UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PLATAFORMA DE DEPURADO PARA RENDERIZADORES BASADOS EN RAYTRACING

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

STEFANO GIOIA CHEHADE

PROFESOR GUÍA:
ALEJANDRO ECHEVERRIA GÁLVEZ

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER
GONZALO NAVARRO BADINO

SANTIAGO DE CHILE
2019

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: STEFANO GIOIA CHEHADE
FECHA: MARZO 2019
PROF. GUÍA: ALEJANDRO ECHEVERRIA GÁLVEZ

PLATAFORMA DE DEPURADO PARA RENDERIZADORES BASADOS EN RAYTRACING

Encontrar un error en una aplicación gráfica mediante las herramientas de depurado que proveen los lenguajes de programación convencionales suele no ser una tarea fácil. En este trabajo se propone un método y se implementa una solución para llevar a cabo el depurado de aplicaciones que utilizan *raytracing* como técnica de renderizado.

En concreto, se describe el diseño y construcción de una **plataforma web** que permite la carga, manipulación y visualización de rayos a través de un esquema particular, que consta de propiedades que describen rayos como típicamente se encuentran en los raytracers, como el origen, dirección, y punto de término, en el caso de rayos finitos.

Se propone un sistema de etiquetado, que consiste en asignarle etiquetas a los rayos durante el proceso de generación del registro, y se muestra cómo se pueden realizar consultas del tipo *"¿cuáles son los rayos que fueron reflejados desde un espejo?"*, o *"¿cuáles son los rayos que intersectan con el n-ésimo objeto de la escena?"*.

Por último, la herramienta se valida con 4 desarrolladores, a quienes se les presenta un conjunto de 5 problemas. Estos consisten de una descripción de una escena, una imagen con un error generada por un raytracer con algún tipo de problema, y un registro de rayos generado por este raytracer. El objetivo en cada caso es descubrir, mediante el uso de la aplicación web cargada con este registro, cuál es el problema con la imagen. En 2 de los 5 casos, todos los desarrolladores fueron capaces de encontrar el problema, mientras que en los restantes 3, la mitad fue capaz de encontrar el problema.

A mi familia

Tabla de Contenido

Índice de Tablas vii

Índice de Ilustraciones viii

1. Introducción 1

2. Raytracing 4
2.1. El proceso de renderizado . . . 4
2.2. Renderizado por raytracing . . . 4
2.2.1. Sombreado (*shading*) . . . 7
2.2.2. Reflexiones y refracciones . . . 9
2.2.3. Múltiples rayos y antialiasing . . . 11

3. El problema del depurado 13
3.1. Formato de escenas . . . 13
3.1.1. Componentes de una escena . . . 15
3.2. Motivación . . . 19
3.2.1. Raytracing en la industria . . . 20
3.3. Análisis del problema y solución propuesta . . . 21

4. Diseño de la solución 22
4.1. Estructura . . . 22
4.2. Características esenciales . . . 22
4.3. Registro de rayos . . . 23
4.3.1. Propiedades requeridas . . . 23
4.3.2. Propiedades opcionales . . . 23
4.3.3. Esquema propuesto . . . 24
4.4. Serialización del registro y convenciones . . . 26
4.5. Librería de logging . . . 26
4.6. Características concretas y soluciones . . . 28
4.6.1. Etiquetas . . . 28
4.6.2. Parentesco de rayos . . . 28

5. Implementación de la solución 30
5.1. Visión general . . . 30
5.2. Herramientas utilizadas . . . 30
5.3. Arquitectura . . . 32

5.3.1. Arquitectura de hardware 32
5.3.2. Arquitectura de software 33
5.4. Funcionalidades 34
5.4.1. Filtros 34
5.4.2. Inspector 37
5.4.3. Configuraciones 37
5.4.4. Carga del registro 37
5.5. Visualización y renderizado de la escena 39

6. Validación y pruebas **46**
6.1. Casos de estudio 47
6.1.1. Caso 1 47
6.1.2. Caso 2 52
6.1.3. Caso 3 58
6.1.4. Caso 4 63
6.1.5. Caso 5 67
6.2. Resultados 67

Conclusión **70**

Bibliografía **73**

A. Escenas **75**
A.1. Caso 1 75
A.2. Caso 2 76
A.3. Caso 3 77
A.4. Caso 4 79
A.5. Caso 5 80

B. Registros **82**
B.1. Esquema 82
B.2. Ejemplo 82

Índice de Tablas

6.1. Resoluciones por usuarios de los casos de estudio. 70

Índice de Ilustraciones

2.1. Proceso de renderizado 4
2.2. Recepción de la luz en el ojo 5
2.3. Resultado de aplicar el algoritmo 1 sobre la escena del listado 2.1. 7
2.4. Imagen resultante de raytracer con sombreado. 8
2.5. Raytracing 9
2.6. Cálculo de rayo reflejado. 10
2.7. Imagen resultante de raytracer con reflexiones. 10
2.8. Ley de Snell. 11
2.9. Imagen resultante de raytracer con refracciones. 11
2.10. Antialiasing. 12

3.1. Una escena generada por un raytracer. 15
3.2. Modelo de cámara 16
3.3. Malla de triángulos que da forma a un delfín. 17
3.4. Luz puntual 17
3.5. Luz en área. 18
3.6. Spot light 19
3.7. A la izquierda, la imagen correcta. A la derecha, la que tiene el error. 20

4.1. Flujo de alto nivel 23

5.1. Vista general 31
5.2. A la izquierda, una esfera renderizada por un raytacer. A la derecha, una vista del visualizador cargada con los rayos generados por el raytracer. 31
5.3. Posible arquitectura de hardware. 32
5.4. Arquitectura de aplicación web 33
5.5. Panel de filtros 34
5.6. Filtrado por etiquetas. 35
5.7. Filtrado por pixeles 36
5.8. Filtro por largo de rayo 37
5.9. Filtrar rayos con intersección 37
5.10. Inspector con tres rayos seleccionados. 38
5.11. Overlay del visualizador. 38
5.12. La figura 5.11, pero sin el overlay. 38
5.13. Opciones del loader. 39
5.14. Carga por escrito. 40
5.15. Visualizador con la escena A.2 cargada. 41

5.16. Visualizador con la escena A.2 cargada y un rayo seleccionado. 42
5.17. Visualizador con la escena A.2 cargada y un rayo al que se le ha aplicado la operación de seguimiento. 43

6.1. Caso 1 47
6.2. Caso 1 tras carga inicial 48
6.3. Caso 1 tras filtrar por una de las esferas pequeñas. 49
6.4. Caso 1 tras filtrar por la esfera grande. 50
6.5. Caso 1 tras mover la cámara. 51
6.6. Caso 2 52
6.7. Pestaña de filtros del caso 2. 53
6.8. Caso 2 tras carga inicial. 54
6.9. Caso 2 mostrando sólo rayos con intersecciones. 55
6.10. Caso 2 mostrando sólo rayos de sombra. 56
6.11. Caso 2 tras aplicar el filtro por largo de rayo. 57
6.12. Caso 3 58
6.13. Filtros para para el caso 3. 59
6.14. Caso 3 tras carga inicial. 60
6.15. Caso 3 mostrando rayos de reflexión. 61
6.16. Caso 3 mostrando rayos de refracción. 62
6.17. Caso 4 63
6.18. Caso 4 tras carga inicial. 64
6.19. Caso 4 mostrando sólo rayos con puntos de intersección. 65
6.20. Filtros en el caso 4. 65
6.21. Caso 4 mostrando sólo rayos de reflexión. 66
6.22. Caso 5 67
6.23. Caso 5 tras la primera carga. 68
6.24. Caso 5 mostrando rayos asociados al pixel $(1, 1)$ 69
6.25. Filtrado para el caso 5. 69

Capítulo 1

Introducción

El depurado de aplicaciones gráficas se considera difícil por las características inherentes del área. En particular, porque los problemas suelen involucrar imágenes o video. Es comúnmente claro cuando hay un error de este tipo, porque *se ve*, pero no hay indicaciones necesariamente sobre cuál es la entidad o componente responsable.

Por otro lado, la técnica de renderizado *rayracing* [17], utilizada principalmente de manera *offline* (es decir, a una tasa de imágenes generadas por segundo lo suficientemente baja para no poder interactuar con ellas en la práctica), ha adquirido creciente popularidad producto de avances en hardware [9]. Por el contrario, el esfuerzo se ha dedicado a las herramientas de desarrollo y, en particular, de depurado, a otra técnica de renderizado, *rasterizado* [16], en desmedro de raytracing.

Objetivos

El objetivo de este trabajo es desarrollar un método efectivo para el depurado de raytracers, y la construcción de una plataforma en sintonía con lo anterior. Se introduce *Raylogger* como una solución a este problema, una plataforma web, en conjunto con un formato estándar para serializar rayos y una implementación concreta de una librería para generar registros de rayos con este formato. En concreto, debe ser:

- Agnóstica al lenguaje y al entorno de desarrollo del consumidor.
- Tolerante frente a una gran cantidad de información de rayos.
- Capaz de visualizar los datos de una manera útil.
- Capaz de filtrar y/o indicar los datos relevantes.

En conjunto con, y a modo de complemento a la plataforma, se quiere construir una herramienta para un lenguaje específico, C++, de forma que se tenga un ejemplo de cómo podrían integrarse proyectos que utilicen entornos de desarrollo distintos.

Metodología

Se trabajó iterativamente en períodos variables de una a dos semanas. En las primeras iteraciones el objetivo es construir un marco estable sobre el cual hacer modificaciones. En particular, se configura el proyecto para poder hacer deployments a Heroku [1]. En esta parte inicial, el esfuerzo se dedica a la carga de los rayos y a una versión muy sencilla del renderizado.

Las siguientes iteraciones terminaban con la implementación de alguna característica en particular del sistema, como el sistema de filtro y etiquetado. Hacia el final, se persiguen mejoras de rendimiento del renderizador.

Trabajo relacionado

Las herramientas de depurado convencionales suelen no brindar la información clave que es relevante para un desarollador de un raytracer, por ser típicamente de muy bajo nivel. En [13] se introduce *rtVTK* (Ray Tracing Visualization Toolkit) como una solución a este problema. Consiste de un conjunto de librerías y herramientas de visualización que se integran con raytracers. *rtVTK* tiene similaridades con lo expuesto en este trabajo, pero se piensa que la capacidad de filtrado no es lo suficientemente flexible. Por otro lado, se cree que se puede reducir el esfuerzo de integración del sistema con el raytracer en estudio.

En [14] se propone un sistema que considera filtros sofisticados y que permite realizar consultas sobre el dominio de los rayos de una forma muy versátil. Sin embargo, exige que se identifiquen los rayos (con un identificador único), bajo la asunción de que todos los raytracers ya lo hacen de alguna forma. En este misma línea, para que los filtros sean relevantes es necesario incluir la información a través de una librería que requiere integrarse.

Por último, se considera un depurador con fines educativos [8], que provee una forma de seguir rayos relevantes y filtrados básicos según tipo de rayos. Se considera la idea una inspiración para este trabajo. Sin embargo, no es extensible, pues la herramienta está cohesionada a un raytracer específico.

Todos las implementaciones recién mencionadas tienen en común que involucran una aplicación nativa. Uno de los beneficios del acercamiento descrito aquí es la capacidad de integrarse con múltiples proyectos, por tratarse de una aplicación web. Por otro lado, el formato propuesto para el registro de rayos es suficientemente genérico como para exigir lo mínimo de conocimiento de un raytracer con respecto a sus rayos, sin perder expresividad en el sistema de filtros.

[1] Plataforma Cloud que permite alojar aplicaciones.

Contenido de la memoria

En el capítulo 2 se introduce la técnica y los conceptos relevantes para el desarrollo del trabajo. Se explica el proceso de cómo una imagen es generada por un raytracer, y progresivamente se aumenta el modelo para que incluya shading.

En el capítulo 3 se dan los detalles del problema. Se introduce un caso de motivación y se explica por qué es relevante brindar una solución, considerando el contexto de la industria. Además, se introduce el formato de *escenas*, que es el utilizado por una implementación concreta del raytracer que se ocupa para analizar los casos de estudio discutidos en el capítulo 6.

En el capítulo 4 se describe conceptualmente la solución. Se discute el esquema de registro de rayos sobre el cual se construyen las consultas que el sistema es capaz de resolver. Se señala, además, el rol de la librería de logging, y se explica el sistema de etiquetado y de parentesco de rayos.

En el capítulo 5 se discuten los componentes concretos de la aplicación. En particular, se introduce la solución como una plataforma web. Se detalla, también, la arquitectura de software y de hardware de la implementación. Se explican en detalle todas las funcionalidades relevantes de la plataforma, como el sistema de filtros, y se discuten posibles casos de uso. Además, se profundiza en el proceso de renderizado de la aplicación, y sobre cómo hacerlo eficiente.

Por último, en el capítulo 6 se estudian cinco casos de uso de la plataforma, y se discute cómo podría ser utilizada para resolver el problema en cada caso. Como parte de ese trabajo, además, se le envían los casos a cuatro desarrolladores de raytracers, y los resultados de este experimento se presentan en esta sección.

Capítulo 2

Raytracing

2.1. El proceso de renderizado

El renderizado es la generación de una imagen a partir de un modelo bi o tridimensional. Se puede pensar como un algoritmo que cuya entrada se denomina **escena** y que tiene como salida una **imagen**.

Definición 2.1 ***Renderizador*** *Aplicación que implementa una técnica de renderizado.*

En términos prácticos en el contexto de un renderizador, una escena podría ser una codificación en texto de modelos tridimensionales que contenga una descripción de sus vértices y posiciones en el espacio en conjunto con directrices de cómo estos objetos reaccionan a alguna fuente de luz (materiales). El resultado de aplicar el algoritmo podría ser una matriz bidimensional que codifique, de alguna forma, colores que puedan posteriormente ser interpretados como una imagen.

2.2. Renderizado por raytracing

Raytracing [17] es una técnica de renderizado que se caracteriza por el lanzamiento analítico de rayos (en el sentido geométrico, i.e, una semirrecta con origen y dirección) que intenta emular el modelo de la luz. La forma en la que los rayos interactúan con la escena determinará los colores de la imagen.

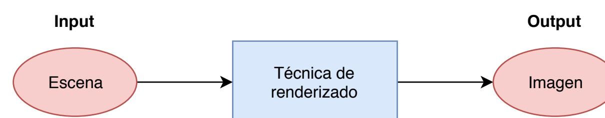
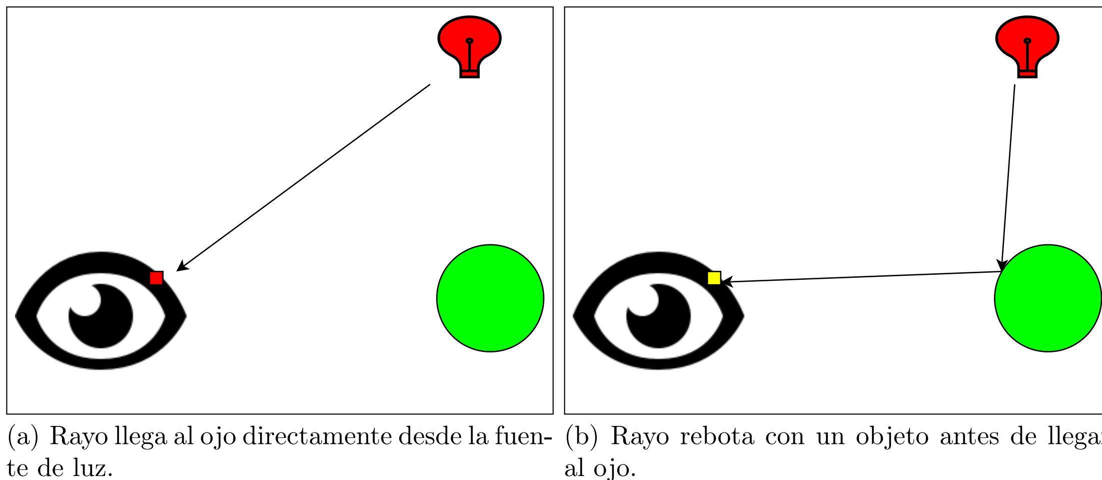


Figura 2.1: Proceso de renderizado

Definición 2.2 ***Raytracer*** *Renderizador que utiliza raytracing.*

Para razonar sobre la técnica, es útil pensar en el ojo humano como un renderizador. Para que se forme la imagen en la retina, el cerebro considera los (conceptualmente infinitos) rayos emitidos por una fuente de luz (como el sol). Cuando un rayo de luz de algún color (blanca en el caso del sol) llega al ojo, porque rebotó con algo o porque llegó directamente desde la fuente de luz, entonces se computa en el cerebro el color que se verá en la pequeña zona en la que ha llegado el rayo.



(a) Rayo llega al ojo directamente desde la fuente de luz.

(b) Rayo rebota con un objeto antes de llegar al ojo.

Figura 2.2: Recepción de la luz en el ojo

Si el rayo vino directamente desde la fuente (i.e, sin que haya existido algún objeto en su camino), entonces el color será el de la fuente (figura 2.2(a)). Si el rayo chocó con un objeto, entonces el color será una mezcla entre el color de la fuente y el color del objeto en el punto de choque (figura 2.2(b)). La forma en la que se hace la mezcla depende de las propiedades del objeto.

Se podría pensar entonces en un algoritmo de renderizado que intente emular el comportamiento de la luz en la vida real. Consistiría en considerar un rectángulo (lona) que represente la imagen final, un ojo y una fuente de luz. Se lanzarían infinitos rayos desde la fuente de luz. Si un rayo pasa por la lona y llega al ojo, significa que el color de ese punto en la lona será función de la información que traiga ese rayo.

Esto conceptualmente funcionaría para modelar la luz. Desafortunadamente, falla en la parte de "lanzar **infinitos** rayos", porque no se puede hacer en la práctica. El raytracing real funciona parecido a esto, pero con un pequeño ajuste: los rayos se lanzan **desde el ojo** en búsqueda de una fuente de luz (figura 2.5). El algoritmo considera una cámara (análoga al ojo en el ejemplo anterior), un rectángulo (lona) que representa el lugar donde se van a proyectar los colores, y los elementos de la escena, que pueden ser objetos y fuentes de luz, entre otros.

Un **primer acercamiento** a un algoritmo de raytracing es el algoritmo 1.

Data: Una escena
Result: Una imagen
Para cada punto de la lona (el rectángulo) lanzar un rayo con origen en la cámara y dirección tal que se dirija hacia el punto de la lona;
if *el rayo intersecta con un objeto* **then**
| Asignarle el color del objeto en el punto de intersección a la lona;
else
| Asignarle un color por defecto a la lona;
end
Retornar la lona coloreada;

Algorithm 1: Raytracing simple

Considere ahora que existe una codificación de una escena que describe dos esferas con colores con la forma del listado 2.1.

Listado 2.1: Posible codificación de escena

```
{
  camera: {
    position: [0, 0, 5]
  },
  objects: [
    {
      type: 'sphere',
      center: [2, 0, 0],
      radius: 0.5,
      color: 'red'
    },
    {
      type: 'sphere',
      center: [0, 0, 0],
      radius: 1,
      color: 'green'
    }
  ]
}
```

Se puede convenir que esta descripción de una escena signifique que hay dos esferas, una ubicada en $(x = 2, y = 0, z = 0)$ de color rojo y radio 0.5, la otra en $(x = 0, y = 0, z = 0)$ de color verde y radio 1, además de una cámara situada en $(x = 0, y = 0, z = 5)$.

Aplicar el algoritmo 1 sobre la escena anterior tendría como resultado una imagen como la que se muestra en la figura 2.3.

El ejemplo anterior no considera fuentes de luz. Por el contrario, asume que los objetos están completamente iluminados en todos los puntos. Se puede aumentar el algoritmo 1 para que considere la posición de una fuente de luz y calcule **sombreado**.

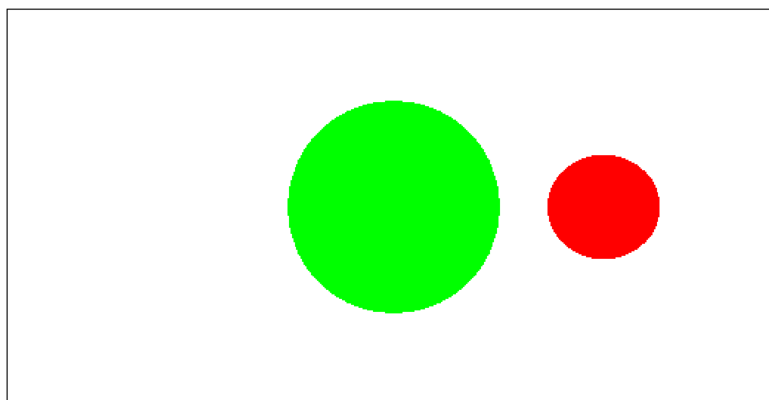


Figura 2.3: Resultado de aplicar el algoritmo 1 sobre la escena del listado 2.1.

2.2.1. Sombreado (*shading*)

Para resolver el problema del sombreado en raytracing, hay que responder potencialmente dos preguntas cuando un rayo intersecta con un objeto:

1. ¿La luz emitida por la fuente alcanza a llegar al punto?
2. Si la luz llega, entonces ¿cómo se calcula el color del punto?

Para responder la primera pregunta hay que lanzar **rayos de sombra**. Un rayo de sombra es un rayo que se lanza desde el punto de intersección del rayo original con el objeto en dirección a la fuente de luz (ver figura 2.5). Si el rayo intersecta algo **antes** que la fuente, es decir, existe un obstáculo entre la fuente y el punto de intersección original, entonces a ese punto le corresponde **sombra**.

Si al punto **no** le corresponde sombra, es decir, está siendo iluminado, entonces el color resultante se computa como una función de la fuente de luz y el **material** del objeto. Un **material** es un conjunto de propiedades que se le asignan a un objeto para modelar sus interacciones con la luz. Entre estas propiedades típicamente se encuentra el color y características como la capacidad reflexiva/refractiva del objeto, especularidad, etc. Con lo anterior, una versión aumentada del algoritmo 1 queda como el algoritmo 2.

Si se considera que a la escena del listado 2.1 se le agrega una fuente de luz ubicada en $(x = -2, y = 0, z = 2)$, y se aplica el algoritmo 2, entonces resulta la imagen de la figura 2.4.

El resultado puede cambiar dependiendo del tipo de fuente de luz y el material que se considere. Hasta este punto la elección de estas propiedades se ha mantenido abierta, pues se profundizan estos conceptos en las secciones que siguen.

En la figura 2.5 se muestra una cámara que está lanzando rayos (*View Ray*) hacia la lona (*Image*). Cuando uno de los rayos (View Ray) intersecta el objeto, entonces se verifica si hay algún obstáculo entre la fuente de luz (*Light Source*) y el objeto (*Scene Object*). Se lanzan rayos de sombra (*Shadow Ray*) para verificar si existe un obstáculo entre el objeto y la fuente de luz.

Data: Una escena
Result: Una imagen
Para cada punto de la lona (el rectángulo) lanzar un rayo con origen en la cámara y dirección tal que se dirija hacia el punto de la lona;
if *el rayo intersecta con un objeto* **then**
    Lanzar un rayo de sombra desde el punto de intersección hacia la fuente de luz;
    if *hay un obstáculo* **then**
        Asignar el color negro a la lona;
    else
        Computar el color de la lona como una función del material del objeto y la fuente de luz;
    end
else
    Asignarle un color por defecto a la lona;
end
Retornar la lona coloreada;

Algorithm 2: Raytracing con sombreado

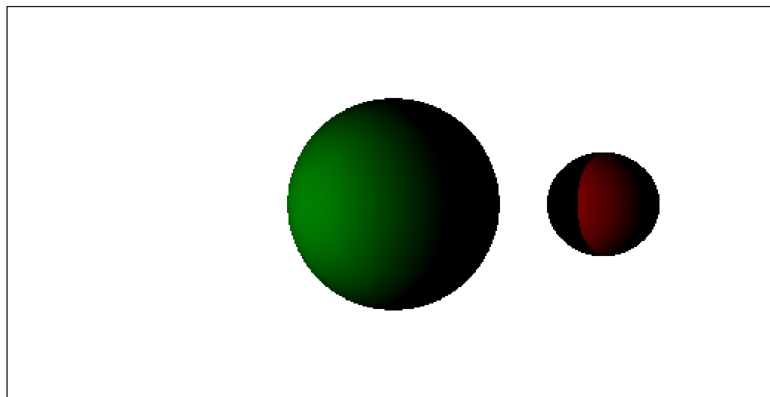


Figura 2.4: Imagen resultante de raytracer con sombreado.

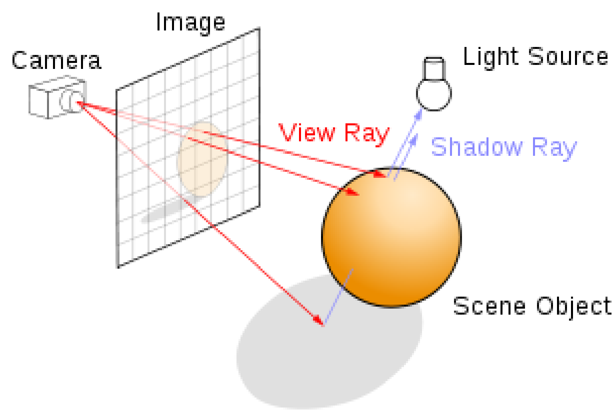


Figura 2.5: Raytracing

Uno de los beneficios principales de la técnica es la facilidad con la que se consiguen emular fenómenos propios de la luz real. La consecuencia de esto es que se pueden conseguir efectos relativamente realistas simplemente traduciendo el modelo de la luz a rayos de un raytracer.

2.2.2. Reflexiones y refracciones

Se pueden conseguir múltiples efectos **cambiando el comportamiento de los rayos cuando intersectan con un objeto con un material determinado**. A modo de ejemplo, se consideran en esta sección reflexiones y refracciones.

Reflexiones

Para calcular el rayo reflejado $\vec{R_r}$ de un vector $\vec{R_\text{i}}$ con respecto a un vector normal $\vec{N}$, se utiliza la ecuación 2.1 (figura 2.6).

$$\vec{R_r} = 2\vec{N}(\vec{R_\text{i}} \cdot \vec{N}) - \vec{R_1} \tag{2.1}$$

Así, para implementar reflexiones en un raytracer, cuando un rayo intersecta con un objeto que tiene un material reflexivo se pueden seguir los siguientes pasos:

1. Calcular el rayo reflejado $\vec{R_r}$ del rayo incidente con respecto al vector normal a la superficie del objeto, $\vec{N}$ con la ecuación 2.1.
2. Recursivamente lanzar el rayo $\vec{R_r}$ y calcular el color que le corresponde.
3. Determinar que el color que le corresponde al rayo incidente es el mismo que el color que se le asigna a $\vec{R_r}$.

Se extiende la escena del listado 2.1 para que contenga una esfera con un material reflexivo (*"white_mirror"*) y propiedades como indica el listado 2.2. El resultado de hacer esta adición es la figura 2.7.

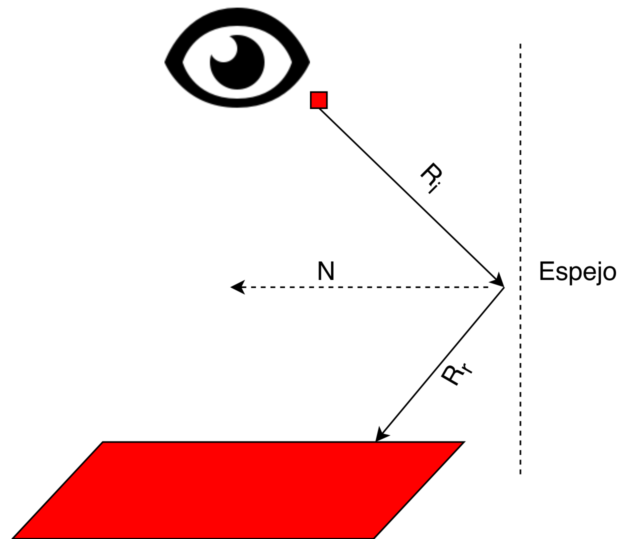


Figura 2.6: Cálculo de rayo reflejado.

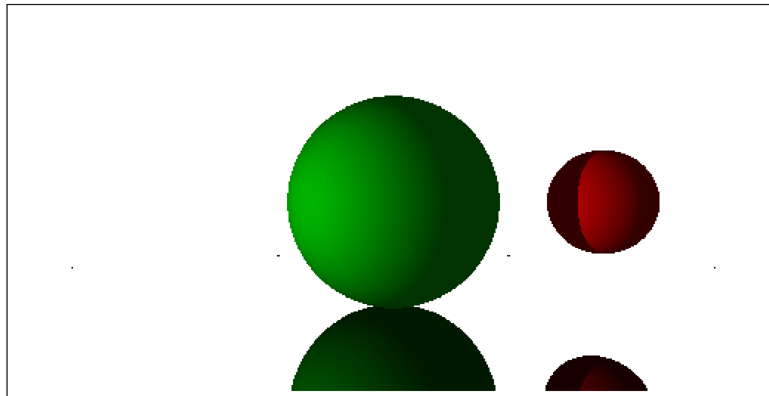


Figura 2.7: Imagen resultante de raytracer con reflexiones.

Listado 2.2: Descripción de una esfera reflectiva.

```
{
  type: "sphere",
  position: [0, -101, 0],
  radius: 100,
  materials: [ "white_mirror" ],
}
```

Refracciones

Un material **dieléctrico** es un material transparente que refracta la luz. Para implementar refracciones en un raytracer se puede utilizar la ley de Snell, que establece que cuando un rayo de luz pasa de un medio (como del agua al aire, por ejemplo) a otro, entonces se cumple que

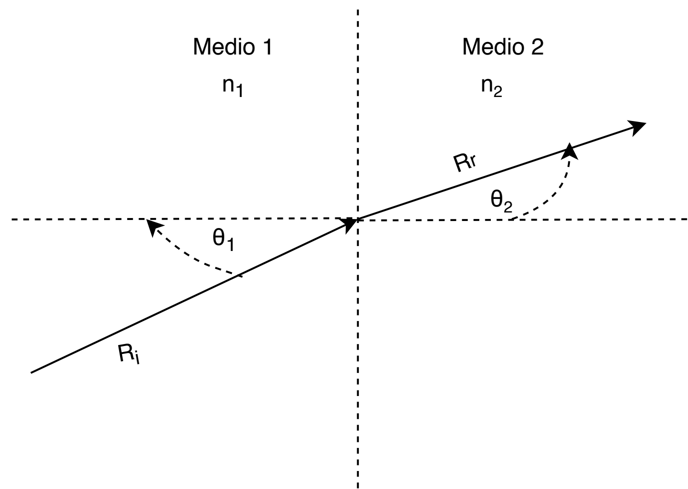


Figura 2.8: Ley de Snell.

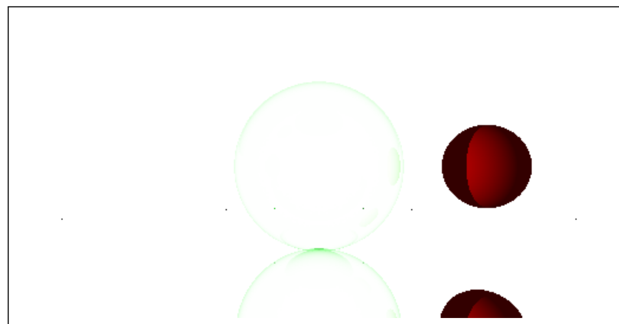


Figura 2.9: Imagen resultante de raytracer con refracciones.

$$n_1 sin(\theta_1) = n_2 sin(\theta_2) \tag{2.2}$$

donde n_i y θ_i indican los coeficientes de refracción de los dos medios, y los ángulos de incidencia y refracción de dos rayos, según se se muestra en la figura 2.8.

Con la ecuación 2.2 se puede calcular el rayo **reflectado** de un objeto que tiene un material dieléctrico, y seguir un procedimiento similar al de la sección anterior para computar el color correspondiente. La figura 2.9 muestra el resultado de asignarle un material dieléctrico a la esfera verde.

2.2.3. Múltiples rayos y antialiasing

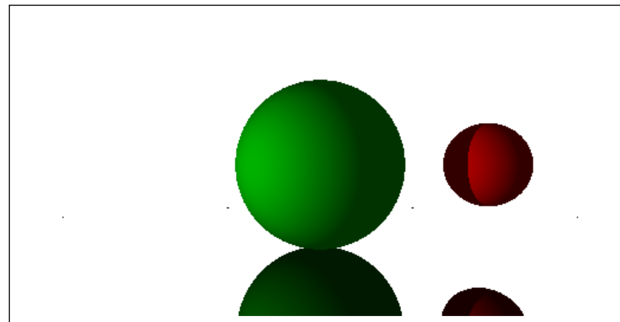
La sección anterior describe brevemente la implementación de reflexiones y refracciones. Estas técnicas tienen en común que requieren un tratamiento determinado de los rayos según el material del objeto con el que estos interactúen.

En esta parte se discute *antialiasing*, que cae en la categoría de aquellos efectos que necesitan que se lancen múltiples rayos por pixel para lograrse.

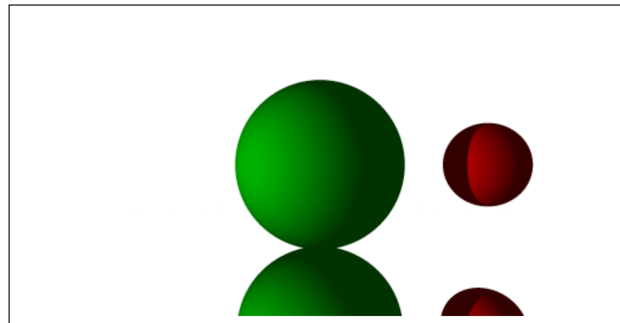
Antialiasing, en computación gráfica, es una técnica que permite suavizar las secciones de una imagen que quedan distorsionadas producto de la transformación del modelo a la imagen final que es de naturaleza discreta (geometrías continuas vs. pixeles).

La implementación concreta consiste en dividir **cada pixel** en n puntos o zonas a través de las cuales se lanzarán rayos, uno por cada una. En este caso, se dice que se lanzan n rayos por pixel. Luego, se calcula el color de cada rayo de la misma forma en la que se haría en el caso de un rayo por pixel, y se asigna que el color del pixel es el **promedio** de los colores de todos los rayos lanzados para ese pixel.

La figura 2.2.3 muestra la imagen de la figura 2.7 sin antialiasing, junto con otra en la que sí se utilizó antialiasing con **16 rayos por pixel**.



(a) Sin antialiasing.



(b) Con antialiasing, 16 rayos por pixel.

Figura 2.10: Antialiasing.

Capítulo 3

El problema del depurado

El desarrollo de aplicaciones en computación gráfica no está exento de problemas. En el caso de los raytracers, la salida es una imagen y, por lo tanto, la existencia de un error que es producto de lanzar malos rayos en el programa probablemente producirá que se genere una imagen que no es la que se espera. Esto significa que los colores de algunos pixeles que conforman la imagen no quedaron bien asignados, y la causa de este problema puede ser difícil de descubrir dado que los algoritmos de raytracing no son fáciles de seguir como en aplicaciones convencionales. Si bien no son típicamente difíciles de escribir, el hecho de que un color sea generado como consecuencia de haber lanzado y mezclado recursivamente un conjunto no necesariamente pequeño de rayos, hace que esta tarea se vuelva particularmente compleja.

Además, un color incorrecto no dice nada sobre la naturaleza del responsable de la mala asignación, pues no es más que un color, en oposición a lo que ocurre en, por ejemplo, una aplicación web en la que si hay un botón que no funciona, se tiene una pista concreta de que probablemente exista algún error o algo que acerque al error en la lógica que gobierna el comportamiento de ese botón.

El problema es entonces que el depurado convencional para aplicaciones gráficas es tedioso y no tiene direcciones claras: prácticamente lo que se puede hacer es sumergirse en el código con la esperanza de que se encontrarán pequeñas pistas que conducirán a la raíz del error.

3.1. Formato de escenas

A lo largo de este trabajo se presentan imágenes generadas por un raytracer que acepta escenas descritas en un formato que se presenta a continuación. Lo que sigue es un ejemplo:

Listado 3.1: Escena simple con esfera verde.

```
1 {
2   "__type__": "scene",
3   "camera": {
```

```
    "__type__": "camera",
    "fov": 45.0,
    "position": [ 0, 0.5, 1 ],
    "up": [ 0, 1, 0 ],
    "target": [ 0, 0, 0 ]
  },
  "objects": [
    {
      "__type__": "sphere",
      "radius": 10,
      "position": [ 0, -10.0, 0.0 ],
      "materials": [ "white_lambert" ]
    },
    {
      "__type__": "sphere",
      "radius": 0.3,
      "position": [ 0, 0.35, 0.0 ],
      "materials": [ "green_lambert", "white_bph_100" ]
    }
  ],
  "lights": [
    {
      "__type__": "area_light",
      "position": [ 0, 20, 0 ],
      "color": [ 0.5, 0.5, 0.5 ],
      "sizeA": 2,
      "sizeB": 2,
      "directionA": [ 1.0, 0.0, 0.0 ],
      "directionB": [ 0.0, 0.0, 1.0 ]
    },
    {
      "__type__": "ambient_light",
      "color": [ 0.2, 0.2, 0.2 ]
    }
  ],
  "params": {
    "background_color": [ 0.1, 0.1, 0.1 ]
  }
}
```

La salida del raytracer utilizado cuando la entrada es la escena anterior es la figura 3.1.

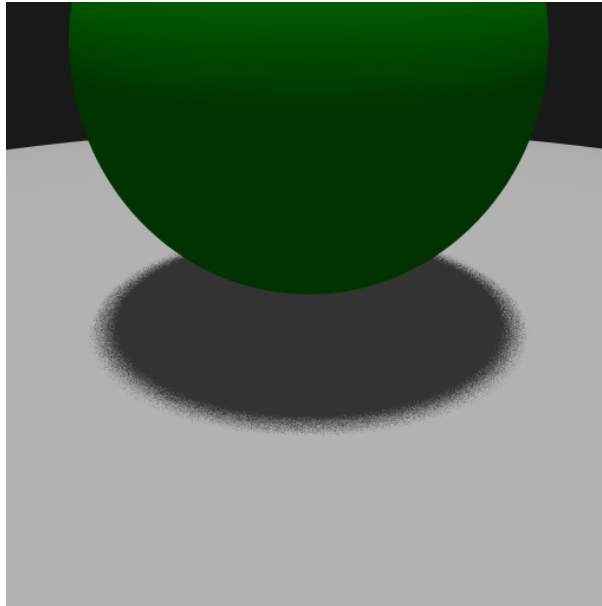


Figura 3.1: Una escena generada por un raytracer.

3.1.1. Componentes de una escena

Para comprender las descripciones de las escenas incluidas en el apéndice A, y que son utilizadas en la sección 6, se proveen en este punto explicaciones de lo que significan algunos conceptos importantes y la forma en la que se presentan en el formato anterior.

Para que una escena tenga sentido, es necesario contar con, al menos:

- Cámara: representa el punto de vista desde el cual la escena será plasmada.
- Objetos: representa el contenido de la escena, es lo que será renderizado.
- Fuentes de luz: representa la iluminación de la escena. Sin luz la cámara no "ve" nada.

Cámara

Es **desde dónde** se renderiza la escena. El modelo de cámara considerado en este trabajo, ilustrado en la figura 3.2, tiene como propiedades ***fov***, ***position***, ***up*** y ***target***.

Propiedades

- **position**: vector que representa la posición de la cámara en el espacio.
- **target**: punto al que la cámara esta apuntando. Se puede pensar como la forma de especificar la dirección de la cámara.
- **up**: vector que representa, desde el punto de vista de la cámara, dónde está "arriba". Como analogía, si se considera la retina del ojo como una cámara, y se fija un punto al que se está mirando (**target**), se puede girar la cabeza sin cambiar de dirección. En

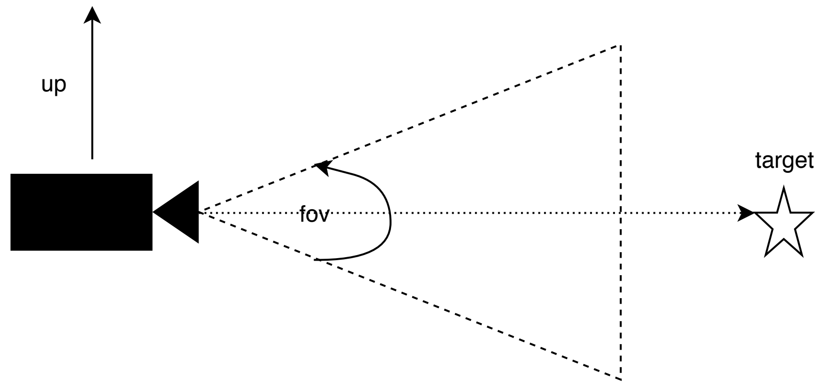


Figura 3.2: Modelo de cámara

este caso, se está cambiando el "up" de la cámara y, en consecuencia, también la imagen resultante.

- **fov**: El campo de visión **vertical** en grados (ver figura 3.2). El campo de visión horizontal queda determinado por el *aspect ratio* de la imagen, es decir, la razón entre el ancho y el alto en pixeles de la imagen.

Objetos

Corresponde al contenido, a **qué** se renderizará en una escena. Dependiendo del tipo ("__*type*__" en el formato), los objetos pueden tener distintas propiedades. En el contexto de este trabajo se consideran dos tipos de objetos: **esferas** (*sphere*) y **mallas** (*mesh*).

Esferas Un objeto con tipo esfera representa una esfera, i.e, una geometría que queda bien definida con un **radio** y una **posición**.

Propiedades

- **radius:** valor númerico que indica el radio de la esfera.
- **position:** vector que representa la posición de la esfera en el espacio.
- **materials:** los materiales asociados a la esfera. Ver sección 3.1.1.

Mallas Para modelar geometrías que no son fácilmente describibles analíticamente (como una esfera), se utilizan **mallas**. Una malla es un estructura que define la forma de un poliedro especificando los vértices que lo componen y la manera en la que estos vértices deben formar aristas y caras. La figura 3.3 muestra una malla de triángulos que da forma a un delfín.

Propiedades

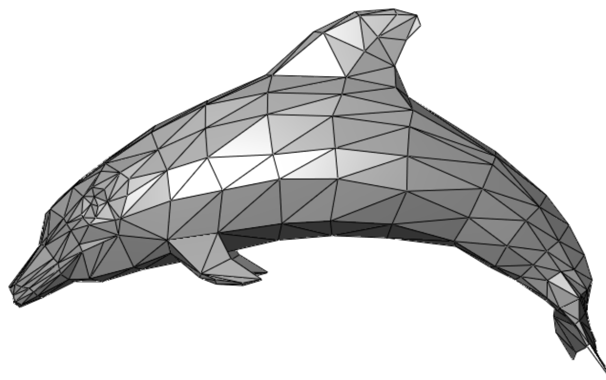


Figura 3.3: Malla de triángulos que da forma a un delfín.

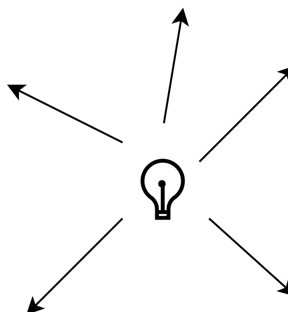


Figura 3.4: Luz puntual

- **file_path:** ruta a archivo que contiene la información de la malla.
- **materials:** lista de materiales asociados a la malla. Ver sección 3.1.1.

Fuentes de luz

Corresponden a entidades que conceptualmente emiten luz de una forma determinada que depende del tipo de la fuente. En este trabajo se consideran cinco tipos de fuentes de luz: **puntual** (*"point_light"*), **direccional** (*"directional_light"*), **ambiental** (*"ambiental_light"*), **en área** (*"area_light"*) y ***spot light*** (*"spot_light"*).

Las propiedades de cada fuente de luz depende de su tipo:

Luz ambiental Afecta a **todos** los objetos de la escena uniformemente, sin importar sus ubicaciones.

Propiedades

- **color:** representación RGB del color que emite la luz.

Luz puntual Emite luz en todas las direcciones desde un punto determinado en el espacio. Ver figura 3.4.

Propiedades

- **position:** vector que representa la posición en el espacio de la luz.
- **color:** representación RGB del color que emite la luz.

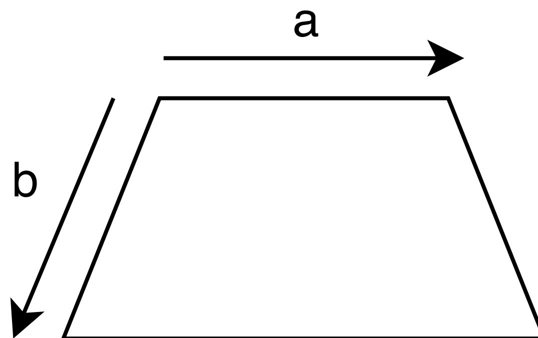


Figura 3.5: Luz en área.

Luz en área Emite luz de forma perpendicular al plano al que pertenece un trapecio isósceles definido por dos vectores, $\vec{a}$ y $\vec{b}$. La luz es emitada por todos los puntos interiores al trapecio. Ver figura 3.5.

Propiedades

- **position:** punto que define el origen de los vectores $\vec{a}$ y $\vec{b}$.
- **sizeA:** tamaño del vector $\vec{a}$.
- **directionA:** dirección del vector $\vec{a}$.
- **sizeB:** tamaño del vector $\vec{b}$.
- **directionB:** dirección del vector $\vec{b}$.
- **color:** representación RGB del color que emite la luz.

Spot light Similar a la luz puntual, pero restringe la dirección en la que se emite la luz. Define una dirección y un ángulo de apertura. La luz se emite sólo en la dirección del cono construido de esta manera. Ver figura 3.6.

Propiedades

- **position:** punto de origen de la luz.

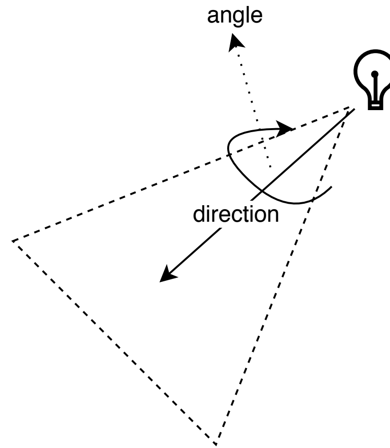


Figura 3.6: Spot light

- **direction:** hacia donde apunta la luz. Bisecta el cono definido en conjunto con **angle**.
- **angle:** ángulo de apertura dentro del cual se emite luz.
- **color:** representación RGB del color que emite la luz.

Materiales

Un material representa un conjunto de características que pueden ser asociadas a un objeto. Propiedades como el color, especularidad o textura son incluidas dentro de un material. A modo de ejemplo, se considera la escena del listado 3.1 que genera la imagen de la figura 3.1:

La esfera verde que se observa tiene dos materiales: "*green_lambert*" y "*white_bph_100*".

El material **"*green_lambert*"** le da el color verde a la esfera gobernado por el modelo de luz difusa de Lambert [15]. El material **"*white_bph_100*"** le asigna especularidad.

En general, se pueden combinar distintos materiales para lograr distintos efectos sobre un objeto. Dar la sensación de rugosidad sobre una malla plana, otorgar propiedades refractivas o reflectivas son casos en los que normalmente se utilizan materiales para describir tales efectos.

3.2. Motivación

A continuación se muestran dos imágenes. Una fue generada con una implementación correcta de un raytracer y la otra fue generada por uno que tiene un error. La descripción de la escena se encuentra en la sección A.1 del apéndice.

Para descubrir cuál puede ser el problema hay que entender la descripción de la escena. Hay una esfera de radio 100 y tres esferas de radio menor a 1. La esfera grande es un espejo

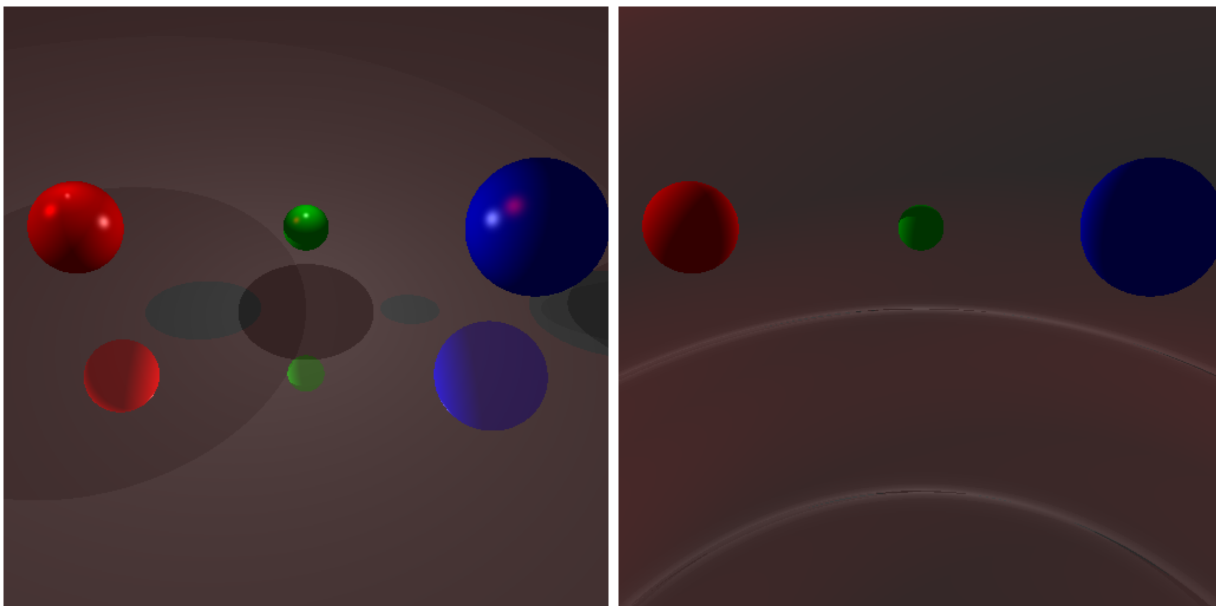


Figura 3.7: A la izquierda, la imagen correcta. A la derecha, la que tiene el error.

(tiene el material “white_mirror”) y en la imagen correcta se observa que están las tres esferas por flotando por sobre la esfera más grande que es demasiado grande como para alcanzar a verse entera en la imagen, sin embargo, se ve cómo las esferas están siendo reflejadas por ella.

En la imagen que muestra el raytracer incorrecto no se está reflejando ninguna de las esferas, esto podría indicar que quizás hay algún problema con las reflexiones. Por otro lado, las esferas pequeñas parecen tener una sombra extraña y se aprecian curvas extrañas más claras con respecto a la esfera grande. En el capítulo 6 se profundiza en este caso y se propone un camino que se puede seguir para descubrir el error.

3.2.1. Raytracing en la industria

Los desarrolladores de aplicaciones gráficas han históricamente favorecido la técnica de rasterizado [16] para renderizar las imágenes de sus proyectos. Esta técnica requiere de más esfuerzo para lograr efectos realistas, pues, en esencia, consiste en generar un mapeo entre la escena y pixeles mediante algún método, y no intenta parecerse necesariamente al modelo de la luz.

En contraste, raytracing es relativamente lento, porque generalmente involucra lanzar al menos un rayo por pixel, y calcular intersecciones de los rayos con una potencialmente gran cantidad de objetos. Para lograr algunos efectos es necesario lanzar múltiples rayos por pixel, y luego seguir su recorrido en la escena, que puede involucrar relanzarlos en muchas ocasiones.

Esta diferencia en costo computacional produce que se prefiera la técnica de rasterizado para aplicaciones que necesitan renderización en tiempo real, i.e, generar múltiples cuadros por segundo para crear la sensación de que la escena está en continuo movimiento, como un video. Por otro lado, no es raro que en los casos de uso en las que uno se puede permitir

invertir mucho tiempo para generar cada cuadro se prefiera raytracing, como ocurre, por ejemplo, en las películas de animación digital [11].

Hasta hace poco, no se habían hecho intentos relevantes de llevar el uso de raytracing a aplicaciones orientada a consumidores. Las tarjetas de video disponibles en el mercado tienen una arquitectura pensada para acelerar la implementación de técnicas por rasterizado. La razón por la que esta tecnología (raytracing) se está masificando es que se han llevado a cabo los avances necesarios para que el hardware sea capaz de utilizar esta técnica en tiempo real.

La empresa Nvidia, que está fuertemente involucrada en el diseño de tarjetas de video para los consumidores, ha provisto al mercado de raytracing acelerado por hardware [9]. La consecuencia de esto es que ahora los desarrolladores de aplicaciones gráficas pueden dar soporte a raytracing y esperar, razonablemente, que sus usuarios puedan explorarlo con los nuevos dispositivos.

En el marco del creciente interés y soporte de la industria por raytracing, es de esperar que la cantidad de proyectos que utilicen esta técnología aumente. De aquí que surge la necesidad de contar con herramientas y metodologías que apoyen el proceso de depurado, que puede consumir mucho tiempo.

3.3. Análisis del problema y solución propuesta

Como se ha mencionado, se piensa que el problema fundamental del depurado en aplicaciones gráficas es que los pixeles, que forman parte de la salida de una técnica de renderizado, no ofrecen pistas con respecto a qué posiblemente podría estar causando un error. Sin embargo, **la salida de los programas que ocupan raytracing está principalmente gobernada por los rayos**. Entonces se podría pensar en la utilidad que brindaría poder observar y/o estudiar los rayos que fueron responsables de generar una imagen incorrecta.

Dado que todas las aplicaciones que ocupan raytracing tienen en común que se basan en rayos para generar resultados, lo que se propone en este trabajo es:

1. La definición de un estándar de registro de rayos que pueda ser consumido por un visualizador.
2. La implementación de una librería en un lenguaje específico que se adhiera a este estándar.
3. La construcción de una plataforma que integre lo anterior y provea una forma de cargar este registro y visualizar rayos de forma útil.

Capítulo 4

Diseño de la solución

4.1. Estructura

El flujo principal de interacción con la solución propuesta se ilustra en la figura 4.1. Un usuario que tiene un raytracer, se integra con una **librería de logging**. Mediante esta librería, se genera un **registro de rayos** de acuerdo a un estándar acordado. Finalmente, este registro se carga en una **plataforma web**, con la cual el usuario interactúa.

4.2. Características esenciales

El objetivo de la plataforma web es poder permitirle al usuario analizar sus datos de una forma tal que le sea útil. En este contexto, se listan una serie de capacidades ilustrativas que debe poseer la plataforma:

1. Visualizar todos los rayos que intersectan con un objeto cualquiera.
2. Visualizar todos los rayos que están relacionados con un conjunto de pixeles.
3. Visualizar todos los rayos cuyo largo (cuando intersectan con algo) está en un intervalo.

Con esto, se propone qué datos serán parte del registro.

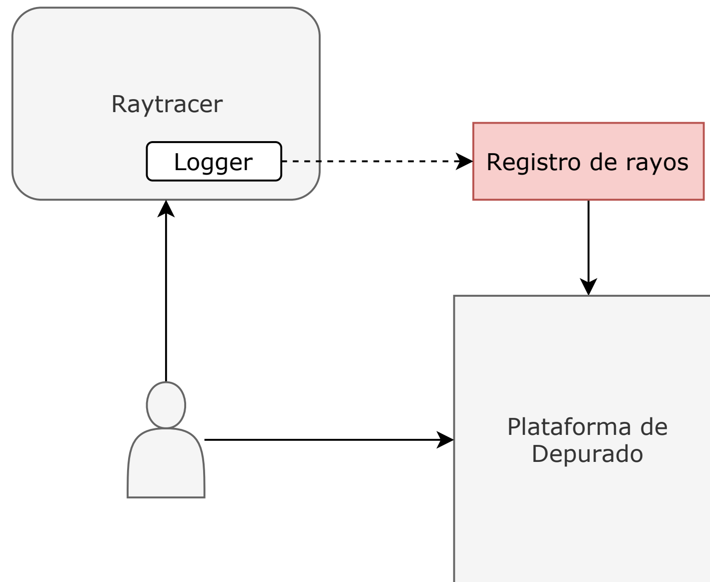


Figura 4.1: Flujo de alto nivel

4.3. Registro de rayos

4.3.1. Propiedades requeridas

Origen y dirección

Un rayo queda completamente definido con un punto de origen y una dirección (y queda mal definido sin ellos). Por lo tanto, se incluyen como propiedades requeridas del rayo.

4.3.2. Propiedades opcionales

Punto de intersección

Un rayo como tal es infinito, pero muchas veces lo relevante en raytracing es si efectivamente un rayo intersecta con algo no, y cuál es el punto en el que la intersección ocurre. Cuando un rayo intersecta con algo, la parte que viene *después* de la intersección es típicamente descartable. Como es importante analizar si un rayo choca o no con algo, se incluye el punto de intersección como una propiedad opcional en el registro. Si la propiedad no existe, se asume que el rayo es infinito.

Etiquetas

Hay potenciales problemas que pueden ser descartados al observar un conjunto de rayos específicos. Por ejemplo, a uno le gustaría poder expresar consultas como *"¿Cuáles son los*

rayos que fueron lanzados producto de una reflexión?", con la esperanza de que esto de alguna pista sobre si hay un problema con el cálculo de rayos reflejados. Si tras filtrar de esta forma se ve algo sensato, entonces valdría la pena redirigir el esfuerzo a otro tipo de sospechas.

¿Qué ocurre si la escena en general se ve bien, pero hay un objeto en particular que tiene un sombreado extraño? Sería interesante estudiar que está ocurriendo con los rayos que intersectan con ese objeto.

En general, uno podría agrupar esta categoría de consultas asignándole etiquetas (*tags*) a los rayos. Así, por ejemplo, aquellos que intersectan con la esfera grande de la figura, se les podría asociar con la etiqueta *"BIG_SPHERE"*. Si además los rayos son producto de una reflexión, entonces se les puede asignar la etiqueta *"REFLECTION"*. En general, para cualquier propiedad que uno considere razonable asociarle a un rayo, se puede crear una etiqueta.

Pixeles

A veces es claro que existe algún problema con alguna sección específica de una escena, pero no hay ningún indicio de que algún componente específico (como las reflexiones) esté funcionando mal. Entonces sería interesante visualizar exclusivamente los rayos que afectan a esa zona. Por otro lado, la mayoría de los rayos se lanzan con el objetivo de calcular el valor de un **pixel** específico. Se propone, entonces, considerar el pixel como una propiedad del rayo.

4.3.3. Esquema propuesto

El formato utilizado en la implementación es JSON [3] (JavaScript Object Notation) sujeto a la especificación JSON Schema draft-07 [6].

En este contexto, se definen esquemas para los conceptos de vectores de 3 y 2 dimensiones como los indican los listados 4.1 e 4.2, respectivamente.

Listado 4.1: Esquema de Vector3

```
1 {
2   "$id": "http://raylogger.com/schemas/vector3.json",
3   "title": "Vector3",
4   "type": "array",
5   "items": { "type": "number" },
6   "maxItems": 3,
7   "minItems": 3
8 }
```

Listado 4.2: Esquema de Vector2

```
1 {
```

```
  "$id": "http://raylogger.com/schemas/vector2.json",
  "title": "Vector2",
  "type": "array",
  "items": { "type": "number" },
  "maxItems": 2,
  "minItems": 2
}
```

Con esto, se puede definir el concepto de rayo (*Ray*), especificando que las propiedades requeridas de uno son el origen y la dirección, como se muestra en el listado 4.3.

Listado 4.3: Esquema de Ray

```
{
  "$id": "http://raylogger.com/schemas/ray.json",
  "title": "Ray",
  "type": "object",
  "properties": {
    "origin": { "$ref": "http://raylogger.com/schemas/vector3.
       json" },
    "direction": { "$ref": "http://raylogger.com/schemas/vector
       3.json" },
    "intersection": { "$ref": "http://raylogger.com/schemas/
       vector3.json" },
    "pixel": { "$ref": "http://raylogger.com/schemas/vector2.
       json" },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  },
  "required": [ "origin", "direction" ]
}
```

En este caso, se indica que el origen, la dirección y el punto de intersección son vectores de 3 dimensiones, mientras que el pixel es un vector de 2 dimensiones. Por último, las etiquetas quedan definidos como una lista de strings.

Finalmente, se define un **registro de rayos** como una lista de *Ray*, como se muestra en el listado 4.4.

Listado 4.4: Esquema de Raylog

```
{
  "$id": "http://raylogger.com/schemas/raylog.json",
  "title": "Raylog",
  "description": "A registry of rays",
```

```
5     "type": "array",
6     "items": {
7       "$ref": "http://raylogger.com/schemas/ray.json"
8     }
9 }
```

4.4. Serialización del registro y convenciones

Para poder operar las propiedades de los rayos especificadas en la sección 4.3.3, se conviene que:

- Las propiedades *origin*, *direction* e *intersection* representan vectores tridimensionales, con sus 3 componentes ordenadas. Por ejemplo, si el valor de la propiedad es la lista $[1, 2, 3]$, entonces representa el vector $(x = 1, y = 2, z = 3)$.
- La propiedad *pixel* representa un vector bidimensional con sus 2 componentes ordenadas, análogo al caso tridimensional.
- La propiedad *tags* representa un conjunto de palabras. Si en la lista aparece una palabra más de una vez, entonces el conjunto no está bien definido.
- $p \in r \Leftrightarrow$ el rayo r posee la propiedad p.
- $r.p$ significa: del rayo r, su propiedad p.

4.5. Librería de logging

Este trabajo incluye una implementación en C++ (que se integra con raytracers escritos que utilizan este mismo lenguaje) de una librería generadora de registros que se adhieren al formato de la sección 4.3.3. Los registros de los casos estudiados en la sección 6 fueron generados con esta librería.

El objetivo es práctico e ilustrativo, en el sentido de que sirve como prueba tangible que es posible generar registros producidos por un raytracer real.

La implementación está constituida por una clase principal, *raylogger::logger*, y una serie de estructuras (dentro de las cuales destaca *raylogger::ray*). El modo de uso es directo: se crea una instancia de *raylogger::logger*, la cual almacena instancias de *raylogger::ray* en memoria, y luego construye el registro correspondiente.

Se considera el siguiente ejemplo, que genera la parte del registro que contiene los rayos que son lanzados desde la cámara de un raytacer:

Listado 4.5: Código de raytracer registrando rayos de cámara

```
1 const geo::vec3 rayOrigin = nextCamPosition;
2 const geo::vec3 rayDirection =
```

```
3     focalPoint - nextCamPosition.normalized();
4   geo::Ray ray(rayOrigin, rayDirection);
5
6   raylogger::ray loggedRay(
7     { rayOrigin.x, rayOrigin.y, rayOrigin.z },
8     { rayDirection.y, rayDirection.y, rayDirection.z }
9   );
10
11  loggedRay
12    .pixel({ static_cast<double>(i), static_cast<double>(j) })
13    .tag("CAMERA");
14
15  RaycastHit closestHitObject;
16  if (!raycast(ray, _scene->_objects, closestHitObject)) {
17    sumOfColors += renderer::Params::get().backgroundColor;
18    logger->addRay(loggedRay);
19    continue;
20  }
21
22  const geo::vec3 intersectionPoint =
23    closestHitObject.hitInfo.hitpoint;
24  loggedRay.intersection({
25    intersectionPoint.x,
26    intersectionPoint.y,
27    intersectionPoint.z
28  });
29  logger->addRay(loggedRay);
```

De la línea 1 a la línea 4 se genera el origen y dirección del rayo que será registrado, que se nombran *rayOrigin* y *rayDirection* respectivamente.

De la línea 6 a la línea 9 se crea una instancia de *raylogger::ray*, que será agregado posteriormente al logger.

De la línea 11 a la línea 13 se indica que el rayo está relacionado con el pixel $[i, j]$, y que tiene el tag *"CAMERA"*.

Desde la línea 15 a la 20 se incluye la lógica del lanzado del rayo. La instrucción **if** de la línea 16 se traduce como "¿el rayo *ray* intersecta con algo?". Si no intersecta con nada, entonces se agrega al registro (línea 18) y se termina.

De la línea 22 en adelante, se sabe que el rayo intersectó con algo, y el punto de intersección queda almacenado en *intersectionPoint* (línea 22). Finalmente, en la línea 24 se le asigna al rayo el punto de intersección, y en la línea 29 se agrega al registro.

Cuando ya se han agregado todos los rayos que se quieren incluir, se puede redigir la instancia de *raylogger::logger* por algún canal:

```
outputStream << logger;
```

4.6. Características concretas y soluciones

4.6.1. Etiquetas

Si $TAGS$ es un conjunto de etiquetas y $RAYS$ es un conjunto de rayos, entonces se pueden filtrar aquellos rayos aquellos que poseen **todas** las etiquetas, i.e, todos los rayos r tales que:

$$\bigwedge_{t \in TAGS} t \in r, \tag{4.1}$$

La implementación de este filtro permite resolver consultas como:

- Todos los rayos que intersectan con un objeto determinado.
- Todos los rayos que salen de la cámara.
- Todos los rayos que intersectan con un objeto determinado y que fueron producto de una reflexión.

El sistema de etiquetado es poderoso, en el sentido de que está limitado por la entidad responsable de asignar los tags.

4.6.2. Parentesco de rayos

En el contexto de un raytracer, los rayos tienen una jerarquía: se puede pensar en los rayos que fueron lanzados antes como los *padres* de otros rayos. A modo de ejemplo, se pueden considerar los rayos que son lanzados desde la cámara hacia la lona. Si uno de estos rayos intersecta con un objeto que tiene un material reflexivo, entonces el raytracer calculará un rayo reflejado desde el punto de intersección. En este caso, el rayo reflejado es *hijo* del rayo de la cámara.

Definición 4.1 ***Rayo padre*** *Un rayo r_1 es padre de otro rayo r_2, $r_1 \neq r_2$, con precisión P si se cumple que*

$$intersection \in r_1 \wedge ||r_1.intersection - r_2.origin|| < P \tag{4.2}$$

Definición 4.2 ***Rayo descendiente*** *Un rayo r_2 es descendiente de otro rayo r_1, $r_1 \neq r_2$, con precisión P si se cumple alguna de estas dos condiciones:*

1. *r_1 es padre de r_2 con precisión P.*
2. *El padre de r_2 es descendiente de r_1 con precisión P.*

Si se tiene una idea de cómo se comportarían los hijos de un rayo en particular (por ejemplo, si se sabe que se tendría que generar un rayo reflejado porque hay un espejo), entonces sería interesante poder verificar que los rayos hijos de otro tienen la forma que uno espera. La pregunta es entonces cómo resolver la siguiente consulta:

¿Cuáles son los descendientes de un rayo?

Se consideran dos acercamientos:

1. El usuario indica la relación de parentesco entre dos rayos.
2. La plataforma automáticamente determina la relación de parentesco en función de los puntos de intersección y orígenes de los rayos.

La diferencia entre ambas alternativas radica en quién va a tener que realizar mayor esfuerzo: la plataforma o el usuario. En este trabajo se escoge la segunda opción, porque se considera que es importante mantener la simplicidad del formato. Hacer que el usuario establezca la relación implica que su aplicación tenga información más detallada de los rayos de la que potencialmente tiene.

Cómputo de las relaciones de parentesco

Por la definición 4.2, para calcular los desciendentes de un rayo r con precisión P, hay que saber qué puntos de origen de los rayos de la escena están a menos de P de distancia del punto de intersección de r. En este trabajo se propone el uso de un *kd tree* [10]. Esta estructura permite una consulta de la forma:

$$\text{nearest_neighbour}(p, n, \mathrm{d}) \tag{4.3}$$

que calcula los n puntos más cercanos a p dada una distancia máxima d hasta p. Si se escoge $p = r.\text{intersection}$ y $\mathrm{d} = P$, entonces se pueden computar todos los puntos de origen de otros rayos que están ubicados a una distancia menor que P del punto de intersección de p, que es la definición de parentesco.

n debería escogerse como la suma entre la cantidad de puntos de origen y puntos de intersección que hay en la escena, pero, para efectos prácticos y sin afectar la complejidad de la consulta final, se puede escoger $n = ||RAYS|| * 2 = O(||RAYS||)$, donde $RAYS$ es el conjunto de los rayos considerados.

Capítulo 5

Implementación de la solución

5.1. Visión general

La plataforma cuenta con dos ejes principales:

1. Visualizador
2. Panel de control

El visualizador (número 1 en la figura 5.1) se encarga de renderizar la estructura interna de rayos que es mantenida por la aplicación, y cuya información fue cargada por el usuario en la sección de carga del panel de control (número 2 en la figura 5.1).

El usuario puede desplazar la cámara del visualizador mediante algún dispositivo (como teclado y mouse si está accediendo desde un computador personal). Se le permite mover libremente, hacer zoom y seleccionar rayos de la escena. Se profundizará en las funcionalidades específicas en las secciones que siguen.

5.2. Herramientas utilizadas

En esta sección se listan las herramientas principales utilizadas en la implementación de la solución. El uso que se les da se describe en la sección 5.3. La librería de logging no utiliza ninguna herramienta en particular además de la librería estándar de C++, por lo que sólo se incluyen las herramientas presentes en la aplicación web.

React

React [4] es una librería (considerado framework) de JavaScript (lenguaje de programación con mucha presencia en las tecnologías web) para la construcción de aplicaciones web

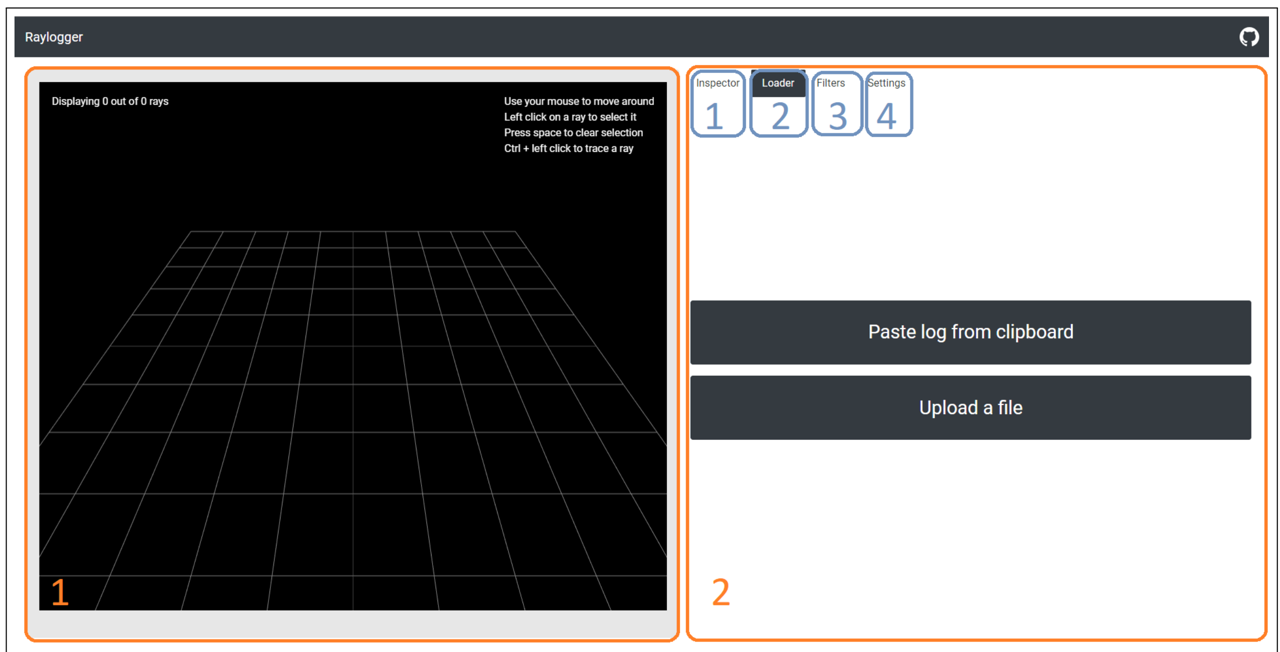


Figura 5.1: Vista general

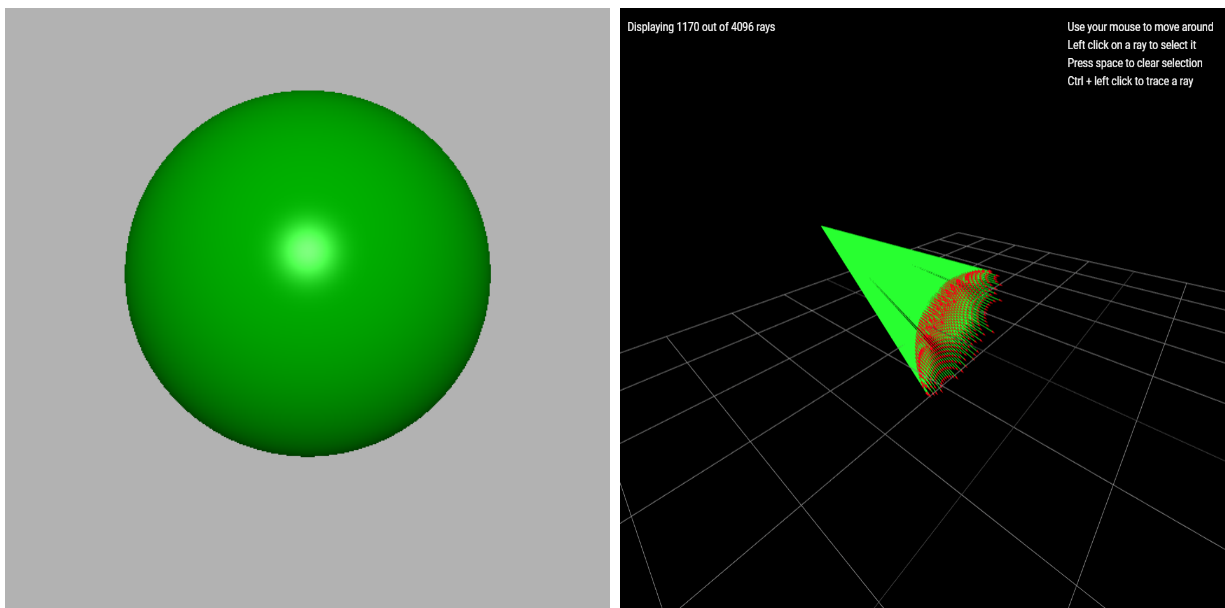


Figura 5.2: A la izquierda, una esfera renderizada por un raytacer. A la derecha, una vista del visualizador cargada con los rayos generados por el raytracer.

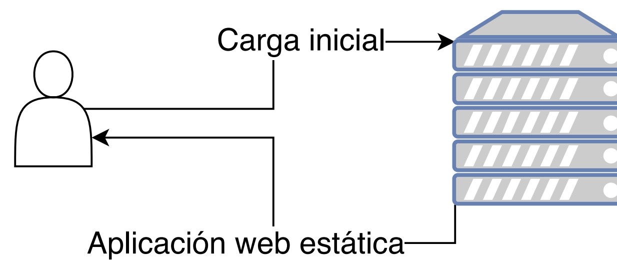


Figura 5.3: Posible arquitectura de hardware.

interactivas.

Redux

Redux [5] es una librería de JavaScript para manejar el estado de una aplicación, inspirada en la arquitectura **Flux** [1], que constituye una alternativa a la arquitectura Modelo-Vista-Controlador.

Three.js

Three.js [7] es una API de JavaScript para el renderizado de contenido 3D (y 2D). Hace uso de WebGL para lograr lo anterior.

5.3. Arquitectura

5.3.1. Arquitectura de hardware

La aplicación web está construida para que, una vez que el usuario cuenta con el contenido estático, funcione exclusivamente en el lado del cliente. Para un navegador en un computador personal, esto significa que tras la primera carga no se hacen más peticiones al servidor. En la figura 5.3 se aprecia lo anterior.

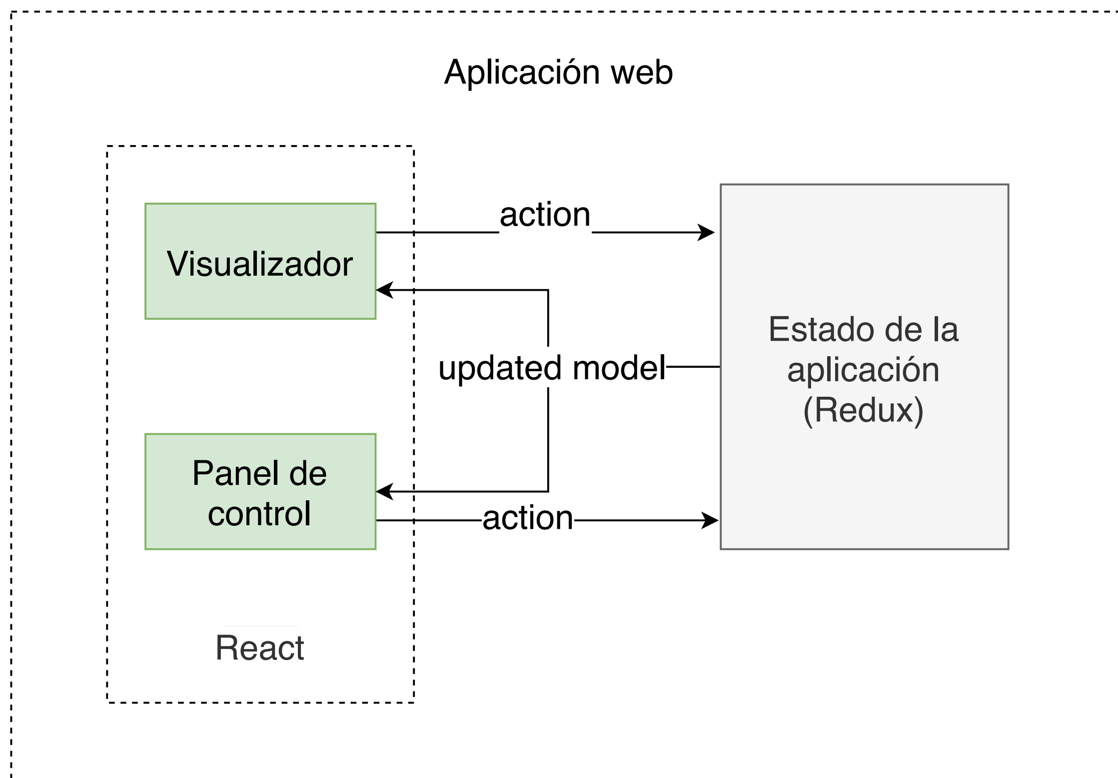


Figura 5.4: Arquitectura de aplicación web

5.3.2. Arquitectura de software

La capa de presentación se implementa con **React** [4]. Cada elemento de la interfaz representa un **componente** de React, cuyo contenido depende del **modelo** (o **estado**) de la aplicación, que interactúa con el resto de las entidades por medio de **Redux**.

El rol de React es el de mantener los componentes actualizados en función del modelo. En la práctica, React notifica a los componentes cada vez que existe algún cambio en el modelo que le sea pertinente al componente. **Toda** la información dinámica y estática de la aplicación reside en el modelo. Esto incluye, en particular, a los rayos.

El estado puede cambiar por influencia de una **acción**. Una acción ("*action*" en la figura 5.4) se puede pensar como un evento que es gatillado. Cada vez que se recibe una acción, ésta es procesada por un *reducer*. Un reducer es una entidad que, dada un acción y un estado, entrega un nuevo estado. Con este nuevo estado, se actualizan los componentes de React.

A modo de ejemplo, se considera el módulo de carga (sección 5.4.4). Cuando un usuario hace click en el botón *"Load"*, se gatilla un acción que es procesada por un reducer que recibe la nueva información cargada y genera una nueva instancia del estado para que contenga nuevos rayos. Como el estado ha cambiado, se le notifica de vuelta a los componentes que dependen de los rayos, que incluyen, por ejemplo, al visualizador.

La figura 5.4 ilustra lo anterior.

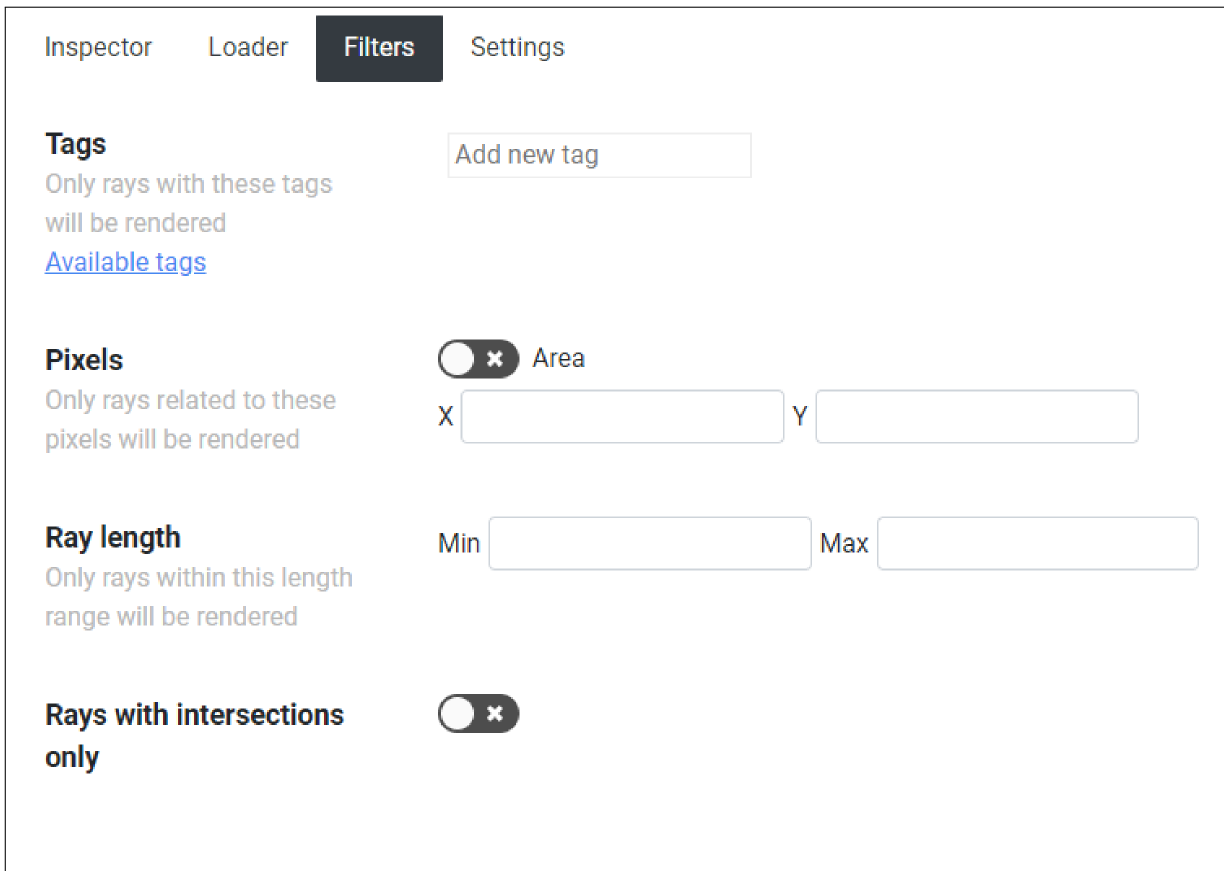


Figura 5.5: Panel de filtros

5.4. Funcionalidades

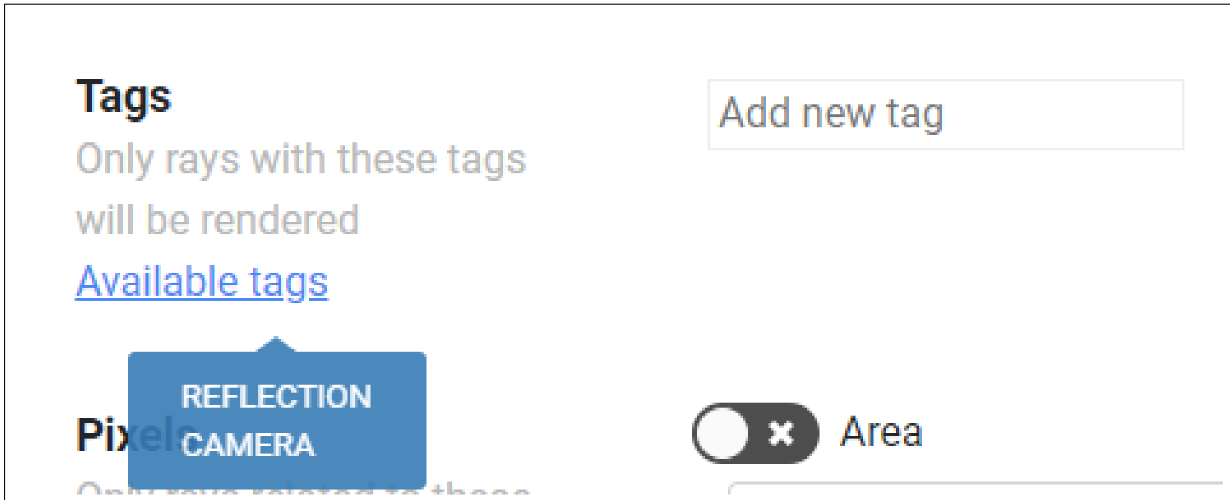
5.4.1. Filtros

En la pestaña *"Filters"* (figura 5.5, número 5 en la figura 5.1), el usuario puede escoger los filtros que serán aplicados sobre el conjunto de rayos cargados. Los filtros que se proveen son los de:

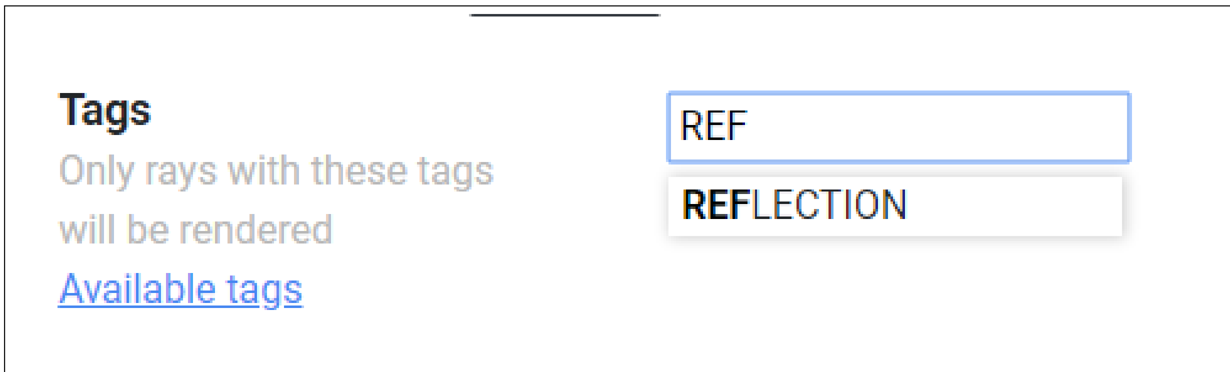
1. Etiquetas
2. Pixeles
3. Largo de rayos
4. Rayos con interseccionens

Filtro por etiquetas

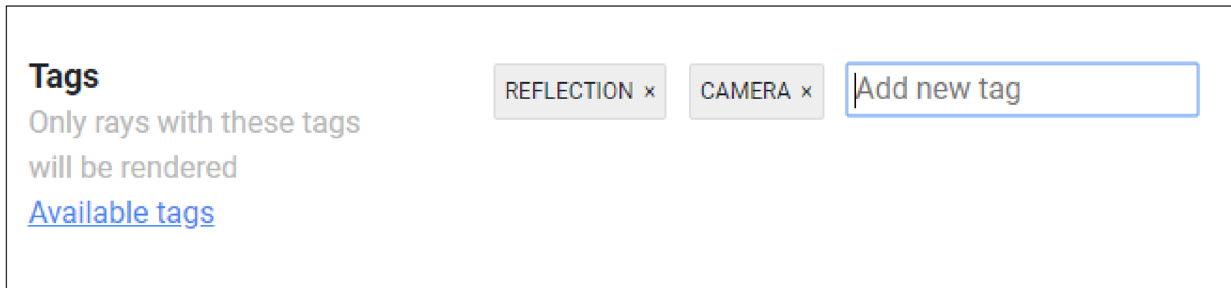
Desde un computador personal, pasar el mouse por encima del enlace *"Available tags"*, muestra todos los tags disponibles en la escena cargada (figura 5.6(a)). Se puede escribir en el campo de texto de la derecha la etiqueta por la cual se quiere filtrar. La aplicación sugiere,



(a) Etiquetas disponibles.



(b) Etiquetas sugeridas.



(c) Etiquetas escritas.

Figura 5.6: Filtrado por etiquetas.

Pixels
Only rays related to these pixels will be rendered
Area
X 20 Y 40
Rays with their pixel set to (x = 20, y = 40)

(a) Un pixel

Pixels
Only rays related to these pixels will be rendered
Area
X_1 20 Y_1 20
X_2 40 Y_2 40
Pixels within the rectangle defined by a top left corner at (x = 20, y = 20) and a bottom right corner at (x = 40, y = 40).

(b) Múltiples pixeles

Figura 5.7: Filtrado por pixeles

además, algún tag, en función de lo que ha escrito hasta ese punto el usuario (figura 5.6(b)). En la figura 5.6(c) se muestra un caso en el que un usuario ingresó dos tags: *"REFLECTION"* y *"CAMERA"*. De acuerdo a lo descrito en la sección 4.6.1, esto significa que se mostrarán todos los rayos que tienen, por separado, ambos tags.

Filtro por pixeles

Este filtro tiene dos modos, *un pixel* y *múltiples pixeles*.

Un pixel Permite definir un pixel con sus dos componentes. Todos los rayos que tengan asignado ese pixel se mostrarán, y el resto no (figura 5.7(a)).

Múltiples pixeles Permite definir un rectángulo de pixeles. Esto se logra indicando cuál es la esquina superior izquierda del rectángulo y cuál es la esquina inferior derecha del mismo. Todos los rayos cuyo pixel asignado esté dentro (inclusivamente) del rectángulo se mostrarán, y el resto no (figura 5.7(b)). En ambos casos, si un rayo no tiene un pixel asignado, entonces no pasa el filtro.

Filtro por largo del rayo

Permite definir una cota superior e inferior para un largo deseado de los rayos que se quieren ver. Si alguna de las cotas no está presente, entonces se considera como infinito, en el caso de la cota superior, y como infinito negativo en el caso de la cota inferior (figura 5.8).

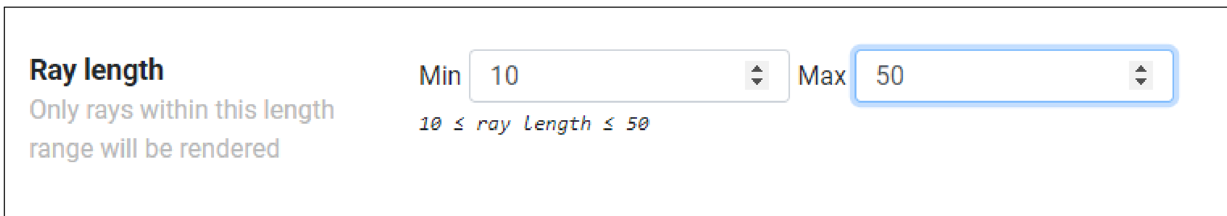


Figura 5.8: Filtro por largo de rayo

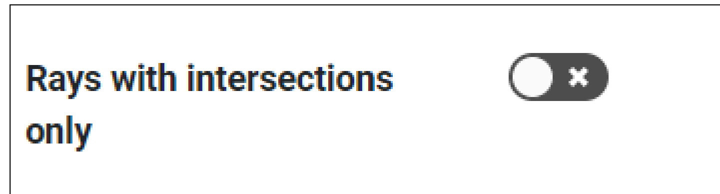


Figura 5.9: Filtrar rayos con intersección

Filtro por rayos con intersecciones Si está activado, muestra todos los rayos que tienen la propiedad *intersection* (figura 5.9).

5.4.2. Inspector

Cuando uno o más rayos son seleccionados desde el visualizador, el inspector (número 3 en la figura 5.1) muestra las propiedades que tienen junto con sus valores respectivos (figura 5.10).

5.4.3. Configuraciones

En panel de configuraciones (número 6 en la figura 5.1) se pueden realizar pequeños ajustes. Se permite

- Cambiar el color base de los rayos, es decir, el color que tienen cuando no han sidos eleccionados (que, por defecto, es verde).
- Ajustar el largo efectivo de los rayos infinitos en el visualizador: el manejo de la aplicación con respecto a mostrar rayos infinitos es el de asignarle un largo finito pero "grande". Si los rayos finitos del usuario son muy largos, entonces es conveniente ajustar el tamaño en este punto.
- Activar o desactivar el *overlay* del visualizador. Por defecto, se provee información por encima del visualizador (figura 5.11). Si se desactiva, entonces se esconde (figura 5.12).

5.4.4. Carga del registro

La aplicación soporta dos medios para cargar el registro con el formato de la sección 4.3.3:

Inspector Loader Filters Settings

Ray

Origin	X	2.000	Y	0.000	Z	0.000
Direction	X	0.000	Y	1.000	Z	0.000
Intersection	X	2.000	Y	4.000	Z	0.000
Length		4.000				
Tags		CAMERA REFLECTION				

Ray

Origin	X	0.000	Y	0.000	Z	0.000
Direction	X	1.000	Y	0.000	Z	0.000
Intersection	X	2.000	Y	0.000	Z	0.000
Length		2.000				

Ray

Origin	X	2.000	Y	0.000	Z	0.000
Direction	X	0.000	Y	-1.000	Z	0.000
Intersection	X	2.000	Y	-3.000	Z	0.000
Length		3.000				
Tags		CAMERA				

Figura 5.10: Inspector con tres rayos seleccionados.

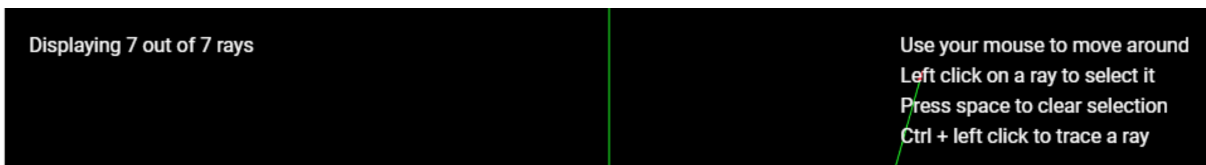


Figura 5.11: Overlay del visualizador.

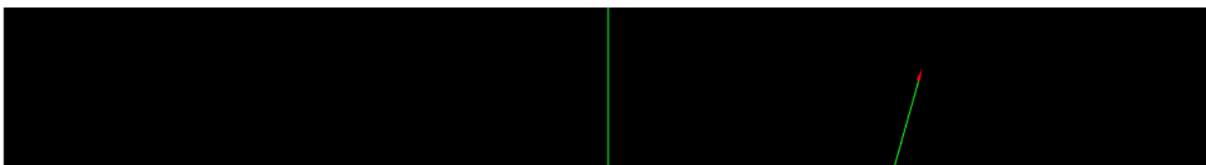


Figura 5.12: La figura 5.11, pero sin el overlay.

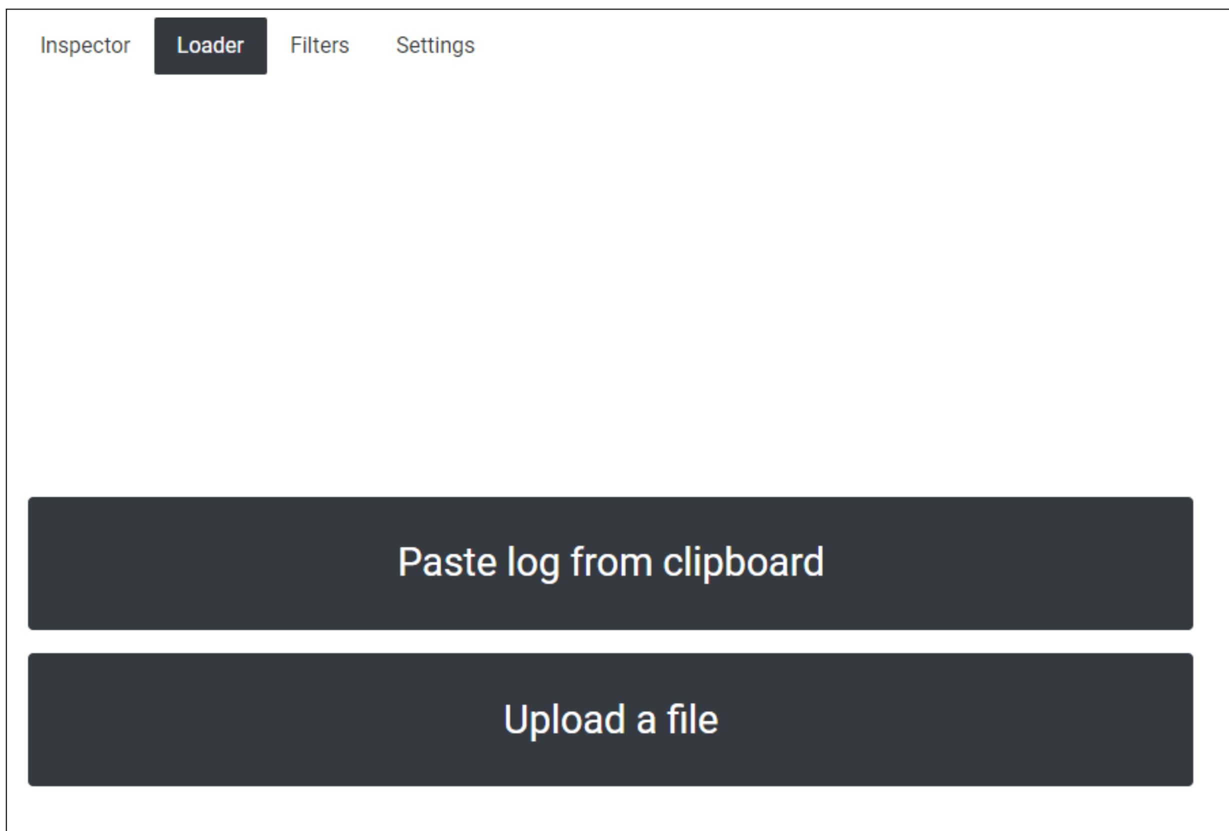


Figura 5.13: Opciones del loader.

1. Escribir el registro en un campo de texto (figura 5.14).
2. Escoger un archivo local con el explorador de archivos de la plataforma (segundo botón de la figura 5.13).

Independiente del método de carga, el registro es validado, y el sistema de visualización es notificado para que renderice los rayos.

5.5. Visualización y renderizado de la escena

Además del libre movimiento de la cámara, el visualizador cuenta con dos funcionalidades esenciales:

1. Selección de uno o más rayos.
2. Seguimiento de un rayo, según lo discutido en la sección 4.6.2.

Las figuras 5.15, 5.16 y 5.17 muestran el visualizador cuando no hay nada seleccionado, cuando un rayo ha sido seleccionado, y cuando a un rayo se le ha aplicado la operación de seguimiento, respectivamente. Todas las imágenes son de la aplicación cargada con la escena que se incluye en el apéndice A.2.

El contenido del visualizador se puede pensar como una serie de imágenes (o **cuadros**)

Inspector | Loader | Filters | Settings

Back

Paste your log here...

```
1  [
2    {
3      "origin": [0, 0, 0],
4      "direction": [0, 1, 0],
5      "intersection": [0, 5, 0],
6      "pixel": [30, 30],
7      "tags": ["REFLECTION"]
8    },
9    {"origin": [0, 0, 0], "direction": [1, 0, 0], "intersection": [2, 0, 0]},
10   {
11     "origin": [2, 0, 0],
12     "direction": [0, 1, 0],
13     "intersection": [2, 4, 0],
14     "tags": ["CAMERA", "REFLECTION"]
15   },
16   {
17     "origin": [2, 0, 0],
18     "direction": [0, -1, 0],
19     "intersection": [2, -3, 0],
20     "tags": ["CAMERA"]
21   },
22   {"origin": [2, -3, 0], "direction": [1, 0, 0], "intersection": [5, -3, 0]},
23   {
24     "origin": [0, 0, 0],
25     "direction": [0, -1, 0],
26     "intersection": [0, -3, 0],
27     "tags": ["CAMERA", "REFLECTION"],
28     "pixel": [20, 20]
29   },
30   {
31     "origin": [0, 0, 0],
32     "direction": [-1, 0, 0],
33     "tags": ["CAMERA"],
34     "pixel": [10, 10]
35   }
36 ]
```

Load

Figura 5.14: Carga por escrito.

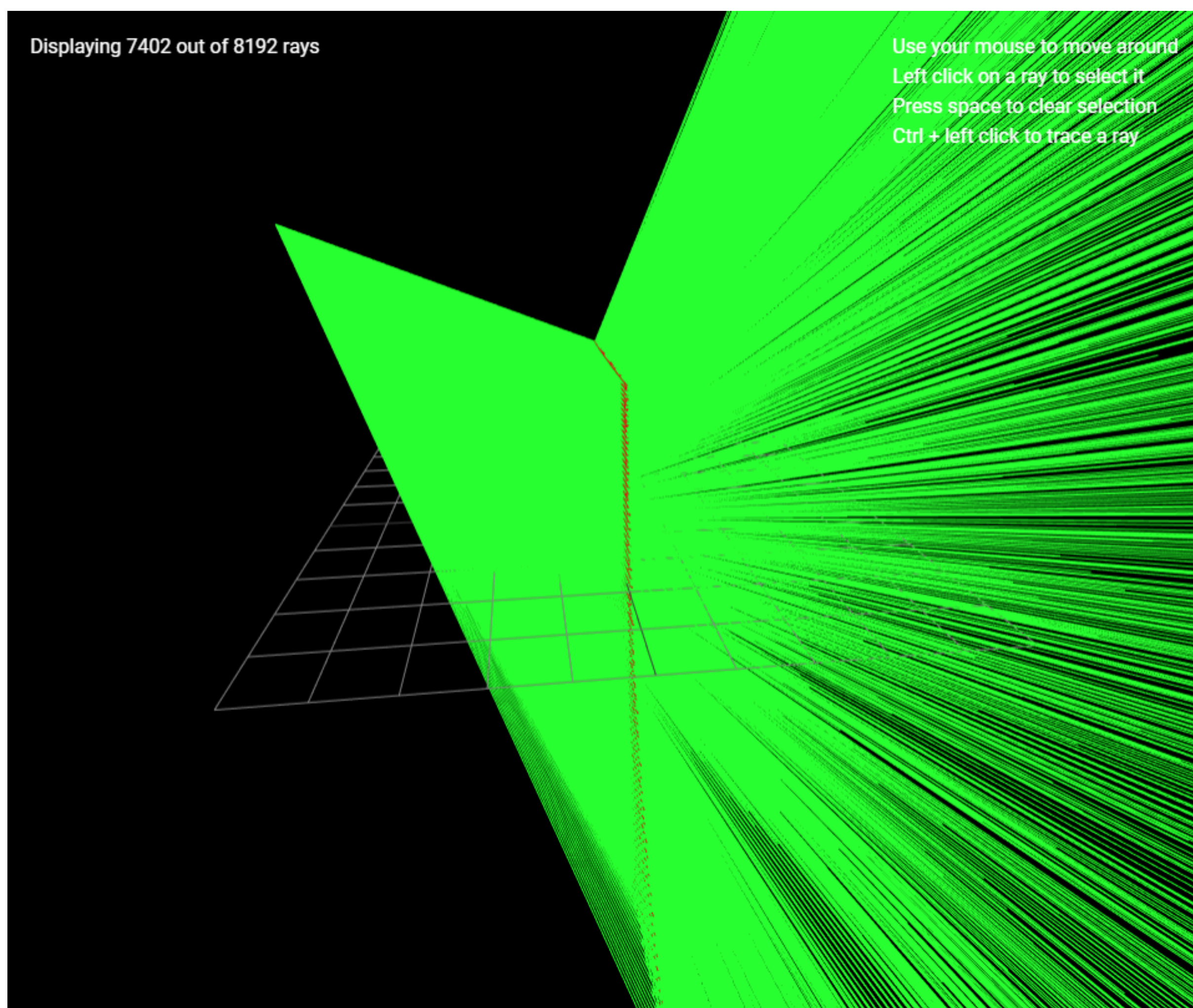


Figura 5.15: Visualizador con la escena A.2 cargada.

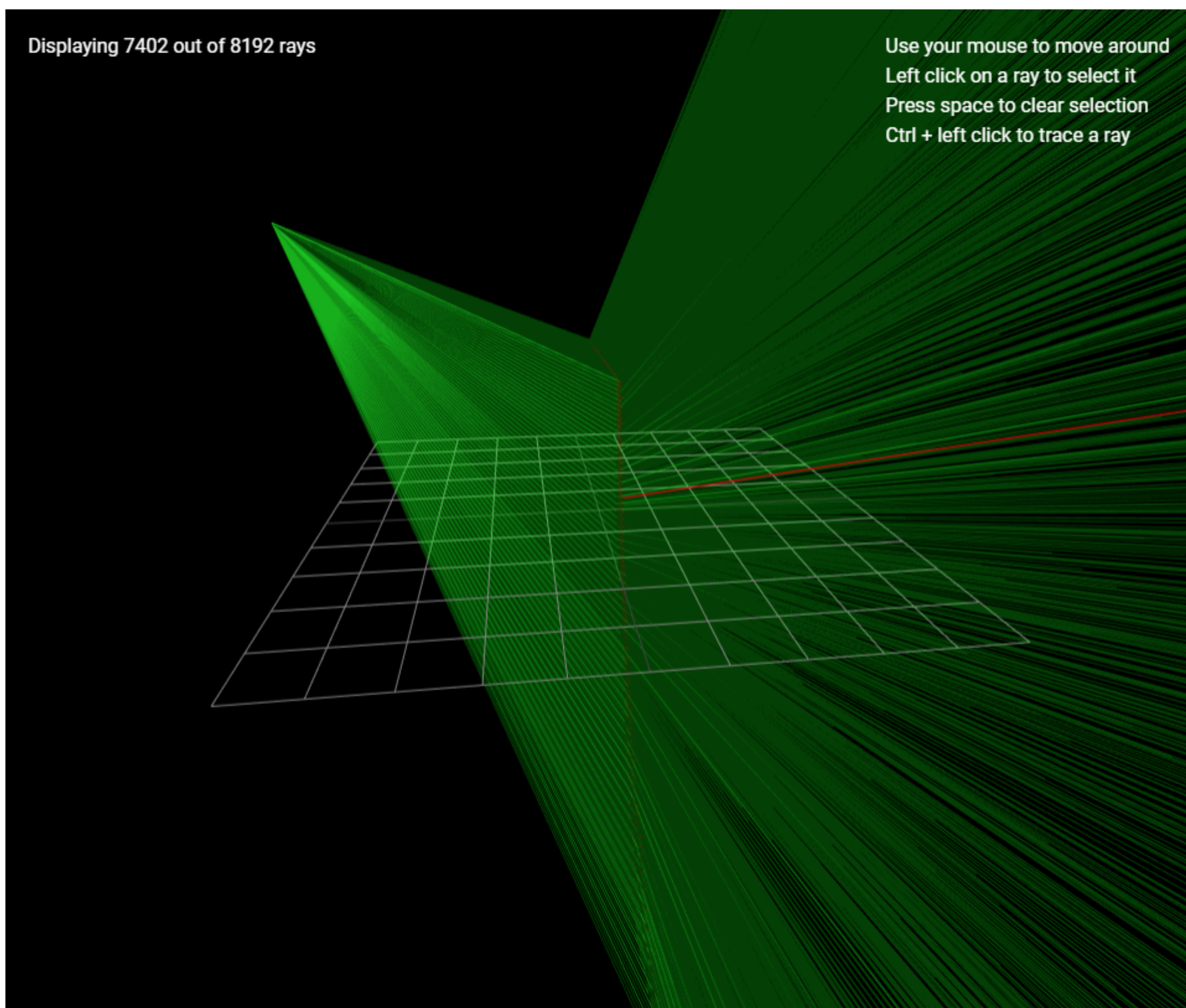


Figura 5.16: Visualizador con la escena A.2 cargada y un rayo seleccionado.

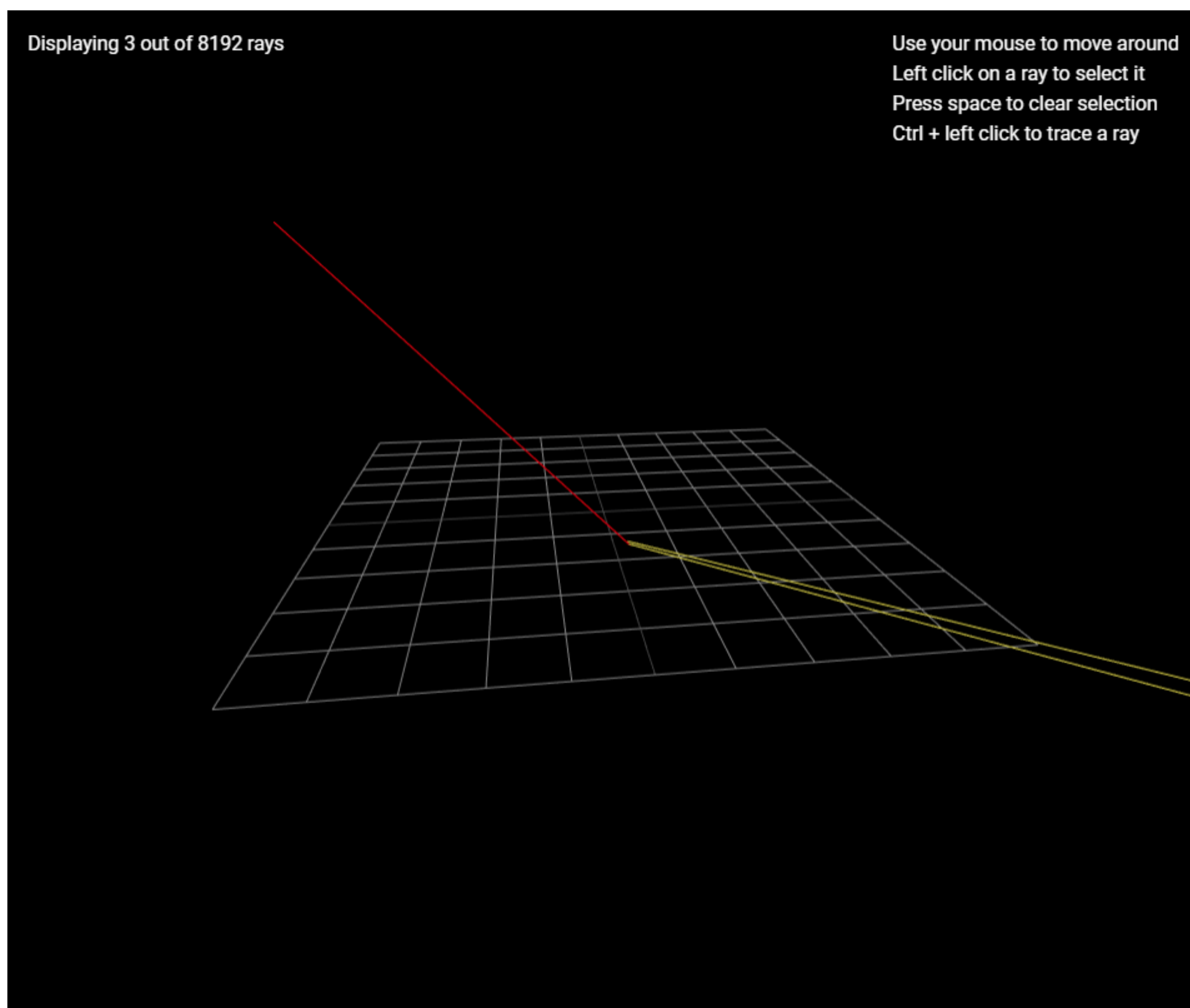


Figura 5.17: Visualizador con la escena A.2 cargada y un rayo al que se le ha aplicado la operación de seguimiento.

que son generadas una tras otra, en este caso por Three.js (a través de WebGL y la GPU). En Three.js se considera la abstracción de **escena**. Una escena puede contener uno o más **objetos**. Finalmente, el renderizador de Three.js produce los cuadros a partir de la información de la escena.

Una implementación directa consistiría en una traducción de los rayos que residen en el modelo de la aplicación (y que fueron cargados por un usuario) a objetos de Three.js. Es decir, para cada rayo, crear un objeto de tipo *línea* (o algún otro modelo apropiado para un rayo). Sin embargo, este acercamiento se vuelve ineficiente, dado que cada objeto renderizado por Three.js implicaría un *draw call* a la GPU. Un *draw call* es una llamada que se le hace a la GPU y que le indica que debe renderizar un conjunto de vértices de alguna forma específica. Por ejemplo, podría ser un conjunto de 3 vértices en forma de triángulo, lo que produciría un triángulo, o podría ser un conjunto de 4 vértices en forma de línea, lo que produciría 2 líneas.

```
for each ray do
|  object ← new Object(LINES, vertices(ray));
|  scene.add(object);
end
```

Algorithm 3: Renderizado ingenuo de las líneas para un cuadro.

En el caso anterior, en el que se considera un objeto de tipo línea por cada rayo, se hace **un** *draw call* por cada **dos** vértices. Los *draw call* son computacionalmente costosos y la cantidad de veces que se hacen se quiere minimizar.

En la implementación de este trabajo se agrupan todos los rayos en un gran conjunto de vértices que constituyen un único objeto de tipo línea **que puede ser renderizado en un solo *draw call***:

```
vertexBuffer ← new Buffer();
for each ray do
|  vertexBuffer.add(vertices(ray));
end
object ← new Object(LINES, vertexBuffer);
scene.add(object);
```

Algorithm 4: Renderizado de los rayos eficiente en draw calls.

Por otro lado, los rayos del visualizador también tienen asociada una malla con forma de cono en la punta que indica la dirección del rayo. Al igual que en el caso anterior, uno podría pensar en crear un objeto de tipo *Mesh* (malla) por cada cono, pero esto no es eficiente en términos de draw calls. Como es una malla, en este caso corresponde aplicar la técnica de **instancing** [12], que consigue el renderizado eficiente de múltiples objetos que comparten la misma malla. Estos objetos pueden tener datos asociados distintos (como distinta posición, distinto color, etc), pero lo importante es que comparten la malla. Si esto ocurre, entonces se puede enviar a la GPU la malla en común en conjunto con toda la información que no se comparte (como las posiciones). A los datos no compartidos se les denomina **datos de instancia** (*instanced data*).

La generación de conos con instancing queda como sigue:

```
instancedData ← new Buffer;
mesh ← new Cone(...);
for each ray do
    instancedData.add(computeInstancedData(ray));
end
object ← new Object(instancedData, mesh);
scene.add(object);
```

Algorithm 5: Generación de buffers de conos con instancing.

Lo descrito hasta este punto conforma una implementación eficiente del renderizado de rayos estáticos. El problema cuya solución se presenta a continuación considera rayos cuyas propiedades cambian, que es lo que ocurre en la práctica en la aplicación.

Una acercamiento ingenuo que resuelve la reactividad a rayos que cambian sus propiedades es el de reconstruir los buffers que se envían a la GPU en **cada cuadro (frame)**, y tendría la forma del algoritmo 6.

```
for each frame do
    Aplicar el algoritmo 4 para construir el buffer de las líneas;
    Aplicar el algoritmo 5 para construir el buffer de los conos;
end
Enviar ambos buffers a la GPU;
```

Algorithm 6: Renderizado eficiente para contenido estático.

Si bien esto funciona, y los rayos serán constantemente actualizados en el visualizador con lo más reciente que se encuentre en el estado hasta ese momento, se puede hacer mejor. La razón es que no es necesario reconstruir los buffers todos los cuadros, porque los rayos no cambian necesariamente todos los cuadros.

Cuando un usuario selecciona un rayo, su color cambia. Asimismo, la transparencia de los rayos no seleccionados también cambia. En general, existe una serie de eventos (como la selección), que gatillan cambios en las propiedades de los rayos y, en consecuencia, en los buffers que deben ser enviados a la GPU.

En la implementación concreta se propone que se notifique, mediante acciones, al renderizador cuando hay rayos que tienen que ser reconstruidos. En la jerga utilizada en el código, se dice que los rayos están ***dirty***. Luego, el algoritmo 6 queda como el algoritmo 7.

```
for each frame do
    if rays are dirty then
        Aplicar el algoritmo 4 para construir el buffer de las líneas;
        Aplicar el algoritmo 5 para construir el buffer de los conos;
    end
end
Enviar ambos buffers a la GPU;
```

Algorithm 7: Rendering eficiente para contenido dinámico.

Capítulo 6

Validación y pruebas

Para para poner a prueba la efectividad de la herramienta e ilustrar un modo de uso propuesto, se preparan cinco casos de estudio. Cada caso de estudio describe un problema que tiene un raytracer, observable mediante imágenes generadas por el mismo y que se incluyen. Se considera la perspectiva de un desarrollador que no ha visto la imagen correcta y que trata de hacerse una idea a través de la descripción de la escena.

Además, se hacen pruebas con usuarios que han desarrollado raytracers. La metodología del experimento consiste en proveer a los participantes de:

- La descripción de la escena.

- Una imagen sin errores generada por un raytracer a partir de la descripción de la escena.

- Una imagen con uno o más errores generada a partir de la misma descripción.

- Un registro de rayos compatible con la plataforma emitido por el raytracer cuando generó la imagen con errores.

La distribución de la prueba se realiza por correo, y se aceptan respuestas por un período de dos semanas. Cada participante descarga el contenido mencionado, accede a una versión online de la plataforma y procede a completarla.

Para recopilar los resultados, se incluye además un enlace a una encuesta en línea a través de Google Forms [2]. La encuesta incluye, para cada caso, la pregunta *¿Cuál crees que es el bug en la escena?*. A partir de las respuestas a esta pregunta se puede concluir la **cantidad** de participantes que resolvieron cada problema.

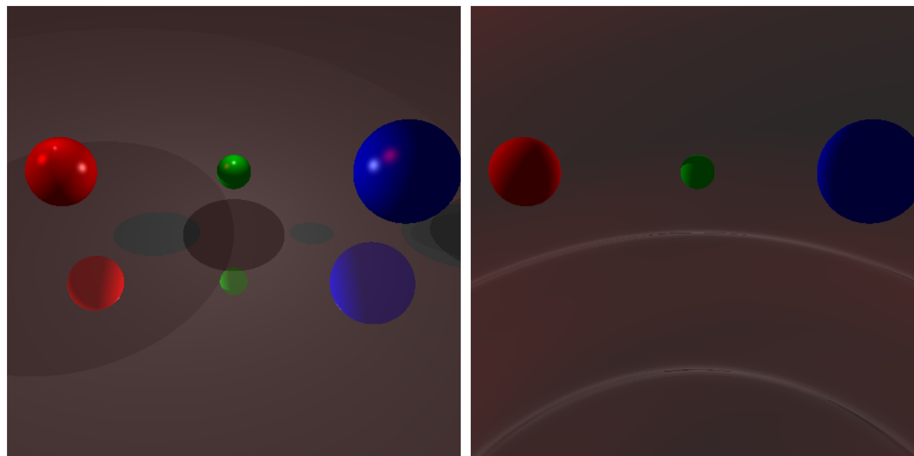
El desarrollo del experimento es flexible, en el sentido de que se permite bastante tiempo y que no se exige responder todas las preguntas. La encuesta fue respondida por **4** personas, y se incluye en la sección que sigue la cantidad de personas que resolvió el problema en cada caso.

6.1. Casos de estudio

6.1.1. Caso 1

Descripción de la escena: apéndice A.1.
Etiquetas disponibles:

- **CAMERA:** Los rayos que son lanzados en un inicio desde la cámara.
- **OBJECT_1:** Los rayos que intersectan con la esfera grande.
- **OBJECT_2, OBJECT_3 y OBJECT_4:** Los rayos que intersectan con las tres esferas pequeñas.



(a) Imagen correcta (b) Imagen incorrecta

Figura 6.1: Caso 1

Este es el caso de motivación de la sección 3.2. De la descripción de la escena, se puede construir una idea de lo que tiene que tener la imagen correcta: hay cuatro esferas en la escena, tres de ellas tienen radio menor a 1 y la otra es de radio 100. Es decir, son tres esferas relativamente pequeñas con respecto a otra grande.

Tras analizar las posiciones, se observa que las tres esferas pequeñas **están alineadas en el eje X**, con las otras coordenadas fijas en $Y = 0$, $Z = 1$. La cámara está ubicada en $(0, 5, 5)$ y está apuntando a $(0, 0, 0)$. En conjunto con el ángulo de apertura de la cámara, que es de 45 grados, se concluye que las tres esferas pequeñas deberían estar en el campo de visión de la cámara.

Por otro lado, la esfera grande es de radio 100 y está ubicada en $(0, 0, -100)$. Recordando que la cámara está en $(0, 5, 5)$, se determina que esta esfera no alcanzará a apreciarse por completo en la imagen por ser demasiado grande.

Tras observar los materiales, se destaca que la esfera grande es un espejo, pues tiene el material "white_mirror". La intuición final es, entonces, que las esferas pequeñas deberían reflejarse en la esfera grande.

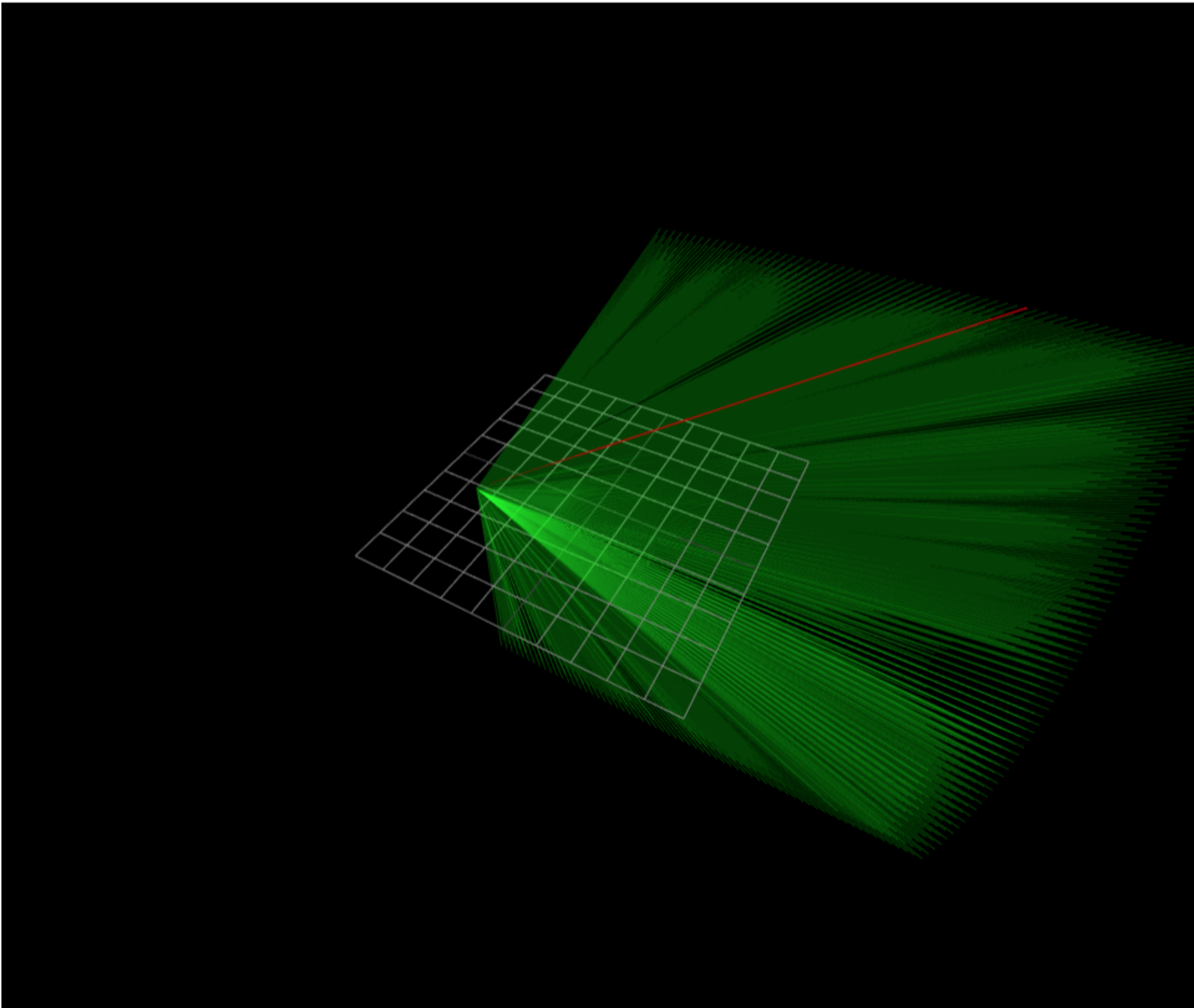


Figura 6.2: Caso 1 tras carga inicial

La imagen 6.1(b) muestra la imagen generada por el raytracer que se está depurando. De inmediato se hace notar la ausencia de elementos reflejados. Además, llaman la atención dos contornos circulares de un color grisáceo. La especularidad de las esferas (por el material "white_bph_100") tampoco se hace presente. La fuente del error no es clara.

Tras cargar los rayos en la plataforma, se obtiene lo de la figura 6.2. No se alcanza a apreciar la forma de las esferas, lo que se considera extraño. Se procede, entonces, a filtrar por los rayos que intersectan con alguna de las esferas.

Se intenta, en primer lugar, filtrar por alguna de las esferas pequeñas. En particular, por el tag "OBJECT_2" y resulta la imagen de la figura 6.3. El resultado parece sensato. Lo que se intenta a continuación es filtrar por la esfera grande (con el tag "OBJECT_1"). El efecto de lo anterior es la figura 6.4.

Aquí llama de inmediato la atención que los rayos parecen ser muy largos en comparación con los rayos de la esfera pequeña. Según la descripción de la escena, las esferas pequeñas no

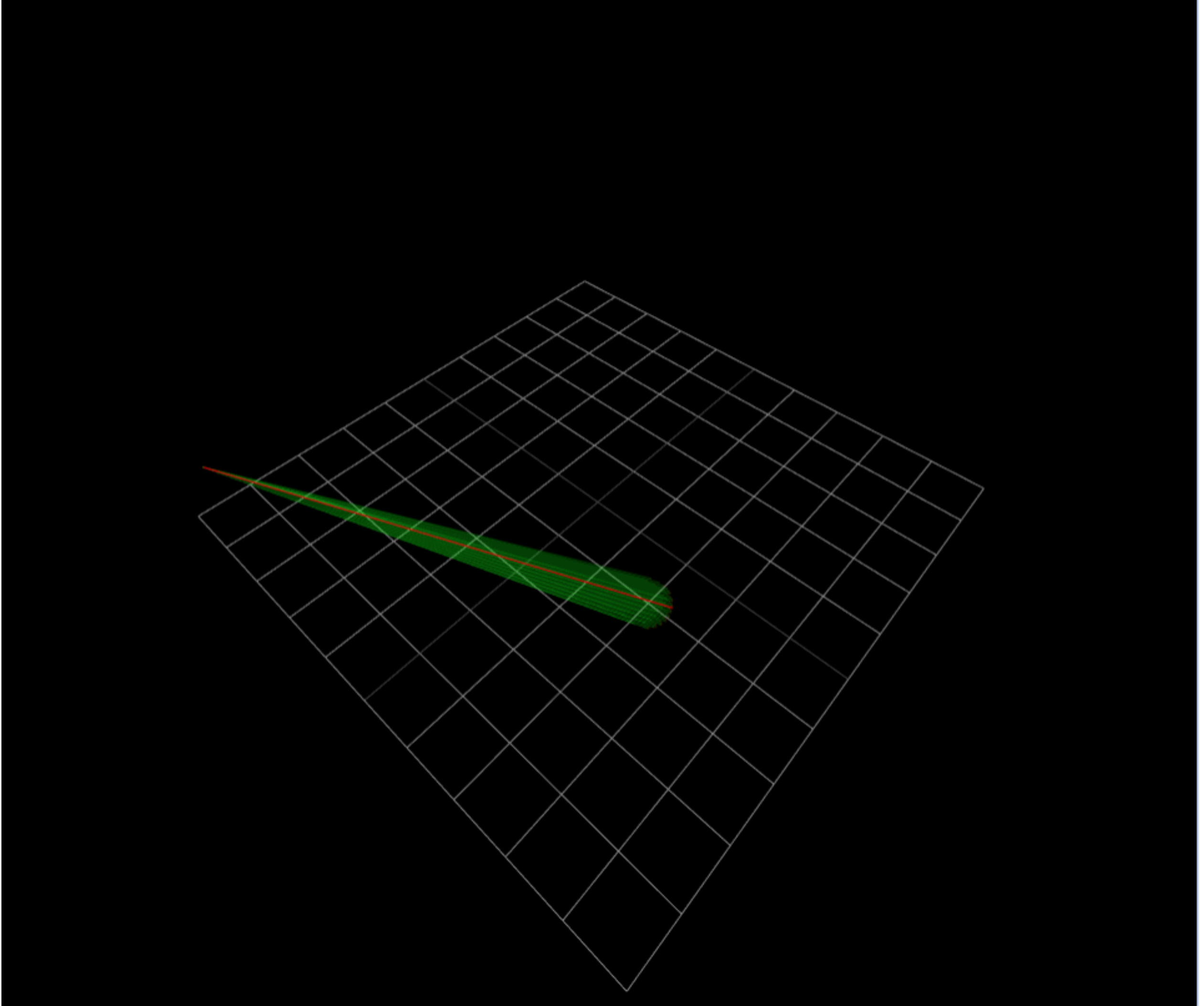


Figura 6.3: Caso 1 tras filtrar por una de las esferas pequeñas.

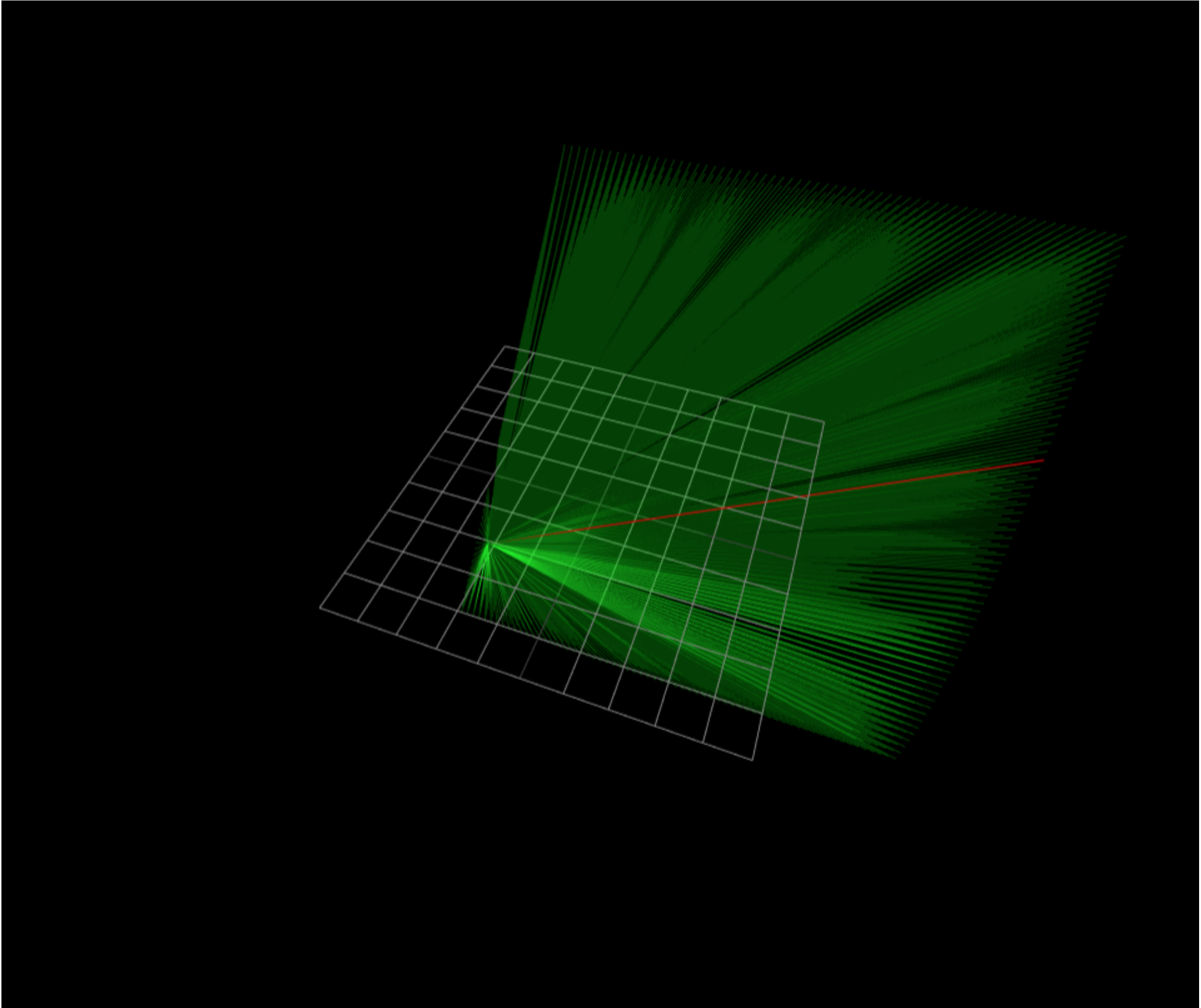


Figura 6.4: Caso 1 tras filtrar por la esfera grande.

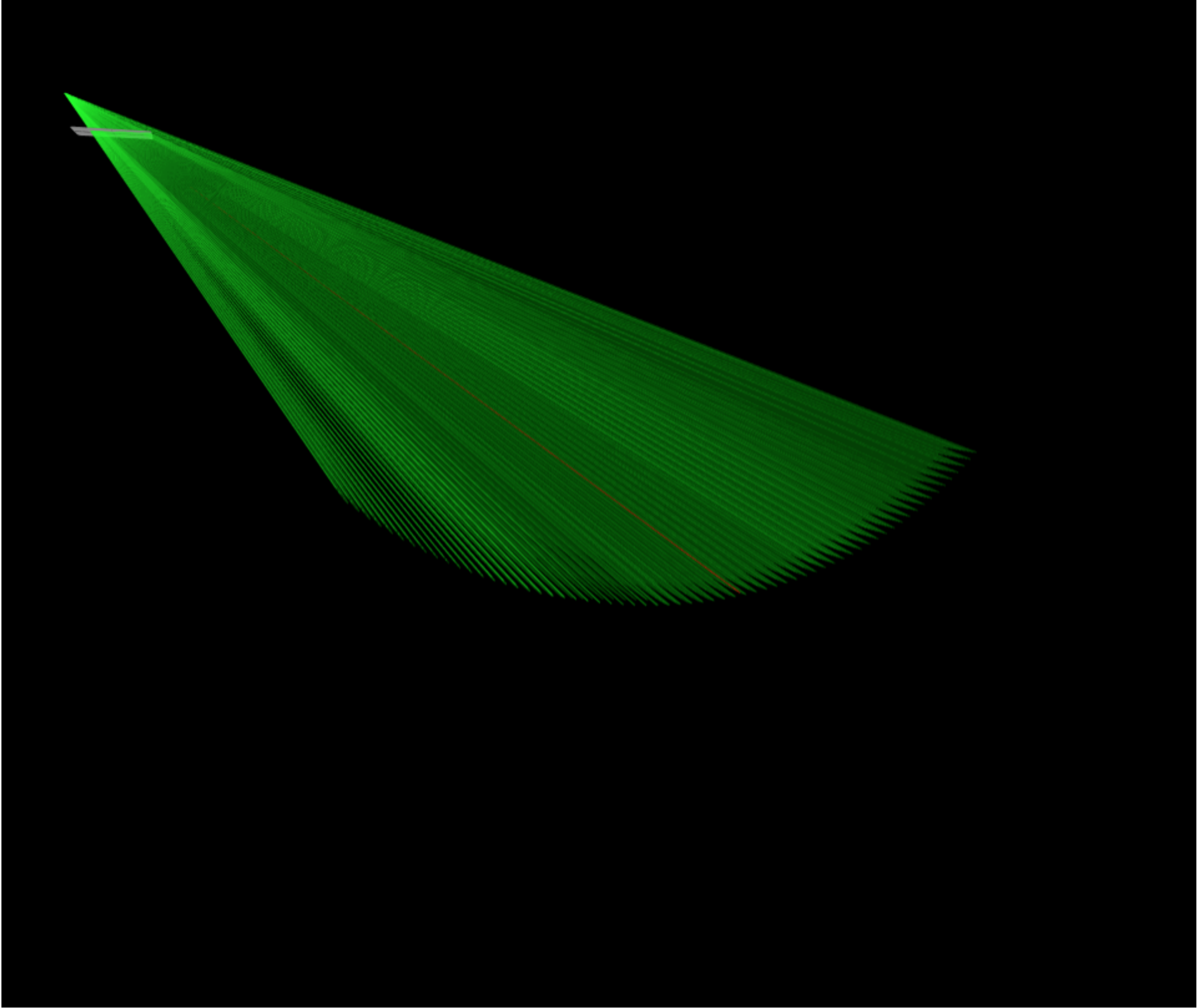


Figura 6.5: Caso 1 tras mover la cámara.

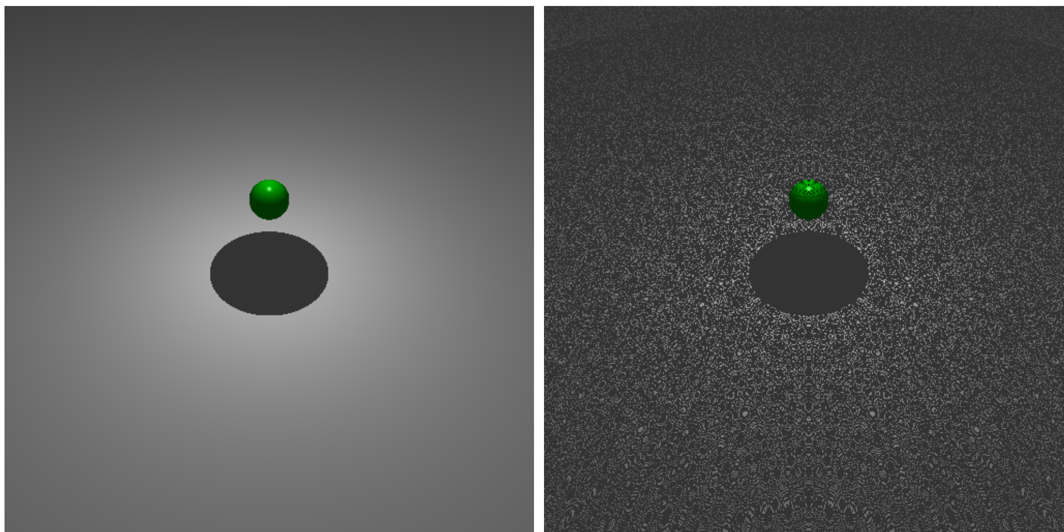
están tan lejos de la esfera grande. Esto podría inspirar a pensar que hay un problema con la forma en la que se calcula la intersección con las esferas. Inspeccionando un poco más de cerca el filtrado anterior, y posicionando la cámara en otro sitio, se obtiene lo de la figura 6.5.

Recordando que el resultado de intersectar una recta con una esfera puede tener dos puntos como solución, y en conjunto con la forma cóncava de los puntos de intersección de la figura 6.5, hay suficiente información para sospechar que el problema es que se está ocupando el punto de intersección más lejano con respecto al punto de lanzado, en lugar que el más cercano, como debería ser.

6.1.2. Caso 2

Descripción de la escena: apéndice A.2.
Etiquetas disponibles:

- **CAMERA:** Rayos que son lanzados al inicio desde la cámara.
- **OBJECT_1:** Rayos que intersectan con la esfera grande.
- **OBJECT_2:** Rayos que intersectan con la esfera pequeña.
- **POINT_LIGHT_SHADOW_RAY:** Rayos de sombra para la luz puntual que está en la escena. Se lanzan desde un punto en dirección a la luz para saber si existe un obstáculo entre ese punto y la luz. Si existe un obstáculo, entonces a ese punto le corresponde sombra.

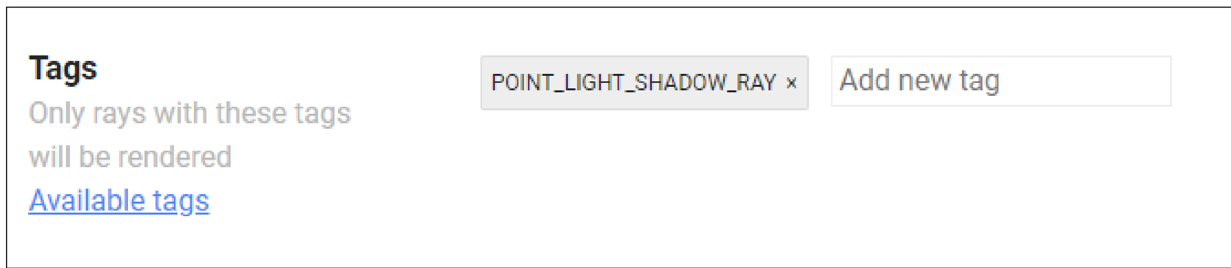


(a) Imagen correcta (b) Imagen incorrecta

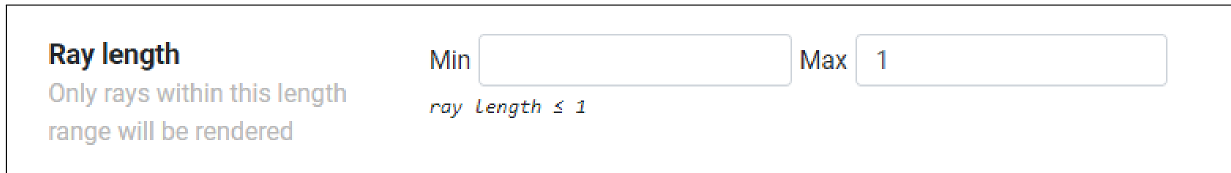
Figura 6.6: Caso 2

La descripción de la escena indica que hay dos esferas, una grande y una pequeña.

Similar al caso anterior, el hecho de que la cámara esté ubicada en $(0, 5, 5)$, que la esfera



(a) Etiquetado para rayos de sombra.



(b) Filtro por largo del rayo.

Figura 6.7: Pestaña de filtros del caso 2.

pequeña, de radio 0.2, esté en $(0, 0, 1)$, y que la esfera grande, de radio 100, esté en $(0, 0, 1)$, se puede intuir que en la imagen final se verá la esfera pequeña en su totalidad, mientras que de la esfera grande se apreciará una porción.

De los materiales, se espera que la esfera pequeña sea de color verde (porque tiene el material "*green_lambert*"), y que la esfera pequeña sea de color blanco (porque tiene el material "*white_lambert*"). La figura 6.6(b) muestra la imagen en cuestión.

El problema es claro a la vista, y es que existe una serie de pequeños puntos negros sin causa aparente. Tras cargar el registro en el visualizador se obtiene la imagen de la figura 6.8. Se intenta, en primer lugar, quitar de la vista los rayos infinitos (sin intersección) utilizando el filtro correspondiente. El resultado es la figura 6.9.

Se sospecha ahora de los rayos de sombra, porque los puntos negros podrían ser sombras mal asignadas. Utilizando el etiquetado (figura 6.7(a)), se muestran el resultado con sólo los rayos de sombra (figura 6.10). Esta imagen también parece estar bien, pues los rayos que se observan tienen la forma de rayos que están calculando las sombras de ambas esferas. Se decide entonces filtrar por este tipo de rayos (figura p), y el resultado se ve razonable. Por otro lado, como hay muchos rayos juntos, se pierde precisión en la visualización. Para limpiar la escena, se filtra por largo del rayo (figura 6.7(b)). El resultado es la figura 6.11.

Destaca el hecho de que existan rayos que parecen ser muy cortos. Al inspeccionar uno de estos, la herramienta indica que son de largo 0, lo que se interpreta como que son rayos de largo muy pequeño (i.e, con el punto de origen e intersección muy cercanos entre ellos). Con esto, ya se puede sospechar fuertemente que los puntos negros sean *shadow acne*, que es efectivamente el problema.

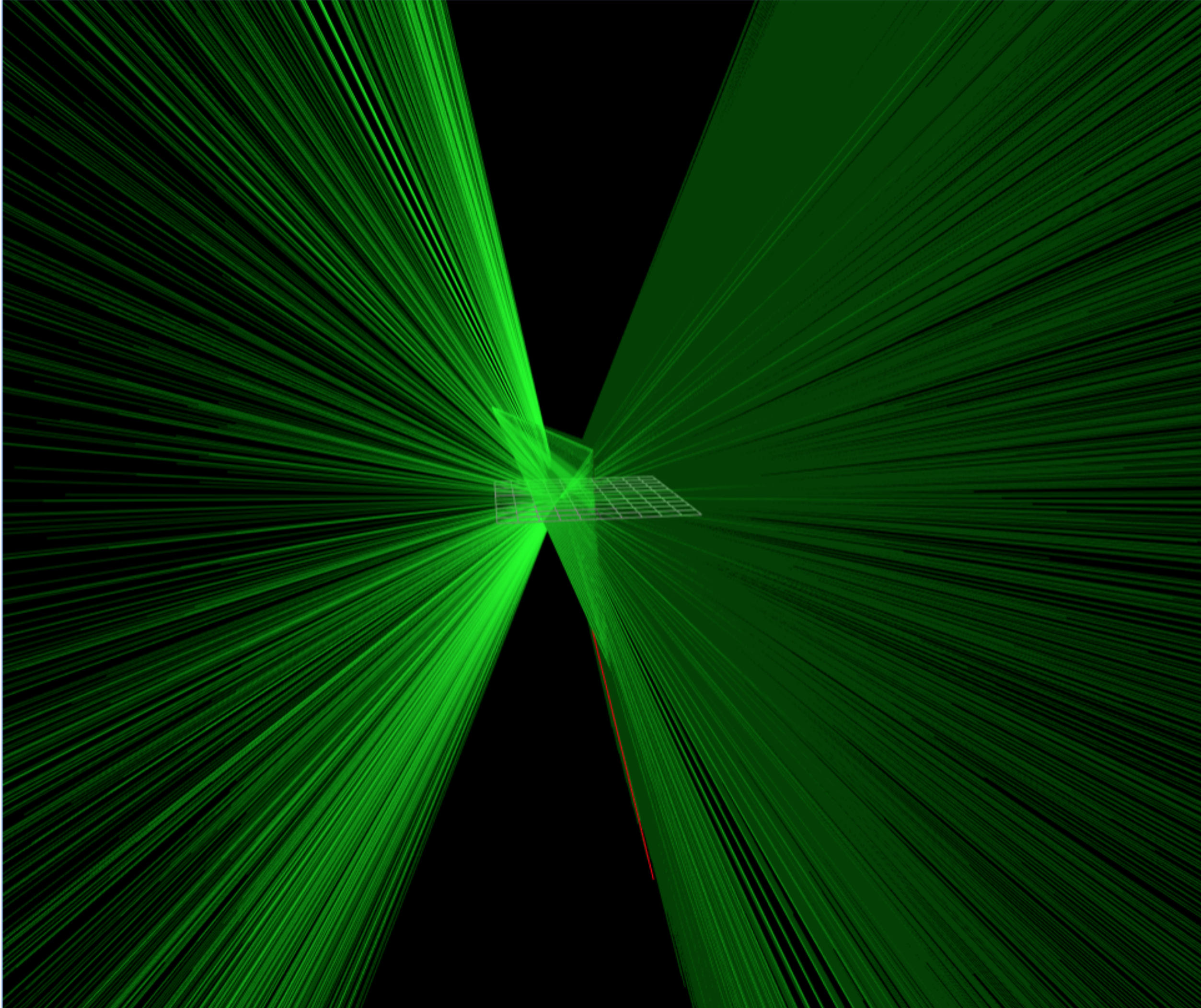


Figura 6.8: Caso 2 tras carga inicial.

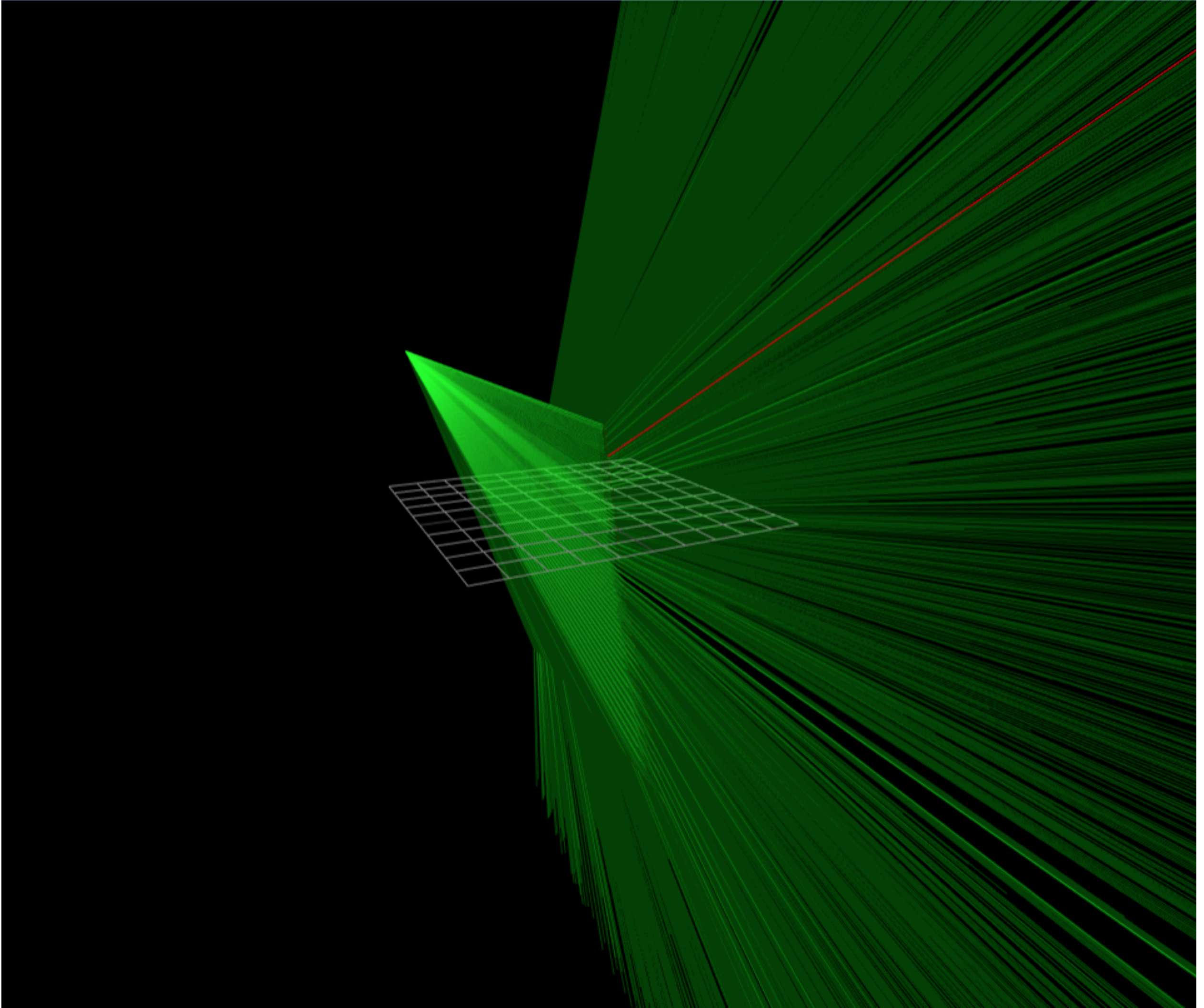


Figura 6.9: Caso 2 mostrando sólo rayos con intersecciones.

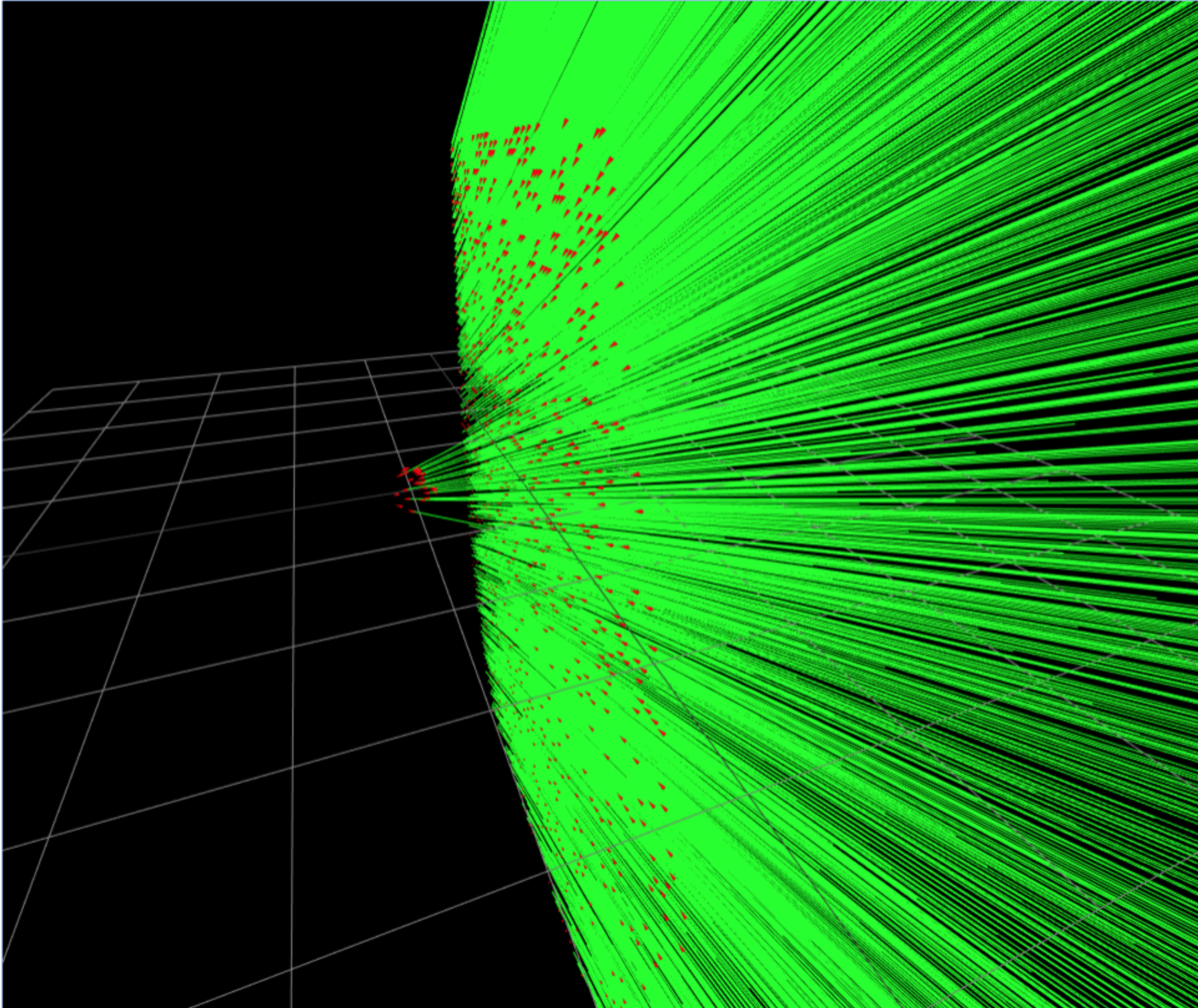


Figura 6.10: Caso 2 mostrando sólo rayos de sombra.

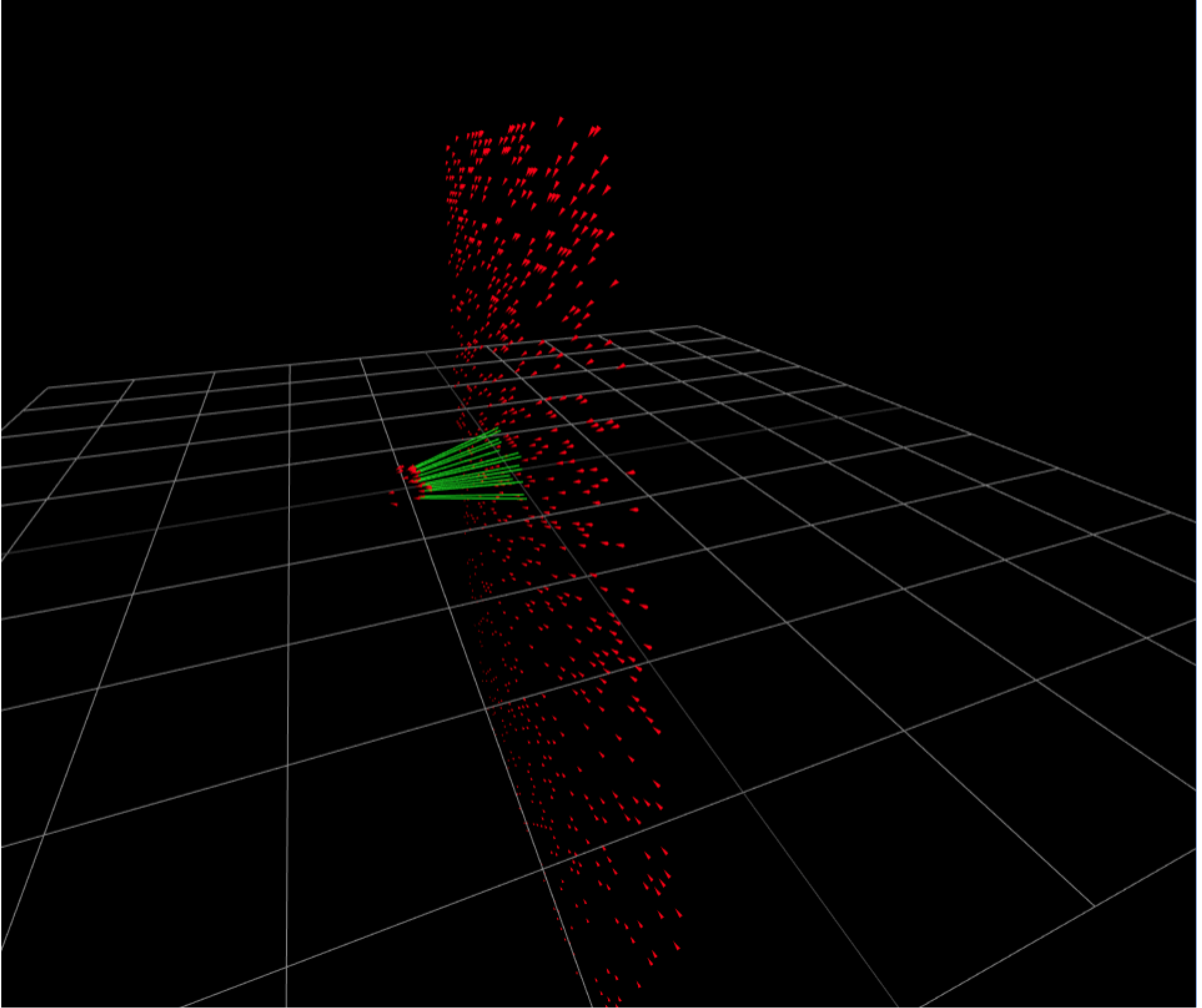
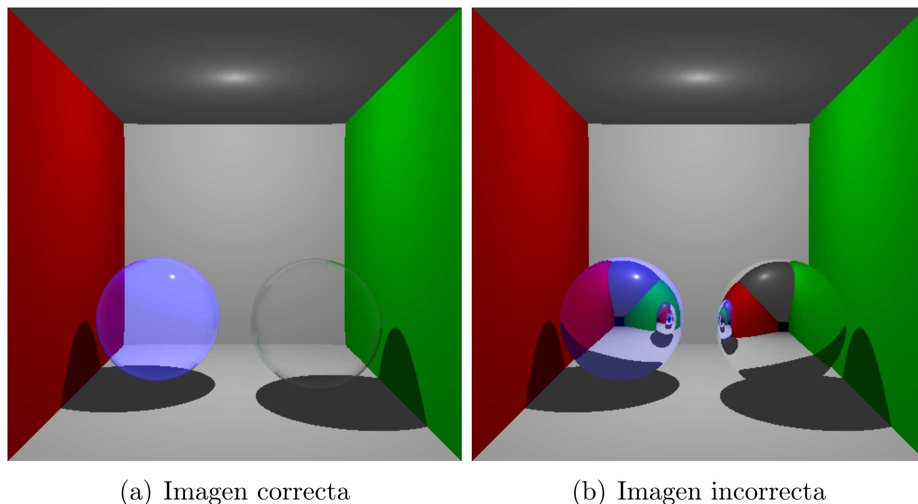


Figura 6.11: Caso 2 tras aplicar el filtro por largo de rayo.

6.1.3. Caso 3

Descripción de la escena: apéndice A.3.
Etiquetas disponibles:

- **CAMERA:** Los rayos que son lanzados en un inicio desde la cámara.
- **OBJECT_4 y OBJECT_7:** Los rayos que intersectan con las esferas pequeñas.
- **OBJECT_1, OBJECT_2, OBJECT_3, OBJECT_5 y OBJECT_6:** Los rayos que intersectan con las esferas grandes que actúan como murallas.
- **REFRACTION:** Los rayos que se lanzan como producto de una refracción.
- **REFLECTION:** Los rayos que se lanzan como producto de una reflexión.



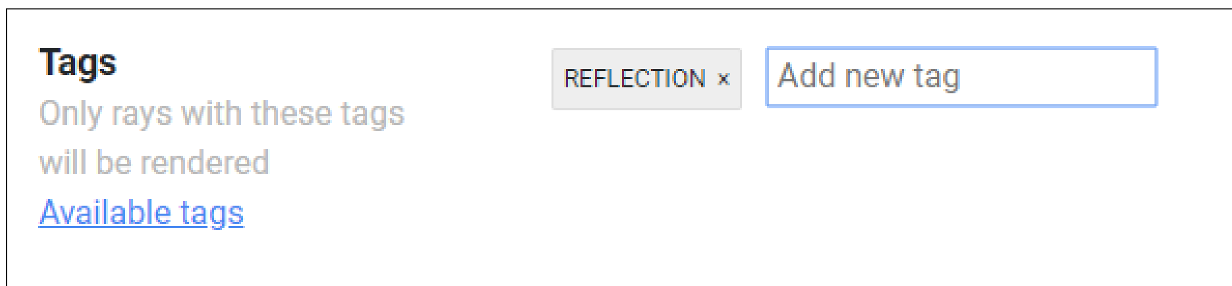
(a) Imagen correcta (b) Imagen incorrecta

Figura 6.12: Caso 3

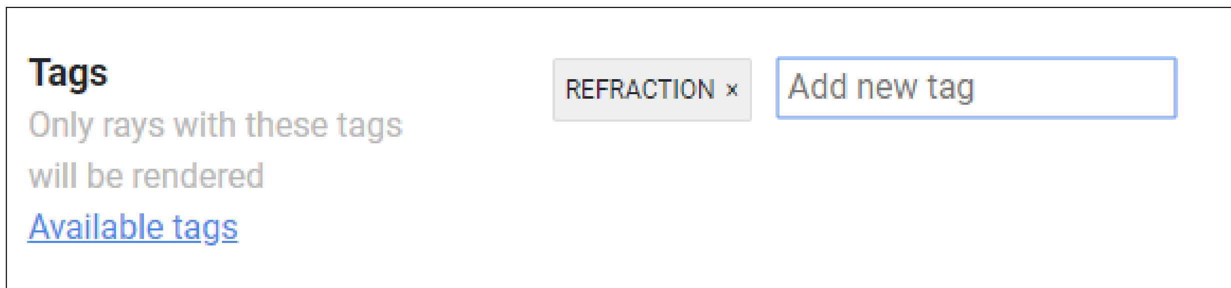
De la descripción de la escena, se concluye que hay dos esferas relativamente pequeñas y cinco esferas grandes. Las cinco esferas grandes están posicionadas de forma tal que forman murallas levemente curvas por la cercanía con la cámara. Las esferas pequeñas tienen un material refractivo ("*air*"). Lo que se espera intuitivamente es que se vean las dos pequeñas rodeadas por las cinco murallas, y el problema apreciable es que la refracción no está funcionando correctamente (figura 6.12(b)).

Tras cargar el registro en el visualizador se obtiene lo de la figura 6.14. Los rayos parecen estar bien en una primera impresión. Si se filtra por los rayos de reflexión (figura 6.13(a)), i.e, rayos que fueron lanzados como producto de una **reflexión**, la visualización queda como la figura 6.15. Los rayos tienen una forma sensata en este punto. Si se filtra por los rayos de refracción (figura 6.13(b)), i.e, los rayos que fueron lanzados como producto de una **refracción**, se obtiene la figura 6.15.

Lo que destaca es que los rayos de refracción son muy parecidos a los rayos de reflexión, y esto es incorrecto. Lo que implícitamente sugiere la plataforma en este caso es revisar la lógica de lanzado de rayos de refracción.



(a) Filtro mediante etiquetado para rayos de reflexión



(b) Filtro mediante etiquetado para rayos de refracción

Figura 6.13: Filtros para para el caso 3.

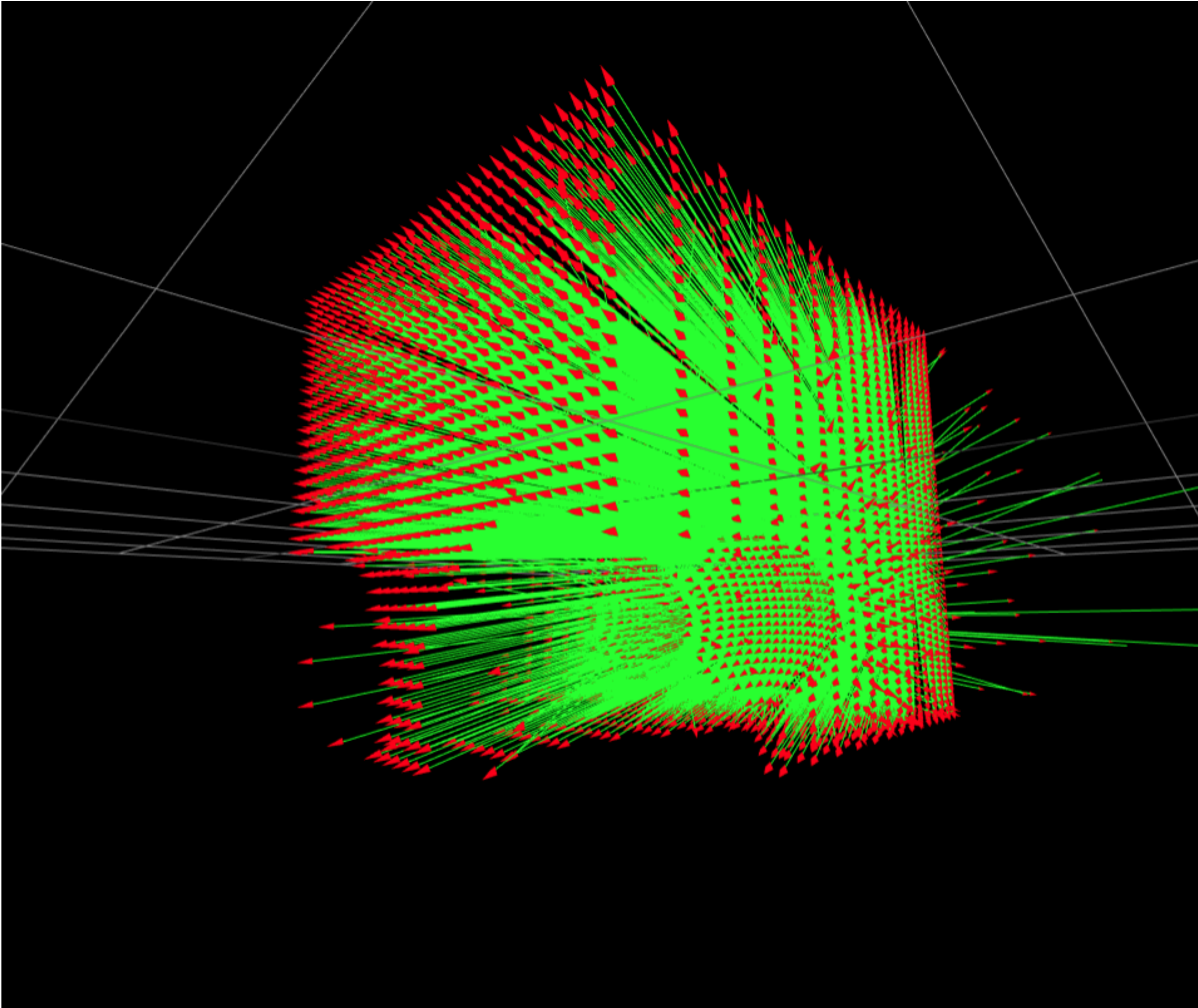


Figura 6.14: Caso 3 tras carga inicial.

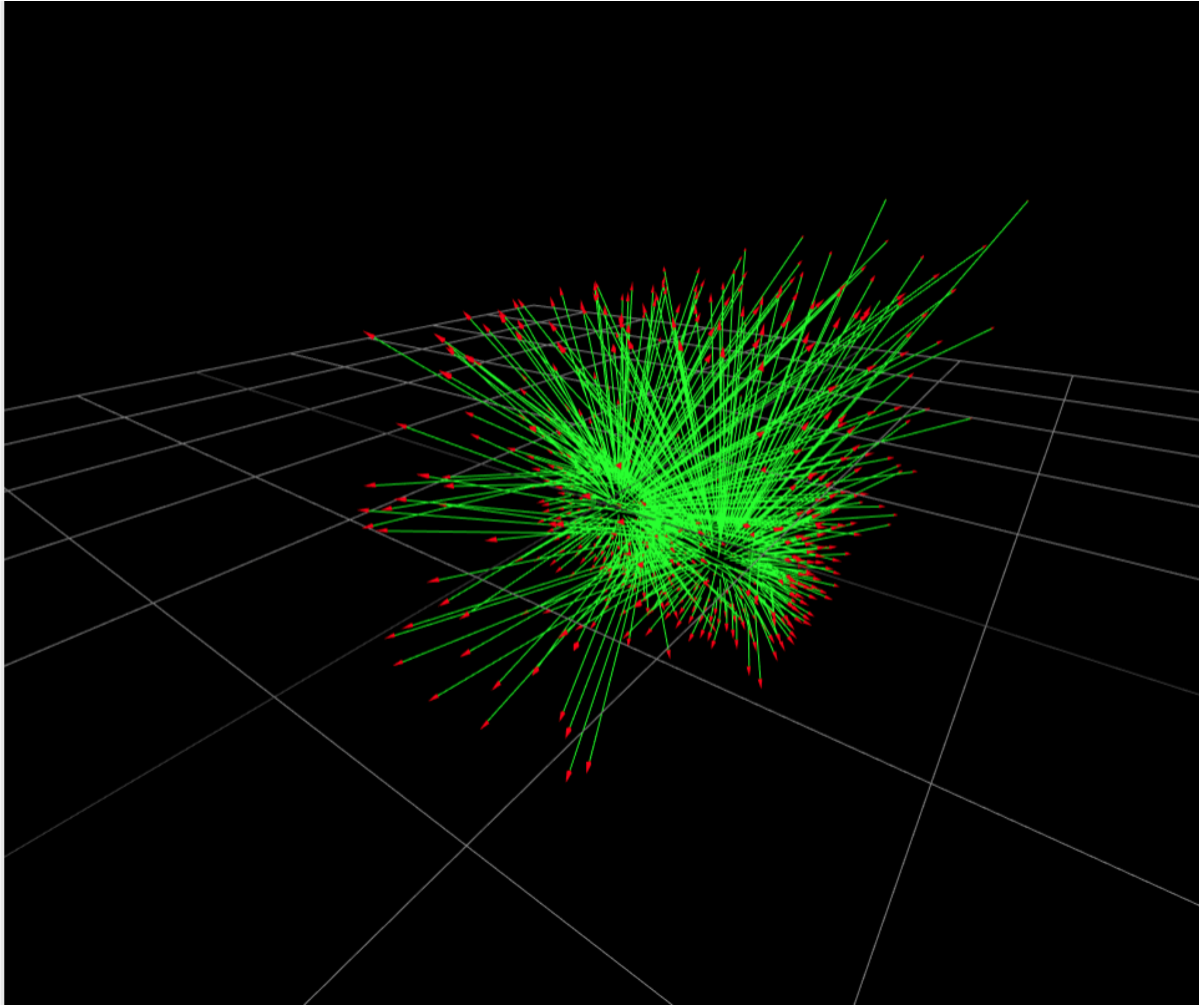


Figura 6.15: Caso 3 mostrando rayos de reflexión.

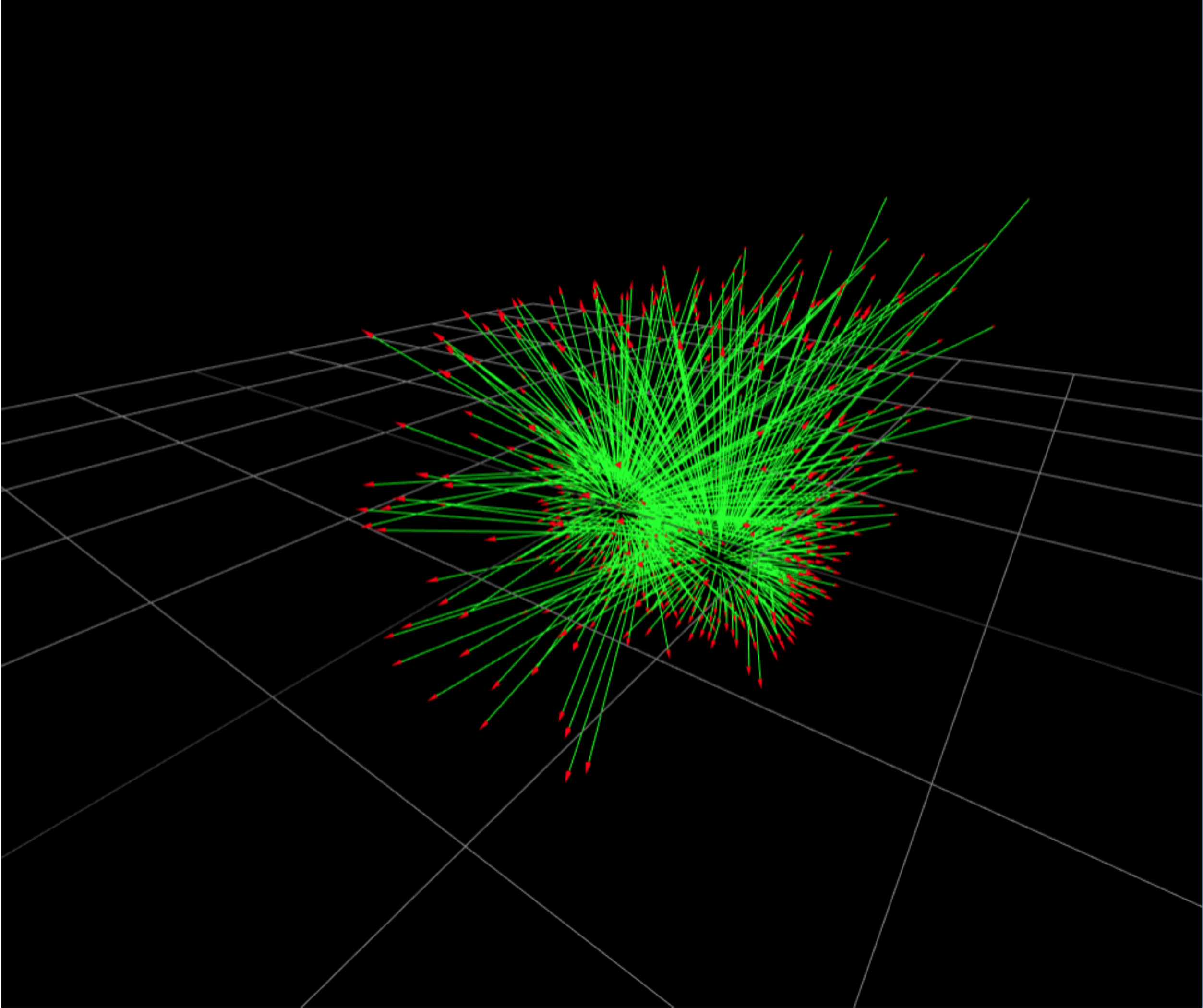
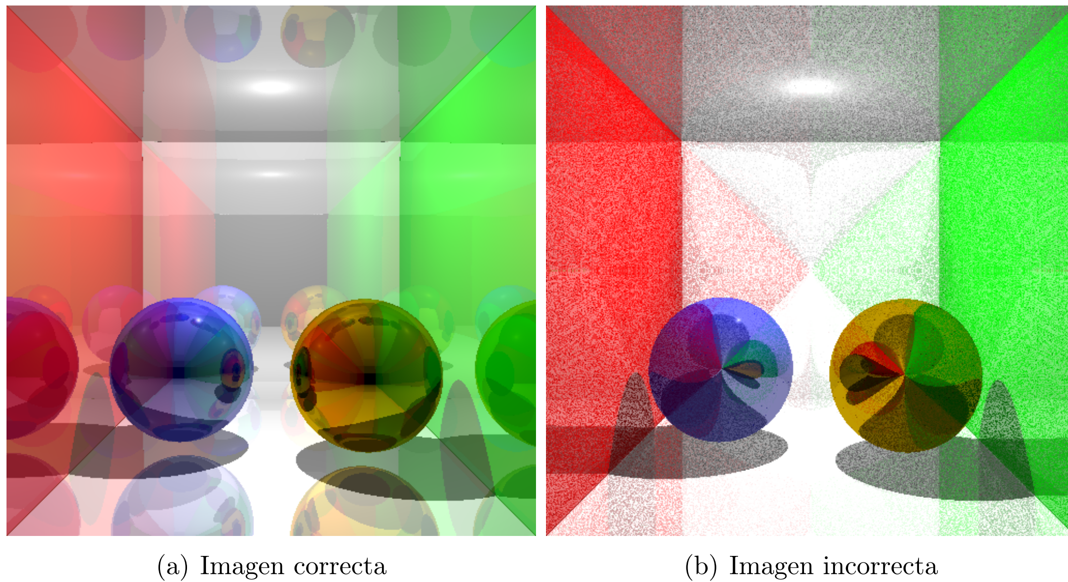


Figura 6.16: Caso 3 mostrando rayos de refracción.

6.1.4. Caso 4

Descripción de la escena: apéndice A.4.
Etiquetas disponibles:

- **CAMERA:** Los rayos que son lanzados en un inicio desde la cámara.
- **OBJECT_4 y OBJECT_7:** Los rayos que intersectan con las esferas pequeñas.
- **OBJECT_1, OBJECT_2, OBJECT_3, OBJECT_5 y OBJECT_6:** Los rayos que intersectan con las esferas grandes que actúan como murallas.
- **REFLECTION:** Los rayos que se lanzan como producto de una reflexión.



(a) Imagen correcta (b) Imagen incorrecta

Figura 6.17: Caso 4

La estructura de esta escena es similar al caso anterior. La diferencia radica en los materiales utilizados para las esferas pequeñas. En este caso, ambas esferas tienen un material reflexivo. En la figura 6.22(b), la esfera de la izquierda tiene el material "*gray_mirror*", mientras que la de la derecha tiene el material "*gold_mirror*".

El error que se aprecia no da un indicio claro de su naturaleza. La figura 6.18 muestra el registro cargado en el visualizador. Para limpiar la escena, se filtra, en primer lugar, por todos los rayos con intersecciones, y se obtiene la imagen de la figura 6.19. Si se filtra por rayos reflejados (figura 6.20, entonces resulta la figura 6.21. Se observa que hay rayos reflejados que se ubican de forma paralela a una de las murallas, lo cual no tiene sentido. Por otro lado, ninguna de las murallas tiene materiales reflectivos. El error, en este caso, se encuentra en la fórmula para calcular rayos reflejados, en sintonía con lo anterior.

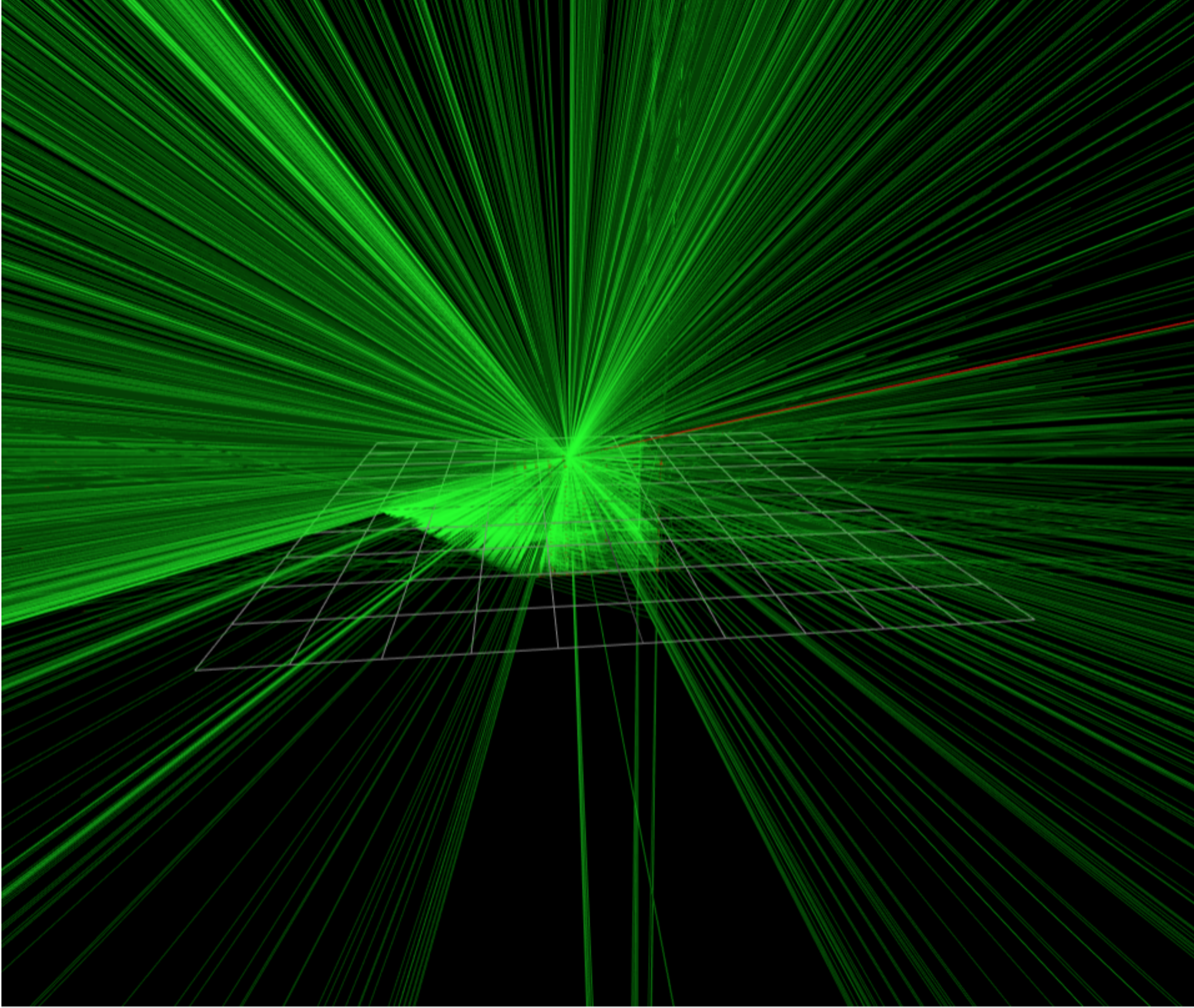


Figura 6.18: Caso 4 tras carga inicial.

Figura 6.19: Caso 4 mostrando sólo rayos con puntos de intersección.

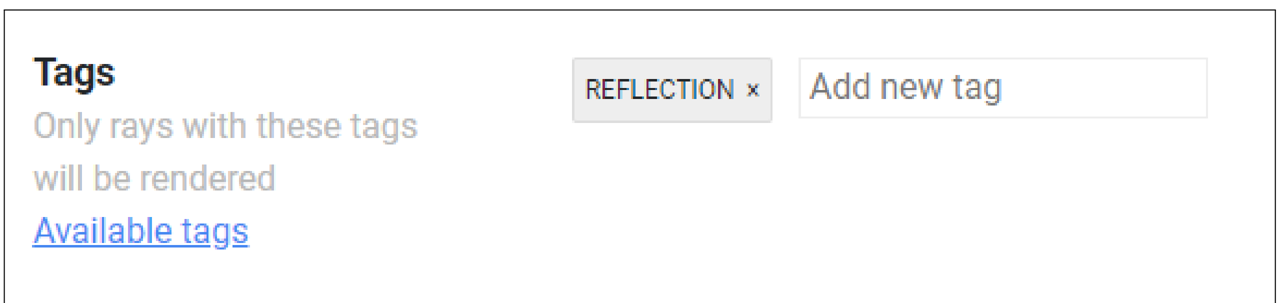


Figura 6.20: Filtros en el caso 4.

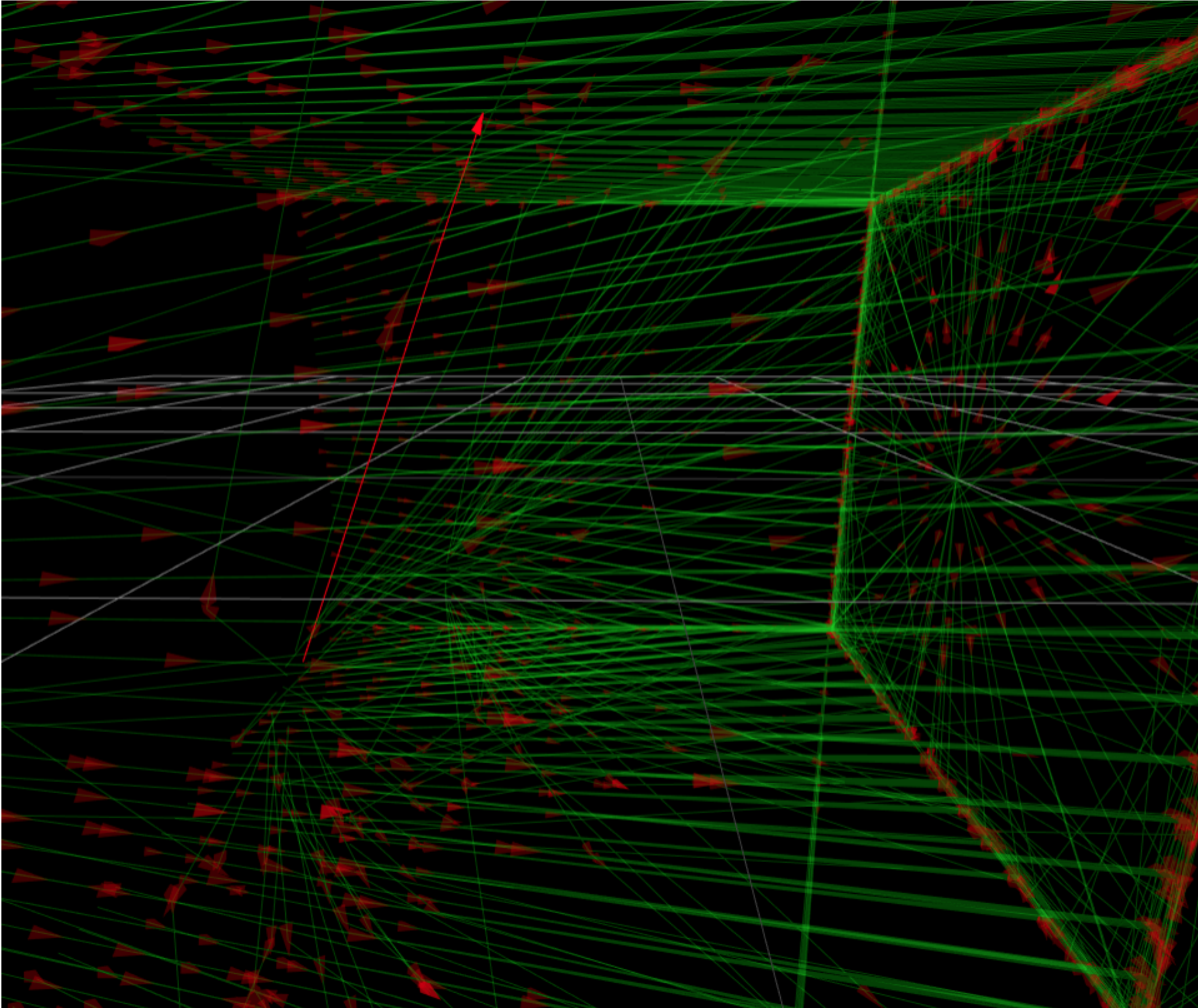
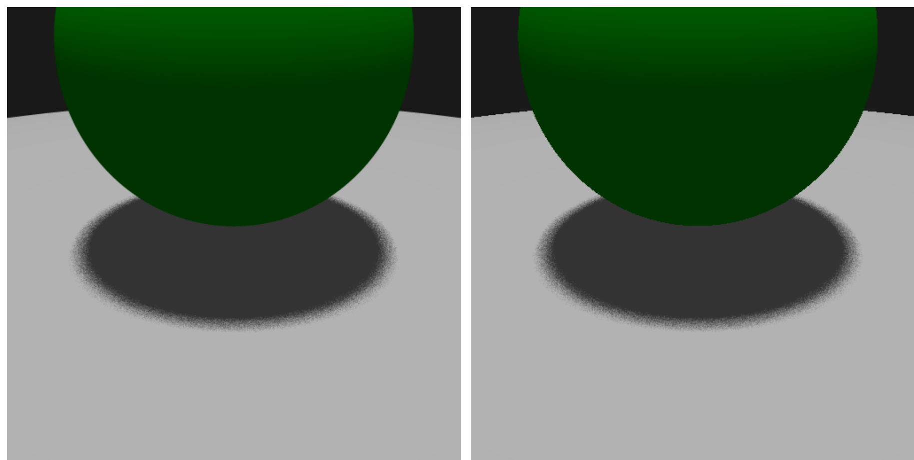


Figura 6.21: Caso 4 mostrando sólo rayos de reflexión.

6.1.5. Caso 5

Descripción de la escena: apéndice A.5.
Etiquetas disponibles:
- **CAMERA:** Los rayos que son lanzados en un inicio desde la cámara.
- **OBJECT_4 y OBJECT_7:** Los rayos que intersectan con las esferas pequeñas.
- **OBJECT_1, OBJECT_2, OBJECT_3, OBJECT_5 y OBJECT_6:** Los rayos que intersectan con las esferas grandes que actúan como murallas.
- **REFRACTION:** Los rayos que se lanzan como producto de una refracción.
- **REFLECTION:** Los rayos que se lanzan como producto de una reflexión.



(a) Imagen correcta (b) Imagen incorrecta

Figura 6.22: Caso 5

En este caso el problema es que, por alguna razón, el *antialiasing* no funciona (ver sección 2.2.3). La figura 6.23 muestra el visualizador tras la primera carga.

La forma en la que se implementa *antialiasing* en el raytracer en estudio involucra lanzar múltiples rayos por pixel. Por lo anterior, vale la pena ver qué ocurre con uno cualquiera en específico. En la figura 6.24 se aprecia el visualizador cuando se filtran los rayos que afectan al pixel $(1, 1)$.

La figura 6.24 muestra cómo hay un rayo solamente afectando al pixel $(1, 1)$. Esto llama la atención, ya que se deberían ver múltiples rayos asociados a este pixel. En efecto, el problema es que el raytracer está lanzando múltiples rayos, pero **iguales**, para un pixel determinado.

6.2. Resultados

La tabla 6.1 muestra la cantidad de respuestas correctas que enviaron los 4 participantes del experimento para cada caso. Se considera un acierto cuando el usuario da una descripción

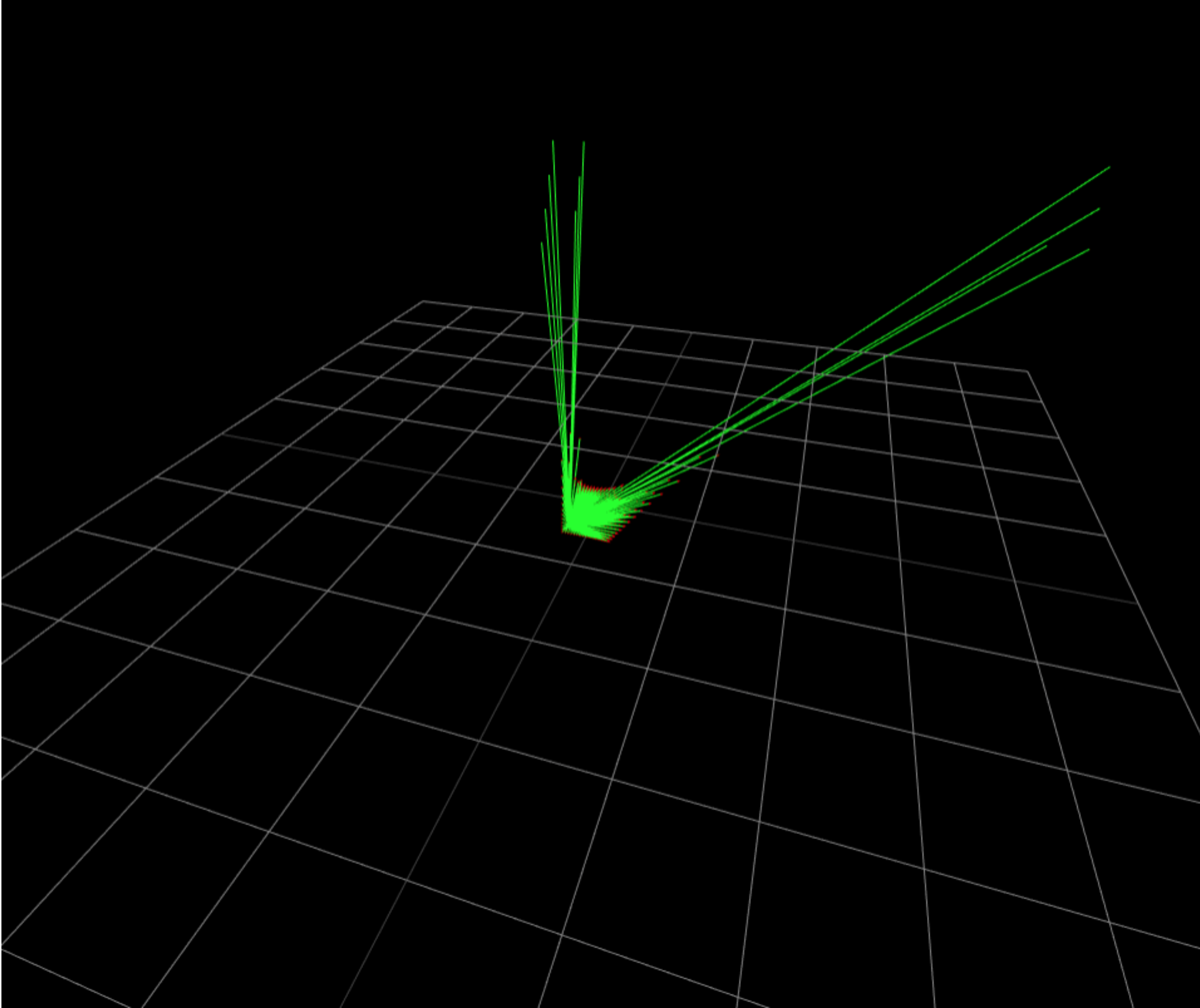


Figura 6.23: Caso 5 tras la primera carga.

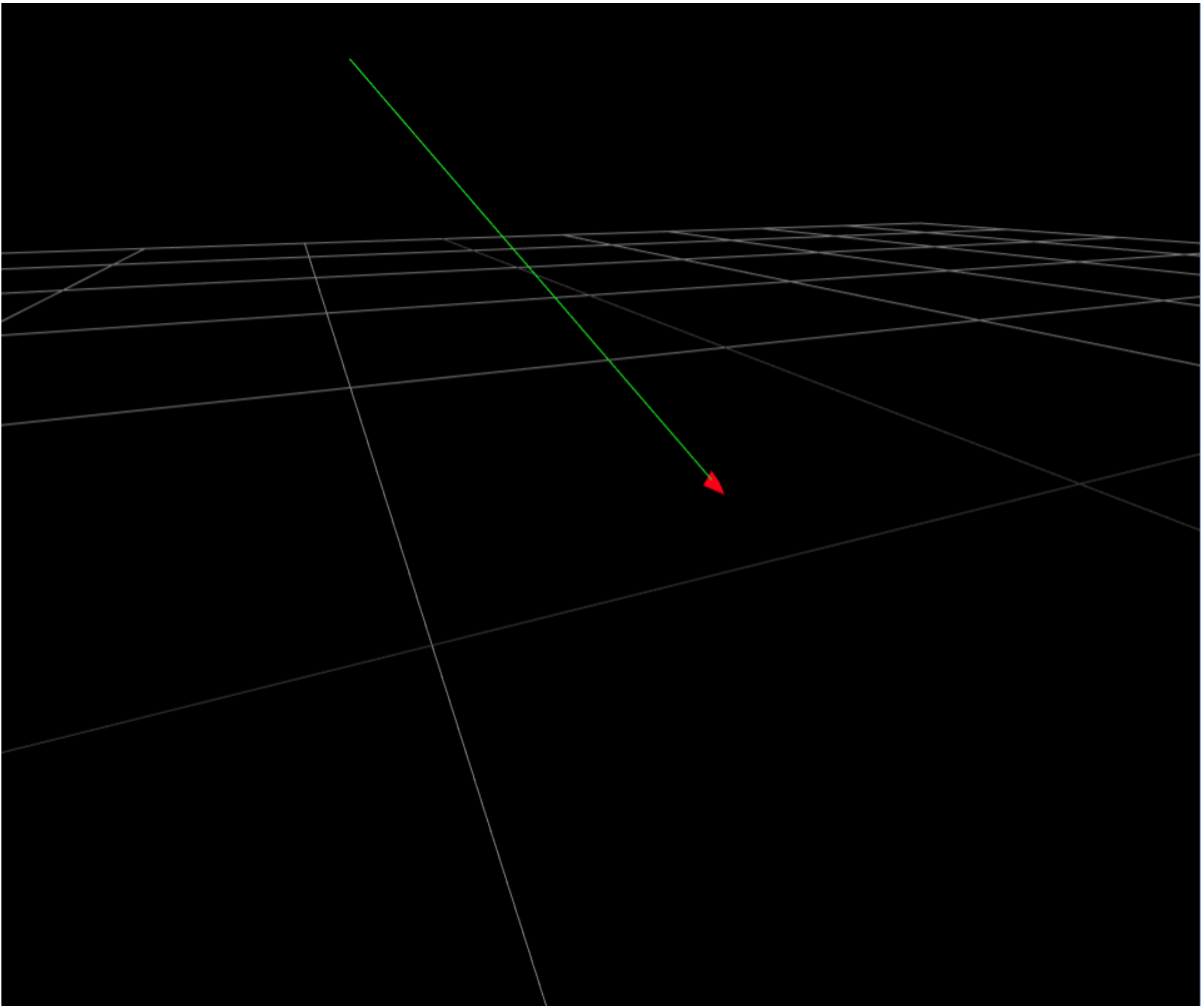


Figura 6.24: Caso 5 mostrando rayos asociados al pixel $(1, 1)$

Pixels

Only rays related to these pixels will be rendered

Area

X 1 Y 1

Rays with their pixel set to (x = 1, y = 1)

Figura 6.25: Filtrado para el caso 5.

Tabla 6.1: Resoluciones por usuarios de los casos de estudio.

Caso	Número de Resoluciones	% del total
1	2	50
2	2	50
3	4	100
4	2	50
5	4	100

del problema que se piensa que lo dejará suficientemente cerca de encontrar el error.

Conclusión

En este trabajo se estudió el problema del depurado en aplicaciones que utilizan la técnica de renderizado raytracing y se propuso e implementó una solución.

Se estableció un estándar de comunicación de rayos, y se piensa que se ha conseguido un balance adecuado entre la simplicidad del formato en función de la complejidad de las funcionalidades de la herramienta que se quería construir. En esta misma línea, se implementó una librería de logging para un lenguaje específico (C++) que consiguió el cometido de proveer una prueba concreta de que el estándar propuesto es compatible con la solución.

Por otro lado, se construyó una plataforma web que se integra correctamente con el esquema recién mencionado. Se estudiaron cinco casos de uso en los que la aplicación logró ser de utilidad. En esta misma línea, los comentarios recibidos por parte de los usuarios, aunque pocos en número, dejan un alto grado de satisfacción. Sin embargo, hay que destacar que hay que realizar pruebas con más desarrolladores para sacar mejores conclusiones en este sentido.

La idea podría haber sido implementada a través de una miríada de arquitecturas posibles. En este caso, se escogió el medio web por su ubicuidad y disponibilidad de herramientas disponibles para llevar a cabo el desarrollo de la solución. Al ser uno de los objetivos secundarios del proyecto tener la capacidad de integrarse con diversas implementaciones de raytracers, se piensa que fue la elección adecuada.

Con respecto al conjunto de funcionalidades implementadas, se cree que conforman un grupo sólido para llevar a cabo tareas de depurado en raytracers. Los casos de estudio, de complejidad variable, son prueba de esto.

El sistema de etiquetado resulta ser particularmente flexible, pues permite hacer consultas de carácter lógico que están limitadas a las posibilidades que tenga el usuario de etiquetar sus rayos. Hay consultas de relativa complejidad que fueron ilustradas en los capítulos anteriores, que no serían necesariamente directas de construir con otro tipo de sistema en consideración de la usabilidad.

En general, se lograron los objetivos propuestos y se consiguió construir un *framework* de trabajo para el depurado de raytracers.

Trabajo futuro

Los participantes del experimento descrito en la sección 6 tuvieron la posibilidad de indicar características que, en su opinión, mejorarían la aplicación. En consideración de esto, se piensa que algo que podría incluirse en el futuro es la posibilidad de asignar colores a los rayos según sus etiquetas, pues potencialmente ayudaría a identificarlos en la visualiación. Por otro lado, se podría permitir subir la imagen en conjunto con los rayos con el objetivo de tener una mejor referencia de a qué objeto de la escena están afectando los rayos. Por último, se piensa que sería útil permitirle al usuario poder definir qué rayos son padres de otros, en complemento a lo que ocurre en la actualidad, en la que el cómputo de las relaciones de parentezco lo realiza la plataforma.

Bibliografía

[1] Flux | application architecture for building user interfaces. https://facebook.github.io/flux/.

[2] Google forms. https://www.google.com/forms/about/.

[3] Introducing json. http://json.org/.

[4] React – a javascript library for building user interfaces. https://reactjs.org/.

[5] Redux · a predictable state container for js apps. https://redux.js.org/.

[6] Specification. https://json-schema.org/specification.html.

[7] Three.js - javascript 3d library. https://threejs.org/.

[8] Trace project debugging user interface. https://courses.cs.washington.edu/courses/cse557/09au/projects/trace/extra/debug.html.

[9] Introducing the nvidia rtx ray tracing platform. https://developer.nvidia.com/rtx/raytracing/, Oct 2018.

[10] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Jan 1975.

[11] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Trans. Graph.*, 37(3):30:1–30:21, August 2018.

[12] Randima Fernando. Gpu gems. chapter 28, pages 473–486. Addison-Wesley, 2004.

[13] Christiaan Gribble, Jeremy Fisher, Daniel Eby, Ed Quigley, and Gideon Ludwig. Ray tracing visualization toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 71–78, New York, NY, USA, 2012. ACM.

[14] Hristo Lesev and Alexander Penev. A framework for visual dynamic analysis of ray

tracing algorithms. *Cybernetics and Information Technologies*, 14(2):38–49, 2014.

[15] Michael Oren and Shree K. Nayar. Generalization of lamberts reflectance model. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH 94*, 1994.

[16] Juan Pineda. A parallel algorithm for polygon rasterization. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH 88*, page 17–20, 1988.

[17] Turner Whitted. An improved illumination model for shaded display. *ACM SIGGRAPH 2005 Courses on - SIGGRAPH 05*, 2005.

Apéndice A

Escenas

A.1. Caso 1

```
{
    "__type__": "scene",
    "camera": {
        "__type__": "camera",
        "fov": 45.0,
        "position": [0,5,5],
        "up": [0,0,1],
        "target": [0,0,0]
    },
    "objects": [
        {
            "__type__": "sphere",
                        "radius": 100,
                        "position": [0, 0, -100],
            "materials": ["white_mirror", "gray_lambert"]
        },
                {
            "__type__": "sphere",
                        "radius": 0.2,
                        "position": [0, 0, 1],
            "materials": ["green_lambert", "white_bph_100"]
        },
                {
            "__type__": "sphere",
                        "radius": 0.4,
                        "position": [2, 0, 1],
            "materials": ["red_lambert", "white_bph_100"]
        },
                {
```

```
            "__type__": "sphere",
                        "radius": 0.6,
                        "position": [-2, 0, 1],
            "materials": ["blue_lambert", "white_bph_100"]
        }
    ],
    "lights": [
           {
          "__type__": "point_light",
          "position": [0, 0, 1.5],
          "color": [0.5, 0.5, 0.5]
        },
                {
          "__type__": "directional_light",
          "direction": [-1, 0, -1],
          "color": [0.5, 0.0, 0.0]
        },
                {
          "__type__": "spot_light",
          "position": [2, 0, 1.5],
          "direction": [0, 0, -1],
          "angle": 150,
          "color": [0.3, 0.3, 0.3]
        },
        {
          "__type__": "ambient_light",
          "color": [0.2, 0.2, 0.2]
        }
    ],
    "params": {
        "background_color": [0.1,0.1,0.1]
    }
}
```

A.2. Caso 2

```
{
    "__type__": "scene",
    "camera": {
        "__type__": "camera",
        "fov": 45.0,
        "position": [0,5,5],
        "up": [0,0,1],
        "target": [0,0,0]
```

```
9       },
10      "objects": [
11          {
12              "__type__": "sphere",
13                          "radius": 100,
14                          "position": [0, 0, -100],
15              "materials": ["white_lambert"]
16          },
17                  {
18              "__type__": "sphere",
19                          "radius": 0.2,
20                          "position": [0, 0, 1],
21              "materials": ["green_lambert", "white_bph_100"]
22          }
23      ],
24      "lights": [
25             {
26            "__type__": "point_light",
27            "position": [0, 0, 1.5],
28            "color": [0.5, 0.5, 0.5]
29          },
30          {
31            "__type__": "ambient_light",
32            "color": [0.2, 0.2, 0.2]
33          }
34      ],
35      "params": {
36          "background_color": [0.1,0.1,0.1]
37      }
38  }
```

A.3. Caso 3

```
1   {
2     "__type__": "scene",
3     "camera": {
4       "__type__": "camera",
5       "fov": 45.0,
6       "position": [ 0, 0, 4 ],
7       "up": [ 0, 1, 0 ],
8       "target": [ 0, 0, 0 ]
9     },
10    "objects": [
11      {
```

```
12        "__type__": "sphere",
13        "radius": 100,
14        "position": [ 0.0, 0.0, -101 ],
15        "materials": [ "white_lambert" ]
16      },
17      {
18        "__type__": "sphere",
19        "radius": 100,
20        "position": [ 101.0, 0.0, 0.0 ],
21        "materials": [ "green_lambert" ]
22      },
23      {
24        "__type__": "sphere",
25        "radius": 100,
26        "position": [ -101.0, 0.0, 0.0 ],
27        "materials": [ "red_lambert"]
28      },
29      {
30        "__type__": "sphere",
31        "radius": 100,
32        "position": [ 0.0, 101.0, 0.0 ],
33        "materials": [ "white_lambert" ]
34      },
35      {
36        "__type__": "sphere",
37        "radius": 100,
38        "position": [ 0.0, -101.0, 0.0 ],
39        "materials": [ "white_lambert" ]
40      },
41      {
42        "__type__": "sphere",
43        "radius": 0.4,
44        "position": [ 0.5, -0.55, 0.6 ],
45        "materials": [ "air" ]
46      },
47      {
48        "__type__": "sphere",
49        "radius": 0.4,
50        "position": [ -0.5, -0.55, 0.4 ],
51        "materials": [ "blue_lambert", "white_bph_1000", "air" ]
52      }
53    ],
54    "lights": [
55      {
56        "__type__": "point_light",
57        "position": [ 0.0, 0.9, 0.5 ],
58        "color": [ 0.5, 0.5, 0.5 ]
```

```
59      },
60      {
61        "__type__": "ambient_light",
62        "color": [ 0.2, 0.2, 0.2 ]
63      }
64    ],
65    "params": {
66      "background_color": [ 0.0, 0.0, 0.0 ],
67          "refraction_index": 1
68    }
69  }
```

A.4. Caso 4

```
1   {
2     "__type__": "scene",
3     "camera": {
4       "__type__": "camera",
5       "fov": 45.0,
6       "position": [ 0, 0, 4 ],
7       "up": [ 0, 1, 0 ],
8       "target": [ 0, 0, 0 ]
9     },
10    "objects": [
11      {
12        "__type__": "sphere",
13        "radius": 100,
14        "position": [ 0.0, 0.0, -101 ],
15        "materials": [ "white_lambert", "gray_mirror" ]
16      },
17      {
18        "__type__": "sphere",
19        "radius": 100,
20        "position": [ 101.0, 0.0, 0.0 ],
21        "materials": [ "green_lambert", "gray_mirror" ]
22      },
23      {
24        "__type__": "sphere",
25        "radius": 100,
26        "position": [ -101.0, 0.0, 0.0 ],
27        "materials": [ "red_lambert", "gray_mirror" ]
28      },
29      {
30        "__type__": "sphere",
```

```
31        "radius": 100,
32        "position": [ 0.0, 101.0, 0.0 ],
33        "materials": [ "white_lambert", "gray_mirror" ]
34      },
35      {
36        "__type__": "sphere",
37        "radius": 100,
38        "position": [ 0.0, -101.0, 0.0 ],
39        "materials": [ "white_lambert", "gray_mirror" ]
40      },
41      {
42        "__type__": "sphere",
43        "radius": 0.4,
44        "position": [ 0.5, -0.55, 0.6 ],
45        "materials": [ "gold_mirror" ]
46      },
47      {
48        "__type__": "sphere",
49        "radius": 0.4,
50        "position": [ -0.5, -0.55, 0.4 ],
51        "materials": [ "blue_lambert", "white_bph_1000", "
            gray_mirror" ]
52      }
53    ],
54    "lights": [
55      {
56        "__type__": "point_light",
57        "position": [ 0.0, 0.9, 0.5 ],
58        "color": [ 0.5, 0.5, 0.5 ]
59      },
60      {
61        "__type__": "ambient_light",
62        "color": [ 0.2, 0.2, 0.2 ]
63      }
64    ],
65    "params": {
66      "background_color": [ 0.0, 0.0, 0.0 ]
67    }
68  }
```

A.5. Caso 5

```
1  {
2    "__type__": "scene",
```

```
3   "camera": {
4     "__type__": "camera",
5     "fov": 45.0,
6     "position": [ 0, 0.5, 1 ],
7     "up": [ 0, 1, 0 ],
8     "target": [ 0, 0, 0 ]
9   },
10  "objects": [
11    {
12      "__type__": "sphere",
13      "radius": 10,
14      "position": [ 0, -10.0, 0.0 ],
15      "materials": [ "white_lambert" ]
16    },
17    {
18      "__type__": "sphere",
19      "radius": 0.3,
20      "position": [ 0, 0.35, 0.0 ],
21      "materials": [ "green_lambert", "white_bph_100" ]
22    }
23  ],
24  "lights": [
25    {
26      "__type__": "area_light",
27      "position": [ 0, 20, 0 ],
28      "color": [ 0.5, 0.5, 0.5 ],
29      "sizeA": 2,
30      "sizeB": 2,
31      "directionA": [ 1.0, 0.0, 0.0 ],
32      "directionB": [ 0.0, 0.0, 1.0 ]
33    },
34    {
35      "__type__": "ambient_light",
36      "color": [ 0.2, 0.2, 0.2 ]
37    }
38  ],
39  "params": {
40    "background_color": [ 0.1, 0.1, 0.1 ]
41  }
42 }
```

Apéndice B

Registros

B.1. Esquema

B.2. Ejemplo

```
[
  {
    "origin": [0, 0, 0],
    "direction": [0, 1, 0],
    "intersection": [0, 5, 0],
    "pixel": [30, 30],
    "tags": ["REFLECTION"]
  },
  {"origin": [0, 0, 0], "direction": [1, 0, 0], "intersection":
      [2, 0, 0]},
  {
    "origin": [2, 0, 0],
    "direction": [0, 1, 0],
    "intersection": [2, 4, 0],
    "tags": ["CAMERA", "REFLECTION"]
  },
  {
    "origin": [2, 0, 0],
    "direction": [0, -1, 0],
    "intersection": [2, -3, 0],
    "tags": ["CAMERA"]
  },
  {"origin": [2, -3, 0], "direction": [1, 0, 0], "intersection"
     : [5, -3, 0]},
  {
    "origin": [0, 0, 0],
```

```
25      "direction": [0, -1, 0],
26      "intersection": [0, -3, 0],
27      "tags": ["CAMERA", "REFLECTION"],
28      "pixel": [20, 20]
29    },
30    {
31      "origin": [0, 0, 0],
32      "direction": [-1, 0, 0],
33      "tags": ["CAMERA"],
34      "pixel": [10, 10]
35    }
36  ]
```