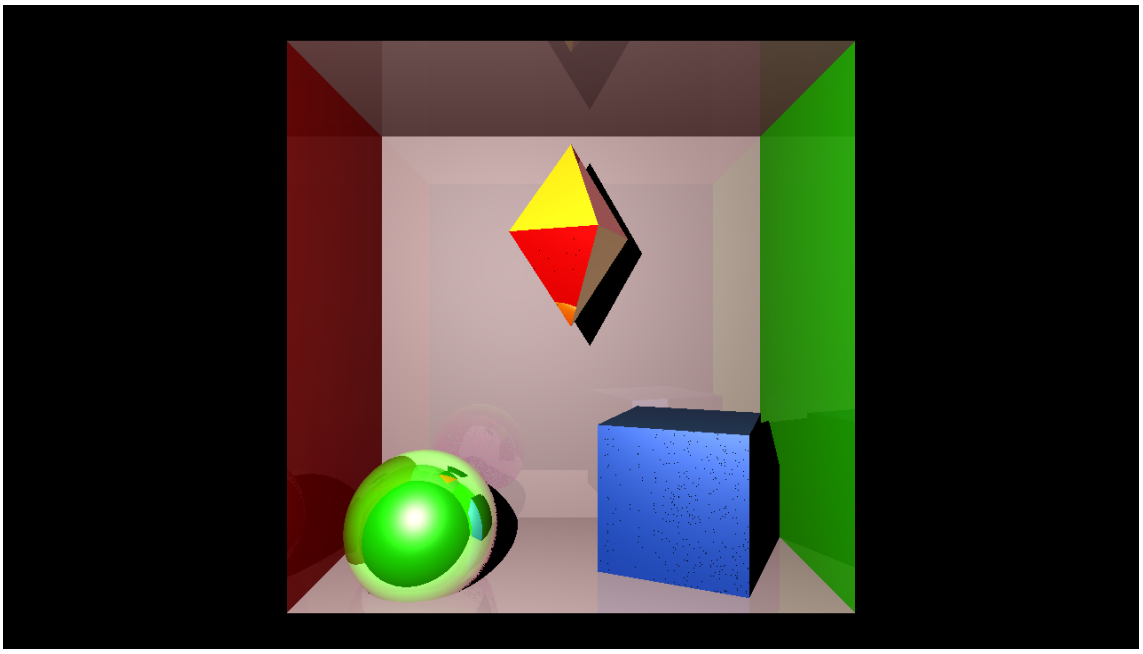

Figura 4.20: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 3e-1, \sigma = 1e-1, n = 10$. Fuente: Elaboración propia.

Esta vez ya tenemos una reducción considerable del número de autointersecciones con respecto a la imagen de referencia.

4.6.5. Prueba de rendimiento

Al igual que hemos hecho con el método de Montecarlo con valores aleatorios uniformes, haremos aquí una prueba de rendimiento del método midiendo el tiempo que tarda en renderizar la escena de las tres esferas, tal y como se muestra en la Figura 4.1. Lo haremos también a una resolución de 600x400, y con el mismo equipo utilizado para las pruebas de

la Sección 4.5.

Iteraciones	Mínimo	Media	Máximo
$n = 0$	16,458271741867065	16,608784365653992	16,741737365722656
$n = 50$	17,351613998413086	17,459129667282106	17,588117122650146
$n = 100$	17,551210880279541	17,636811041831969	17,815421581268311
$n = 150$	17,815160036087036	17,878045225143431	17,951473712921143
$n = 200$	17,847248554229736	18,047540903091431	18,313628673553467
$n = 250$	18,112385511398315	18,338250112533569	18,459803581237793
$n = 300$	18,429094552993774	18,587766075134276	18,791694879531860
$n = 350$	18,757297277450562	18,848754835128783	18,981142759323120
$n = 400$	19,011215209960938	19,096272754669190	19,259053707122803
$n = 450$	19,229952573776245	19,460469770431519	19,618365526199341
$n = 500$	19,828798532485962	19,939676642417908	20,055594444274902
$n = 550$	20,038781881332397	20,209596133232118	20,345814943313599
$n = 600$	20,346701383590698	20,521212339401245	20,669715642929077
$n = 650$	20,804662942886353	20,995446300506593	21,274324655532837
$n = 700$	21,119857311248779	21,290971350669860	21,606620073318481
$n = 750$	21,531534433364868	21,711121153831481	21,869796752929688
$n = 800$	22,032909154891968	22,162517833709718	22,252809762954712
$n = 850$	22,287664413452148	22,494331932067873	22,710852622985840
$n = 900$	22,944627285003662	23,012097907066345	23,161762237548828
$n = 950$	23,300788402557373	23,464033770561219	23,679044723510742
$n = 1000$	23,915745973587036	24,007209014892577	24,094969034194946

Tabla 4.3: Tiempos de renderizado con el método normal (en segundos).

A continuación, la Figura 4.21 muestra la representación gráfica de los valores mínimos, medios y máximos para cada valor de n considerado.

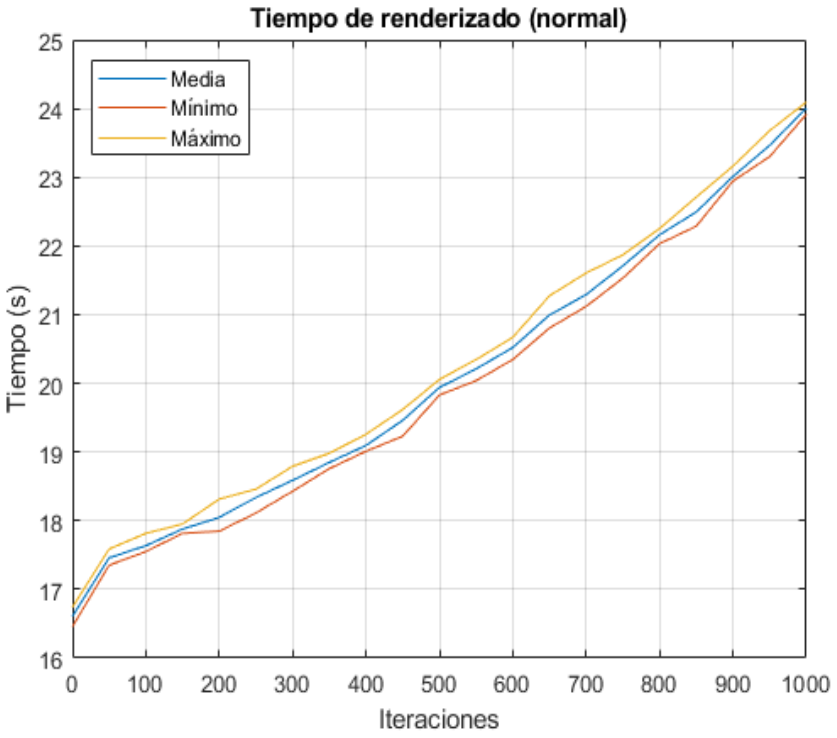

Figura 4.21: Gráfica de rendimiento del método normal. Fuente: elaboración propia.

Con los parámetros utilizados, se obtiene una gráfica de rendimiento muy similar a la que habíamos obtenido con el método de los números uniformes (Figura 4.12).

Capítulo 5

Conclusiones y Trabajo Futuro

La idea principal de este trabajo consiste en abordar el problema de las autointersecciones desde una nueva perspectiva, llevando a cabo experimentos de Montecarlo que permitan corregir las imprecisiones cometidas por el algoritmo original. De esta forma, se puede conseguir una imagen final más precisa, y de una forma sencilla evitando la búsqueda de un *offset* fijo apropiado para cada escena en particular. Tras todo el estudio realizado en relación con el tema, se ha llegado a las siguientes conclusiones:

- Se han expuesto dos alternativas diferentes: una utilizando la generación de valores aleatorios según una distribución uniforme continua, y la otra utilizando valores de una distribución normal. La idea subyacente en ambos casos es la realización de experimentos de Montecarlo para automatizar el proceso de corrección del cálculo de intersecciones, siendo esta la principal aportación del trabajo.

- La implementación de ambos métodos se ha hecho de forma simple e intuitiva, requiriendo únicamente el uso de los generadores de números aleatorios de la librería NumPy de Python.

- Al establecer una comparativa visual entre el método de ray tracing original y cada uno de los métodos de Montecarlo presentados, se ha conseguido una reducción clara de las autointersecciones. Por otro lado, los métodos de Montecarlo también pueden dar problemas con los reflejos si el rango de valores aleatorios es excesivamente grande, aunque en las pruebas presentadas el resultado final es favorable al nuevo algoritmo.

- El coste computacional de los métodos de Montecarlo es, lógicamente, mayor que en

el método original. Sin embargo, el uso de valores aleatorios los dota de una mayor adaptabilidad a la escena, lo que evita las dificultades de buscar un *offset* fijo que se adapte bien a todo el conjunto de objetos presentes en la imagen final.

5.1. Líneas de trabajo futuro

Los dos métodos presentados en este trabajo son solamente el primer paso en la aplicación de los métodos de Montecarlo en la eliminación de autointersecciones. Por ello, se abren posibles nuevas vías de investigación siguiendo estas mismas ideas:

- En este trabajo se ha hecho una implementación del algoritmo de ray tracing que fuese asumible en el tiempo previsto. Por lo tanto, aunque en este caso se han conseguido buenos resultados, es necesario comprobar el funcionamiento de estos métodos en entornos con una implementación más compleja del ray tracing.
- En relación con el punto anterior, es especialmente interesante la implementación de estos métodos utilizando aceleración por GPU, aunque no sean viables para su uso en tiempo real debido al alto coste computacional.
- Se ha intentado que los métodos de Montecarlo afecten lo mínimo posible a aquellos puntos donde no hay ninguna autointersección, pero en ciertas situaciones hemos encontrado problemas similares con los reflejos a los que se presentarían con el método usual y un *offset* demasiado grande. Se podría buscar alguna forma de reducir el efecto de estos métodos sobre los reflejos de la escena.

Bibliografía

[1] Aflak, O. (2020). Ray Tracing From Scratch in Python. En *The Startup - Medium*. Recuperado el 2 de abril de 2021 de: https://medium.com/swlh/ray-tracing-from-scratch-in-python-41670e6a96f9

[2] Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M. y Hillaire, S. (2018). Real-Time Ray Tracing. *Real-Time Rendering* (4ª ed.). CRC Press.

[3] Appel, A. (1968). *Some Techniques for Shading Machine Renderings of Solids.* https://doi.org/10.1145/1468075.1468082

[4] Dammertz, H. y Keller, A. (2006). Improving Ray Tracing Precision by Object Space Intersection Computation. *IEEE Symposium on Interactive Ray Tracing,* 25-31. https://doi.org/10.1109/RT.2006.280211

[5] Freniere, R. y Tourtellott, J. (1997). Brief history of generalized ray tracing. *Lens Design, Illumination, and Optomechanical Modeling*, 170-178. https://doi.org/10.1117/12.284059

[6] Goral, C. M., Torrance, K. E., Greenberg D. P. y Battaile, B. (1984). Modeling the interaction of Light Between Diffuse Surfaces. *Computer Graphics (SIGGRAPH '84 Proceedings), 18*(3), 213-222. https://doi.org/10.1145/964965.808601

[7] Hanika, J. y Keller, A. (2007). Towards Hardware Ray Tracing using Fixed Point Arithmetic. *IEEE Symposium on Interactive Ray Tracing*, 119-128. https://doi.org/10.1109/RT.2007.4342599

[8] Heinly, J., Recker, S., Bensema, K., Porch, J. y Gribble, C. (2009). Integer Ray Tracing. *Journal of Graphics, GPU, and Game Tools, 14*(4), 31-56. https://doi.org/10.1080/2151237X.2009.10129289

[9] Hofmann, G.R. (1990). Who invented ray tracing?. *The Visual Computer, 6*(3), 120-124. https://doi.org/10.1007/BF01911003

[10] Institute of Electrical and Electronics Engineers. (2008). IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 1-70. https://doi.org/10.1109/IEEESTD.2008.4610935

[11] Kim, D., Nah, JH. y Park, WC. (2016). Geometry transition method to improve ray-tracing precision. *Multimed Tools Appl, 75*(10), 5689-5700. https://doi.org/10.1007/s11042-015-2534-4

[12] Moncada, L. F., Barrios, R. A. y Montes, J. (S. f.). *Estructura y funcionamiento de un algoritmo de renderizado: Raytracing*. Recuperado el 28 de diciembre de 2020 de http://repositorio.utp.edu.co/dspace/bitstream/handle/11059/5385/0066M779%20_Anexo.pdf?sequence=2&isAllowed=y

[13] Pharr, M., Jakob, W. y Humphreys, G. (2017). *Physically Based Rendering: From Theory to Implementation* (3ª ed.). Elsevier.

[14] Phong, B.T. (1975). Illumination for computer generated pictures. *Communications of the ACM, 18*, 311-317. Recuperado el 30 de mayo de 2021 de https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf

[15] Reshetov, A., Soupikov, A. y William, M. (2010). Consistent Normal Interpolation. *ACM Transactions on Graphics, 29*(6), Article 142. https://doi.org/10.1145/1882261.1866168

[16] Rueckert, D. (2002). *Lecture 11 and 12: Ray tracing*. Recuperado el 5 de abril de 2021 de https://www.doc.ic.ac.uk/~dfg/graphics/graphics2008/GraphicsLecture09.pdf

[17] Salomon, D. (2011). *The Computer Graphics Manual*. Springer-Verlag London. https://doi.org/10.1007/978-0-85729-886-7

[18] Scratchapixel. (s. f.) *Ray-Triangle Intersection: Geometric Solution*. Scratchapixel 2.0. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution

[19] Wächter, C. y Binder, N. (2019). A Fast and Robust Method for Avoiding Self-Intersection. En Haines, E. y Akenine-Möller, T. (Eds.). *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (pp. 77-85). Springer Nature. https://doi.org/10.1007/978-1-4842-4427-2

Apéndice A

Ray tracer de Aflak (2020)

```
import numpy as np
import matplotlib.pyplot as plt

def normalize(vector):
    return vector / np.linalg.norm(vector)

def reflected(vector, axis):
    return vector - 2 * np.dot(vector, axis) * axis

def sphere_intersect(center, radius, ray_origin, ray_direction):
    b = 2 * np.dot(ray_direction, ray_origin - center)
    c = np.linalg.norm(ray_origin - center) ** 2 - radius ** 2
    delta = b ** 2 - 4 * c
    if delta > 0:
        t1 = (-b + np.sqrt(delta)) / 2
        t2 = (-b - np.sqrt(delta)) / 2
        if t1 > 0 and t2 > 0:
            return min(t1, t2)
    return None

def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = [sphere_intersect(obj['center'], obj['radius'],
       ray_origin, ray_direction) for obj in objects]
    nearest_object = None
    min_distance = np.inf
```

```
25      for index, distance in enumerate(distances):
26          if distance and distance < min_distance:
27              min_distance = distance
28              nearest_object = objects[index]
29      return nearest_object, min_distance
30
31  width = 300
32  height = 200
33
34  max_depth = 3
35
36  camera = np.array([0, 0, 1])
37  ratio = float(width) / height
38  screen = (-1, 1 / ratio, 1, -1 / ratio) # left, top, right, bottom
39
40  light = { 'position': np.array([5, 5, 5]), 'ambient': np.array([1,
        1, 1]), 'diffuse': np.array([1, 1, 1]), 'specular': np.array([1,
         1, 1]) }
41
42  objects = [
43      { 'center': np.array([-0.2, 0, -1]), 'radius': 0.7, 'ambient':
            np.array([0.1, 0, 0]), 'diffuse': np.array([0.7, 0, 0]), '
            specular': np.array([1, 1, 1]), 'shininess': 100, '
            reflection': 0.5 },
44      { 'center': np.array([0.1, -0.3, 0]), 'radius': 0.1, 'ambient':
             np.array([0.1, 0, 0.1]), 'diffuse': np.array([0.7, 0, 0.7])
            , 'specular': np.array([1, 1, 1]), 'shininess': 100, '
            reflection': 0.5 },
45      { 'center': np.array([-0.3, 0, 0]), 'radius': 0.15, 'ambient':
            np.array([0, 0.1, 0]), 'diffuse': np.array([0, 0.6, 0]), '
            specular': np.array([1, 1, 1]), 'shininess': 100, '
            reflection': 0.5 },
46      { 'center': np.array([0, -9000, 0]), 'radius': 9000 - 0.7, '
            ambient': np.array([0.1, 0.1, 0.1]), 'diffuse': np.array
            ([0.6, 0.6, 0.6]), 'specular': np.array([1, 1, 1]), '
            shininess': 100, 'reflection': 0.5 }
47  ]
```

```
48
49  image = np.zeros((height, width, 3))
50  for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
51      for j, x in enumerate(np.linspace(screen[0], screen[2], width))
            :
52          # screen is on origin
53          pixel = np.array([x, y, 0])
54          origin = camera
55          direction = normalize(pixel - origin)
56
57          color = np.zeros((3))
58          reflection = 1
59
60          for k in range(max_depth):
61              # check for intersections
62              nearest_object, min_distance =
                    nearest_intersected_object(objects, origin,
                    direction)
63              if nearest_object is None:
64                  break
65
66              intersection = origin + min_distance * direction
67              normal_to_surface = normalize(intersection -
                    nearest_object['center'])
68              shifted_point = intersection + 1e-5 * normal_to_surface
69              intersection_to_light = normalize(light['position'] -
                    shifted_point)
70
71              _, min_distance = nearest_intersected_object(objects,
                    shifted_point, intersection_to_light)
72              intersection_to_light_distance = np.linalg.norm(light['
                    position'] - intersection)
73              is_shadowed = min_distance <
                    intersection_to_light_distance
74
75              if is_shadowed:
76                  break
```

```
77
78              illumination = np.zeros((3))
79
80              # ambiant
81              illumination += nearest_object['ambient'] * light['
                    ambient']
82
83              # diffuse
84              illumination += nearest_object['diffuse'] * light['
                    diffuse'] * np.dot(intersection_to_light,
                    normal_to_surface)
85
86              # specular
87              intersection_to_camera = normalize(camera -
                    intersection)
88              H = normalize(intersection_to_light +
                    intersection_to_camera)
89              illumination += nearest_object['specular'] * light['
                    specular'] * np.dot(normal_to_surface, H) ** (
                    nearest_object['shininess'] / 4)
90
91              # reflection
92              color += reflection * illumination
93              reflection *= nearest_object['reflection']
94
95              origin = shifted_point
96              direction = reflected(direction, normal_to_surface)
97
98          image[i, j] = np.clip(color, 0, 1)
99      print("%d/%d" % (i + 1, height))
100
101 plt.imsave('image.png', image)
```

Apéndice B

Código completo del ray tracer

```
import numpy as np
import math
import matplotlib.pyplot as plt
from Sphere import *
from Light import *
from Plane import *
from Triangle import *
from Quad import *
import time

start_time = time.time()

def normalize(vector):
    return vector / np.linalg.norm(vector)

def reflected(vector, axis):
    return vector - 2 * np.dot(vector, axis) * axis

def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = [o.intersect(ray_origin, ray_direction) for o in
        objects]
    nearest_object = None
    min_distance = np.inf
    for index, distance in enumerate(distances):
        if distance and distance < min_distance:
```

```
25                min_distance = distance
26                nearest_object = objects[index]
27        return nearest_object, min_distance
28
29    #Resolucion de imagen de salida
30    width = 1280
31    height = 720
32
33    max_depth = 3
34
35    camera = np.array([0, 0, 1])
36    ratio = float(width) / height
37    screen = (-1, 1 / ratio, 1, -1 / ratio)
38
39    #light = Light(np.array([0, 0, 1]), np.array([1, 1, 1]), np.array
          ([1, 1, 1]), np.array([1, 1, 1])) #Esferas
40    light = Light(np.array([-0.5e15, 0.5e15, 0]), np.array([1, 1, 1]),
          np.array([1, 1, 1]), np.array([1, 1, 1])) #Cornell
41
42    #Matrices de rotacion (sentido antihorario)
43    theta = math.pi / 8 #Angulo
44    giro_x = np.array([[1,0,0],[0,math.cos(theta),-math.sin(theta)],[0,
          math.sin(theta),math.cos(theta)]])
45    giro_y = np.array([[math.cos(theta),0,-math.sin(theta)],[0,1,0],[
          math.sin(theta),0,math.cos(theta)]])
46    giro_z = np.array([[math.cos(theta),-math.sin(theta),0],[math.sin(
          theta),math.cos(theta),0],[0,0,1]])
47
48    ################################################################
49    #El siguiente bloque afecta unicamente a la caja de Cornell
50    ################################################################
51    #Escalas
52    scale_box = 1e15 #Escala de la caja
53    scale = 0.3*scale_box #Escala del cubo
54    scale_oct = 0.4*scale_box #Escala del octaedro
55
56    #Traslaciones
```

```
57 loc_box = np.array([0, 0, -2*scale_box]) #Traslacion de la caja
58 loc = np.array([0.5*scale_box, -0.7*scale_box, -2.5*scale_box]) #
       Traslacion del cubo
59 loc_oct = np.array([0, 0.4*scale_box, -2.5*scale_box]) #Traslacion
       del octaedro
60 ##############################################################
61
62 objects = [
63     #Escribir aqui la descripcion matematica de la escena
64 ]
65
66 image = np.zeros((height, width, 3))
67 for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
68     for j, x in enumerate(np.linspace(screen[0], screen[2], width))
          :
69         pixel = np.array([x, y, 0])
70         origin = camera
71         direction = normalize(pixel - origin)
72
73         color = np.zeros((3))
74         reflection = 1
75
76         for k in range(max_depth):
77             nearest_object, min_distance =
                   nearest_intersected_object(objects, origin,
                   direction)
78             if nearest_object is None:
79                 break
80
81             intersection = origin + min_distance * direction
82             normal_to_surface = nearest_object.normal(intersection)
83             shifted_point = intersection + 1e-3 * normal_to_surface
84             intersection_to_light = normalize(light.position -
                   shifted_point)
85
86             _, min_distance = nearest_intersected_object(objects,
                   shifted_point, intersection_to_light)
```

```
87              intersection_to_light_distance = np.linalg.norm(light.
                    position - intersection)
88              is_shadowed = min_distance <
                    intersection_to_light_distance
89
90              ########################################################
91              #METODO DE MONTECARLO
92              iteraciones = 3 #Numero de iteraciones del metodo de
                    Montecarlo
93              nalea = 0
94
95              while is_shadowed and nalea<iteraciones:
96
97                  #aleatorios = np.random.uniform(size=iteraciones) #
                        Uniforme
98                  aleatorios = np.abs(np.random.normal(loc = 3e-1,
                        scale = 1e-1, size = iteraciones)) #Normal
99                  np.sort(aleatorios)
100
101                 #shifted_point = intersection + aleatorios[nalea] *
                        5e-1 * normal_to_surface #Uniforme
102                 shifted_point = intersection + aleatorios[nalea] *
                        normal_to_surface #Normal
103                 intersection_to_light = normalize(light.position -
                        shifted_point)
104
105                 _, min_distance = nearest_intersected_object(
                        objects, shifted_point, intersection_to_light)
106                 intersection_to_light_distance = np.linalg.norm(
                        light.position - intersection)
107
108                 is_shadowed = min_distance <
                        intersection_to_light_distance
109                 nalea+=1
110             ########################################################
111
112             if is_shadowed:
```

```
113                 break
114
115             illumination = np.zeros((3))
116
117             # ambient
118             illumination += nearest_object.ambient * light.ambient
119
120             # diffuse
121             illumination += nearest_object.diffuse * light.diffuse
                    * np.dot(intersection_to_light, normal_to_surface)
122
123             # specular
124             intersection_to_camera = normalize(camera -
                    intersection)
125             H = normalize(intersection_to_light +
                    intersection_to_camera)
126             illumination += nearest_object.specular * light.
                    specular * np.dot(normal_to_surface, H) ** (
                    nearest_object.shininess / 4)
127
128             # reflection
129             color += reflection * illumination
130             reflection *= nearest_object.reflection
131
132             origin = shifted_point
133             direction = reflected(direction, normal_to_surface)
134
135         image[i, j] = np.clip(color, 0, 1)
136     print("%d/%d" % (i + 1, height))
137
138 plt.imsave('render.png', image)
139
140 #time.sleep(1)
141
142 end_time = time.time()
143
144 print('Tiempo de ejecucion: ')
```

```
145  print(end_time - start_time)
```

Apéndice C

Caja de Cornell

```
#Cornell box

Quad(scale_box*np.array([-1, -1, -1])+loc_box, scale_box*np.array
    ([1, -1, -1])+loc_box, scale_box*np.array([1, 1, -1])+loc_box,
    scale_box*np.array([-1, 1, -1])+loc_box,np.array([0.1, 0, 0]),
    np.array([0.6, 0.6, 0.6]), np.array([0.1, 0.1, 0.1]),100,0.1), #
    Frente
Quad(scale_box*np.array([1, 1, 0])+loc_box,scale_box*np.array([-1,
    1, 0])+loc_box,scale_box*np.array([-1, 1, -1])+loc_box,scale_box
    *np.array([1, 1, -1])+loc_box,np.array([0.1, 0, 0]), np.array
    ([1, 1, 1]), np.array([1, 1, 1]),100,0.1), #Arriba
Quad(scale_box*np.array([-1, -1, 0])+loc_box,scale_box*np.array([1,
     -1, 0])+loc_box,scale_box*np.array([1, -1, -1])+loc_box,
    scale_box*np.array([-1, -1, -1])+loc_box,np.array([0.1, 0, 0]),
    np.array([1, 1, 1]), np.array([1, 1, 1]),100,0.1), #Abajo
Quad(scale_box*np.array([1, -1, 0])+loc_box,scale_box*np.array([1,
    1, 0])+loc_box,scale_box*np.array([1, 1, -1])+loc_box,scale_box*
    np.array([1, -1, -1])+loc_box,np.array([0.1, 0, 0]), np.array
    ([0, 1, 0]), np.array([1, 1, 1]),100,0.1), #Derecha
Quad(scale_box*np.array([-1, -1, 0])+loc_box,scale_box*np.array
    ([-1, -1, -1])+loc_box,scale_box*np.array([-1, 1, -1])+loc_box,
    scale_box*np.array([-1, 1, 0])+loc_box,np.array([0.1, 0, 0]), np
    .array([1, 0, 0]), np.array([1, 1, 1]),100,0.1), #Izquierda

```

```
9  Quad(np.matmul(giro_y,scale*np.array([-1, -1, 1]))+loc, np.matmul(
       giro_y,scale*np.array([1, -1, 1]))+loc, np.matmul(giro_y,scale*
       np.array([1, 1, 1]))+loc,np.matmul(giro_y,scale*np.array([-1, 1,
        1]))+loc,np.array([0.1, 0, 0]), np.array([0, 0.3, 0.8]), np.
       array([1, 1, 1]),100,0), #Frente
10 Quad(np.matmul(giro_y,scale*np.array([-1, 1, 1]))+loc,np.matmul(
       giro_y,scale*np.array([1, 1, 1]))+loc,np.matmul(giro_y,scale*np.
       array([1, 1, -1]))+loc,np.matmul(giro_y,scale*np.array([-1, 1,
       -1]))+loc,np.array([0.1, 0, 0]), np.array([0.1, 0.6, 0.9]), np.
       array([1, 1, 1]),100,0), #Arriba
11 Quad(np.matmul(giro_y,scale*np.array([1, -1, 1]))+loc,np.matmul(
       giro_y,scale*np.array([-1, -1, 1]))+loc,np.matmul(giro_y,scale*
       np.array([-1, -1, -1]))+loc,np.matmul(giro_y,scale*np.array([1,
       -1, -1]))+loc,np.array([0.1, 0, 0]), np.array([0.1, 0.6, 0.9]),
       np.array([1, 1, 1]),100,0), #Abajo
12 Quad(np.matmul(giro_y,scale*np.array([1, -1, 1]))+loc,np.matmul(
       giro_y,scale*np.array([1, -1, -1]))+loc,np.matmul(giro_y,scale*
       np.array([1, 1, -1]))+loc,np.matmul(giro_y,scale*np.array([1, 1,
        1]))+loc,np.array([0.1, 0, 0]), np.array([0.1, 0.6, 0.9]), np.
       array([1, 1, 1]),100,0), #Derecha
13 Quad(np.matmul(giro_y,scale*np.array([-1, -1, 1]))+loc,np.matmul(
       giro_y,scale*np.array([-1, 1, 1]))+loc,np.matmul(giro_y,scale*np
       .array([-1, 1, -1]))+loc,np.matmul(giro_y,scale*np.array([-1,
       -1, -1]))+loc,np.array([0.1, 0, 0]), np.array([0.1, 0.6, 0.9]),
       np.array([1, 1, 1]),100,0), #Izquierda
14
15 Sphere(np.array([-0.6e15, -0.8e15, -2.3e15]), 0.3e15, np.array
       ([0.1, 0, 0]), np.array([0, 1, 0]), np.array([1, 1, 1]), 100,
       0.8),
16
17 Triangle(np.matmul(giro_y,scale_oct*np.array([-0.5, 0, 0.5]))+
       loc_oct, np.matmul(giro_y,scale_oct*np.array([0.5, 0, 0.5]))+
       loc_oct, np.matmul(giro_y,scale_oct*np.array([0, 1, 0]))+loc_oct
       ,np.array([0.1, 0, 0]), np.array([1, 1, 0]), np.array([1, 1, 1])
       ,100,0.5),
18 Triangle(np.matmul(giro_y,scale_oct*np.array([0.5, 0, 0.5]))+
       loc_oct, np.matmul(giro_y,scale_oct*np.array([0.5, 0, -0.5]))+
```

```
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, 1, 0]))+loc_oct
    ,np.array([0.1, 0, 0]), np.array([1, 0, 0]), np.array([1, 1, 1])
    ,100,0.5),
19  Triangle(np.matmul(giro_y,scale_oct*np.array([0.5, 0, -0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([-0.5, 0, -0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, 1, 0]))+loc_oct
    ,np.array([0.1, 0, 0]), np.array([1, 1, 0]), np.array([1, 1, 1])
    ,100,0.5),
20  Triangle(np.matmul(giro_y,scale_oct*np.array([-0.5, 0, -0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([-0.5, 0, 0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, 1, 0]))+loc_oct
    ,np.array([0.1, 0, 0]), np.array([1, 0, 0]), np.array([1, 1, 1])
    ,100,0.5),
21
22  Triangle(np.matmul(giro_y,scale_oct*np.array([-0.5, 0, 0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, -1, 0]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0.5, 0, 0.5]))+
    loc_oct,np.array([0.1, 0, 0]), np.array([1, 0, 0]), np.array([1,
     1, 1]),100,0.5),
23  Triangle(np.matmul(giro_y,scale_oct*np.array([0.5, 0, 0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, -1, 0]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0.5, 0, -0.5]))+
    loc_oct,np.array([0.1, 0, 0]), np.array([1, 1, 0]), np.array([1,
     1, 1]),100,0.5),
24  Triangle(np.matmul(giro_y,scale_oct*np.array([0.5, 0, -0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, -1, 0]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([-0.5, 0, -0.5]))+
    loc_oct,np.array([0.1, 0, 0]), np.array([1, 0, 0]), np.array([1,
     1, 1]),100,0.5),
25  Triangle(np.matmul(giro_y,scale_oct*np.array([-0.5, 0, -0.5]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([0, -1, 0]))+
    loc_oct, np.matmul(giro_y,scale_oct*np.array([-0.5, 0, 0.5]))+
    loc_oct,np.array([0.1, 0, 0]), np.array([1, 1, 0]), np.array([1,
     1, 1]),100,0.5),
```

Apéndice D

Galería de imágenes

Para facilitar la comparativa visual de las imágenes renderizadas con los distintos algoritmos, se exponen en este anexo las figuras mostradas en el Capítulo 4 de la memoria.

D.1. Esferas: método uniforme

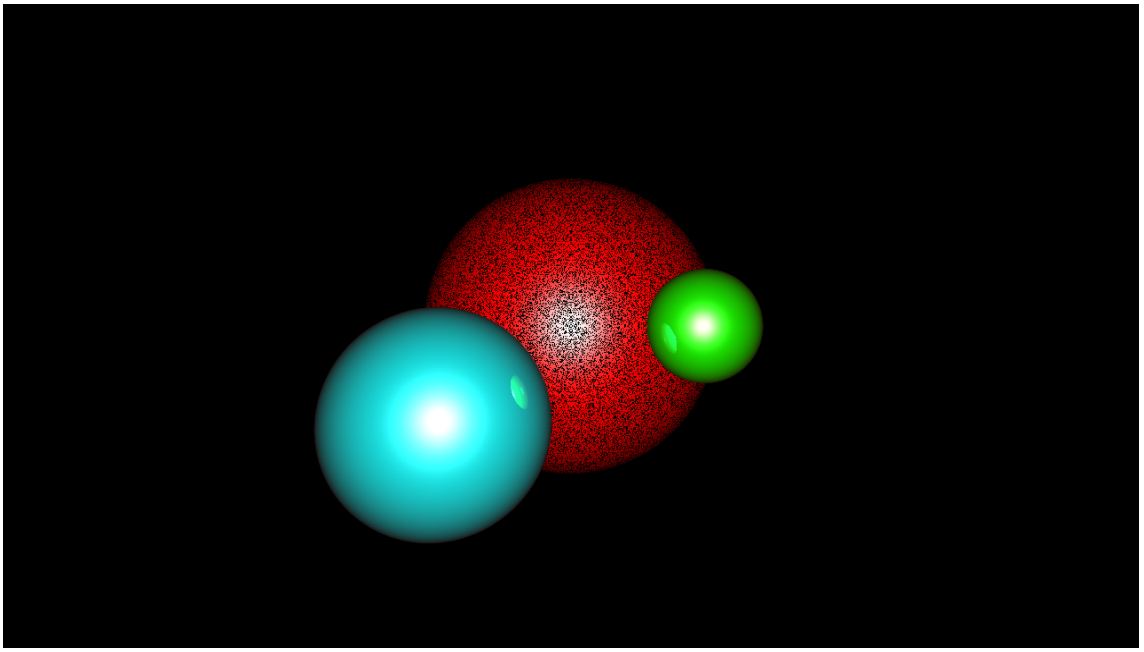

Figura D.1: Esfera distante con problemas de autointersección. Fuente: Elaboración propia.

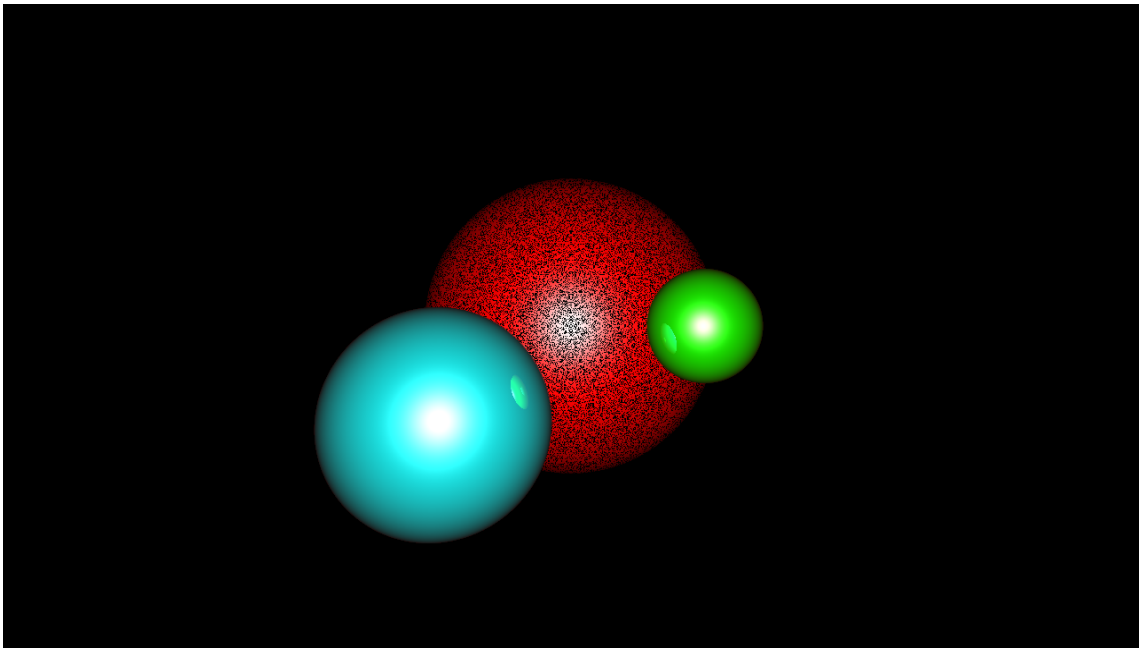

Figura D.2: Ray tracer original, $\varepsilon = 1e-5$. Fuente: Elaboración propia.

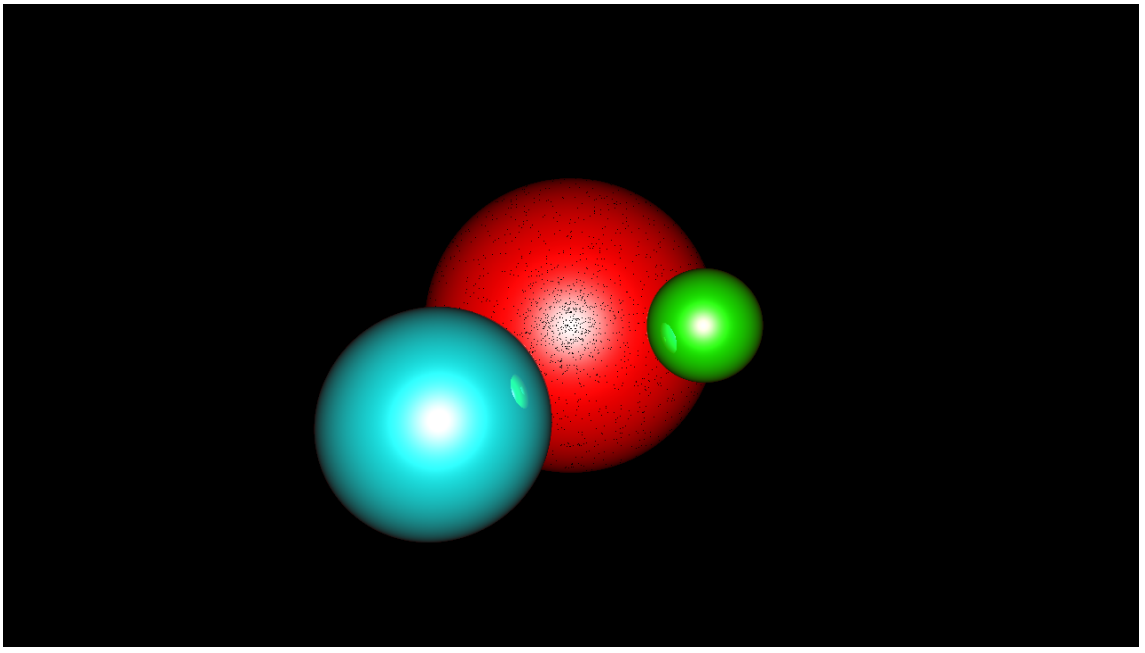

Figura D.3: Ray tracer de Montecarlo, $n = 3$, $\varepsilon_0 = 1e-5$, $\varepsilon_1 = 1e-4$. Fuente: Elaboración propia.

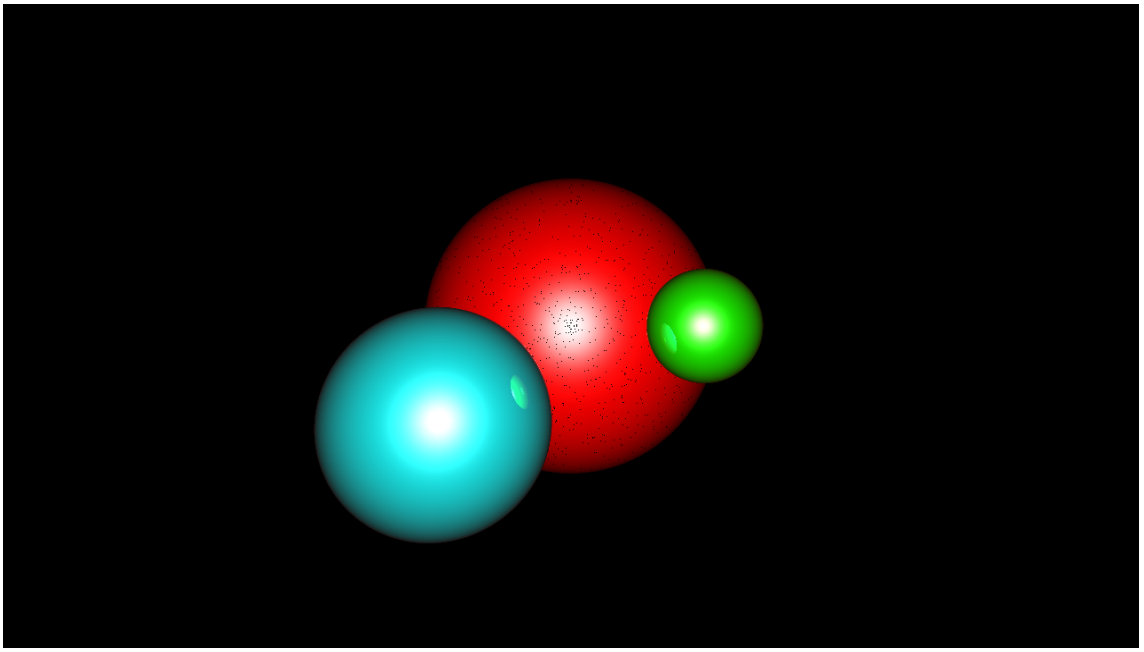

Figura D.4: Ray tracer de Montecarlo, $\varepsilon_0 = 1e-5,\ \varepsilon_1 = 1e-3,\ n = 3$. Fuente: Elaboración propia.

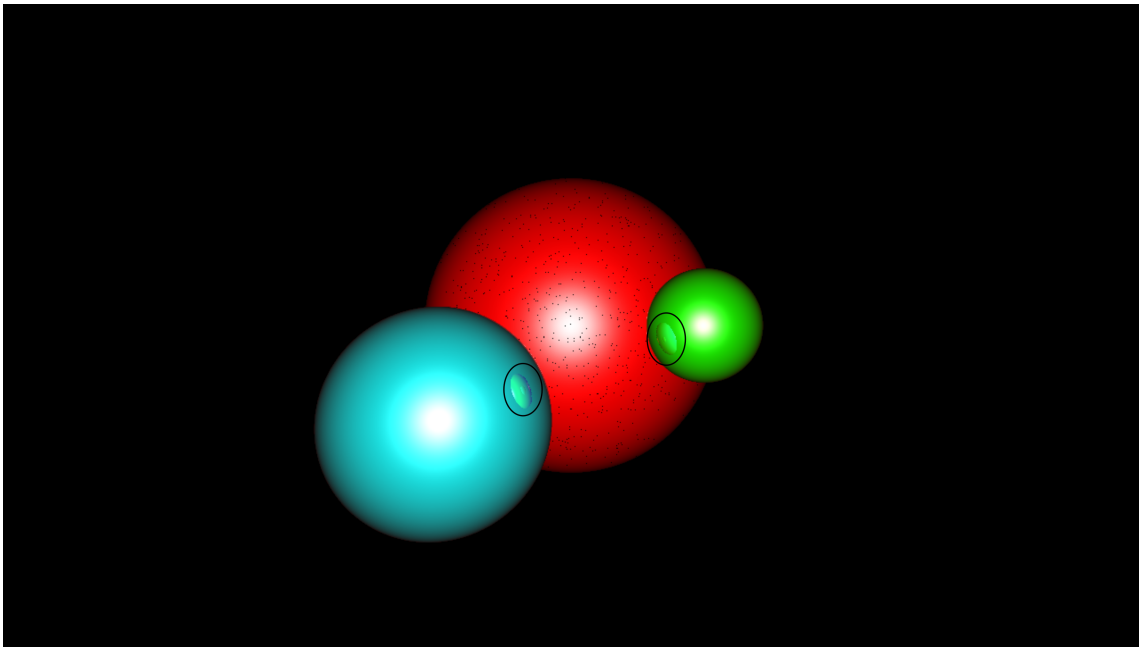

Figura D.5: Ray tracer de Montecarlo, $\varepsilon_0 = 1e-5,\ \varepsilon_1 = 1e-1,\ n = 3$. Fuente: Elaboración propia.

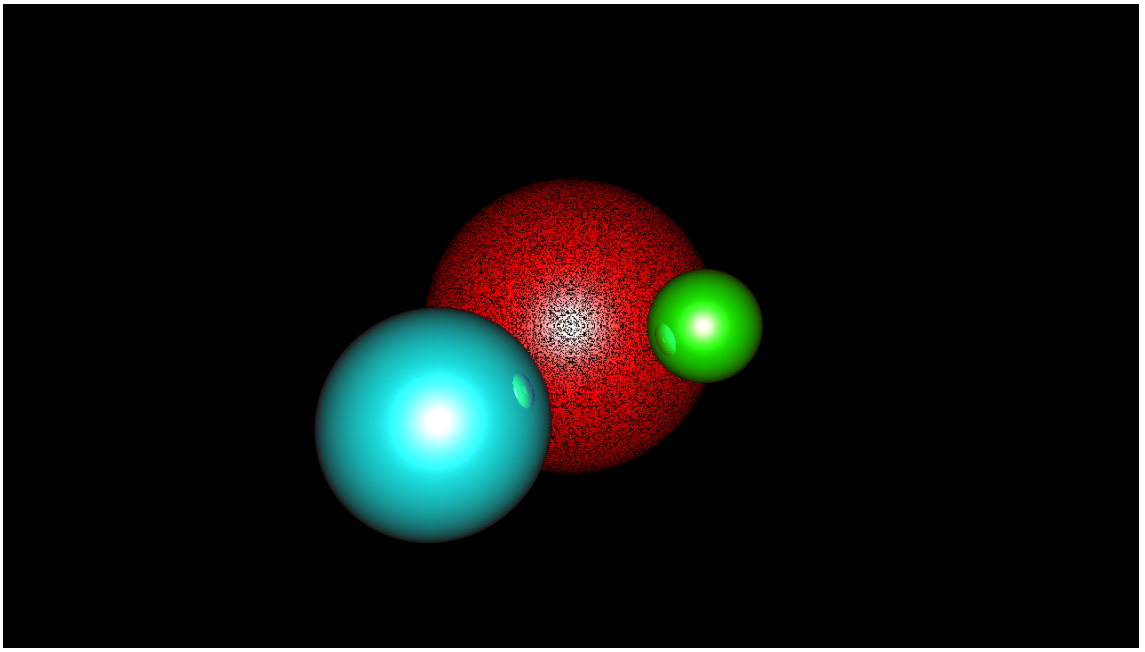

Figura D.6: Ray tracer original, $\varepsilon_0 = 1e-1$. Fuente: Elaboración propia.

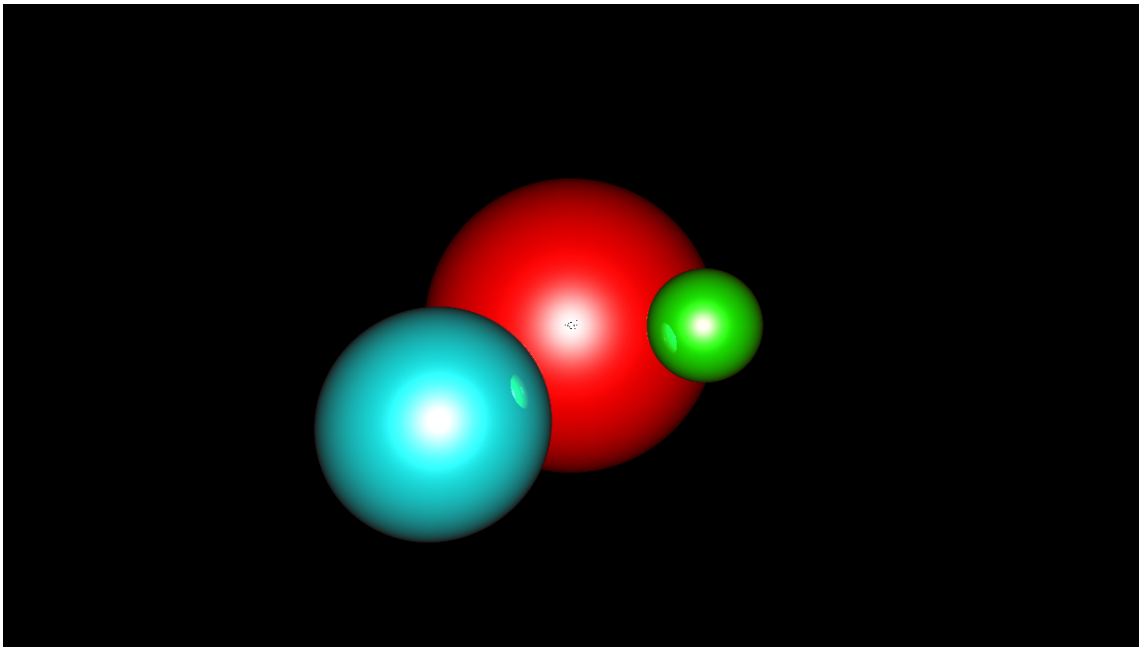

Figura D.7: Ray tracer de Montecarlo, $\varepsilon_0 = 1e{-}5,\ \varepsilon_1 = 1e{-}3,\ n = 10$. Fuente: Elaboración propia.

D.2. Caja de Cornell: método uniforme

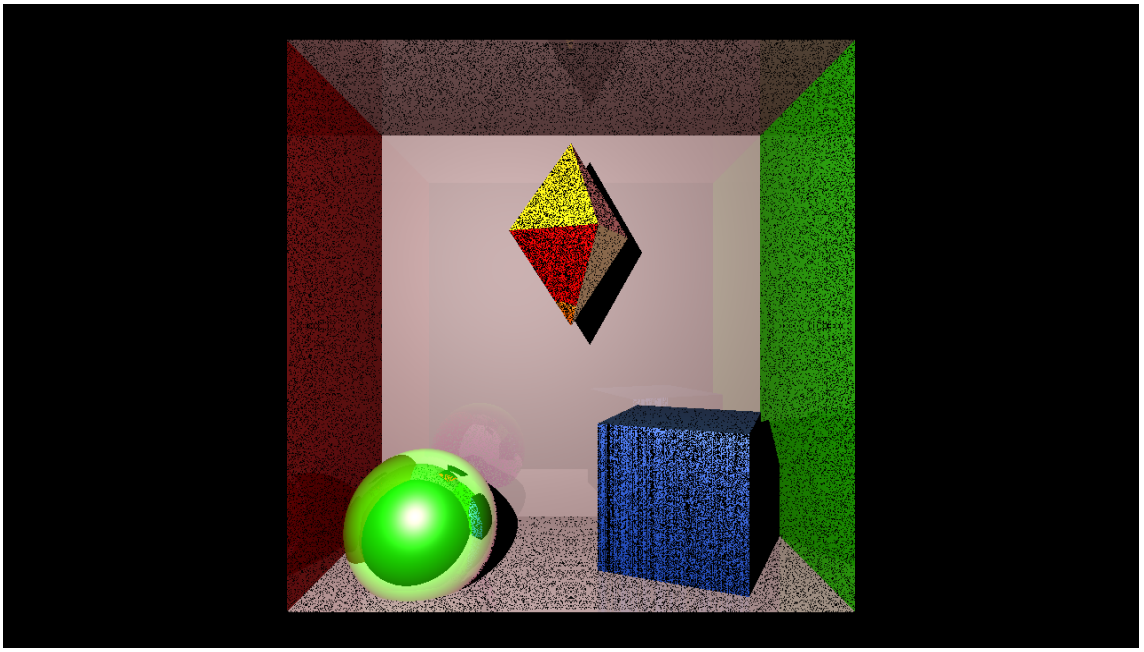

Figura D.8: Ray tracer original, $\varepsilon = 1e-3$. Fuente: Elaboración propia.

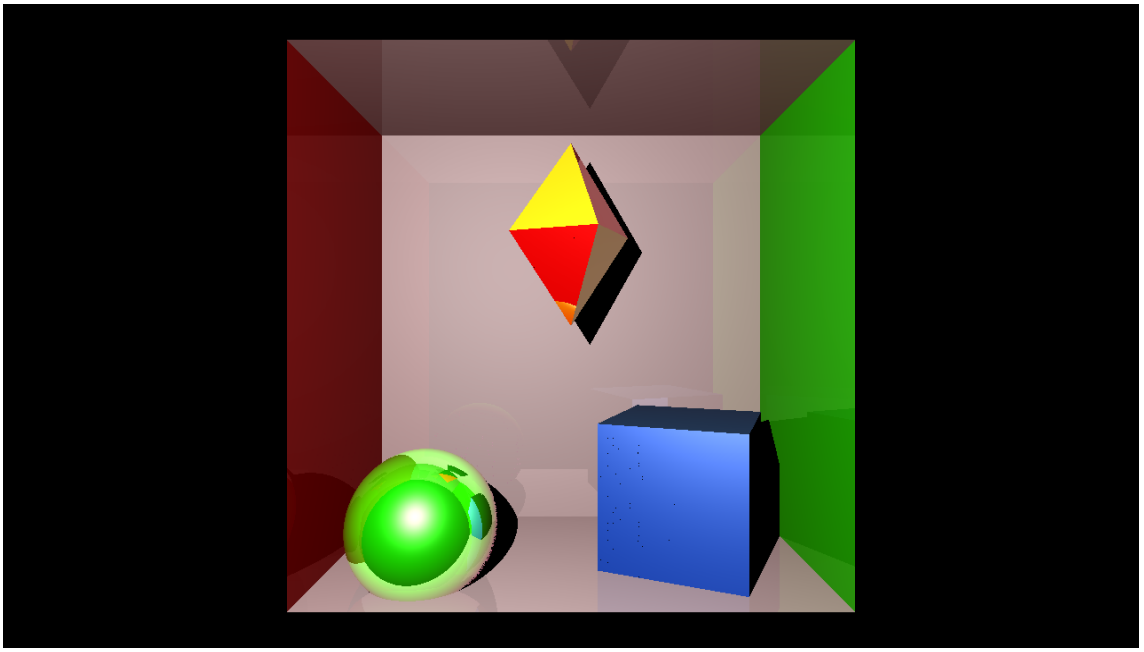

Figura D.9: Ray tracer original, $\varepsilon = 5e-1$. Fuente: Elaboración propia.

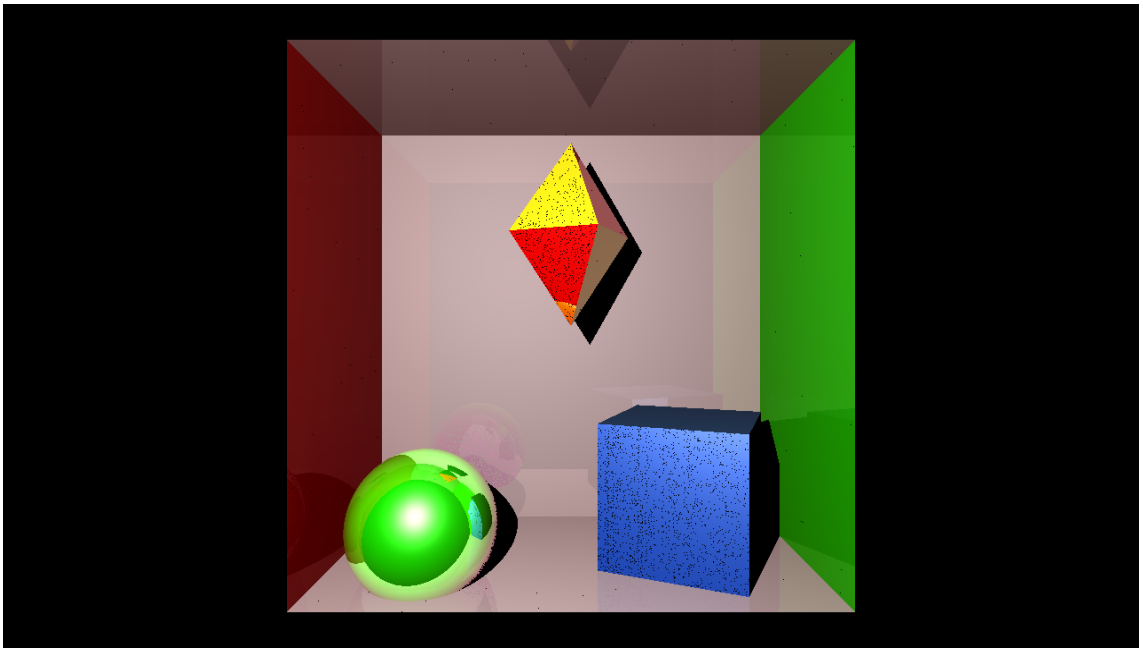

Figura D.10: Ray tracer de Montecarlo, $\varepsilon_0 = 1e{-}3,\ \varepsilon_1 = 5e{-}1,\ n = 3$. Fuente: Elaboración propia.

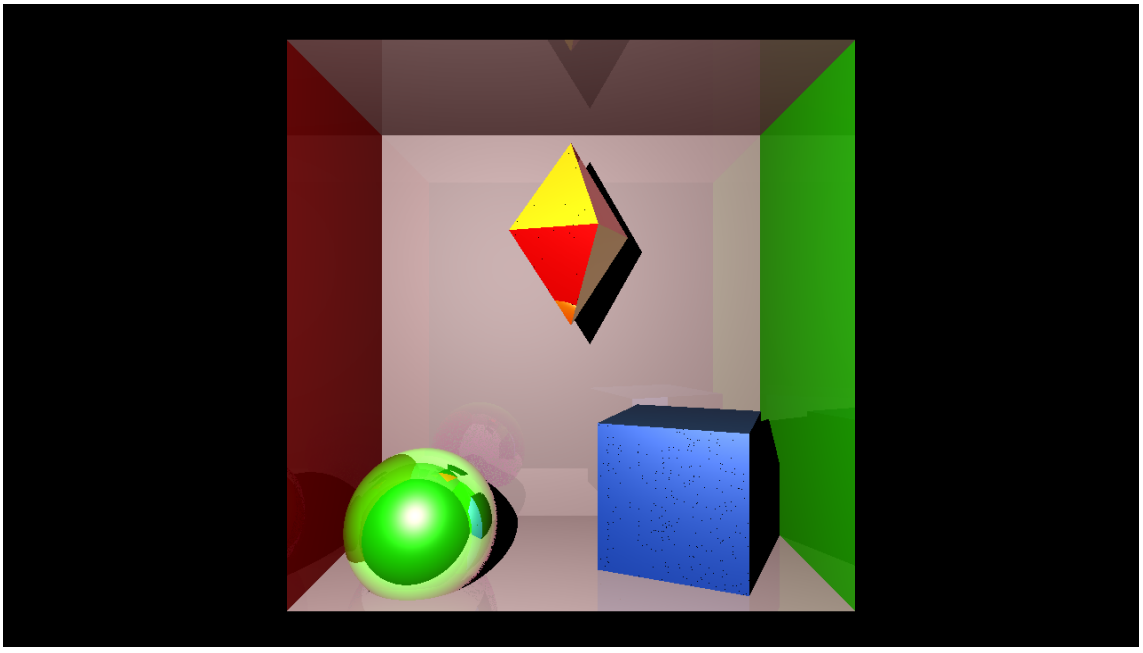

Figura D.11: Ray tracer de Montecarlo, $\varepsilon_0 = 1e-3,\ \varepsilon_1 = 5e-1,\ n = 10$. Fuente: Elaboración propia.

D.3. Esferas: método normal

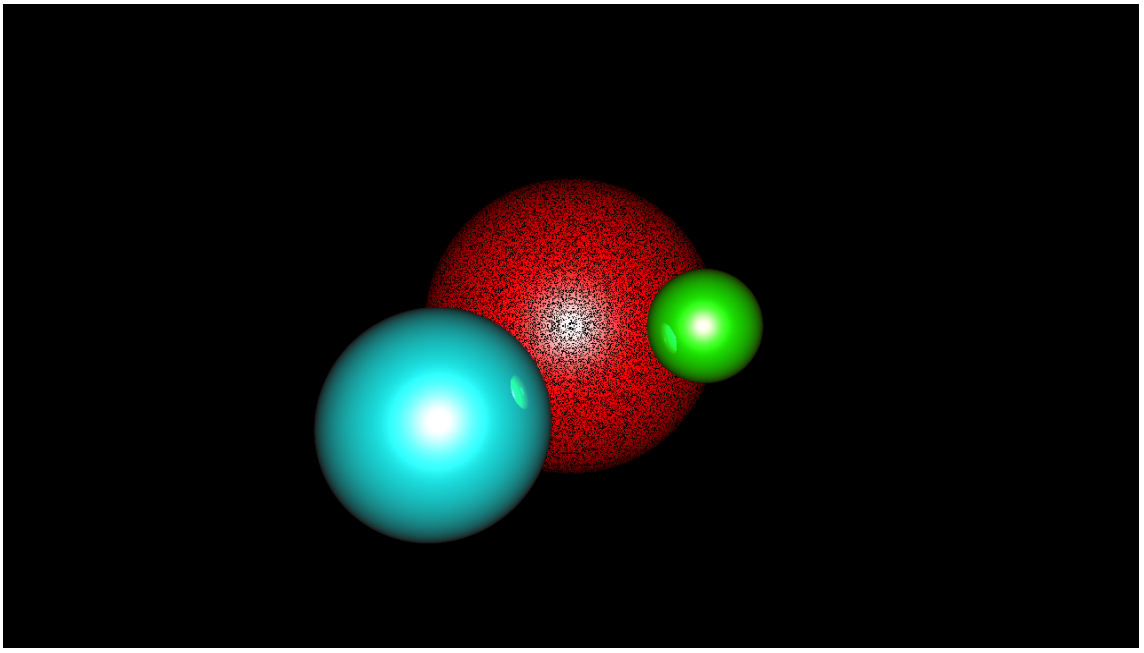

Figura D.12: Ray tracer original, $\varepsilon = 1e-3$. Fuente: Elaboración propia.

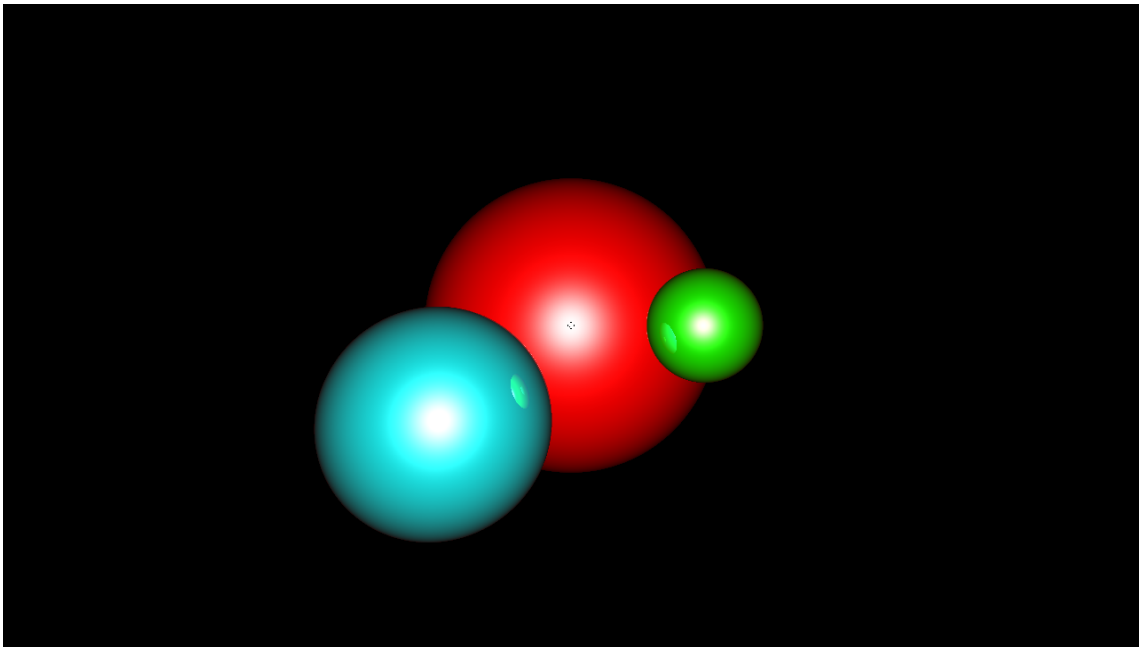

Figura D.14: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 1e-3, \sigma = 1e-3, n = 10$. Fuente: Elaboración propia.

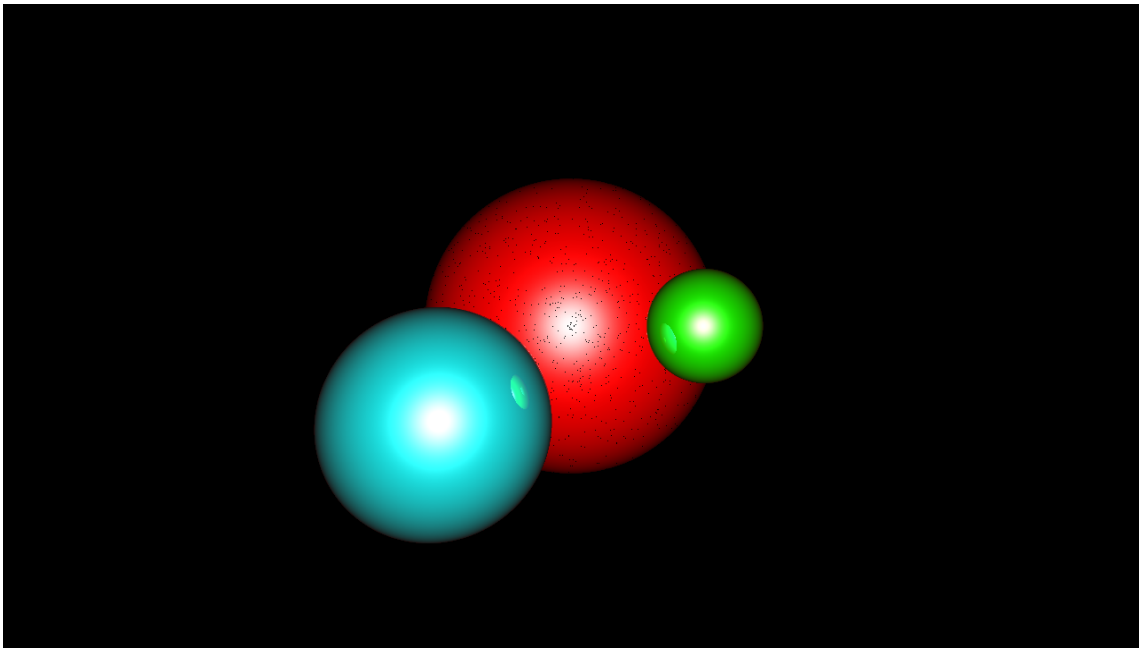

Figura D.13: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 1e-3, \sigma = 1e-3, n = 3$. Fuente: Elaboración propia.

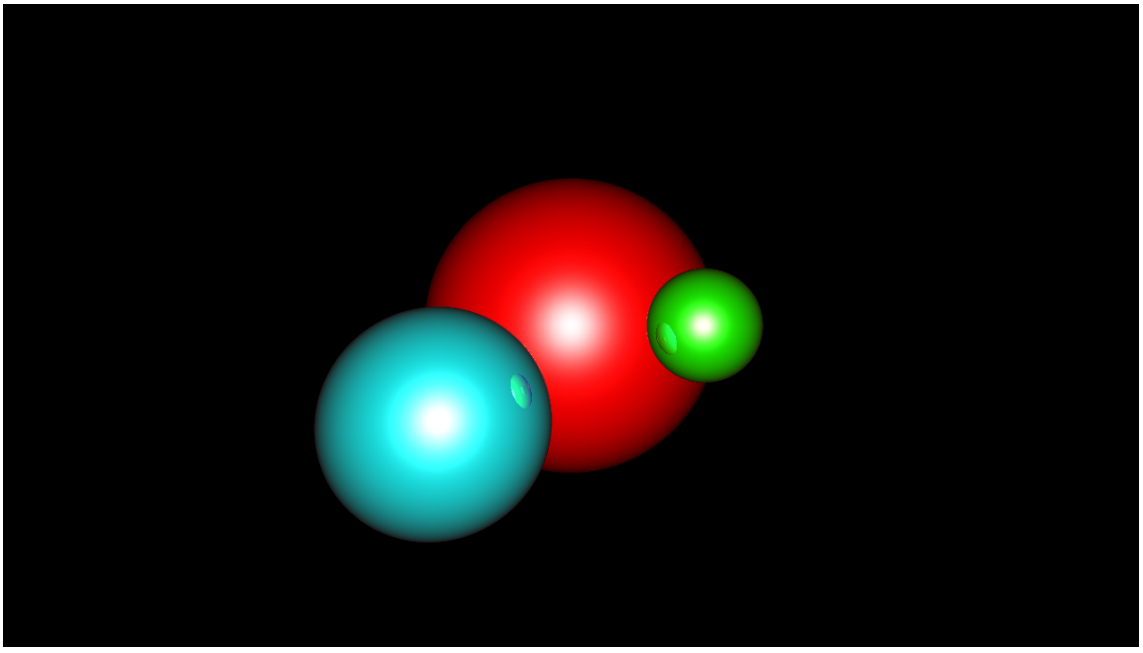

Figura D.15: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 1e-3, \sigma = 1e-1, n = 10$. Fuente: Elaboración propia.

D.4. Caja de Cornell: método normal

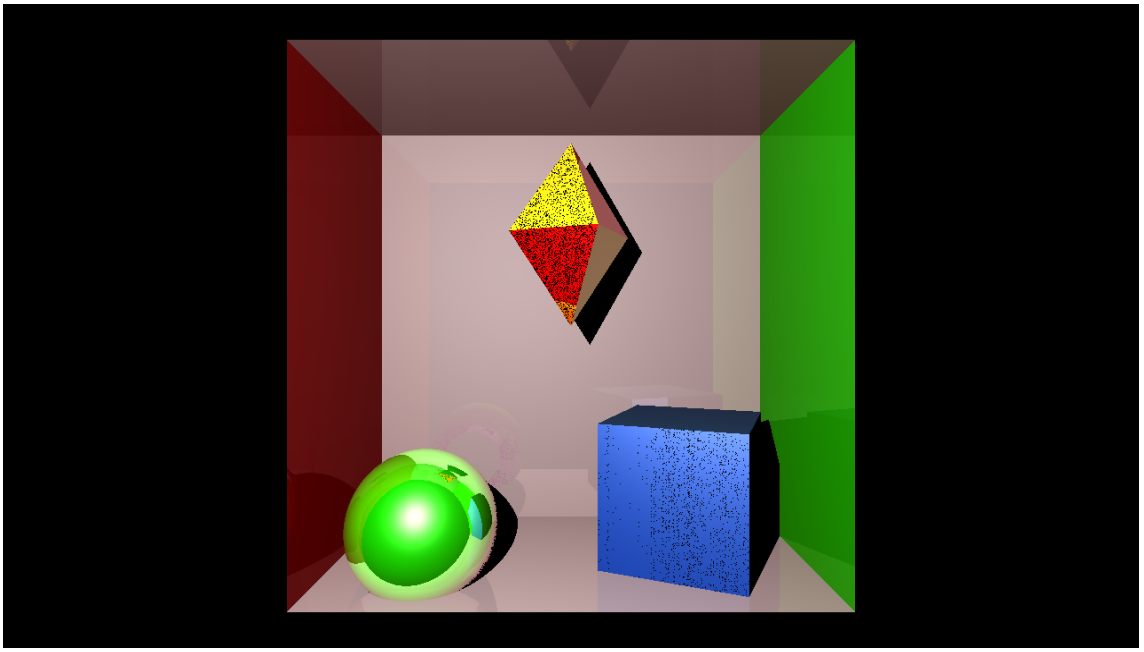

Figura D.16: Ray tracer original, $\varepsilon_0 = 3e - 1$. Fuente: Elaboración propia.

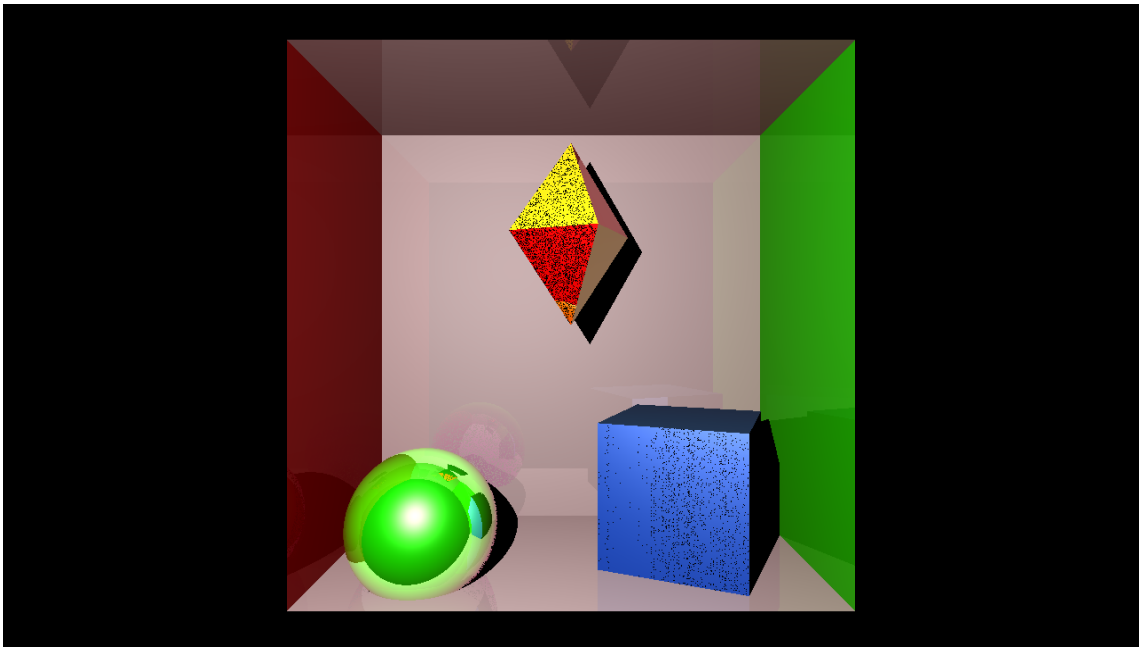

Figura D.17: Montecarlo normal, $\varepsilon_0 = 1e - 3, \mu = 3e - 1, \sigma = 1e - 3, n = 3$. Fuente: Elaboración propia.

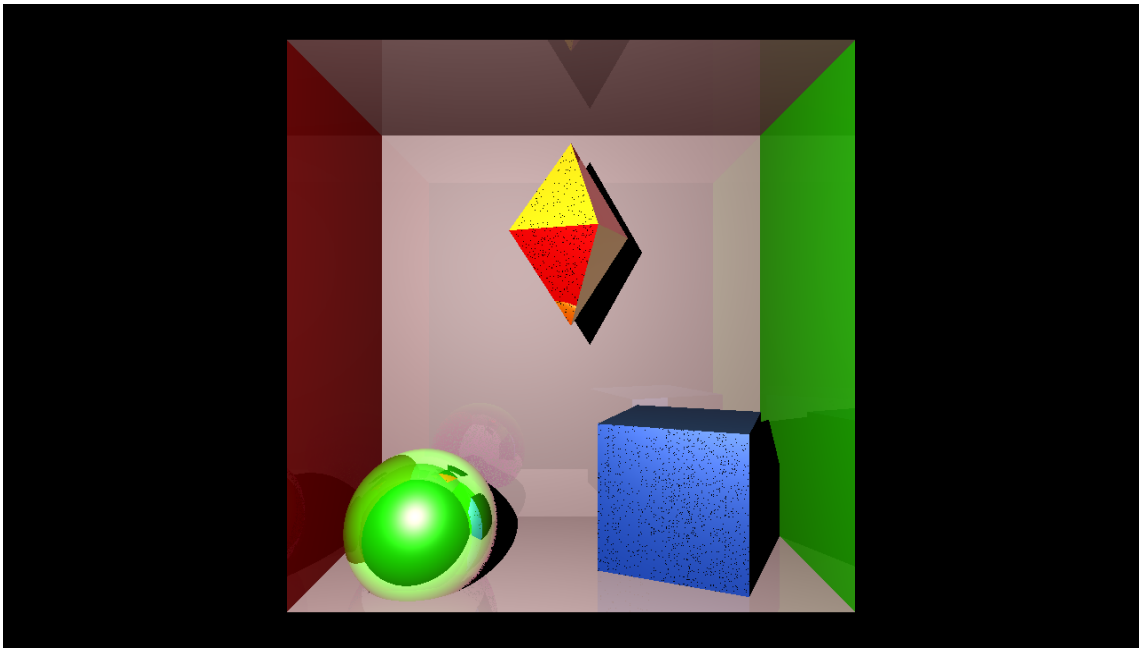

Figura D.18: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 3e-1, \sigma = 1e-1, n = 3$. Fuente: Elaboración propia.

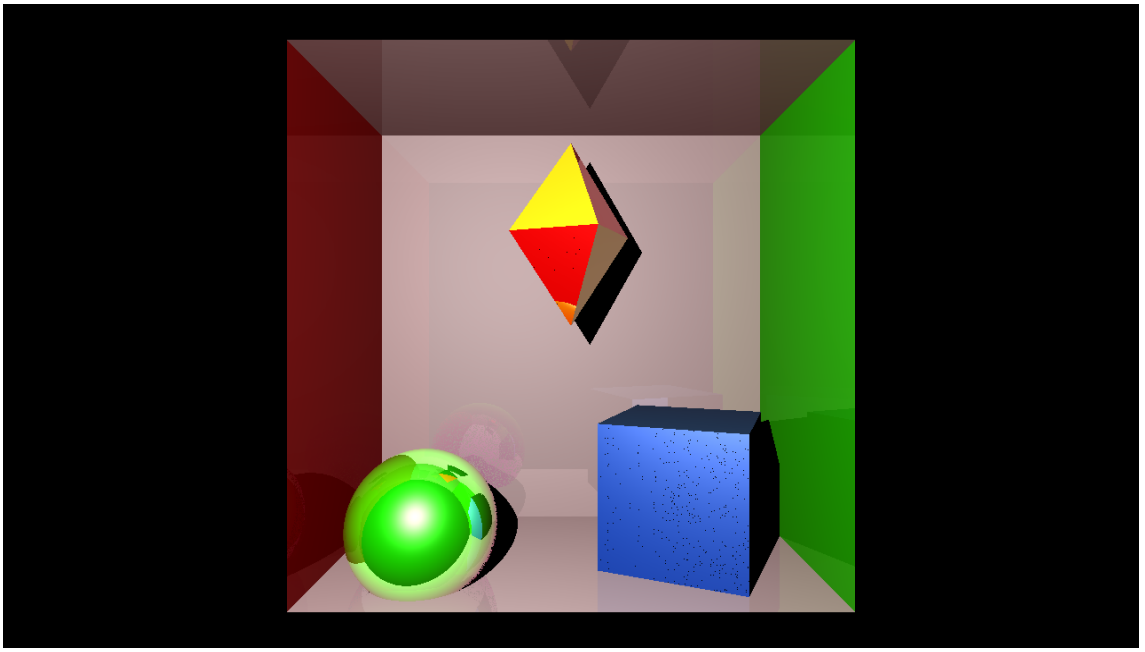

Figura D.19: Montecarlo normal, $\varepsilon_0 = 1e-3, \mu = 3e-1, \sigma = 1e-1, n = 10$. Fuente: Elaboración propia.

Apéndice E

Tiempos de renderizado

En este anexo se recogen todas las medidas del tiempo de renderizado utilizadas para la elaboración de las tablas de las secciones 4.5 y 4.6.5.

E.1. Método uniforme

Iteración	$n = 0$	$n = 50$	$n = 100$
1	16,77623724937439	17,474265098571777	17,569011449813843
2	16,773140907287598	17,512163639068604	17,811756134033203
3	16,79109239578247	17,49221706390381	17,66575264930725
4	16,708313465118408	17,483241319656372	17,732573986053467
5	16,815815925598145	17,351593255996704	17,726590394973755
6	16,849934339523315	17,529118061065674	17,575993061065674
7	16,829988479614258	17,35558247566223	17,618878602981567
8	16,6345112323761	17,17506504058838	17,924062490463257
9	16,533546924591064	17,512162923812866	17,741549968719482
10	16,69335389137268	17,530115604400635	17,75252079963684

Tabla E.1: Tiempos de renderizado (en segundos).

Iteración	$n = 150$	$n = 200$	$n = 250$
1	17,817347288131714	18,048728704452515	18,59227466583252
2	18,09556818008423	18,047731161117554	18,453645944595337
3	17,80936908721924	18,142478227615356	18,424723148345947
4	17,888158082962036	18,228248119354248	18,549389839172363
5	17,78842544555664	17,95996642112732	18,55238175392151
6	17,891149520874023	18,177384853363037	18,383832454681396
7	17,94201397895813	18,224608898162842	18,53596258163452
8	17,87048888206482	18,251708269119263	18,348925590515137
9	17,99038004875183	18,157437562942505	18,474590063095093
10	17,75750732421875	17,945006608963013	18,269139289855957

Tabla E.2: Tiempos de renderizado (en segundos).

Iteración	$n = 300$	$n = 350$	$n = 400$
1	18,681037425994873	19,11527395248413	19,286418437957764
2	18,56534719467163	18,8815016746521	19,029104232788086
3	18,806701183319092	18,859560012817383	19,252509355545044
4	18,756834506988525	18,935357570648193	19,12285614013672
5	18,615213632583618	19,075327396392822	19,226142644882202
6	18,753825664520264	18,87053155899048	19,40011477470398
7	18,704723596572876	18,93834924697876	19,261603593826294
8	18,805704593658447	18,866541862487793	19,283426523208618
9	18,60424256324768	18,808695554733276	19,262482166290283
10	18,539416074752808	18,88648772239685	19,16773557662964

Tabla E.3: Tiempos de renderizado (en segundos).

Iteración	$n = 450$	$n = 500$	$n = 550$
1	19,6434645652771	19,95862078666687	20,18102741241455
2	19,372189044952393	19,96859383583069	20,227900743484497
3	19,573650598526	19,92171883583069	20,27752685546875
4	19,69634461402893	19,826972723007202	20,255826234817505
5	19,490917444229126	20,037410497665405	20,434349298477173
6	19,62351703643799	19,90510129928589	20,22391128540039
7	19,453970432281494	19,96260643005371	20,085282802581787
8	19,588610410690308	19,827970504760742	20,190999269485474
9	19,564674854278564	19,899778127670288	20,48222041130066
10	19,68136239051819	19,843926906585693	20,03242325782776

Tabla E.4: Tiempos de renderizado (en segundos).

Iteración	$n = 600$	$n = 650$	$n = 700$
1	20,44232702255249	21,147441387176514	21,395777225494385
2	20,587937593460083	20,919052362442017	21,298038244247437
3	20,751500129699707	20,980887174606323	21,365857124328613
4	20,507153511047363	20,90209722518921	21,398768663406372
5	20,591591835021973	20,98886513710022	21,44364833831787
6	20,598907947540283	20,807350635528564	21,291056871414185
7	20,641793727874756	21,031750679016113	21,446640968322754
8	20,5869402885437	20,988865852355957	21,375831127166748
9	20,45429515838623	21,09358501434326	21,308011531829834
10	20,487207174301147	20,965378046035767	21,3229718208313

Tabla E.5: Tiempos de renderizado (en segundos).

Iteración	$n = 750$	$n = 800$	$n = 850$
1	21,745840311050415	22,351221561431885	22,598560333251953
2	21,693979740142822	22,089920043945312	22,606539726257324
3	21,76080083847046	22,096901893615723	22,56265664100647
4	21,41572403907776	22,153749704360962	22,50246787071228
5	21,578288555145264	22,157739400863647	22,528747081756592
6	21,550938606262207	21,83161187171936	22,500821352005005
7	21,775760889053345	21,897435903549194	22,53971791267395
8	21,58726477622986	21,85355257987976	22,60753607749939
9	21,723899126052856	21,86950969696045	22,4184889793396
10	21,734869956970215	22,071969270706177	22,41106152534485

Tabla E.6: Tiempos de renderizado (en segundos).

Iteración	$n = 900$	$n = 950$	$n = 1000$
1	23,00048518180847	23,541040182113647	23,953934907913208
2	22,957599878311157	23,46125292778015	24,153401613235474
3	23,078277587890625	23,371492862701416	23,839242219924927
4	23,151081800460815	23,629802227020264	23,933988571166992
5	22,87382411956787	23,446292638778687	23,85177707672119
6	22,974554777145386	23,36850070953369	24,039705753326416
7	22,904741525650024	23,571956634521484	23,861182928085327
8	23,06331706047058	23,63379144668579	23,826276540756226
9	23,014447689056396	23,72554612159729	24,129465579986572
10	22,9236900806427	23,56397795677185	23,885119676589966

Tabla E.7: Tiempos de renderizado (en segundos).

E.2. Método normal

Iteración	$n = 0$	$n = 50$	$n = 100$
1	16,50923180580139	17,493027925491333	17,555434226989746
2	16,562080144882202	17,465834617614746	17,57421851158142
3	16,707451820373535	17,588117122650146	17,601198434829712
4	16,67275595664978	17,458885669708252	17,576486110687256
5	16,458271741867065	17,43067169189453	17,81542158126831
6	16,495969772338867	17,50799798965454	17,65951943397522
7	16,5990092754364	17,485477447509766	17,637592792510986
8	16,741737365722656	17,388522148132324	17,55121088027954
9	16,601152181625366	17,42114806175232	17,74973440170288
10	16,74018359184265	17,351613998413086	17,64729404449463

Tabla E.8: Tiempos de renderizado (en segundos).

Iteración	$n = 150$	$n = 200$	$n = 250$
1	17,847521781921387	17,998953104019165	18,369504928588867
2	17,92532467842102	18,075175523757935	18,147969722747803
3	17,870181798934937	18,313628673553467	18,459803581237793
4	17,921871662139893	18,04425811767578	18,41882634162903
5	17,951473712921143	18,07586431503296	18,112385511398315
6	17,824650287628174	17,972469568252563	18,26125168800354
7	17,88867712020874	17,97207522392273	18,390814304351807
8	17,896540880203247	17,847248554229736	18,35135555267334
9	17,815160036087036	18,12432861328125	18,414843320846558
10	17,83905029296875	18,05140733718872	18,455746173858643

Tabla E.9: Tiempos de renderizado (en segundos).

Iteración	$n = 300$	$n = 350$	$n = 400$
1	18,679649114608765	18,784860134124756	19,156144857406616
2	18,429094552993774	18,83174777030945	19,259053707122803
3	18,553739070892334	18,854886770248413	19,099870443344116
4	18,621689319610596	18,86286449432373	19,112087726593018
5	18,79169487953186	18,840571641921997	19,029861211776733
6	18,46501326560974	18,93306064605713	19,011215209960938
7	18,57949447631836	18,98114275932312	19,05260944366455
8	18,603606939315796	18,8231418132782	19,068479776382446
9	18,653373956680298	18,75729727745056	19,073365211486816
10	18,50030517578125	18,81797504425049	19,100039958953857

Tabla E.10: Tiempos de renderizado (en segundos).

Iteración	$n = 450$	$n = 500$	$n = 550$
1	19,429999828338623	20,055594444274902	20,249037981033325
2	19,61836552619934	19,828798532485962	20,310122966766357
3	19,229952573776245	19,91624116897583	20,038781881332397
4	19,406652212142944	19,830158233642578	20,28284525871277
5	19,47579550743103	19,96311378479004	20,191973209381104
6	19,516971111297607	19,885153770446777	20,3458149433136
7	19,489699840545654	20,037892818450928	20,204818964004517
8	19,499624490737915	19,94589877128601	20,20661425590515
9	19,50321054458618	19,94178009033203	20,184784173965454
10	19,434426069259644	19,99213480949402	20,081167697906494

Tabla E.11: Tiempos de renderizado (en segundos).

Iteración	$n = 600$	$n = 650$	$n = 700$
1	20,669715642929077	20,96127986907959	21,439467430114746
2	20,539067268371582	21,06881856918335	21,188143014907837
3	20,519563674926758	20,804662942886353	21,30600333213806
4	20,460076570510864	20,847116947174072	21,11985731124878
5	20,52507495880127	20,951026678085327	21,60662007331848
6	20,3467013835907	21,008766651153564	21,1743803024292
7	20,4620680809021	21,000873565673828	21,29669165611267
8	20,45154619216919	21,142889976501465	21,251808166503906
9	20,641844272613525	21,274324655532837	21,195695161819458
10	20,596465349197388	20,894703149795532	21,33104705810547

Tabla E.12: Tiempos de renderizado (en segundos).

Iteración	$n = 750$	$n = 800$	$n = 850$
1	21,717872858047485	22,189769983291626	22,28766441345215
2	21,660438299179077	22,209312438964844	22,627811431884766
3	21,794246673583984	22,140074491500854	22,420527696609497
4	21,78404211997986	22,032909154891968	22,39063024520874
5	21,651448726654053	22,208087921142578	22,467212200164795
6	21,869796752929688	22,091781854629517	22,71085262298584
7	21,710817575454712	22,11737895011902	22,65367865562439
8	21,64296269416809	22,252809762954712	22,47502589225769
9	21,748051404953003	22,22166609764099	22,58003544807434
10	21,531534433364868	22,16138768196106	22,329880714416504

Tabla E.13: Tiempos de renderizado (en segundos).

Iteración	$n = 900$	$n = 950$	$n = 1000$
1	22,975322484970093	23,440964698791504	24,030784130096436
2	22,944627285003662	23,427793502807617	23,954249382019043
3	22,97809147834778	23,679044723510742	23,971707105636597
4	22,980222940444946	23,586455821990967	24,094969034194946
5	23,093404054641724	23,464227199554443	23,9585599899292
6	22,989298820495605	23,300788402557373	24,074932098388672
7	23,161762237548828	23,47042465209961	24,034778594970703
8	22,971872806549072	23,428890466690063	23,915745973587036
9	22,953914880752563	23,34506893157959	24,037914991378784
10	23,07246208190918	23,496679306030273	23,998448848724365

Tabla E.14: Tiempos de renderizado (en segundos).