Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de computación

Gestor de tareas para un *render farm* basado en GPUs.

Tesis que presenta

Lucía Araceli Oviedo Díaz

para obtener el Grado de

Maestra en Ciencias en Computación

Directores

Dr. Sergio Víctor Chapa Vergara.

Dr. Amilcar Meneses Viveros

México, Distrito Federal **Diciembre 2012**

Resumen

En la actualidad, la evolución de las imágenes se ha incrementado notablemente, se observan tanto en nuestra vida cotidiana como en el ámbito académico animaciones cada vez más realistas y de alta definición. Éstas animaciones están compuestas por una secuencia de imágenes llamados fotogramas ó *frames*. Pero para la generación de éstas imágenes se requiere de todo un proceso detrás: el diseño del modelo de la escena, determinación de materiales, iluminación de escena y un proceso de render encargado de la generación de las imágenes en 3D. Y a nivel mas bajo, se requiere de un algoritmo de render, ente los cuales se encuentra el trazado de rayos (mejor conocido en la literatura como *ray tracing*); capaz de generar imágenes de muy alta calidad; sin embargo, tiene un costo computacional relativamente alto. Es decir, para generar un segundo de animación se requieren de entre 24 a 30 fotogramas por segundo y para la producción de una película de animación los tiempos de render se vuelven inmanejables.

Para resolver éste problema se crearón los *render farm* (conjunto de computadoras con diferentes tipos de arquitectura que permite la generación masiva de imágenes mediante algún proceso de render). En la actualidad generalmente existen *render farms* para arquitecturas distribuidas y multiprocesador.

De reciente aparición se tienen a los GPUs, tarjetas diseñadas para trabajar con gráficos, cuentan con alto nivel de paralelismo y logran disminuir los tiempos del proceso sobre las operaciones involucradas en la generación de gráficos. El algoritmo para el trazado de rayos es uno de éstos y gracias a su inherente propiedad paralelizable se adapta bien para trabajar sobre GPUs.

Esta tesis se enfoca a realizar un *render farm* destinado a renderizar imágenes mediante la técnica de sintetizado de imágenes de trazado de rayos, mediante el uso del motor de propósito general OptiX (que corre sobre GPUs) y conjunto de tarjetas GPU's. Para ésto, es indispensable el uso de un buen despachador de trabajos cuyo objetivo principal es la administración equitativa de las tareas y los recursos que componen el *render farm*.

Abstract

Nowadays, the evolution of images has increased significantly, we can find realistic and high definition animations in academic research and in our daily lives. These animations are composed of a sequence of images called frames. But for the generation of all these images requires a whole process behind: the model design of the scene, determination of materials, lighting and a rendering process who makes possible the generation of 3D images. Specifically talking about the rendering process, is required a rendering algorithm; this thesis is focus on the ray tracing algorithm. Which is capable of producing very high quality images. However, this algorithm has a disadvantage that is a high computational cost. And, for generate a second of animation are required from 24 to 30 frames per second, then the total time for render images becomes intractable.

To solve this problem has been created the render farm systems, which is a set of computers that allows the generation of images using a massive rendering process. Most of the render farms that has been created are for multiprocessor and distributed architectures.

Recently with the introduction of GPUs (which are cards designed to work with graphics with high level of parallelism), has been reduced the process times of the operations involved in generating graphics has been. Thank to that, the ray tracing algorithm, who has an inherent property paralelizable, has been well adapted to work on GPUs. This thesis is focus in the design and development of render farm based on the ray tracing algorithm, using the general purpose engine OptiX (that work on GPUs) and a set of GPU's devices. But for this, is essential the help of a good dispatcher with the main objective of tasks management.

Agradecimientos

Para comenzar quiero darle las gracias al ser que me creo, que ha dotado mi vida de todos los elementos necesarios para para estar aquí, ahora. Gracias Dios por haberme dado la oportunidad de aprender tanto durante esta etapa de mi vida, gracias por darme amor, salud, felicidad... y ¿porque no? gracias por las dificultades, sin las cuales sería imposible crecer como persona y en realidad apreciar lo que se tiene.

Quiero agradecer a mi familia por todo el apoyo aportado; a mis padres Araceli y Guillermo por su apoyo incondicional, por sus consejos, por su amor...; gracias a mis hijas Alizeé y Natalie por darme la fuerza, felicidad, amor y claro por regresarme de vuelta a la tierra, gracias a mi esposo Iquer por apoyarme, valorarme, estar conmigo y por siempre buscar lo mejor para nosotros; Gracias a mis hermanos, abuelita, tíos, suegros por su apoyo y compañía.

Sin dejar de lado a mis amigos y compañeros, les quiero dar las gracias por sus consejos y porque aún y cuando las distancias son largas, en los momentos más difíciles siempre están ahí regalando consejos, sonrisas y abrazos, porque sin ellos habría sido más difícil culminar esta etapa. De manera especial quiero agradecer a Joel, Eduardo, Liliana, Tanibeth y Marie Ely.

Por supuesto que tengo que agradecer a todos mis maestros, por todos sus conocimientos compartidos, por dedicar su tiempo a la educación que tanta falta le hace a nuestro país, especialmente al Dr. Amilcar que siempre se toma un tiempo para charlar y de paso ayudar a los alumnos, y de manera personal quiero agradecerle el haberme apoyado y ayudado a retomar mis estudios.

Quiero agradecer al CINVESTAV por permitirme la oportunidad de estudiar en el departamento de computación y por acercarme al mundo de la investigación y al CONACYT porque gracias al apoyo económico brindado, pude llevar a fin este proyecto de vida.

Finalmente y no menos importante, un agradecimiento especial a Sofi que siempre esta dispuesta a escucharnos y aconsejarnos y al final regalarnos una linda sonrisa. También quiero darle las gracias a Felipa, Arcadio y en general a todo el personal del departamento, ya que gracias a ellos se mantiene funcionando y siempre dando lo mejor.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Actualmente se vive una época basada en el contenido digital; digitalmente, los documentos de guardan ya sea en forma texto, imágenes, bases de datos, etc. Específicamente, las imágenes las se pueden encontrar en distintos tipos de archivos e incluso generadas con diferentes algoritmos y es precisamente a éstas últimas a las cuales se enfocan en esta tesis.

La evolución de las imágenes ha sido tal, que actualmente se cuenta con imágenes muy realistas y de alta definición; se pueden encontrar en la industria de las películas, arquitectura, diseño, medicina, investigación simulación, entre muchas otras. Logrando generar imágenes de alta calidad que inclusive se proyectan en mega pantallas.

Pero para generar imágenes realistas y de alta definición, se requiere de algoritmos avanzados que demandan un costo computacional alto y dependiente tanto del tamaño de la imagen y como de la complejidad de su modelo; uno de éstos algoritmos es la técnica de sintetizado de imagen llamada trazado de rayos, con el cual, la generación de un sólo fotograma nos llevaría horas e inclusive días en una computadora ordinaria. Considerando, que para realizar una película de animación, se requiere de entre 24 y 30 fotogramas por segundo, entonces, el tiempo dedicado a la producción de ésta película llega a incrementarse considerablemente.

Por consiguiente, el procesamiento de render normalmente se lleva acabo en clúster o agrupación de computadoras, llamado comúnmente *render farm*, en donde, haciendo uso de la premisa divide y vencerás, se hace posible la reducción del tiempo de procesamiento de cómputo. A lo largo del tiempo han surgido distintas arquitecturas para administrar un *render farm* por medio del cómputo distribuido, ya sea usando memoria distribuida, compartida ó sus variantes. Recientemente, con la introducción del *GPU* (de las siglas en inglés *Graphics Processing Unit*), se redujeron los tiempos del proceso de render debido a sus propiedades como: alta paralización, alto rendimiento de cómputo y la latencia, ésta ultima permite la transmisión de datos a una alta velocidad. Además, considerando el beneficio ecológico debido al costo energético al utilizar los GPUs en vez de CPUs, reafirma aún más la utilización de un *render farm* basado sobre GPUs.

1.1. Introducción a los render farms

Los *render farm* surgieron de la necesidad de minimizar los costos relacionados con los tiempos de render, teniendo como objetivo cumplir los lapsos de tiempo específicados para la producción de películas de animación.

Se pueden encontrar con distintos tipos tanto de arquitectura, paralelización de datos y algoritmos de render.

Hablando de los tipos de algoritmos, recientemente, con la evolución de la tecnología se ha empezado a considerar el algoritmo para el trazado de rayos, el cual consiste en lanzar un rayo desde el ojo del espectador hacia el modelo de la escena, guardando el color del pixel en una cuadricula del tamaño de la resolución de la imagen deseada; ademas del color del modelo, se consideran también las propiedades de los materiales y de las luces. Para determinar las sombras, las reflexiones y refracciones se tienen que lanzar rayos secundarios en las direcciones correspondientes.

Tanto la paralelización de los datos como el tipo de arquitectura a utilizar van ligadas, debido a la transmisión de información y la latencia del canal que considere la arquitectura. En un sistema de render basado en el algoritmo para el trazado de rayos, se puede encontrar una granularidad fina cuando se habla de un tipo de paralelización por rayo; y se va incrementando conforme la partición de la imagen sea más grande; considerando una granularidad gruesa cuando el proceso de render se paraleliza por fotograma ó inclusive por proyecto. Dentro del tipo de arquitectura que se puede llegar a emplear se encuentra el distribuido en varias computadoras, el distribuido con varias computadoras multiprocesadores, o una sola computadora con varios procesadores.

Como módulo central y muy importante es necesario de un gestor de trabajos, el cual se encarga de distribuir la demanda de trabajo entre los procesadores encargados de realizar el proceso de render.

1.2. Motivación de la tesis

Aún y cuando los tiempos de render han logrado reducirse, con la introducción de procesadores cada vez más potentes; siempre existirá el reto y la necesidad de generar imágenes de mejor calidad, con mayor definición y sobretodo con mayor rapidez.

Afortunadamente el avance tecnológico ha permitido crear computadoras con *hardware* cada vez más potente, tal es el caso de las GPUs, los cuales son procesadores dedicados al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central.

Adicionalmente se encuentra con otro problema, que no tiene que ver con el sistema sino con los usuarios finales; es decir, debido a que los diseñadores de escenas no cuentan el suficiente capital para adquirir un *render farm* propio, existen compañías dedicadas al proceso de render de imágenes, en las cuales, su negocio principal esta en rentar su *render farm* por un intervalo de tiempo a los clientes; sin embargo, el proceso casi siempre se hace en forma presencial.

1.3. Objetivo de la tesis

El objetivo principal de este proyecto de tesis es realizar un sistema encargado de gestionar las tareas de render dentro de un *render farm* conformado por GPUs y utilizando el algoritmo de sintetizado de imagen para el trazado de rayos. El propósito final es reducir en medida de lo posible, tanto la distribución de trabajo en el *render farm*, como los tiempos del proceso de render, atendiendo las peticiones de los clientes a través de una interfaz Web.

Para poder lograr el objetivo principal de la tesis se tienen que cubrir los siguientes pasos:

- Construcción y configuración del servidor: El *render farm* esta conformado por GPUs, entre los cuales, se realiza la repartición de las tareas de render, por lo tanto se requiere de la instalación tanto del manejador de las tarjetas Nvidia correspondientes y la instalación de CUDA que es la arquitectura y el modelo de programación necesarios para la ejecución de programas que hagan uso del cómputo paralelo basado en GPUs.

- Proceso de render: Se usa el motor Nvidia®OptiX™[2] como base para el proceso de render.

- Distribución del trabajo de render: Se crea un módulo destinado a la gestión de tareas de render, (considerando algunas de las características del balance de cargas) encargado de nivelar la utilización de los GPUs conectados al *render farm*, administrando de la manera más eficiente la demanda de los trabajos a renderizar, para lo cual se consideran las propiedades de las tarjetas que se encuentren conectadas al *render farm*. Es importante destacar que la granularidad de fragmentación del archivo OBJ será del tamaño del fotograma; es decir, el archivo será enviado en su totalidad al GPU; debido a que el método de paralización del trazado de rayos usa memoria compartida, realizando una granularidad fina dentro del GPU, gracias a que los actuales GPUs vienen con una memoria superior a 1GByte; hablando más específicamente, las tarjetas Tesla, se pueden encontrar con una memoria de 3GByte o 6 GBytes; razón por la cual no existe problema alguno para evitar la partición del archivo. Otra de las razones por la cual se decidió ésto, es que actualmente se empieza a considerar el algoritmo para el trazado de rayos en tiempo real.

- Medio de acceso: Brindar el servicio de render a través de una interfaz web, notificando al cliente vía correo electrónico cuando su trabajo haya finalizado y con el enlace donde se pueden descargar las escenas generadas; el tiempo para obtener las imágenes resultantes estará determinado por al ancho de banda de la red a la que se encuentre conectado.

1.4. Contribuciones de la tesis

Para empezar, se realiza un estudio sobre el proceso en la creación de los contenidos digitales y el software tanto libre como comercial que auxilian al diseñador durante las etapas de pre-producción y producción del proceso; específicamente para el diseño y renderizado de las imágenes. Como consecuencia y parte central de la tesis se realiza el estudio de los *render farm* existentes hasta éste momento que auxilian en la generación masiva de imágenes, considerando sus propiedades como la arquitectura, algoritmo y motor de render empleado.

Se establecen las bases para poder generar un gestor de tareas para un render farm basado en GPUs, considerando el balance de cargas para la asignación de tareas dentro de los GPUs, migración de tareas dentro del sistema para el proceso de render; mediante el estudio de la determinación del tiempo de render y memoria requerida para diferentes escenas de prueba, con el fin de lograr un calculo aproximado y útil en la asignación de las tareas a los GPUs.

Se diseña e implementa la arquitectura de un *render farm* basado en GPUs a partir del estudio de las técnicas de asignación y planificación de tareas dentro de un esquema particionado con colas locales.

1.5. Estructura de la tesis

En el capítulo 2, se da una pequeña introducción de los contenidos digitales, específicamente de las imágenes generadas por medio de computación gráfica, se considera una clasificación de las imágenes basado en tiempo ó fps, basada en el método de rasterización, enfocando principalmente la clasificación basada en el proceso de render; se explica rápidamente el proceso de producción de películas de animación; se detalla el algoritmo de sintetizado de imagen para el trazado de rayos, central para esta tesis; se muestra el funcionamiento general de un render y se expone una generalización de los sistemas de render en paralelo. Finalmente se exponen las propiedades de los sistemas multiprocesadores, explicando sus propiedades y técnicas en la administración de los recursos mediante el balance de cargas.

En el capítulo 4, se realiza un estudio del estado del arte correspondiente a nuestra área, se empieza con la definición e importancia del *render farm*, seguida de la explicación del modelado y las herramientas existentes, se continua con el proceso de render exponiendo algunas herramientas auxiliares para su procesamiento; para fines específicos del algoritmo para el trazado de rayos, se hace realiza un estudio de los motores de render que se han creado y finalmente se hace un estudio de los *render farms* tanto comerciales como los realizados en el área de investigación. A partir de lo señalado anteriormente se introduce el GPU y se realiza una observación de los costos energéticos que conllevaría realizar un *render farm* basado en GPU comparado con el *render farm* de Pixar.

En el capítulo 5, para comenzar se introduce la propuesta de esta tesis mediante la arquitectura general del sistema y a nivel más bajo, se expone la arquitectura

específica del sistema detallando la construcción de cada uno de los módulos empleados dentro del sistema de *render farm*: el motor del proceso de render, el gestor de trabajos, la interfaz web y los métodos de intercomunicación empleados.

En el capítulo 6, se exponen los resultados experimentales de cada uno de los módulos desarrollados, principalmente se trabajan con 9 modelos de escena de las cuales se extraen sus características principales y se estudia su comportamiento para realizar tanto las aproximaciones de memoria como de tiempo de render. Éstos dos factores son primordiales para el sistema, ya que se el administrador se basa en éstos términos para poder realizar la asignación de las tareas de render. Se exponen las técnicas de administración de tareas experimentadas y se muestran las interfaz del portal web del *render farm*.

Para finalizar, en el capítulo 7 se presentan las conclusiones de la tesis, contribuciones y se plantea el posible trabajo a futuro referente a esta tesis.

Capítulo 2

Contenidos Digitales

Esta tesis se centra en los contenidos digitales en forma de imágenes y video, mediante éstos se puede visualizar y representar la riqueza de nuestro entorno ó de un entorno imaginario. Para su despliegue y creación se requiere el uso de la computadora y la ayuda de técnicas de computación gráfica. Se les puede encontrar en diferentes áreas, tales como industria de la arquitectura, diseño de entretenimiento, diseño de modelado, en la industria de efectos visuales, investigación, entre otros.

2.1. Clasificación de los contenidos digitales creados mediante computación gráfica

Los contenidos digitales creados por medio de computación gráfica se pueden clasificar de diferentes formas; ya sea considerando como parámetro los fotogramas por segundo utilizados para su creación (en el caso de video), por la resolución de la imagen, considerando el tipo de formato de las imágenes o inclusive por el método de render empleado. En [5] se presenta una clasificación basada en el tiempo, rasterizado y el proceso de render. A continuación se describe ésta clasificación y en las siguientes secciones se detallan a profundidad:

Clasificación basada en tiempo El almacenamiento es en forma de píxeles o mues-

(a) Weta Digital, 2009 Fox © (b) Imagen obtenida de [3] (c) Imagen obtenida de [4]

Figura 2.1: Industria de las imágenes realistas creadas mediante computación gráfica.

tras por intervalo de tiempo, conocido como fotogramas por segundo ó fps. Se usa en contenidos digitales como la animación, cine o video.

Clasificación basada en rasterizado El almacenamiento se hace forma de píxeles fijos, algunos formatos conocidos son jpeg, mapa de bits png, entre otros. Se usa en imágenes o fotogramas, fotografías, etc.

Clasificación basada en el proceso de render Este tipo de contenido digital es generado mediante ecuaciones matemáticas o conjuntos de datos numéricos; tales como diseños en 2D y 3D, modelos y objetos. Se pueden almacenar en forma de vectores o en forma de escenario gráfico.

En las siguientes secciones se da una explicación detallada de cada una de las clasificaciones mencionadas anteriormente.

2.1.1. Clasificación basada en tiempo

Los fps son la base fundamental de la clasificación de los contenidos digitales basada en tiempo, éstos representan la frecuencia con la que los dispositivos reproducen imágenes consecutivas únicas permitiendo ser apreciadas en forma de animación. Se fundamenta en el hecho de que el sistema visual humano es capaz de procesar de 10 a 12 imágenes separadas por segundo, buscando como objetivo garantizar que no se llegue a apreciar la discontinuidad del movimiento.

Apartir de lo señalado anteriormente surgieron los denominados estándares televisivos [6], detallados a continuación:

Comisión nacional de sistema de televisión (del inglés *National Television System Committee*, NTSC), fue desarrollado en EUA y usado por primera vez en 1954, consiste de 525 líneas horizontales de pantalla y 60 líneas verticales. Éste formato se usa particularmente en EUA, México, Canada y Japón, la frecuencia de actualización de$29.97Hz$ o $29.97fps$ y la resolución digital estándar de $720x480$ píxeles para DVDs, $480x480$ para Super Video CDs (SVCD) y $352x240$ píxeles para Video CDs (VCD).

Línea de fase alternada (del inglés *Phase Alternating Line*, PAL), fue desarrollado en Alemania en el año de 1967, usa 625 líneas verticales y 50 líneas horizontales de pantalla. Éste formato es usado en Europa Occidental, Australia, Nueva Zelenada y en algunas zonas de Asia; siendo la frecuencia de actualización de $25Hz$ o $25fps$ y la resolución estándar es de $720x576$ píxeles para DVDs, $480x576$ píxeles para SVCD's y $352x288$ píxeles para VCDs.

Color secuencial con memoria (del francés *Système en Couleur avec Mémoire*, SECAM), fue desarrollado en Francia en el año de 1967, usa 625 líneas verticales y 50 líneas horizontales de pantalla, éste formato es usado particularmente en Francia y en algunas áreas de Rusia; comparte algunas similitudes técnicas con PAL sin considerar los estándares de DVD.

Para resumir, en la tabla 2.1 se muestran los estándares descritos anteriormente.

Estándar	fps	resolución	
NTSC	29.97 a 59.94	DVD SVCD VCD	720x480 480x480 352x240
PAL	25 a 50	DVD SVCD VCD	720x576 480x576 352x288
SECAM	25 a 50	DVD SVCD VCD	— 480x576 352x288

Tabla 2.1: Estándares de los formatos televisivos analógicos.

El estándar típico de señales de vídeo se le conoce como señal de video entrelazado, en donde cada fotograma de datos de video que se despliega en el sistema estándar es dividido en dos campos, el primer campo puede contener las líneas impares del fotograma y el segundo fotograma puede contener las líneas pares del mismo fotograma; así, los dos campos que constituyen un solo fotograma son recibidos y desplegados sucesivamente en el sistema estándar, aparentando ser uno solo.

De reciente aparición se tiene una nueva industria de estándares para transmitir imágenes y sonidos mediante el uso de señales digitales y se denominada Televisión digital (DTV) [7] Comparado con la tecnología analógica, la compresión digital permite transmitir más canales, con mejor calidad de imagen y por lo tanto se mejoran las aplicaciones interactivas. La televisión digital, es capaz de procesar las señales de video de los canales en una arquitectura paralela y convertir el estándar de señales de video en un fotograma no-entrelazado.

Los formatos televisivos digitales pueden subdividirse en las siguientes categorías: Televisión de baja definición (LDTV), Televisión de definición estándar (SDTV), Televisión de definición mejorada (EDTV) y Televisión de definición alta (HDTV).

El formato HDTV ya se utiliza en los EE.UU y en muchos países se desea reemplazar los sistemas analógicos de televisión PAL y NTSC. Hoy en día es posible contar con una resolución de $1920x1080$ píxeles (progresivo, velocidad de fotograma 23.976, 24, 25, 29,97, o 30, formato 1080p).

En la tabla 2.2 se detallan estos formatos televisivos digitales y en la figura 2.2 se muestran las resoluciones comunes de video.

2.1.2. Clasificación basada en raster

Una imagen rasterizada, es una matriz de puntos de datos que representa una estructura de una red, generalmente rectangular de píxeles o puntos de color; las imágenes rasterizadas se almacenan en archivos de imagen con distintos formatos y son dependientes de la resolución, es decir, no se pueden escalar arbitrariamente sin perder

Estándar	Velocidad de lectura	Tasa de imagen (fps)	Resolución
LDTV	15.75 kHz (60i)	24p, 30p, 60p or 60i	480 x 640
SDTV	31.5 kHz (60p)	24p, 30p, 60p or 60i	480 x 704
EDTV	45 kHz (60p)	24p, 30p, 60p	720 x 1080
HDTV	33.75 kHz (60i)	24p, 30p, 60i	1080 x 1920

Tabla 2.2: Estándares de los formatos televisivos digitales.

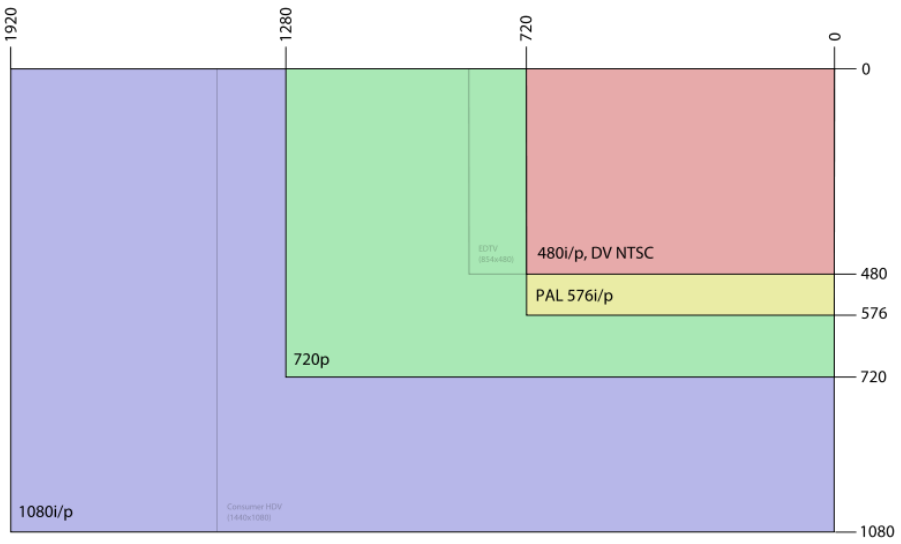


Figura 2.2: Resoluciones comunes de vídeo.

cierta calidad, a diferencia de las imágenes vectoriales, las cuales se pueden escalar a la calidad del dispositivo que la renderiza. Las imágenes rasterizadas trabajan de forma más práctica con fotografías e imágenes foto-realistas, mientras que los gráficos vectoriales a menudo funcionan mejor para la composición tipográfica o de diseño gráfico. En la tabla 2.3 se muestran los diferentes tipos de formatos de archivo que existen para cada tipo de imagen.

2.1.3. Clasificación basada en el proceso de render

El proceso del render es un término heredado del arte, hace referencia a la creación de imágenes sombreadas para formar modelos por computadora en 3D. A lo largo del tiempo han surgido diferentes algoritmos para realizar el proceso de render que permiten la generación de imágenes, a continuación se detallan los principales.

Tipo de imagen	Formato de archivo
Raster	ANI, ANIM, APNG, ART, BEF, BMP, BSAVE, CAL, CIN, CPC, CPT, DPX, ECW, EXR, FITS, FLIC, FPX, GIF, HDRi, ICER, ICNS, ICO, CUR, ICS, ILBM, JBIG, JBIG2, JNG, JPEG, JPEG 2000, JPEG-LS, JPEG-HDR, JPEG XR, MNG, MIFF, PBM, PCX, PGF, PGM, PICtor, PNG, PPM, PSD / PSB, PSP, QTVR, RAD, RGBE, SGI, TGA, TIFF, WBMP, WebP, XBM, XCF, XPM
Raw	CIFF, DNG, ORF
Vectoriales	AI, CDR, CGM, DXF, EVA, EMF, Gerber, HVIF, IGES, PGML, SVG, VML, WMF, XAR
Compuestas	CDF, DjVu, EPS, PDF, PICT, PS, SWF, XAML
Relacionadas	Exchangeable image file format (Exif), Extensible Metadata Platform (XMP)

Tabla 2.3: Formatos de archivo de imágenes.

Determinación de la superficie visible

Muestra solo aquellas partes de la superficie que son visibles para el espectador, dentro de los principales algoritmos encontramos: [8]:

- Z-buffer: Desarrollado por Catmull en 1974 [9], es uno de los algoritmos de superficie visible más simples de implementar, ya sea en software o en hardware; se encuentra en el hardware de casi todos los videojuegos y tarjetas gráficas. Se enfoca en que el problema real es encontrar el polígono más cercano al centro de cada píxel, el cual puede ser un problema más fácil que encontrar un orden verdadero de profundidad en un espacio de pantalla continua. El algoritmo consiste en que para cada píxel, se almacena un valor real z, que es la distancia al triángulo rasterizado más cercano hasta el momento. El z-buffer se inicializa por primera vez para manejar el valor más lejano que puede ser representado.
- Árbol BSP (del inglés *binary space partitioning*, BSP): Desarrollado por Fuchs, Kedem y Naylor en 1980[10], se utiliza en casos en donde se generan muchas imágenes de geometría similar desde diferentes puntos de vista, tal es el caso de aplicaciones como los videojuegos. El aspecto clave del árbol BSP es que realiza un preprocesamiento para crear una estructura de datos útil desde cualquier punto de vista. De tal forma que desde cualquier cambio de punto de vista se usa la misma estructura sin necesidad de algún cambio. Éste algoritmo es un ejemplo del algoritmo del pintor, en donde se dibuja cada objeto de atrás hacia enfrente con cada nuevo polígono sobrepuesto sobre el polígono anterior. El problema con este algoritmo es el ordenamiento de los polígonos, ya que el orden relativo de múltiples objetos no esta bien definido.

- Trazado de rayos o Ray casting: Propuesto por Appel en 1968 [11], determina la visibilidad de las superficies por medio del trazado de rayos de luz imaginarios provenientes el ojo del espectador hacia los objetos en la escena sobreponiendo una ventana sobre la vista arbitraria del plano seleccionado. La ventana puede ser pensada como si fuera dividida dentro de una malla regular, cuyos elementos corresponden a píxeles de una resolución definida. Entonces, para cada píxel de la ventana, es lanzado un rayo desde el centro de proyección a través del centro de los píxeles en la escena; estableciendo el color del píxel al color del objeto más cercano al punto de intersección.

Determinación de iluminación y sombra (iluminación global)

Un modelo de iluminación procesa el color en un punto en términos de la luz emitida directamente por la fuente de luz y la luz que alcanza al punto después de ser reflectada y trasmitida a través de otras superficies. Ésta transmisión y reflexión de luz es llamada iluminación global. En contraste, la iluminación local es la luz que viene directamente desde la fuente de luz al punto que será sombreado [8].

- Trazado de rayos recursivo: Desarrollado por Appel en 1968 [11], el algoritmo simple para el trazado de rayos determina el color de un píxel en la intersección más cercana entre un rayo proveniente del ojo del espectador y un objeto. Para calcular las sombras, se dispara un rayo adicional desde el punto de intersección hacia cada una de las fuentes de luz; y si uno de estos rayos de sombra intersecta cualquier objeto a lo largo del camino, entonces el objeto esta en la sombra en ese punto y el algoritmo de superficie ignora la contribución de la fuente de luz del rayo de sombra.

- Trazado de rayos recursivo de *Whitted*: Propuesto por Whitted en 1980 [12], extiende el algoritmo para el trazado de rayos para incluir reflexiones especulares y trasparencias refractivas. En adición a los rayos de sombra, éste algoritmo genera rayos de reflexión y rayos de refracción desde el punto de intersección.

- Radiosidad: Aunque el método de trazado de rayos hace un excelente trabajo al modelar la reflexión especular y en la falta de dispersión en la trasparencia de refracción, aún no hace uso de dirección en la luz ambiental. El algoritmo de radiosidad por su parte, utiliza modelos de ingeniería térmica para la emisión y reflexión de radiación, eliminando la necesidad de la luz ambiental y proporcionando un tratamiento adecuado de reflexiones entre los objetos. Fue introducido por [13] en 1985; estos algoritmos suponen la conservación de energía de luz en un ambiente cerrado. Toda la energía emitida o reflectada por cada superficie es representada por la reflexión absorbida o transmitida por otras superficies. La velocidad a la cual la energía deja una superficie se llama radiosidad, y es la suma de las velocidades a la cual la superficie emite energía y la reflecta ó la trasmite desde la superficie u otras superficies. Al contrario con otros algoritmos de render, éste método primero determina todas las interacciones de luz en un

ambiente, de manera independiente, dentro de una vista . Entonces, una o más vistas son renderizadas, sólo la determinación del *overhead* de una superficie visible y el sombreado de interpolación.

2.2. Industria del entretenimiento

Con la introducción de la computadora en la industria del entretenimiento, se revolucionó la forma en que se realizaban los procesos, propiciando resultados con efectos especiales cada vez más realistas; como ejemplos tenemos los videojuegos y las películas de animación.

Concretamente en el pasado, en el desarrollo de una película animada se realizaba mediante el uso dibujos hechos a mano en donde las dificultades principales eran de carácter técnicas y de plazos de tiempo, ya que resultaba muy tardado y difícil dibujar cada una de las escenas con propiedades similares. Con la introducción de la computadora y por medio de técnicas de graficación, varios de los procesos que conforman la generación de imágenes han ido evolucionando, permitiendo la reducción de los plazos de tiempo de producción y obteniendo animaciones con mucho mejores resultados y estándares de calidad.

Pero para que todo sea posible, se hace uso de un *pipeline* que permite llevar al término la producción de una película de animación. Las etapas que comúnmente conforman éste proceso pre-producción, producción y post-producción, cada una a si vez tiene subprocesos enlistados a continuación[14]:

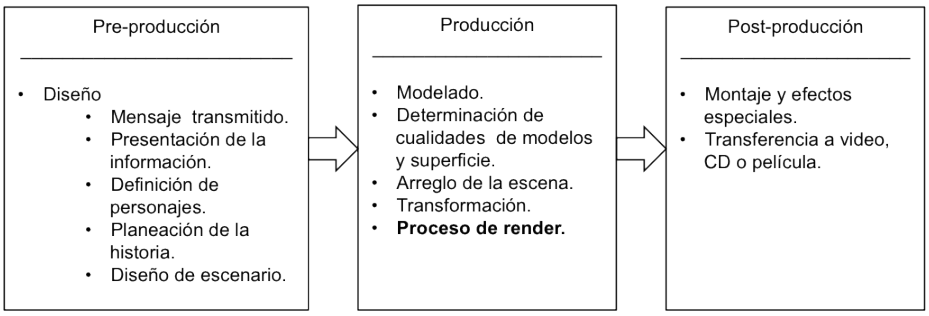


Figura 2.3: *Pipeline* de la producción de una película de animación.

Pre-producción - Diseño Es un tipo de proceso de planeación, se enfoca en el mensaje que será transmitido a la población y en el método a seguir en la presentación de la información; se consideran tanto el tiempo como la expresión de los personajes. Se hace un borrador de los personajes, modelos, imágenes y sonidos que van a ser utilizados en la animación. El escenario es diseñado a partir de la planeación de la historia.

Producción - Modelado Aquí empieza la producción de la animación en 3D, se modelan los personajes considerando el ancho, el largo y la profundidad con

valores numéricos. Como primer paso y con ayuda de la computadora se hace un esqueleto tridimensional del modelo de cada personaje, el cual pareciera estar hecha de hilos.

Producción - Determinación de cualidades de modelos y superficie En la superficie de éste esqueleto se definen las propiedades de los colores, texturas y materiales (éstas características son las simulaciones de las superficies de los objetos en vida real). Se producen imágenes realistas definiendo los materiales de los objetos, por ejemplo: trasparencias como cristal, luz muy ligera o brillante como el cromo de luz reflejante, mate como el plastico y de luz absorbente, entre otros.

Producción - Arreglo de la escena Se establecen las escenas donde los eventos van a ocurrir, situando los arreglos en la escena de acuerdo a las posiciones de los personajes, objetos y accesorios; se consideran los movimientos que se van a realizar en el tiempo determinado por el guión técnico y el guión gráfico. En ésta sección, también se posicionan las fuentes de luz y las cámaras de acuerdo al tipo de atmósfera que va a ser creada; determinando la densidad y color de la luz. Finalmente se posiciona la cámara en escena de acuerdo al punto de vista deseado.

El arreglo de escena se hace de forma tal que sea visto el primer fotograma de la animación. Se colocan en la línea de tiempo los fotogramas claves de los objetos o los personajes que se planea sean movidos, en éstos puntos específicos, se aplican los movimientos de acuerdo al flujo del escenario y los efectos que los personajes deben seguir.

Producción - Transformación Los movimientos intermedios entre dos fotogramas claves son calculados por computadora a través de software de animación; cabe señalar que los movimientos en la escena no se limitan solo a los movimientos de personajes. Con ayuda del software para la animación en 3D, es posible cambiar ángulos de cámara, y el color y la densidad de la luz con respecto al tiempo, alterando las características de las superficies de forma tal que es posible obtener imágenes que no existen en la vida real.

Producción - Proceso de render de los modelos Una vez que se cuenta con la escena diseñada en 3D, por medio de esta operación se puede apreciar en la pantalla de la computadora las características de las superficies y la incidencia de las luces sobre ésta. Este proceso termina cuando los fotogramas se sitúan en secuencia.

Post-Producción - Montaje y efectos especiales En éste proceso se posicionan las imágenes renderizadas sobre una líneas de tiempo, se añaden los sonidos, música y efectos especiales. Éstos son elementos muy importantes, ya que para la audiencia, el nivel de percepción aumenta cuando la imagen viene con sonidos. La planeación de éstos elementos son planeados durante el proceso de diseño.

Post-Producción -Transferencia a video, CD o película Es el paso final del proceso, se toman los vídeos descomprimidos y los convierte en un video al formato requerido para la entrega. Por ejemplo: película, CD, DVD, Quick time, etc.

2.2.1. Ejemplos del tiempo de proceso de render en la generación de películas de animación

Los estudios de animación Pixar (líder en la creación de historias originales en el medio de la animación por computadora), en 1995 creó la primer película totalmente animada mediante la computadora llamada *Toy Story*. Tiene una duración de 77 minutos, para su desarrollo generó un terabyte de datos y requirió de 800,000 horas máquina para el proceso de render, se requirió de un *render farm* compuesto de 117 computadoras *SUN Microsystems*. Se usaron tres sistemas de software propietario: $Marionette^{TM}$, es un software de animación utilizado para modelar, animar e iluminar; $Ringmaster^{TM}$, un sistema de software la gestión de la producción para planificar; y una versión mejorada de *RenderMan* para alta definición y síntesis de imágenes fotorealistas.

Otro ejemplo es *Revenge of the Sith* creada en el 2005, el tiempo total del proceso de render fue de 6.6 millones de horas; *Madagascar 2* creada en el 2008, requirió de 30 millones de horas de proceso de render. Finalmente para *Monsters vs Aliens* creada en el 2009, el tiempo de render total fue de aproximadamente 45 millones de horas de cómputo.

Como se puede observar la tendencia en las películas de animación es crear animaciones cada vez más realistas y por lo tanto se consumen más recursos de cómputo.

2.3. Algoritmo para el trazado de rayos

Para fines de esta tesis, es necesario detallar el algoritmo para el trazado de rayos recursivo (mencionado en la sección 2.1.3 propuesto por Whitted [12]), éste algoritmo extiende el algoritmo para el trazado de rayos propuesto por Appel [11], incluyendo reflexiones especulares y trasparencias refractivas. Además de rayos de sombra, éste algoritmo genera rayos de reflexión y rayos de refracción partiendo del punto de intersección. Los rayos de sombra, reflexión y la refracción son comúnmente llamados rayos secundarios para diferenciarlos de los rayos primarios del ojo del espectador. Si el objeto es especularmente reflectivo, entonces el rayo de reflexión es reflectado sobre la superficie normal en dirección de $\bar{R}$. Si el objeto es trasparente, y si el total de reflexión interna no ocurre, entonces el rayo de refracción es enviado hacia el objeto a lo largo de $\bar{T}$, en el ángulo determinado por la ley *Snell* [8].

Cada uno de estos rayos de reflexión y refracción pueden, a su vez, generar rayos de sombra, reflexión y refracción recursivos; formando en consecuencia un árbol de rayos. En el algoritmo, una rama es terminada si los rayos reflectados y refractados fallan en la intersección de un objeto, si un máximo de profundidad específicado es

alcanzado, o ó si el sistema corre fuera de almacenamiento. El árbol es evaluado desde abajo hacia arriba, y cada intensidad del nodo es procesada como una función de las intensidades de sus hijos.

La ecuación de iluminación de Whitted esta representada en la ecuación 2.1.

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + \sum_{1\leq i\leq m} S_i f_{att_i} I_{p\lambda_i}[k_dO_{d\lambda}(\bar{N}\cdot\bar{L}_i) + k_s(\bar{N}\cdot\bar{H}_i)^n] + k_sI_{\gamma\lambda} + k_tI_{t\lambda} \quad (2.1)$$

Donde $I_{\gamma\lambda}$ es la intensidad del rayo reflectado, k_t es el coeficiente de transmisión entre un rango de 0 y 1, y $I_{t\lambda}$ es la intensidad de los rayos refractantes trasmitidos. Los valores $I_{\gamma\lambda}$ y $I_{t\lambda}$ son determinados evaluando recursivamente la ecuación 2.1 en la superficie más cercana donde se intersectan los rayos reflejados y transmitidos. Para aproximar la atenuación con la distancia, Whitted multiplica I_λ calculado para cada rayo por la inversa de la distancia que viaja por el rayo en lugar de tratar S_i como una función delta.

En el algoritmo 1 se muestra el pseudocódigo del trazado de rayos de Whitted, en donde se realiza el trazado de rayos para cada uno de los píxeles que componen la malla del plano de vista.

Algoritmo 1 Pseudocódigo del trazado de rayos propuesto por Whitted.

```
seleccionar el centro de proyección y la malla en el plano de vista
for cada línea escaneada en la imagen do
    for cada píxel en la línea escaneada do
        determinar el rayo desde el centro de proyección a través del píxel;
        pixel ←RT_trace(ray, 1)
```

En el algoritmo 2 se determina la intersección más cercana del rayo con un objeto y llama a RT_shade para determinar la superficie en ese punto. En el algoritmo 3 se determina la intersección con el color ambiental, después un rayo de sombra se dispara hacia cada luz en el lado de la superficie que va a ser sombreada para determinar la contribución de color. Un objeto opaco bloquea la luz totalmente, mientras que uno transparente escala la contribución de luz. Se hace una llamada recursiva a RT_trace para manejar los rayos de reflexión y refracción hasta que se alcanza la profundidad definida.

Resumiendo, el algoritmo para el trazado de rayos calcula independientemente el color de cada píxel como una función de los componentes de luz ambientales, difusos, reflectivos y refractivos. En un principio, los rayos son trazados desde el observador hacia los objetos de la escena, atravesando una pantalla virtual, y en dirección a las fuentes de luz (Véase algoritmo 1). Se determina el primer objeto intersectado por el rayo (Véase algoritmo 2), y después se calcula el color del píxel como la suma de los diferentes componentes de luz (Véase algoritmo 3). La luz ambiental depende solo de las propiedades de las superficies de los objetos, la luz difusa se calcula considerando cada luz. Para determinar las sombras, los rayos son trazados desde el punto de

Algoritmo 2 Función de trazado de rayos.

```
function RT_color RT_TRACE(RT_ray ray, int profundidad)
    determinar la intersección más cercana del rayo con un objeto;
    if objeto golpeado then
        calcular normal en la intersección;
        RT_shade(objeto más cercano golpeado,rayo,intersección,normal,profundidad)
    else
        return BAKGROUND_VALUE;
```

intersección hacia cada fuente de luz y la contribución del color del píxel dependerá de si el rayo intersecta o no al objeto. Los rayos de reflexión y refracción se trazan de manera similar, en sus direcciones apropiadas, y sus contribuciones se añaden al color final del píxel; cuando uno de estos rayos de reflexión o refracción intersecta otro objeto, el proceso se repite.

2.3.1. Complejidad del algoritmo para el trazado de rayos de Whitted

Aunque las imágenes creadas por el algoritmo para el trazado de rayos son de alta calidad, la técnica se considera muy costosa debido al tiempo de cómputo tan alto que se requiere para calcular la intersección de cada rayo con todos los objetos. El número de total de rayos trazados dependen de la resolución de la imagen, es decir es igual al producto de m líneas por n píxeles en cada línea; entonces el orden el algoritmo para el trazado de rayos para escenas simples seria $O(n^2)$; sin embargo, la complejidad para escenas complejas llegaría a ser $O(n^3)$, puesto que el algoritmo tambien depende del número de objetos.

La mayoría del tiempo se consume en procesar el calculo de las intersecciones de los rayos y las superficies, Whitted reporta que en escenas complejas, el procesamiento de las intersecciones consumen más del 95 % del programa [15, 16]. Para reducir la complejidad de las intersecciones de los rayos con las superficies, han surgido varios algoritmos, denominados estructuras de aceleración, las cuales se enfocan en minimizar el tiempo de procesamiento de las intersecciones de los rayos con las superficies.

Algunos ejemplos de estas estructuras de aceleración son: BSP-tree [17] con una complejidad de $O(n \log^2 n)$, kd-tree [18] con una complejidad de $O(n \log n)$, octrees, BVH, etc. La complejidad logaritmica del trazado de rayos con respecto al tamaño de la escena permite el manejo de grandes modelos que anteriormente eran imposible manejar. Lo que significa que el tiempo del proceso de render varia lentamente solo para los modelos con más de cien mil triángulos [19].

Algoritmo 3 Determinación de los rayos de sombra, rayos de reflexión y de refracción.

function *RT_color* RT_SHADE(*RT_object objecto*, *RT_ray rayo*, *RT_point punto*, *RT_normal normal*, *int profundidad*)
  RT_ color *color*; ▷ Color del rayo
  RT_ ray *rRay*, *tRay*, *sRay*; ▷ Rayo reflejante, refractante y de sombra
  RT_color *rColor*, *tColor*; ▷ Color del rayo reflejante y refractante
  color ←término ambiental;
  for cada rayo **do**
    sRay ←rayo hacia la luz desde el punto;
    if (**then** producto punto de la normal y la dirección hacia la luz es positivo)
      calcular la luz bloqueada por superficies opacas y transparentes;
      escalar la contribución de los términos difusos y especulares;
      agregar al color;
  if $profundidad < maxProfunfidad$ **then**
    if el objeto es reflectivo **then**
      $rColor \leftarrow RT\_trace(rRay, profundidad + 1)$;
      escalar rColor mediante el coeficiente especular y agregar a color;
    if el objeto es trasparente **then**
      tRay←rayo en dirección de refracción desde el punto;
      if la reflección total interna no ocurre **then**
        escalar tColor mediante el coeficiente de transmisión y agregar a color;
  return color;

2.4. Funcionamiento general del render

Dentro del proceso de renderización de las imágenes, se requiere de una escena descrita mediante un archivo de texto llamado "descriptor de escena". El descriptor consiste del modelo que constituye la escena, de las propiedades de los materiales de las superficies y de las fuentes de fuentes de luz que iluminan la escena. También se requiere de la posición de una cámara virtual (la cual hace la función de la vista del ojo del espectador) y opcionalmente la atmósfera (espacio en donde esta puesta la escena, por ejemplo, un cuarto lleno de humo).

El descriptor de escena se manda al render para sintetizar una imagen correspondiente a la descripción de la escena, mediante cálculos embebidos en los algoritmos de render. Como salida se obtiene la imagen resultante que en realidad es una red de números que representan colores. Ver la Figura 2.4.

En la figura 2.5, se muestra el *pipeline* general del render como un diagrama de bloques donde los datos fluyen desde arriba hacia abajo, la descripción de escena consiste de superficies hechas de materiales y espacialmente compuestas, iluminadas por fuentes de luz, la cámara virtual se posiciona en algún lugar de la escena, y el render observa a través de ella para crear una imagen de la escena desde un punto de vista específico.

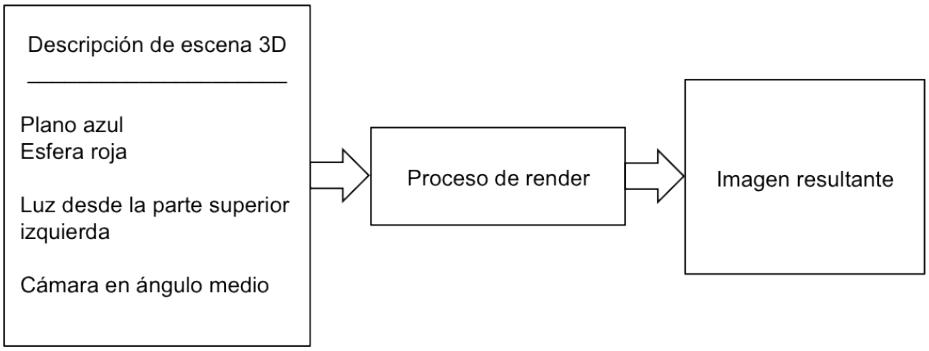


Figura 2.4: Descriptor de escena.

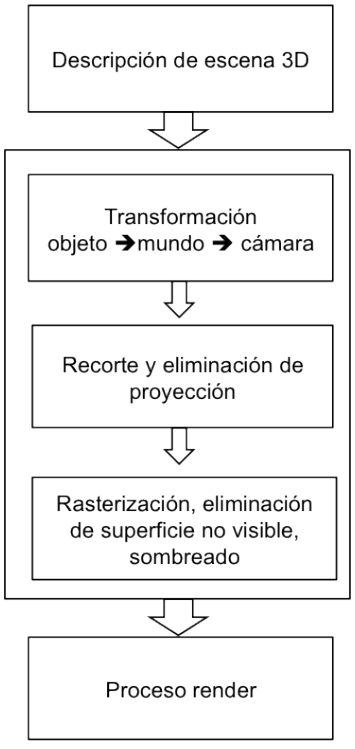


Figura 2.5: Etapas del *pipeline* del proceso de render.

2.5. Métodos de paralelizar el algoritmo para el trazado de rayos

Considerando lo descrito anteriormente, han surgido varios métodos para paralelizar el algoritmo para el trazado de rayos ejemplos de estos son los expuestos por [20], a continuación se resumen:

Particionamiento por espacio de imagen La vista del plano es dividida en regiones, cada cual es completamente renderizada por un procesador individual. Es decir, para cada píxel en una región, el procesador asignado a esa región calcula el rayo de árbol completo, como inconveniente se tiene la memoria local del procesador y el desequilibrio de carga, ya que la imagen puede estar concentrada en cierta parte de la escena.

Particionamiento en el espacio de objeto El espacio 3D donde se encuentran los objetos se dividen en subvolúmenes, los cuales pueden no ser del mismo tamaño. En la fase de inicialización del trazado de rayos, cada subvolúmen es empaquetado y enviado a un procesador en particular. Cuando los rayos son emitidos durante el proceso de render, son pasados de procesador en procesador ya que viajan a través del espacio de objeto. Como ventaja se tiene un balance de carga equilibrado, como desventajas tiene una comunicación excesiva.

Particionamiento del objeto Se asigna cada objeto a un procesador individual. los rayos son pasados como mensajes entre procesadores, que a su vez prueban el rayo para intersectar con los objetos a los que son asignados, Como ventaja se tiene un balance de carga equilibrado, como desventajas tiene una comunicación excesiva.

Clasificación por balance de carga Se clasifica en dos categorías la estática y la dinámica; en el balanceador de carga estático las tareas se asignan a los procesadores y correrán en él durante todo el proceso, sin embargo se debe tener cuidado en asegurar que la carga este equilibrada, sino el algoritmo sufre de bajo rendimiento. En el balanceador de carga dinámico: las asignaciones de los procesos son determinadas en un inicio, después son impulsados por demanda, es decir, cuando un procesador determina que necesita más trabajo por hacer, pedirá una nueva tarea. La clave es distribuir la carga lo más uniformemente posible.

Clasificación por *hardware* Una alternativa a usar sistemas multiprocesador, es emplear una red de estaciones de trabajo actuando como una sola máquina, usualmente basadas en UNIX, las cuales soportan algún tipo de ambiente distribuido como PVM (por las siglas en inglés de *Parallel Virtual Machine*) o MPI (por las siglas en inglés de *Message Passing Interface*). Para este tipo de ambientes han sido propuestos algoritmos para SIMD (por las siglas en inglés

de *Single Instruction, Multiple Data*) y para MIMD (por las siglas en inglés *Multiple Instruction, Multiple Data*).

Particularmente, en la clasificación por hardware, han surgido varias propuestas arquitecturales en ambientes distribuidos, a continuación se mencionan algunos ejemplos [20]:

- Una forma de realizar el proceso de render en paralelo es usar una sola computadora multiprocesador, como Thinking Machines CM-5, Intel Paragon, Cray T3E o sobre un procesador paralelo de propósito general.
- Alternativamente, se puede usar una red de estaciones de trabajo, a éste enfoque se le conoce como cómputo distribuido o cómputo en clúster; es conceptualmente similar a un multiprocesador, pero cada elemento de procesamiento consiste de una máquina interconectada a una red, la cual es en muchos casos más lenta que la interconexión en red de multiprocesadores. Para comunicarlas se requiere de algún tipo soporte de programación en ambientes distribuidos, tales como PVM [21] ó MPI [22] .

El tener un clúster dedicado al proceso de render no es nada barato y muchas de las personas que se dedican al diseño de gráficos por computadora no poseen el capital para por costear uno, como consecuencia han surgido otra forma de clasificación, en la cual los propietarios del clúster lo ponen en renta mediante un portal web, en donde los clientes envían los trabajos que necesiten renderizar.

Capítulo 3

Planificación en sistemas multiprocesador

Desde el origen de la computadora, siempre se ha buscado mayor potencia de cómputo. En un inicio, ésta se obtenía incrementando la velocidad del ciclo del reloj, hasta que se alcanzó los límites fundamentales de la velocidad de reloj. Según la teoría de la relatividad de Einstein, ninguna señal eléctrica puede propagarse a una velocidad mayor que la de la luz, que es de aproximadamente 30 cm/ns en el vacio y de aproximadamente 20 cm/ns en un alambre de cobre. Esto implica, que una computadora con un ciclo de reloj de 10 GHz, las señales no pueden viajar más de 2 cm en total; entonces se empezaron a reducir las computadoras. Sin embargo ésta reducción también llego a un limite, encontrándose con el problema fundamental de la disipación de calor. Cuanto más rápido procesa una computadora mayor será el calor generado, y cuanto más pequeña sea, es más difícil controlar el calor generado.

Alternativamente surgió una solución al problema de aumentar la velocidad de procesamiento, el cual es el uso de computadoras paralelas; constituidas de muchas CPUs que en conjunto obtienen una mayor potencia de cómputo. Surgiendo así diferentes arquitecturas de sistemas multiprocesador por ejemplo multiprocesadores de memoria compartida, multiprocesadores con trasferencias de mensajes, los sistemas distribuidos de área extensa ó sistemas distribuidos. Y con esto surge cierta dificultad para poder comunicar unas con otras y dependiendo del tipo de arquitectura es el tipo de método de programación y comunicación empleada.

Hablando específicamente del software para sistemas multiprocesador, éstos se pueden clasificar en sistemas donde cada procesador cuenta con su sistema operativo propio ó los procesadores con sistema operativo compartido[26].

3.1. Planificación de tareas en sistemas multiprocesador

Con la introducción más procesadores, ahora es necesario resolver el problema de planificación. Mientras que en un sistema monoprocesador el único problema que existe en la planificación es saber que proceso será el siguiente en ejecutarse; en un sistema multiprocesador, el problema es bidimensional. Es decir, el planificador además de elegir el siguiente proceso en ejecutarse, tiene que decir cuando y en que orden debe hacerlo. Estos problemas comúnmente son denominados:

- El problema de la asignación: Implica decidir en que procesador se ejecutarán cada una de las tareas.
- El problema de planificación: Definir cuando y en que orden se ejecutará cada tarea.

Adicionalemente, otro problema que se tiene es la dependencia de tareas, sin embargo particularmente para esta tesis solo nos interesan las tareas independientes.

3.1.1. Enfoques de programación en sistemas multiprocesador

Desde la aparición de los sistemas multiprocesador, se han desarrollado varios enfoques de planificación para adecuar el funcionamiento del sistema a su arquitectura; es importante hacer una revisión de los tipos de esquemas de planificación de tareas que existen, de manera que se comprenda la arquitectura que la mayoría de los *render farms* adoptaron [27, 28]:

Esquema de planificación global con tiempo compartido Todas las tareas listas para ejecutarse son almacenadas en una cola única; para ejecutar una tarea se selecciona a la primera tarea en la cola ó aquella con la prioridad de ejecución más alta (dependiendo del tipo de planificación), la ejecuta por un periodo de tiempo, regresando a la cola si aún no ha finalizado y continua con la siguiente tarea. Éste enfoque es muy común en los sistemas multiprocesadores con memoria compartida. La ventaja principal es que provee un balance de cargas automático debido a que ningún procesador puede estar en estado ocioso si existe una tarea esperando en la cola. Como desventajas se encuentra es que la contención para la cola global crece con el número de procesadores, otra es que una tarea típicamente se ejecutará en diferentes procesadores cada vez que sea planificado. Como resultado, los hilos no pueden almacenar datos dentro de la memoria local y el estado del cache se limpia cada vez que se vuelve a planificar. Un tercer problema con la cola global es que las tareas en un trabajo se planifican de manera no coordinada, si las tareas no interactuán con otras no hay problema; sin embargo, en caso contrario y si la interacción es alta, la

falta de coordinación en la planificación implica que los patrones no se ejecuten al mismo tiempo.

Esquema particionado con colas locales Es una alternativa a la cola global, se usa en los tipos de arquitecturas de memoria distribuida o en donde los procesadores son heterogéneos. Por cada procesador existe una cola local, en donde se reciben tareas independientes, para su posterior ejecución. La principal ventaja del enfoque particionado es un problema de planificación multiprocesador se reduce a un problema monoprocesador. Como desventaja al utilizar una cola local se tiene que puede existir distribución de trabajo inequitativo entre los procesadores, llegando a casos en los cuales se tienen colas locales con mucha carga de trabajo y otras que por el contrario pueden incluso llegar a estar en estado ocioso. Lo ideal en estos casos es contar con una política de asignación eficiente y de bajo costo que logre mantener la equidad de carga de trabajo entre los procesadores y una migración ocasional para superar el desbalance.

Particionamiento variable Otro enfoque es dividir el procesador en un conjunto disjunto y ejecutar cada trabajo en una partición distinta, existen diferentes enfoques de particionamiento: particionamiento arreglado: es cuando los tamaños de partición se definen anteriormente por el sistema administrador; particionamiento variable: los conjuros de nodos son particionados de acuerdo al requerimiento de usuarios cuando se envían los trabajos. particionamiento adaptativo: es cuando las particiones son automáticamente establecidas por el sistema de acuerdo a la carga existente de trabajos enviados. particionamiento dinámico: es cuando el tamaño puede cambiar en el tiempo de ejecución para reflejar cambios entre os requerimientos y carga.

Particionamiento dinámico con dos niveles de planificación Una forma de reducir el tiempo de espera de las colas de trabajos es prevenir a las tareas de monopolizar demasiado a los procesadores cuando el sistema esté muy cargado. Esto se realiza mediante esquemas de particiones adaptativas. Muchas ideas acerca de como asignar los procesadores se han propuesto. El problema con el particionamiento adaptativo, es que una vez que se asignan un número de procesadores son asignados a un trabajo, el número se ajusta hasta que el trabajo termina. Los cambios en la asignación durante el tiempo de ejecución se provee mediante el particionamiento dinámico. Para soportar tal comportamiento, las aplicaciones requieren usar un modelo de programación que pueda expresar cambios en los requerimientos y maneje los cambios de asignación inducidos por el sistema.

Planificacion por grupos La única manera de garantizar los tiempo de respuesta interactivos es mediante el rebanado de tiempos. Sin embargo si se hace de una manera no coordinada con una global global, puede generar varias ineficiencias. Mejor dicho, el cambio de contexto debe ser coordinado a través de los procesa-

dores; así todos las tareas del trabajo se ejecutan al mismo tiempo, permitiendo a los procesadores interactuar a una granularidad fina.

3.2. Asignación de tareas en sistemas multiprocesador

El objetivo principal de la asignación de tareas es diseñar un algoritmo de balance de cargas en donde se asigne las tareas a cada procesador en proporción de su rendimiento con el objetivo de tener una carga equitativa del total del trabajo y manteniendo bajas transferencias de comunicación. En muchas aplicaciones es posible realizar estimaciones de la distribución de trabajo, así el programador puede construir un balance de cargas adecuado en cada aplicación específica del programa. Existen dos tipos asignación [29]:

Asignación estática Se realiza en tiempo de compilación. Es eficiente y no introduce costos de procesamiento en tiempo de ejecución. Para multiprocesadores paralelos de tipo UMA (de las siglas en inglés *Uniform Memory Access*), usualmente se pueden planificar las iteraciones de ciclo, de forma de ciclica o de bloques. Para los multiprocesadores tipo NUMA (de las siglas en inglés *Non-Uniform Memory Acces*) el planificador ciclico tiene que tomar la distribución de datos en cuenta.

Asignación dinámica Se realiza en tiempo de ejecución. Y las estrategias de planificación recaen en el tipo de modelo usado, por ejemplo el modelo de cola de tareas empleadas (previamente explicadas en 3.1.1), modelos de difusión y esquemas basados en la predicción del futuro a consecuencia de las pasadas.

- Modelo de difusión: Inicialmente las tareas se encuentran distribuidas y con movimiento entre los procesadores adyacentes, si se detecta un desbalance con los procesadores vecinos. El modelo gradiente, el cual es un mecanismo de balance de carga con una interconexión de red lógica, en donde cada procesador interactúa solo con un conjunto de procesadores predefinidos (vecinos), intercambiando la cantidad de carga entre éstos. La carga se puede clasificar como ligera, moderada y alta; en donde el estado ideal en todos los procesadores es moderado.
- Predicción futura: Otro enfoque es la predicción futura del rendimiento basada en información pasada; en donde la carga del procesador se da como el porcentaje promedio de procesamiento por procesador y el balance de carga involucra cambios de información periodica .

Es muy importante intentar balancear la carga entre los procesadores sin olvidarse tanto de las propiedades de las tareas como de los procesadores en las cuales van a ejecutarse, con el fin de minimizar los errores.

3.3. Balance de cargas dinámico

El objetivo principal es asignar a cada procesador tareas de manera proporcional a su rendimiento, mientras se minimiza la ejecución de la aplicación. La mayoría de los algoritmos para el balance de cargas dinámicos se pueden dividir en dos clases: geométricos y topológicos. Los métodos geométricos dividen el dominio computacional explotando la localización de los objetos en la simulación; éstos métodos se usan para los problemas con interacciones inherentemente geométricos. Los métodos topológicos funcionan con conectividad de interacciones en vez de coordenadas geométricas. La conectividad es en general descrita como un grafo [30].

Maestro esclavo Éste enfoque es un simple paradigma de computación paralela. En su constitución más básica se tiene un procesador maestro que mantiene una cola de tareas que se reparten entre los procesadores esclavo. Los esclavos realizan un procesamiento y después preguntan al maestro por más trabajo. Es un modelo flexible y tiene variantes como incluir múltiples maestros, esclavos jerárquicos y variaciones en la unidad de trabajo. Éste enfoque sólo es apropiado si las tareas se pueden realizar de forma asíncrona e independiente por un solo procesador.

Geometria simple La mayoría de los cálculos mecánicos tienen un fondo geométrico. Y muchas simulaciones físicas, objetos, interactuán sólo si se encuentran geométricamente cerca uno del otro. Estas propiedades permiten el uso de métodos geométricos para dividir la cantidad de trabajo entre los procesadores. Muchos algoritmos de partición explotan la idea de dividir recursivamente el dominio usando líneas o planos, como el método de bisección de coordenadas recursiva (RCB), bisección recursiva imbalanceada (URB), bisección recursiva inercial (RIB).

***Octrees* y curvas en el espacio de llenado** Es un particionado geométrico diferente basado en la division de grano fino de la geometria. La pequeñas piezas son combinadas por una región propietaria de un procesador. La división espacial de grano fino se realiza mediante la división a la mitad simultanea de cada eje de coordenada. Esto produce subregiones en 2D y ocho subregiones en 3D. Nótese que a diferencias de los algoritmos de corte plano, la división es enteramente geométrica y no toma en cuenta la localización de los objetos. Cada subregión se divide nuevamente si contiene múltiples objetos. Una simple estructura de datos conocida como *octree* mantiene el rastro de las relaciones entre estas regiones geométricas. La raíz del *octree* representa la geometría entera. Cuando una región geometrica es dividida, cada uno de los 8 octantes se convierte en hijos del vertice que representa la region. Este tipo de estructura de datos es muy usada en la generación de malla y refinamiento de malla adaptativa. Los *Octrees* tambien son usadas para particionar. La trasversal de los arboles definen un orden global en los niveles del árbol, los cuales corresponden

a objetos individuales. Esta lista ordenada puede ser rebanda para generar cualquier número de particiones. Al algoritmo básico se le llama particionameinto *octree* ó particionamiento de llenado de espacio de curva.

Mejora local Los métodos de balance de cargas locales, se usan en conjuntos de tamaño pequeños, o vecindades, de procesadores cercanos conectados para mejorar el balance de cargas con cada vecindario. Los vecindarios son escogidos para coincidir, sobre muchas iteraciones del balance de cargas local. el trabajo se puede propagar de un vecindario a otro y eventualmente a traves de un arreglo de procesadores locales. Las conexiones topológicas de los datos se usan para seleccionar objetos para migración de forma que intentos para minimizar el costo de comunicación de las aplicaciones.

A diferencia de los métodos globales, cada iteración de los métodos locales es, usualmente, más rápida y eficiente. Porque toda la información y comunicación se realiza dentro de pequeños conjuntos de procesadores, los métodos se escalan bien con el número de procesadores. Por diseño, los métodos locales son incrementales, porque mueven objetos sólo en un grupo pequeño de procesadores. Además, pueden ser invocados sólo para mejorar el balance de cargas, más bien que requerir un balanceo global antes de terminar. Usa sola iteración de los métodos locales pueden reducir una carga de trabajo pesada de procesador significativamente.Ya que el tiempo total de procesamiento se determina mediante el tiempo requerido para la carga más pesada, un pequeño número de iteraciones puede ser todo o que se necesite para reducir el desbalance a un nivel aceptable.

1. Determinación del flujo de trabajo: Para determinar el flujo de trabajo entre los procesadores, se usa un algoritmo de difusión. Estos algoritmos fueron propuestos por Cybenko [31]. En su forma simple el modelo de la carga de trabajo esta dada por la ecuación 3.1

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{3.1}$$

Donde u es la carga de trabajo, α es el coeficiente de difusión. Usando las conexiones del hardware o los patrones de las comunicaciones para describir la malla computacional. La ecuación 3.1 se resuelve usando un esquema de diferencias de primer orden. Desde la plantilla del esquema de diferencias compacto (usando información desde un proceso vecino), el método es local. El método resultante toma la forma de:

$$u_i^{t+1} = u_i^t + \sum_j \alpha_{ij}(u_j^t - u_i j) \tag{3.2}$$

En donde u_i^t es la carga de trabajo después de la iteración t, y la suma se toma sobre todos los procesadores j. Los pesos $\alpha_{ij} \geq 0$ es cero si

los procesadores i y j so están conectados en el grafo del procesador y $1 - \sum_j \alpha_{ij} \geq 0$ para toda i. La elección de α_{ij} afecta la estimación de convergencia del método. Éste depende de la conectividad del procesador debido a la arquitectura del patrón de la comunicación de la aplicación.

2. Selección de objetos para migrar: El segundo paso para el método local es decidir cuales objetos migrar para satisfacer la trasferencia de trabajo en el primer paso. Un número de heurísticas han sido usadas para determinar que objetos se deben transferir. Los objetivos típicos incluyen la minimización del costo de comunicación (a través de la minimizaron del número de los cortes de bordes en el grafo de comunicación de aplicación), minimizar la cantidad de datos migrados, minimizar el número de procesadores vecinos y optimizando las formas de los subdominios ó una combinación de éstos objetivos.

Grafo particionado Un número de algoritmos y herramientas de software han desarrollado para el problema de particionamiento estático una malla computacional. La herramienta más poderosa de estos algoritmos es usar un modelo de grafo de computación, y aplicar las técnicas de particionamiento de grafo para dividirlo a través de los procesadores. En principio, un grafo puede estar construido para cualquier proceso, pero este modelo es más comúnmente usado para aplicaciones basadas en mallas donde el grafo ésta muy relacionado con la malla. A diferencia de los algoritmos de mejora local, los algoritmos de particionamiento estático son globales, examinando todos los datos de una sola vez e intentando encontrar la mejor partición posible.

Métodos híbridos Muchos de los algoritmos de balance de cargas no les convine escoger en particular uno de los algoritmos vistos. Por lo cual combinan métodos para tener algunas ventajas de cada uno.

Capítulo 4

Estado del arte

En la industria del entretenimiento, la generación una película animada requiere de mucho tiempo de procesamiento de CPU (veánse los ejemplos mostrados en la sección 2.2). La solución a este problema, es la construcción de un *render farm* para reducir el tiempo del proceso de render mediante el cómputo paralelo de fotogramas individuales dentro de un ambiente distribuido. Un *render farm* es un clúster o conjunto de computadoras adaptadas para realizar el proceso de render mediante cómputo paralelo; algunos pueden estar compuestos por miles de computadoras interconectadas, en donde cada computadora puede tener su propia memoria, almacenamiento y acceso a memoria compartida.

Los principales elementos que se hallan en un sistema de *render farm* son: el render, el algoritmo de sintetizado de imagen empleado para generar la imagen en 3D, el clúster de computadoras y la arquitectura utilizada, los trabajos a renderizar y el gestor que se encarga de distribuir los trabajos entre los procesos. específicamente, el proceso de render consiste en convertir el modelado de la escena en una imagen, mediante el uso de un algoritmo de sintetizado de imagen (veánse los ejemplos en la sección 2.1.3). Por tal motivo, éste cápitulo se comienza con una breve introducción acerca del modelado, seguida por el proceso de render, los motores de render y el estado del arte de los *render farms* encontrados en la literatura.

4.1. Modelado de escena

Para comenzar, un modelo es una abstracción y representación del mundo real; específicamente hablando del modelado en 3D, se usan modelos para describir objetos en 3D mediante el uso de primitivas matemáticas, por ejemplo: esferas, cubos, conos y polígonos. El tipo de modelo más común esta compuesto por triángulos en 3D que comparten vértices y comúnmente se llaman malla de triángulos. Éstas mallas pueden ser generadas por artistas gráficos usando un programa interactivo de modelado y algunas veces por dispositivos de exploración de objetos del mundo real. En cualquiera de los dos casos, usualmente contienen triángulos tan pequeños que los programas se deben de optimizar para manejar éste tipo de datos.

Para desarrollar la representación matemática de una superficie en 3D normalmente se utilizan programas dedicados, en la tabla 4.1 se dan ejemplos de los principales programas de software auxiliares en el modelado de gráficos en 3D y se muestran algunas de las propiedades principales de éstos.

La mayoría de los programas de software para el modelado mencionados anteriormente cuentan con un proceso de render integrado, sin embargo, se pueden conectar con otros motores de render externos, para poder obtener los resultados deseados.

Una característica importante a resaltar, es que casi todos los programas tienen la posibilidad de poder importar el tipo de archivo OBJ. Lo cual lo hace del formato de archivo, un formato portable.

Nombre	Utilidad	Motor de render	Licencia	Tipo de archivo	Precio
Autodesk 3ds Max	Modelado, animación, iluminación, proceso de render	3DS Max, mental ray, RenderMan, V-Ray, Arion, Indigo, Octane, Luxrender	Propietario	FBX, 3DS, PRJ, AI, DAE, DEM, DDF, DWG, DXF, FLT, HTR, IGE, IGS, IGES, IPT, WIRE, IAM, LS, VW, LP, OBJ, SAT, SHP, SKP, STL, TRC	$3,675
Autodesk Maya	Modelado, animación (video), iluminación, proceso de render, efectos visuales en 3D	Mental ray,	Propietario	AI, PIX, AVI, EPS, GIF, SWF, MAYA IFF, MAYA16 IFF, PSD, PNG, RLA, SVG, SGI, SGI16, SGI MOVIE, MAYA ASCII, MAYA BINARY, MEL, FBX, DXF, OBJ, IGES, AIFF, VRML2, STL	$3,675
Autodesk Softimage	Modelado, animación, creación de juegos de video, iluminación, proceso de render, efectos visuales en 3D	Mental ray	Propietario	AI, AIFC, AIFF, AIF, AMC, AVI, BVH, DAE, DDS, EANI, EPS, EXPR2, FBX, FRAW, FRAW2, LWO, MDD, MI2, MIA, MOV, OBJ, PRESET, PSC, RTF, SCNTOC, WAV	$3,145

Nombre	Utilidad	Motor de render	Licencia	Tipo de archivo	Precio
Blender	Animación, iluminación, modelado, proceso de render, creación de juegos de video, efectos visuales en 3D, esculpido, proceso de render	Cycles pathtracer, scanline ray tracing, YafRay, POV-Ray, MegaPOV, Kerkythea, Sunflow, Indigo, 3Delight, Metropolight, LuxRender	GPL 2+	DAE, BVH, SVG, PLY, STL, 3DS, OBJX3D, WLR	Libre
Cinema 4D	Animación, iluminación, modelado, efectos visuales en 3D	finalRender , FryRender, Maxwell Render, Pixar, Indigo, Octane, SunFlow, V-ray, mental ray	Propietario	FBX, OBJ, 3DS, QD3D, STL, DEM, VRML 1 ,VRML 2 , LWO/LWS, DXF	$995 - $3,695
Houdini	Animación, iluminación, modelado, efectos visuales en 3D	Mantra: Renderman, mental ray	Propietario	FBX	$1,995- $ 4,495
LightWave 3D	Modelado, animación, ilumination, proceso de render, previsualización cine y televisión , Creación de juegos de video	Basado en ilumination global	Propietario	FBX, OBJ, MDD, Autodesk archivos Geometry Cache files	$1,495

Nombre	Utilidad	Motor de render	Licencia	Tipo de archivo	Precio
Modo	Modelado, animación, proceso de render	Fisicamente basado en trazado de rayos	Propietario	LXO, SLDPRT, SLDASM*, LWO2, LWOB, DXF, FBX, 3DS, GEO, EPS, AI, PSD, OBJ, DAE, MDD	$1,195.00

Tabla 4.1: Software auxiliar para el modelado en 3D.

4.2. Proceso de render

En el diseño del modelo, se añaden propiedades a los materiales de los objetos de la escena, se posicionan las luces dentro de la escena y se define el punto de vista del observador; se guardan todas éstas características en un descriptor de archivo y se procede a realizar el proceso de render de la imagen. Arquitectónicamente hablando los tipos de proceso de render se poden clasificar en: local, en red, en clúster y en tiempo real. En la tabla 4.2 se encuentran los principales herramientas de software que auxilian en el proceso de render.

El proceso de render local, se realiza en una sola computadora; el proceso de render en red se puede realizar mediante varias computadoras conectadas en LAN o inclusive a traves de la Web; para el proceso de render en clúster se hace uso del computo distribuido interconectando una red dentro de una red de área local.

Render	Licencia	Algoritmo	Arquitectura	Precio
3Delight	Propietario	Algoritmo de reyes con buena capacidad para el trazado de rayo e ilumina-ción global	Multi-hilos y proceso de render distribuido para *host* con multi-CPU	Dos núcleos - Libre multi-núcleo ilimitado- $900
FinalRender	Propietario	Trazado de rayos e iluminación global	Proceso de render distribuido mediante el núcleo TCP/IP	$1,295
Gelato	Propietario	Algoritmo de reyes modificado	Proceso de render mediante aceleración de GPUs	—
Indigo	Propietario	Proceso de render distribuido con aceleración GPU y multi-hilos	Fotorealistico y basado fisicamente	€595.00
Kerkythea	Freeware	Path Tracing, Photon Maps/Mesh Maps + Irradiance Caching, Diffuse Interreflection	—	—

Render	Licencia	Algoritmo	Arquitectura	Precio
Luxrender	GPL	Para multicomputadoras, soporta IPV6 y aceleración GPU para *path tracing*	Basado en PBRT (fisicamente basado en el trazado de rayos)	Libre
Maxwell Render	Propietario	Iluminación global basado en la variación de la transferencia de luz	Distribuido	$995 incluye 5 nodos libres y $245 por nodo
POV-Ray y MegaPOV	Freeware	Basado en DKB-Trace	No específicado	Libre
Mental ray	Propietario NVIDIA	Iluminación global a través de radiosidad y mapeo de fotones, cáusticos, movimiento borroso de sombras y reflejos, cáusticos de volumen, Final Gather, iluminación fotométrica	Distribuido y limitado en el número de CPUs, Para Maya completo limitado a 2 CPUs y para Maya ilimitado a 8 CPUs	Libre para Maya 4.5 o posteriores, para *render farm* se vende por separado
Metropolight	Freeware	Transportación de luz Metropolis	—-	Libre
Octane	Propietario	Basado fisicamente, transporte espectral de luz	Render imparcial basado en GPUs	Licencia €99, Licencia 3 paquetes €279, Licencia 5 paquetes €459, Licencia 10 paquetes €899
RenderMan	Propietario	Algoritmo reyes y trazado de rayos e iluminación global integrada en RenderMan Pro Server	Proceso de render mediante red distribuida	RenderMan for Maya 4.0 $4,975.00 (5), RenderMan Pro Server $2,000 por licencia y RenderMan Studio 3.0 $ 3,500
SunFlow	Open source	Iluminación global	Desarrollado en java	Libre

Render	Licencia	Algoritmo	Arquitectura	Precio
V-Ray	Propietario	Iluminación global como path tracing, photon mapping, irradiance maps	Multiprocesador y corre sobre GPUs en su version V-Ray RT	V-Ray 2.0 para 3ds Max €70
YafaRay	Open source	Ray tracing	Estructura modular	Libre

Tabla 4.2: Software auxiliar para el proceso de render de escenas (precios dados en dólar y euro).

4.2.1. Motores del proceso de render mediante el trazado de rayos

Desde que Whitted introdujo el algoritmo de trazado de rayos como una técnica de generación de imágenes de alta calidad, se señaló que el trazado de diferentes rayos son procesos independientes y por lo tanto, se pueden ejecutar concurrentemente; existen distintos niveles de granularidad de paralelización del algoritmo para el trazado de rayos[32]: el basado en frame, el basado en imagen y el basada en rayo. En el basado en frame, cada proceso es responsable de generar la imagen completa; por otro lado en el basada en imagen, todos los procesos cooperan para la generación total de una imagen; el basada en rayo tiene una granularidad fina, las estaciones de trabajo se cooperan para generar uno o más rayos y sólo se considera cuando se tiene un canal de comunicación muy bueno.

Así es que, al seleccionar la granularidad a utilizar es indispensable considerar las cuestiones correspondientes a la comunicación entre los procesos y la arquitectura que se tenga, ya que las granularidades muy finas comúnmente exigen muchas transferencias de información en el canal de comunicación.

Uno de los primeros ambientes distribuidos para renderizar escenas mediante el algoritmo de trazado de rayos fué ASTRA [32], utiliza un canal de comunicación mediante RPCs (de las siglas en inglés *Remote Procedure Call*) dentro una LAN (de las siglas en inglés *Local area network*). Se una distribución de carga mediante demanda, entre los principales limitaciones que tiene son que gastan demasiado tiempo de inicialización de la estructura de aceleración del rayo y el canal de comunicación es muy lento. Como objetivo principal tenía el de mantener el laboratorio de cómputo trabajando durante la noche para realizar el proceso de de render sin incomodar a nadie.

Después se fueron desarrollando otros sistemas de render cada vez más rápidos y sofisticados, en donde la parte central es el motor de render, es decir, la implementación del algoritmo de trazado de rayos. A lo largo del tiempo se han ido desarrollando

varios motores de render, específicamente, para fines de esta tesis se mencionan ejemplos de trazado de rayos interactivos, debido a que se adoptaron para crear *render farms* cada vez más rápidos, sofisticados y de alta calidad.

StarRay Fué desarrollado en 2002, es considerado el primer sistema de trazado de rayos interactivos; se uso para visualizar diferentes tipos de datos y se diseño para correr sobre memoria compartida grande en supercomputadoras, siendo posible escalar casi linearmente hasta 1024 procesadores en una sola configuración. A pesar de que se ha aplicado a una serie de problemas de visualización diferentes, emplea un modelo de objeto y define una interface de programación extendible. Su desarrollo precedió a la amplia adopción de paquetes de rayo [33].

OpenRT Desarrollado en el 2002 [19], su diseño es muy similar al de OpenGL, pero usa fotogramas o semántica en modo retenido en vez del modo inmediato de OpenGL; soporta algunos comandos del núcleo de OpenGL y los extiende cuando se necesita mayor soporte para el motor de trazado de rayos. El sistema está dirigido para arquitecturas en *clúster*. Los sistemas basados en OpenRT disponen de *shaders* extensibles y han desplegado con éxito una variedad de capacidades industriales de visualización.

PBRT Los autores del proceso de render basado fisicamente, lanzarón una implementación basada en su libro [34] y lo llamaron PBRT; en él introducen conceptos teóricos de proceso de render fotorealista y van de la mano con el código fuente para crear un render sofisticado. Es un buen libro para comprender, diseñar y construir sistemas para representar imágenesfisicamente realistas.

Razor Desarrollado en el 2006 [35], introduce la estructura de aceleración *KD-tree*, construyendo a demanda cada frame; lo cual reduce el costo de soportar escenas dinámicas mejorando el acceso a los datos y los patrones de cómputo para los rayos secundarios. La arquitectura desacopla los cálculos de sombreado de los cálculos de visibilidad utilizando un esquema de sombreado de dos fases; incorpora la agrupación de rayos en paquetes SIMD (de las siglas en inglés *Single Instruction, Multiple Data*) para un procesamiento y acceso a datos eficiente. Además de que agrega la capacidad de multiresolusión al sistema con el inconveniente de que implica una estructura de aceleración más complicada y a su vez más lenta.

Manta Desarrollado en 2006 [33], es un sistema interactivo para el trazado de rayos, diseñado para estaciones de trabajo ó *clúster* de computadoras y super computadoras. Manta logra un rendimiento notable en varias configuraciones a través del empleo de un modelo *pipeline*. Éste modelo divide las tareas basándose en sus características, se usan paquetes de rayo para tomar ventaja de casos especiales de optimización, segmentación de software, instrucciones SIMD y coherencia

de rayo para optimizar el código. Usan transacciones de software para mantener los cambios de estado en un ambiente *multithreading* en vez de un estado *multi-buffer*.

NVIDIA® Optix™ desarrollado en el 2009 [36, 2], es un sistema programable diseñado para trabajar con GPU's (de las siglas en inglés *Graphics Processing Unit*). Combina un *pipeline* con el proceso de trazado de rayos programable. Cuenta con una representación de escena liviana. Compone una compilación de dominio específico con un conjunto flexible de controles sobre una escena jerárquica, creación de la estructura de aceleración y transversal, una actualización de escena sobre la marcha y un balance de cargas dinámico sobre el modelo de ejecución basado en GPU. Tiene 7 diferentes tipos de programas, los cuales operan sobre un rayo a la vez: generación del rayo, intersección, cuadro delimitador, impacto más cercano, cualquier impacto, fracaso, excepción y selección de visita. El motor consiste de dos APIs distintas una para el lado de *host* y otra para el lado del *device*; además usa un balance de cargas de grano fino dinámico.

Algunos *render farm* que han existido han adoptado como motor de proceso de render, alguno de los detallados anteriormente, con la finalidad de adoptar el algoritmo de trazado de rayos.

4.3. Trabajo relacionado sobre *render farms*

En la actualidad existen programas que nos ayudan a la gestión de *render farms*, tanto comerciales como libres, entre los comerciales más comunes encontramos a Qube! de PipelineFX, Enfuzion de Axceleon, LSF de Platform, Muster de Virtual Vertex, Deadline de Frantic Fil, los cuales soportan casi todos los software de modelado mostrados en la tabla 4.1 y proceso de render como maya, 3ds max, mental ray, mostrados en la tabla 4.2. En la tabla 4.3 se muestran algunos detalles de éstos.

Generalmente, el tiempo de proceso de render se reduce utilizando una mayor cantidad de computadoras dedicadas al proceso; sin embargo, la eficiencia de proceso de render disminuye cuando las computadoras sobrepasan cierta escala. Es por esto que se necesita un método de planificación para el proceso de render.

Para administrar *render farms* muy grandes, se introduce un administrador de colas que automáticamente distribuye el proceso a varios procesadores. Cada proceso puede renderizar algunas imágenes, una imagen completa, ó inclusive porciones de ésta. El software tiene típicamente una arquitectura cliente- servidor que facilita la comunicación ente los procesos y el administrador de colas; incluso existen algunos administradores que no manejan una cola central.

Algunas propiedades de los administradores, son el uso de las colas prioritarias, de licencias de software y algoritmos para optimizar a través de varios tipos de hardware dentro del *render farm*. A continuación se dan unos ejemplos de administradores de

render farm que se han ido desarrollando a lo largo del tiempo dentro del area de investigación.

4.3.1. Diseño e implementación de un gestor de *render farm* basado en OpenPBS

En [37] se propone un gestor de proceso de render uniforme, el cual se compone de un portal web y un administrador de *render farm* basado en OpenPBS. OpenPBS es un paquete de gestión de sistemas de cómputo y de trabajo por lotes; el administrador es un middleware entre el portal y OpenPBS y cuenta con funciones como dividir, monitorear, re-planificar, detener, eliminar, volver a correr. Las tareas se distribuyen dentro del clúster(compuesto por CPUs) mediante una política de planificación FIFO.

Render farm	Propietario	Tipo	Motores de integración	Arquitectura	Precio
Arion	Random Control	Administrador	Producción del motor de render fisicamente basado	Enfoque GPU+CPU	€595
Qube!	PipelineFX	Web	Todos los archivos de escena referenciados por Maya	distribuido y multithreading, considera la prioridad, número de CPUs y el número de hilos	$ 2,00 por dia
Enfuzion	Axceleon	Administrador	3ds Max, Maya, Softimage XSI, RealFlow 2012, Combustion, Blender , After EffectsCINEMA 4D, Nuke, LightWave, Houdini, etc	Sistema basado en cliente- servidor, distribuido, multicore, considera la prioridad	—
Muster	Virtual Vertex	Administrador	Maya, 3D Studio Max, Mental Ray, Lightwave and XSI	Distribuido con granularidad basada en imagen	Licencia para nodos ilimitados $2,499. Licencia para un cliente $85
Deadline	Thinkbox	Administrador	3ds Max, After Effects, Blender, Cinema 4D, Composite, Fusion, Generation, Houdini, Lightwave, Maya, Modo, Nuke, RealFlow, Rhino, Softimage, and Vue	Descarta la necesidad de un administrator central mediate el uso sompartido de archivos, monitor mediante interfaz, uso de prioridades	Licencia- Libre (2 esclavos), 2 Estaciones de trabajo y 10 licencias de render $5395.00, Por esclavo $185.00

Render farm	Propietario	Tipo	Motores de integración	Arquitectura	Precio
Nuke	The foundry	Administrador y web	Motor de render basado en scanline, integration con RenderMan	*multi-threading*, herramienta de modelado basada en imagen, resolusión independiente	**Administrador** Por nodo-bloqueado $2,900, Por correr tareas de fondo $450 **Web** Por nodo-bloqueado $1,600.00, Por correr tareas de fondo $110
Drqueue	open source	Administrador	3Delight, 3DSMax, After Effects, Aqsis, Blender, BMRT, Cinema 4D, Lightwave, Luxrender, Mantra, Maya, Mental Ray, Nuke, Pixie, Shake, Terragen, Turtle, V-Ray and XSI	Proceso de render distribuido, administración de servicios mediante colas y monitoreo	Libre

Tabla 4.3: Render farms (Precios en dólar).

4.3.2. Render farm basado en una red de computadoras de escritorio

En [38, 39], proponen la construcción de un *render farm* basado en un grid compuesto por computadoras de escritorio y una infraestructura P2P, donde el objetivo principal es utilizar el procesamiento de CPU de las máquinas que se encuentren ociosas y en el momento en que alguien la ocupe, suspender el procesamiento. Usan Condor como motor de render y blender, enlazados mediante phyton. Dentro de sus funcionalidades tiene: enviar trabajo, registrar trabajo, partición de la animación, render por malla, monitor de trabajos, entrega de resultados y modificación de resultados.

4.3.3. Render farm distribuido para producción de animación

Dentro del trabajo [40] se presenta el diseño de un *render farm* llamado DRFarm en un ambiente distribuido y multicore conectado mediante la infraestructura ethernet. El servidor administra los almacenamientos y las fuentes a renderizar y provee servicios a usuarios locales mediante una interface de servicio de render. Se introduce un método de subdivisión de tareas por jerarquía lo cual asegura una fusión y división de tareas flexible. Agrupa las tareas asignándolas dinámicamente en diferentes nodos, reduciendo los tiempos de proceso de render. En este *render farm* hacen uso de un balanceador de cargas dinámico, diseñando algoritmos orientados a la realización y utilización del cliente, llamados modo activo y pasivo.

En el modo activo el servidor agrupa y distribuye las tareas a renderizar mediante la capacidad del render. Si un cliente termina todas sus tareas asignadas, el buscará la cola para encontrar tareas sin finalizar asignadas a otros clientes. En el modo pasivo, todas las tareas son introducidas en la cola de tareas. El proceso de render del cliente obtiene las tareas cada vez que su estado esta inactivo. El modo pasivo no garantiza la finalización del trabajo como granularidad gruesa.

4.3.4. Investigación y diseño para le servicio de sistema de gestión el *render* farm Deadline

En [41], se diseña un sistema para la administración de servicios que soporten la administración de cuentas de usuario y cuotas de render, basados en el análisis de funciones y características del *render farm* Deadline. Se basan en el consumo de recursos como el tiempo de procesador, consumo de RAM, consumo de ancho de banda de la red, prioridad y velocidad del proceso de render, entre otros, para realizar una formula que determine la cuota total por trabajo de render.

4.3.5. Sistema adaptativo en clúster basado en una arquitectura navegador/servidor

En [42] se presenta un sistema robusto para el proceso de render en clústers, el cual esta diseñado con un sistema de balance de cargas con la arquitectura B/S. Su clúster esta constituido por un nodo gestor, un nodo de proceso de render y otro nodo de almacenamiento. Fue desarrollado basándose en la plataforma Drqueue. Existen dos roles para clientes remotos, los clientes que suben sus tareas de proceso de render y los usuarios que tomarán las tareas de proceso de render llamados nodos de proceso de render. Las computadoras de los clientes que serán nodos de proceso de render tienen que añadirse a la lista de almacenado en el nodo gestor, usa una planificación FIFO con prioridades. Los detalles de los nodos de proceso de render incluyen tipo de CPU, tamaño de RAM y el tipo de sistema operativo y para manejar el balance de carga el nodo gestor debe calcular la capacidad de carga de cada nodo de render que se ejecuta en el cluster, considerando la velocidad del CPU, tamaño en RAM, número de hilos, etc..

4.3.6. Diseño e implementación del sistema de gestión de tareas basada en el control por retroalimentación

Otro sistema gestor de tarea para el proceso de render es presentado en [43], en éste sistema crean una plataforma de producción distribuida y colaborativa, en donde los usuarios crean y ayudan a crear contenido digital mediante el uso de la Web. Para lo cual, se diseño un sistema administrador de trabajos mediante un mecanismo de retroalimentación basado en ELM (de las siglas en inglés *Extreme Learning Machine*) para optimizarse a si misma y es capaz de recuperarse automáticamente al fallo de errores. El sistema esta compuesto por un navegador, un portal, un gestor de aplicación (encargado de descomponer la tarea en un conjunto de subtareas para que cada una de éstas sean ejecutadas por un nodo de ejecución) y un sistema gestor de tareas el cual de acuerdo con la estrategia de plantificación es distribuido en los nodos apropiados, que a su vez esta conformado por un control de trabajos y el monitoreo del estado del trabajo.

Para realizar bien el algoritmo de planificación necesita conocer el tiempo de ejecución de cada tarea en cada máquina, pero éste dato depende de muchas cosas, del tamaño de la escena, del tamaño de la imagen, etc. Como consecuencia se crea un algoritmo de estimación de tiempo de ejecución para optimizar el rendimiento del sistema basado en una red neuronal en donde las capacidades de la máquina se puede caracterizar mediante el uso de vectores de referencia. Su arquitectura contiene los siguientes componentes.

Navegador del cliente Es el medio por el cual el cliente envía sus trabajos, incluyendo materiales y descripción del trabajo.

Portal web Acepta la descripción del trabajo y de los materiales enviados por el

cliente, el trabajo de materiales se envia al sistema de almacenamiento y la descripción del trabajo se carga a la aplicación del gestor.

Aplicación del gestor Descompone los trabajos en un subconjunto de trabajos de acuerdo al tipo de trabajo. Un trabajo se descompone en subconjuntos de trabajos hasta que cada subtrabajo se pueda realizar mediante una sola ejecución.

Sistema de gestor de trabajos De acuerdo a la estrategia de planificación, el trabajo se distribuye a un nodo apropiado, provee el control del trabajo, el monitoreo del estado del trabajo, persistencia de la información relacionada el trabajo dentro de la base de datos. Este componente a su vez contiene módulos, tales como:

- Gestor del estado de trabajo: Mantiene cada estado de cada subtrabajo y lo guarda dentro de la base de datos, el estado del subtrabajo incluye enviado, mapeado despachado y excepción.
- Planificación del trabajo: Periodicamente checa el estado de cada subtrabajo y ejecuta el método asociado a el. El estado saltará a diferentes tareas de acuerdo al si el método fué exitoso. Es posible que si el método no este asociado con el estado, en ese caso el estado avanza a una acción no asociada.
- Mapeo de los recursos: Mantiene la simulación de cada nodo para todos los nodos de ejecución. La simulación del nodo representan los estados concurrentes correspondientes a cada nodo de ejecución, configuración, tiempo linsto, subtrabajos asignados y trabajos en ejecución.

Nodos de ejecución: De acuerdo a la descripción del trabajo, toma el trabajo de los materiales del sistema de almacenamiento, completa el trabajo y reporta los estados de información de cada subtrabajo durante el proceso, por ejemplo, no empezado, corriendo, completado, etc.

4.3.7. Administración de tareas para cargas de trabajo irregulares basadas en GPU

En [44] se exploran los mecanismos de software para la gestión de tareas irregulares en GPU. Propone un sistema gestor de tareas; se analizan diferentes esquemas de balance de cargas para GPU y sus efectos en distintos tipos de carga de trabajo, se experimenta con colas de tareas única y monolítica y con colas distribuidas con tareas de donación y robo. De las cuales se demuestra que la de donación es mejor debido a que se tiene menos sobrecarga de memoria. Los resultados se validan mediante la implementación del algoritmo de Reyes con una división irregular dado que la carga de trabajo es capaz de lograr imágenes por segundo en una sola GPU.

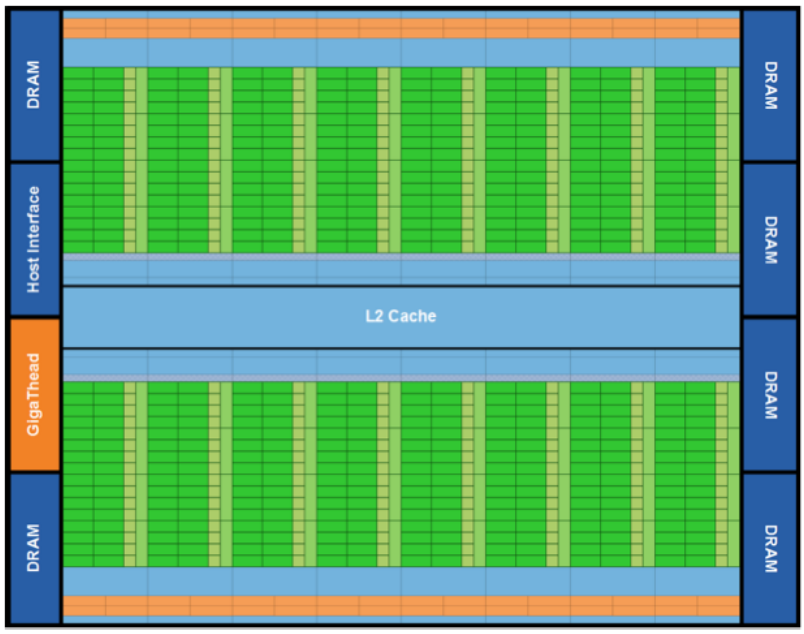


Figura 4.1: Arquitectura Fermi.

4.4. Unidad de procesamiento gráfico

GPU (de las siglas en inglés *graphics processing unit*), es un circuito electrónico diseñado para manipular y alternar rapidamente memoria para acelerar la construcción de imágenes dentro de un buffer, con la intención de ser visualidadas por medio de una computadora. Son utilizados en sistemas embebidos como teléfonos móviles, computadoras personales estaciones de trabajo y consolas de juegos.

Pero los GPUs modernos no son sólo potentes motores gráficos, si no también procesadores programables altamente paralelos con aritmética pico y ancho de banda de memoria que supera sustancialmente su contraparte CPU [45].

El rápido aumento de la GPU, tanto en programación como en capacidad, ha llamado la atención de la comunidad de investigación y se han logrado mapear con éxito una amplia gama de problemas complejos al GPU. A esté enfoque se le conoce como cómputo sobre GPUs. Los GPUs fueron diseñados para trabajar en aplicaciones particulares con las siguientes características:

- Requerimiento computacional muy alto, por ejemplo el proceso de render en tiempo real requiere de billones de píxeles por segundo, donde cada píxel requiere de una gran cantidad de procesamiento.

- El paralelismo es sustancial, afortunadamente, el *pipeline* de gráficos esta adecuadamente paralelizado, operaciones sobre los vértices y fragmentos son adaptados para procesar unidades paralelas de grano fino.

- La latencia es muy importante, es la medida del retardo experimentado en el sistema y por ningún motivo ésta debe ser mayor al rendimiento del sistema.

Desde que surgió el GPU, ha ido evolucionando convirtiéndose en un potente procesador programable con interfaz de programación de aplicaciones (API) y hardware cada vez más centrada en los aspectos programables de la GPU. El resultado es un procesador con una enorme capacidad aritmética y memoria de transmisión de ancho de banda, ambos mayores que los CPUs de la más alta gama.

Las tarjetas NVIDIA soportan un API extendida del lenguaje de programación C llamado CUDA (de las siglas en inglés *Compute Unified Device Architecture*) y OpenCL. Estas tecnologías permiten que funciones específicas de un lenguaje normal en C corran sobre los procesadores de flujo del GPU.

En general, la entrada del GPU es una lista de primitivas geométricas tipicamente triángulos en un sistema de coordenadas en 3D, a través de varios pasos, esas primitivas son sombreadas y mapeadas en una pantalla, donde son armadas para crear una imagen final.

En 4.1 se puede observar la arquitectura Fermi basada en GPU, implementada con 3.9 billones de transistores, 512 núcleos CUDA. Un núcleo CUDA puede ejecutar instrucciones de punto flotante y enteros por reloj para un hilo, los núcleos están organizados en 16 SMs de 32 núcleos cada uno. El GPU tiene 6 particiones de memoria de 64-bits, para una memoria interface de 384-bit, soporta un total de 6 GB de memoria GDDR5 DRAM. El planificador global GigaThread distribuye los bloques de hilo a los hilos planificadores SM [46].

4.5. *Render Farm* de Pixar

Figura 4.2: Foto del *render farm* de Pixar [1].

En la figura 4.2, se muestra una foto del *render farm* de los estudios de producción Pixar, el plantel central de dato cuenta con 13,500 ft^2, el cual alberga al *render farm*, servidores de archivos y sistemas de almacenamiento. La instalación incluye más de 3,000 procesadores AMD, herramientas propietarias para la administración del render,

permite la adición de estaciones de trabajo de escritorio al *render farm* durante horas, expandiendo aún más su capacidad a más de 5,000 procesadores.

Dentro de las características con las que cuenta el *render farm* se tiene: usan fibra óptica conectada a cada computadora de los artistas gráficos, permitiendo entregar imágenes en alta resolución en un lapso de tiempo corto; cuenta con almacenamiento de datos de más de 100 *terabytes*; sistemas para la edición de imagen y sonido, administración y corrección de color y composición de alta velocidad; un cuarto destinado al control de multimedia de entrada, salida y conversión y duplicación de formatos; entre otros.

En un video Jay Weiland, gerente de construcción de IT, comenta acerca del precio energético que conlleva el proceso de render de las películas dentro de Pixar y da un panorama de un proyecto en el cual Pixar usa un sistema de contención de pasillos frios de *Polargy* para abordar el aumento de las cargas de calor en su *render farm*. Expone que en uno de los cuartos del *render farm* de 1,600 ft^2 se usa una carga de energia de 335 kilowatts, resultando una densidad más de 200 watts por ft^2. Al final del proyecto, logran un ahorro de energia de 306,600 kWh. Aunque si consideramos que esta cantidad de energia se consume para sólo una octava parte del plantel del *render farm*, entonces el total del consumo energético debe ser bastante algo.

Por otra parte, hablando de la eficiencia energética de los GPUs, podemos encontrar un gran beneficio energético. Comparado con versiones secuenciales y paralelas, los versiones programadas sobre GPU consumen más potencia, pero la diferencia real es el tiempo de procesamiento; es decir, aunque el consumo de potencia sea alto, los lapsos de tiempo son muy cortos reduciendo de ésta forma el consumo energético. Existen algunos artículos en donde realizan experimentos comparativos con relación al consumo energético [47, 48].

Dicho lo anterior, un *render farm* basado en GPUs debe ser más eficiente tanto en rendimiento como en consumo energético.

4.6. Conclusiones

Del estudio del estado del arte de los sistemas de *render farms* y del software existente tanto para el modelado de escenas y para el software auxiliar en el proceso de render se concluye:

1. Existen muchos sistemas de software para el modelado de escenas que auxilian al diseñador gráfico en la tarea de modelar las escenas, sin embargo para poder realizar el proceso de render de una animación se requiere de un software especial que en varios casos esta muy ligado al software utilizado en el modelado de la escena.

2. Se han creado varios sistemas para el proceso de render con múltiples tipos de algoritmos, dentro de los cuales destaca el algoritmo para el trazado de rayos, gracias a que nos permite crear imágenes con mayor calidad y de alta definición.

Sin embargo el algoritmo consume mucho tiempo de procesamiento de CPU y para realizar el render de todo un conjunto de imágenes se requiere del uso de un *render farm* especializado.

3. Los *render farm* existentes, en sus mayoría de uso comercial, utilizan motores de render comerciales y aún no explotan las ventajas que trajo consigo la aparición de la GPU. Además la renta de estos *render farm* son muy caros, y normalmente no se tiene un resultado satisfactorio por lo que necesitan volver a utilizar el *render farm* y lo hace aún más caro. Incluso, en el caso de los gestores de *render farm* que venden para crear uno propio, requieren de licencias normalmente controladas por el número de CPUs ó nodos a utilizar.

4. De los gestores creados en el ámbito de investigación se concluye que normalmente se componen por un módulo encargado de recibir los trabajos de render ya sea mediante una interfaz web o mediante un *script*; un módulo encargado de distribuir la carga de trabajo entre los nodos en los cuales se realizará el proceso de render; el proceso de render en sí, en donde usa un algoritmo específico o hacen uso de un sistema de render como blender; finalmente un módulo encargado de notificar al cliente del resultado de sus trabajos. Adicionalmente varios de éstos sistemas integran un monitor para visualizar el estado del *render farm*.

Capítulo 5

Desarrollo

En éste capítulo se detallan cada uno de los módulos, y comunicaciones que intervienen en el *render farm* RentiX, diseñado a partir de la revisión del estado del arte. En la figura 5.1 se muestra la arquitectura general del sistema en donde se consideran básicamente 3 módulos dentro del *render farm*:

- Portal web: Es el medio mediante el cual se envían los trabajos al *render farm*.
- Sistema gestor: Es el encargado de distribuir la carga de trabajo entre los GPUs, considerando las propiedades tanto del servidor como de los GPUs que conforman al sistema, la memoria consumida y el tiempo de render de cada una de las escenas,; para mantener un control adecuado en el balance de cargas.
- Sistema de render: Es el encargado del proceso de render, corre tanto en el CPU del servidor (host) como en el GPU (device). Recibe como parámetro de entrada un archivo OBJ indispensable para su funcionamiento, éste archivo es el descriptor de la escena y comúnmente esta acompañado de un archivo MTL [49] (el cual es el archivo de bibliotecas de los materiales y describe las propiedades de los materiales a emplear en los objetos), adicionalmente pueden acompañarlos las imágenes que definen las texturas de los objetos dentro de la escena.

5.1. Propuesta de la tesis

El objetivo principal de éste proyecto de tesis es realizar un sistema encargado de gestionar las tareas de proceso de render dentro de un *render farm* conformado por GPUs. El algoritmo de sintetizado de imagen es el trazado de rayos (mejor conocido dentro de la literatura como *ray tracing*) considerando al motor de proceso de render NVIDIA®OptiX™como base fundamental para el proceso de render; el propósito final es reducir, en medida de lo posible, tanto la distribución de trabajo en el *render farm*, como los tiempos de proceso de render; atendiendo las peticiones de los clientes a través de una interfaz web. Ver imagen 5.1.

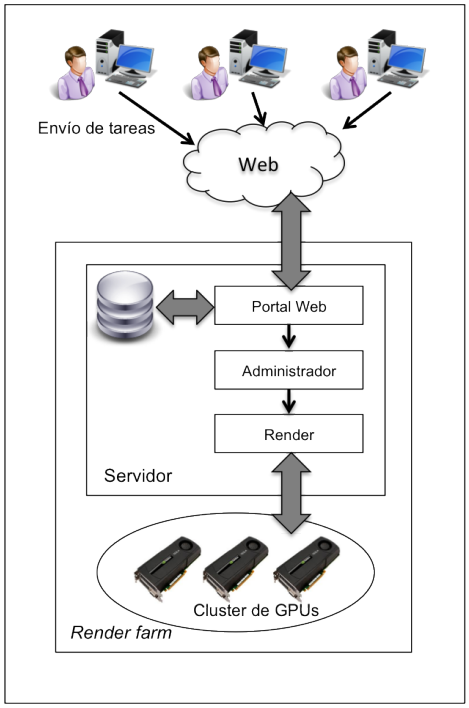


Figura 5.1: Arquitectura general del sistema.

Para la distribución del trabajo se crea un módulo destinado a la gestión de tareas del proceso de render, (considerando algunas de las características de los balanceadores de carga existentes) encargado de nivelar la utilización de los GPUs conectados al *render farm*, administrando de la manera más eficiente la demanda de los trabajos a renderizar, para lo cual se considerarán las propiedades de las tarjetas que se encuentren conectadas al *render farm* y los posibles inconvenientes que pudieran resultar. Es importante destacar que la granularidad de fragmentación es del tamaño del fotograma; es decir, el archivo será enviado en su totalidad al GPU y por tanto, el nivel de paralelización es considerado de granularidad gruesa. Dentro del motor OptiX realiza una paralelización de granularidad fina a nivel del GPU, reduciendo aún más los tiempos del proceso de render mediante el algoritmo de trazado de rayos.

Se brinda el servicio de proceso de render a través de una interfaz web, notificando al cliente vía correo electrónico cuando su trabajo haya sido finalizado, junto con un enlace en donde los clientes podrán descargar las escenas generadas; el tiempo para obtener sus imágenes estará dado por al ancho de banda de la red a la que se encuentre conectado. Cabe señalar que todos los trabajos enviados, sus características iniciales y las características del proceso de render se almacenan en una base de datos.

5.1.1. Mapeo de los sitemas multiprocesador al caso de estudio

Se tiene como principal objetivo proveer un esquema de balance de cargas distribuido y de bajo costo a través de los GPUs, y la planificación de las actividades tiene una influencia significativa en el rendimiento del sistema, ya que algunos GPUs pueden resultar ociosos mientras otros por el contrario pueden contar con una carga de trabajo alta.

Anteriormente en los sistemas paralelos con procesadores iguales y memoria compartida, compartían una sola cola global, de la cual se extraía los procesos y se les designaba un tiempo de computo definido de tal forma, que todos los procesadores siempre se mantenían activos. Sin embargo en sistemas como el nuestro, aún y cuando tenemos la posibilidad de tener memoria compartida, tenemos la desventaja de que no todos los GPUs son iguales y por lo tanto hay tareas que se pueden ejecutar mejor en un GPU que en otro. Practicamente dentro de los datos que se consideran son la memoria utilizada y el tiempo de procesamiento.

Para nuestro caso de estudio y realizando un mapeo de los sistemas con multiprocesadores a un sistema multi-GPU se tienen consideradas las siguientes propiedades:

- Propiedades de los GPU, la arquitectura del *render farm* considera GPU heterogéneos, es decir no cuentan con la misma velocidad de procesamiento, ni con misma capacidad de memoria, inclusive algunas tareas requieren más memoria en un GPU que en otro.

- Propiedades de las tareas, en lo que respecta a las tareas, tampoco son iguales; las escenas cuentan con número diferente de vértices, diferentes propiedades de los materiales, diferente complejidad, resolusión e inclusive algunas requieren de mayor memoria que otras debido a que requieren de imágenes que usan como textura.

- Fragmentación de la tarea: Se cuantifica el trabajo en términos de tareas, todas las tareas no requieren la misma cantidad de tiempo para completarse y no se pueden descomponer en subtareas; una tarea se ejecuta solo en un solo GPU. Es decir, las tareas son independientes y por lo tanto el orden en que se ejecuten son irrelevantes. Para esta tesis se considera que la fragmentación de la tarea es de grano grueso, es decir por cada archivo OBJ corresponderá a una tarea y ésta tarea se ejecutará en el GPU hasta su finalización.

- Propiedades del servidor: En lo que respecta al servidor, se considera que se usa memoria compartida para la administración de la tareas.

- Trasferencias: En este tipo de sistemas la transferencia de información juega un papel muy importante; en el caso de la transferencia de pasos de mensajes no es mucha por la propiedad de la memoria compartida dentro del servidor; sin embargo la transferencia de información del servidor al GPU es muy alta y conlleva mucho tiempo de CPU.

- Los factores determinantes en la elección del GPU son: número de vértices, calculo del tamaño de memoria a utilizar y calculo del tiempo del proceso de render. Los últimos dos valores dependerán del número de vértices y de las imágenes que se utilicen como textura ya que también implican un costo tanto de memoria como de tiempo de transferencia de datos entre el servidor y el GPU. Además las tareas son independientes y se ejecutan de inicio a fin.

- Escalabilidad, se planea que sistema sea altamente escalable al añadir un número mayor de GPUs.

Una vez definidas las propiedades del GPU, se procede a hacer un estudio del tipo de arquitectura que debe tener el sistema para su mejor funcionamiento.

Se considera la opción de la cola global, encontrando las siguientes ventas y desventajas: El proceso designación de tareas de render es más rápido debido que, al igual que en la explicación de procesadores homogéneos, no se gasta tiempo de trasferencias de paso de mensajes entre colas, la carga de trabajo esta balanceado con base a las tareas atendidas y no existe el caso en que un procesador se encontrará ocioso mientras que otro por su contraparte se encontrará con una carga de trabajo alta; sin embargo, no todas las escenas pueden ser renderizadas por todos los GPU, es decir, si la escena demanda una capacidad de memoria superior a la que cuenta el GPU, entonces mandaría error; aún y cuando existiera un GPU con las capacidades suficientes para poder llevar a la tarea a un estado de éxito. Otra desventaja es que la capacidad de memoria del GPU no se aprovecha en un 100 %.

Es por esto que también se crea un balance de cargas dinámico, para poder realizar la asignación de acuerdo a la capacidad de memoria y nivelación del tiempo de procesamiento. Sin embargo, como desventaja a esto es que existe una contención de colas que hace imposible escalar a mayor número de GPUs el sistema.

Para solventar ésta problemática se considera la opción de las colas locales, encontrando las siguientes ventajas y desventajas: Las tareas son asignadas considerando tanto las propiedades de la misma, como de los GPU que se le asignaran; logrando así un mapeo adecuado entre las tareas y las GPUs.

5.2. *Render farm* RentiX

En la figura 5.2 se puede observar la secuencia que realiza el *render farm*; a continuación se detalla en términos generales el proceso del sistema. El primer paso es primordial, ya que es el medio por el cual los clientes envían los trabajos que desean renderizar al *render farm*, hay que puntualizar que un trabajo esta constituido por uno o más archivos descriptor de escena; éste proceso se realiza a través de un portal web. El portal web está montado sobre un servidor apache dentro del sistema; una vez que el trabajo se recibe directamente en el servidor, se almacena en la base de datos, se genera un archivo en donde se encuentran detalladas cada una de las escenas y se envía al módulo de recepción de trabajos.

La comunicación entre el servidor web y el módulo de recepción de trabajos, se realiza mediante la conexión a un socket tipo TCP/IP, dentro de éste módulo se descompone el trabajo en las n-tareas correspondientes al número de escenas y las envía a una cola de mensajes llamada conjunto de tareas, que a su vez va a ser leída por el modulo de admisión de tareas, el cual verificará el estado del servidor para ver si es posible mandar a ejecución la tarea o en el caso contrario, se tendrá que esperar a que los recursos del servidor se liberen, si la tarea se puede mandar a ejecución entonces se envia al despachador, el cual considerando las propiedades de la tarea y la utilización dentro del GPU designara en que device se ejecutara, enviado a la respectiva cola local del device las tareas para ser leída por el despachador local relacionado a cada nodo.

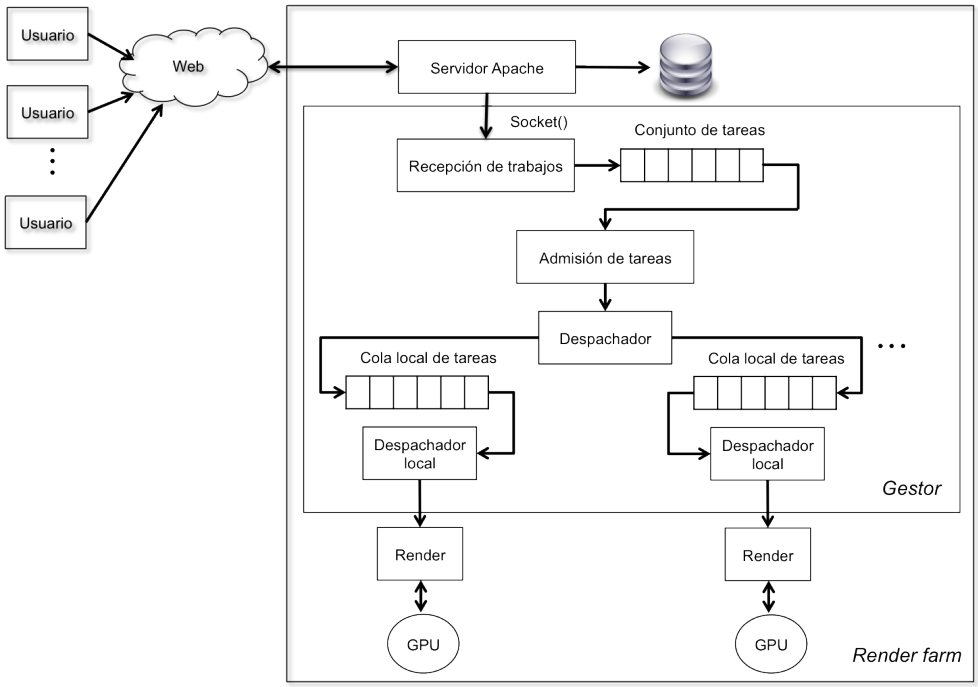


Figura 5.2: Diagrama a bloques del *render farm*.

Finalmente se ejecuta el proceso de *render* de cada tarea y al finalizar se manda al módulo de notificación, el cual a su vez cuando el trabajo haya sido completado notificará la cliente. A lo largo de este capítulo se explica detalladamente el desarrollo de cada uno de los módulos empleados.

5.2.1. Portal web y servidor web del *render farm*

Para poder definir bien los elementos que van a intervenir dentro de éste módulo es necesario mostrar el diagrama de casos correspondiente, en el cual queda delimitadas las funciones que intervienen en la interfaz web. En la figura 5.3 se encuentra el actor

que interviene dentro del sistema, el cual es básicamente el cliente. El cliente puede realizar las siguientes acciones: registro, envío de trabajos, descarga de resultados y consulta de trabajos. El servidor web es el encargado de brindar éstos servicios mediante la ayuda de una base de datos en donde almacena los datos de todos los clientes, los proyectos que éste haya creado y sus respectivas propiedades.

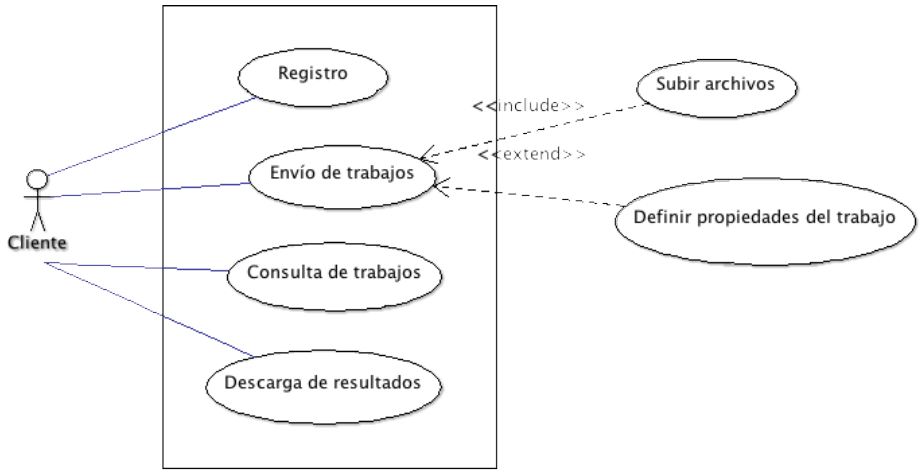


Figura 5.3: Diagrama de casos de usos del sistema.

Registro

En términos generales el cliente a través de su navegador web entra al portal web de Rentix; sin embargo para poder ingresar al sistema primero necesita registrarse. Para registrarse es necesario que el cliente proporcione un nombre de usuario, contraseña y correo electrónico. Una vez registrado se le permite iniciar sesión.

Envío de trabajos

Para poder realizar un trabajo de render primero es necesario crear un proyecto y definir las propiedades del trabajo. Las propiedades que puede tener el trabajo son las siguientes:

- Posición de la cámara.
- Posición de la luz.
- Resolución de las imágenes resultantes.
- Tipo de formato y nombre de las imágenes resultantes.

Un dato muy importante es que un trabajo esta compuesto por n-tareas, cada una de las cuales puede tener las propiedades descritas anteriormente y se le da la oportunidad al cliente de generalizarlas a nivel proyecto o específicar cada una de las tareas.

La forma en que se envía el trabajo es en una carpeta comprimida con el formato ZIP, el cual se encontrarán todas las escenas pertenecientes a éste proyecto que el usuario quiera renderizar, archivos de materiales e imágenes que puedan formar parte de las texturas del modelo. Se sube la carpeta al servidor, se descomprime y el servidor es el encargado de verificar el número de tareas contenidas dentro del trabajo, puesto que cada escena se encuentra descrita por un archivo OBJ, posteriormente se le despliegan las tareas dentro de una tabla junto con un combo para poder específicar las propiedades de cada una de éstas.

Dentro de las propiedades, el nombre de la imagen resultante se predefine similar al nombre del archivo descriptor de escena correspondiente. Una vez que se se tiene definidas las propiedades se reciben por el servidor web, crea un archivo en el cual se específican cada una de las propiedades de las escenas, se guarda y finalmente se envía la petición de trabajo de render con el directorio del archivo creado al módulo de recepción de trabajos mediante la conexión a un socket tipo TCP/IP cliente, en la figura 5.4 se muestran las interacciones de comunicación existentes entre el servidor web y el módulo de recepción de trabajos.

Consulta de trabajos y descarga de resultados

Para poder realizar la consulta de trabajos, dentro de la interfaz web existe un apartado en donde el cliente puede observar los trabajos que ha enviado y sus estados, los cuales son: enviado, aceptado, en cola de espera, en ejecución y finalizado. Si el proyecto ya se finalizó entonces aparecerá un enlace correspondiente a las imágenes resultantes.

5.2.2. Sistema gestor del *render farm*

Dentro de la arquitectura del *render farm* se consideran los siguientes módulos:

- Recepción de trabajos
- Admisión de tareas
- Despacho de tareas
- Despacho local de tareas
- Notificación

Para definir el esquema de programación de las tareas, se hace un estudio del comportamiento de la cola global y el esquema particionado[50], considerando varias alternativas de balance de cargas como lo es el método de donación y robo. En un

esquema particionado de colecciones de tareas concurrentes, cada tarea es asignada a un procesador y todos los trabajos generados por la tarea requieren ejecutarse en ese procesador.

En el caso del esquema particionado con colas locales, para el servicio de colas se utilizarán 2 diferentes tipos de colas: Una correspondiente al conjunto de tareas llamada cola de recepción y una cola local de tareas relacionada a cada GPU existente dentro del *render farm*.

Recepción de trabajos

Dentro de este módulo se utiliza un modelo de comunicación cliente-servidor en donde el servidor recibe mediante el uso de un *socket* la petición del trabajo de render junto con un archivo que contiene la información de cada una de las escenas. En la figura 5.4 se observa la secuencia de llamadas para la comunicación orientada a conexión entre el portal web y el módulo de recepción de trabajos.

Éste archivo contiene la siguiente información: nombre del descriptor de escena con su correspondiente información: nombre de la imagen resultante, resolución (las opines a elegir son alta definición y alta definición completa), formato de la imagen, posición de la cámara y posición de la luz. Posteriormente se crea una estructura correspondiente a cada una de las escenas a procesar, envía los datos al módulo de *admisión de tareas* mediante el uso de una cola de mensajes (mecanismo de comunicación entre procesos que brinda el sistema V de UNIX) llamada *conjunto de tareas* y finalmente devuelve la instrucción de recibido.

Éste módulo es de tipo concurrente ya que se encarga de recoger la petición de servicio y para atenderlo crea un hilo encargado de enviar las tareas a la cola de mensajes, brindando mayor rendimiento al sistema debido a que se pueden atender un número de peticiones a mayor velocidad.

Admisión de tareas

Este módulo lee la cola de mensajes de *conjunto de tareas*, obtiene las características de cada una de las escenas tomando como base el archivo descriptor de escena, y con respecto a éstas se encarga de calcular los tiempos de render y memoria utilizada; una vez que se extraen todas las propiedades de cada escena, se almacenan dentro de la estructura y se envía al *despachador de tareas*. Las características que se obtienen son:

- Resolución;
- Número de vértices;
- Número de normales;
- Número de coordenadas de textura ;
- Número de normales de cada faceta;

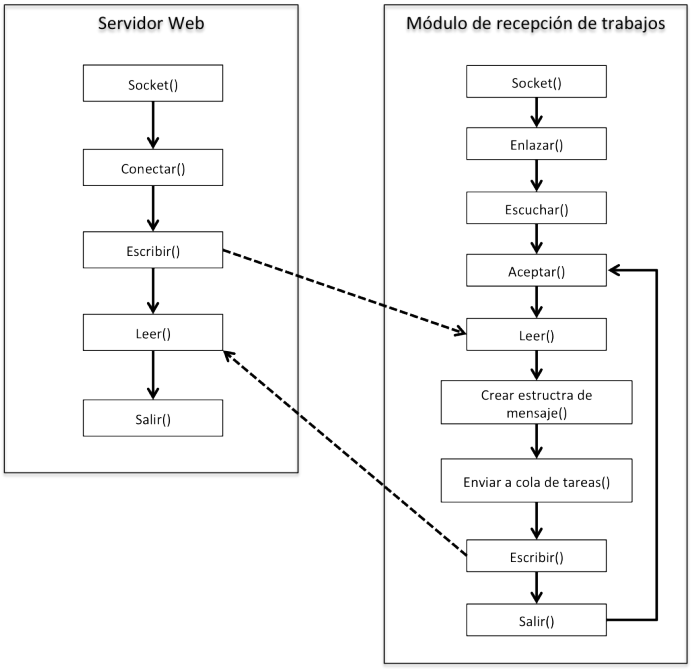


Figura 5.4: Secuencia de llamadas para la comunicación orientada a conexión entre el portal web y el módulo de recepción de trabajos.

- Número de triángulos;
- Número de materiales;
- Número de grupos;

A partir de éstas características, se realizara un estimado tanto de la capacidad de memoria requerida como de el tiempo de proceso de render para poder tener un balance de cargas óptimo en el sistema de *render farm*.

Despacho de tareas

Éste módulo es de vital importancia, en el se tiene como objetivo el balace de la carga de trabajo entre el número de nodos utilizando un algoritmo FIFO. Se considera las propiedades del servidor como tamaño en RAM disponible, utilización de CPU e inclusive considera las propiedades de cada device para conforme a eso designar a que device se va a ejecutar la tarea. Teniendo en cuenta las estimaciones realizadas previamente.

Una vez designado el device, en el caso de la cola global lo envia a ejecución y en el caso del esquema particionado de colas locales, se envia al módulo de despacho local de tareas perteneciente al device.

Despacho local de tareas

El despacho local de tareas lee la *cola local de tareas* correspondiente y obtiene la tarea más próxima para su ejecución, considerá las propiedades del GPU donde se ejecutará comparando con la memoria libre y si existen las condiciones adecuadas para el éxito de su ejecución la envía al proceso de render.

Notificación

El módulo de notificación se encarga de realizar la colección de las tareas para llevar un inventario de cada una de las tareas y su correspondiente estado. Los estados que puede tener una tarea es: aceptada, en cola de espera, en ejecución, finalizado o error. Al completarse el trabajo envía un mail al cliente notificando que su trabajo ha sido finalizado.

5.2.3. Proceso de render del *render farm*

Para la implementación del algoritmo ray tracing basado en GPU se utiliza el motor de trazado de rayos NVIDIA®OptiX ™, el cual utiliza CUDA para su implementación.

En si, OptiX no es un algoritmo de trado de rayos, más bien es un *framework* escalable para construir aplicaciones basadas en el trazado de rayos. Esta compuesto por dos partes: Un API basado en el host que define las estructuras de datos para el trazado de rayos y un sistema de programación basado en CUDA que produce nuevos rayos, intersecta rayos con las superficies y responde a estas intersecciones. Estas dos piezas proveen un soporte de bajo nivel para el trazado de rayos. Lo cual permite aplicaciones de usuario escritas que usan trazado de rayos para gráficos, detección de colisiones, propagación del sonido, determinación de visibilidad, etc.

En el núcleo de OptiX [51], tiene un modelo abstracto pero poderoso para trazar rayos, emplea un programa proporcionado por el usuario que controla la inicialización de los rayos con las superficies, las sombras, los materiales y la creación de nuevos rayos. Los rayos pueden tener cargas útiles específicadas por el usuario y un mecanismo de ejecución interna que administra todos os detalles de recursión de pila. OptiX también provee una función dinámica de despacho y un sofisticado mecanismo de herencia variable que permite escribir sistemas de trazado de rayo de manera genérica y compacta.

Además provee a los usuarios la habilidad de escribir sus propios programas para generar rayos y para definir el comportamiento cuando los rayos se intercepten con los objetos. El pipeline del trazado de rayos contiene muchos componentes programables, estos componentes son invocados por el GPU en puntos específicos durante la ejecución del algoritmo de trazado de rayos genéricos. Existen 8 tipos de programas;

- Generación del rayo: El punto de entrada del pipeline del trazado de rayos, invocado por el sistema en paralelo para cada píxel.

- Exception: El manejado de excepciones es invocado por condiciones como desborde de pila y otros errores
- Closest hit: Es llamado cuando un rayo trazado encuentra la intersección más cercana a un punto como para el sombreado de materiales
- Any hit: Se llama cuando el trazado de rayos encuentra nuevas intersecciones más cercanas al punto de intersección, como para el calculo de sombra
- Intersection: Implementa la prueba de la primitiva de intersección de rayo, invocada durante la traversal.
- Bounding Box: Procesa la primitiva de cuadro delimitador del espacio del mundo llamada cuando el sistema construye una nueva estructura de aceleración sobre la geometría.
- Miss: llamada cuando el trazado de rayos pierde toda la escena geométrica.
- Visit: Es llamada durante la transversal de un nodo selector para determinar si el hijo de un rayo atravesará.

El lenguaje de entrada de estos programas es PTX. El SDK de OptiX también provee un conjunto de clases envoltura y cabeceras para usar con el compilador nvcc que habilita el uso de CUDA como una forma apropiada de generar el PTX.

En la instalación de OptiX se cuenta con un SDK que nos muestra algunos ejemplos para utilizar diversos tipos de propiedades de los materiales, sombras, reflexiones y refracciones entre otros. Cuenta con la posibilidad de importar archivos de tipo *.obj e imágenes tipo *.ppm y *.hdr. Una vez, que se comenzó a trabajar con el *framework*, se realizarón varias modificaciones para adaptarlo.

- Se eliminó la posibilidad de porder ejecutar el motor en modo visual interactivo debido a que consume más procesamiento de CPU y solo nos interesa tenerlo como proceso de fondo.
- Se tuvo que modificar el *glm.cpp*, *glm.h*, debido a que dentro de la especificación de las propiedades de los materiales no contaba con la posibilidad de manejar las reflexiones y refracciones.
- Se modifico el archivo *ObjLoader.cpp* para poder incluir las modificaciones realizadas en el punto anterior.
- Se implementó el archivo *obj_material.cu* correspondiente al proceso de render por medio de trazado de rayos en el cual, dados los términos de la Tabla 5.2.3, se consideran los siguientes modelos :

  0. Modelo de iluminación con color constante. Ver ecuación 5.1.

$$color = Kd \tag{5.1}$$

1. Modelo de iluminación difuso usando *Lambertian shading*. El color incluye un término ambiental constante y un término de sombra difusa para cada fuente de luz. Ver ecuación 5.2.

$$color = K_a I_a + \sum_{1 \leq j \leq m} I_j K_d (\bar{N} * \bar{L}_j) \tag{5.2}$$

2. Modelo de iluminación difuso y especular usando *Lambertian shading* e interpretacion de *Blinn* del modelo de iluminación especular de Phong. El color incluye un término ambiental constante y un término de sombra difuso y especular para cada fuente de luz. Ver ecuación 5.3.

$$color = K_a I_a + \sum_{1 \leq j \leq m} I_j [K_d (\bar{N} * \bar{L}_j) + k_s (\bar{H} * \bar{H}_j)^{N_s}] \tag{5.3}$$

3. Modelo de iluminación difuso y especular con reflexión usando *Lambertian shading* e interpretación de *Blinn* del modelo de iluminación especular de Phong y un término de reflexión similar al del modelo de iluminación de Whitted. El color incluye un ambiente constante y un término de sombreado especular y difuso para cada fuente de luz. Ver ecuación 5.4, donde I_r = intensidad del mapa de reflexión+ trazado de rayos.

$$color = K_a I_a + \sum_{1 \leq j \leq m} I_j [K_d (\bar{N} * \bar{L}_j) + k_s (\bar{H} * \bar{H}_j)^{N_s}] + I_r \tag{5.4}$$

4. Modelo de iluminación difuso y especular para simular el vidrio usando el mismo modelo de iluminación que el 3. Cuando se usa un bajo disolvente (aproximadamente 0.1), los reflejos especulares desde las luces o de las reflexiones se vuelven imperceptibles. Ver ecuación 5.4.

5. Modelo de iluminación difuso y especular simula al modelo de iluminación 3, excepto que se introduce la reflexión debido al efecto fresnel dentro de la ecuación. La reflexión fresnel resulta de chocar con una superficie difusa en un angulo de perspectiva. Ver ecuación 5.5.

$$color = K_a I_a + \sum_{1 \leq j \leq m} I_j [K_d (\bar{N} * \bar{L}_j) + k_s (\bar{H} * \bar{H}_j)^{N_s} * F_r (\bar{L}_j * \bar{H}_j, K_s, N_s)] + F_r (\bar{N} * \bar{V}, K_s, N_s) I_r \tag{5.5}$$

6. Modelo de iluminación difuso y especular similar al usado por [12] que permite rayos de refracción a traves de las superficies. La cantidad de refracción esta basada en la densidad optica N_i. LA intensidad de la luz

que refracta es igual a 1.0 menos el valor de K_s y la luz resultante es filtrada por T_f, el filtro de transmisión, al atravesar el objeto. Ver ecuación 5.6.

$$color = K_a I_a + \sum_{1 \leq j \leq m} I_j [K_d(\bar{N} * \bar{L}_j) + k_s(\bar{H} * \bar{H}_j)^{N_s}] + I_r + (1.0 - K_s) T_f I_t \tag{5.6}$$

7. Modelo de iluminación similar al 6, excepto que la reflexión y transmisión debido al efecto Fresnel son introducidos en la ecuación. En un angulo de perspectiva, es reflectada más luz y menos luz es refractada a traves del objeto.Ver ecuación 5.4, donde Ir = intensidad del mapa de reflexión

$$color = K_a I_a + \sum_{1 \leq j \leq m} I_j [K_d(\bar{N} * \bar{L}_j) + k_s(\bar{H} * \bar{H}_j)^{N_s} * F_r(\bar{L}_j * \bar{H}_j, K_s, N_s)] + F_r(\bar{N} * \bar{V}, K_s, N_s) I_r + (1.0 - K_s) F_t(\bar{N} * \bar{V}, 1 - K_s, N_s) T_f I_t \tag{5.7}$$

8. Modelo de iluminación similar al modelo de iluminación 3 sin trazado de rayos. Ver ecuación 5.4, donde Ir = intensidad del mapa de reflexión
9. Modelo de iluminación similar al modelo de iluminación 4 sin trazado de rayos. Ver ecuación 5.4, donde Ir = intensidad del mapa de reflexión

- Debido a que el motor de OptiX solo tiene soporte para guardar las imágenes en el formato *.PPM, se utilizó OpenCV para dar soporte para los siguientes formatos de archivos:
  - Windows bitmaps - *.bmp, *.dib
  - JPEG files - *.jpeg, *.jpg, *.jpe
  - JPEG 2000 files - *.jp2
  - Portable Network Graphics - *.png
  - Portable image format - *.pbm, *.pgm, *.ppm
  - Sun rasters - *.sr, *.ras
  - TIFF files - *.tiff, *.tif
- Aún y cuando OptiX permite el manejo de diversos tipos de resolución, para fines de esta tesis, nos limitamos a guardar las imágenes en HFD (1920x1080) y HD(1280x720).
- Se utiliza una estructura de aceleración Bvh el cual se enfoca en la calidad sobre el rendimiento de la construcción y es comúnmente usado para geometrías no uniformes.

Término	Definición
Ft	Reflexión fresnel
Ft	Transmisión fresnel
Ia	Luz ambiental
I	Intensidad de luz
Ir	Intensidad de la dirección reflejante (mapa de reflexión y/o trazado de rayos)
It	Intensidad de la dirección de transmisión
Ka	Ambiente reflejante
Kd	Reflexión difusa
Ks	Reflexión especular
Tf	Filtro de transmisión
H	Vector bisector unitario entre L y V
L	Vector de luz unitario
N	Superficie normal unitaria
V	Vector de vista unitario

Tabla 5.1: Términos utilizados en el render.

Capítulo 6

Implementación, pruebas y resultados

Para empezar con éste capítulo, es necesario mencionar primero como se encuentra conformado el sistema del *render farm* para poder abstraer las propiedades generales de hardware y junto con los resultados arrojados en las pruebas de render, formar una base que nos permita poder realizar las estimaciones tanto de tiempo como de memoria (útiles en la administración de las tareas de render). Además se mencionan las técnicas de balance de cargas experimentadas en la tesis y finalmente tomando como referencia los problemas y circunstancias suscitadas a lo largo de los experimentos, se rescatan para poder realizar un balance de cargas óptimo para trabajar en sistema de *render farm* basada en GPUs.

6.1. Infraestructura del sistema

El sistema de *render farm* construido para el desarrollo de esta tesis, cuenta con las siguientes características tanto de hardware como de software.

6.1.1. Hardware

Para el desarrollo del sistema del *render farm* RentiX se cuenta con dos GPUs y un servidor con las siguientes caracteristicas de cómputo:

Servidor del sistema: Dual Xeon 6 cores con 25 TB en RAM y 900 GB de disco duro, esta destinado para albergar al servidor web, el sistema gestor de trabajos. Cuenta con 12 procesadores y cada uno de éstos cuenta con las características descritas en la tabla 6.1.

Tarjetas Nvidia: Se cuentan con dos tarjetas GPU Nvidia destinadas para correr el proceso de render, y es precisamente sobre éstas que se realizará la repartición de las tareas de render. En la tabla 6.2 se muestran sus propiedades.

Modelo	Intel(R) Xeon(R) CPU X5675 @ 3.07GHz
Número de procesadores	12
Velocidad	3066.866 MHz
Tamaño de cache	12288 KB
Núcleos por procesador	6

Tabla 6.1: Propiedades de cada procesador del servidor.

device 0	GeForce GTX 460
Versión CUDA del driver / Versión del tiempo de ejecución	5.0 / 4.2
Capacidad CUDA número de versión mayor / menor	2.1
Cantidad de memoria global	1024 MBytes (1073414144 bytes)
(7) Multiprocesadores x (48) Núcleos CUDA /MP	336 Núcleos CUDA
Frecuencia de reloj de GPU	1530 MHz (1.53 GHz)
device 1	Tesla C2070
Versión CUDA del driver / Versión del tiempo de ejecución	5.0 / 4.2
Capacidad CUDA número de versión mayor / menor	2.0
Cantidad de memoria global	5375 MBytes (5636554752 bytes)
(14) Multiprocesadores x (32) Núcleos CUDA/MP	448 CUDA Cores
Frecuencia de reloj de GPU	1147 MHz (1.15 GHz)

Tabla 6.2: Propiedades de las tarjetas NVIDIA.

6.1.2. Software

El software involucrado en el desarrollo del render farm, se puede clasificar entre el software a emplear en el servidor del sistema, el software empleado para el sistema gestor de tareas, el software a emplear durante el proceso de render y el sistema utilizado tanto en la interfaz web como en el servidor web empleado:

- Servidor del sistema: Sistema operative GNU/Linux, distribución CentOS release 5.8 (Final).
- Sistema gestor de tareas: Se utiliza el lenguaje C, utilizando pthreads para brindar al sistema de alto rendimiento y colas de mensaje, proporcionadas por el UNIX System V, como mecanismo de comunicación entre procesos.
- Render: Para realizar el proceso de render, el sistema se divide en dos, del lado del *host* y del lado el *device*. Para el proceso de render se utiliza el motor de trazado de rayos Nvidia®OptiX ™en su versión 3.0, el cual del lado del *host* (en donde se encarga de leer el modelo, declarar las variables, generar la estructura de aceleración, guardar la imagen resultante, entre otros) se realiza en C y C+, mientras que del lado del *device* (encargado de trazar los diversos tipos de rayos, y añadir los materiales al buffer resultante) se realiza mendiante CUDA en su versión 4.2.
- Portal y servidor web: Se utilizará el servidor apache y el lenguaje de programación java con una base de datos tipo HD2, hibernete para la persistencia de los datos y icefaces para el portal web.

6.2. Escenas de prueba

Para realizar las pruebas sobre el *render farm* se utilizan varias escenas con diferentes propiedades: diferentes números de vétices, con texturas, sin texturas y con diversos tipos de materiales. Las escenas a utilizar se detallan en la tabla 6.3; se específican las propiedades principales como: el número de vétices, el número de normales, el número de coordenadas de textura, el número de triángulos, el número de materiales, el número de grupos y la memoria que se necesita para cargar en modelo. En la figura 6.1 se pueden apreciar las escenas de prueba en relación con el número de vértices y triángulos que poseen cada una. Se observa que se incrementan de manera simultánea, por lo tanto se elige, a partir de éste momento, trabajar respecto al número de vértices de las escenas.

De las escenas de prueba; se tienen a bunny, dragon y monkey, que son escenas con un solo objeto y normalmente se utilizan como *benchmark* en los trabajos de investigación de render; las escenas de CornelBox son de carácter demostrativo de algunos materiales soportados por el sistema de render y las escenas de *Ben*, *Conference*, *Fairy forest* y *Sibernik* son escenas más complejas con mayor diversidad de materiales, texturas añadidas y por lo tanto mayor número de vétices.

Considerando en cuenta éstos datos se puede tener una idea vaga del tiempo necesario para realizar el proceso de render. Por ejemplo, considerando los modelos de la tabla, aquellos con un número bajo de vértices se tiene a los de *CornellBox* y *Monkey* y por su contraparte se tienen escenas como *Conference* y *Fairy forest* con un número de vértices mayor. Por consiguiente se deduce que en medida del aumento de complejidad del modelo, aumentará la memoria necesaria para almacenarlo, se conllevará más tiempo de cómputo para realizar la construcción de la estructura de aceleración de la escena y por lo tanto, más tiempo en el proceso de render.

Un dato importante es el número de materiales que conforman las propiedades de la escena, puesto que para aquellas escenas que cuenten con texturas, se necesita cargar las imágenes correspondientes en memoria del *host* y en la memoria del *device*, lo cual conlleva un tiempo de lectura y transferencia de imagen determinado por el tamaño y tipo del archivo. Para las escenas que necesiten materiales tipo vidrio ó tipo espejo, por ejemplo, se trazan un número de rayos mayor debido a las reflexiones y refracciones; por consiguiente, procesamiento del render se incrementará. Se puntualiza que como propiedad se tiene predefinido un número máximo de profundidad para los rayos de reflexión y refracción de 10; indudablemente el número de objetos y el tamaño de éstos será un factor determinante para el proceso del render.

Lamentablemente, todas estas consideraciones no se pueden generalizar debido a que no se puede saber exactamente la complejidad de la escena, hasta una vez que se haya generado la imagen, por lo tanto se intentan realizar estimaciones en este ámbito.

Nombre escena	Vétices	Normales	Texcoords	Triángulos	Materiales	Grupos	Memoria
CornellBox-Empty [1]	24	0	0	12	7	6	0.005348
CornellBox-Original [1]	72	0	0	36	9	8	0.007790
CornellBox-Mirrow [1]	72	0	0	36	9	8	0.007790
Monkey [2]	507	0	0	968	1	1	0.045933
CornellBox-Glossy [1]	578	561	632	1112	9	8	0.055317
CornellBox-Sphere [1]	1116	1101	1256	2188	9	8	0.102856
CornellBox-Water [1]	3666	3533	1262	7088	10	9	0.309475
Dragon [3]	23484	30508	0	47001	1	1	2.000351
Bunny [3]	34835	0	0	69666	1	1	2.791874
Ben [4]	41474	41474	39046	78029	20	19	3.455467
Sibernik [5]	43085	240393	240393	80131	17	15	5.066631
Fairy forest [4]	97124	97124	86448	174117	114	113	7.789444
Conference [1]	194399	0	188873	331179	36	41	14.120087

Tabla 6.3: Propiedades de las escenas de prueba.

[1]Modelo obtenido de *Computer Graphics Archive* [52].
[2]Modelo exportado del software *Blender*.
[3]Modelo obtenido de *The Stanford 3D Scanning Repository* [53].
[4]Modelo obtenido de *The Utah 3D Animation Repository* [54].
[5]Modelo obtenido de *Understanding the Efficiency of Ray Traversal on GPUs* [55].

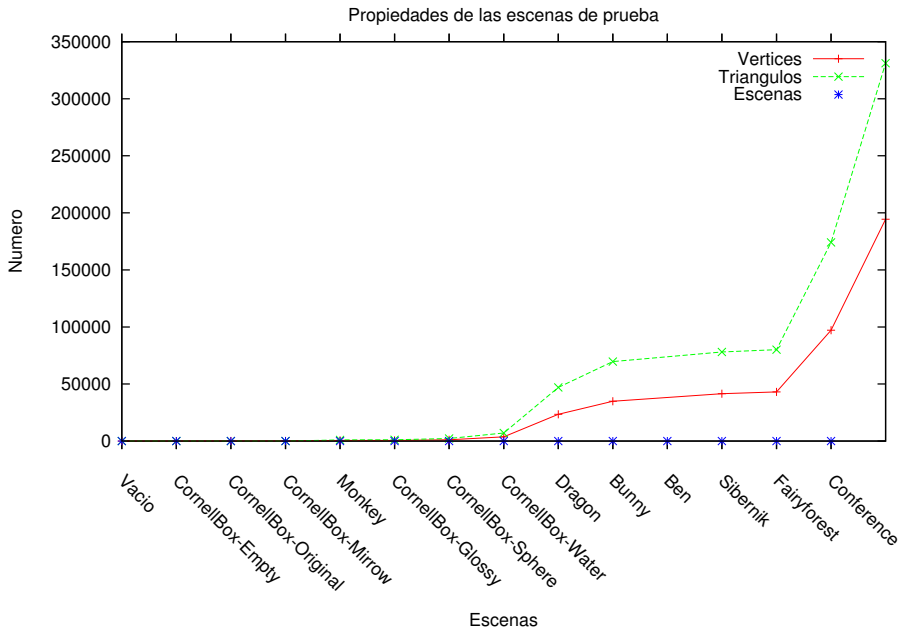


Figura 6.1: Propiedades de las escenas de prueba.

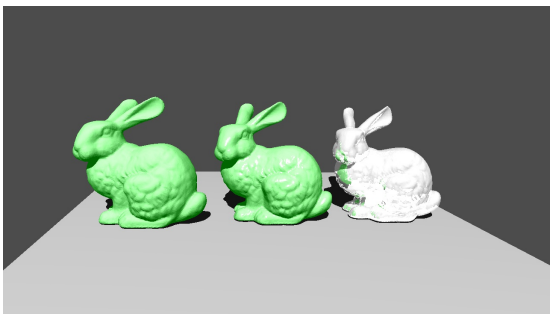
6.3. Resultados del proceso de render

Para iniciar esta sección, en la figura 6.2, se muestran los tipos de materiales resultantes de los 9 modelos mencionados en la sección 5.2.3, con respecto a la implementación del archivo *obj_material.cu*. Éstos tipos de materiales son soportados por el programa de render implementado, basado en el algoritmo para el trazado de rayos y utilizando el motor Nvidia®OptiX™.

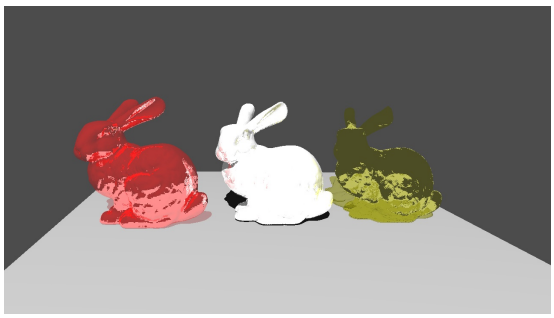
En la figura 6.3 se presentan las imágenes resultantes del proceso de render, con nuestra implementación basada en el trazado de rayos, de los 9 escenas de prueba descritas en la sección anterior. Con respecto al tema de iluminación, se utilizan luces básicas. Y todas las pruebas que se han hecho, trabajan solo con una luz. Sin embargo, para un número de luces superior, los tiempos de render se incrementarán.

Para obtener los resultados de los tiempos involucrados en el proceso de render para las escenas, se realizaron varios scripts de prueba para una resolusión de $1280x720$, para ser ejecutados tanto en el *device* 0, que corresponde a la tarjeta GeForce GTX 460, como para el *device* 1, que corresponde a la tarjeta Tesla C2070. Adicionalmente se realizaron ejecuciones en donde se considera el *warmup*, término correspondiente al periodo de calentamiento que afecta a las aplicaciones en la etapa de inicialización, por ejemplo, la carga de bibliotecas ó de imágenes en memoria.

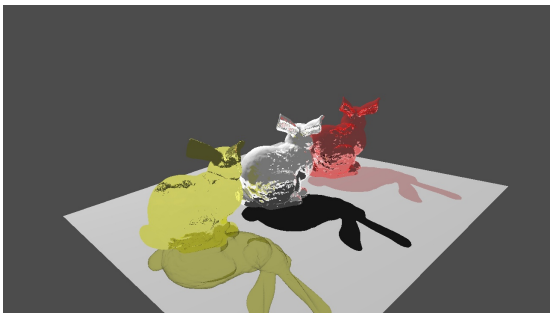
Los tiempos de render para cada escena con el *device* 0 se detallan en la tabla 6.4. Los resultados se clasifican en tiempo de inicialización de escena (etapa en la cual se carga el modelo de la escena y las propiedades de sus materiales, se construye la estructura de aceleración y se envía toda la información al GPU); tiempo del proceso



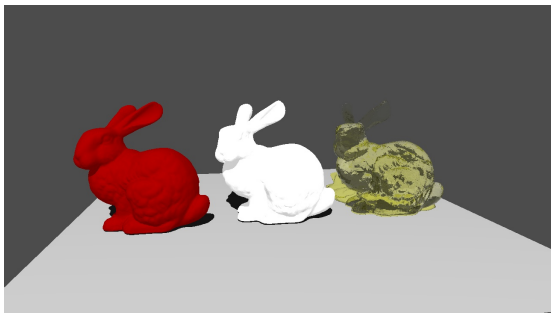
(a) Modelos 1, 2, 3



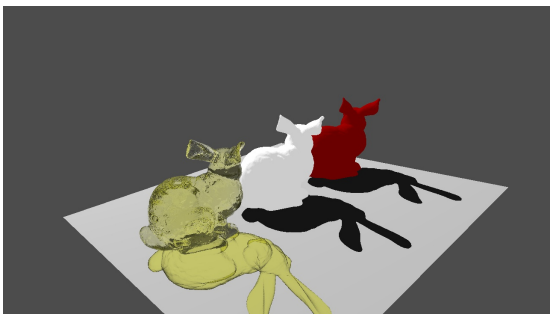
(b) Modelos 4, 5, 6



(c) Modelos 4, 5, 6 (vista trasera)



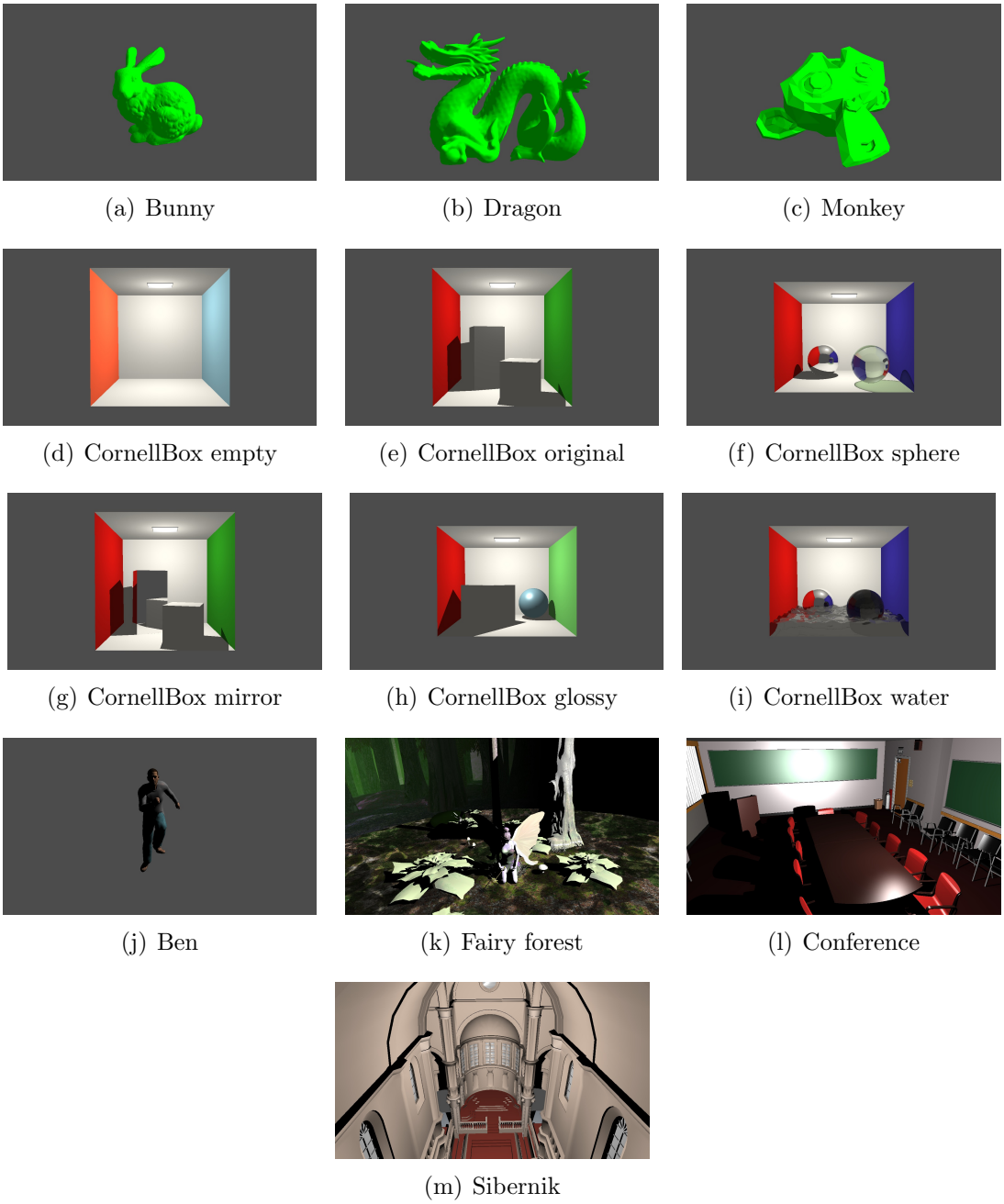
(d) Modelos 9, 8, 7



(e) Modelos 9, 8, 7 (vista trasera)

Figura 6.2: Modelos de iluminación soportados por RentiX.

de render el cual se subdivide en dos, ya que se realizaron ejecuciones tanto con periodo de calentamiento, como sin periodo de calentamiento (con el objetivo de diferenciar los resultados y poder realizar las conclusiones pertinentes); tiempo que le toma al procesador el guardar la imagen; tiempo total de ejecución por escena; y la memoria utilizada durante la ejecución tanto del *host* como del *device*. Todos éstos son índices necesarios para poder administrar adecuadamente el render. Cabe aclarar que los tiempos fueron tomados con el servidor libre de cualquier otro proceso de cómputo. Por lo tanto, cuando el servidor se encuentre ejecutando los procesos necesarios para mantener funcionando el sistema de *render farm* se espera que los tiempos de render se incrementen debido a la carga de trabajo atendida por el servidor.



(a) Bunny (b) Dragon (c) Monkey

(d) CornellBox empty (e) CornellBox original (f) CornellBox sphere

(g) CornellBox mirror (h) CornellBox glossy (i) CornellBox water

(j) Ben (k) Fairy forest (l) Conference

(m) Sibernik

Figura 6.3: Imagenes generadas mediante el proceso de render de las escenas de prueba.

Nombre escena	T. inicio	Render				T. imagen	T. total	Host	Device
		S/warmup		C/warmup					
		Tiempo	Fps	Tiempo	Fps				
Vacio	0.988 0	0.005 5	186.505 0	0.002 0	491.654 0	0.023 4	1.016 9	111	88
CornellBox-Empty	1.152 2	0.009 9	99.631 9	0.005 5	182.013 0	0.024 9	1.187 0	115	91
CornellBox-Original	1.153 0	0.011 9	83.906 2	0.006 6	150.739 0	0.025 2	1.190 2	115	91
CornellBox-Mirrow	1.152 7	0.012 3	80.238 4	0.007 2	139.313 0	0.024 9	1.190 0	115	91
Monkey	1.122 7	0.033 5	29.984 2	0.008 5	117.534 0	0.025 3	1.181 6	115	93
CornellBox-Glossy	1.156 6	0.035 8	28.041 1	0.005 9	168.690 0	0.024 8	1.217 2	116	93
CornellBox-Sphere	1.161 7	0.068 8	14.502 6	0.014 5	69.194 7	0.025 5	1.256 0	116	94
CornellBox-Water	1.178 1	0.235 5	4.248 4	0.030 4	32.851 4	0.025 5	1.439 1	116	95
Dragon	1.291 3	1.131 0	0.882 0	0.016 7	59.794 8	0.027 9	2.450 2	124	96
Bunny	1.311 3	1.556 6	0.642 7	0.008 2	121.758 0	0.025 7	2.893 6	132	98
Ben sin text	1.452 0	1.185 5	0.843 6	0.005 1	195.694 0	0.024 1	2.661 5	136	101
Ben	23.891 9	1.191 9	0.836 0	0.005 1	196.381 0	0.023 8	25.107 6	169	141
Sibernik	2.025 3	1.381 8	0.725 4	0.036 0	27.796 2	0.037 1	3.444 2	142	106
Conference	2.562 6	2.475 9	0.402 5	0.031 2	32.022 5	0.031 4	5.069 9	206	125
Fairy forest sin text	1.855 1	1.609 0	0.621 5	0.031 0	32.248 5	0.027 7	3.491 9	161	190

Tabla 6.4: Detalles del proceso de render de las escenas de prueba con del *device* 0.

De igual manera, los tiempos de render para cada escena con el *device* 1 se detallan en la tabla 6.5; clasificadas de manera similar que la tabla anterior. Cabe resaltar que para ésta tabla, a diferencia de la anterior, se muestran los resultados para la escena *Fairy forest*, ya que debido a que su ejecución requiere de mayor capacidad de memoria que la soportada por el *device* 0. Ésta propiedad ésta considerada dentro del gestor para determinar la distribución de los trabajos entre las diferentes tarjetas que componen el *render farm*.

Nombre escena	T. inicio	Render				T. imagen	T. total	Host	Device
		S/warmup		C/warmup					
		Tiempo	Fps	Tiempo	Fps				
Vacio	1.1146	0.0057	176.1790	0.0097	493.8540	0.0230	1.1474	110	167
CornellBox-Empty	1.1438	0.0111	89.8234	0.0042	239.7980	0.0253	1.1802	115	172
CornellBox-Original	1.1459	0.0133	75.8601	0.0049	202.1840	0.0252	1.1844	116	172
CornellBox-Mirrow	1.1484	0.0137	73.8161	0.0053	188.1780	0.0253	1.1874	116	172
Monkey	1.1168	0.0320	31.3223	0.0062	161.6050	0.0253	1.1741	115	174
CornellBox-Glossy	1.1484	0.0373	26.7330	0.0053	188.1780	0.0248	1.2105	116	174
CornellBox-Sphere	1.1552	0.0695	14.3707	0.0119	84.0761	0.0254	1.2500	116	175
CornellBox-Water	1.1674	0.2322	4.2987	0.0240	41.6979	0.0255	1.4251	115	176
Dragon	1.2858	1.1295	0.8897	0.0116	86.2068	0.0282	2.4435	125	177
Bunny	1.2980	1.5491	0.6435	0.0060	167.4770	0.0257	2.8727	132	179
Ben sin text	2.0436	1.1858	0.8433	0.0041	245.2240	0.0239	3.2533	141	182
Ben	23.8321	1.1986	0.8361	0.0041	243.8970	0.0240	25.0547	169	222
Sibernik	2.0187	1.3758	0.7261	0.0253	39.4509	0.0373	3.4317	138	1877
Conference	2.5607	2.4926	0.4036	0.0217	46.0386	0.0312	5.0844	211	206
Fairy forest	10.5763	2.0068	0.4956	0.0318	31.4949	0.0429	12.6260	1592	1610
Fairy forest sin text	1.8616	1.6259	0.6151	0.0310	32.2592	0.0278	3.5153	161	190

Tabla 6.5: Detalles del proceso de render de las escenas de prueba con del *device* 1.

De los datos arrojados por las tablas descritas anteriormente se derivan las siguientes conclusiones:

- Dentro del proceso de render, se consume mayor tiempo en la etapa de inicialización de la escena, ya que es en este paso en donde se tiene que generar la estructura de la escena y se cargan en memoria todos las imágenes que serán útiles para las texturas del modelo. Dentro de los tiempos más altos se tienen a la escena de Ben y a la escena de Fairy forest, que son las dos escenas que importan imágenes. Éste tiempo también se ve afectado por el tipo archivo de imagen que se importa; se tiene dos imágenes, la de tipo binaria y la de tipo ascii. Se observa que para la escena de Ben se cuenta con el índice más alto (importando imágenes de tipo ascii), y para la de Fairy forest (importando imágenes tipo binarias) el tiempo es mucho menor, otro factor determinante en el tiempo de inicialización es la información que se transfiere del *host* al *device*.

- Haciendo una comparación del tiempo de render con y sin *warmup*, se ve una gran diferencia, en algunos casos incluso triplica los fps generados. Éste factor es determinante en el tiempo de procesamiento; sin embargo, para fines de esta tesis queda excluido, debido a que nuestro objetivo queda limitado a realizar el proceso de render para cada escena, aunque se retornará dentro de las conclusiones y trabajo a futuro.

6.3.1. Estimaciones de las propiedades del proceso de render

Del estudio de éstas pruebas, se realizó los siguientes pasos para obtener un calculo estimado de cada una de las propiedades involucradas en el proceso de render para cada uno de los *devices*.

1. Tiempo de inicialización
2. Tiempo de render
3. Tiempo de guardado de imagen
4. Tiempo total
5. Memoria del *host*
6. Memoria del *device*

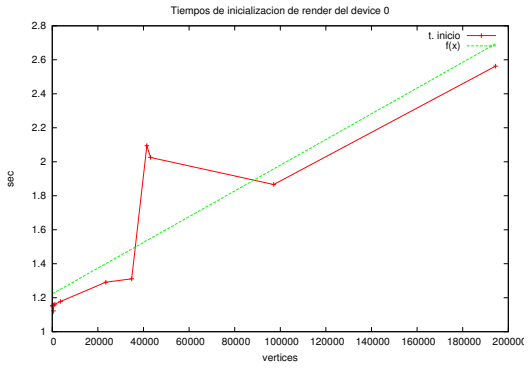
Estimación del tiempo inicial

El tiempo inicial corresponde a la inicialización de variables, carga del modelo, generación de la estructura de aceleración, lectura de imágenes desde disco y trasferencia

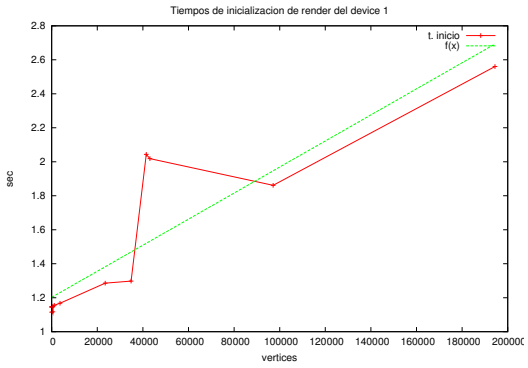
de datos entre el *host* y el *device*. La función general del tiempo de inicialización queda de la siguiente forma:

$$t_{inicial} = f(x) + t_{carga} + t_{transferencia} \tag{6.1}$$

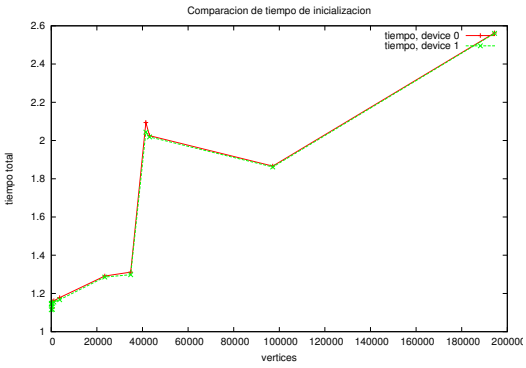
En la figura 6.4 se muestran los datos tanto del *device* 0 como del *device* 1 (en éstas gráficas no se consideran los tiempos de inicialización para Ben y para Fairy forest con texturas, más adelante se realizara el calculo necesario para poder considerar éste tipo de escenas). Como se puede observar, el tiempo de inicialización para ambos *devices* es muy similar, se considera la función de ajusta de $f(X) = 1.21 + 7.619e - 06x$, donde x es el número de vértices.



(a) *device* 0



(b) *device* 1



(c) Comparativa de tiempos de inicialización

Figura 6.4: Gráficas correspondiente a la inicialización del proceso de render.

Para aquellas imágenes que cuentan con imágenes de entrada correspondiente a las texturas, se clasifican en dos: imágenes tipo *ascii* e imágenes binarias. Para imágenes de tipo *ascii* el tiempo de carga desde disco a memoria es mucho mayor al de las imágenes binarias. Para éste servidor en particular, el tiempo de carga (t_{carga}), desde disco para las imágenes binarias es de 0.001266851 seg. por MB y para las imágenes tipo ascii es de 2.48804 seg. por MB; ésta propiedad es muy importante en el calculo de la estimación de tiempo inicial.

Otro factor muy importante es el tiempo de trasferencia ($t_{trasferencia}$) de datos de subida entre el *host* y el *device*. Para el *device* 0 (GeForce GTX 460), el tiempo de subida es de $0.0053776MB/seg$, mientras que para el *device* 1 (Tesla C2070) el tiempo de subida es de $0.00657088MB/seg$. De tal forma que la fórmula resultante para el *device* 0 es 6.2 y la fórmula resultante para el *device* 1 es 6.3:

$$t_{inicial_{D0}} = 1.21 + 7.619e-06x + t_{carga} + 4 * 0.0053776y \tag{6.2}$$

$$t_{inicial_{D1}} = 1.21 + 7.619e-06x + t_{carga} + 4 * 0.00657088y \tag{6.3}$$

donde x es el número de vértices y y es el número de MB perteneciente a la cantidad de memoria de las imágenes de textura. En la tabla 6.6 se muestra la comparación entre los tiempos de inicialización obtenidos y los tiempos de inicialización estimados. Un punto importante a resaltar es que el tamaño de la resolución de la imagen no altera éste tiempo de inicialización.

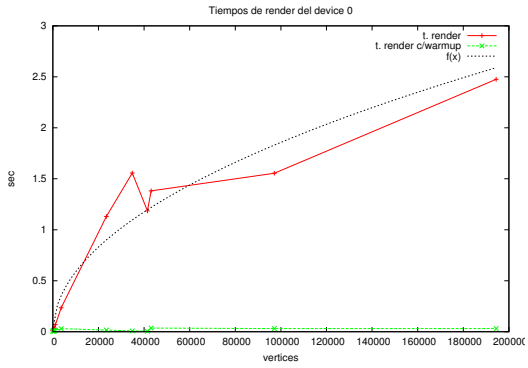
Nombre escena	*device* 0			*device* 1		
	Tiempo	T. est	Error	Tiempo	T. est	Error
Vacio	0.988 0	1.210 0	0.222 1	1.114 6	1.210 0	0.095 4
CornellBox-Empty	1.152 2	1.210 2	0.058 0	1.143 8	1.210 2	0.066 4
CornellBox-Original	1.153 0	1.210 5	0.057 5	1.145 9	1.210 5	0.064 6
CornellBox-Mirrow	1.152 7	1.210 5	0.057 8	1.148 4	1.210 5	0.062 1
Monkey	1.122 7	1.213 9	0.091 1	1.116 8	1.213 9	0.097 1
CornellBox-Glossy	1.156 6	1.214 4	0.057 8	1.148 4	1.214 4	0.066 0
CornellBox-Sphere	1.161 7	1.218 5	0.056 8	1.155 2	1.218 5	0.063 3
CornellBox-Water	1.178 1	1.237 9	0.059 9	1.167 4	1.237 9	0.070 6
Dragon	1.291 3	1.388 9	0.097 6	1.285 8	1.388 9	0.103 1
Bunny	1.311 3	1.475 4	0.164 1	1.298 0	1.475 4	0.177 5
Ben sin text	2.094 2	1.526 0	0.568 2	2.043 6	1.526 0	0.517 6
Ben	23.891 9	23.096 0	0.795 8	23.115 5	23.189 9	0.074 5
Sibernik	2.025 3	1.538 3	0.487 0	2.018 7	1.538 3	0.480 4
Conference	2.562 6	2.632 0	0.069 4	2.560 7	2.691 1	0.130 5
Fairy forest sin text	1.866 2	1.894 9	0.028 7	1.861 6	1.950 0	0.088 4
Fairy forest				10.576 3	11.710 4	1.134 0

Tabla 6.6: Tabla de estimación del tiempo de inicialización.

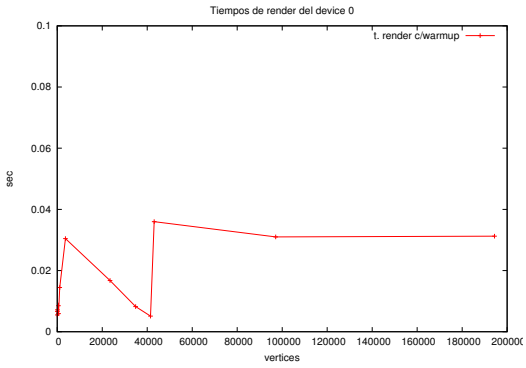
Estimación del tiempo de render

Para calcular este tiempo se requiere considerar muchos factores: resolución de la imagen, número de objetos en la escena, número de luces, complejidad de la escena, tipo de materiales que lo componen y por consiguiente el número y tipo de rayos

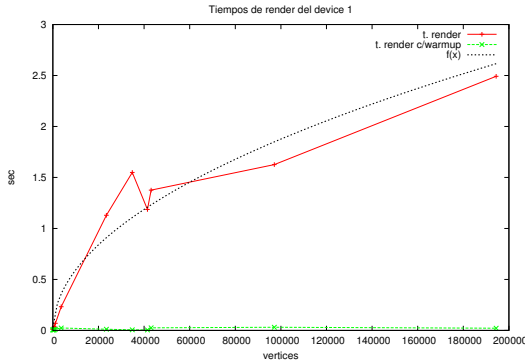
generados, tiempo de duración de cada tipo de rayo. Sin embargo, no todos estos componentes son fáciles de calcular.



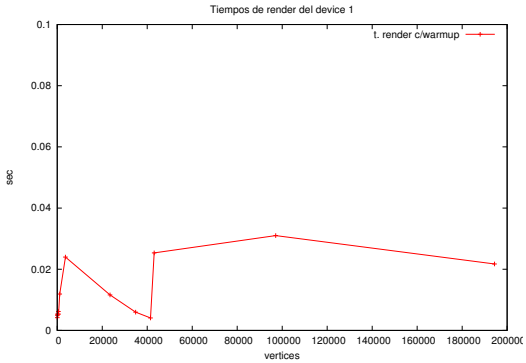
(a) Tiempos de render para el *device* 0



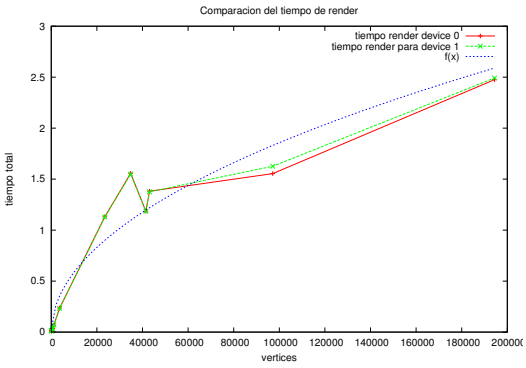
(b) Tiempos de render con *warmup* para el *device* 0



(c) Tiempos de render para el *device* 1



(d) Tiempos de render con *warmup* para el *device* 1



(e) Gráfica comparativa de los tiempos de render

Figura 6.5: Gráficas del proceso de render para el *device* 0.

En el caso de la complejidad de la escena, no se puede formar una idea con antelación de que tan complicada puede llegar a ser, sino hasta después de haber sido renderizada. Para el número de rayos que se generan, se puede llegar a realizar un

estimado, sin embargo, éste factor es dependiente de tipo de materiales con el que se encuentren compuestos y los objetos que se encuentran cerca de él; además hay que considerar que procesamiento se corre dentro del GPU y debido a su capacidad paralelizable impide realizar un calculo exacto del número de rayos generados por cada escena.

En las figuras 6.5 se muestran los tiempos de procesamiento de render para cada uno de los *devices*. En la figura 6.5 también se realiza una comparación de los tiempos de render para los dos *device*; como se puede apreciar, los tiempos de render varían por muy poco por lo cual, para calcular el tiempo de render se utilizará la fórmula $f(x) = 0.00778 * \sqrt{0.575x}$.

Nuevamente, el tamaño de la resolución de la imagen de salida no altera el tiempo de render.

Estimación del tiempo de guardado de imagen

El proceso de guardado de la imagen se encuentran involucrados dos pasos: la obtención del buffer de salida del GPU y el almacenamiento de la imagen dentro de un archivo. En la figura 6.6 se puede observar los tiempos que lleva en guardar la imagen para los dos *devices*. Nuevamente se aprecia que el tiempo es casi igual variando por centésimas de segundo.

El tamaño de la resolución de la imagen de salida, si altera el tiempo de guardado, sin embargo no por mucho, ya que la diferencia es de alrededor de una centésima de segundo.

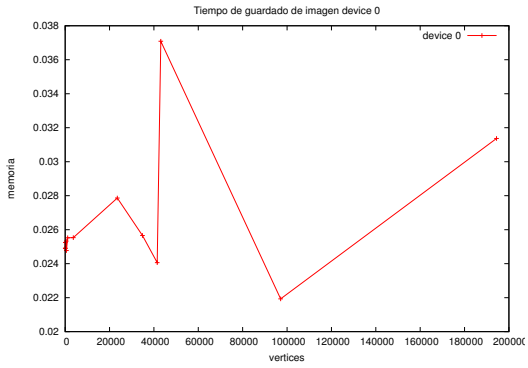
Estimación del tiempo total

En la figura 6.7 se aprecia el tiempo total del proceso de render, como se puede apreciar el tiempo para los dos *devices* es equivalente. De los análisis anteriores, se obtiene una fórmula general para ambos casos, $f(x) = 1.21 + 7.619e^{-6}x + (0.00778 * sqrt(0.575x))$; sin olvidar considerar las escenas que necesitan las imágenes, sumando el tiempo de carga y la tasa de trasferencia entre el *host* y el *device* comentadas en la estimación del tiempo de inicialización. En la tabla 6.7 se observan los resultados totales, los estimados y el error.

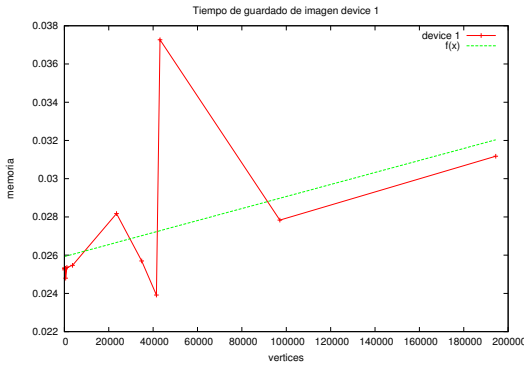
Como conclusión a los tiempos de comparación entre los tiempos de render se tiene que se puede usar una única función para determinar el tiempo de render, sin embargo, no hay que olvidar ni dejar de lado aquellas imágenes que requieren de texturas, puesto que tanto la lectura de las imágenes a disco como la subida de información hacia los GPUs conllevan un costo en el tiempo de procesamiento dependiente de el tamaño de información.

Estimación de la memoria utilizada en el *host* y en el *device*

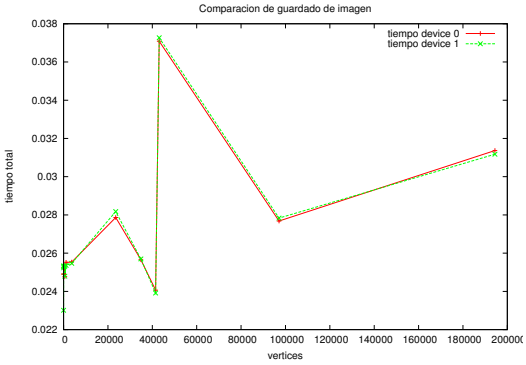
Como se puede observar en la figura 6.8, la memoria ocupada por cada uno de los *devices*, es muy diferente, éste es un factor a considerar en el balance de cargas, puesto



(a) Tiempos de guardado *device* 0

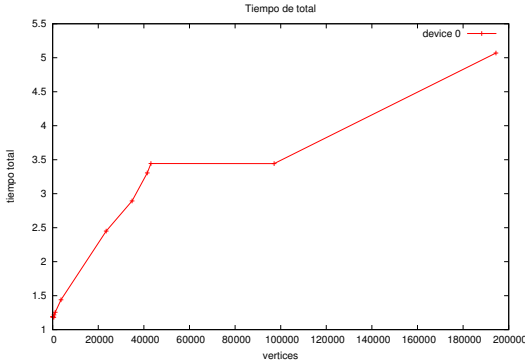


(b) Tiempos de guardado *device* 1

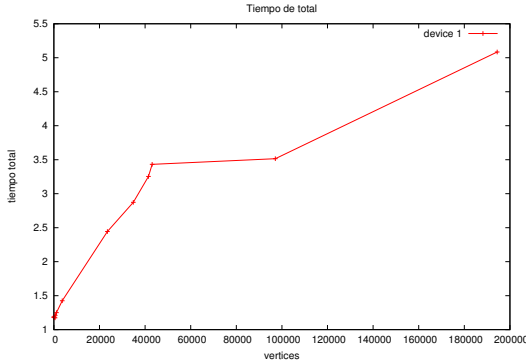


(c) Comparación de tiempos de guardado

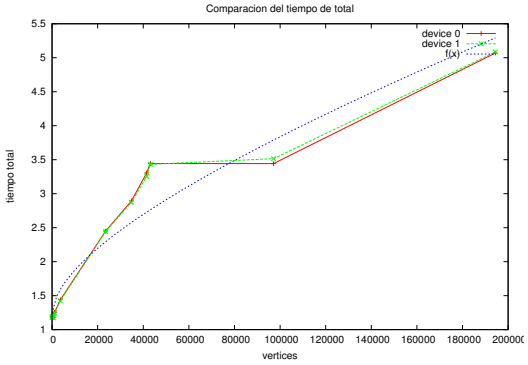
Figura 6.6: Gráficas del tiempo de guardado.



(a) Tiempo total del *device* 0



(b) Tiempo total del *device* 1



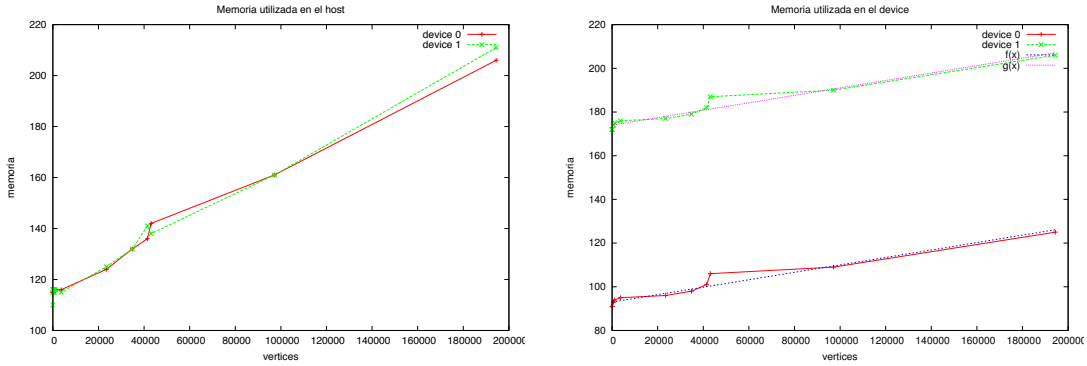
(c) Comparativa del tiempo total

Figura 6.7: Gráficas del tiempo total.

Nombre escena	T. total D0	T. total D1	T. estimado	Error
Vacio	1.016 9	1.147 4	1.235 9	0.236 3
CornellBox-Empty	1.187 0	1.180 2	1.265 0	0.115 2
CornellBox-Original	1.190 2	1.184 4	1.286 5	0.140 4
CornellBox-Mirrow	1.190 0	1.187 4	1.286 5	0.138 4
Monkey	1.181 6	1.174 1	1.372 6	0.275 6
CornellBox-Glossy	1.217 2	1.210 5	1.382 2	0.238 1
CornellBox-Sphere	1.256 0	1.250 0	1.441 5	0.266 7
CornellBox-Water	1.439 1	1.425 1	1.621 2	0.267 6
Dragon	2.450 2	2.443 5	2.319 7	0.179 9
Bunny	2.893 6	2.872 7	2.603 5	0.395 8
Ben sin text	2.661 5	3.253 3	2.754 7	0.507 3
Ben	25.107 6	25.054 7	24.377 5	0.995 8
Sibernik	3.444 2	3.431 7	2.790 1	0.916 2
Conference	5.069 9	5.084 4	5.324 3	0.349 6
Fairy forest sin text	3.491 9	3.515 3	3.817 5	0.444 3
Fairyforest		12.626 0	13.577 9	0.951 9

Tabla 6.7: Tabla de estimación del tiempo total.

que es muy posible que uno pueda albergar más trabajos que el otro. La memoria del *host* se estima con la fórmula $f(x) = 115.43 + 0.00048x$; para estimar la memoria del *device*0 $f(x) = 93 + 0.000170273x$ y para el *device* 1 $g(x) = 174 + 0.000170273x + 4 * y$, en donde y son los MB correspondientes a las imágenes de textura utilizadas. En la tabla 6.8 se observan los datos relacionados con la memoria del *host* y la memoria de cada uno de los *devices*, así como su respectiva estimación.



(a) Memoria del *host* (b) Memoria del *device*

Figura 6.8: Memoria utilizada.

Nombre escena	Mem. *host*	Mem. dev. 0	Mem. dev. 1	Est. *host*	Est. dev. 0	Est. dev. 1
Vacio	111.000 0	88.000 0	167.000 0	115.430 0	93.000 0	174.000 0
CornellBox-Empty	115.000 0	91.000 0	172.000 0	115.441 5	93.004 1	174.004 1
CornellBox-Original	115.000 0	91.000 0	172.000 0	115.464 6	93.012 3	174.012 3
CornellBox-Mirrow	115.000 0	91.000 0	172.000 0	115.464 6	93.012 3	174.012 3
Monkey	115.000 0	93.000 0	174.000 0	115.673 4	93.086 3	174.086 3
CornellBox-Glossy	116.000 0	93.000 0	174.000 0	115.707 4	93.098 4	174.098 4
CornellBox-Sphere	116.000 0	94.000 0	175.000 0	115.965 7	93.190 0	174.190 0
CornellBox-Water	116.000 0	95.000 0	176.000 0	117.189 7	93.624 2	174.624 2
Dragon	124.000 0	96.000 0	177.000 0	126.702 3	96.998 7	177.998 7
Bunny	132.000 0	98.000 0	179.000 0	132.150 8	98.931 5	179.931 5
Ben sin text	136.000 0	101.000 0	182.000 0	135.337 5	100.061 9	181.061 9
Ben	171.000 0	141.000 0	222.000 0	169.802 4	134.526 7	215.526 7
Sibernik	142.000 0	106.000 0	187.000 0	136.110 8	100.336 2	181.336 2
Conference	196.000 0	125.000 0	206.000 0	208.741 5	126.100 9	207.100 9
Fairy forest sin text	161.000 0	190.000 0	190.000 0	162.049 5	109.537 6	190.537 6
Fairy forest	1 592.000 0		1 610.000 0	1 579.144 5		1 607.632 5

Tabla 6.8: Tabla de estimación de la memoria.

6.3.2. Gestores de tareas implementados

Para obtener un *render farm* óptimo, se experimentó principalmente con dos enfoques de programación de granularidad gruesa: Esquema de planificación global y el esquema de planificación particionado.

Esquema de programación de planificación global

Para implementar éste esquema considera un módulo de recepción de trabajos, y una cola en donde se reciben todas las tareas de render que se tienen que despachar de forma independiente.

Como primer esquema de planificación se intenta generar un equilibro entre la carga de trabajo de los GPUs considerando como base a las tareas; es decir, la asignación a los GPUs se realiza mediante el intercalado de tareas,éste esquema, como bien lo índica la teoría, se adapta perfectamente cuando tanto los procesadores ó en nuestro caso GPUs como cuando las tareas que se van a ejecutar cuentan con propiedades homogéneas. Sin embargo no es nuestro caso, en el sistema del *render farm* se cuentan con GPUs con diversas características y con tareas que consumen recursos de diferente manera; por ejemplo la memoria RAM requerida ó incluso la cantidad de tiempo de procesamiento. Por lo tanto, la ejecución de unas tareas se llegan a adaptar mejor en un GPU que en otro.

Esquema de programación de planificación particionada

Éste esquema considera un módulo de recepción de trabajos y una cola en donde se reciben todas las tareas de render que se tienen que despachar de forma independiente, se lleva acabo un balance de cargas y se envia al despachador local correspondiente al *device* asignado. El despachador de tareas local es el encargado de elegir que, cuando y como se ejecutan las tareas. Sin embargo, debido a que el tiempo de procesamiento de render no es exacto y puede variar con respecto a la carga de trabajo que se tenga en el servidor, se implemento dos políticas de balanceo, la de donación de tareas y robo.

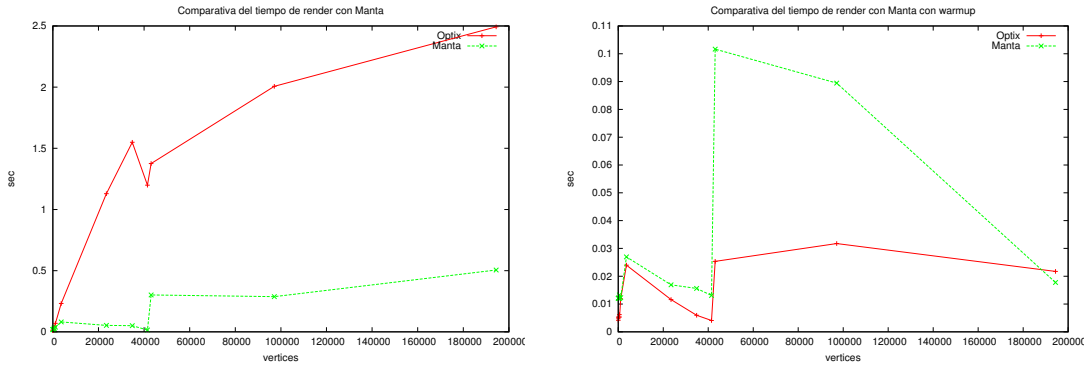
La política de donación resulta costosa por los tiempos de transmisión tanto de verificación como de información, pero en la de robo, los tiempos de verificación se minimizan, resultando una mejor opción en la elección de un gestor de tareas.

6.3.3. Comparativa con Manta

Para confirmar que el camino viable para que el proceso de render sea mediante el uso de GPUs, se realiza una comparación con el motor de render Manta, el cual corre sobre 12 procesadores con memoria compartida. En la tabla reftabla:Comparativa se pueden observar los resultados.

Nombre escena	Optix				Manta			
	S/warmup		C/warmup		S/warmup		C/warmup	
	Tiempo	Fps	Tiempo	Fps	Tiempo	Fps	Tiempo	Fps
CornellBox-Empty	0.011 1	89.823 4	0.004 2	239.798 0	0.019 4	51.541 1	0.012 1	82.417 6
CornellBox-Original	0.013 3	75.860 1	0.004 9	202.184 0	0.017 3	57.783 4	0.012 1	82.690 2
CornellBox-Mirrow	0.013 7	73.816 1	0.005 3	188.170 0	0.018 2	55.026 7	0.012 1	82.596 8
Monkey	0.032 0	31.322 3	0.006 2	161.605 0	0.031 5	31.746 0	0.013 3	75.405 3
CornellBox-Glossy	0.037 3	26.733 0	0.005 3	188.178 0	0.025 5	39.283 5	0.012 0	83.169 3
CornellBox-Sphere	0.069 5	14.370 7	0.011 9	84.076 1	0.038 6	25.916 1	0.012 3	81.395 7
CornellBox-Water	0.232 2	4.298 7	0.024 0	41.697 9	0.081 0	12.352 1	0.027 0	37.089 7
Dragon	1.129 5	0.889 7	0.011 6	86.206 8	0.052 9	18.912 5	0.016 9	59.138 9
Bunny	1.549 1	0.643 5	0.006 0	167.477 0	0.050 1	19.978 4	0.015 6	64.051 9
Ben	1.198 6	0.836 1	0.004 1	243.897 0	0.014 8	67.645 3	0.013 1	76.476 0
Sibernik	1.375 8	0.726 1	0.025 3	39.450 9	0.301 3	3.318 5	0.101 7	9.835 8
Conference	2.492 6	0.403 6	0.021 7	46.038 6	0.505 2	1.979 5	0.017 7	56.375 1
Fairy forest	2.006 8	0.495 6	0.031 8	31.494 9	0.287 9	3.474 0	0.089 5	11.178 3

Tabla 6.9: Comparativa de los tiempo de render de OptiX con el motor Manta.



(a) Comparativa de tiempos de render

(b) Comparativa de tiempos de render con *warmup*

Figura 6.9: Gráfica de los tiempos de render de Optix vs Manta.

Como se puede apreciar en la figura 6.9, los tiempos de render para el motor Manta superan en tiempo a los de Optix, sólo en las pruebas realizadas sin *warmup*. Por lo contrario en las pruebas los tiempos de render con *warmup* son muy superiores, lo cual nos indica que en efecto, el motor de trazado de rayos OptiX funciona de manera óptima para el trazado de rayos "interactivo"; puesto que para la generación fotogramas posteriores al tiempo de inicialización se reduce el tiempo de render. Hay que tomar en cuenta que el programa de Manta no ésta diseñado para trabajar con la lectura del archivo MTL de manera óptima; es decir cuando se utilizan materiales con modelos de iluminación superiores a 2 no los considera, lo cual en efecto, reduce el tiempo de procesamiento debido a que no traza rayos de reflexión y refracción; aún y cuando el motor en sí, si los soporta. En el apéndice 7.3 se observan las imágenes resultantes de Manta.

Capítulo 7

Conclusiones y trabajo a futuro

En esta tesis se trabajo con la validación de un método de balanceo de cargas con distintas políticas de planificación y migración de tareas para un *render farm* basado en GPUs, la parte del proceso de render se realizó con el motor para el trazado de rayos OptiX, el cual es un framework auxiliar para diferentes aplicaciones que se puedan resolver mediante el algoritmo para el trazado de rayos. Particularmente se trabajo con archivos *.obj, utilizando los modelos de iluminación dados por el archivo de biblioteca de materiales *.mtl [49] y detallados la sección 5.2.3.

7.1. Conclusiones finales

A continuación se enlistan cada una de las conclusiones finales.

- El uso del GPU en la industria de computación gráfica ha venido a revolucionar en muchos aspectos la generación de imágenes en 3D: Prueba de esto es que ahora podemos observar en nuestro mundo cotidiano animaciones cada vez más realistas con mejores imágenes, con mayor resolución y de mayor calidad; sin dejar de lado los plazos de tiempo que se tienen designados a esta tarea. La influencia que ha tenido sobre el algoritmo para el trazado de rayos es muy alta por las propiedades del algoritmo de ser altamente paralelizable. Logrando generar un mayor número de fps, a los que se habían obtenido con sistemas convenciones multiprocesador o distribuidos.

- Propiedades de los trabajos: En un sistema de *render farm*, no todos los trabajos son iguales, el tiempo de procesamiento de render en un GPU no es el mismo que en otro GPU, inclusive los recursos utilizados no son iguales. Ésta propiedad es determinante en la asignación de tareas, debido a además de mantener la carga equilibrada, se busca que la tarea finalice en el periodo más corto.

- Para la generación de imágenes realistas se tienen que considerar muchos factores, tales como la iluminación, las sombras, propiedades de los materiales,

texturas, resolución, entre otros: Por lo tanto es difícil, en un inicio, poder determinar la complejidad de la escena, sino hasta una vez generada. Puede por ejemplo utilizar un número alto de rayos a consecuencia de algunos tipos de materiales, numero de luces, resolución, entre otros; todos estos factores influyen en el proceso de render y es difícil poder realizar una aproximación viable del número de rayos finales utilizados.

- El tiempo de render esta dado por muchos factores: tiempo de cargas de bibliotecas, trasferencias de datos entre el *host* y el *device* y viceversa, tiempo de guardado del archivo y el tiempo correspondiente al trazado de rayos. Por tal motivo es difícil generalizar los tiempos de procesamiento por tarea. Aún y cuando este dato es de vital importancia para poder mantener una carga equilibrada de trabajo entre los distintos GPUs.

- De la comparación de Optix con Manta: Se concluye que la trasferencia de datos entre el host y el device es un factor que reduce el rendimiento del GPU en su primer intento por renderizar la escena. Es por esto que en las pruebas sin periodo de calentamiento el motor Manta resultó ser más eficiente. Sin embargo, cuando se intentó realizar el proceso de render con un periodo de calentamiento (relativamente chico $w = 2$), se demostró el verdadero poder cómputo que tiene el GPU.

- La arquitectura basada en el esquema de participando basado en colas locales: Debido a que algunas escenas demandan más capacidad de memoria que otras, y el tamaño de memoria es primordial para garantizar el éxito del proceso de render, se eligió adecuadamente la arquitectura logrando emparejar las propiedades de las escenas con las propiedades del GPU.

- La técnica de balanceo de cargas funcionó adecuadamente: Aunque se deben de perfeccionar el calculo de tiempo de render por escenas, considerando un número mayor de éstas.

- De las técnicas de administración de cargas: la mejor que funcionó fue la de cola por donación ya que evita la comunicación excesiva entre los GPUs.

- El módulo de notificación es muy importante en el sistema de *render farm*, ya que permite mantener el estado de las tareas y poder notificarle al cliente cuando su trabajo ha terminado; sin embargo implica muchas transferencias de información, reduciendo de manera significativa el rendimiento del sistema y evitando el escalamiento del sistema a gran escala. Es un cuello de botella en el sistema.

Finalmente, ¿es posible implementar un *render farm* basado en GPUs? Si, aún y cuando son muchos los elementos y propiedades a considerar, se puede realizar un sistema de render cuyo algoritmo de trazado de rayos corra completamente sobre GPUs. Sin embargo si se quiere explotar al máximo la capacidad y rendimiento de

los GPUs se tiene que crear, en casos de renderizado de tareas de animación, un sistema que evite tener que cargar los elementos necesarios para las propiedades de las escenas nuevamente, creando una actualización del modelo de la escena anterior y la actual. Explotando el potencial del GPU demostrado en el uso de escenas interactivas. Además, se ha demostrado que el uso del GPU cuenta con un beneficio ecológico significativo dándole una ventaja muy grande sobre los sistemas multiprocesadores y sobre todo sobre los distribuidos. Es importante poder realizar cómputo de alto rendimiento sin dejan de lado los problemas ocasionados por el consumo excesivo de energía.

7.2. Contribuciones

- Se realiza un estudio sobre los contenidos digitales y el software tanto libre como comercial que auxilian al diseñador durante la etapa de pre-producción y producción. Ver capítulo 4.

- Se establecierón las bases para poder generar un gesto de tareas para un *render farm* basado en GPUs, considerando el balance de cargas para la asignación de tareas dentro de los GPUs, migración de tareas dentro del sistema para el proceso de render. Mediante el estudio de la determinación del tiempo de render y memoria requerida para diferentes escenas de prueba, con el fin de lograr un calculo aproximado útil en la asignación de los procesos.

- Diseño e implementación de la arquitectura de un *render farm* basado en GPUs a partir del estudio de las técnicas de asignación y planificación de tareas dentro de un esquema particionado con colas locales.

7.3. Trabajo a futuro

En lo referente a trabajo a futuro para el algoritmo para el trazado de rayos empleado se tiene:

- Formatos de archivos: Esta tesis solo estuvo enfocada en trabajar con archivos OBJ, sin embargo para poder expandir su uso, es conveniente utilizar otro tipos de archivos, considerando de manera similar las propiedades de los materiales.

- Tipo de iluminación: El algoritmo desarrollado solo contempla luces básicas, incorporar otro tipo de luces como las triangulares y utilizar otro tipo de algoritmos de iluminación global como radiosidad, con el objetivo de generar imágenes aún más realistas.

- Para poder visualizar la capacidad del GPU, en casos del proceso de render de trabajos de animación, realizar un sistema que evite tener que volver a cargar los

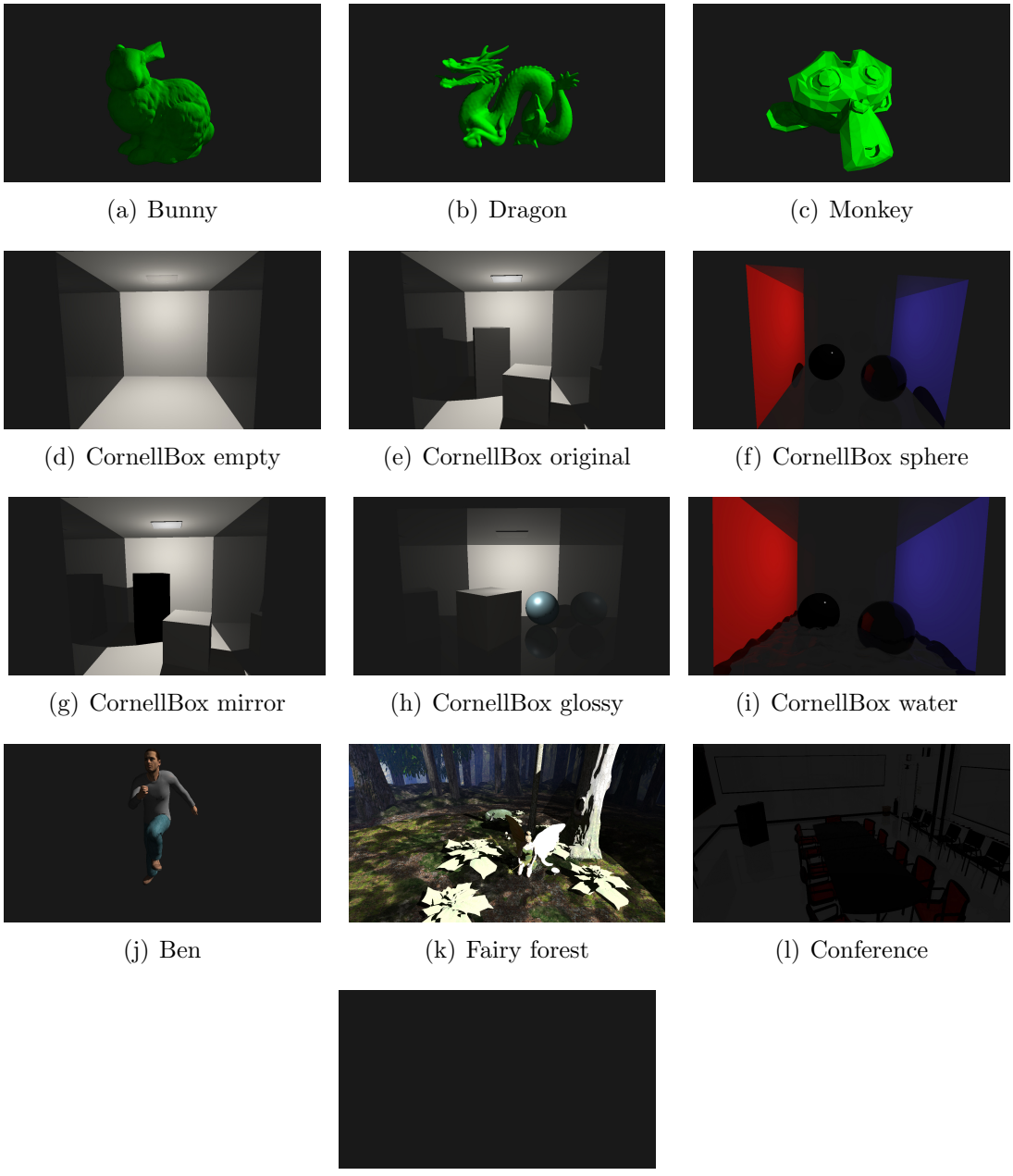
elementos necesarios para las propiedades de las escenas nuevamente, mediante la actualización de los vértices del modelo de la escena anterior y la actual.

Una vez definida la arquitectura del sistema de *render farm*, se puede experimentar con otros tipos de planificación, y asignación de tareas e inclusive se puede usar otros tipos de migración de tareas diferentes a los experimentados en esta tesis.

Se puede añadir un módulo auxiliar en la post-producción de películas de animación.

Apéndice 1

En éste apartado, se mencionan los experimentos realizados con Manta y los resultados obtenidos. Para éstos experimentos se usaron 12 procesadores, con una profundidad máxima de 10; las luces fueron ajustadas manualmente; sin embargo en algunas escenas no se pudo obtener resultados satisfactorios. Hay que puntualizar que no es error del motor Manta, simplemente no se pudieron ubicar adecuadamente. En algunos casos, éste problema redujo los tiempos del proceso de render. En la figura 1 se observan las imágenes resultantes. Como se puede observar, las imágenes que utilizan varios tipos de materiales como tipo espejo, tipo vidrio ó tipo agua no se leen bien desde el archivo MTL, pero es muy importante aclarar que el motor de manta en sí si soporta éste tipo de materiales. Éste problema también lo tenia el ObjLoadar.cpp de Optix, sin embargo se solucionó.



(a) Bunny (b) Dragon (c) Monkey

(d) CornellBox empty (e) CornellBox original (f) CornellBox sphere

(g) CornellBox mirror (h) CornellBox glossy (i) CornellBox water

(j) Ben (k) Fairy forest (l) Conference

(m) Sibernik

Figura 1: Imágenes generadas mediante el proceso de render de las escenas de prueba.

Bibliografía

[1] Peter Sciretta. A look at pixar and lucasfilm's renderfarms, February 25th 2010.

[2] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[3] Urszula Strojny. Modern computer graphics.

[4] Vahid Tehrani. 3d production.

[5] Jon Peddie Research. Digital content creation software market, 2007.

[6] Raffael Trappe. Television standards - formats and techniques, 02 2012.

[7] Robert Gove. Digital television system. 1996.

[8] J.D. Foley. *Computer graphics: principles and practice*. The Systems Programming Series. Addison-Wesley, 1996.

[9] Carmull E. A subdivision algorithm for computer display of curved surfaces. *University of Utah, Salt Lake City*, December 1974.

[10] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '80, pages 124–133, New York, NY, USA, 1980. ACM.

[11] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[12] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[13] Tomoyuki Nishita and Eihachiro Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, pages 23–30, New York, NY, USA, 1985. ACM.

[14] Yucel Gursac. 3-d computer animation production process on distance education programs through television. *Turkish Online Journal of Distance Education-TOJDE*, 2(2):57–68, June 2001.

[15] Andrew S. Glassner. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.

[16] Isaac D. Scherson and Elisha Caspary. Data structures and the time complexity of ray tracing. *The Visual Computer*, 3:201–213, 1987. 10.1007/BF01952827.

[17] Kelvin Sung and Peter Shirley. Graphics gems iii. chapter Ray tracing with the BSP tree, pages 271–274. Academic Press Professional, Inc., San Diego, CA, USA, 1992.

[18] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006.

[19] Ingo Wald and Carsten Benthin. Openrt - a flexible and scalable rendering engine for interactive 3d graphics. Technical report, 2002.

[20] Alan Chalmers and Erik Reinhard, editors. *Practical Parallel Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[21] Jack Dongarra Weicheng Jiang Robert Manchek Vaidyalingam S. Sunderam Al Geist, Adam Beguelin. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. The MIT Press, November 1994.

[22] Anthony Skjellum William Gropp, Ewing Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, volume 1. The MIT Press, October 1994.

[23] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21:703–712, July 2002.

[24] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[25] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Andrew S. Tanenbaum. *Sistemas Operativos Modernos*. 9702603153, 9789702603153. Pearson Educación, 2 edition, 2003.

[27] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS*. Chapman Hall/CRC, Boca, 2004.

[28] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches, 1995.

[29] M.J. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. Technical report, University of Rochester, Department of Computer Science, 1995.

[30] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. In *Computer Methods in Applied Mechanics and Engineering*, pages 485–500, 2000.

[31] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, October 1989.

[32] Kelvin Sung; Jason Loh Jen Shiuan; A.L. Ananda. Ray tracing in a distributed environment. *Computers and Graphics (Pergamon)*, 20(1):41–49, 1996.

[33] James Bigler, Abe Stephens, and Steven G. Parker. Design for parallel interactive ray tracing systems. In *in: Proceedings of IEEE Symposium on Interactive Ray Tracing*, pages 187–196, 2006.

[34] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, August 18, 2004.

[35] Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. Razor: An architecture for dynamic multiresolution ray tracing. Technical report, The University of Texas at Austin, Department of Computer Sciences, January 24 2007.

[36] Holger Ludvigsen and Anne Cathrine Elster. Real-time ray tracing using nvidia optix. *Eurographics Association*, pages 65–68, 2010.

[37] Huajun Jing Bin Gong. The design and implementation of render farm manager based on openpbs. In *Computer-Aided Industrial Design and Conceptual Design, 2008. CAID/CD 2008. 9th International Conference on*, Kunming, 2008. IEEE Computer Society.

[38] Zeeshan Patoli, Michael Gkion, Abdullah Al-Barakati, Wei Zhang, Paul Newbury, and Martin White. How to build an open source render farm based on desktop grid computing. In D.M.Akbar Hussain, AbdulQadeerKhan Rajput,

BhawaniShankar Chowdhry, and Quintin Gee, editors, *Wireless Networks, Information Processing and Systems*, volume 20 of *Communications in Computer and Information Science*, pages 268–278. Springer Berlin Heidelberg, 2009.

[39] M. ; Al-Barakati A. ; Zhang W. ; Newbury P. ; White M. Patoli, M.Z. Gkion. An open source grid based render farm for blender 3d. In *Power Systems Conference and Exposition, 2009. PSCE '09. IEEE/PES*, March 2009.

[40] Jiali Yao, Zhigeng Pan, and Hongxin Zhang. A distributed render farm system for animation production. In *Proceedings of the 8th International Conference on Entertainment Computing*, ICEC '09, pages 264–269, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Chaojian Fang, Yujiao Zhao, and Zhengjun Wang. Research and design of a service management system for deadline render farm. In *Environmental Science and Information Application Technology, 2009. ESIAT 2009. International Conference on*, volume 2, pages 542 –545, july 2009.

[42] Liang Zhi-yuan, Xu Zhi-qi, and Zhang Ling. Adaptive rendering cluster system based on browser/server architecture. In *Proceedings of the 2010 Second International Conference on MultiMedia and Information Technology - Volume 01*, MMIT '10, pages 71–74, Washington, DC, USA, 2010. IEEE Computer Society.

[43] HaiBin Cao, BaoFeng Lu, and Ming Zhu. The design and implementation of job management system based on feedback control. In *Proceedings of the 2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, DCABES '10, pages 198–201, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[45] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[46] NVIDIA. Nvidia's next generation cudatm compute architecture:fermitm. Technical report, 2009.

[47] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[48] Martin Peres-Koji Inoue Kazuaki Murakami Shinpei Kato Yuki Abe, Hiroshi Sasaki. Power and performance analysis of gpu-accelerated systems. *USENIX Symposium on Operating Systems Design and Implementation*, October 2012.

[49] Linda Rose Diane Ramey and Lisa Tyerman. *MTL material format (Lightwave, OBJ).* Copyright 1995 Alias—Wavefront, Inc., http://paulbourke.net/dataformats/mtl/, 4.2 edition, October 1995.

[50] Sanjoy K. Baruah and Enrico Bini. Partitioned scheduling of sporadic task systems: an ilp-based approach. In *In Proc. of the International Conference on Design and Architectures for Signal and Image Processing (DASIP 2008)*, 2008.

[51] Nvidia. *Nvidia Optix Ray Tracing Engine, Programming guide.* Nvidia Optix, version 2.1 edition, March 2011.

[52] Morgan McGuire. Computer graphics archive.

[53] Marc Levoy. The stanford 3d scanning repository.

[54] Ingo Wald. The utah 3d animation repository.

[55] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, 2009.