SÍNTESIS DE UNA ESCENA MEDIANTE *RAY TRACING* EN *FPGA*

Rafael Andrés Herrera Guaitero
Juan Manuel Gómez Cruz
Daniel Andrés Sáenz Rodelo

Trabajo de Grado
para optar al título de Ingeniero Electrónico

Director
Ingeniero Juan Carlos Giraldo, MSc

Co-director
Ingeniero Andrés Maldonado, MSc

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA DEPARTAMENTO DE ELECTRÓNICA
BOGOTA, COLOMBIA
DICIEMBRE 2019

Dedicado a
nuestra familia

AGRADECIMIENTOS

Primeramente, agradecer a Dios por permitirnos cumplir satisfactoriamente con todo lo propuesto en este trabajo de grado. Gracias a nuestros padres, hermanos y demás integrantes del núcleo familiar por apoyarnos y sostenernos en cada etapa del desarrollo de nuestras vidas y acompañarnos durante nuestro proceso formativo como profesionales. Al Ing. Juan Carlos Giraldo por todo el apoyo, esfuerzo y dedicación frente a cada situación que se presentó durante el desarrollo del proyecto, al Ing. Andrés Maldonado por asumir el reto de ser nuestro director a pesar de la distancia y dedicar tiempo para ayudarnos a sobreponernos frente a las dificultades presentadas. Por último, agradecerle a la Pontificia Universidad Javeriana por la educación brindada y los conocimientos que permitieron culminar con todo el trabajo realizado durante el segundo semestre del año 2019.

TABLA DE CONTENIDOS

LISTA DE TABLAS

LISTA DE FIGURAS

ACRÓNIMOS Y ABREVIACIONES

AGP	*Accelerated Graphics Port*
BSP Trees	*Binary Space-Partitioning Tree*
FPGA	*Field-Programmable Gate Array*
GNU	*GNU is Not Unix* (sigla recurrente)
GPROF	*GNU Profiler*
GPU	*Graphics Processing Unit*
GUI	*Graphical User Interface*
IP	*Intellectual property*
ISA	*Industry Standard Architecture*
MIMD	*Multiple Instruction, Multiple Data*
NFF	*Neutral File Format*
OpenCV	*Open Computer Vision*
PCI	*Peripheral Component Interconnect*
PDF	*Portable Document Format*
PLL	*Phase-Locked Loop*
RAM	*Random Access Memory*
RGB	*Red, Green, Blue*
RMS	*Root Mean Square*
SaarCOR	*Saarbrücken's Coherence Optimized Ray Tracer*
SPD	*Standard Procedural Database*
VGA	*Video Graphics Array*

RESUMEN

El algoritmo de trazados de rayos o *ray tracing* es una de las técnicas de síntesis de imagen con mayor nivel de detalle y realismo. Sin embargo, debido a su alto costo computacional y altas latencias no es muy popular en trabajos de producción masiva y constantemente se ha visto relegado a producciones *offline*. Teniendo en cuenta lo anterior, el presente trabajo de grado se centra en diseñar e implementar un prototipo funcional que sintetice una escena mediante *FPGA*, y un sistema de referencia basado en *software*, que permita analizar e identificar los cuellos de botella presentados en este algoritmo. Basados en los resultados, se encontró que el producto punto es la función más llamada dentro de la ejecución del algoritmo y el *test* de intersección la función donde el programa mantenía la mayor parte de su tiempo. Además, al comparar los tiempos de *rendering* de ambos sistemas se logró determinar que en todas las escenas de prueba el tiempo fue menor en la *FPGA*, logrando ganancias de hasta 18 veces frente al sistema de referencia (*software)*.

INTRODUCCIÓN AL RAY TRACING

1.1 ¿Qué es *ray tracing?*

Actualmente en la industria audiovisual existen diversas técnicas de síntesis y procesamiento de imágenes, que tienen como objetivo el detalle y realismo del producto. Una de estas técnicas está basada en un algoritmo conocido como trazado de rayos (que se referenciará de ahora en adelante como *ray tracing*). En la literatura científica y técnica, la palabra *ray tracing* tiene dos acepciones. Una de ellas se usa en la comunidad de computación electromagnética como método de cálculo de fenómenos de propagación cuando la longitud de onda es muy pequeña con relación al tamaño de las estructuras; sin embargo, para efectos de este trabajo se usará la palabra *ray tracing*, ampliamente aceptada en la comunidad de ciencias de la computación y procesamiento de imágenes como un método de *rendering*.

En este sentido, *ray tracing* simula los fenómenos ópticos de reflexión, refracción y absorción de la luz utilizando modelos de iluminación. Estos modelos permiten computar la contribución de color para reproducir, entre otros, efectos visuales de iluminación global [1] y de oclusión o sombras, ver Figura 1. La escena a la cual se le hace el proceso de *rendering*[1] está compuesta de una lista de primitivas geométricas tales como polígonos, esferas, cilindros y conos. De hecho, cualquier clase de objeto puede ser usado como primitiva mientras que el algoritmo de cálculo de intersección de un rayo con dicho objeto sea computable [2].

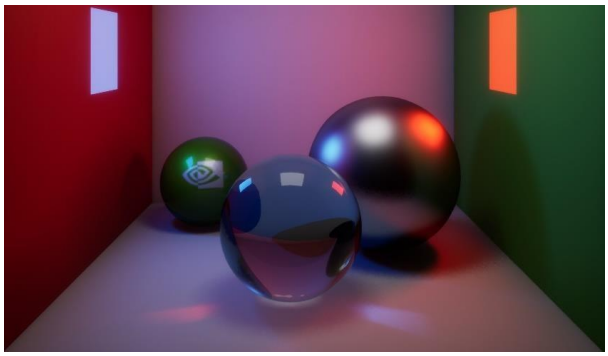

Figura 1. Imagen sintetizada mediante el algoritmo *ray tracing* [3]

1.2 Explicación del algoritmo

Existen tres operaciones fundamentales que deben ser realizadas en la síntesis de imágenes usando *ray tracing*

1. Cálculo de intersecciones de rayos con los objetos en la escena (rayo primario con esfera naranja en Figura 2)
2. Cálculo de rayos de sombra para determinar la oclusión de objetos en la dirección de cada una de las fuentes de luz (rayos violetas en Figura 2) y de rayos secundarios para simular fenómenos de transparencia y reflexión sobre las superficies.
3. Cálculo de color de cada píxel de la pantalla, basado en las contribuciones de rayos secundarios a lo largo de toda la trayectoria, según nivel de profundidad definido.

[1] *Rendering* es el proceso de generar una imagen fotorrealista a partir de un modelo 2D o 3D [35].

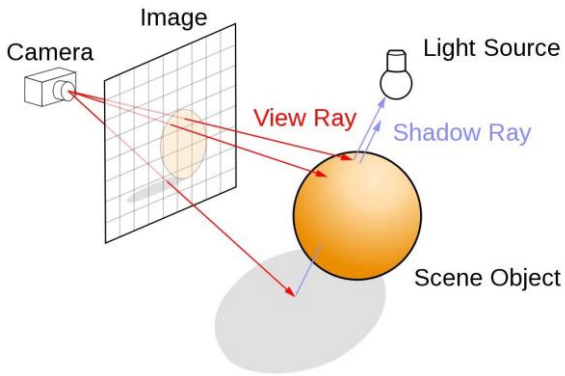

Figura 2. Técnica del *ray tracing* [4]

El algoritmo consiste en proyectar un rayo desde el observador hacia cada píxel de la pantalla y comprobar si golpea algún objeto de la escena. Esto implica que la cantidad de rayos que se lanza corresponde a $(n \times p)$, siendo la variable (n) el número de objetos y la variable (p) el número de píxeles [5]. Si el rayo interseca la superficie y la distancia entre el observador y el punto de impacto es la menor, se almacena dicho impacto y se traza un conjunto de rayos desde este punto hacia las fuentes de luz y se comprueba si hay sombra. Posteriormente, a partir de las características del material, se calcula el color y la intensidad con que las fuentes de luz afectan la superficie basándose en un modelo de iluminación o mejor conocido técnicamente como *shading*, luego se retorna la información del píxel [1]. El *flowchart* que muestra el comportamiento del algoritmo es el de la Figura 3.

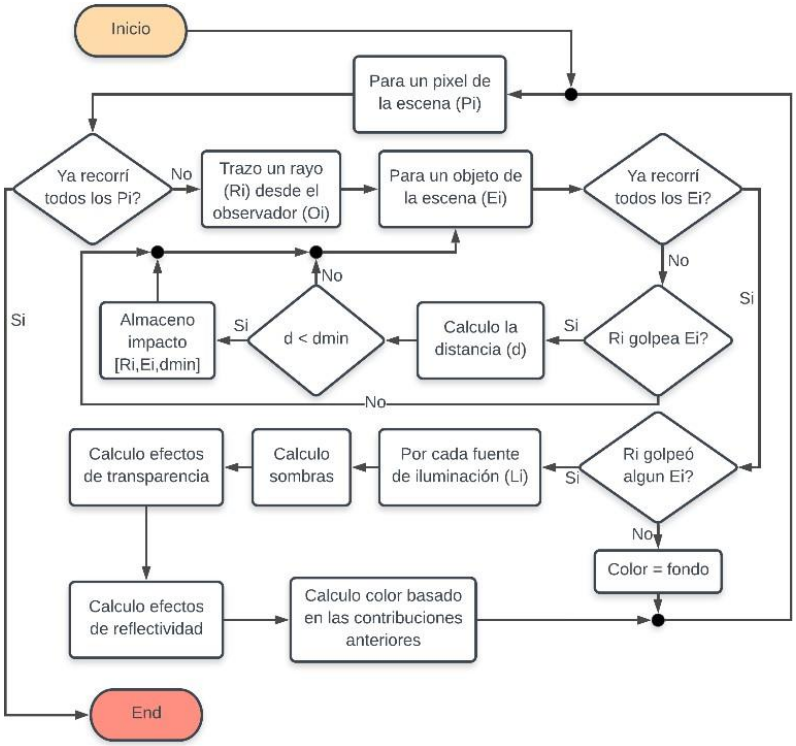

Figura 3. *Flowchart* del algoritmo *ray tracing*

1.3 Matemática del *ray tracing*

Revisar Anexo 1.

ESTADO DEL ARTE

El primer concepto sobre el principio básico del trazado de rayos se remonta al siglo XVI. Este aporte se debe al matemático, artista y pintor alemán Albrecht Dürer a través de su famosa colección de cuatro libros bajo el título "*Underweysung der Messung mit dem Zirckel und Richtscheyt*". En sus dos primeros libros, trata problemas de geometría general en dos y tres dimensiones. Dürer además propone métodos para la construcción de proyecciones en perspectiva incluyendo luces y sombras. En su obra también concluye que las sombras en una escena son aquellos lugares ocultos de la fuente de luz que pueden ser vistos desde una perspectiva donde la fuente de iluminación es vista como un punto [6].

Posteriormente, el filósofo René Descartes, en un corto ensayo titulado *De l'arc-en-ciel*, propone la primera teoría para explicar la causa del fenómeno atmosférico conocido como arcoíris. En su obra, describe la formación de un arcoíris primario (basándose en sus conocimientos de fenómenos ópticos) como el rayo que incide en el ojo después de dos refracciones y una reflexión, y un segundo arcoíris como otro rayo que alcanza el ojo después de dos refracciones y dos reflexiones. Esta teoría determina que el segundo arcoíris sea menos visible que el primero[7].

Hacia finales del siglo XX, Arthur Appel (1968) por medio de su trabajo: *"Some techniques for shading machine rendering of solids"* propone el primer algoritmo de *ray casting*[2]. Por medio de métodos como el *Point by point shading*, *Quantitative Invisibility* y *Cutting Planes*, brinda una solución al problema básico acerca de cómo la luz afecta un objeto, dándole importancia a las sombras en la textura y profundidad del objeto [8]. Hacia 1978 el ingeniero eléctrico e informático Turner Whitted publica su trabajo: *"An Improved Ilumination Model for Shaded Display"*. En este trabajo, Whitted propone el primer modelo de iluminación global que le permitía modelar los fenómenos de refracción y reflexión de la luz, además de conocer la existencia de líneas de vista entre el observador y los objetos, o fuentes de iluminación dentro de la escena [9].

El trabajo de Whitted, a principios de los 80, sirvió como punto de partida para comenzar a experimentar, profundizar y desarrollar el algoritmo de *ray tracing* como técnica de síntesis de imágenes. En 1985, Robert L. Cook por medio de su trabajo *"Stochastic Sampling in Computer Graphics"*, mejoró el algoritmo básico de *ray tracing* para lograr simular otros efectos en las imágenes como el desenfoque de movimiento, la profundidad de campo o el submuestreo para eliminar efectos de dientes de sierra o del inglés *aliasing* en la imagen resultante [10]. Debido a su alto costo computacional, se usó principalmente en instituciones militares, productoras de cine, corporaciones y compañías que tenían un alto poder de cómputo. En esta misma década, se desarrollan cortometrajes animados como *"The Compleat Angler"* (1978), *"Tin Toy"* (1988 - Pixar), secuencias de películas como las de TRON (1982 - Disney) y *"Start Trek II: The Wrath of Khan"* (1982 – Lucas Films) y películas completas como *"Toy Story"* (1995 – Pixar/Disney) que fue animada totalmente con el programa especializado de *ray tracing* llamado RenderMan (continuación de REYES que fue utilizado por la industria durante los años 90) [1].

La implementación del algoritmo de *ray tracing* en *hardware* comienza a finales de los años 80 y principios de los años 90, debido a la presión ejercida por la industria cinematográfica y la ahora sumada industria de los videojuegos, que comienza a establecerse como un mercado de gran potencial. La necesidad de técnicas interactivas hace que en los años siguientes la técnica de *rendering* utilizada en ese momento

[2]Algoritmo básico del *ray tracing* que consiste en trazar rayo por píxel desde un observador

fuera lentamente desplazada por una técnica alternativa conocida como *rasterization*, la cual ofrece mejores tiempos de respuesta pese a no contar con los efectos de realismo que ofrece *ray tracing*, utilizada en aquella época en proyectos *offline* [1].

Durante los siguientes años, se trabajó en distintas formas de optimización para mejorar el desempeño de los algoritmos de síntesis de imágenes. Entre estas mejoras se resaltan las particiones espaciales que permitieron reducir el número de primitivas que deberían leerse de memoria. El trabajo realizado en 1991 por Dan Gordon y Shuhong Cheng titulado *"Front to Back Display of BSP Trees"* consistió en un método de partición binaria. Todas estas optimizaciones fueron diseñadas especialmente para la técnica de *rasterization* debido a que el algoritmo de *ray tracing* seguía presentado los mismos problemas de alto costo computacional [11]. No fue sino hasta 1992 cuando Nieh y Levoy implementan el algoritmo de *ray tracing* en arquitecturas *MIMD* (de sus siglas en inglés *Multiple Instruction, Multiple Data*) con múltiples procesadores y memoria compartida, que a pesar de no ser práctico en aplicaciones interactivas comienza a apuntar al mejoramiento del desempeño del algoritmo en arquitecturas de *hardware* [12].

En 1995, el algoritmo comenzó a ejecutarse en computadores paralelos virtuales. A partir de estos resultados, se concluye que la implementación del algoritmo era propicio para las arquitecturas de cómputo en paralelo. En 1996, Humphreys y Ananian publicaron: *"TigerSHARK: A Hardware Accelerated Ray-Tracing Engine"*, donde se implementa el algoritmo en una de las tarjetas de la empresa Texas Instruments [13]. Para finales de los 90, algunas empresas como *Advanced Rendering Technology* comenzaron a implementar el algoritmo en arquitecturas de *hardware* que permitieron acelerar las operaciones más críticas. A partir de ese momento, empiezan a surgir los chips AR250 y AR350 en tarjetas madre como *ISA*, *PCI* y *AGP*, que podrían considerarse como las primeras tarjetas *GPU* [14].

A principios del siglo XXI, se desarrollan más implementaciones del algoritmo de *ray tracing*. Esto se puede observar en trabajos como el de Carr, et al. donde implementan "*The Ray Engine*" en una *GPU* utilizando procesadores para el cálculo de la contribución de color mediante lo que llaman *pixel shader processors* que fueron un gran hito en ese momento [15]. Purcell, et al. publican *"Ray Tracing on Programmable Graphics Hardware"* donde comparan el desempeño del algoritmo entre una arquitectura *CPU* frente a *GPU* [16]. Posteriormente Purcell divulga su trabajo *"Photon Mapping on Programmable Graphics Hardware"* y *"Ray Tracing on a Stream Processor"* donde se comienzan a exponer conceptos de la técnica *photon mapping* para cálculo de iluminación global y *stream processing* [17], [18]. En 2010, NVIDIA presenta su motor de *ray tracing "Optix: A General Purpose Ray Tracing Engine"* con mejoras significativas en la resolución de las imágenes [19].

Algunos investigadores comenzaron a incursionar en la implementación del algoritmo en arquitecturas de *hardware* de alto rendimiento, tales como las *FPGA*, para acelerar el cómputo de iluminación global. Otros investigadores trabajaron en la reducción del tamaño y el uso de materiales para la fabricación de *chips* que permitieron una mayor síntesis de imagen por segundo en sistemas paralelos. Es así como comienzan a publicarse trabajos como "VIZZARD II" (2002 - Meissner) [20], "*Hardware Accelerated Ray Tracing"* (2002 - Srinivasan) [21], "*Realtime ray tracing of dynamic scenes on a FPGA chip"* (2004 - Slusallek) [22], "*Fixed Math Ray Tracing"* (2007 – Hanika) [23], "*Real-Time Ray Tracer for Visualizing Massive Models on a Cluster"* (2011 - Ize, Brownlee y Hansen) [24], entre otros.

Actualmente distintas empresas dedicadas al procesamiento y síntesis de imágenes como NVIDIA están desarrollando tarjetas de video (como GeForce RTX) que permiten trabajar en tiempo real basándose en el

algoritmo de *ray tracing*. También a nivel de investigación se está empezando a incursionar en la utilización del algoritmo en áreas como la medicina, el cine, los videojuegos, la educación, entre otras. Además de la búsqueda para acelerar los tiempos de procesamiento, se desarrollan estrategias para disminuir el costo computacional.

Un resumen del estado del arte, tanto del algoritmo de *ray tracing* como su implementación en diferentes plataformas *hardware*, se puede apreciar en la línea de tiempo de la Figura 4.

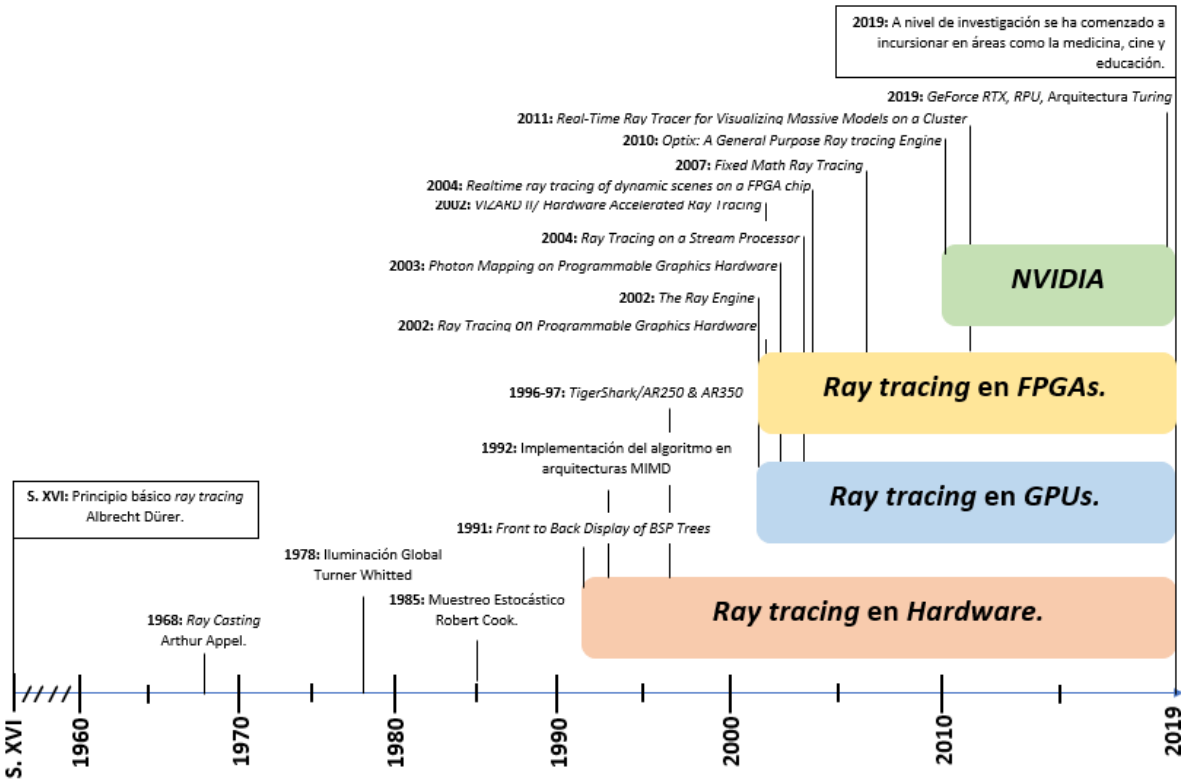

Figura 4. Línea de tiempo *ray tracing*

PROBLEMÁTICA Y OBJETIVOS

Una de las grandes problemáticas al momento de hacer el proceso de *rendering* en una escena, es el alto tiempo de procesamiento. Una razón de ello se debe a que, para cada píxel de la imagen, el algoritmo proyecta un rayo desde el observador y determina con cuál elemento de la escena interseca [25]. Si la escena cuenta con 1 millón de objetos y se desea sintetizar una imagen de calidad 1080p, la cual cuenta con 1920 x 1080 píxeles que equivale a 2.073.600 píxeles (2 Megapíxeles) [26], el proceso se vuelve computacionalmente costoso. En el año 2011, para la producción de *Cars 2*, Pixar triplicó el tamaño de su infraestructura, y hoy en día, la *render farm* cuenta con más de 12.500 *cores* en servidores *blade* de procesamiento de Dell [27]. Cabe resaltar que el costo promedio para el *rendering* en una *render farm* es de 0.1 dólares por hora del núcleo [28].

Por otro lado, existen cuellos de botella en el cómputo de las operaciones del algoritmo que generan altas latencias, como es el caso de la raíz cuadrada, cuya latencia, basado en las *IP* (*Intellectual Property*) *cores* de Altera Corporation, es de 16.28 ciclos de reloj para precisión simple en formato IEEE-754, y de 30.57 ciclos de reloj en precisión doble [6]. Estos retrasos afectan a grandes empresas como *Pixar Animation Studios* (Pixar), quienes indican que cada *frame* de sus películas, como *Cars 2*, toma aproximadamente 107 minutos, y tarda en promedio 11.5 horas en hacer el respectivo proceso de *rendering* [5].

Por sus características, *ray tracing* ha sido sistemáticamente relegado a producciones *offline* de imágenes fotorealísticas en áreas como arquitectura, diseño industrial, animaciones de películas y efectos especiales. A pesar de que este algoritmo produce imágenes de alta calidad su campo de aplicaciones se ha visto limitado [29]. Por lo tanto, se considera que el desarrollo de este proyecto permite analizar los factores críticos mencionados en el planteamiento de la problemática, con el propósito de generar un primer hito en una línea de investigación en la Pontificia Universidad Javeriana para dar continuidad al trabajo propuesto mediante la futura optimización de este proceso. Así, teniendo en cuenta lo anteriormente mencionado se plantearon los siguientes objetivos:

Objetivo general:
Diseñar e implementar un prototipo funcional que sintetice una escena mediante *FPGA*.

Objetivos específicos:

- Analizar cuellos de botella en procesamiento y cómputo del algoritmo *ray tracing*.
- Diseñar e implementar un sistema digital que a partir de la descripción de una escena compute la contribución de iluminación de un píxel de la imagen en *FPGA*.
- Diseñar un sistema de referencia para síntesis de una escena basado en *software*.
- Verificar para una misma escena la funcionalidad del sistema digital frente al sistema de referencia.

DESCRIPCIÓN GENERAL DE LA SOLUCIÓN

El diagrama en bloques general del trabajo de grado es el de la Figura 5.

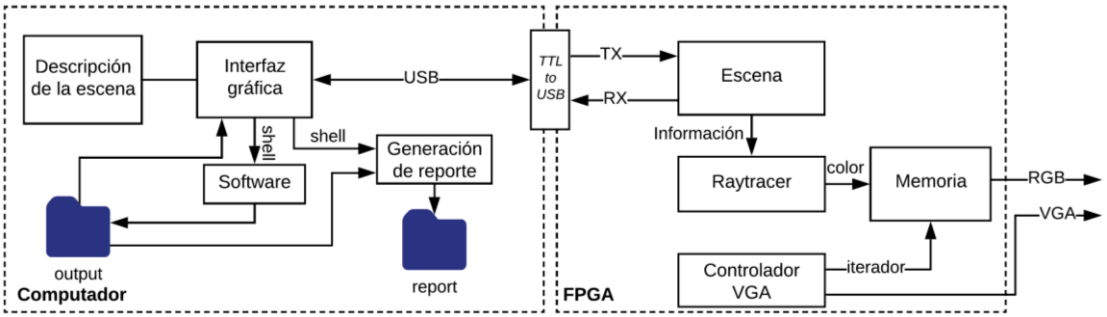

Figura 5. Diagrama en bloques general de la solución

En la solución planteada se tiene una interfaz implementada en el computador, mediante la cual se puede ejecutar el algoritmo ya sea en *software*, *hardware* o ambas. Además, se tiene un botón llamado *pick* con el cual se escoge cualquiera de las escenas previamente elaboradas y otro llamado *run* para ejecutar la opción previamente seleccionada. Al seleccionar la opción de *software* se corre el *RayTracer* desde la interfaz, al seleccionar *hardware* se envía, vía serial, la información de la escena a la *FPGA* para que esta realice todo el proceso de *rendering* y luego se recibe la información de cada píxel que envía la *FPGA*.

Si se selecciona ambas opciones, la interfaz ejecuta primero el *RayTracer* y después el proceso para la *FPGA*. Adicionalmente, la interfaz tiene la opción de generar de manera automática y abrir un reporte de *performance* para el código de *Software*, esto con el fin de realizar el análisis de cuellos de botella en el algoritmo. Además, este reporte incluye una serie de comparaciones entre las imágenes generadas (*Software* vs *Hardware)* que permiten verificar las similitudes entre las imágenes y ventajas de un sistema frente al otro. Los códigos fuentes se pueden encontrar en el Anexo 3 y Anexo 4.

1.4 Selección de componentes

Para la implementación de la solución, se utilizó una tarjeta Altera DE2-115 *Development and Education Board* de la empresa Terasic cuyo *chipset* principal es Altera Cyclone IV EP4CE115F29C7N. Se eligió esta tarjeta debido a las siguientes características:

- Periférico *VGA*
- 532 multiplicadores.

Aunque el laboratorio de electrónica de la Pontificia Universidad Javeriana tenía entre su inventario tarjetas con mayor capacidad de memoria y elementos lógicos, como la Stratix, esta tenía menor cantidad de multiplicadores, lo cual era una gran desventaja puesto que el algoritmo involucra varias operaciones matemáticas (ver Sección 1.16). Además, se usó un conversor TTL-*USB* para la transmisión bidireccional entre el computador y la *FPGA*.

1.5 Descripción de la escena

La descripción de la escena es un archivo de texto plano que contiene toda la información de la escena. Para su generación se usaron los *Standard procedural databases* (*SPD*) de Eric Haines, autor que sirvió como referencia para el desarrollo del trabajo de grado. Este *software* genera un archivo en formato *NFF* (*Neutral file format*) con las bases de datos de objetos que son bastante familiares y estándar para la comunidad gráfica [30] . El archivo generado contiene la información de las siguientes subsecciones:

1.5.1 Pantalla

- Dimensión de la pantalla.
  - Largo, número de píxeles a lo largo.
  - Ancho, número de píxeles a lo ancho.
- Color, define el color de imagen (*RGB*, escala de grises, blanco y negro).

1.5.2 Escena

- Dirección hacia arriba de la escena.
- Centro de la escena.
- Primitivas: Son los elementos de escena. Están compuestos por:
  - Posición, coordenada que determina la ubicación en el espacio.
  - Propiedades, propiedad física del material.
    - Índice de absorción, define el porcentaje del rayo incidente que absorbe la primitiva.
    - Índice de refracción, define el porcentaje del rayo incidente que se refracta al pasar la primitiva.
    - Índice de reflexión, define el porcentaje del rayo incidente que se refleja al golpear la primitiva.
  - Color, define el color de la primitiva.
- Fuentes de iluminación.
  - Posición, coordenada que determina la ubicación en el espacio.
  - Color, define el color de la fuente de iluminación.
- Color de fondo.

1.5.3 Observador

- Posición, coordenada que determina la ubicación en el espacio.
- Dirección, posición a la que el observador está mirando.

1.6 RayTracer

Se hará referencia, de aquí en adelante, como *RayTracer* al programa desarrollado para dar cumplimiento al tercer objetivo específico de la propuesta en este trabajo de grado. El programa *RayTracer* fue implementado completamente por los autores de este trabajo en C++ aplicando metodologías propias de ingeniería de *software* y específicamente los principios de diseño y programación orientada a objetos tales como herencia, sobrecarga y sobreescritura de métodos. El diagrama de clases que explica la relación entre las clases es el de la Figura 6.

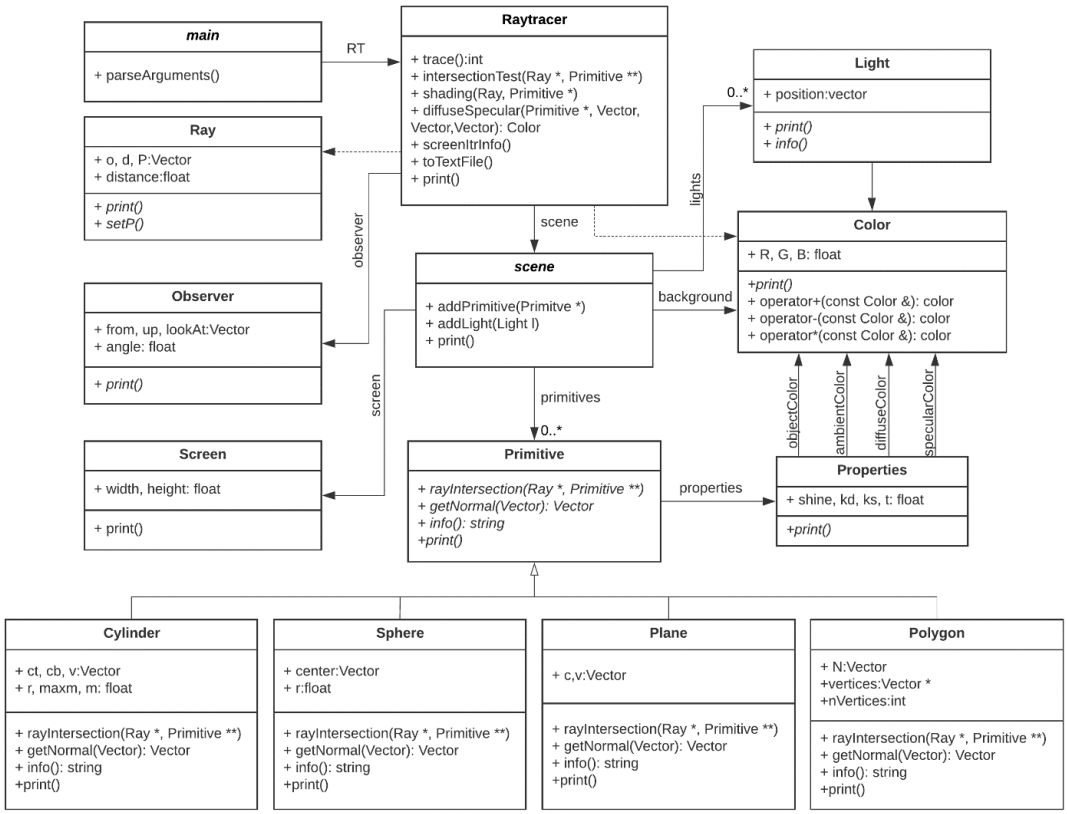

Figura 6. Diagrama de clases

El programa *RayTracer* permite leer descripciones de escena en formato *NFF*[3] y generar imágenes en escala de grises o a color, según la elección del usuario. Los parámetros para su ejecución son los siguientes:

./object nombreEscena opción

Opción	0/1	Generar imagen a color (por defecto)
	2	Generar imagen en escala de grises
	3	Generar imagen en escala de grises sin sombra
	4	Generar archivo para envío a *FPGA*

Tabla 1. Opciones de ejecución *RayTracer*

Los pseudocódigos de los métodos principales de la clase principal *RayTracer* son los siguientes:

```
Trace()
   for cada pixel x,y
      rayoPrimario = generarRayoPrimario(x,y)
      [primitiva, rayoPrimario] = testIntersección(rayoPrimario)
      color = shading(rayoPrimario, primitiva, profundidad)
```

[3] NFF es una descripción de archivo creado por Eric Haines

```
testIntersección (rayoPrimario)
   for cada primitiva de la escena
      distancia = encontrarDistancia(rayoPrimario,primitiva)
      if distancia < distanciaMinima
          rayoPrimario.distancia = distancia
          primitivaCercana = primitiva
   return [primitivaCercana, rayoPrimario]
```

```
Shading(rayoPrimario, primitiva, profundidad)
   primitivaSombra = primitiva()
   for cada fuente de iluminación de la escena
      vectorIluminacion = Vector(iluminacion.posicion,primitiva)
      rayoSombra = rayo()
      [primitivaSombra, rayoSombra] = testIntersección(rayoSombra)
      if rayoSombra.distancia ¡= infinito
          difuso = contribucionDifusa()
          especular = contribucionEspecular()
          color += ((difuso * primitiva.color) + especular)* intensidad
   if profundidad < 5
      rayoreflejado = rayo()
      [primitivaSombra, rayoreflejado] = testIntersección(rayoreflejado)
      colorReflejado = shading(rayoreflejado, primitivaSombra, profundidad+1)
      color += colorReflejado * primitiva.especular
   return color
```

1.7 Reporte de *performance*

Con el propósito de analizar los cuellos de botella del algoritmo se usó la herramienta de *profiling GPROF*. Según Thiel, "*GPROF* ... revolucionó el campo del análisis de rendimiento y rápidamente se convirtió en la herramienta elegida por los desarrolladores de todo el mundo ... la herramienta aún mantiene un gran seguimiento ... la herramienta aún se mantiene activamente y permanece relevante en el mundo moderno"[31]. *GPROF* permite conocer dónde el programa pasó el tiempo y qué funciones llamaron a qué otras funciones mientras se ejecutaban. Esta herramienta genera los siguientes reportes en un archivo con extensión "*txt*" [32]:

- El perfil plano muestra cuánto tiempo pasa el programa en cada función y cuántas veces se llamó a esa función. Si se desea saber qué funciones consumen la mayoría de los ciclos, aquí se indica de manera concisa.
- El gráfico de llamadas muestra, para cada función, qué funciones la llamaron, qué otras funciones llamó y la frecuencia con la que se realizaron los anteriores procesos. También hay una estimación de cuánto tiempo se pasó en las subrutinas de cada función. Esto puede sugerir lugares donde podría tratar de eliminar las llamadas a funciones que requieren mucho tiempo.
- La lista fuente anotada es una copia del código fuente del programa, etiquetada con el número de veces que se ejecutó cada línea del programa.

1.8 Interfaz gráfica (*GUI*)

Para poder cumplir de manera concreta y eficaz con el cuarto objetivo específico planteado, se procedió a realizar e implementar una interfaz gráfica que permitiera una comunicación directa con la *FPGA* para transmisión y recepción de datos. Para ello, se definieron las siguientes características que permitirían delimitar las funciones del programa:

- Permitir comunicarse con la *FPGA* de manera serial a través de un módulo de comunicación serial.
- Ejecutar tanto en *hardware* como en *software* los correspondientes programas encargados de la ejecución del algoritmo de *ray tracing*.
- Selección de escenas para implementación del algoritmo.
- Generación de documento *PDF* para comparación de desempeño entre *hardware* y *software* con información suministrada al término de la ejecución.
- Ser de fácil uso para el usuario, permitiéndole un fácil acceso y ejecución.

Para la implementación de la interfaz se eligió *Python* debido a su versatilidad, librerías y amplia documentación. Como funciones principales, la interfaz cuenta con cuatro botones fundamentales: *Pick, View Image, Open* y *Run*, y con tres *checkbuttons*: *Software, Hardware* y *Report*. Adicionalmente, permite al usuario visualizar la escena seleccionada y escoger el puerto para el envío serial de la información.

- ***Pick*:** Permite al usuario buscar y escoger la escena que se desea sintetizar por medio del algoritmo, esto equivale tanto en *software* como en *hardware*.
- ***View Image*:** Abre la imagen en formato.ppm generada al ejecutar el algoritmo en *software*.
- ***Open*:** Abre el reporte del *performance*.
- ***Run:*** Corre el algoritmo dependiendo la opción seleccionada previamente en los *checkbuttons*.
- ***Software:*** Opción que permite ejecutar el algoritmo mediante *software* a través del código en C++ diseñado previamente.
- ***Hardware:*** Opción que permite ejecutar el algoritmo mediante *FPGA*, para ello cuenta con una *droplist* que inmediatamente al seleccionar la opción aparece para escoger el puerto correspondiente para el envío de la información de manera serial.
- ***Report:*** Opción que permite generar el reporte del *performance*.

ARQUITECTURA EN *HARDWARE*

Para el desarrollo del segundo objetivo específico, se decidió realizar la implementación del algoritmo de *ray tracing* para primitivas cuya intersección se calcula resolviendo una ecuación cuadrática, en este caso se emplearon dos primitivas distintas: cilindros y esferas. Esta decisión, acordada con los directores, tenía como propósito hacer una propuesta de arquitectura *hardware* que demostrara la viabilidad futura de incorporar primitivas distintas, ya que otras propuestas académicas solo se enfocaban en una sola primitiva.

La solución se divide en tres partes principales: Generación de rayo primario, *test* de intersección y *shading*. Dentro de la generación del rayo primario se encuentra toda la adquisición de información sobre la escena y el cálculo del rayo primario desde el observador hacia cada uno de los pixeles. En el *test* de intersección se realizan los cálculos para determinar si el rayo interseca con alguna de las primitivas de la escena, y en el *shading* se calcula la contribución difusa de la fuente de iluminación.

Adicionalmente, se implementó la estrategia de *pipeline* que permite tener un alto *throughput* en la salida del diseño. Para ello fue necesario conocer la latencia de cada bloque del sistema (ver Tabla 8) puesto que ciertos resultados dentro de todo el desarrollo del algoritmo de *ray tracing* deben ser propagados a lo largo de bloques para no perder información. Por último, las operaciones matemáticas utilizadas (ver Tabla 7) fueron implementadas usando las *IP* de punto flotante (formato IEEE-754) precisión sencilla proveídas por la herramienta Quartus II.

1.9 Diagrama en bloques

La solución planteada está compuesta por seis bloques principales encargados de la implementación del algoritmo que son escena, iterador de pantalla, generación del rayo primario, *test* de intersección, mínima distancia y *shading*, y dos bloques adicionales para la visualización que son controlador VGA y memoria. El diagrama de bloques general se muestra en la Figura 7.

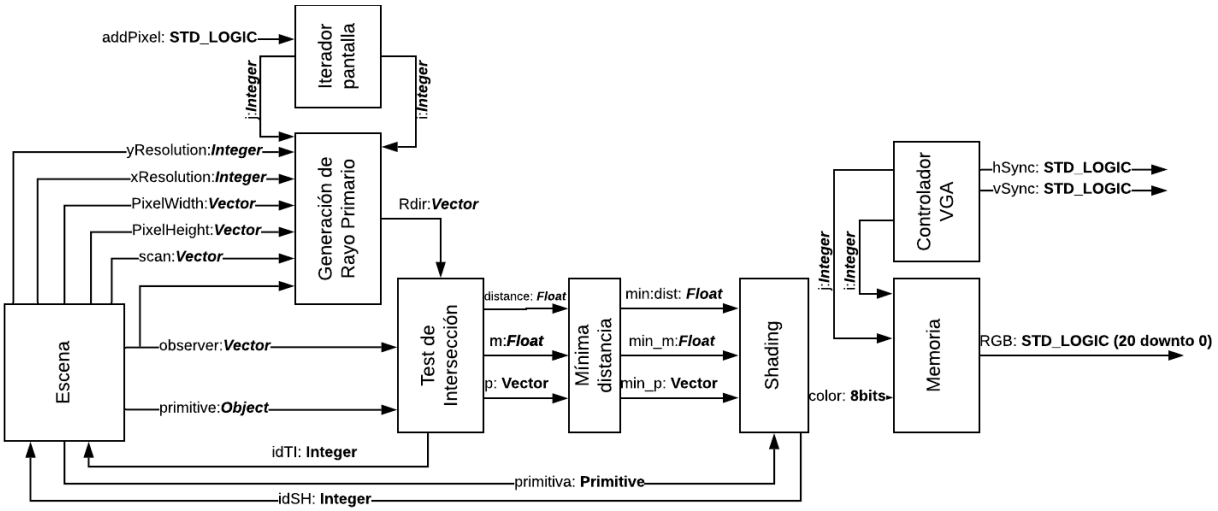

Figura 7. Diagrama en bloques

1.9.1 Escena

Contiene toda la información de la escena y la pantalla (ancho y largo de cada píxel, observador, fuente de iluminación, cantidad de primitivas e información de cada primitiva). Está compuesto por una memoria que contiene la información de cada primitiva y un bloque que se encarga de obtener la escena vía serial y guardar en memoria la información de la escena a la cual se le hará *rendering*. Por facilidad, se asumió que una primitiva es un conjunto de ocho *floats* que corresponden a las características de cada una de las primitivas (centro, radio, vector V y m máximo) y un bit que indica si la primitiva es un cilindro o una esfera (0 para esfera y 1 para cilindro). Este bloque además provee al *test* de intersección y *shading* la información necesaria para el cálculo de la distancia y del color de cada píxel respectivamente y al bloque de generación de rayo primario la información necesaria para generar el rayo hacia cada pixel de la pantalla. Su diagrama de bloques es el de la Figura 8.

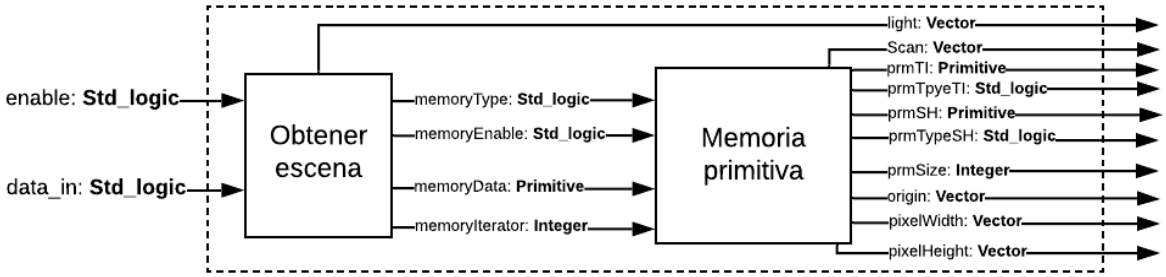

Figura 8. Diagrama en bloques Escena

1.9.2 Iterador pantalla

Es el encargado de generar las coordenadas de cada píxel al bloque de generación de rayo primario, esto lo hace a través de dos contadores, I para las filas y J para las columnas. Para ello, se recorren primero las columnas y cuando se llega al máximo de columnas la fila aumenta en uno su valor y se reinicia el contador de columnas. El bloque recibe dos señales de entrada, *RTenable,* la cual le indica que inició el proceso de *ray tracing* y *addPixel*, la cual le indica cuando cambiar la coordenada de píxel. Cuando ambas señales se encuentran en uno el bloque cambia de píxel. El diagrama en bloques se aprecia en la Figura 9.

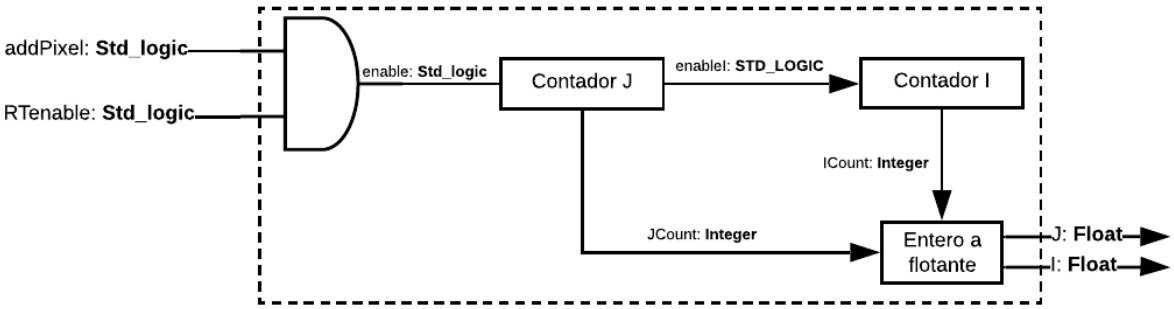

Figura 9. Diagrama de bloques de iterador de pantalla

1.9.3 Generación del rayo primario

Genera un rayo desde el observador hacia cada uno de los pixeles de la pantalla. Sus entradas son *ScanLine* (coordenada desde la cual se inicia a recorrer la pantalla), *PixelWidht* (ancho de un píxel), *PixelHeight* (largo de un píxel), *observer* (coordenada del observador), *j* (iterador de columnas de la pantalla) e *i* (iterador de filas de la pantalla). Su diagrama de bloques se puede observar en la Figura 10.

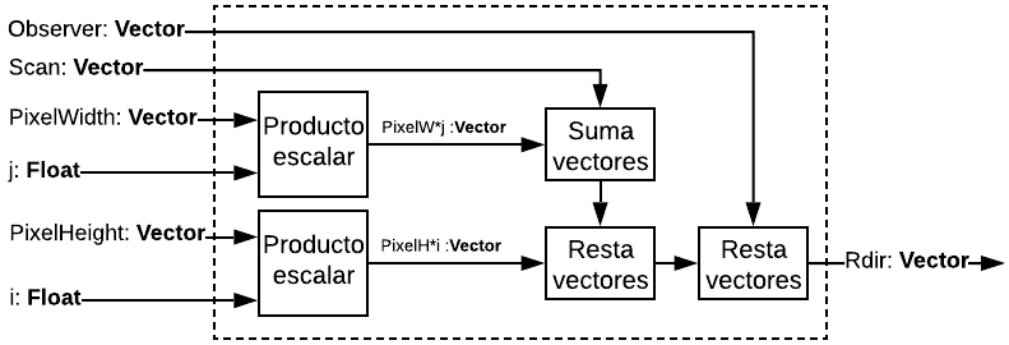

Figura 10. Diagrama de bloques del rayo primario

1.9.4 Test de intersección

Está compuesto de varios bloques que son: cálculo de las constantes de ecuación cuadrática a, b, c, para esfera y cilindro, selector, solución cuadrática y *Hit body*. Su función es determinar si el rayo primario interseca con alguna de las primitivas de la escena. Para ello, encuentra el vector entre el observador y el centro. Luego, calcula los parámetros (a, b y c) de la cuadrática y dependiendo del tipo de primitiva de entrada selecciona el resultado. Posteriormente, con estos parámetros se encuentra la solución a la cuadrática que permite identificar en caso de la esfera, si el rayo interseca o no con ella, o en caso del cilindro, si el rayo golpea algún punto a lo largo de su plano. Por último, se verifica si el punto de intersección se encuentra dentro del cuerpo del cilindro. El diagrama de bloques corresponde a la Figura 11.

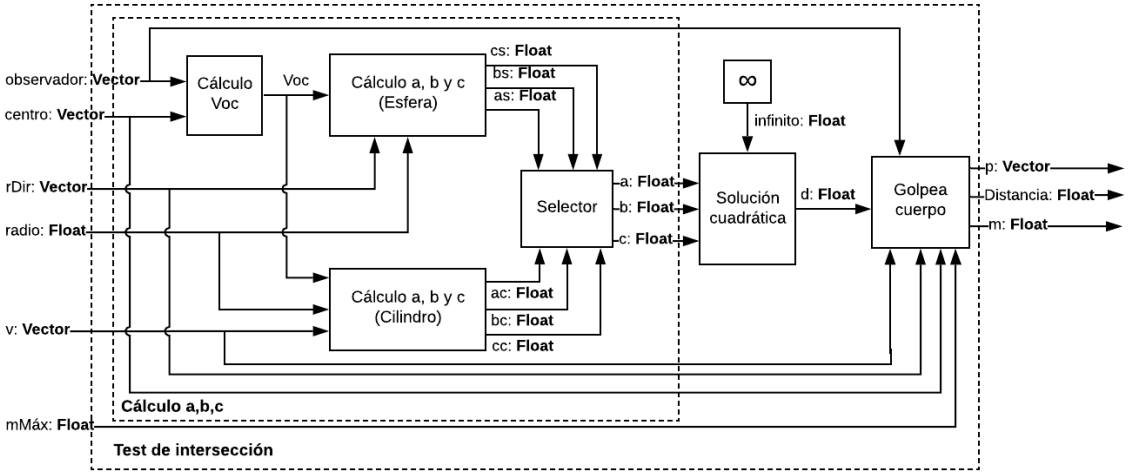

Figura 11. Diagrama en bloques del *Test* de intersección

Esfera (a, b, c) y Cilindro (a, b, c)

Estos bloques calculan las componentes a, b y c de la ecuación cuadrática para la esfera y el cilindro. Sus diagramas en bloques se observan en la Figura 12 y Figura 13.

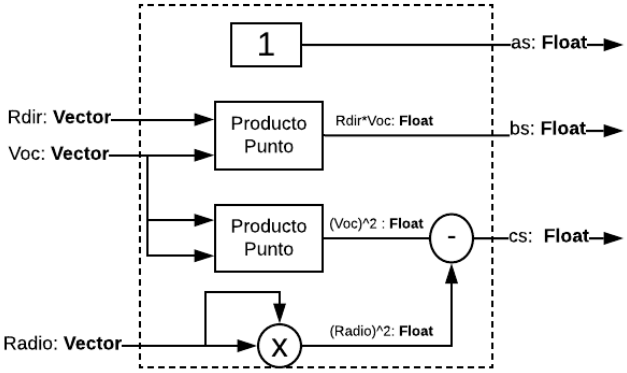

Figura 12. Diagrama de bloques Cálculo a, b, c (esfera)

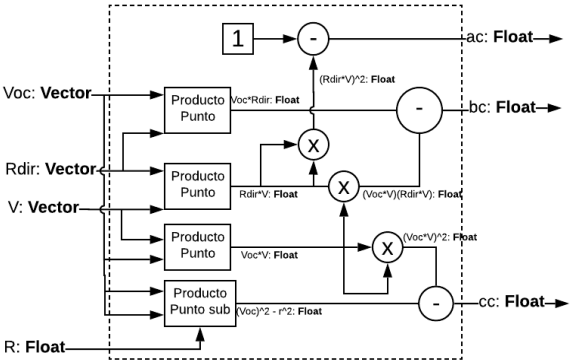

Figura 13. Diagrama de bloques Cálculo a, b, c (cilindro)

Selector

Selecciona los parámetros (a,b y c) previamente calculados que se utilizarán en el resto del *test* de intersección. La selección la hace basada en el valor de la señal *primitiveType*. Así, si *primitiveType* es 1, la salida toma los valores calculados en el bloque Cálculo a, b, c (cilindro), mientras que, si *primitiveType* es 0, la salida toma los valores calculados en el bloque Cálculo a, b, c (esfera).

Solución cuadrática

Resuelve una ecuación cuadrática, para ello, como primer paso encuentra las dos soluciones y posteriormente elige la menor. En caso de que las dos soluciones sean menores a cero (el rayo no interseca con la primitiva), el bloque asigna a su salida el valor infinito. Su diagrama de bloques es el de la Figura 14.

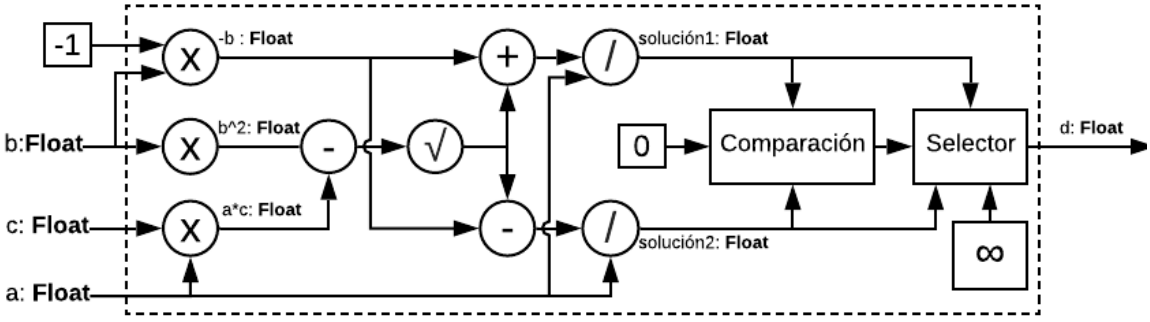

Figura 14. Diagrama de bloques Solución cuadrática

1.9.5 Mínima distancia

Guarda la información de la primitiva más cercana al punto de intersección. Para ello compara los resultados de distancia entregados por el bloque de solución cuadrática con los datos guardados de la primitiva con menor distancia y de ser menor la distancia actual, guarda en sus registros la nueva distancia, el punto de intersección y el *m* de la primitiva. El diagrama de bloques es el de la Figura 15.

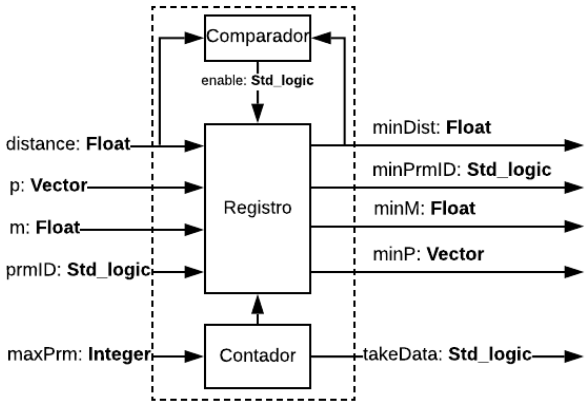

Figura 15. Diagrama de bloques de mínima distancia

1.9.6 Hit Body

Su función es acotar el cilindro, pues el *test* de intersección indica si el rayo interseca con un cilindro infinito. Para ello, calcula el *m*, que corresponde al punto más cercano al punto de impacto sobre el vector normal al plano de cilindro y verifica si está entre cero y *Mmáx. S*i se cumple la condición, el rayo interseca el cilindro y la salida *distance* toma el valor hallado en el bloque de solución cuadrática, de lo contrario la distancia toma infinito. Si la primitiva es una esfera la distancia de entrada se propaga a lo largo del bloque y es asignada a la salida *distance.* Su diagrama de bloques se puede observar en la Figura 16.

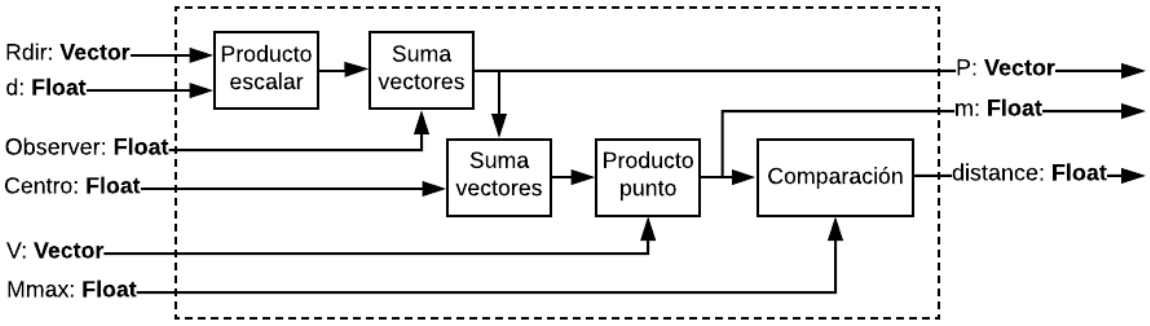

Figura 16. Diagrama de bloques de *Hit Body*

1.9.7 Shading

El bloque de *shading* es el encargado de calcular la componente difusa de la fuente de iluminación de la escena. En este bloque se calcula un vector con origen en el punto de intersección y destino la posición de la fuente de iluminación y otro normal a la superficie de la primitiva desde el punto de impacto. Posteriormente, ambos se normalizan y se calcula el producto punto entre ambos. El resultado de esta operación corresponderá a la componente difusa de la fuente de luz. Finalmente, se verifica que este resultado sea mayor a cero para después escalarlo por 255. Si el resultado es menor a cero se asigna como salida el color determinado para el fondo. Su diagrama de bloques se puede observar en la Figura 17.

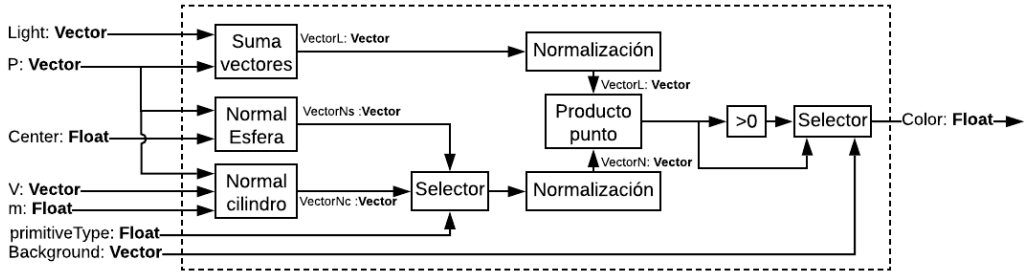

Figura 17. Diagrama de bloques de *Shading*

1.9.8 Controlador VGA

Se encarga de la generación de las señales de sincronización para la visualización de la imagen generada en una pantalla mediante VGA. Sus salidas *videoOn, h_sync, v_sync* van directamente al periférico de salida VGA y *row* y *column* corresponden a los iteradores de lectura del bloque Memoria. El diagrama de entradas y salidas es el de la

Figura 18. Diagrama de entradas y salidas Controlador VGA

1.9.9 Memoria

Este bloque es aquel donde se almacenan los valores de los píxeles, los cuales luego serán mostrados en la pantalla a través del VGA. Este consiste en una matriz de 640x480 posiciones que corresponden a cada uno de los píxeles de la imagen.

RESULTADOS

1.10 Métodos de Comparación de Imágenes

Para determinar la similitud de las imágenes tanto en el caso de aquellas generadas en *hardware* frente a las generadas a través de *software*, como también el caso de las imágenes generadas en *software* frente a ciertas imágenes realizadas por Eric Haines se realizaron tres métodos de comparación para poder abarcar toda la información posible y que el análisis de los resultados fueran los más claros y exactos para determinar las posibles diferencias que se puedan presentar.

- **Método de Histogramas usando *OpenCV*.** Como primer método se emplearon las funciones provistas por la librería de *OpenCV* para poder determinar los histogramas de las imágenes a evaluar; para ello después de obtener la imagen se procede a pasar la imagen a escala de grises para posteriormente hallar el histograma; el histograma utilizado hace referencia a la frecuencia con la que aparecen los distintos niveles de intensidad de una imagen a escala de grises, cuyo rango de intensidad va de 0 a 255, a partir de un histograma también se puede determinar información extra como el brillo, contraste y dinámica de una imagen.

- **Método de Histogramas en archivo de texto plano.** El segundo método es muy parecido al anterior ya que cumple el mismo principio de representar una imagen a través de un histograma en donde podemos evaluar la distribución de los distintos niveles de intensidad, pero a diferencia del primer método, en este no se lee directamente la imagen sino el archivo en texto plano que contiene toda la información de la imagen.

- ***RMS*.** Para este tercer método se utilizó la librería de *PIL* y se importaron las funciones de *ImageChops* la cual nos permite hallar la diferencia entre dos imágenes, así como también su respectivo histograma. De este método se obtiene el valor *RMS* de la diferencia de las imagénes. Para ello, se halla la diferencia píxel por píxel utilizando la función *ImageChops.difference()*, posteriormente se obtiene el histograma de la diferencia utilizando la función histograma y a este resultado se le calcula su correspondiente valor *RMS* (ver (1), entre más cercano sea el valor *RMS* de la diferencia de las imágenes a cero, más idénticas son las imágenes. Para tener una relación entre el resultado del *RMS* y la similitud de las imágenes, a través de pruebas se realizaron los rangos de valoración de la Tabla 2.

$$\textbf{RMS} = \sqrt{\frac{\sum p^2}{Tamaño\ img1 * Tamaño\ img2}}$$, donde p corresponde a los píxeles de la imagen. **(1)**

0-15	Imágenes visualmente idénticas
15-50	Imágenes visualmente similares
50-85	Imágenes con cambios notórios
85-253	Imágenes totalmente distintas

Tabla 2. Tabla con rangos de valores RMS

1.11 Software

Como cumplimiento del tercer objetivo específico de este trabajo de grado se diseñó e implementó un *RayTracer* en el lenguaje de programación C++, el cual permite sintetizar distintas imágenes, con distinta distribución de primitivas y fuentes de iluminación. Algunos de los resultados obtenidos se pueden visualizar en la Figura 19, Figura 20, Figura 21 y Figura 22 donde algunas de las primitivas utilizadas fueron: esferas, cilindros y polígonos distribuidos de distintas formas para representar diferentes objetos. Estas pruebas fueron ejecutadas en un computador con las características de la Tabla 3.

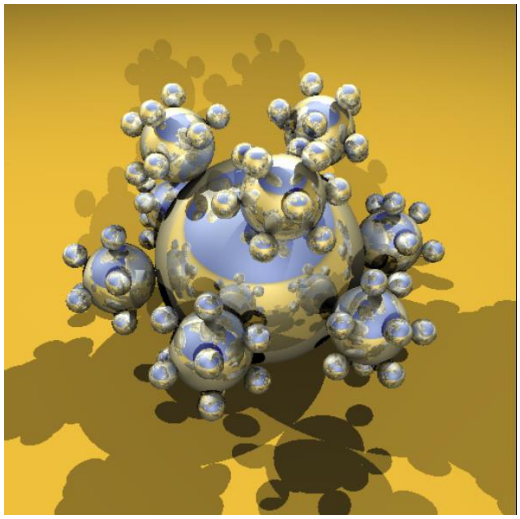

Figura 19. ***Sphereflake***

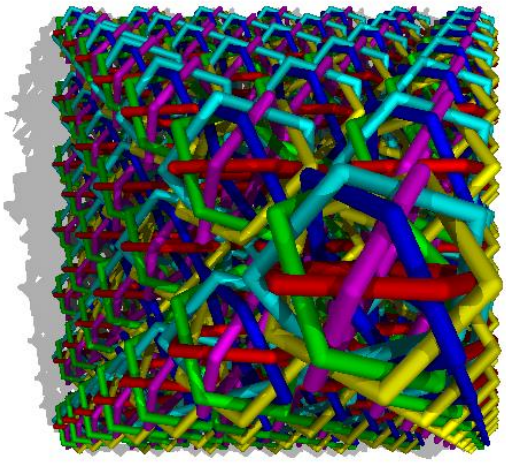

Figura 20. ***Rings***

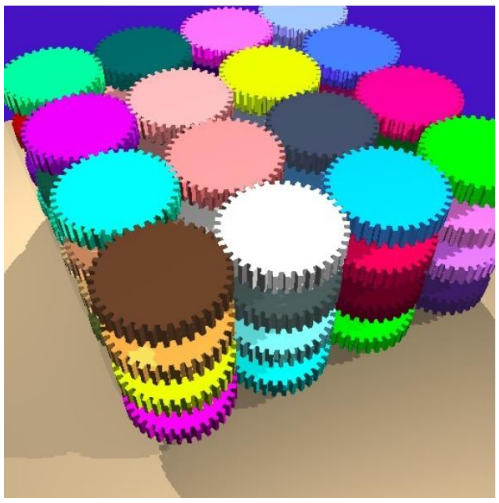

Figura 21. ***Gears***

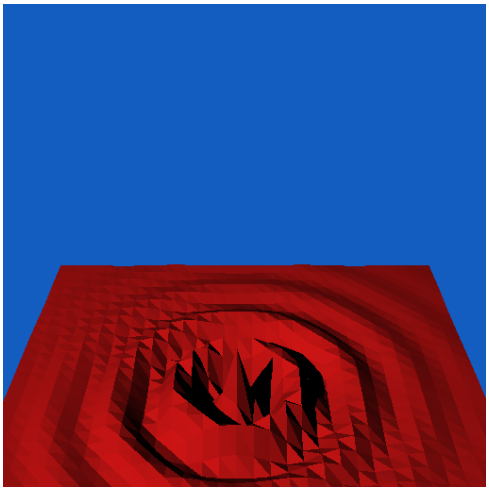

Figura 22. ***Sombrero***

Sistema operativo	Windows 10
Procesador	Intel64 Family 6 Model 60 Stepping 3
CPU	Intel(R) Core (TM) i5-4570 CPU@3.20GHz

Tabla 3. Información computador

1.11.1 Comparación de Imágenes en Software vs Imágenes de Eric Haines

Con el fin de determinar si las imágenes generadas en *software* eran correctas, se procedió a evaluarlas utilizando los métodos propuestos en la Sección 1.10 frente a algunos archivos *SPD* generados por Eric Haines. Los resultados obtenidos ante la escena de prueba *Tetra* se aprecian en la Figura 23 y Figura 24 , siendo esta primera una imagen generada por el *RayTracer* y la segunda por Eric Haines. Posteriormente estas imágenes fueron comparadas usando los métodos 1 y 3 descritos en la Sección 1.10.

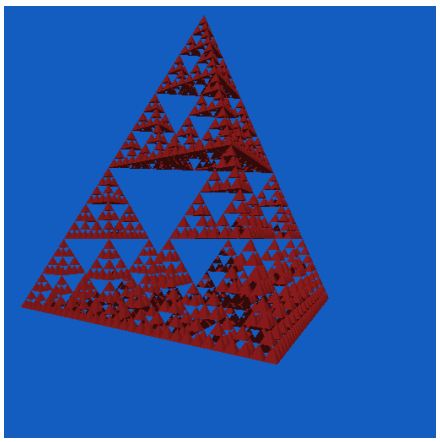

Figura 23. Imagen generada por el *software RayTracer*

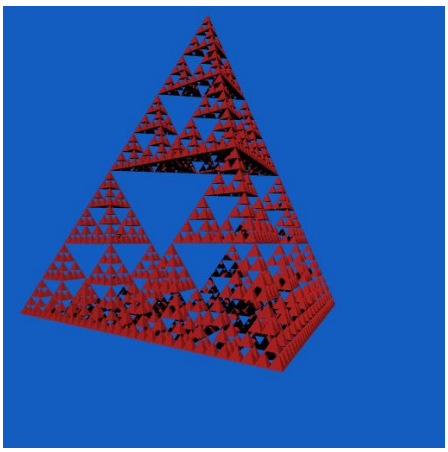

Figura 24. Imagen generada por Eric Haines

En la Figura 25 se presenta el histograma de las imágenes anteriores utilizando el método 1. Allí la frecuencia de los niveles de intensidad es muy similar, salvo unos pequeños picos en el nivel 50 por parte de la imagen de Haines; sin embargo, lo anterior indica que las figuras son casi idénticas.

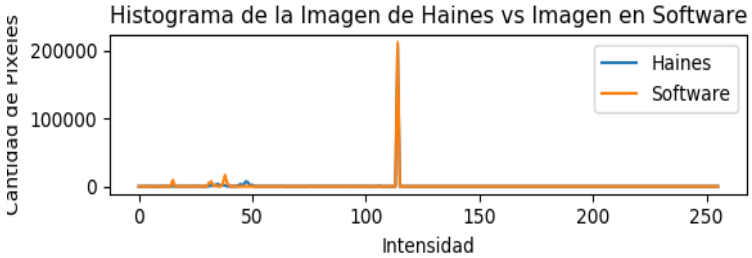

Figura 25. Resultado método 1 para tetra

En la Tabla 4 se observa el resultado del método 3. Para poder llegar hasta el resultado primero se obtiene la Figura 26 encontrando la diferencia píxel por píxel entre las imágenes evaluadas y posteriormente la Figura 27 que corresponde el histograma de la respectiva diferencia. De aquí podemos determinar, desconociendo el resultado del valor *RMS*, que existen pequeñas diferencias entre la imagen de Haines y la generada en *software* ya que la Figura 26 no se encuentra totalmente negra. La información provista en esta tabla nos muestra que la imagen en *software* es bastante similar con respecto a la imagen de Haines (Revisar Tabla 2) por lo cual se podría determinar que el *RayTracer* es bastante robusto para utilizarse como banco de datos para la comparación de las imágenes generadas en *FPGA*.

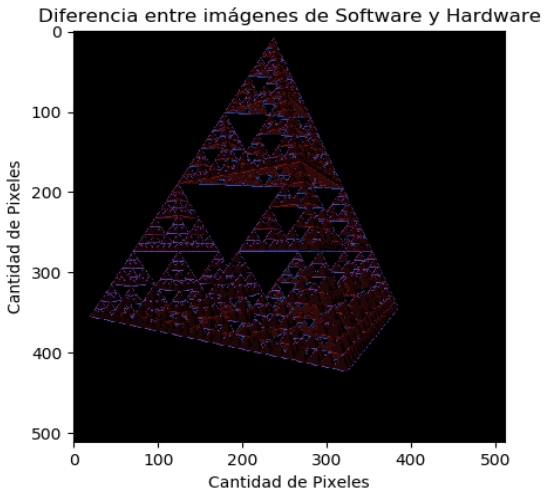

Figura 26. Imagen de la Diferencia Haines frente a *Software*

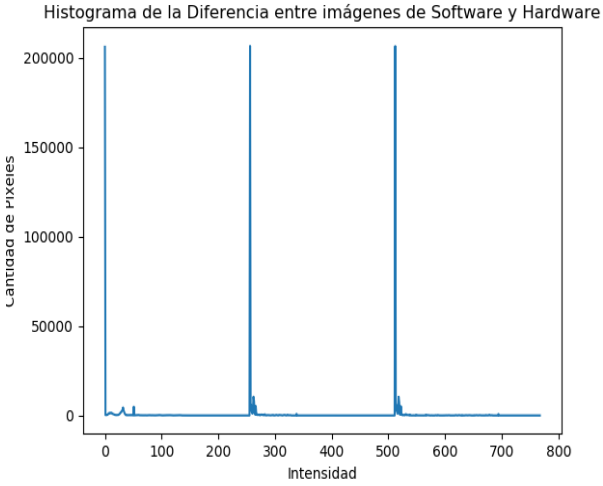

Figura 27. Histograma de la Diferencia Haines frente a *Software*

Valor RMS	22.17

Tabla 4. Resultado método 3 para *Gears*

1.12 Hardware

Como cumplimiento del objetivo general de este trabajo de grado se diseñó e implementó un sistema digital para *ray tracing*, el cual permite sintetizar distintas imágenes, con distinta distribución de primitivas y fuentes de iluminación. Algunos de los resultados obtenidos se visualizan en la Figura 28 y Figura 29. Las imágenes únicamente tienen componente difuso y fueron generadas en escala de grises.

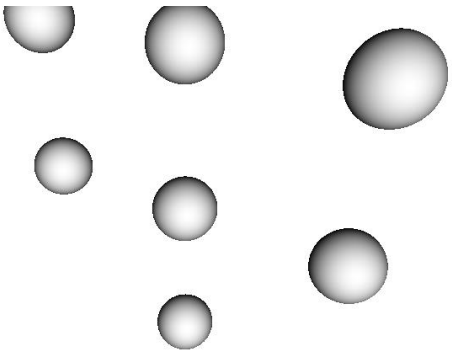

Figura 28. Imagen generada (7 esferas)

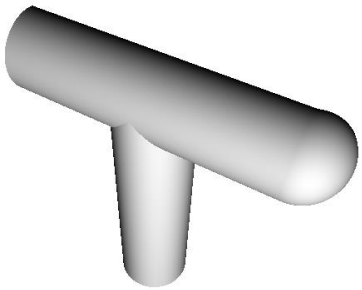

Figura 29. Imagen generada (1 esfera y dos cilindros)

1.12.1 Recursos utlizados.

Algunos de los recursos utilizados en la implementación del algoritmo en la *FPGA* son:

- 2.412 *Kbits embedded memory*
- 2 *PLLs*
- 2 pines para comunicación serial.
- Conexión *VGA*
- 532 multiplicadores.
- 100.000 elementos lógicos

1.13 Comparaciones *Software* frente a Arquitectura de *Hardware*

Con el fin de comparar las imágenes generadas en la *FPGA* se procede a contrastar ante la misma escena el resultado de *Software* frente a *Hardware*. Para ello se generó un conjunto de escenas cuyas descripciones se presentan en el Anexo 2. Luego, se procedió a evaluarlas utilizando los métodos propuestos en la Sección 1.10. A continuación, se tiene la Figura 30 y la Figura 31 como resultado de la síntesis de las imágenes en *Software* y *Hardware* respectivamente, ante una misma escena de prueba.

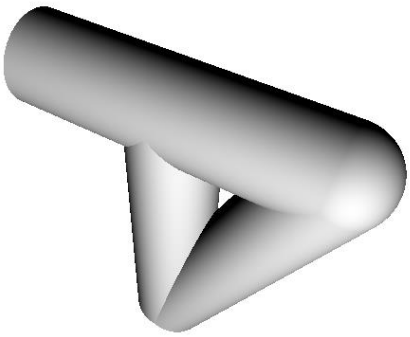

Figura 30. Imagen sintetizada por el software *RayTracer*

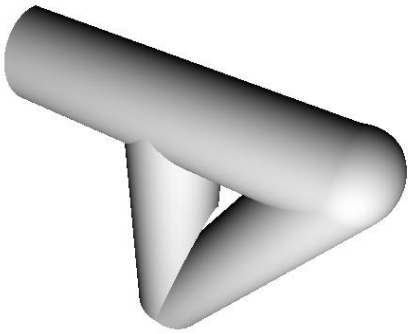

Figura 31. Imagen sintetizada en la FPGA

En la Figura 32 se aprecia el resultado del método 1. De esta figura se puede concluir que la distribución de los pixeles por los niveles de intensidad es casi igual. Posteriormente se evaluó el método 2 (Figura 33), lo cual refuerza la afirmación anterior. En esta última figura se ve con mayor nivel de detalle que las imágenes no son iguales, puesto que no todos los pixeles tienen la misma contribución de color.

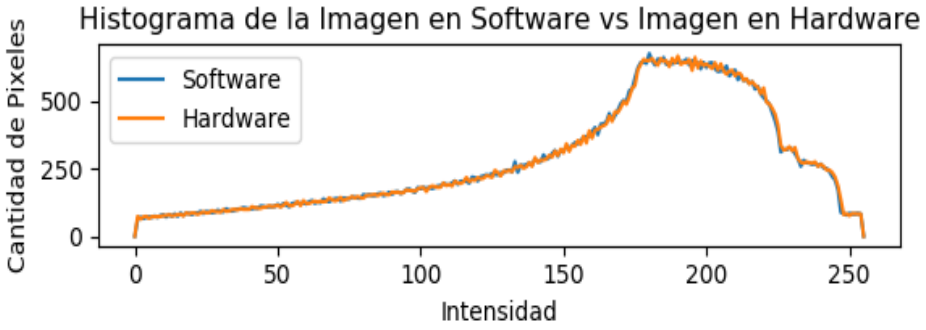

Figura 32. Resultado método 1 para *both 5*

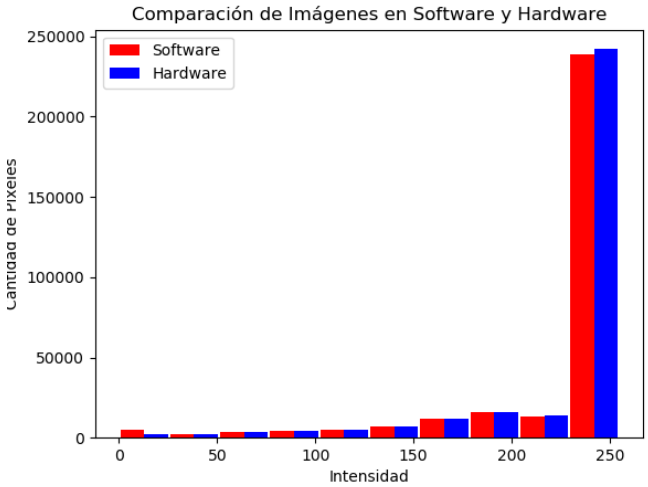

Figura 33. Resultado método 2 para *both 5*

Para el cálculo del método 3, se procede a calcular la diferencia píxel por píxel entre las imágenes, dando como resultado la Figura 34. En esta figura se observa como la diferencia es mínima ya que la imagen casi en su totalidad es negra (exceptuando ciertas áreas). Posteriormente se encuentra el histograma (Figura 35). De allí se refuerza las afirmaciones anteriores, ya que esta figura indica que la distribución de píxeles en su mayoría se ubica en intensidades negras y blancas. Finalmente, el valor *RMS* (ver Tabla 5) se encuentra dentro del rango donde las imágenes son visualmente similares.

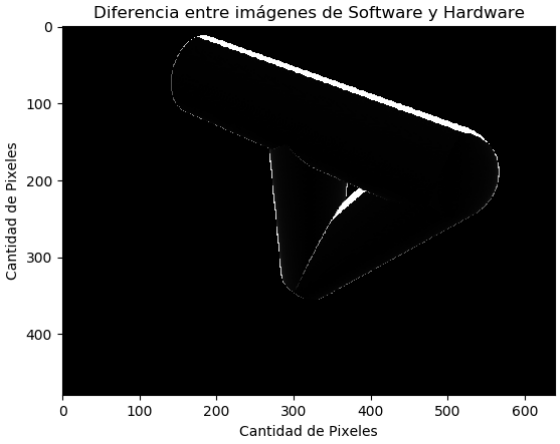

Figura 34. Diferencia entre las imágenes de la escena *both 5*

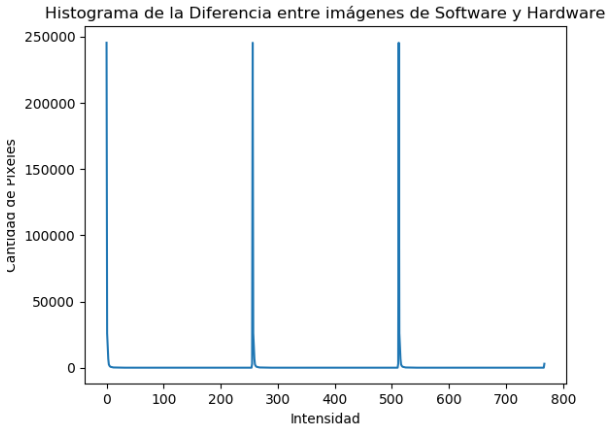

Figura 35. Histograma de las diferencias entre las imágenes

Valor RMS	26.6

Tabla 5. Resultados método 3 para *both 5*

Para finalizar se procedió a calcular el valor *RMS* de todas las escenas de prueba. El resultado se evidencia en la Figura 36. De allí se concluye que el valor *RMS* para todas las escenas de prueba se encuentran dentro del rango que indica que las imágenes son visiblemente muy similares. Las imágenes renderizadas para cada prueba se encuentran en el Anexo 2.

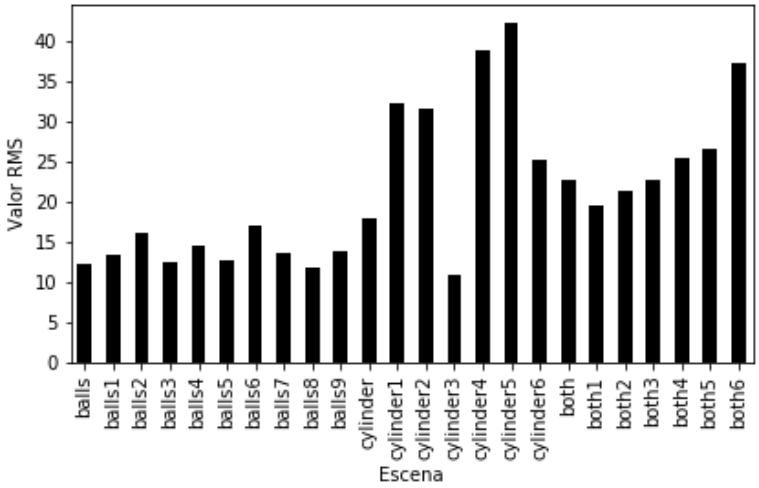

Figura 36. Método 3 para escenas de prueba

1.14 Análisis de rendimiento.

Usando la misma descripción de las escenas descrita en el **Anexo 2**, se procedió a analizar el rendimiento del *RayTracer* frente a *hardware,* para ello se midió el tiempo de *rendering* para todas las escenas y, posteriormente se realizaron diagramas de cajas para cada uno de los casos con el fin de observar la variación de los tiempos con respecto al número de primitivas. Los resultados se evidencian en la Figura 37 y Figura 38.

Para la implementación en *Software* se observa que los datos tienen una amplia dispersión en cada uno de los casos, lo que muestra que el tiempo de *rendering* para escenas con una misma cantidad de primitivas puede variar bastante. Esto da a entender que existen más variables que determinan el tiempo de *rendering* de la imagen, tales como: el número de primitivas, tipo de primitiva, posición de los elementos y radio.

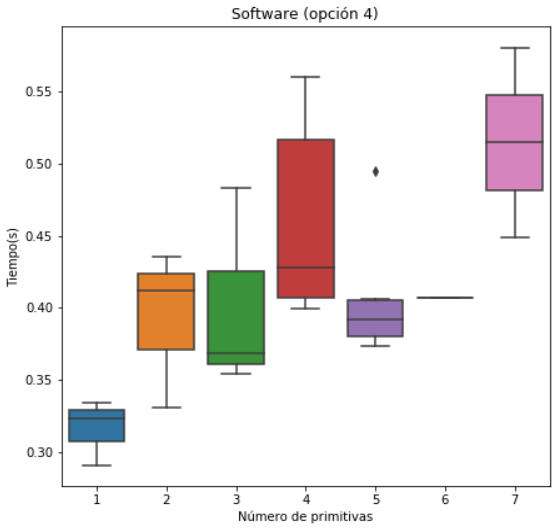

Figura 37. Tiempos de *rendering* en *Software*

A diferencia del *RayTracer*, al implementar el algoritmo en la *FPGA*, el tiempo de *rendering* se mantiene constante para cada número de primitivas. Esto se observa en la Figura 38, en la cual se encuentra el diagrama de cajas del tiempo con solo líneas, lo que significa que los valores de tiempo para cada número de primitivas no presentan variación. A partir de esto, se concluye que el único factor que determina el tiempo de *rendering* de la imagen con *FPGA* es la cantidad de primitivas que tenga la escena. La ecuación que describe el comportamiento de la Figura 38 es la siguiente, donde n corresponde al número de primitivas:

$$t = 0.018n, \text{ para n} = 1, 2, 3\ldots \quad \textbf{(2)}$$

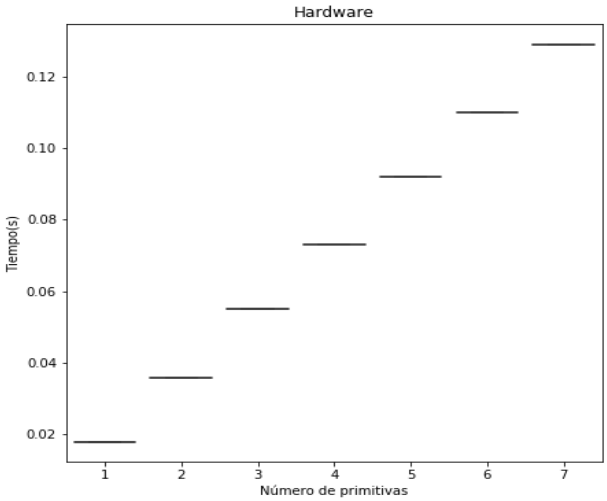

Figura 38. Tiempos de *rendering* en *Hardware*

Posteriormente, se graficaron los tiempos de *rendering* de *software* y *hardware* para tener una comparación visual de la diferencia entre los dos, esta se muestra en la Figura 39. De allí se concluye que el tiempo de *rendering* en *hardware* es menor para todos los casos.

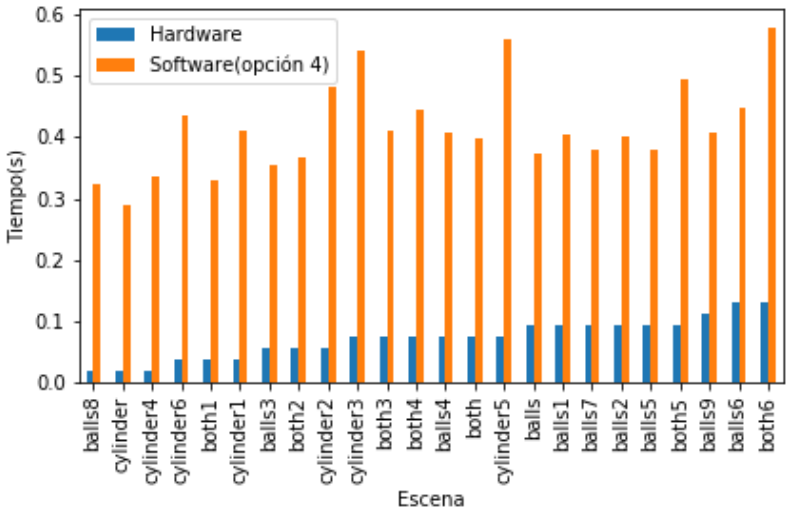

Figura 39. Comparaciones tiempos de *rendering*

También se calculó la proporción *hardware/software* de los tiempos de *rendering* y se graficaron (ver Figura 40). Para todos los casos de prueba se logra ver que el rendimiento de *hardware* es mucho mayor al del *Software.* De los resultados se puede resaltar que la mayor ganancia obtenida fue en la escena *cylinder4* (1 cilíndro) con un valor de 18 y la de menor ganancia fue en la escena *balls6* (7 esferas) con un valor de 4.6.

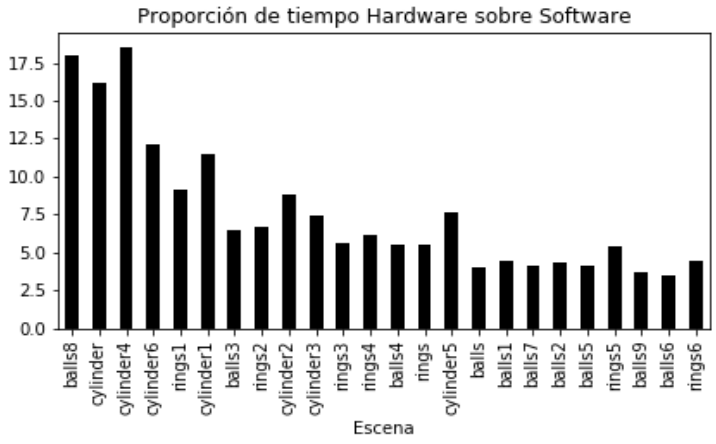

Figura 40. Proporciones tiempos de *rendering* (*Hardware/Software)*

ANÁLISIS CUELLOS DE BOTELLA

1.15 Reporte de *software*

Para poder analizar los cuellos de botella del algoritmo en *software* se usa la herramienta mencionada en la Sección 1.7 frente a una escena con alto nivel de complejidad y gran cantidad de distintas primitivas para poder obtener la mayor cantidad de información posible. La escena evaluada cuenta con aproximadamente 4200 cilindros, 4200 esferas y 2 polígonos, y fue sintetizada por el *RayTracer* en la opción 1 (imagen a color). La Tabla 6 muestra cuánto tiempo pasa el programa en cada función y cuántas veces se llamó a esa función.

Time	Accumulative seconds	Self seconds	calls	s/call	s/call	name
0.00	446.18	0.02	1.000000e+00	0.02	673.95	RayTracer
12.00	369.81	53.56	1.005600e+06	0.00	0.00	Intersection test
0.00	446.30	0.00	3.072000e+05	0.00	0.00	Shading
0.00	446.29	0.01	2.328000e+05	0.00	0.00	Color Contribution
0.00	446.20	0.01	4.789101e+06	0.00	0.00	Vector +
7.55	403.51	33.70	4.156549e+09	0.00	0.00	Vector -
0.00	446.24	0.01	1.312802e+06	0.00	0.00	Vector cross
21.91	97.78	97.78	3.902026e+09	0.00	0.00	Vector dotPoint
0.00	446.21	0.01	3.182902e+06	0.00	0.00	Vector normalize
0.01	445.94	0.06	3.363227e+07	0.00	0.00	Vector scale
0.04	445.80	0.18	4.761192e+07	0.00	0.00	SQRT
0.00	446.30	0.00	4.103440e+05	0.00	0.00	Pow
0.00	446.30	0.00	1.000000e+00	0.00	0.00	Primary ray info
15.15	259.63	67.60	4.223520e+09	0.00	0.00	Sphere Intersection test
21.12	192.03	94.25	4.223520e+09	0.00	0.00	Cylinder Intersection test
0.01	446.00	0.06	1.005600e+06	0.00	0.00	Polygon Intersection test
0.02	445.88	0.08	1.005600e+06	0.00	0.00	Plane Intersection test

Tabla 6. Tabla para análisis de *performance* del *RayTracer*

Una forma gráfica de ver la tabla anterior se aprecia en la Figura 41 y Figura 42. Para el caso de la Figura 41, se grafican cuantas veces fue llamada cada función, de esta grafica se concluye que la funciones más usada en todo el *RayTracer* son *Vector dotPoint*, *Vector* –, y las funciones donde se realizan los *test* de intersección, para este caso, por la escena seleccionada: *Sphere Intersection test* y *Cylinder Intersection test.* La diferencia de veces que fueron llamadas estas funciones con respecto a las otras es tan grande que visualmente parece que las demás funciones nunca fueron llamadas.

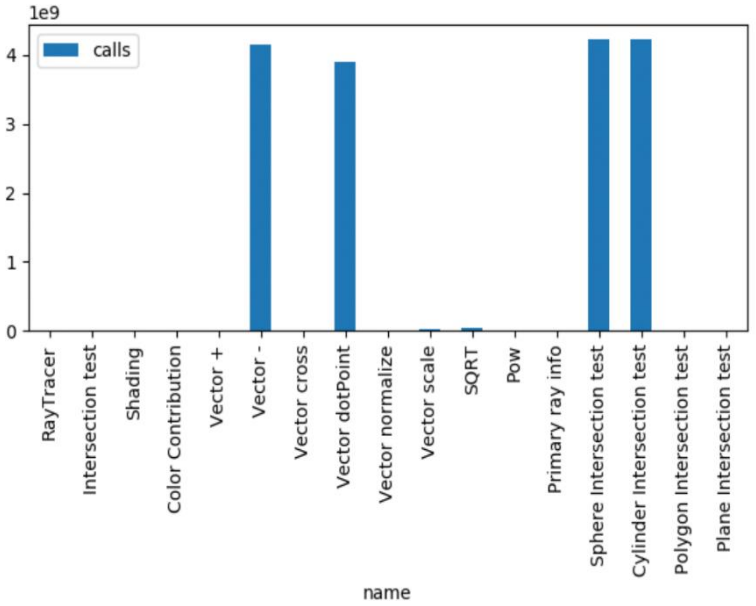

Figura 41. Gráfica de *calls* de la Tabla 6.

Para el caso de la Figura 42, se grafican cuantos segundos el programa estuvo dentro de cada función. De allí se concluye que el programa mantiene la mayoría de su tiempo realizando producto puntos, y posteriormente realizando los *test* de intersección.

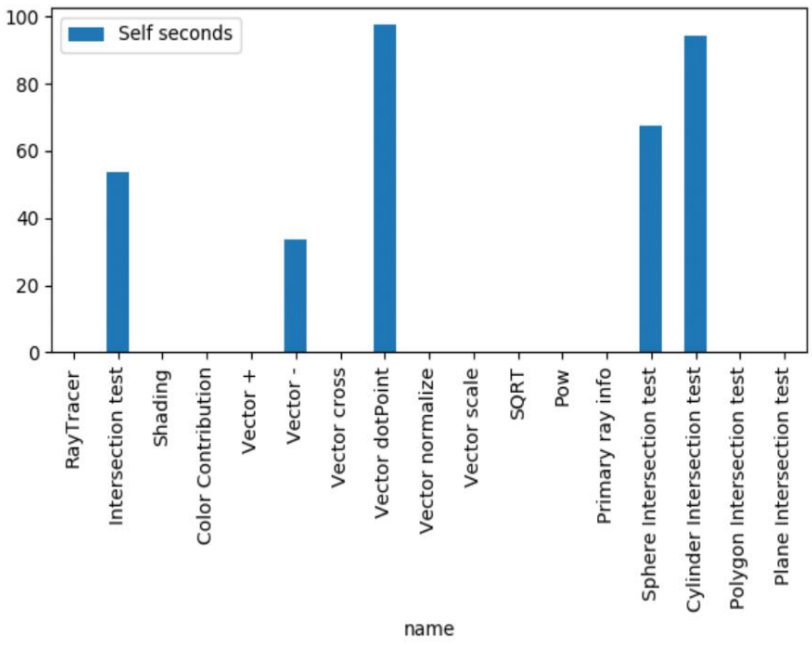

Figura 42. Gráfica de *Self seconds* de la Tabla 6

Para reforzar lo concluido en las dos anteriores gráficas se procedió a generar la Figura 43 la cual permite visualizar el árbol de funciones, con el número de veces que fue llamada cada función y el tiempo que el programa estuvo en estas.

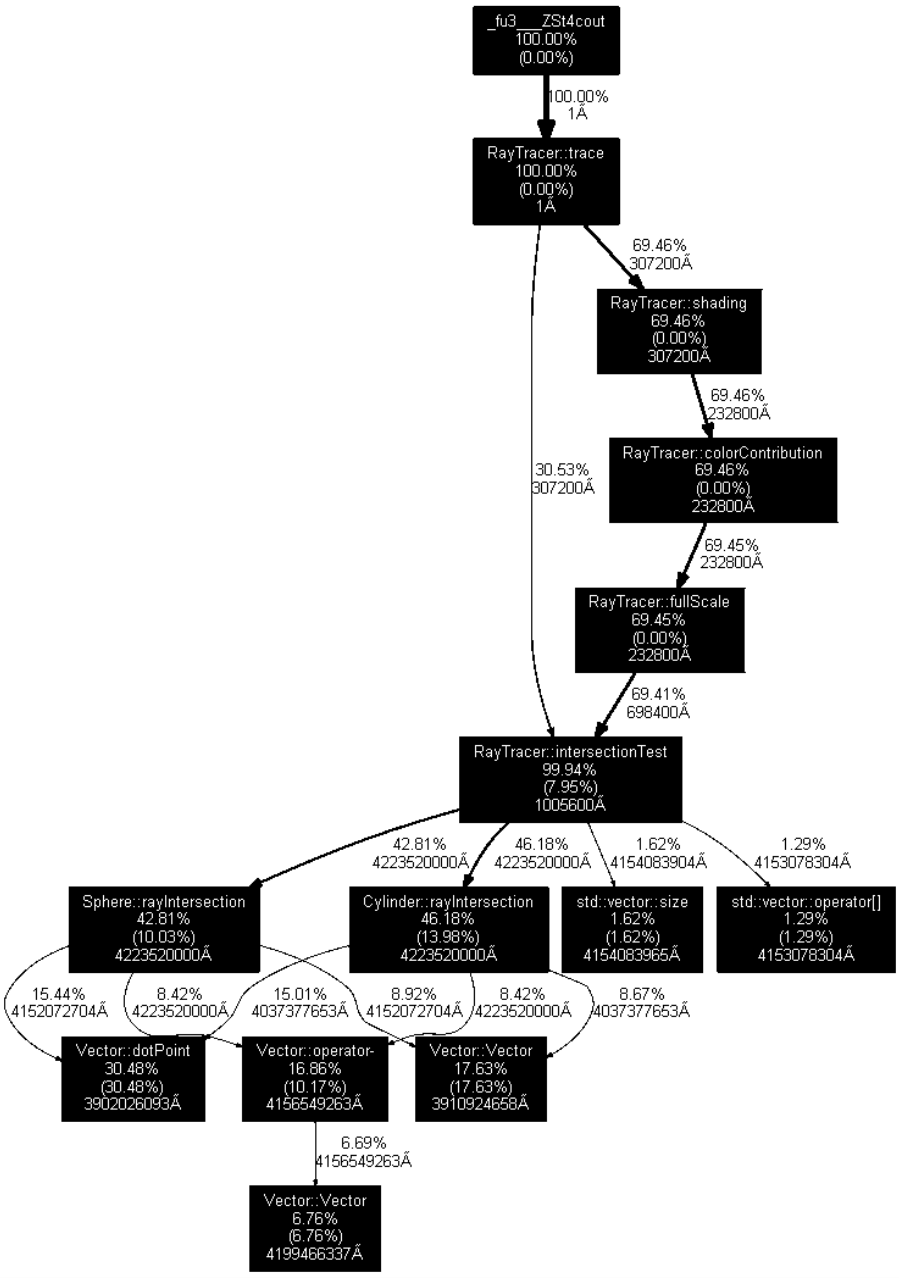

Figura 43. Árbol de funciones

De los resultados anteriores se concluye que el cuello de botella del algoritmo se encuentra en el *test* de intersección, siendo esta identificada como la función que más veces se llama a lo largo de la ejecución del código. Por ende, la operación a optimizar debe ser el producto punto, puesto que es la función más usada dentro de los *test* de intersección y además es la función donde más tiempo mantiene el código.

1.16 Operaciones y latencias

Para poder analizar los cuellos de botella del algoritmo primero es necesario conocer la latencia de las principales operaciones utilizadas. A continuación, se encuentra la Tabla 7 con información sobre las operaciones utilizadas con su respectiva latencia, elementos lógicos, funciones combinacionales, registros lógicos, *bits* de memoria y multiplicadores utilizados por cada una de ellas. Esta información fue obtenida mediante la herramienta Quartus II.

Operación	Latencia	Elementos lógicos	Funciones combinacionales	Registros lógicos	Bits de memoria	Multiplicadores
Producto punto	19	3947	3288	2144	0	28
Producto escalar	7	883	638	654	0	21
Normalización de vector	50	6812	4127	4813	0	76
Suma/resta vectorial	7	2719	2411	1246	0	0
Suma/resta (*IP*)	7	906	802	415	0	0
Multiplicación (*IP*)	5	301	222	218	0	7
Raíz cuadrada (*IP*)	16	1207	826	670	0	0
División (*IP*)	6	352	260	237	4608	16
Inversa raíz (*IP*)	26	1948	1225	1818	0	34

Tabla 7. Latencia y recursos utilizados por distintas operaciones

Además, se procedió a encontrar la misma información de la tabla anterior, pero en este caso para los bloques principales. Los resultados se encuentran en la Tabla 8.

Operación	Latencia	Elementos lógicos	Funciones combinacionales	Registros lógicos	Bits de memoria	Multiplicadores
Rayo primario	62	14008	11362	9889	0	118
Test de intersección	136	43173	32384	24302	21152	221
Distancia mínima	4	290	254	173	0	0
Shading	121	34512	24999	20826	8448	201

Tabla 8. Latencia y recursos utilizados por los bloques principales

Así, con la información de las tablas encontramos que el bloque con mayor latencia y mayor gasto de recursos es el *test* de intersección y le sigue la operación de normalización de vector. Con esto, se refuerza la conclusión de la Sección 1.15, ya que el *test* de intersección es uno de los mayores cuellos de botella del algoritmo pues resulta ser una operación demorada y que debe realizarse de manera iterativa a lo largo de todo el proceso.

1.17 Optimizaciones

Con base en los resultados y análisis de cuellos de botella se plantearon posibles optimizaciones para la mejora de desempeño.

1.17.1 Operaciones aritméticas

***Fast SQRT*:** La raíz cuadrada es una de las operaciones que más recursos consume y mayor latencia tiene debido a su complejidad matemática (ver Tabla 8). En la literatura sobre aritmética computacional, se reconoce la dificultad de esta operación junto con la división. Un aporte destacable de esta investigación fue desarrollado por el Ingeniero Juan Carlos Giraldo que le permitió encontrar un algoritmo de *Sqrt*, basado en un artículo de J. N. Mitchell [33], que tiene un error máximo de 3%. El código de este algoritmo es el siguiente:

```
float Sqrt(float radicand ){
   register union {float f; unsigned int i;} d = {.f = radicand };
   return d.i = (d.i + 0x3F800000U) >> 1, d.f;
} /* Sqrt */
```

Este código permite encontrar la raíz cuadrada de un número flotante de 32 *bits* realizando una suma de enteros y un posterior desplazamiento a la derecha. Por su simplicidad se puede implementar una solución tanto secuencial como exclusivamente combinatoria que permita generar el resultado en un ciclo de reloj. Se realizó la prueba de *rendering* para una esfera, y se llegó a la Figura 45. Con el fin de comparar, se agregó la Figura 44, la cual muestra una imagen sintetizada con la raíz cuadrada tradicional.

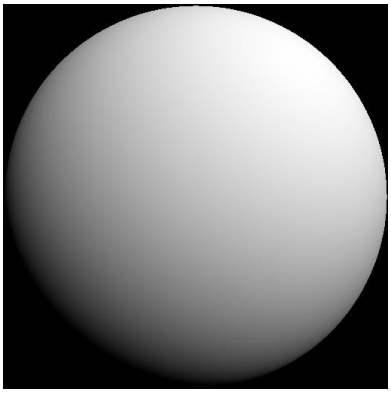

Figura 44. Imagen generada usando *SQRT* tradicional

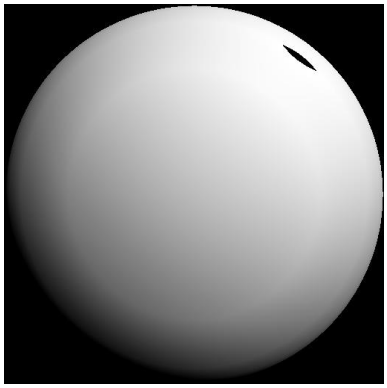

Figura 45. Imagen generada usando *Fast SQRT*

De los resultados se puede concluir que a excepción de la esquina superior derecha de la Figura 45 donde se aprecia una parte oscura, la distribución de color de las dos esferas es muy similar. Además, vale la pena recalcar que en aplicaciones visuales la precisión matemática no es importante mientras visualmente el cambio no sea apreciable, incluso muchos métodos de compresión de imagen aprovechan esta ventaja. Ese es el caso de JPEG, que utiliza una forma de compresión con pérdida basada en la transformada discreta del coseno (*DCT*)[34].

1.17.2 Generación de rayo primario

Escena acotada: Si todos los elementos de la escena están acotados por 1 tanto en el eje x, como en y (ver Figura 46), es posible recorrer todos los pixeles de la pantalla únicamente con dos contadores. Para esto se supone que el observador está sobre el eje z. Esto permite evitar toda la matemática necesaria para generar el *PixelWidth*, *PixelHeight* y *Scan.*

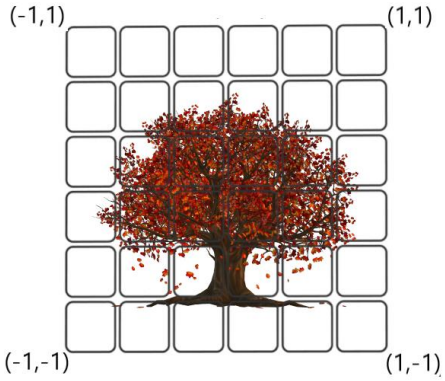

Figura 46. Escena acotada por 1

Observador en el origen: Complementando la optimización anterior, si el observador está siempre en el origen, se evita la necesidad de encontrar la diferencia entre el centro de la primitiva con el observador. Esto permite eliminar una resta entre vectores. El código que permite generar el rayo primario de forma optimizada (optimización Observador en el origen y Escena acotada) es el siguiente:

```
for( scan.e1 = SIDE/2.0 - 1.0; scan.e1 >= -SIDE/2.0; scan.e1 -= 1.0 )
   for( scan.e0 = -SIDE/2.0; scan.e0 < SIDE/2.0; scan.e0 += 1.0 ) {
      ray.d = unit( scan );
      distance = intersect( ray, &sph, light, &color );
      pixel = distance > 0.0 ? color : 0;
   }
```

Proyección ortográfica: La optimización consiste en mover el observador por cada uno de los pixeles de la escena, trazando en cada uno de ellos su rayo correspondiente, como consecuencia se pierde perspectiva debido a que el origen del observador no es fijo. De esta forma se suprime la tarea de normalizar la dirección del rayo cada vez que este sea generado, por lo que se logra evitar operaciones de alto costo computacional (raíz cuadrada, producto punto) que además tienen una alta latencia (ver Sección 1.16 Operaciones y latencias). En la Figura 47 se puede observar de manera ilustrada la optimización.

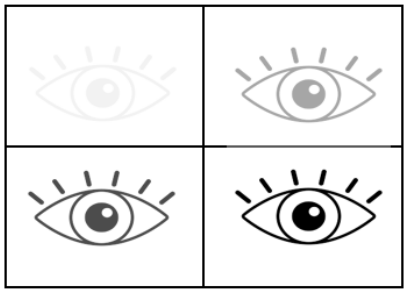

Figura 47. Proyección ortográfica

El código de esta optimización es el siguiente:

```
for( scan.e1 = SIDE/2.0 - 1.0; scan.e1 >= -SIDE/2.0; scan.e1 -= 1.0 )
   for( scan.e0 = -SIDE/2.0; scan.e0 < SIDE/2.0; scan.e0 += 1.0 ) {
      ray.o = scale( vec3( scan.e0, scan.e1, -scan.e2 ), 4.0/SIDE );
      distance = intersect( ray, &sph, light, &color );
      pixel = distance > 0.0 ? color : 0;
   }
```

El resultado se observa en la Figura 48. Como fue explicado anteriormente, a pesar de usar la misma función de *shading*, el ángulo con que incide el rayo respecto a la dirección de la luz tiene un efecto que cambia un poco el *shading* general, lo cual se puede observar al comparar la Figura 49 con la Figura 48.

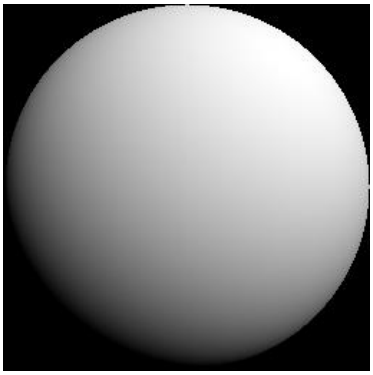

Figura 48. Imagen renderizada usando normalización del vector

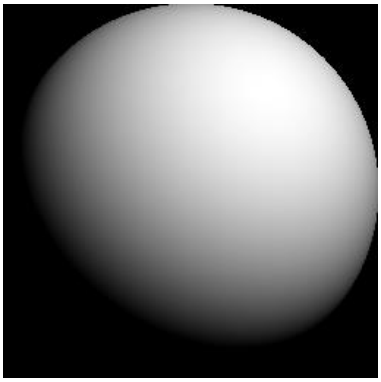

Figura 49. Imagen renderizada usando proyección ortográfica

1.17.3 Test de intersección

Diseño arquitectónico con múltiples unidades: Lo que se busca con esta optimización es acelerar el proceso de uno de los bloques con mayores cuellos de botellas y latencia en el algoritmo: *test* de intersección. Para ello, basados en un diseño propuesto por el ing. Juan Carlos Giraldo, se implementan varias unidades aritméticas que, en paralelo, se encarguen del cálculo de la discriminante, y que posteriormente pasen a un bloque donde se determina la distancia mínima entre los objetos impactados. Como beneficio de esta optimización también se reducen la cantidad de operaciones. La Figura 50 muestra una ilustración que describe la optimización descrita.

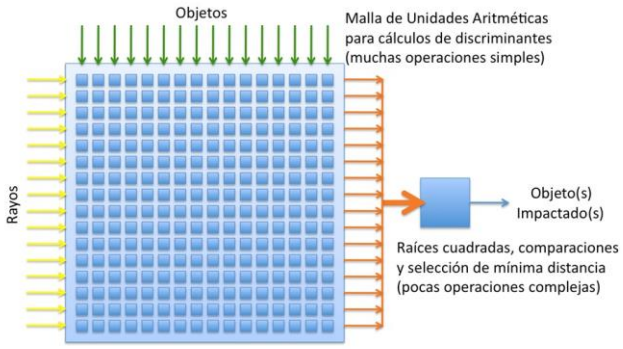

Figura 50. Ilustración de la optimización para el *test* de intersección.

1.17.4 Shading

Evitar recíproco de la raíz cuadrada. La variable "*shading*" corresponde a la iluminación sobre la superficie de la esfera. Esta toma el máximo valor de 1.0 para blanco y el mínimo de 0.0 para negro. Cuando la distancia entre el punto de máximo *shading* y el de intersección de rayo con esfera es superior a $(r\sqrt{2})$ el valor de shading corresponde a la semiesfera en negro de la Figura 51Figura 51. Ilustración de la optimización para el *shading*, de lo contrario será una fracción $(d_{im}/r\sqrt{2})$ entre el mínimo valor de *shading* y el máximo valor en el punto $\mathbb{M}$. El pseudocódigo de esta optimización es el siguiente;

$\mathbb{M} = \mathbb{C} - r\widehat{\mathbb{L}}$
$d_{im} = \|\mathbb{I} - \mathbb{M}\|$
`If (` $d_{im} < r\sqrt{2}$ `)`
`   shading = 0.0;`
`else`
`   shading = 1.0 -` $d_{im}/r\sqrt{2}$

Las notaciones usadas se explican en la Tabla 9.

Notación	Descripción
$\widehat{\mathbb{L}}$	Vector unitario con la dirección de una fuente de iluminación que se supone está lejos en el infinito y todos sus rayos llegan paralelos a la escena
$\mathbb{C}$	Vector con la posición del centro de la esfera
r	Escalar que indica la magnitud del radio de la esfera
$\mathbb{M}$	Vector con posición donde se da máxima iluminación sobre la esfera
$\mathbb{I}$	Punto de intersección del rayo del observador con la esfera
d_{im}	Escalar que indica la distancia euclidiana entre el punto de intersección del rayo del observador con esfera y punto de máxima iluminación

Tabla 9. Notaciones optimización para *shading*

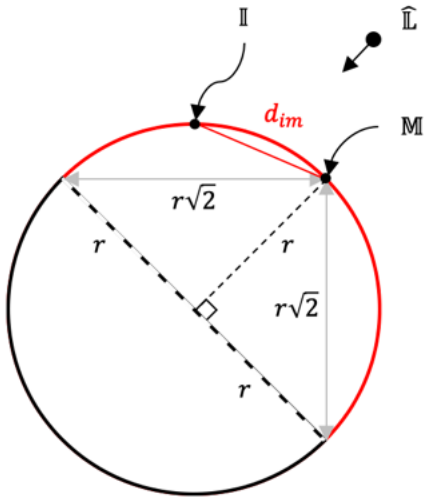

Figura 51. Ilustración de la optimización para el *shading*

Este modelo produce la misma contribución de la iluminación Lambertiana comúnmente explicada en libros de texto de computación gráfica sin tener que acudir a una operación de normalización que involucra el cálculo del recíproco de una raíz cuadrada.

1.17.5 Otras

Promedio de los pixeles: Para esta optimización se lanzan rayos desde el origen hacia la escena cada píxel de por medio, aquellos píxeles que no se calculan mediante la función del *ray tracing* se obtienen mediante el promedio de sus píxeles vecinos. Para la contribución de color, se aplica el mismo método de promedio de píxeles vecinos y se añade la diferencia entre la contribución de color del píxel superior frente a la del píxel inferior. El pseudocódigo de esta optimización sería el siguiente:

```
for cada píxel de la escena (i,j)
    if píxel es par
       color(i,j) = Raytracer()
    else
       color(i,j) = (colorPixelS – colorPixelI)/2
```

CONCLUSIONES Y TRABAJOS FUTUROS

1.18 Conclusiones

- En el presente trabajo de grado se diseñó e implementó un sistema digital que a partir de la descripción de una escena compute la contribución de iluminación de un píxel de la imagen en *FPGA*

- Se diseñó e implemento un *RayTracer* en *software* que sintetiza imágenes.

- Se tiene una interfaz para pruebas que verifica para una misma escena la funcionalidad del sistema digital frente al sistema de referencia y generé informes con medidas que permitan analizar cuellos de botella en procesamiento y cómputo del algoritmo *ray tracing*.

- Se logró cumplir a cabalidad con los objetivos propuestos.

- Se concluye que dentro del sistema digital para *raytracing* en *FPGA*, el bloque que más latencia presenta y más recursos consume es el *test* de intersección. Esto se debe en gran parte a la cantidad de operaciones que se deben realizar para poder calcular si el rayo interseca o no con una primitiva. Por otro lado, el de menor latencia y recursos consumidos fue el cálculo de la menor distancia ya que solo cuenta con una pequeña lógica que permite determinar si la distancia es la menor, junto con algunos registros que almacenan la información de la primitiva más cercana al punto de intersección.

- Los tiempos de *render* en *FPGA* solo dependen de la cantidad de primitivas presentes en la escena, a diferencia del *RayTracer*, cuyos tiempos dependen de distintos factores tales como la posición, tipo y cantidad de primitivas.

- Se diseñó un sistema funcional en *FPGA* con mayor eficiencia frente al de referencia (*software)*, puesto que en todas las escenas de prueba la implementación en *FPGA* presentó menores tiempos de *rendering* independientemente de las características de la escena. Vale la pena resaltar que el algoritmo en la *FPGA* que fue implementado no cuenta con ninguna optimización, sin embargo, se lograron ganancias de hasta 18 veces en el conjunto de pruebas realizadas.

- El producto punto al ser identificada como la operación más usada a lo largo del algoritmo debe ser optimizada al máximo.

- Aunque no fueron utilizados todos los recursos de la *FPGA*, se recomienda elegir una tarjeta de mayor capacidad y multiplicadores para aprovechar al máximo las optimizaciones propuestas.

1.19 Trabajos futuros

- Diseñar e implementar múltiples unidades de *rendering* en paralelo que se encarguen de ejecutar el algoritmo de *raytracing* exclusivamente en un píxel de la escena con el fin de disminuir los tiempos gastados en la generación de la imagen.

- Implementar las optimizaciones propuestas en este trabajo de grado.

- Observando el costo en tiempo y en cálculo generado por el *test* de intersección, y la constante utilización de la operación de producto punto, se considera importante buscar nuevos diseños y formas de implementación de estos para optimizar los tiempos. Es importante buscar optimizaciones que no impliquen el uso excesivo de recursos.

- Continuar con el *shading* para agregar efectos que permitan un mayor realismo en la imagen tales como: efectos de transmisión, reflexión, transmisión y *antialiasing*.

- Con el fin de generar imágenes más variadas, en cuanto a colores se refiere, se propone modificar la unidad de *shading* diseñada en este trabajo de grado para utilizar un espacio de color distinto al de escala de grises (ya sea RGB, HSV, CMYK, etc.).

- Ampliar el diseño de este trabajo a más primitivas como polígonos, triángulos y/o planos.

GLOSARIO DE TÉRMINOS

Elemento lógico:	La estructura básica de una *FPGA* se compone de los siguientes elementos: Tabla de búsqueda (*LUT*): este elemento realiza operaciones lógicas. *Flip-Flop* (*FF*): este elemento de registro almacena el resultado de la *LUT*. Cables: estos elementos conectan elementos entre sí.
Fuente de luz:	Una entidad geométrica con un conjunto asociado de características de emitancia.
Iluminación global:	Se considera iluminación global (IG) al conjunto de técnicas de iluminación que tienen en cuenta las interacciones entre los distintos objetos de la escena, es decir, considera la luz refleja- da por un punto teniendo en cuenta toda la luz que llega y no solo la energía que proviene de las fuentes de luz, como en el caso de la iluminación local.
Latencia:	En la arquitectura de *hardware*, la latencia de una unidad se define como el número promedio de ciclos que requiere desde el comienzo del cálculo hasta que genera el resultado.
Objeto:	Una entidad geométrica o procesal con un conjunto asociado de características de superficie que reflejan y posiblemente transmiten luz.
***Offline*:**	El proceso de *rendering* sin conexión se refiere a cualquier cosa en la que los cuadros se procesan en un formato de imagen, y las imágenes se muestran más tarde como imágenes fijas o como una secuencia de imágenes (por ejemplo, 24 cuadros constituyen 1 segundo de video preprocesado). Buenos ejemplos de sistemas de *rendering offline* son *Mental Ray, VRay, RenderMan.*
***Path Tracing*:**	El *path tracing* funciona generando rutas que comienzan desde el visor, el trazado de ruta bidireccional [58, 108] genera rutas desde el visor y las fuentes de luz y las conecta en el medio.
Primitiva:	Puede ser un objeto de una escena o una fuente de iluminación.
***Profiling*:**	En ingeniería de *software* el análisis de rendimiento, comúnmente llamado *profiling* o perfilaje, es la investigación del comportamiento de un programa de ordenador usando información reunida desde el análisis dinámico del mismo.
Proyección ortográfica:	La proyección ortográfica es un sistema de representación gráfica que consiste en representar elementos geométricos o volúmenes en un plano mediante proyección ortogonal. Se obtiene de modo similar a la sombra generada por un foco de luz procedente de una fuente muy lejana.
***Rasterization*:**	Proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital, como una pantalla de computadora, una impresora electrónica o una imagen de mapa de bits.

Rayo inicial:	Un rayo que se origina en el ojo y pasa a través de un píxel en el plano de la imagen.
Rayo reflejado:	Un rayo que se origina en la intersección de una superficie reflectante en la escena; su origen es el punto de intersección.
***Ray Tracing*:**	Para evitar confusiones, el término *ray tracing* o trazado de rayos se define aquí como un algoritmo para encontrar la intersección más cercana entre un rayo y un conjunto de primitivas geométricas, p.e. triángulos [Glassner89, Badouel92, Erickson97, Möller97, Shoemake98, Shirley02, Wald04] o parches de forma libre [Sweeney86, Parker99b, Nishita90, Martin00, Wang01, Benthin04]. De acuerdo con la ecuación de rayos $R(t) = O + t * D$, donde O es el origen del rayo y D la dirección del rayo, el algoritmo de trazado de rayos devuelve la intersección con la menor distancia $t_{min} \in [0, \infty)$.
Rayo de Luz:	Un rayo que se origina en la intersección de una superficie reflectante en la escena; su origen es el punto de intersección y su dirección es hacia una luz específica.
Rayo refractado:	Un rayo que se origina en la intersección de una superficie de transmisión en la escena; su origen es el punto de intersección.
Rayo Transmitidos:	En el caso de objetos en mayor o menor grado transparentes, y de forma análoga al tratamiento para los rayos reflejados, se generará un rayo transmitido. De igual forma, este nuevo rayo se comportará como un rayo primario en la siguiente iteración del algoritmo.
Rayo:	Un vector con un origen y dirección específicos.
Rayos de sombra:	Parten del punto de intersección con el objeto y tienen dirección hacia las fuentes de luz. De nuevo se realiza una prueba de intersección del rayo con todos los objetos de la escena para ver si hay algún objeto que corte su trayectoria, en cuyo caso el punto de origen del rayo estaría en sombra.
Rayos Primarios o Visuales:	Son los rayos que parten de la cámara virtual, pasando por cada uno de los píxeles en el plano de imagen. Para cada elemento de la escena se comprueba si el rayo visual interseca con alguno de ellos, quedándonos con el punto de intersección más cercano de toda la lista de objetos.
Rayos Reflejados:	Si el objeto donde intersecó el rayo tiene propiedades de reflexión de tipo espejo, se generará un nuevo rayo reflejado en ese punto. Este rayo se construirá típicamente en un procedimiento recursivo, pasando a comportarse como un rayo primario en la siguiente iteración del algoritmo.
Rayos secundarios	Todos los rayos que emitimos después de los rayos primarios (el rayo de la superficie del espejo al objeto difuso y el rayo del objeto difuso a la luz) se llaman rayos secundarios.

***Rendering*:**	En el contexto de los gráficos por computadora, el término *rendering* se refiere al proceso de generar una imagen bidimensional a partir de una escena virtual tridimensional. El *rendering* forma la base de muchos campos de los gráficos de computadora actuales, p.e. juegos de computadora, visualización y efectos gráficos utilizados para producciones de películas. Según el algoritmo utilizado para el proceso de *rendering*, se pueden clasificar dos categorías principales de *rendering*: *rendering* basado en rasterización y *rendering* basado en trazado de rayos.
***Shade processor*:**	Genera rayos iniciales y secundarios; también calcula la contribución que un rayo hace hacia el píxel final.
***Shading*:**	Calcular la iluminación y el color del píxel en función de la intersección con el objeto primitivo y la recopilación de las contribuciones de los segmentos de rayos secundarios.
***Standard Procedural Database*:**	Base de datos de procedimientos estándar es un paquete disponible de forma gratuita donde se armaron escenas y se utilizó ampliamente para comparaciones cuantitativas a finales de los 80 y principios de los 90.
***Stochastic Sampling*:**	El muestreo estocástico es una técnica de Monte Carlo [ll] en la que la imagen se muestrea en ubicaciones apropiadas con espacios no uniformes en lugar de en ubicaciones regularmente espaciadas. Este enfoque es inherentemente diferente del supermuestreo o el muestreo adaptativo, aunque se puede combinar con cualquiera de ellos.
***Throughput*:**	El *throughput* de una unidad de *hardware* se define como el número promedio de resultados producidos por ciclo de reloj. Un *throughput* de 4 significa que cada ciclo cuatro resultados (como cuatro cálculos de un solo rayo / transversal) abandonan la unidad de hardware, mientras que un throughput de 1/2 significa que solo se calcula un resultado cada segundo ciclo (como una intersección de un rayo / triángulo cada dos ciclos).

BIBLIOGRAFÍA

[1] J. A. G. Reyes, "Sistema de síntesis de imágenes de trazado de rayos en una plataforma de desarrollo para Hardware embebido," pp. 1–75, 2012.

[2] M. Christen, "Ray tracing on GPU," *Diploma Thesis*, 2005.

[3] Animation World Network, "NVIDIA Unveils Quadro RTX, World's First Ray-Tracing GPU." [Online]. Available: https://www.awn.com/news/nvidia-unveils-quadro-rtx-worlds-first-ray-tracing-gpu. [Accessed: 28-Mar-2019].

[4] NVIDIA, "Ray Tracing | NVIDIA Developer," *Ray Tracing*. [Online]. Available: https://developer.nvidia.com/discover/ray-tracing. [Accessed: 28-Nov-2019].

[5] I. Arroyo Gómez, L. Fernando, A. Pérez, and P. Escobar De La Oliva, "Implementación de la aplicación de raytracing en un entorno de procesadores y FPGAs," 2009.

[6] G. R. Hofmann, "Who Invented Ray Tracing?," *Vis. Comput.*, vol. 6, no. 3, pp. 120–124, May 1990.

[7] R. L. Lee and A. B. Fraser, *The rainbow bridge : rainbows in art, myth, and science*. Pennsylvania State University Press, 2001.

[8] A. Appel, "Some techniques for shading machine renderings of solids," 1968.

[9] J. D. Foley and T. Whitted, "Graphics and An Improved Illumination Model for Shaded Display."

[10] R. L. Cook Pixar, "Stochastic Sampling in Computer Graphics."

[11] D. Gordon and S. H. CHEN, "Front-to-back display of BSP trees," *Comput. Graph. Appl. IEEE*, vol. 11, pp. 79–85, 1991.

[12] J. Nieh and M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures."

[13] Greg, C. Ananian, A. Wolfe, and D. Clark, "TigerSHARK - A Hardware Accelerated Ray-tracing Engine," 2003.

[14] C. Cassagnabère, F. Rousselle, and C. Renaud, "Path tracing using the AR350 processor," 2004, pp. 23–29.

[15] N. Carr, J. Hall, and J. Hart, "The Ray Engine," *Graph. Hardw. 2002*, pp. 37–46, 2002.

[16] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 703–712, 2002.

[17] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon Mapping on Programmable Graphics Hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003, pp. 41–50.

[18] T. J. Purcell, “RAY TRACING ON A STREAM PROCESSOR,” 2004.

[19] A. R. F. G. Parker *et al.*, “OptiX™: A General Purpose Ray Tracing Engine,” *ACM Trans. Graph*, vol. 29, 2010.

[20] M. Meißner *et al.*, “VIZARD II: A reconfigurable interactive volume rendering system,” *Proc. SIGGRAPH/Eurographics Work. Graph. Hardw.*, pp. 137–146, 2002.

[21] J. R. Srinivasan, “Hardware Accelerated Ray Tracing,” 2002.

[22] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, “Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip,” 2004.

[23] U. Ulm, J. Hanika, A. Keller, and -Ing Frank Slomka, “Fixed Point Hardware Ray Tracing.”

[24] T. Kuhlen, R. Pajarola, and K. Zhou, “Real-Time Ray Tracer for Visualizing Massive Models on a Cluster,” 2011.

[25] I. Arroyo, L. F. Ayuso Perez, and P. Escobar de la oliva, “Implementación de la aplicación de raytracing en un entorno de procesadores y FPGAs,” pp. 1–71, 2010.

[26] AfterDawn: Glossary of technology, “1080p.” [Online]. Available: https://www.afterdawn.com/glossary/term.cfm/1080p. [Accessed: 28-Mar-2019].

[27] Daniel Terdiman, “New technology revs up Pixar’s ‘Cars 2,’” 2011. [Online]. Available: https://www.cnet.com/news/new-technology-revs-up-pixars-cars-2/. [Accessed: 11-Feb-2019].

[28] Fox Render Farm, “Render Farm Pricing,” 2018. [Online]. Available: https://www.foxrenderfarm.com/pricing.html.

[29] H. Ferreira, “Ray Tracing in Electronic Games,” 2008.

[30] E. Haines, “Standard Procedural Databases.” [Online]. Available: http://www.realtimerendering.com/resources/SPD/. [Accessed: 15-Oct-2019].

[31] Justin Thiel, “An Overview of Software Performance Analysis Tools and Techniques: From GProf to DTrace.” [Online]. Available: https://www.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors1/index.html#sec2.1.1. [Accessed: 28-Nov-2019].

[32] “GNU gprof.” [Online]. Available: https://sourceware.org/binutils/docs/gprof/. [Accessed: 31-Oct-2019].

[33] J. N. Mitchell, “Computer Multiplication and Division Using Binary Logarithms,” *IRE Trans. Electron. Comput.*, vol. EC-11, no. 4, pp. 512–517, 1962.

[34] A. M. Raid, W. M. Khedr, M. A. El-Dosuky, and W. Ahmed, “Jpeg Image Compression Using Discrete Cosine Transform-A Survey,” *Int. J. Comput. Sci. Eng. Surv.*, vol. 5, no. 2, 2014.

[35] M. Levo and P. Hanrahan, “Light Field Rendering,” 1996.

ANEXOS

Los anexos de este trabajo de grado se encuentran ubicados en el siguiente enlace: ***RayTracer*- Javeriana.**

Anexo 1. Matemática del *raytracing.*
Anexo 2. Banco de pruebas.
Anexo 3. *RayTracer software.*
Anexo 4. *RayTracer FPGA*.