Universidad Zaragoza

1542

Trabajo Fin de Grado

Aplicación de técnicas de paralelización en el trazado de rayos para la síntesis de hologramas generados por computador

Applying Ray tracing parallelization techniques for Computer Generated Holograms synthesis

Autor

Diego Sanz Fuertes

Directores

Dr. D. Alfonso Blesa Gascón
Dr. D. Francisco José Serón Arbeloa

Escuela Universitaria Politécnica de Teruel

2023/2024

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

Resumen

El objetivo de este Trabajo Final de Grado (TFG) consiste en diseñar, implementar y evaluar un sistema para la generación de hologramas sintéticos por computador (CGH) utilizando las técnicas de trazado de rayos y modelado geométrico basado en nubes de puntos.

Para abordar la elevada demanda computacional asociada a la generación de hologramas, los algoritmos se han desarrollado teniendo en cuenta la alta paralelización del problema tanto en CPU como en GPU, utilizando los lenguajes de programación C++ y CUDA respectivamente.

Los los hologramas generados se han validado mediante simulaciones de la propagación de ondas electromagnéticas y experimentos en un laboratorio de óptica holográfica. Además, se han comparado los tiempos de computo de distintas CPUs, GPUs y distintos números de hilos en CPUs.

Abstract

The objective of this Bachelor's Degree Final Project (TFG) is to design, implement and evaluate a system for the generation of synthetic computer generated holograms (CGH) using ray tracing and geometric modelling techniques based on point clouds.

To address the high computational demand associated with hologram generation, the algorithms have been developed taking into account the high parallelisation of the problem on both CPUs and GPUs, using the C++ and CUDA programming languages respectively.

The generated holograms have been validated by simulations of electromagnetic wave propagation and experiments in a holographic optics laboratory. Furthermore, the computational times of different CPUs, GPUs and different numbers of threads in CPUs have been compared.

Índice

1. Introducción y objetivos **3**

2. Estado del arte **4**

3. Proceso de síntesis de escenas virtuales en 3D mediante hologramas digitales **8**

3.1. Definición de la escena virtual 8

3.1.1. Geometría 9

3.1.2. Textura 10

3.1.3. Iluminación 14

3.2. Generación de imágenes por computador (CGI) 15

3.3. Generación de hologramas por computador (CGH) 19

3.4. Reconstrucción de la escena 21

4. Técnicas de paralelización **24**

4.1. CPU 24

4.2. GPU 25

4.3. Implementación 26

5. Resultados **28**

5.1. Tablas comparativas de tiempos. 28

6. Conclusiones **33**

7. Calendario **33**

8. Trabajo futuro **35**

9. Dificultades encontradas **35**

Referencias **36**

10. Anexos **37**

10.1. Librerías y licencias 37

1. Introducción y objetivos

La holografía es una técnica que permite capturar y reconstruir el campo de ondas completo de la luz generado en una escena determinada [1]. Se utiliza hoy en día para varios propósitos, entre los que se encuentran los sistemas de pantallas tridimensionales (3D) [2]. Las pantallas 3D holográficas permiten reproducir fielmente y sin restricciones todas las señales de profundidad visuales naturales conocidas, como la oclusión, la acomodación del ojo, la convergencia y la estereopsis [3]. Algunas de esas cualidades no se pueden conseguir con otras tecnologías como, por ejemplo, cuando se obtiene una representación 3D estéreo.

Los hologramas de un objeto se obtienen capturando el frente de ondas que proviene de un objeto cuando se ilumina, proceso análogo a realizar una fotografía. Dicho fenómeno que también se puede simular generando el holograma mediante un computador (CGH), actividad análoga pero muchísimo más demandante de cálculo, a la conocida generación de imágenes sintéticas por computador. Sin embargo, uno de los mayores desafíos de CGH's es obtenerlos en tiempos computacionales aceptables [2].

En este trabajo de final de grado se aborda la generación de hologramas sintéticos, para ello, se han diseñado e implementado:

- Un trazador de rayos estándar para generar imágenes sintéticas (2D) por computador el cuál simula el comportamiento de los rayos de luz basado en la física (Physically Based Ray Tracing o PBRT, en inglés).
- Un sistema de generación de hologramas (3D) por computador que hace uso de la técnica de modelado geométrico basada en nubes de puntos y la técnica de trazado de rayos, pero aplicada a múltiples posiciones del ojo.
- Por último, con objeto de demostrar la validez de los resultados obtenidos, se visualizan los hologramas sintéticos generados a partir de una escena virtual, en un laboratorio de óptica holográfica.

Dado que la exigencia computacional de este problema es muy alta y mucho mayor que las aplicaciones tradicionales de la computación gráfica, este proyecto propone una solución adecuada para abordar este problema que considera la arquitectura del software, los algoritmos y la alta paralelización del problema. Esta última, tanto en CPU como en GPU utilizando los lenguajes de programación C++ y CUDA.

2. Estado del arte

La generación de imágenes sintéticas por computador (Computer-Generated Imagery o CGI, en inglés) es un campo de la computación gráfica bien estudiado que se utiliza en una gran variedad de áreas, como el diseño, la industria, el cine y los videojuegos. Consiste, en esencia, en capturar lo que vería un ojo al observar una escena virtual formada por objetos y fuentes de iluminación, a través de una pantalla pixelada obteniendo una imagen (tipo fotográfica) de dicha escena.

Una de las técnicas más utilizadas en CGI es el trazado de rayos (o ray tracing, en inglés). En la literatura se pueden encontrar libros especializados que abordan tanto la teoría como la implementación con distintos niveles de profundidad [4] [5].

Esta técnica simula la interacción de la luz con una escena para sintetizar escenas virtuales (véase Figura 1). Se considera una técnica computacionalmente costosa, por lo que es utilizada principalmente para renderización offline (no en tiempo real) con dispositivos computacionales habituales, o en tiempo real utilizando granjas de potentes procesadores y GPUs.

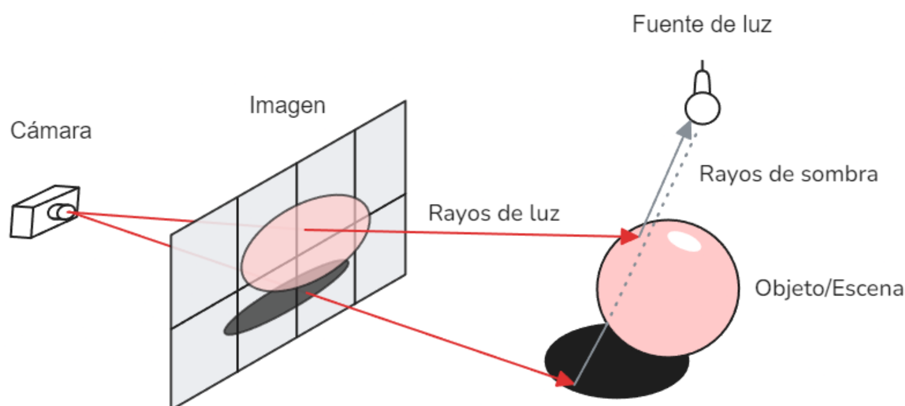


Figura 1: Diagrama de la técnica de trazado de rayos.

Un trazador de rayos completo ha de simular al menos los siguientes objetos y fenómenos [5]:

1. Cámaras: El modelo de una cámara determina cómo y desde dónde la escena se observa, incluyendo como una imagen de la escena es recogida por un sensor.

2. Intersecciones rayo-objeto: Es necesario conocer precisamente cuándo y dónde un rayo intersecta un objeto geométrico, además de determinar algunas propiedades del objeto en el punto de intersección.

3. Fuentes de luz: El trazador de rayos ha de modelar la distribución de la luz en la escena.

4. Visibilidad: Se debe poder conocer si una luz determinada deposita energía en un punto de una superficie.

5. Dispersión de la luz en superficies: Cada objeto ha de proveer información sobre como la luz interactúa con la superficie del objeto.

6. Transporte indirecto de luz: La luz puede llegar a la superficie después de rebotar o atravesar otras superficies.

7. Propagación de rayos: Se necesita conocer el comportamiento de la luz mientras atraviesa un espacio, siendo su velocidad constante en el vacío.

Tal y como se ha mencionado anteriormente, se tiene como objetivo generar hologramas sintéticos simulando la captura del frente de ondas (3D) de una escena en un plano.

Para entender el comportamiento de los CGH se debe tener presente la naturaleza de la luz. En esencia, una onda electromagnética se puede caracterizar en el espacio con su amplitud, su fase y su longitud de onda. El formalismo matemático general viene dado por las ecuaciones de Maxwell [1] y su solución general es inabordable con la capacidad de cálculo actual.

Por este motivo, se buscan simplificaciones que permitan modelar adecuadamente el comportamiento de ondas electromagnéticas. En el caso que nos ocupa, supondremos siempre ondas monocromáticas, en el rango óptico escalar y en medios isótropos. En estas condiciones, dada la velocidad de propagación de las ondas de luz, podemos obviar la parte temporal de la onda y centrar los cálculos en la propagación espacial de la misma.

Supongamos que a un punto de espacio llegan ondas electromagnéticas de N fuentes puntuales de luz (definidas por su amplitud y fase), (véase la Figura 2). El valor de la onda se pueda calcular según la Ecuación 1 [1].

$$H(x,y) = \sum_{j=1}^{N} a_j \exp\left(\frac{\pi i}{\lambda z_j}\left[(x-x_j)^2 + (y-y_j)^2\right]\right) \tag{1}$$

Donde:

- H es el valor de la onda en el punto (x, y)
- a es la amplitud de cada una de las N fuentes puntuales que llegan a (x, y)
- λ es la longitud de onda
- (x_j, y_j, z_j) son las coordenadas de cada fuente puntual
- $\frac{1}{\lambda z_j}[(x-x_j)^2 + (y-y_j)^2])$ es su fase.

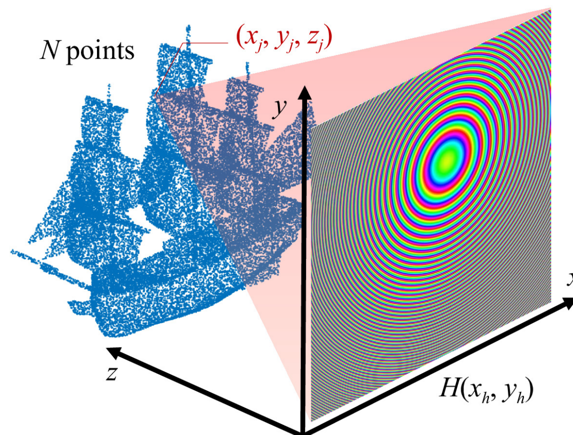


Figura 2: Diagrama de la aplicación de la Ecuación 1 en el plano $H(x_h, y_h)$ del punto (x_j, y_j, z_j). Fuente: Blinder et al. [3].

Dado un sensor de MxN píxeles, el cálculo de un CGH es, en esencia, el cálculo de la onda electromagnética resultante que llega a cada uno de los píxeles del sensor.

Para abordar este cálculo existen varias opciones. En el estudio [3] se ofrece una visión general del estado del arte en CGH, una clasificación de los algoritmos y diferentes técnicas de aceleración.

Entre los algoritmos clasificados se encuentran los dos que se van a utilizar en este trabajo:

- Definida una escena mediante una nube de puntos: La técnica de cálculo consiste en la suma de todas las funciones de dispersión de puntos (PSFs, por sus siglas en inglés) definidas según la Ecuación 1, en el plano del holograma, que emanan de una colección de puntos luminosos. Se puede observar en la Figura 2 la función de dispersión de un punto en el plano $H(x_h, y_h)$. Las principales limitaciones de esta técnica son (1) la falta de soporte de efectos básicos como la oclusión y el sombreado; y (2) el alto coste computacional, aunque el algoritmo es altamente paralelizable [6]

- Trazado de rayos: Es una técnica para modelar el transporte de la luz basada en el seguimiento de rayos de luz individuales que rebotan en la escena e interactúan con materiales, calculando con precisión la cantidad de luz alcanzando cada píxel de la cámara virtual. Esta técnica puede aprovecharse en CGH para modelar también el transporte de la luz. Sin embargo, no puede utilizarse directamente, ya que la holografía se basa fundamentalmente en ondas, lo que difiere sustancialmente de los modelos basados en rayos. Uno de los principales problemas es la necesidad de obtener un continuo de puntos de vista y otro problema es la falta de coherencia de fase de la luz. Estos dos problemas hacen que los métodos basados en el trazado de rayos han de ser adaptados o combinados con otros algoritmos para ser utilizados efectivamente [3].

En [6] se propone un método híbrido que toma los mejores aspectos de los métodos de nube de puntos y del trazado de rayos. En vez de calcular directamente el campo de luz en el plano del holograma, se calcula la intensidad en cada punto de la nube de puntos de la escena mediante trazado de rayos, y se calcula la PSF con esa intensidad. De este modo se consiguen efectos de iluminación realistas, sin comprometer la coherencia de la fase, lo que da lugar a vistas continuas y señales de profundidad precisas.

La naturaleza del algoritmo de trazado de rayos permite que pueda ser acelerado mediante paralelización, ya que los rayos son independientes unos de los otros [7]. En [6] se paraleliza en GPUs mediante CUDA y OptiX.

En el caso de la propagación de ondas electromagnéticas entre dos planos paralelos, existe la posibilidad de disminuir drásticamente los tiempos de cálculo. En estas condiciones se pueden utilizar los algoritmos de convolución y de transformada de Fourier, bien descritos en la literatura [1]. Se trata de conocer los valores de la onda U en el plano (x, y) a partir de los valores conocidos de la misma en el plano (ξ, η) según descrito en la Ecuación 2.

En la Figura 3 se puede observar la propagación con el fin de recuperar un holograma almacenado.

$$U(x,y) = \frac{e^{ikz}}{i\lambda z} \exp\left(\frac{ik}{2z}\left(x^2+y^2\right)\right) \iint_{-\infty}^{\infty} \left\{U(\xi,\eta) e^{\frac{ik}{2z}\left(\xi^2+\eta^2\right)}\right\} e^{-i\frac{2\pi}{\lambda z}(x\xi+y\eta)} d\xi d\eta \tag{2}$$

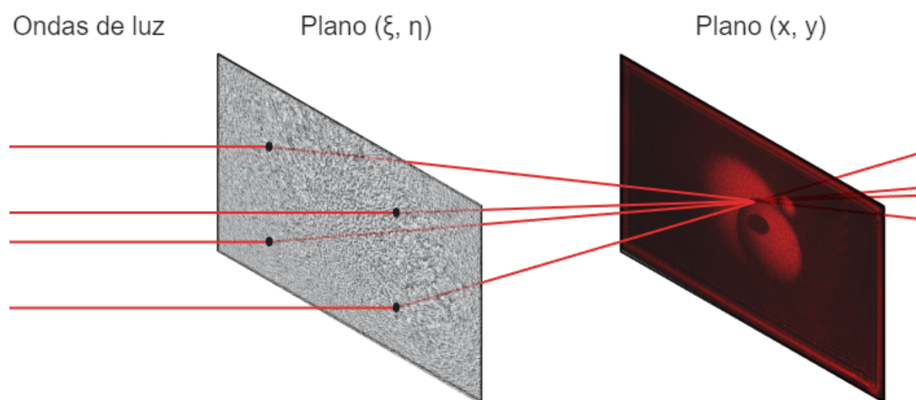


Figura 3: Diagrama de la propagación de ondas electromagnéticas desde el plano (ξ, η) hasta un punto en el plano (x, y).

3. Proceso de síntesis de escenas virtuales en 3D mediante hologramas digitales

La implementación del algoritmo definido en la sección anterior se ha realizado mediante el uso de los lenguajes C++ y CUDA para la definición de la escena virtual y la generación de hologramas digitales, y Python para la reconstrucción de la escena. Las librerías utilizadas se enumeran en el anexo Librerías y licencias.

Todas las figuras mostradas son de elaboración propia, tanto los diagramas como los renderizados, a no ser que se indique lo contrario.

3.1. Definición de la escena virtual

El primer paso para la síntesis de escenas virtuales consiste en definir la escena que se desea producir. Veamos el siguiente ejemplo (Figura 4):

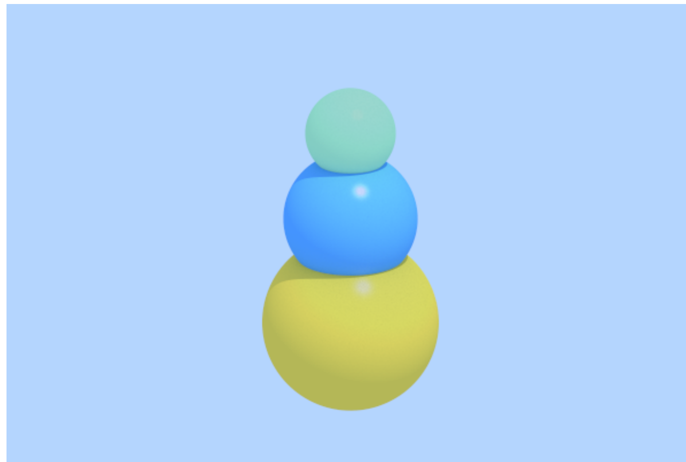


Figura 4: Render de ejemplo.

Esta escena está definida por el color del cielo, la posición, el tamaño y el material de las tres esferas, y la posición y color de una fuente de luz puntual. En pseudocódigo se podría definir de la siguiente manera (números en milímetros):

```
sky_color: light_blue;

Sphere(center: (0, 1.4, -1),
       radius: 0.2,
       material: Lambertian(albedo: light_green);
Sphere(center: (0, 1, -1),
       radius: 0.3,
       material: Lambertian(albedo: light_blue));
Sphere(center: (0, 0.5, -1),
       radius: 0.4,
       material: Lambertian(albedo: yellow);

PointLight(position: (3, 11, 3), color: white);
```

3.1.1. Geometría

La geometría de la escena se definirá mediante primitivas definidas matemáticamente. Estas primitivas son: triángulos, definidos mediante la posición espacial de sus vértices; y esferas, definidas mediante su radio y la posición espacial de su centro.

Utilizando la primitiva del triángulo se pueden representar mallas, definidas por una lista de triángulos. El soporte de mallas resulta muy útil ya que la mayoría de modelos 3D se encuentra en este formato.

3.1.2. Textura

Una vez definida la geometría de la escena, es necesario aplicar texturas a las primitivas. Las texturas que se han utilizado no siguen el significado tradicional de imágenes bidimensionales mapeadas sobre la superficie de la geometría, si no que se emplean distintos tipos de materiales, detallados a continuación:

1. Material difuso (lambertiano): Este material dispersa la luz siguiendo una distribución independiente al ángulo de incidencia y proporcional al coseno del ángulo formado entre la normal de la superficie y la dirección de dispersión. Ley conocida como coseno de Lambert. El color y la intensidad de la luz se ven modificados por el color (o albedo) del material. Este material podría describirse como mate. En la Figura 5 se puede apreciar el comportamiento del material difuso y en la Figura 6 se muestra un diagrama de la reflexión difusa lambertiana.

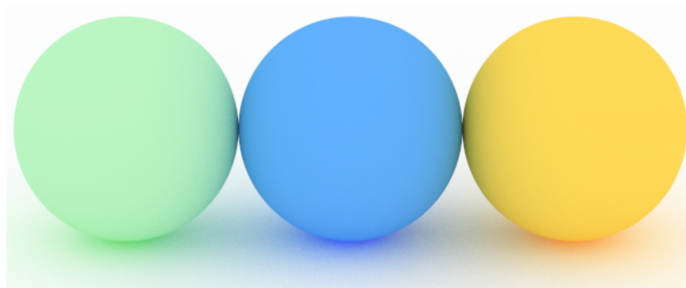


Figura 5: Render de tres esferas de diferentes colores sobre una superficie blanca. Materiales difusos.

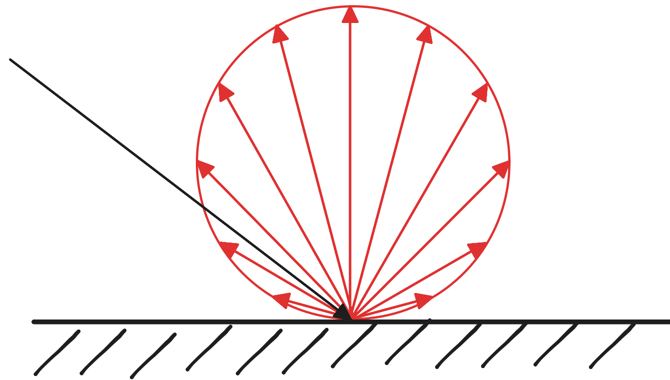


Figura 6: Diagrama de la dispersión de un material difuso lambertiano. En negro, el rayo de luz incidente, en rojo, la distribución de la dirección e intensidad de la dispersión.

2. Material metálico: Al contrario que el material difuso, el material metálico refleja la luz en el mismo ángulo en dirección opuesta respecto al ángulo de incidencia (véase Figura 7). Este efecto produce un reflejo de la misma manera que un espejo. Este material también cuenta con un parámetro que controla la borrosidad (o fuzziness, en inglés) del reflejo. En la Figura 8 se puede observar el efecto espejo del material junto a el parámetro de borrosidad.

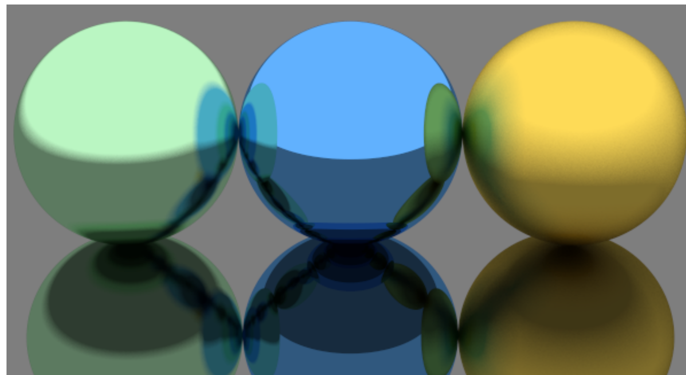


Figura 7: Render de tres esferas de diferentes colores sobre una superficie gris. Materiales metálicos con distinta borrosidad: 0.1 (izquierda), 0 (centro y superficie) y 0.5 (derecha).

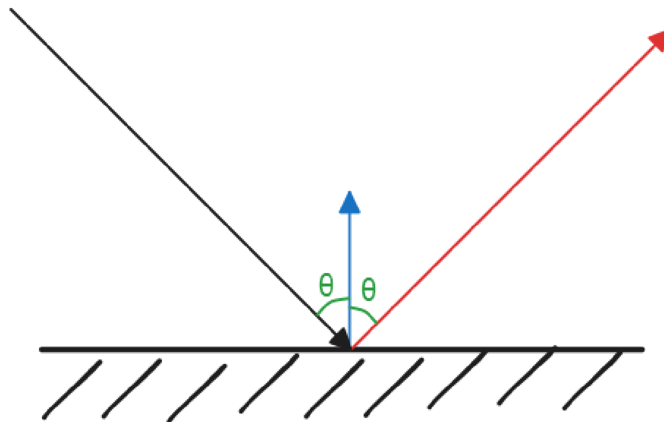


Figura 8: Diagrama de la reflexión de un material metálico sin borrosidad. Rayo de incidencia en negro, rayo reflejado en rojo y vector normal a la superficie en azul.

3. Material dieléctrico: Este material representa materiales transparentes como agua y cristal. Cuando la luz incide sobre el material, se divide en luz reflejada (como el material metálico) y en luz refractada. La reflectividad se describe según las ecuaciones de Fresnel [1] y la refracción según la ley de Snell como se puede observar en la Figura 10. Este material también cuenta con un parámetro que controla el índice de refracción. En la Figura 9 se puede apreciar el efecto de distintos índices de refracción. También se puede observar la luz refractada en la bola de la derecha debido a un índice de refracción elevado. Cabe destacar que las bolas de materiales dieléctricos no tienen sombra, se puede comprobar que ese es el caso observando una bola de cristal en un día nublado.

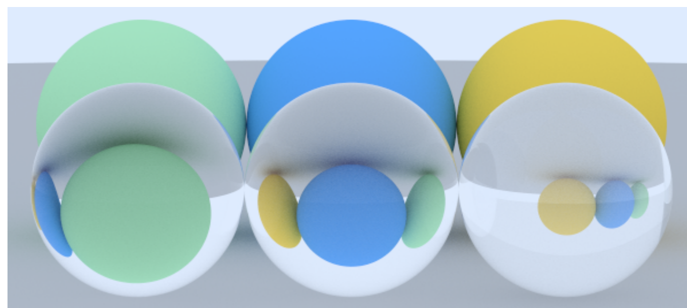


Figura 9: Render de tres bolas con un material dieléctrico y tres esferas de distintos colores, sobre una superficie gris. Los índices de refracción de las tres bolas son: 1.3 (agua, izquierda), 1.5 (cristal, centro) y 2.4 (diamante, derecha).

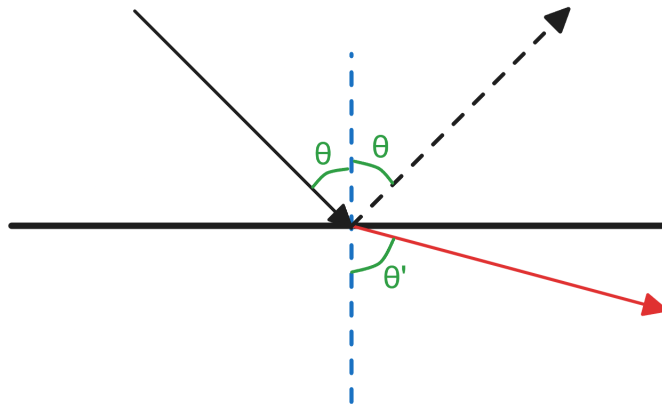


Figura 10: Diagrama de la refracción y reflexión de un rayo de luz al atravesar la superficie de un material dieléctrico. Rayo de incidencia en negro, rayo refractado en rojo y rayo reflejado discontinuo. Se sigue la ley de Snell para la refracción: $n \operatorname{sen} \theta = n' \operatorname{sen} \theta'$ siendo n y n' los índices de refracción.

También existen otras formas más completas de definir materiales como las descritas en los modelos de Disney [8] o OpenPBR [9].

Finalmente, en la Figura 11 se puede observar un render de una escena la cual contiene todos los materiales.

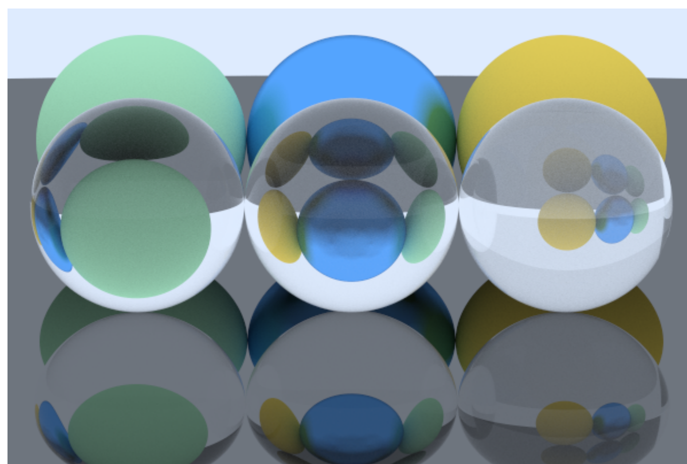


Figura 11: Render demostrando los distintos materiales. Basado en el render de la Figura 10 (superficie y esfera azul metálicas con borrosidad 0 y 0.3 respectivamente).

3.1.3. Iluminación

La última sección de la definición de la escena virtual es la iluminación. La iluminación que se utiliza se puede dividir en dos tipos de fuente: el cielo y fuentes puntuales de luz.

El cielo ilumina de manera uniforme la escena mientras que las fuentes de luz puntuales siguen el modelo de reflexión de Blinn-Phong [10], que describe la forma en la que una superficie refleja la luz como una combinación de la iluminación difusa y la iluminación especular. No se incluye el término ambiental del modelo ya que se utiliza el cielo. En la Figura 12 se pueden observar los componentes por separado.

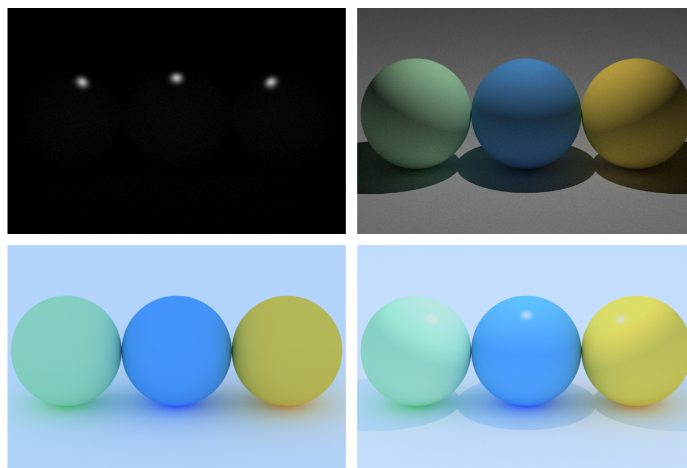


Figura 12: Render de tres esferas con iluminación puntual e iluminación ambiente, separado en sus componentes: Especular (superior izquierda), difuso por punto de luz (superior derecha), difuso por iluminación ambiente (inferior izquierda) y componentes agregados (inferior derecha).

En la Figura 13 se muestra la escena que se utilizará como referencia para la generación de hologramas. Se encuentra iluminada por una fuente de luz puntual de color rojo (longitud de onda de 632.8nm), ya que es la longitud de onda con la que se trabaja en el laboratorio. La definición en pseudocódigo es la siguiente (en mm):

```
Sphere(center: (-0.8, 0.25, -12),
       radius: 3,
       material: Lambertian(albedo: magenta));
Sphere(center: (1, -0.25, 0),
```

```
        radius: 2,
        material: Lambertian(albedo: magenta));
Sphere(center: (2, 2, 5),
        radius: 0.5,
        material: Lambertian(albedo: magenta));

Camera(look_from: (0, 0, 200),
        look_at: (0, 0, 0),
        diffuse_intensity: 100%,
        specular_intensity: 0%,
        sky_intensity: 0%);

PointLight(position: (5000, 5000, 10000), color: red);
```

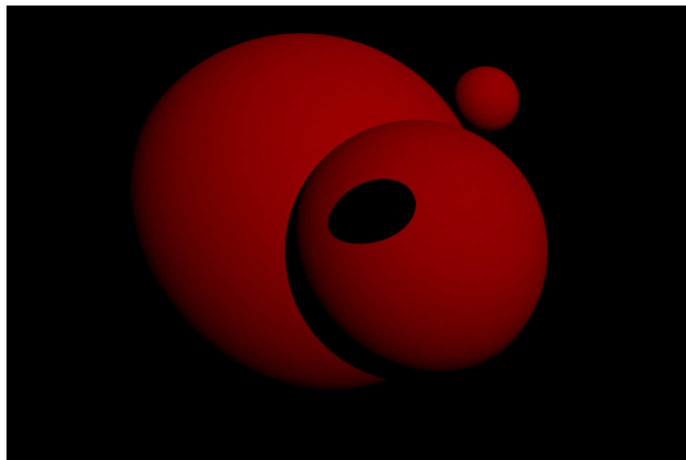


Figura 13: Render de tres esferas que proyectan sombras entre ellas, iluminadas por una fuente de luz puntual de color rojo.

3.2. Generación de imágenes por computador (CGI)

En este apartado se detalla el algoritmo de trazado de rayos utilizado para generar imágenes y hologramas, explicando cómo se simulan los rayos de luz desde la fuente hasta el detector.

En este trabajo concretamente se utiliza el algoritmo de trazado de caminos (o path tracing, en inglés), el cual simula más efectos que el trazado de rayos convencional gracias al uso de simulaciones de Monte Carlo (utilizando muestreo aleatorio).

En la Figura 14 se puede observar el posible comportamiento de un rayo en una escena sencilla.

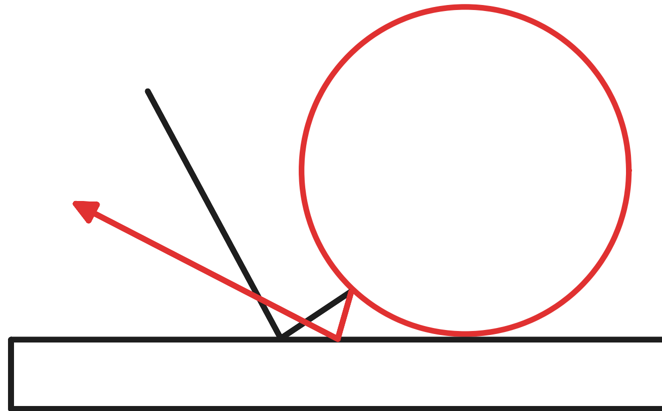


Figura 14: Diagrama que muestra uno de los posibles caminos de un rayo de luz en una escena sencilla.

Como se ha mencionado anteriormente, el algoritmo se ha implementado en el lenguaje de programación C++ debido a su alto rendimiento, control sobre conceptos de bajo nivel (como gestión de la memoria) y compatibilidad con CUDA para acelerar mediante GPUs. La implementación inicial se ha basado en el libro Ray Tracing in One Weekend [4] (el resultado de la escena final se puede observar en la Figura 19) y se han añadido más funcionalidades no incluidas en el libro, como fuentes de luz puntuales y soporte de triángulos y mallas.

El primer componente del trazador de rayos es la cámara, encargada de lanzar los rayos ya que la propagación desde la fuente de luz hasta la cámara es equivalente a la propagación desde la cámara hasta la fuente de luz. La cámara se basa en una cámara estenopeica (o cámara oscura) sin lente, aunque también es capaz de simular una lente para obtener el efecto de profundidad de campo.

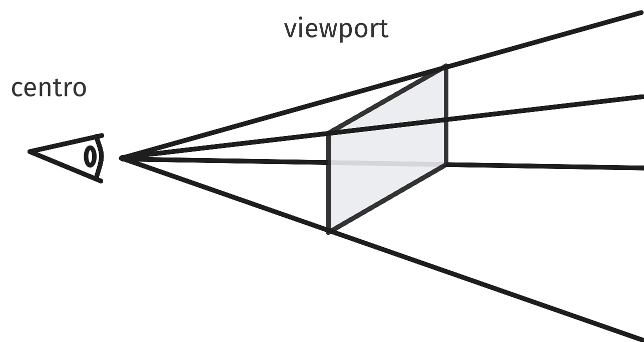


Figura 15: Diagrama del modelo de cámara utilizado, basado en una cámara estenopeica.

La cámara se define mediante su centro y su viewport (véase Figura 15). El centro de la cámara es el punto (o el centro del disco en el caso de simular una lente) desde el cuál se originan los rayos. El viewport es un rectángulo bidimensional discretizado en píxeles utilizado para proyectar la escena. Cada píxel del viewport se corresponde con el de la imagen de salida por lo que se podría hacer un símil con el sensor de la cámara. Cada píxel indica la dirección de un rayo (véase Figura 16) y una vez trazado el rayo se almacena el color resultante en la imagen de salida. Se puede obtener mayor calidad perceptual al elegir un punto aleatorio dentro del píxel en vez de su centro y se puede reducir el aliasing y aumentar la calidad al mediar el resultado de varias muestras calculadas para el mismo píxel (véase Figura 17).

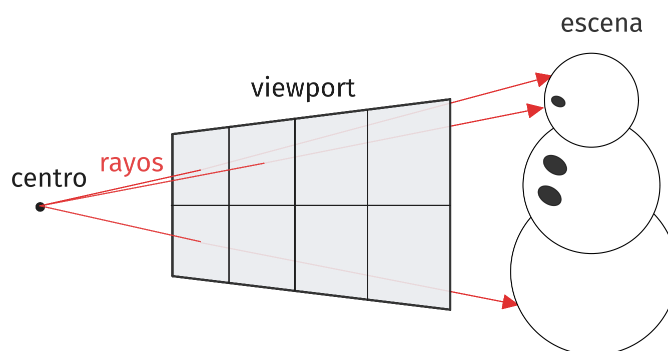


Figura 16: Diagrama del funcionamiento de la cámara en el algoritmo de trazado de rayos. Los rayos comienzan en el centro de la cámara en dirección al centro de cada píxel del viewport.

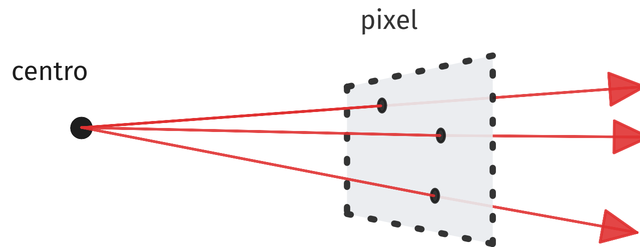


Figura 17: Diagrama del muestreo aleatorio realizado a nivel de píxel. Utilizado para reducir el aliasing y aumentar la calidad.

Una vez lanzado el rayo se comprueba si intersecta con algún objeto de la escena iterando sobre ellos y se selecciona el objeto intersectado más cercano. De la intersección se obtiene el punto, la distancia respecto al origen del rayo, la normal de la superficie y el material del objeto. Con esta información, para la iluminación ambiente, se puede calcular la atenuación y la dirección del rayo dispersado dependiendo del material. Este proceso se repite hasta que el rayo dispersado no intersecta ningún objeto (o escapa de la escena) y se atenúa con el color del cielo o, si alcanza el número máximo de rebotes definido por el trazador de rayos, la atenuación sería total. Y para la iluminación puntual, siguiendo la misma ruta del rayo de la iluminación ambiente, se comprueba si el punto es visible para la fuente de luz trazando un rayo nuevo y, si lo es, se calcula la iluminación especular y la iluminación difusa, esta última en el caso de que el material sea difuso.

También se ha creado una interfaz de usuario para agilizar el proceso de desarrollo que permite modificar en tiempo de ejecución distintos parámetros, como los ajustes de renderizado, la posición y orientación de la cámara, la iluminación y los materiales de los objetos (Figura 18).

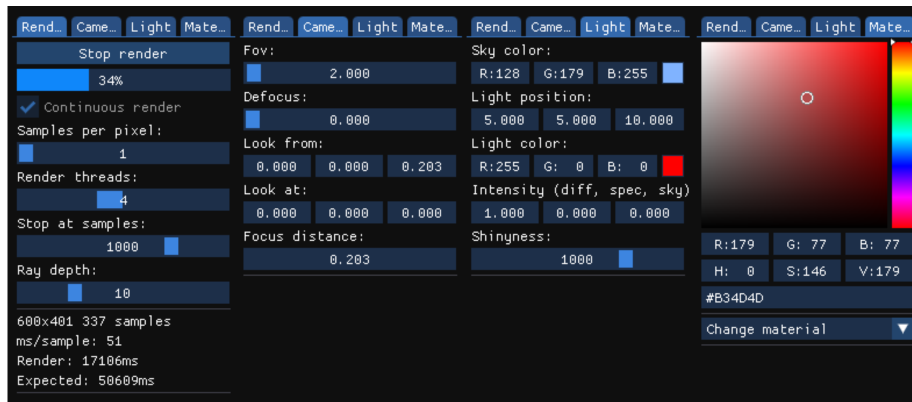


Figura 18: Parámetros de la interfaz de usuario.

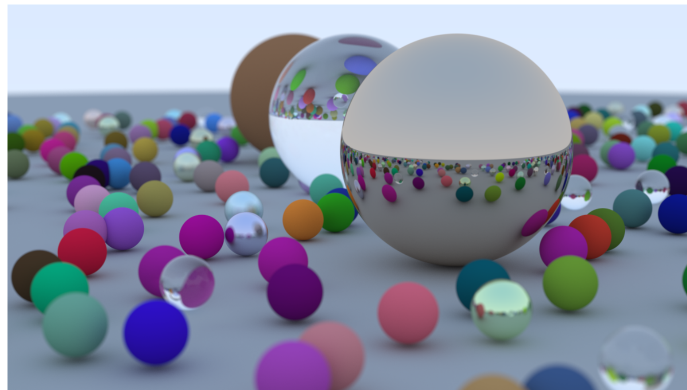


Figura 19: Render de la escena final descrita en el libro Raytracing in one Weekend [4]. Elaboración propia.

3.3. Generación de hologramas por computador (CGH)

Una vez implementado el trazador de rayos para la generación de imágenes, se ha modificado para generar hologramas. Las principales modificaciones realizadas han sido el proceso de lanzar rayos y el cambio del cálculo del color del rayo al cálculo de la amplitud y fase (mediante la Ecuación 1).

Para obtener la amplitud y la fase de cada rayo que llega a cada píxel del holograma se ha de calcular respecto a cada punto de la escena a muestrear, siendo los mismos puntos para cada píxel. Para determinar los puntos de la escena se ha utilizado la técnica de la nube de puntos, según la cual se construye una lista de puntos en las superficies de los objetos de la escena (véase Figura 20). En la implementación, la nube de puntos se calcula una sola vez y se puede generar mediante rayos perpendiculares al viewport (proyecciones

paralelas) o mediante rayos con el origen en el centro del sensor y dirección a cada píxel del viewport (proyecciones perspectiva).

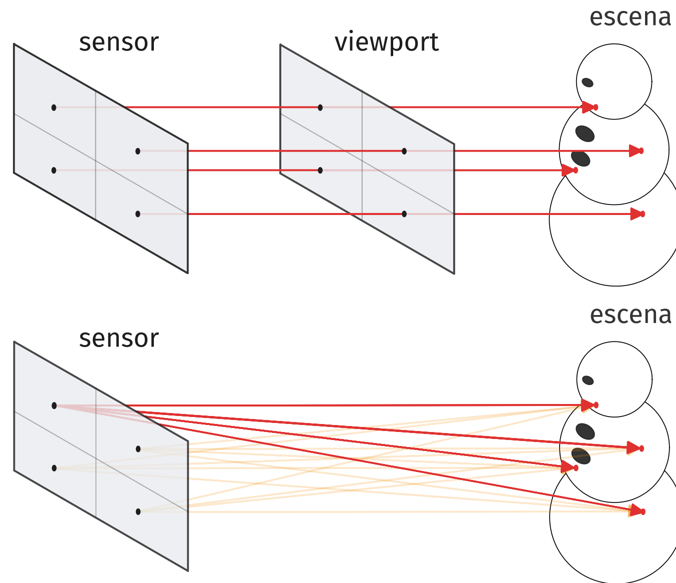
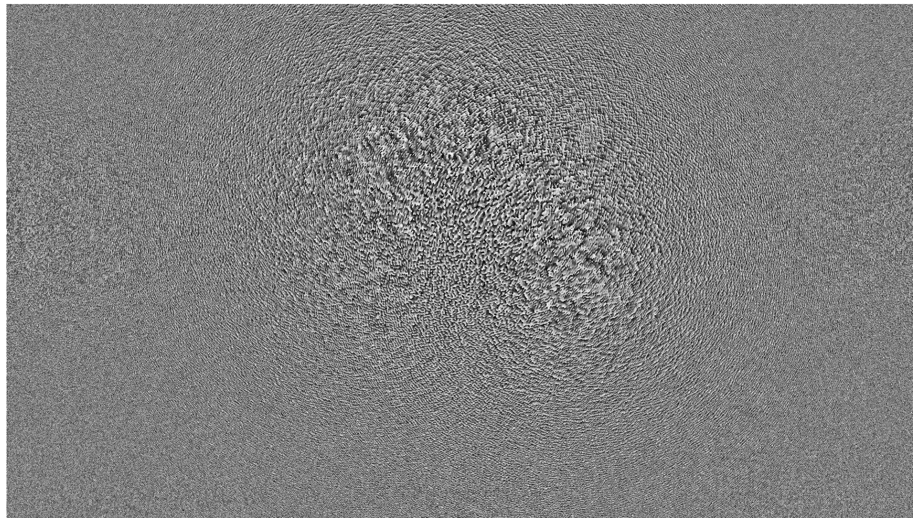


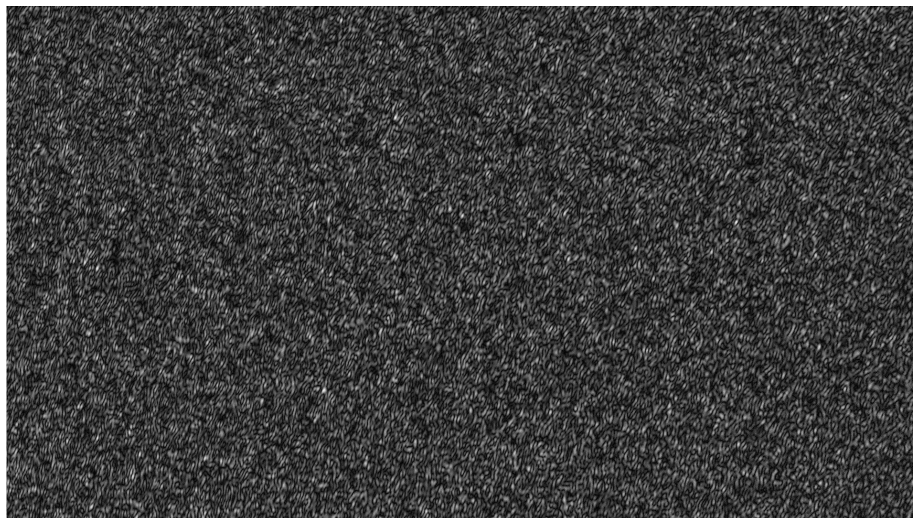
Figura 20: Diagrama del modelo de cámara utilizado para generar hologramas. Arriba, creación de la nube de puntos; abajo, desde cada píxel del sensor se lanza un rayo en dirección a cada punto de la nube de puntos.

La amplitud y la fase se calcula según lo descrito en la Ecuación 1.

Una vez generado el holograma es almacenado en dos archivos, uno contiene la amplitud y el otro la fase. Como el holograma almacenado es bidimensional, se puede almacenar en formatos de imagen como PNG. Estos archivos serán los utilizados posteriormente para la reconstrucción de la escena. Un ejemplo de la representación visual de los archivos se puede observar en la Figura 21.



(a) Amplitud



(b) Fase

Figura 21: Representación gráfica de la amplitud y fase de un holograma almacenado, generado a partir de la escena descrita en la Figura 13.

3.4. Reconstrucción de la escena

Para recuperar una imagen la escena a partir del holograma calculado se debe realizar el proceso inverso al definido en la sección anterior; esto es, se debe propagar una onda electromagnética desde el plano del CGH hasta los diferentes planos que constituyen la escena.

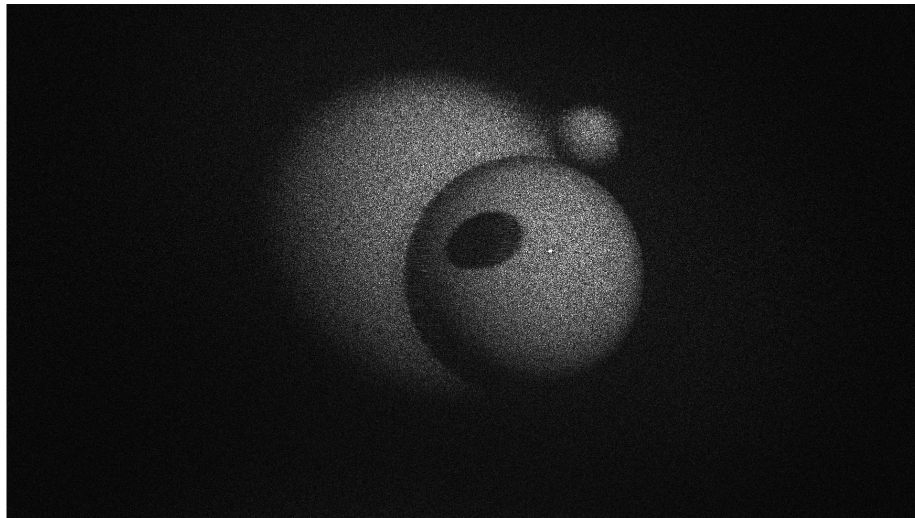
Se han utilizado dos procesos para la reconstrucción de la escena: simula-

ción mediante la propagación de ondas electromagnéticas entre dos planos y propagación en el laboratorio mediante un modulador espacial de luz (SLM), modelo PLUTO-2.1 LCOS, y un láser con longitud de onda de 632.8nm.

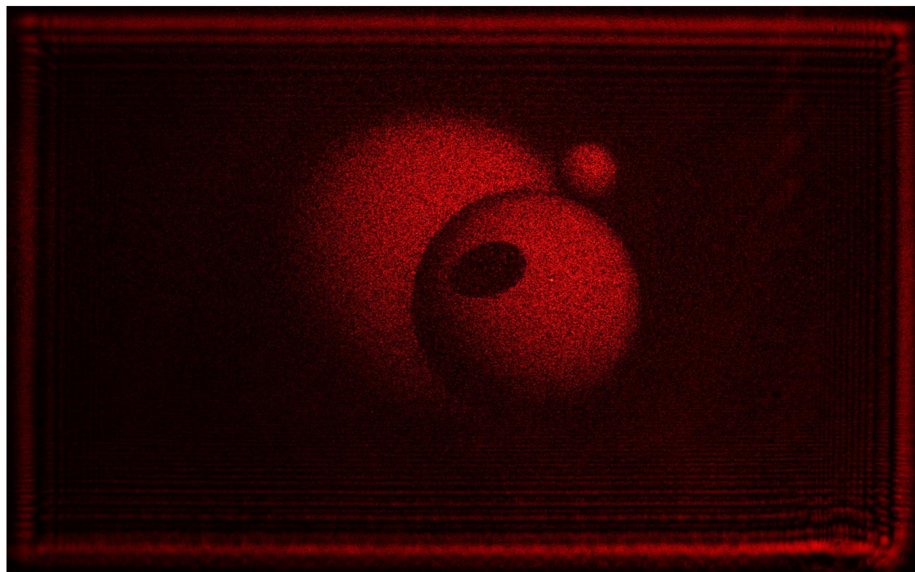
La simulación es computacionalmente mucho menos exigente que la generación, ya que podemos utilizar los algoritmos de convolución y de transformada de Fourier para reducir considerablemente el tiempo de cómputo para simular el comportamiento de la luz a su paso por el CGH [1].

La reconstrucción puede hacer uso de la fase almacenada en el holograma, de la amplitud o de ambas. Ya que el SLM del que se dispone en el laboratorio solamente es capaz de modular la fase, ambas reconstrucciones se han llevado a cabo utilizando solamente la información de la fase con el fin de obtener resultados comparables.

El resultado de reconstruir la escena previamente definida (Figura 13) y almacenada (Figura 21) se puede observar en la Figura 22. Ambas reconstrucciones están enfocadas en la esfera mediana ($z = 200$mm). En la Figura 23 se ha simulado la propagación en distintos valores de z y se puede apreciar el desenfoque de las distintas esferas.



(a) Simulación



(b) Propagación mediante un SLM en el laboratorio

Figura 22: Resultados de la reconstrucción de la escena tanto por simulación como en el laboratorio.

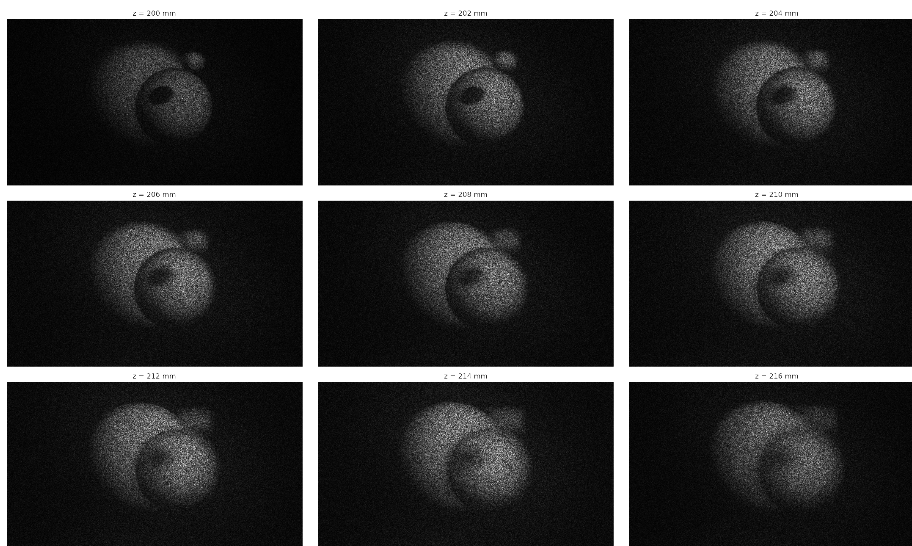


Figura 23: Simulación de la propagación para distintos valores de z.

4. Técnicas de paralelización

La computación paralela es un tipo de computación en la que muchos cálculos o procesos se pueden llevar a cabo simultáneamente [11]. Existen varias técnicas de paralelización entre las que se encuentra la que se utilizará en este trabajo, el paralelismo de datos. Esta técnica consiste en dividir los datos entre distintos hilos de procesamiento. Por ejemplo, se podría dividir una imagen en píxeles, y operar a nivel de píxel o en bloques de píxeles.

4.1. CPU

Las unidades centrales de procesamiento (CPUs) actuales cuentan con múltiples núcleos, lo que permite la ejecución paralela de múltiples hilos de procesamiento. Cada hilo puede ejecutar una serie de instrucciones de manera independiente, permitiendo la realización de tareas en paralelo.

Para aprovechar los diferentes núcleos de la CPU es necesario dividir el trabajo para diferentes hilos (o threads, en inglés). En el caso de un trazador de rayos, el método más sencillo de distribuir el trabajo es dividiendo la imagen en píxeles y asignando a cada hilo un rango de píxeles sobre los que operar. Este método cuenta con la limitación de que puede darse el caso de que un hilo acabe antes que otro.

Para solucionar este problema se puede introducir el uso de un grupo de hilos

(o thread pool, en inglés), al cual se le puede asignar una serie de tareas que los hilos que lo forman ejecutan. Al dividir el trabajo en más tareas que hilos, se soluciona la limitación anterior. Se ha de tener en cuenta que la gestión de los hilos y las tareas tiene un coste computacional.

En el trazador de rayos implementado se ha dividido la imagen en líneas de píxeles y se ha optado por el uso de un grupo de hilos. La elección se debe principalmente a que el primer hilo suele terminar mucho antes que el último ya que en la mayoría de las escenas no se incluyen objetos en la parte superior. Esto no ocurre en el caso del trazador de rayos modificado para generar hologramas, ya que cada píxel lanza un rayo a cada punto y los hilos terminan en tiempos similares. Por este motivo y por el coste extra del grupo de hilos, se ha optado por no utilizar grupos de hilos en la generación de hologramas.

4.2. GPU

Las unidades de procesamiento gráfico (GPUs) son componentes hardware diseñados originalmente para la renderización de gráficos. Sin embargo, debido a su arquitectura masivamente paralela, también son altamente eficaces en la realización de cálculos matemáticos complejos y tareas que pueden beneficiarse del paralelismo masivo.

Las GPUs están compuestas de miles de núcleos más simples y especializados que los de las CPUs, como se puede apreciar en la Figura 24. Esto permite la ejecución de miles de hilos simultáneamente, comparado con las CPUs que no suelen superar los 256 hilos.

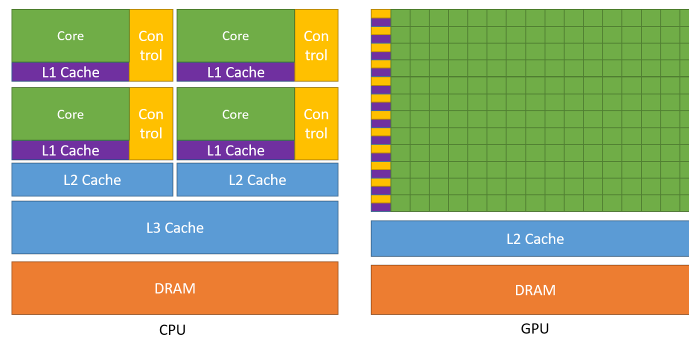


Figura 24: Esquema de la distribución de los recursos del chip para una CPU y para una GPU. Fuente: Nvidia [12].

Para poder utilizar una GPU, es necesario el uso de bibliotecas o herramientas especiales que faciliten la programación en GPUs, como CUDA, exclusivo de

Nvidia, o OpenCL. Estas herramientas permiten escribir código el cual puede ser ejecutado en la GPU. En el caso de CUDA, el lenguaje es similar a C++. Por esto y porque la GPU de la que se dispone es Nvidia, se ha escogido CUDA para este trabajo [12].

Como se ha mencionado anteriormente, el computo de un trazador de rayos puede ser dividido entre cada uno de los píxeles, lo que hace que las GPUs puedan acelerar significativamente el algoritmo. Cada píxel puede ser procesado por un núcleo diferente.

CUDA es una plataforma de computación paralela y una API creada por Nvidia que permite a los desarrolladores utilizar la GPU para tareas de computación general. Proporciona una extensión al lenguaje de programación C++ que incluye una serie de funciones y directivas para interactuar con la GPU. En CUDA, el código se divide en tres tipos, definidos al declarar la función que lo contiene. El tipo host (anfitrión) es el código que se ejecuta en la CPU y no se distingue de C++. El tipo device (dispositivo) es el código que se ejecuta en la GPU. Y el tipo kernel es el código que se llama desde la CPU y asigna el trabajo a la GPU.

4.3. Implementación

El código del proyecto puede ser obtenido en el siguiente repositorio `https://github.com/ElDigoXD/sfml_rt`.

La paralelización mediante hilos se ha implementado asignando un rango de líneas a cada hilo, las cuales se calculan y almacenan en memoria compartida.

La paralelización mediante grupos de hilos se ha implementado gracias a la librería BS::thread_pool. Al grupo se le añade número de inicio, número de fin, el número de bloques en los que dividir el trabajo y una función que, dado un número de línea, la calcula. A continuación, se muestra un ejemplo de código:

```
BS::thread_pool pool{numero_de_hilos};  // inicialización
pool.detach_loop(0,                     // índice de inicio
                 altura_de_la_imagen,   // índice de fin
                 [&](int idx_linea) {   // función
    camera.render_CGH_line(&output_buffer[idx_linea * image_width],
                           *escena, nube de puntos, idx_linea);
                 },
                numero_de_hilos * 4);   // número de divisiones
pool.wait();                            // espera hasta acabar
```

Por último, la paralelización en CUDA se ha implementado mediante un kernel. Una vez los datos necesarios están almacenados en memoria compartida o memoria de la GPU, se ejecuta un kernel en bloques suficientes para cubrir el tamaño de la imagen, en el que está definido como calcular un píxel. A continuación, se muestra un ejemplo de kernel:

```
__global__ void        // "__global__" indica que es un kernel
render(thrust::complex<double> *output_buffer,
       int max_x, int max_y, HoloCamera *d_camera,
       HittableList **d_scene, Point3 *point_cloud,
       int point_cloud_size, curandState *global_state) {

    // Obtiene el índice del pixel a computar.
    uint i = threadIdx.x + blockIdx.x * blockDim.x;
    uint j = threadIdx.y + blockIdx.y * blockDim.y;
    if ((i >= max_x) || (j >= max_y)) return;
    uint pixel_idx = j * max_x + i;

    // Obtiene la instancia del generador de números aleatorios.
    curandState local_state = global_state[pixel_idx];

    // Computa el valor del píxel.
    const auto *scene = *d_scene;
    auto slm_pixel_center = d_camera->slm_pixel_00_location
                        + (i * d_camera->slm_pixel_delta_x)
                        + (j * d_camera->slm_pixel_delta_y);

    // Por cada punto, agrega el valor computado al buffer.
    for (int p_idx = 0; p_idx < point_cloud_size; p_idx++) {
        auto ray = Ray(slm_pixel_center,
                       point_cloud[p_idx] - slm_pixel_center);
        const thrust::complex<double> cgh =
            d_camera->ray_wave_cgh(ray, point_cloud[p_idx],
                                   d_camera->max_depth,
                                   *scene, &local_state);
        output_buffer[pixel_idx] += cgh;
    }

    // Normaliza el buffer
    output_buffer[pixel_idx] /= (d_camera->slm_width_in_px
```

```
                                              * d_camera->slm_height_in_px);
}
```

En la Figura 25 se muestra un diagrama que compara el flujo de ejecución del programa en C++ y CUDA, indicando si se ejecuta en la CPU o en la GPU.

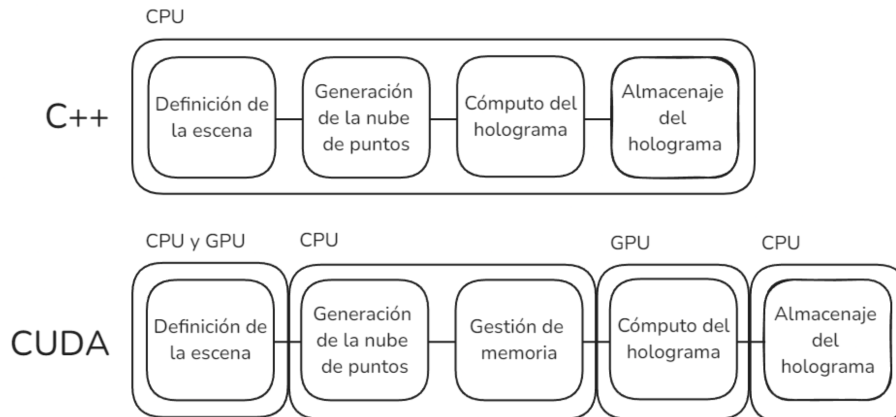


Figura 25: Diagrama de las distintas etapas del programa en C++ y CUDA, y en que procesador son ejecutadas.

5. Resultados

5.1. Tablas comparativas de tiempos.

Para la obtención de los tiempos de este apartado se han utilizado dos ordenadores con las siguientes CPUs y GPUs:

- Ordenador 1:
  - CPU: Intel i7 7700K (8 hilos, 4.4GHz)
  - GPU: Nvidia RTX 3070 (5888 núcleos)
- Ordenador 2:
  - CPU: Intel i7 9900K (16 hilos, 4.7GHz)
  - GPU: Nvidia RTX 2060 (1920 núcleos)

Todos los tiempos se han obtenido al calcular la escena descrita anteriormente (Figura 13) con un sensor de 1920 x 1080 píxeles.

El primer resultado obtenido (Figura 26) compara las dos CPUs con distintos números de puntos y el número máximo de hilos menos uno. Se puede observar una correlación lineal entre tiempo y número de puntos en las dos CPUs, esta correlación se utilizará en los siguientes resultados para obtener los datos en tiempos aceptables. Exceptuando el primer resultado de cada CPU (por poca carga de trabajo), la media de la primera CPU es de 2.85 puntos por segundo, comparado con los 27.95 de la segunda (9.8 veces más rápida).

Points	Threads	Time (s)	P/s
39	7	17	2,294118
152	7	59	2,576271
359	7	129	2,782946
628	7	224	2,803571
987	7	323	3,055728
1920	7	656	2,926829
3954	7	1331	2,970699
CPU: 7700K		**AVG:**	**2,852674**

Points	Threads	Time (s)	P/s
38	15	2	19
151	15	6	25,16667
356	15	13	27,38462
620	15	21	29,52381
975	15	34	28,67647
1901	15	67	28,37313
3918	15	137	28,59854
CPU: 9900K		**AVG:**	**27,95387**

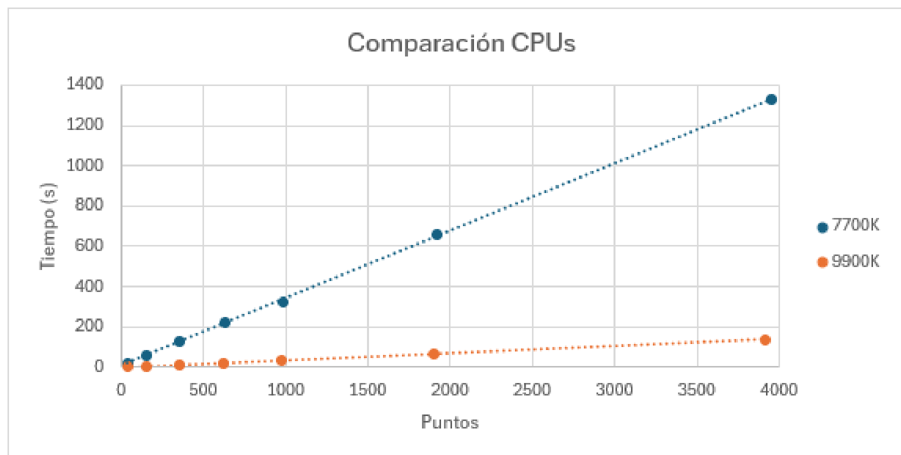


Figura 26: Resultados obtenidos para ambas CPUs.

En la Figura 27 se comparan las dos GPUs de la misma manera. La media de la primera GPU es de 91.26 puntos por segundo y la de la segunda es de 50.14, siendo la primera 1.82 veces más rápida respecto a la segunda y 3.27 veces respecto a la CPU más rápida.

Points	Time (s)	P/s
987	12	82,25
3957	41	96,51
15764	171	92,19
24369	259	94,09
GPU: 3070	**AVG:**	**91,26**

Points	Time (s)	P/s
975	20	48,75
3918	78	50,23
15595	307	50,80
24369	480	50,77
GPU: 2060	**AVG:**	**50,14**

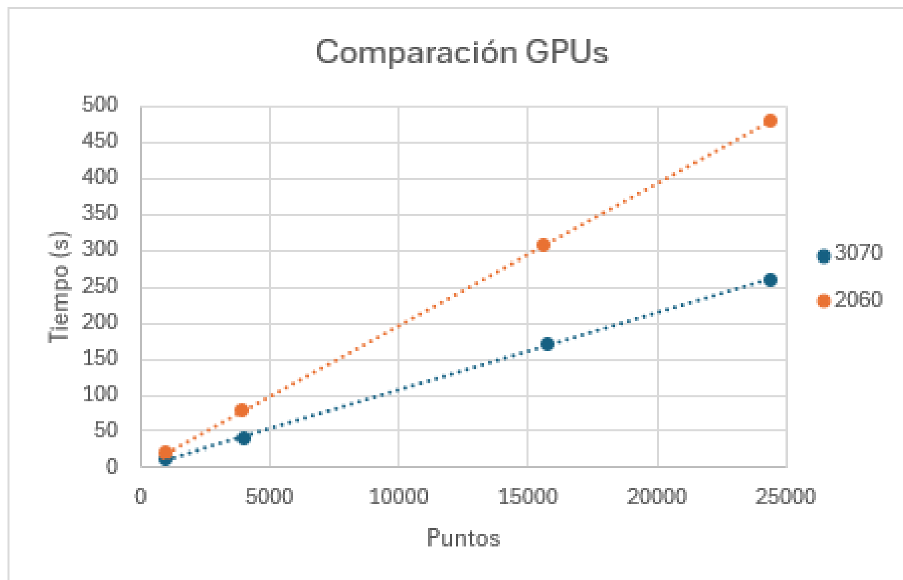


Figura 27: Resultados obtenidos para ambas GPUs.

Finalmente se han obtenido resultados para distintos números de hilos de las CPUs. Como se puede ver en la Figura 28, el aumento del número de hilos conlleva a un aumento del rendimiento no lineal, esto puede deberse al coste de crear y gestionar hilos, la contención por recursos compartidos y el aumento de los fallos de caché y de los conflictos de acceso a la memoria.

Points	Threads	Time (s)	P/s
3918	1	1182	3,314721
3918	2	586	6,686007
3918	4	307	12,76221
3918	8	169	23,18343
3918	12	151	25,94702
3918	16	131	29,9084

Points	Threads	Time (s)	P/s
359	1	526	0,68251
359	2	287	1,250871
359	4	173	2,075145
359	6	123	2,918699
359	8	108	3,324074

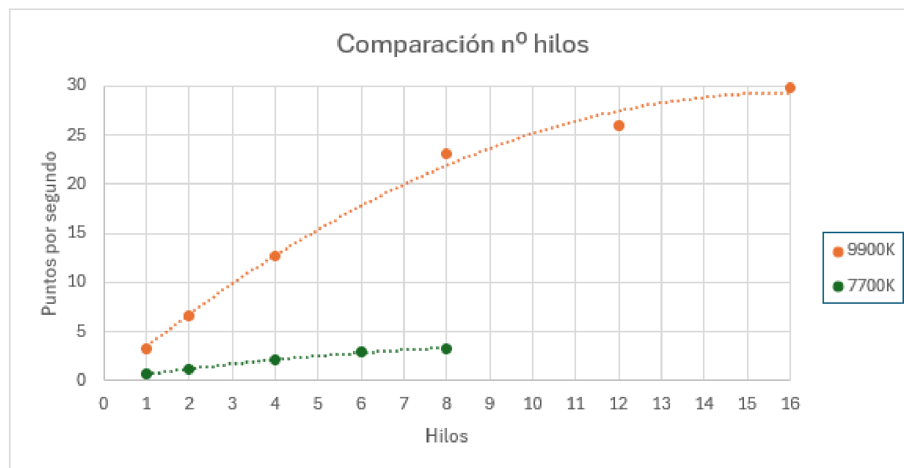


Figura 28: Resultados obtenidos para distintos números de hilos de ambas CPUs.

En la Figura 29 se comparan los resultados tanto de CPUs como de GPUs en una misma gráfica con escala logarítmica en ambos ejes. Se incluye un resultado adicional para la GPU 2060 (455 234 puntos en 9006 segundos (o 2.5 horas)), que también sigue la correlación lineal.

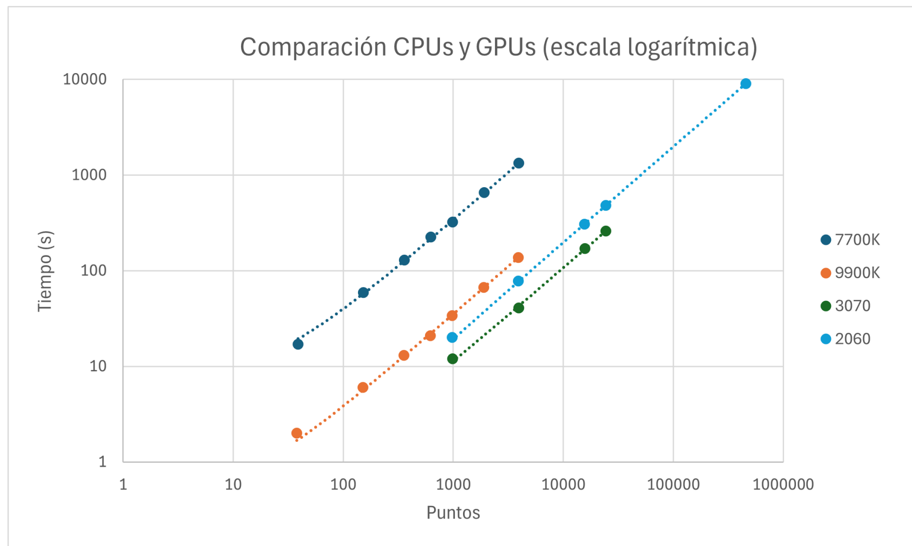


Figura 29: Gráfico que incluye los resultados de CPUs y GPUs utilizando una escala logarítmica en ambos ejes.

El número de puntos determina la calidad visual del holograma generado. En la Figura 30 se pueden observar distintos resultados con distintos números de puntos y se puede ver que, a mayor cantidad de puntos, mayor calidad visual hasta que se llega a un punto en el que no se aprecia el aumento de calidad.

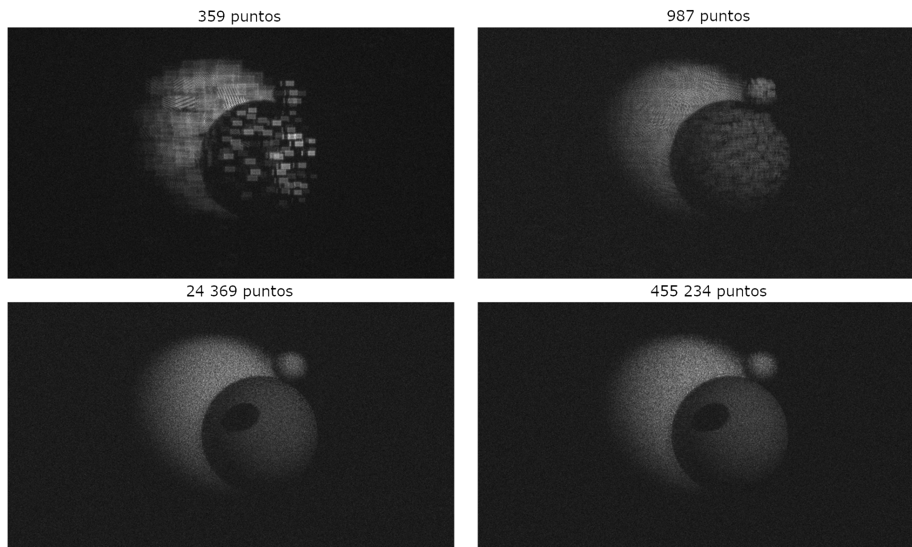


Figura 30: Holograma de una misma escena obtenido con distintos números de puntos.

6. Conclusiones

En este trabajo se ha conseguido implementar satisfactoriamente un trazador de rayos para generación de imágenes sintéticas (CGI) y adaptarlo para cumplir los requisitos de la generación de hologramas por computador (CGH).

El proceso de cálculo de hologramas generados por computador es una tarea que requiere de una considerable cantidad de recursos computacionales debido a su alta complejidad. El uso de técnicas de paralelización ha demostrado ser una estrategia eficaz para acelerar significativamente este proceso ya que las técnicas utilizadas (trazado de rayos y nube de puntos) son altamente paralelizables. A través de la distribución del trabajo en múltiples núcleos de CPU se ha conseguido una mejora sustancial en el tiempo de procesamiento de 9 veces más rápido respecto a un solo núcleo.

Por otro lado, el uso de GPUs en el cálculo de CGHs introduce una capa adicional de complejidad en el proyecto debido a las limitaciones impuestas por CUDA. Estas limitaciones incluyen la necesidad de utilizar una GPU de marca Nvidia, la instalación y configuración del toolkit/SDK de CUDA, la duplicación de partes del código para CPUs y GPUs, y la gestión manual de la memoria entre anfitrión y dispositivo. A pesar de esta complejidad, el uso de GPUs es altamente beneficioso, ya que están especialmente diseñadas para realizar cálculos en paralelo y el problema a resolver es altamente paralelizable. La mejora en tiempo de procesamiento utilizando GPUs ha sido de 3 veces más rápido respecto a la paralelización en CPU. Esta diferencia hace que integrar el uso de GPUs en el proyecto sea justificable.

Los resultados que se han obtenido han sido coherentes. Una representación de la escena utilizada se ha conseguido obtener mediante tres métodos distintos sin grandes diferencias: CGI (Figura 13), CGH propagado mediante simulación (Figura 22a) y CGH propagado en el laboratorio (Figura 22b). Los resultados relacionados con el tiempo de cómputo coinciden con los esperados: existe una correlación lineal entre tiempo y número de puntos, una mejor CPU o GPU (mejor generación y gama) ofrece mayor rendimiento y el número de hilos utilizado en CPUs aumenta el rendimiento, aunque no de manera lineal.

7. Calendario

- Enero (2024):

- Inicio del proyecto.
- Implementación del libro Raytracing in one Weekend [4].
- Diseño y desarrollo de la interfaz gráfica.
- Implementación de la paralelización en CPUs.

- Febrero:
  - Instalación del entorno CUDA (toolkit/SDK).
  - Adaptación del código de C++ a CUDA.
  - Implementación de la paralelización en GPUs.
  - Implementación de una fuente de luz puntual.
  - Implementación de triángulos, mallas e importación de archivos Wavefront obj.
- Marzo:
  - Implementación de la generación de hologramas.
  - Implementación de la paralelización de CGH en CPUs.
- Mayo:
  - Implementación del algoritmo de propagación.
  - Implementación de la paralelización de CGH en GPUs.
- Junio:
  - Limpieza del código.
  - Redacción de la memoria.
- Julio:
  - Redacción de la memoria.
- Agosto:
  - Redacción de la memoria.

- Obtención de resultados.

8. Trabajo futuro

Se han identificado áreas en las que se podrían realizar futuras mejoras, como la adaptación de la arquitectura del programa, la integración de nuevas técnicas de optimización y la incorporación de nuevas características y funcionalidades. Estas iniciativas buscan optimizar y expandir las capacidades del sistema:

- Cambiar la arquitectura del programa para adaptarla a los conocimientos adquiridos sobre GPUs.
- Modificar el planteamiento de los materiales para seguir estándares como OpenPBR o Disney.
- Utilizar otras técnicas de optimización además de paralelismo.
- Definir las escenas fuera del código fuente.
- Crear una interfaz que funcione con CGHs y GPU.
- Investigar si es posible aumentar la precisión de CGHs adaptando la escala de medidas o si es posible aumentar el rendimiento con números de precisión simple (FP32) en GPUs.
- Solucionar los problemas que impiden utilizar triángulos en CGH.
- Añadir trazado de rayos optimizado mediante la librería OptiX.
- Añadir fuentes de luz no puntuales.

9. Dificultades encontradas

Durante la realización de este trabajo se han encontrado distintas dificultades descritas a continuación, clasificadas como normales, medias, difíciles y muy difíciles:

- La primera dificultad encontrada ha sido el uso de un sistema para gestionar el proyecto (**difícil**), ya que el estándar de C++ no incluye uno. Se ha utilizado CMake ya que es el más popular, permite añadir librerías y soporta el uso de CUDA. Aun utilizando CMake, es necesario conocer

varios argumentos de los distintos compiladores (clang para C++ y nvcc para CUDA) para conseguir que funcione correctamente.

- La principal dificultad relacionada con CUDA es la gestión de memoria (**difícil**). Versiones modernas de CUDA soportan memoria gestionada automáticamente, lo cual simplificaría el problema si no fuera porque se hace uso de características no soportadas, como el polimorfismo para materiales y objetos de la escena. Por ello se ha tenido que gestionar la memoria manualmente.

- Otra dificultad relacionada con CUDA es la depuración (**difícil**). Las herramientas de depuración para GPUs no son tan maduras como las de CPUs y esto hace que sea complicado encontrar errores que no ocurren al ejecutar el programa en CPU.

- La librería estándar de C++ para CUDA no incluye algunas funcionalidades (**dificultad normal**) como pueden ser vectores (std::vector) y números complejos (std::complex). La solución para el primer caso es hacer uso de matrices y, para el segundo, utilizar otra opción ofrecida en CUDA (thrust::complex).

- La depuración en CGH (**difícil**), ya que se trabaja con amplitudes y fases y hasta que no se propaga (ya sea por simulación o en el laboratorio) no se puede comprobar su correcto funcionamiento. Ha sido necesario utilizar la propagación en el laboratorio para encontrar un problema que aumentaba todas las distancias en un factor de 4.

- La última dificultad encontrada ha sido poder utilizar triángulos en CGH (**muy difícil**). No se ha conseguido solucionar un problema que hacía aparecer dos triángulos en vez de uno.

Referencias

[1] Joseph Goodman. *Introduction to Fourier Optics*. WH Freeman, mayo de 2017.

[2] David Blinder et al. “Signal processing challenges for digital holographic video display systems”. En: *Signal processing. Image communication* 70 (feb. de 2019), págs. 114-130. DOI: 10.1016/j.image.2018.09.014. URL: https://www.sciencedirect.com/science/article/pii/S0923596518304855.

[3] David Blinder et al. “The state-of-the-art in computer generated holography for 3D display”. En: *Light: Advanced Manufacturing* 3.3 (ene. de 2022), pág. 1. DOI: 10.37188/lam.2022.035. URL: https://doi.org/10.37188/lam.2022.035.

[4] Peter Shirley, Trevor David Black y Steve Hollasch. *Ray Tracing in One Weekend*. Abr. de 2024. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[5] Matt Pharr, Wenzel Jakob y Greg Humphreys. *Physically Based Rendering, fourth edition*. MIT Press, mayo de 2023.

[6] David Blinder et al. “Photorealistic computer generated holography with global illumination and path tracing”. En: *Optics letters/Optics index* 46.9 (abr. de 2021), pág. 2188. DOI: 10.1364/ol.422159. URL: https://doi.org/10.1364/ol.422159.

[7] Alan Chalmers, Timothy Davis y Erik Reinhard. *Practical parallel rendeRiNg*. A K Peters/CRC Press, jun. de 2002.

[8] Brent Burley y Walt Disney Animation Studios. “Physically-based shading at disney”. En: *Acm Siggraph*. Vol. 2012. 2012, págs. 1-7.

[9] Academy Software Foundation. *OpenPBR Surface Specification*. Jun. de 2024. URL: https://academysoftwarefoundation.github.io/OpenPBR/.

[10] James F. Blinn. “Models of light reflection for computer synthesized pictures”. En: *Association for Computing Machinery* (jul. de 1977). DOI: 10.1145/563858.563893. URL: https://doi.org/10.1145/563858.563893.

[11] George S. Almasi y Allan Gottlieb. *Highly parallel computing*. Addison Wesley Publishing Company, ene. de 1989.

[12] Nvidia. *CUDA Toolkit Documentation 12.6*. en. Ago. de 2024. URL: https://docs.nvidia.com/cuda/.

10. Anexos

10.1. Librerías y licencias

A continuación, se enumeran las librerías utilizadas junto a sus licencias:

- Simple and Fast Multimedia Library (SFML):
    - Descripción: Proporciona una interfaz sencilla con el fin de facilitar el desarrollo de aplicaciones multimedia. Utilizada en este proyecto para la creación de ventanas y muestra de imágenes.
    - Repositorio: https://github.com/SFML/SFML

- Licencia: zlib - https://github.com/SFML/SFML/blob/master/license.md

- Dear ImGui:
  - Descripción: Proporciona componentes de interfaz gráfica. Diseñada para permitir iteraciones rápidas y crear herramientas de visualización/depuración. Utilizada en este proyecto para la modificación de parámetros en tiempo de ejecución.
  - Repositorio: https://github.com/ocornut/imgui
  - Licencia: MIT - https://github.com/ocornut/imgui/blob/master/LICENSE.txt

- ImGui-SFML:
  - Descripción: Permite utilizar Dear ImGui con SFML.
  - Repositorio: https://github.com/SFML/imgui-sfml
  - Licencia: MIT - https://github.com/SFML/imgui-sfml/blob/master/LICENSE

- BS::thread_pool:
  - Descripción: Proporciona grupos de hilos.
  - Repositorio: https://github.com/bshoshany/thread-pool
  - Licencia: MIT - https://github.com/bshoshany/thread-pool/blob/master/LICENSE.txt

- Argparse:
  - Descripción: Proporciona un método sencillo de registrar parámetros de línea de comando.
  - Repositorio: https://github.com/morrisfranken/argparse
  - Licencia: Apache 2.0 - https://github.com/morrisfranken/argparse/blob/master/LICENSE

- stb_image, stb_image_write:

- Descripción: Librerías para leer y escribir imágenes en gran variedad de formatos.
- Repositorio: `https://github.com/nothings/stb`
- Licencia: Dominio público o MIT - `https://github.com/nothings/stb/blob/master/LICENSE`

- tinyobjloader:
  - Descripción: Permite cargar archivos Wavefront obj. Utilizado en este proyecto para importar mallas de triángulos.
  - Repositorio: `https://github.com/tinyobjloader/tinyobjloader`
  - Licencia: MIT - `https://github.com/tinyobjloader/tinyobjloader/blob/release/LICENSE`
- Librerías de Python: numpy, matplotlib, pillow, tkinter.