



TEC

Tecnológico de Costa Rica

ESCUELA DE COMPUTACIÓN

MAESTRÍA EN COMPUTACIÓN CON ÉNFASIS EN CIENCIAS DE LA
COMPUTACIÓN

Arquitectura para renderizador 3D tipo ray-cast SVO

Tesis para el grado de

Magister Scientiæ en Ciencias de la computación

Autor
Diego Valverde Garro

Asesor
Ing. Pablo Mendoza. Msc

febrero 2015

Abstract

The three-dimensional rendering of complex scenes is an open problem in the area of computer graphics. Improvements in specialized hardware rendering are not sufficient for the increasing complexity of the models. There is a growing interest from the industry and the computer graphics community to explore alternatives to traditional rendering scheme based on the method of projections and raster, with techniques such as Ray-Casting .

The naive algorithm of Ray-Casting consists in finding the intersection of each ray with each object in the three-dimensional scene without further optimization. This algorithm belongs to the family of algorithms which are known as embarrassingly parallel. This is because each ray is independent of all other rays, and in a system of brute force with potentially as many processors as rays, parallelism can achieve a very high level. One of the main problems of ray-casting is that all processors have to access a shared memory containing the objects in the scene, thus creating a bottleneck in the bus to access this memory. This data traffic congestion on the buses is a major limiting to the performance of parallel Ray-Casting systems.

This research proposes to combine the use of SVOs, sort-first type scene partitions, using Morton codes with depth first tree traversal, and a hierarchy of cache memories in order to obtain a significant decrease in the amount of accesses to the memory that stores the objects in the scene.

Keywords: SVO, trees octants, ray-casting, computer graphics, GPU.

Resumen

La renderización de escenas tridimensionales complejas es, al día de hoy, un problema existente en el área de gráficos por computadora. Las mejoras en el hardware especializado en renderización aún no dan abasto para la progresiva complejidad de los modelos. Existe un creciente interés por parte de la industria y la comunidad de gráficos por computadora en utilizar alternativas al esquema tradicional de rendering basado en el método de proyecciones ortogonales, con técnicas como el Ray-Casting.

El algoritmo de ingenuo de Ray-Casting consiste en hallar la intersección de cada rayo con cada objeto de la escena tridimensional sin ninguna optimización adicional. Este algoritmo pertenece a la familia de algoritmos que se conocen como embarazosamente paralelos. Esto se debe a que cada rayo es independiente de todos los otros rayos, y en un sistema de fuerza bruta con potencialmente tantos procesadores como rayos, el paralelismo puede alcanzar un nivel muy alto.

Uno de los problemas principales del Ray-Casting radica en que todos los procesadores tienen que acceder a una memoria compartida que contiene los objetos de la escena, generando un cuello de botella en el bus de acceso a esta memoria. Esta congestión de tráfico de datos en los buses es uno de principales limitantes del desempeño de los sistemas de Ray-Casting paralelos.

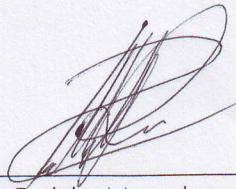
La presente investigación propone combinar el uso de SVOs, particiones en la escena de tipo sort-fist, empleo de códigos de Morton con recorrido del árbol por profundidad, y una jerarquía de memorias cache de con el fin de obtener una disminución significativa en los accesos a la memoria que almacena los objetos en la escena.

Palabras clave: SVO, arboles de octantes, Ray-Casting, gráficos por computadora, GPU.

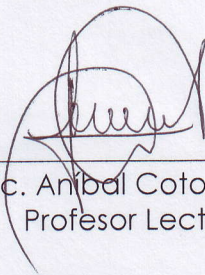
APROBACIÓN DE LA TESIS

“Arquitectura de Hardware para renderizador 3D tipo ray-cast SVO”

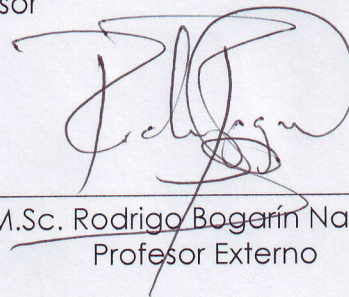
TRIBUNAL EXAMINADOR



M.Sc. Pablo Mendoza Ponce
Profesor Asesor



M.Sc. Anibal Coto Cortés
Profesor Lector



M.Sc. Rodrigo Bogarín Navarro
Profesor Externo



Dr. Roberto Cortés Morales
Coordinador del Programa
de Maestría en Computación

Enero, 2015

Índice general

1	Introducción	1
1.1	Planteamiento del Problema	5
1.2	Justificación del tema	7
2	Marco teórico	9
2.1	Historia	10
2.2	Render en gráficos por computadora	10
2.3	Rasterización	12
2.4	Pipelines Gáficos clásicos	12
2.5	Espacios tridimensionales	14
2.6	Ray Casting sobre superficies	15
2.7	Ray Casting vs. Técnicas convencionales	17
2.8	Árboles de Octantes Dispersos (SVO)	18
2.9	Filtros Trilineales	19
2.10	Códigos de Morton	20
2.11	Algoritmos para recorrer Árboles de Octantes	21
2.12	Intersección entre un rayo y un octante	23
2.13	Taxonomía de Render paralelo de Molnar/Cox	25
2.14	Memorias caché	27
2.15	Latencia	29
2.16	Ancho de Banda	30
2.17	Antecedentes	30
2.17.1	Implementaciones de Hardware de unidades de procesamiento gráfico	30
2.17.2	GigaVoxels	31
2.17.3	Ray tracing de SVO en FPGA	32
3	Propuesta de investigación	34
3.1	Sistema ingenuo	35

3.2	Propuesta general de arquitectura	36
3.3	Definición de métricas	37
3.4	Hipótesis	37
3.4.1	Pregunta de la Hipótesis	37
3.4.2	Hipótesis Nula	38
3.4.3	Hipótesis Alternativa	38
4	Metodología	39
4.1	Factores y niveles	40
4.2	Variables de respuesta	41
4.3	Recolección de datos	41
4.4	Análisis de varianza	42
5	Objetivos y alcance	44
5.1	Objetivos	45
5.1.1	Objetivo general	45
5.1.2	Objetivos específicos	45
5.2	Alcances y limitaciones	45
6	Arquitectura general del sistema simulado	47
6.1	Bloques e interacciones principales del sistema	48
6.2	Memorias caché	49
6.3	Gestor de tareas global	51
6.4	Unidades de ejecución	52
6.4.1	Propagación de tareas y recorrido de SVO	54
6.5	Estructuras de datos	57
6.6	Algoritmo de recorrido del árbol	59
7	Análisis de Resultados	63
7.1	Descripción de los experimentos realizados, recolección de datos y diseño de banco de pruebas	64
7.1.1	Simulador arquitectónico	64
7.1.2	Banco de pruebas	66
7.2	Resultados y aplicabilidad de ANOVA	69
7.3	Test de Kruskal-Wallis para análisis de varianza	69
7.4	Resultados de la variable de respuesta 1 - Total de accesos a la OsM	71
7.5	Resultados de la variable de respuesta 2 - Promedio de aciertos a la memoria caché	76
7.6	Análisis de tamaño de la ventana de partición sort first	76
7.7	Análisis del despeno de los cachés jerárquicos.	81
7.7.1	Análisis del tamaño total de la caché	85
7.7.2	Profundidad recomendada del SVO	86
8	Conclusiones	88
8.1	Conclusiones	89

9 Trabajo Futuro	92
9.1 Trabajo Futuro	93
Bibliografía	94
Acrónimos	96
Apéndices	98
.1 Tablas de resultados de experimentos	98

Índice de figuras

1.1	Aproximación del Utah Teapod mediante voxeles (el tamaño de los voxeles se exagera para efectos ilustrativos) Imagen generada con Blender [21]	3
2.1	Rostro de mono (Suzanne) aproximado como un mesh poligonal.	11
2.2	Pipeline gráfico clásico.	13
2.3	Ray casting	16
2.4	Pipeline gráfico para Ray Cast	16
2.5	Ejemplo de voxelización empleando SVO. Superior izquierda SVO con profundidad 7. Superior derecha: SVO con profundidad 8. Inferior izquierda: SVO con profundidad 9. Inferior derecha: El modelo original representado con triángulos	19
2.6	Empleo de filtros trilineales para suavizar un superficie aproximada con voxeles [4]	20
2.7	Interpolación trilineal como aplicación de dos interpolaciones bilineales	21
2.8	Códigos del Morton para un árbol de octantes	22
2.9	Intersección del rayo y dos de los ejes del <i>AABB</i>	24
2.10	Prueba de intersección. El rayo verde interseca la <i>AABB</i> , mientras que el rayo rojo no la interseca	25
2.11	Sistema de render paralelo	26
2.12	Optimizaciones de cachés según [5]	29
3.1	Sistema Ingenuo de referencia	36
3.2	Propuesta de la arquitectura general	36
6.1	Arquitectura del sistema	48
6.2	Líneas de cache para los tres primeros niveles de profundidad del SVO	50
6.3	Jerarquía de una memoria caché del sistema	51
6.4	Rayos coherentes suelen intersecar los mismos voxeles	52
6.5	barrido de píxeles dentro de una partición rectangular del plano de proyección	53

6.6	Ejemplo de asignación estática de tareas empleando secciones de plano de proyección para 16 núcleos de ejecución	53
6.7	Entradas y salidas a un GT	56
6.8	Estructura de datos para para un nodo del SVO	58
6.9	Estructura de datos para para un rayo	58
6.10	Distribución del trabajo de los octantes visitados del árbol en cada GT para 4 iteraciones del recorrido por profundidad, utilizando 8 GTs. . . .	61
7.1	Flujo de ejecución de herramienta de simulación	65
7.2	Ejemplo de escenas voxelizadas utilizadas en las pruebas. Superior izquierda : Happy Buddah, Superior derecha: Utah teapod. Inferior izquierda: Standord dragon. Inferior derecha: Standford Bunny.	66
7.3	Resultados de ANOVA	70
7.4	Resultados variable de respuesta 1. Comparación de todas las escenas .	73
7.5	Resultados variable de salida 1	74
7.6	Resultados variable de respuesta 1 (continuación)	75
7.7	Resultados variable de salida 2	77
7.8	Promedio de lecturas a OsM para todas las escenas a distintas resoluciones	79
7.9	Promedio de Hits a caché para todas las escenas a distintas resoluciones	80
7.10	Promedio de aciertos al los diferentes niveles de caché para un SVO de profundidad (promedio por escena para todas las resoluciones).	81
7.11	Promedio de aciertos al los diferentes niveles de caché para un SVO de profundidad 9 (promedio de Hits por resolución para todas las escenas).	82
7.12	Tasa de aciertos al los diferentes niveles de caché para un SVO de profundidad 10	83
7.13	Número de accesos al SVO por nodo. Los nodos superiores (direcciones más bajas) son accedidos más veces	84
7.14	Promedio de Hits al caché y tamaños en MB de caché según profundidad del SvO por resolución (promedio para todas las escenas)	85

Introducción

“The sky above the port was the color of television, tuned to a dead channel.”.

William Gibson, *Neuromancer*

La renderización de escenas tridimensionales complejas es, al día de hoy, un problema existente en el área de gráficos por computadora. Las mejoras en el hardware especializado en renderización aún no dan abasto para la progresiva complejidad de los modelos. Existe un creciente interés por parte de la industria y la comunidad de gráficos por computadora en utilizar alternativas al esquema tradicional de rendering basado en el método de proyecciones ortogonales, con técnicas como el ray-casting. El algoritmo ingenuo de ray-casting consiste en hallar la intersección de cada rayo con cada objeto de la escena tridimensional sin ninguna optimización adicional. Este algoritmo pertenece a la familia de algoritmos que se conocen como embarzosamente paralelos. Esto se debe a que cada rayo es independiente de todos los otros rayos, y en un sistema de fuerza bruta con potencialmente tantos procesadores como rayos, el paralelismo puede alcanzar un nivel muy alto.

Uno de los problemas principales del ray-casting radica en que todos los procesadores tienen que acceder a una memoria compartida que contiene los objetos de la escena, generando un cuello de botella en el bus de acceso a esta memoria. Esta congestión de tráfico de datos en los buses es uno de principales limitantes del desempeño de los sistemas de Ray Casting paralelos.

En el ray-casting ingenuo cada rayo tiene que ser intersecado contra cada objeto en la escena. Un ray casting más inteligente puede utilizar una estructura de datos de partición espacial para reducir el número de pruebas de intersección rayo / objeto a un conjunto más pequeño.

Una de las estructuras de datos de partición espacial más populares es el árbol de octantes. Los arboles de octantes dividen recursivamente el espacio en ocho particiones llamadas octantes, esto con el fin de reducir el número de intersecciones rayo / objeto. A menudo, cuando se utiliza una estructura de datos de partición espacial, se almacena una lista de objetos en los nodos hoja. Los Spare Voxel Octree (SVO) son estructuras de datos de partición espacial en las que la mayoría de los octantes están vacíos y en

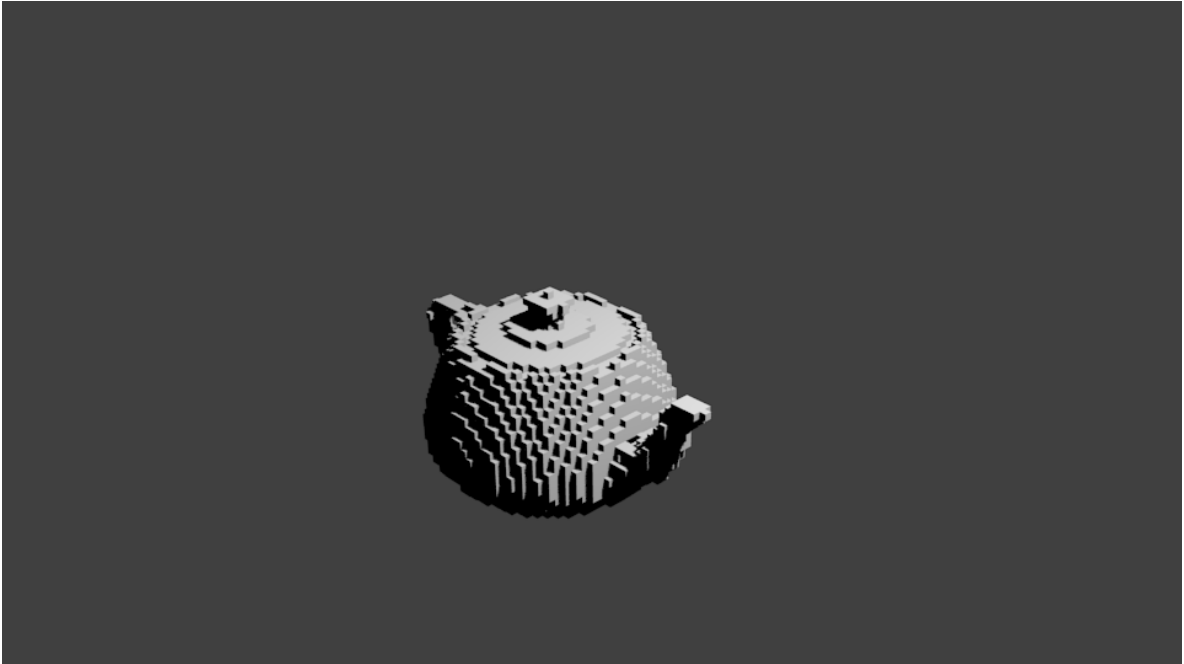


Figura 1.1: Aproximación del Utah Teapot mediante voxeles (el tamaño de los voxeles se exagera para efectos ilustrativos) Imagen generada con Blender [21]

las que los nodos hoja no guardan listas de objetos, en lugar de esto, cada nodo hoja es ya sea un espacio vacío o un voxel. Un voxel o píxel volumétrico, es una primitiva geométrica que se emplea para representar superficies aproximándolas con pequeños cubos sólidos. La figura 1.1 muestra una aproximación del Utah Teapot con voxels.

A este punto ya podemos comenzar a vislumbrar como los SVO reducen el número de accesos a la memoria compartida que contiene los objetos de la escena. En primer lugar, gracias a la estructura de partición espacial, ya no es necesario evaluar la intersección de cada rayo contra cada objeto de la escena, si no que en lugar de esto la búsqueda se vuelve jerárquica. En segundo lugar, cuando el rayo finalmente encuentra una intersección con un nodo hoja, ya no es necesario acceder a la memoria de la escena para obtener la lista de primitivas, ya que la hoja del árbol ya es en sí misma la primitiva, para este caso el voxel.

Un voxel puede tener un valor de color. Para un nodo interno, podemos tomar el color

promedio de todos sus nodos hijos, y adjuntarlo a este nodo. Esto significa que la escena almacenada en el SVO se puede representar a distintas resoluciones, dependiendo del nivel al que llegue el recorrido en profundidad del árbol. Esto ayuda a reducir aún más los accesos a la memoria, ya que si un rayo en su trayectoria se encuentra ya muy alejado de la cámara, entonces no es necesario seguir recorriendo el árbol hasta llegar a un nodo hoja, si no que se usa la representación de uno de los nodos internos ya que no queremos dar gran detalle a los objetos que están muy alejados del observador. Tomando en cuenta esta última observación [2] sugiere ir un nivel más allá, y almacenar en la memoria del GPU únicamente las partes de la escena cercanas a la cámara, y transmitir el resto de ramas del árbol de octantes conforme la cámara avance hacia esas áreas. Esto es equivalente a decir que en un momento del tiempo no queremos almacenar los objetos alejados a la cámara, disminuyendo de esta manera los accesos, el consumo y las consultas a la memoria del GPU.

Los Spare Voxel Octrees (SVO) han tenido mucha atención en los últimos años, habiendo a la fecha varias implementaciones de SVO en software, siendo una de las más notorias la de Unreal Engine 4 [23], basada en la tesis doctoral de Crassin et al. [2] sobre “Giga Voxels”.

Uno de los inconvenientes de la estructura de datos de partición espacial tipo Octree es que en su representación ingenua, requiere que cada nodo interno del árbol almacene ocho punteros a sus nodos hijos. Una forma de solventar este inconveniente es emplear una estructura de datos “libre de punteros”. Existen varias representaciones de árboles de octantes libres de punteros como por ejemplo las curvas de Hilbert o los códigos de Morton (Curvas Z). Los códigos de Morton [14] se han elegido para esta investigación ya que presentan varias ventajas que ayudan a minimizar los accesos a la memoria del GPU. Una de las ventajas principales es que los códigos de Morton constituyen la llave de una función *hash* perfecta hacia cada nodo del árbol. Esta llave puede ser generada por los procesadores en tiempo de ejecución, codificando tanto la profundidad actual

del árbol así como sirviendo de apuntador a una posición en la memoria del GPU que almacena información acerca de uno de los 3 tipos posibles de nodo a saber: 1) nodo voxel, 2) nodo no vacío (tiene al menos un descendiente que es un voxel), 3) nodo vacío (el nodo no es un voxel y no tiene ningún descendiente).

Una tercera optimización posible es considerar trayectorias similares de rayos hacia píxeles vecinos en la pantalla. De este modo cuando se traza un rayo desde la cámara hacia un píxel, los voxels correspondientes al objeto intersecado y otros objetos cercanos en la escena se almacenan en una o más memorias caché. Puesto que las trayectorias de estos rayos serán similares, es probable que se intersecten los mismos objetos, por lo que este puede llegar tener tasas de acierto altas en las memorias caché [22].

La presente investigación consiste en proponer una arquitectura de Hardware para un renderizador tipo ray-cast que minimice los accesos a la memoria que almacena los objetos de la escena. La hipótesis es esencialmente que al combinar las optimizaciones antes mencionadas: empleo de SVOs, particiones en la escena de tipo sort-fist, empleo de códigos de Morton con recorrido del árbol por profundidad, y una jerarquía de memorias cache, se obtendrá una alta tasa de aciertos alta en las memorias cache, y una disminución significativa en los accesos a la memoria que almacena los objetos en la escena.

1.1 Planteamiento del Problema

La renderización de escenas tridimensionales complejas es al día de hoy un problema existente en el área de gráficos por computadora. Las mejoras en el hardware especializado en renderización aún no dan abasto para la progresiva complejidad de los modelos. Existe un creciente interés por parte de la industria y la comunidad de gráficos por computadora en utilizar alternativas al esquema tradicional de rendering basado

en el método de proyecciones ortogonales, con técnicas como el trazado de rayos, cuyo modelo de transporte y propagación de la luz proporciona un mayor realismo en las escenas. No obstante, estas técnicas de la familia del trazado de rayos presentan un reto para el desempeño de los sistemas computacionales de renderización.

Para solventar esta situación, investigaciones recientes se han enfocado en el uso de formas alternativas de representar los objetos geométricos en las escenas, para así mejorar el desempeño del trazado de rayos y así como de otras técnicas de graficación. Técnicas tales como ray-casting, uso de superficies algebraicas de alto orden, B-Splines, superficies de Bezier, B-Splines racionales no uniformes (NURBS), triángulos paramétricos con curvatura, Voxels, etc. se han presentado recientemente como una alternativa viable a la teselación y división tradicional de los modelos en primitivas planas como triángulos o cuadriláteros.

Existe aquí un nicho por explorar con respecto al diseño de arquitecturas de hardware especializadas, que optimicen el uso de estas representaciones geométricas como alternativas al triángulo y que empleen modelos de pipelines gráficos distintos al pipeline clásico de proyección y raster.

La presente investigación se enfoca la técnica de conocida como ray casting para arquitecturas de múltiples núcleos que comparten una memoria de almacenamiento de los objetos de la escena. Uno de los mayores problemas con las arquitecturas multi-núcleo es el cuello de botella del bus de memoria. Para la técnica de renderizado de Ray-Casting, cada núcleo necesita datos de la geometría de la escena para las pruebas de intersección de los rayos y la primitiva. Adicionalmente, se requiere de otra información para textura y sombreado.

El algoritmo de ingenuo ray-casting consiste en hallar la intersección de cada rayo con cada objeto de la escena tridimensional sin ninguna optimización adicional. Este algoritmo pertenece a la familia de algoritmos que conocen como embarazosamente paralelos. Esto se debe a que cada rayo es independiente de todos los otros rayos, y en un siste-

ma de fuerza bruta con potencialmente tantos procesadores como rayos el paralelismo puede alcanzar un nivel muy alto.

Uno de los problemas principales de este sistema radica en que todos los procesadores tienen que acceder a una memoria compartida que contiene los objetos de la escena, generando un cuello de botella en el bus de acceso a esta memoria. Esta congestión de tráfico de datos en los buses es uno de principales limitantes del desempeño de los sistemas de Ray Casting paralelos.

La presente investigación consiste en proponer una arquitectura de Hardware para un renderizador tipo ray-cast que minimice los accesos a la memoria que almacena los objetos de la escena mediante el uso de una serie de optimizaciones como se detalla más adelante en el presente documento.

1.2 Justificación del tema

El manejo de superficies tridimensionales juega un papel crítico en áreas como diseño y manufactura de partes para la industria naval, aeronáutica, automotriz, diseño de utensilios, arquitectura, así como también en diseño de modelos para la industria cinematográfica y del entretenimiento, por mencionar solo algunos ejemplos. La necesidad de representaciones precisas de los modelos así como los requerimientos a nivel estético, demandan que los sistemas computacionales para gráficos en tercera dimensión sean cada vez capaces de representar los modelos con mayor precisión.

Tradicionalmente los modelos de superficies tridimensionales se suelen descomponer en una serie de primitivas más sencillas, usualmente triángulos, para facilitarle al hardware la labor de proyectar los modelos en el medio final de visualización. No obstante, conforme aumenta la complejidad de los modelos las mejoras en el hardware especializado en renderización no son suficientes para la velocidad y los niveles de realismo que

demandan las aplicaciones actuales. En respuesta a esto una serie de implementaciones alternativas de hardware han surgido en los últimos años. Uno de los esquemas más recientes son los sistemas basados en ray-casting. Estos sistemas se han empleado recientemente en productos de software, sin embargo su potencial en hardware no ha sido del todo aprovechado. Una de las razones de esto está relacionada al cuello de botella en los accesos a la memoria que almacena los objetos de la escena. Es así como se puede ver la gran importancia de investigar sobre nuevas arquitecturas de hardware que mejoren el desempeño de los accesos a memoria de los sistemas de render de Hardware tipo ray-casting.

Marco teórico

“If I have seen further than others, it is by standing upon the shoulders of giants.”.

Isaac Newton

2.1 Historia

Durante la última década, el rendimiento de las arquitecturas gráficas ha aumentado a una velocidad asombrosa. Este rendimiento mejorado se debe a factores tales como el aumento de la cantidad de transistores superior a la ley de Moore, y el uso de múltiples núcleos para explotar el paralelismo. Como ejemplo de esto, en sólo diez años, la tecnología de GPUs NVIDIA evolucionó a partir de 25 millones de transistores en un solo núcleo en 1999, hasta 3 millones de transistores con 512 núcleos en 2009 [15]. Además, las GPU modernas se han vuelto más flexibles, pasando de ser máquinas diseñadas para llevar a cabo una tarea específica, que era la representación del triángulo, a pipelines completamente programables capaces de realizar una amplia gama de algoritmos. En el año 2001, el primer procesador de vértices programables, la NVIDIA GeForce 3 fue introducido. Más tarde, en 2002 ATI Radeon ATI introdujo el 9700 que contó con un procesador de pixel-fragmento programable [15]. Gracias a la naturaleza de streaming de las GPU modernas, ahora es posible realizar cálculos de propósito general para una amplia gama de aplicaciones que explotan alto paralelismo de estas máquinas.

2.2 Render en gráficos por computadora

En los gráficos por computadora, el 3D render es un proceso en el que se genera una imagen 2D a partir de un modelo en el espacio 3D. El 3D render tiene aplicaciones en videojuegos, arquitectura, simuladores, películas, televisión, etc. Ya que el aspecto visual de un objeto 3D depende en gran medida del exterior del objeto, en gráficos por computadora los modelos 3D son a menudo modelos tipo Shell (o de cascarón), lo que significa que representan la superficie del objeto, pero no su volumen como una entidad sólida. Por lo tanto, los modelos 3D son generalmente representaciones matemáticas de superficies tridimensionales. A menudo, estos modelos son representados usando una colección de puntos en el espacio 3D. Estos puntos pueden ser conectados

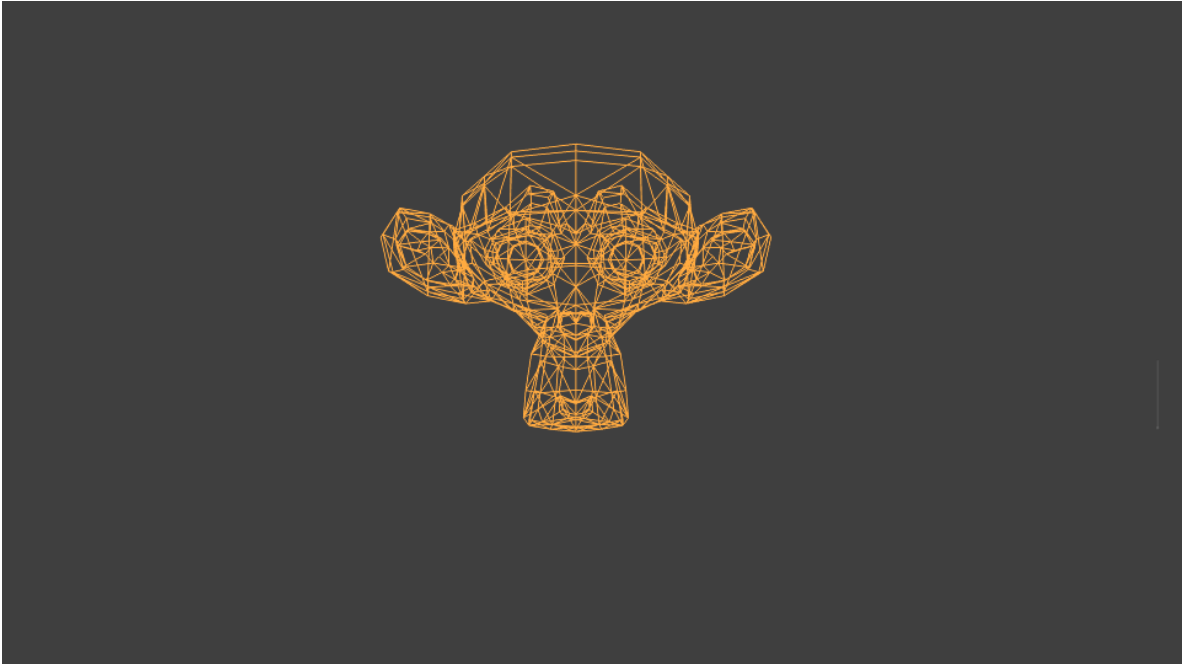


Figura 2.1: Rostro de mono (Suzanne) aproximado como un mesh poligonal.

mediante entidades geométricas como líneas, superficies curvas, triángulos, cuadriláteros, cubos, etc. Generalmente, las representaciones matemáticas puras de superficies, tales como esferas, conos, etc. son aproximadas utilizando su representación poligonal correspondiente en lo que se llama una malla o Mesh.

El *render* basado en el uso de mallas de polígonos es una técnica ampliamente adoptada para representar superficies 3D en aplicaciones gráficas. El proceso de creación de representaciones poligonales de superficies 3D genéricas se conoce como teselación. La teselación se suele realizar mediante software especializado o hecho a mano por expertos diseñadores gráficos en 3D usando una variedad de herramientas de CAD. Un conjunto de *meshes* se pueden representar colectivamente en lo que se llama una escena 3D. Algunas GPUs modernas son compatibles con teselación de hardware, sin embargo este no es el tema central del presente trabajo.

2.3 Rasterización

La rasterización es actualmente la técnica más popular para renderizar escenas 3D. Consiste en tomar una representación 3D de la escena y convertirla una imagen raster 2D formada por píxeles. Esto se consigue normalmente por alguna variante del algoritmo de escaneo de línea presentado por Sutherland et al. [24] en 1968. La rasterización es el proceso de mapeo de la geometría de la escena a los píxeles, sin embargo la rasterización no prescribe una forma determinada para calcular el color de los píxeles. Esta tarea se programa, generalmente empleando programas especiales llamados Shaders. Tener la capacidad de programar *shaders* personalizados no sólo permite calcular el color del píxel de acuerdo a un modelo físico de transporte de la luz, sino que también permite aplicar efectos artísticos muy creativos en la imagen final.

2.4 Pipelines Gáficos clásicos

Por lo general, los gráficos 3D se crean mediante un proceso de múltiples pasos llamado pipeline gráfico 3D. Este pipeline gráfico se puede dividir conceptualmente en varias etapas o Pipestages. Actualmente no existe una convención con respecto a las etapas del pipeline, el número de pasos de cada Pipestage o el orden de las fases del Pipestage. Además, el pipeline evoluciona constantemente. Los cambios en el pipeline gráfico suelen ser impulsados por empresas privadas, a través de APIs gráficas (interfaces de programación de aplicaciones), como OpenGL y Direct3D. Dicho esto, todavía se podría generalizar el pipeline gráfico en los siguientes pasos:

- Etapa de procesamiento de vértices.
- Etapa de ensamblaje de polígonos.
- Etapa de procesamiento de píxeles.
- Etapa de procesamiento de Frame Buffer.

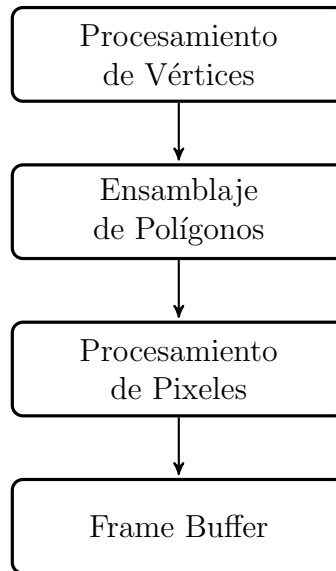


Figura 2.2: Pipeline gráfico clásico.

La figura 2.2 presenta las cuatro etapas del pipeline general mencionados anteriormente. Sin sumergirse en los detalles, la operación del pipeline se puede resumir de la siguiente manera:

Suponiendo que los Meshes utilizan primitivas triangulares, el pipeline inicia cuando las aplicaciones que se ejecutan en la CPU envían los Buffers de vértices a la GPU. Los Buffers de vértices contienen una lista de los vértices de cada triángulo en la escena. Usando transformaciones matemáticas, estos vértices son desplazados, escalados, y finalmente proyectados en un espacio de dos dimensiones. Estas transformaciones tienen lugar durante la etapa de procesamiento de vértices. Una vez que los vértices se transforman, los triángulos son ensamblados a partir de estos vértices en la fase de ensamblaje. A continuación, los píxeles pertenecientes a cada triángulo son determinados en la etapa de procesamiento de píxeles. Típicamente la técnica de rasterización se utiliza para determinar estos píxeles. Desde este punto en adelante las operaciones se llevan a cabo sobre cada píxel. Seguidamente se ejecutan una serie de cálculos por píxel para obtener el resultado visual deseado. Estos cálculos incluyen el mapeo de textura, sombreado de píxeles, etc. Esta fase del pipeline suele ser programable. Por último, durante la etapa

de Frame Buffer, se ejecutan las operaciones para determinar la visibilidad triángulo y se presenta el color final de los píxeles en un monitor de computadora u otro dispositivo.

2.5 Espacios tridimensionales

Diferentes sistemas de coordenadas cartesianas se utilizan en diferentes etapas del pipeline gráfico. Estos sistemas de coordenadas se conocen a veces a como espacios de coordenadas. Los objetos de una escena 3D se transforman normalmente a través de cinco espacios a lo largo del pipeline gráfico como se resume a continuación.

Espacio de Modelo (Model Space): Cada modelo se encuentra en su propio sistema de coordenadas. Esto significa que el origen del sistema de coordenadas del modelo es algún punto del propio modelo. La razón de esto es que por lo general para proyectos grandes, los objetos de la escena son creados por diferentes artistas gráficos, cada uno con su propio espacio de coordenadas para el modelo.

Espacio del Mundo (World Space): Una escena 3D se compone de muchos modelos. Puesto que cada modelo potencialmente puede tener su propio sistema de coordenadas, el espacio mundo unifica todos los sistemas de coordenadas de los modelos en un único sistema de coordenadas global.

Espacio de vista (View Space): La cámara se sitúa en algún punto del sistema de coordenadas del mundo. Luego una matriz de transformación se utiliza de tal manera que el origen del espacio de coordenadas mundiales coincida con la posición de la cámara. Seguidamente, una matriz de proyección se aplica de tal manera que la escena se proyecte delante de la cámara. De esta manera se crea el volumen de vista. Este volumen de vista puede ser rectangular para proyección paralela o piramidal para proyecciones con perspectiva. El volumen de vista piramidal se llama a menudo el Viewing Frustum. El Frustum no es una pirámide infinita, sino que está acotado por dos planos llamados el plano lejano y el plano cercano, que son respectivamente el más lejano y el

más cercano a la cámara. El propósito del Frustum es limitar la visibilidad de la escena para que no se analicen a objetos fuera del que no estén dentro del rango de visión de la cámara.

Espacio de recorte (Clipping Space): El Frustum del View space es transformado en un cubo unitario, con las coordenadas x y y normalizadas entre -1 y 1 , y la coordenada z entre 0 y 1 .

Screen Space: La imagen 3D se transforma a coordenadas 2D en la pantalla. En este punto la transformación de la escena a píxeles (rasterización) aun no a ocurrido.

2.6 Ray Casting sobre superficies

Como se mencionó en la sección 2.4 el pipeline gráfico puede dividirse conceptualmente en 4 etapas. Es importante recordar que este concepto de pipeline es sólo una generalización y que diferentes técnicas pueden ser utilizadas como posibles implementaciones para cada etapa individual. Por ejemplo, la mayoría de las GPUs disponibles comercialmente usan la técnica de rasterización para determinar los píxeles en la etapa de procesamiento de píxeles, pero esta no es la única manera de implementar la etapa de procesamiento de píxeles.

El Ray Casting es una alternativa a la rasterización que permite resolver el procesamiento de píxeles de una manera diferente. El algoritmo de Ray Casting fue descrito por primera vez por Appel [1] en 1968 y más tarde fue propuesto como un método para representar sólidos en 1982 [19]. A grandes rasgos, el algoritmo consiste en generar rayos desde el observador hasta cada píxel de una pantalla virtual y luego determinar el color del objeto visible a través de cada rayo.

El primer paso consiste en emitir un rayo, esta es una trayectoria en el espacio desde un ojo imaginario a través de cada píxel en una pantalla virtual. Se realiza una prueba de intersección entre cada rayo y los objetos en la escena. Una vez que la intersección

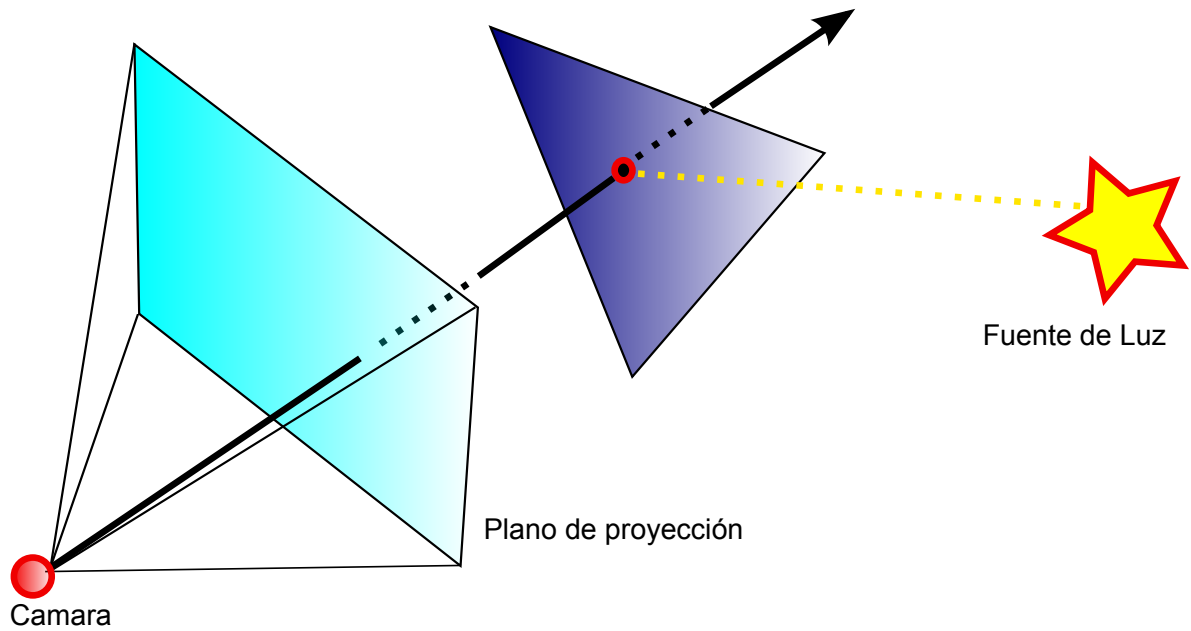


Figura 2.3: Ray casting

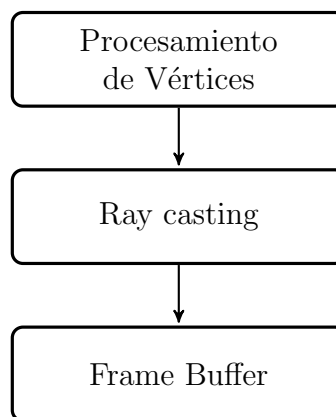


Figura 2.4: Pipeline gráfico para Ray Cast

más cercana para un rayo dado ha sido identificada, el algoritmo estima la luz incidente en el punto de intersección, y considerando las propiedades del material del objeto el color final del píxel correspondiente a ese rayo es calculado.

Si modificamos el pipeline gráfico de la sección 2.4 para introducir el Ray Casting, varias diferencias se ponen de manifiesto. Como se muestra en la figura 2.4, usando Ray Casting, la etapa de ensamblaje en el pipeline gráfico de la figura 2.2 ya no es necesaria, sino que esta también se lleva a cabo como parte del Ray Casting. Además no es necesario proyectar cada vértice individual al plano de proyección ya que esto lo hace el Ray Casting para cada píxel. Es importante notar que las etapas de procesamiento de vértices y Frame Buffer todavía están presentes. También el Ray Casting asume un espacio de coordenadas de mundo como se explica en la sección 2.5 , sin embargo, las transformaciones en el espacio de vista o espacio de recorte (Clipping Space) ya no son necesarias (se hacen implícitamente por el algoritmo de Ray Casting). Además, la determinación de la visibilidad explícita ya no es necesaria, ya que se lanza cada rayo y únicamente las intersecciones entre los rayos y los objetos que están más cerca de la cámara se convertirán en píxeles en la imagen final.

2.7 Ray Casting vs. Técnicas convencionales

Walds y Slusallek [25] presentan una serie de beneficios del trazado de rayos versus la rasterización. Dado que el trazado de rayos es una generalización del Ray Casting los siguientes beneficios del trazado de rayos también aplican para un sistema de Ray Casting.

Oclusión y complejidad logarítmica: Ray Casting cuenta con un mecanismo intrínseco de oclusión como se mencionó en la sección anterior. También usando una estructura simple de datos de búsqueda como los Octree, la búsqueda acaba una vez que la visibilidad se ha determinado.

Shading eficiente: Usando Ray Casting, los cálculos de Shading sólo se realizan una vez que la visibilidad se ha determinado. Esto significa que no hay cálculos redundantes para la geometría invisible. Recordemos de la sección 5.4 que en el Rendering pipeline tradicional, los cálculos de visibilidad se realizan durante la etapa de Frame Buffer, que es después de que los cálculos por píxel (mapeo de textura, sombreado de píxeles, etc) han sido ya realizados durante la etapa de procesamiento de píxeles.

Escalabilidad paralela: Ray Casting es conocido por ser embarazosamente paralelo, por ello una aplicación multiprocesador es sencilla de implementar.

Coherencia: Dado que sólo los rayos primarios son generados, Ray Casting, a diferencia de Ray Tracing presenta un alto grado de coherencia espacial.

2.8 Árboles de Octantes Dispersos (SVO)

Los Árboles de Octantes dispersos (SVO por sus siglas en inglés Spare Voxel Octrees) son estructuras de datos de partición espacial jerárquica. Al igual que los árboles de octantes comunes, los SVO dividen recursivamente el espacio en 8 particiones cúbicas llamadas octantes.

Para efectos de esta tesis se asume que los octantes son cúbicos y que el largo de las aristas del los cubos padre siempre es el doble del largo de las aristas de sus hijos inmediatos.

Los SVO se utilizan para almacenar modelos en los que la mayoría de los octantes están vacíos, por lo cual se dice que el espacio está "disperso". En la representación de modelos tridimensionales este sería el caso, ya que los voxeles existen únicamente en la superficie de los objetos.

La profundidad de un SVO está directamente relacionada al nivel de detalle de la superficie tridimensional que el árbol aproxima. En la figura 2.5 se aprecia como conforme

aumenta la profundidad del SVO el nivel de detalle de la figura mejora cada vez más. Es importante recordar que después de una cierta profundidad del SVO, el nivel de detalle adicional que se obtiene se vuelve incipiente. Esto generalmente ocurre cuando los voxeles en la hojas son tan pequeños o más pequeños que un píxel en la imagen 2D proyectada.

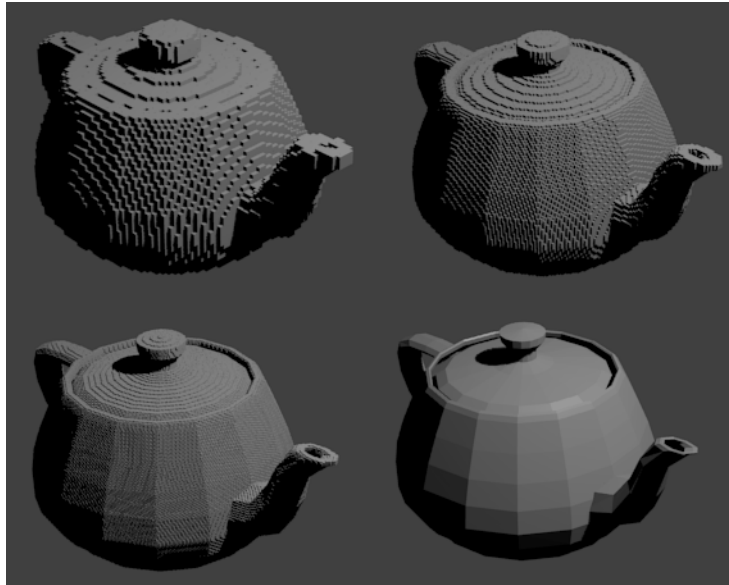


Figura 2.5: Ejemplo de voxelización empleando SVO. Superior izquierda SVO con profundidad 7. Superior derecha: SVO con profundidad 8. Inferior izquierda: SVO con profundidad 9. Inferior derecha: El modelo original representado con triángulos

2.9 Filtros Trilineales

Observando nuevamente la figura 2.5 podemos notar que a menos de que se recorra el SVO hasta una profundidad grande, se siguen apreciando las aristas de los voxeles, lo cual da un efecto visual desagradable en las partes de la imagen donde los voxeles sean considerablemente más grandes que un píxel.

Una solución a este problema (sin necesitar incrementar la profundidad del SVO) es el uso de la interpolación trilineal. Mediante la interpolación trilineal, es posible colorear píxeles adicionales a lo largo de la superficie voxelizada para da un efecto de suavidad

como se aprecia en la figura 2.6.

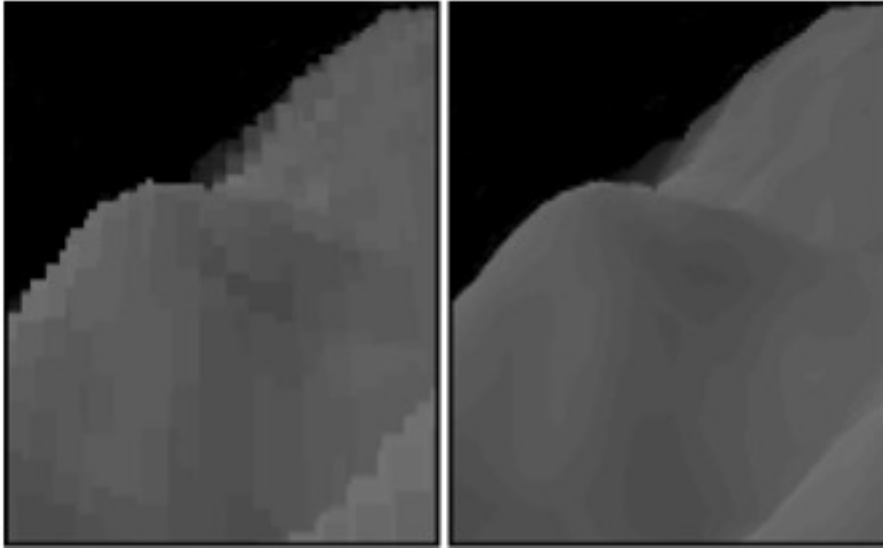


Figura 2.6: Empleo de filtros trilineales para suavizar un superficie aproximada con voxeles [4]

La interpolación trilineal es en esencia una extensión de la técnica de interpolación bilineal.

Se puede observar de la figura 2.7 como la interpolación lineal de dos interpolaciones bilineales corresponde a una interpolación trilineal. En el caso de la figura 2.7 se hace una interpolación para la cara frontal del voxel, representada por los vértices V_0 , V_1 , V_2 y V_3 y luego una segunda interpolación para la cara posterior, representada por los vértices V_4 , V_5 , V_6 y V_7 . Dado que para esta investigación cada octante del SVO es un cubo, resultaría relativamente sencillo calcular estas interpolaciones en el hardware.

2.10 Códigos de Morton

Los códigos de Morton son un codificación que permite generar un índice único para cada nodo del árbol del octantes [14]. En esta codificación el octante raíz tiene el índice 1, y el índice de cada nodo hijo es la concatenación del índice de su padre con la ubicación de su octante, codificado como los 3 bits menos significativos del número. Una de

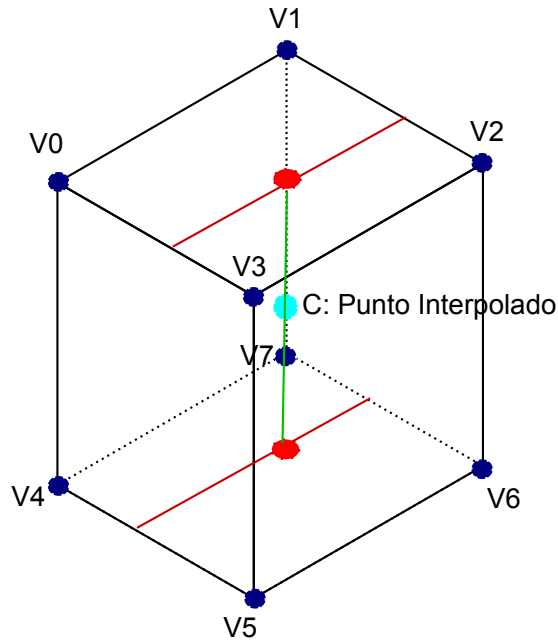


Figura 2.7: Interpolación trilineal como aplicación de dos interpolaciones bilineales

las ventajas de los códigos de Morton es su orden jerárquico, ya que es posible crear un único índice para cada nodo, preservando en cada índice la jerarquía de árbol. Otra de las ventajas de los códigos de Morton es que ya que cada código contiene la información acerca de la profundidad del octante actual en el árbol, entonces es posible obtener las coordenadas del centro del octante actual directamente a partir de este código y de las coordenadas del octante raíz.

La figura 2.8 ilustra la codificación de Morton para los primeros dos niveles de profundidad, en esta figura los 8 hijos de nodo raíz (1) son 1000, 1001, 1010, 1011, 1100, 1101, 1110 y 1111 como se muestra en la figura 2.8.

2.11 Algoritmos para recorrer Árboles de Octantes

La literatura describe una gran variedad de algoritmos para recorrer árboles de octantes. Distintos algoritmos de recorrido tienen diferentes complejidades, y pueden llegar a un rendimiento muy distinto en una arquitectura de computadora determina-

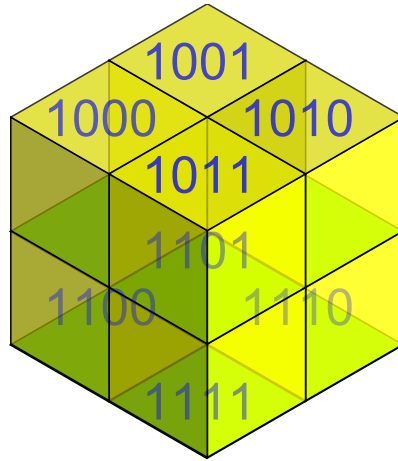


Figura 2.8: Códigos del Morton para un árbol de octantes

da. Un atributo importante que se puede emplear para categorizar los algoritmos de recorrido es el uso de la pila. Una pila es útil para hacer el seguimiento de la trayectoria tomada cuando el árbol se recorre, pero puede ser perjudicial en ciertas arquitecturas. Las siguientes son descripciones de algunos de los métodos más comunes.

- re-inicio : Este un algoritmo de recorrido sin pila (stackless). El algoritmo comienza seleccionando un punto de partida. Este punto de partida suele ser el origen de los rayos. El árbol es atravesado desplazando el punto a lo largo del rayo hasta que se encuentra el primer octante en contener el punto. Si el octante representado por el nodo está vacío, el punto se mueve hasta donde el rayo sale del octante. Luego el algoritmo se reinicia desde el nuevo punto. En otras palabras, el algoritmo atraviesa un octante a la vez, y tiene que descender el árbol desde el nodo raíz en cada iteración. Esto se repite hasta que se encuentra una intersección entre el rayo y un voxel o hasta que el rayo sale del voxel representado por el nodo raíz.
- Backtracking: Los algoritmos de backtracking son una optimización sencilla sobre los algoritmos de re-inicio. Aprovechan el hecho de que podemos hacer un seguimiento del último nodo padre de todos los nodos atravesados por el rayo.

Debido a que el nodo padre puede estar cerca de la raíz, puede que para algunas iteraciones la ganancia sea mínima. En otras palabras, Sólo elimina los pasos de recorrido cerca de la parte superior del árbol, que son los que están compartidas más veces por los vecinos de rayos primarios.

- FullStack: Un algoritmo de recorrido de árbol puede utilizar una pila para realizar un seguimiento de los nodos visitados conforme se desciende en el árbol. Los algoritmos que utilizan una pila son similares a los algoritmos de re-inicio, pero empujan el estado del recorrido de cada nodo a una pila. Si se encuentra un octante vacío este no se apila.
- ShortStack: Similar al enfoque anterior pero con la diferencia de que la pila solo almacena el estado de los últimos N nodos visitados. Si la pila se llena, entonces el algoritmo hace un re-inicio. Nótese que un FullStack es un caso particular de ShortStack donde N es igual a la profundidad del árbol.

2.12 Intersección entre un rayo y un octante

Calcular la intersección entre un rayo y un $AABB$ es realidad conceptualmente bastante sencillo. Para efectos de ilustrar el algoritmo de intersección del rayo con los $AABBs$ vamos analizar inicialmente un escenario en 2D. Las ecuaciones para un recta en 2D se pueden expresar como la ecuación (2.1) o en su forma paramétrica como la ecuación (2.2).

$$y = m * x + b \tag{2.1}$$

$$O = R * t \tag{2.2}$$

Considere la figura 2.9, dada la ecuación (2.1) la línea que representa el límite inferior

del cuadrado es simplemente:

$$y = B0_x. \quad (2.3)$$

El valor de t en (2.2) para la intersección con este eje viene dada por la ecuación 2.4, que es simplemente igualar (2.3) con (2.2) y despejar t .

$$t0_x = (B0_x - O_x)/R_x \quad (2.4)$$

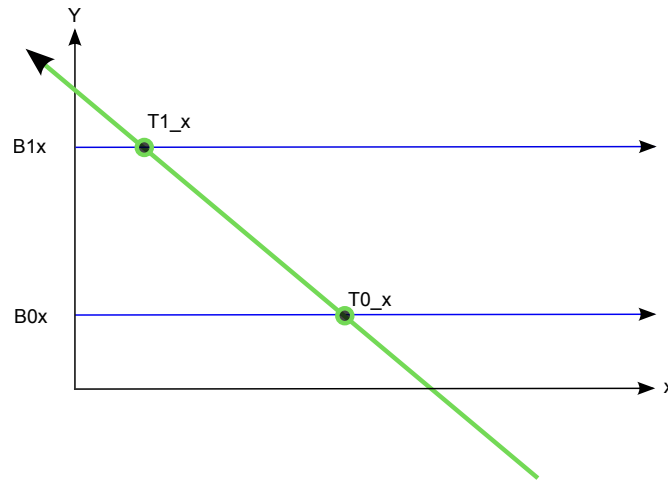


Figura 2.9: Intersección del rayo y dos de los ejes del $AABB$

Aplicando esta misma estrategia al resto de los ejes se llega a que:

$$t0_x = (B0_x - O_x)/R_x \quad (2.5)$$

$$t1_x = (B1_x - O_x)/R_x \quad (2.6)$$

$$t0_y = (B0_y - O_y)/R_y \quad (2.7)$$

$$t1_y = (B1_y - O_y)/R_y \quad (2.8)$$

$$t0_z = (B0_z - O_z)/R_z \quad (2.9)$$

$$t1_z = (B1_z - O_z)/R_z \quad (2.10)$$

El conjunto de estos seis valores de t indica donde el rayo intersecta los planos del $AABB$ para los ejes x , y y z .

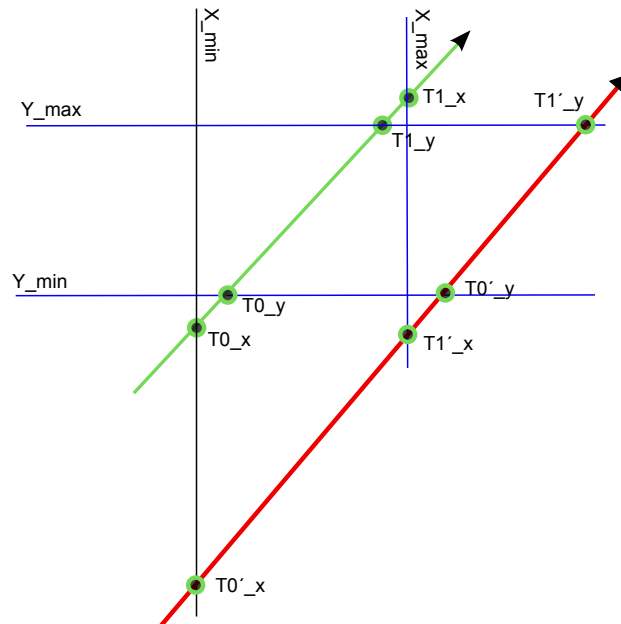


Figura 2.10: Prueba de intersección. El rayo verde intersecta la $AABB$, mientras que el rayo rojo no la intersecta

Una vez calculados los distintos t para las intersecciones del rayo con cada uno de los planos que definen el $AABB$, es necesario evaluar si dados estos valores de t , el rayo efectivamente tiene una intersección con el $AABB$. La figura 2.10 ilustra la lógica para evaluar estas intersecciones, es fácil ver de esta figura que el rayo rojo está por fuera de la caja debido a que $T0'_y > T1'_x$, de igual manera el rayo estaría por fuera de la caja si $T0_x > T1_y$.

2.13 Taxonomía de Render paralelo de Molnar/Cox

Molnar et al. [12] introdujo una clasificación útil para el Render paralelo. Se trata de un modelo conceptual para clasificar el Render en tres categorías principales que ayuda a distribuir la carga en los sistemas de Render paralelos convencionales. El pipeline gráfico estándar tipo Feedforward de la figura 2.4 se puede generalizar para la

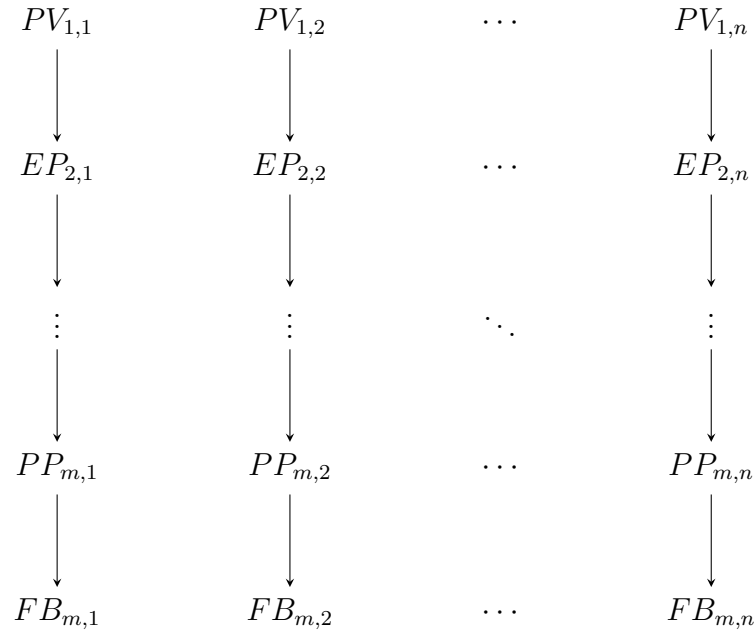


Figura 2.11: Sistema de render paralelo

representación paralela de la siguiente manera.

En la figura 2.11 se dice que el sistema es totalmente paralelo. El procesamiento de vértices (PV) y el ensamblaje (EP) se realizan en paralelo mediante la asignación de cada procesador a un conjunto de objetos en la escena. Asimismo, el procesamiento de píxeles (PP) y el Frame Buffer (FB) también se llevan a cabo en paralelo asignando a cada procesador una parte de los cálculos. En general, el Rendering se trata de calcular el efecto de cada objeto en la escena sobre cada píxel en la pantalla. Un objeto puede estar ubicado en cualquier lugar en el espacio 3D y por lo tanto puede terminar proyectándose en cualquier lugar de la pantalla (o fuera de la pantalla).

Se sugiere por Molnar et al. [12] que el Render puede ser visto como un problema de Sorting de objetos hacia la pantalla, y que la ubicación de este Sort en el pipeline determina en gran medida el sistema de representación paralelo resultante.

Este sorting puede tener lugar en cualquier etapa del pipeline gráfico, pero en términos generales puede ser durante el procesamiento de vértices y entonces se le llama Sort-

First, entre el ensamblaje y el procesamiento de píxeles llamado Sort-Middle o durante la rasterización llamado Sort-Last.

En el Sort-First, los objetos se distribuyen a las unidades de procesamiento en etapas tempranas del pipeline, esto se logra generalmente mediante la división de la pantalla en regiones y la asignación de regiones separadas para cada procesador. De esta forma que cada procesador sólo es responsable de los objetos dentro de su región asignada, esto por lo general logra pre-transformando los objetos para que puedan ser proyectar temprano en regiones específicas de la pantalla. Para el Sort-Middle, los objetos ya se han transformado en coordenadas de pantalla y a cada procesador se asigna a una parte de la pantalla.

Para el Sort-Last, la clasificación se lleva a cabo al final del pipeline gráfico. Cada procesador está asignado un subconjunto arbitrario de objetos de modo que cada procesador tiene que calcular los valores de píxel para su subconjunto. A pesar de que Sort-Last es adecuado para la pipeline clásico como el de la figura 2.2, este Sorting supone que cada procesador es responsable de un conjunto de objetos, por lo tanto, es un enfoque de distribución basada en lo objetos.

2.14 Memorias caché

Una memoria caché es un tipo especial de almacenamiento que suele emplearse para guardar datos que son frecuentemente accedidos por las unidades de ejecución.

Las memorias caché suelen ser extremadamente rápidas pero relativamente pequeñas, y por lo general están directamente en el chip de ejecución o están ubicadas físicamente muy cerca de este. Los caches se basan en dos principios comunes del ámbito de programación, el principio de localidad temporal y el principio de localidad espacial. El principio de localidad temporal tiene que ver con el hecho de que un dato solicitado por una unidad de ejecución tiene la tendencia de volver a ser solicitado a corto plazo por

la misma unidad de ejecución. El principio de localidad espacial responde a datos cuya dirección en memoria es contigua, los cuales tienden a ser solicitados simultáneamente en un momento del tiempo [5].

El evento en el cual una unidad de ejecución encuentra un ítem en el caché se denomina un “Hit” de caché mientras que el evento que corresponde a que el dato no se encuentra en el caché se denomina un “Miss”.

Existen esencialmente tres organizaciones de caché que describen donde y cómo colocar las unidades de información o bloques. El primer tipo de organización se denomina de “mapeo directo”. En los cachés de mapeo directo, cada bloque tiene un único posible lugar en el caché que viene dado por la siguiente fórmula:

$$\text{Posición del Bloque} = \text{Dirección de Bloque} \text{ MOD } (\text{Número Bloques en el caché})$$

El segundo tipo de organización es aquel en el cual el bloque de información puede ser colocado libremente en cualquier posición del caché, este esquema se conoce como “caché completamente asociativo”.

El tercer tipo de organización de memorias caché se denomina “caché asociativo por Sets”. En este esquema, cada bloque puede posicionarse únicamente en un grupo restringido de posiciones en el caché llamadas *sets*.

$$\text{Posición del Bloque} = \text{Dirección de Bloque} \text{ MOD } (\text{Número Sets en el caché})$$

La asociatividad del caché depende de los tipos de conflictos que puedan ocurrir, por lo que el análisis puede ser enfocado en varios frentes. En primera instancia se pueden analizar los misses involucrados a la hora de solicitar un dato.

El primer tipo de Miss bajo análisis es el Miss de conflicto o preceptivo que describe el inicio de una operación en el cache donde los primeros datos deben de ser traídos[5]. Estos misses no pueden ser prevenidos a nivel de arquitectura dado que el caché siempre debe de invalidar las posiciones de memoria al inicio de una computación.

Los segundos tipos de Misses son los Misses de capacidad y los de conflicto. Para analizar estos tipos de Misses vale la pena citar las cinco optimizaciones básicas propuestas por

Cinco optimizaciones para cachés según Hennessy y Patterson [5]		
Descripción	Beneficio	Perjuicio
Incremento del tamaño de bloque de caché	Reducción del Miss Rate	Incremento del Miss Penalty conforme aumenta el tamaño del bloque
Incremento del tamaño total del caché	Reducción del Miss Rate	Incremento del Hit Time, mayor costo de elaboración
Mayor asociatividad	Reducción del Miss Rate	Mayor complejidad
Empleo de caches Multinivel	Mejora en el Miss Penalty	Mayor complejidad
Prioridad a los Misses de lectura por encima de los de escritura	Mejora global de las condiciones de operación ligada a la mayor frecuencia de lecturas.	Las escritura se penalizan más fuertemente.

Figura 2.12: Optimizaciones de cachés según [5]

Hennessy y Patterson [5] para caches de datos, mismas que se tabulan en la figura 2.12.

2.15 Latencia

La latencia es tiempo desde que se invoca una determinada tarea hasta que la tarea empieza propiamente a ejecutarse. En términos de transferencias de datos, la latencia es el tiempo que transcurre desde que se da la orden de iniciar una transferencia de datos hasta que los datos comienzan realmente a ser transferidos. Este retraso puede ocurrir debido a la decodificación de la instrucción, líneas de espera de acceso al bus y otras causas. La latencia puede tener un gran impacto en el rendimiento al transferir pequeñas cantidades de datos, ya que la latencia es un costo inicial constante, que debe ser pagado independientemente de cuántos elementos han de ser transferidos. Cuantificar la latencia no es una tarea trivial. Una posible estrategia consiste en medir la transferencia de datos más pequeña posible y luego utilizar este tiempo como una aproximación de la latencia. Debido a que la presente investigación modela el sistema propuesto desde un punto de visto únicamente arquitectónico, la latencia no se será considerada.

2.16 Ancho de Banda

El ancho de banda es la cantidad de datos que pueden ser transferidos durante un intervalo de tiempo. Para medir el ancho de banda se debe tener en cuenta la latencia. La fórmula para calcular el ancho de banda viene dada como un función del tamaño, el tiempo y la latencia.

$$bandwidth = TransferSize / (t - latency) \quad (2.11)$$

2.17 Antecedentes

Emplear una representación basada en voxeles es una forma más eficiente de almacenar tanto el color así como la información de geometría. La utilización de SVO para representar directamente los modelos tridimensionales corresponde a una evolución directa de las técnicas desarrolladas recientemente para texturización virtual [7], [6]. A continuación se resume brevemente algunos de los antecedentes del estado del arte en SVO y hardware de renderización más relevantes para la presente investigación.

2.17.1 Implementaciones de Hardware de unidades de procesamiento gráfico

Las aproximaciones más cercanas al sistema que se presenta en este trabajo son los PowerRV utilizados en el Sega Dreamcast o las arquitecturas basadas Ray Tracing como los SaarCos por parte de la Universidad del Saarland [22]. El PowerRV cuenta con un enfoque basado Ray-Casting dividiendo la pantalla en azulejos. Sin embargo, el PowerVR no explota el paralelismo de varios núcleos: cada azulejo se procesa secuencialmente; una vez que el azulejo actual se ha terminado de renderizar, se comienza a trabajar en el siguiente azulejo hasta que la escena ha sido completamente renderizada [17]. Además PowerVR no emplea SVOs. SaarCos cuenta con múltiples núcleos, pero

depende de trazado de rayos en lugar de Ray-Casting. Dado que no sólo los rayos primarios son trazados, el algoritmo de trazado de rayos sufre del alto coste computacional de calcular las reflexiones para cada rayo. Los rayos reflejados tienden a ser incoherentes, por lo que el acceso a memoria y almacenamiento en caché es más difícil [25].

Por otro lado la tesis de maestría de Wilhelmsen [27] propone una arquitectura de hardware en FPGA que es quizás la más cercana al sistema propuesto, empleando SVOs, sin embargo la jerarquía de memorias caché es distinta a la planteada en esta investigación y además la estructura para representar el árbol no está basada en códigos de Morton.

2.17.2 GigaVoxels

La tesis doctoral de Cyril Crassin titulada GigaVolxes [2], presenta un enfoque para renderizar eficientemente escenas grandes y objetos detallados en tiempo real. El enfoque de Crassin se basa en una representación geometría volumétrica pre-filtrada y un trazado de conos basado en voxeles. Un elemento clave de la propuesta de [2] es orientar la producción de datos y almacenamiento en caché directamente en función de las solicitudes de datos emitidas en tiempo de ejecución durante el rendering. Uno de los aspectos más interesantes del sistema de Crassin es que su arquitectura es capaz de hacer streaming de partes del octree a la GPU bajo demanda (en tiempo de ejecución). La importancia de este aporte se puede apreciar cuando se utilizan SVO para almacenar modelos grandes, como un terreno, en estos casos se puede llegar a necesitar varios gigabytes almacenamiento para tener escena completa en memoria del GPU, por lo que una arquitectura que utilice streaming resulta muy conveniente.

Con respecto al algoritmo para recorrer el SVO, Crassin se basa en un algoritmo de reinicio, y evita el uso de una pila que es potencialmente ineficiente en un GPU.

2.17.3 Ray tracing de SVO en FPGA

Wilhelmsen [27] propone la arquitectura para un renderizador utilizando un FPGA. En su investigación, Wilhelmsen revisa los esfuerzos más recientes de software para utilizar SVOs con el propósito de renderizar gráficos por computadora en tiempo real, explorando como implementar un selección de estos algoritmos usando hardware dedicado. De esta manera, en la propuesta de [27] se presenta la implementación de unidad de ejecución que es capaz de recorrer SVOs, una unidad de control que puede gestionar la asignación de las tareas a las unidades de ejecución así como un sistema de memorias cachés que está optimizado para los patrones de acceso para este tipo de rendering. Adicionalmente, [27] integra la unidad de procesamiento gráfico con una unidad de procesamiento de propósito general disponible en su tarjeta de desarrollo y explica sus conclusiones. El algoritmo seleccionado por Wilhelmsen para recorrer el SVO es el descrito por Revelles [18], se trata de un algoritmo recursivo top-down con pila, que fue ligeramente modificado para no tomar en cuenta los datos de contorno del objeto. Una de las decisiones de diseño interesantes en la implementación de [27] es el uso del punto fijo en lugar del punto flotante. Wilhelmsen que el algoritmo de intersección las operaciones realizadas eran únicamente sumas o divisiones entre potencias de dos. Por esta razón, la decisión de utilizar fijo se debe a que la síntesis de una suma en punto flotante requiere de muchos menos recursos y es su ejecución es rápida que la punto flotante. Una de las principales diferencias del presente trabajo con el de Wilhelmsen está en las estructuras de datos utilizadas para representar el árbol de octantes. En su investigación Wilhelmsen utiliza estructuras de datos con punteros similares a las propuestas por Laine y Karras [9], mientras que en este trabajo se utilizan estructuras de datos sin punteros como se discute más adelante. Esta diferencia en la elección de las estructuras de datos afecta tanto el desempeño como el tamaño de los SVOs en la memoria. Finalmente, una de las contribuciones más interesantes de Wilhelmsen consiste en presentar una serie de benchmarks de su sistema para cachés con diferentes tipos de asociatividad

n-way, direct-map, etc. El sistema de cachés propuesto en la presente investigación consiste esencialmente en un árbol en el cual cada nivel de profundidad está representado por un caché jerárquico independiente por lo que el aporte es novedoso con respecto a lo presentado por [27].

Propuesta de investigación

“All the speed he took, all the turns he’d taken and the corners he’d cut in Night City, and still he’d see the matrix in his sleep, bright lattices of logic unfolding across that colorless void...”.

William Gibson, *Neuromancer*

La presente investigación consiste en proponer una arquitectura de Hardware para un renderizador tipo ray-cast que minimice los accesos a la memoria que almacena los objetos de la escena.

La arquitectura de Hardware está fuertemente inspirada en el sistema SVO propuesto por Crassin et al. [2] y en la implementación para hardware propuesta por Wilhelmsen [27], siendo hasta cierta medida la intención sobrepasar el desempeño de esta última.

La hipótesis es esencialmente que al combinar una serie de optimizaciones como por ejemplo: empleo de SVOs, particiones en la escena de tipo sort-fist, empleo de códigos de Morton [14] con recorrido del árbol por profundidad, y una jerarquía de memorias cache que se describe en la sección 6.2 , se obtendrá una alta tasa de aciertos alta en las memorias cache, y una disminución significativa en los accesos a la memoria que almacena los objetos en la escena.

3.1 Sistema ingenuo

El sistema ingenuo consiste en un sistema de ray-cast SVO sin ninguna optimización. Este sistema consiste de una serie de componentes que se listan a continuación:

- Una memoria centralizada que almacena los objetos de la escena.
- Una unidad de ejecución que se encarga de generar cada rayo y calcular la intersección de rayo con los octantes del SVO.
- El SVO se representa con punteros a los 8 octantes hijos en cada nivel de la jerarquía.

La idea es utilizar el sistema ingenuo como un sistema de referencia para efectos comparativos de las mejoras propuestas.

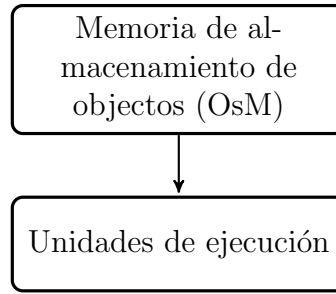


Figura 3.1: Sistema Ingenuo de referencia

3.2 Propuesta general de arquitectura

Tomando como referencia el sistema ingenuo de la figura 3.1 la arquitectura propuesta introduce una serie de mejoras a saber:

- Particiones en la escena de tipo sort-first.
- Empleo de códigos de Morton con recorrido del árbol por profundidad.
- Jerarquía de memorias cache de múltiples niveles.

El esquema general de este sistema mejorado se presenta a en la figura 3.2.

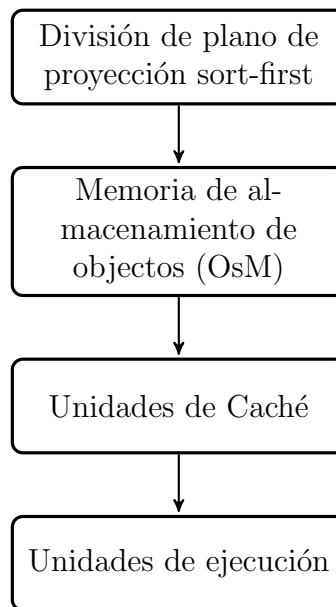


Figura 3.2: Propuesta de la arquitectura general

3.3 Definición de métricas

A continuación se definen las siguientes dos métricas:

Sea M la métrica el número total de accesos de la memoria OsM en una corrida del experimento.

Sea L la métrica la tasa normalizada del aciertos en una memoria caché.

3.4 Hipótesis

Una arquitectura de Hardware para un sistema de render 3D tipo ray-cast con las siguientes características:

- Aproximación de los objetos de la escena empleando primitivas tipo voxel
- Recorrido del SVO por profundidad,
- Subdivisión de la pantalla empleando una política ‘Sort-First’
- Empleo de códigos de Morton para apuntar a los datos de los octantes
- Jerarquía de memoria propuesta similar a la planteada en la sección 3.2

Conllevará a una alta tasa de aciertos en los caches del sistema y consecuentemente a una mejora a los accesos a la memoria de almacenamiento de objetos de la escena.

3.4.1 Pregunta de la Hipótesis

Dado el sistema ingenuo descrito en la sección 3.1, luego de aplicar las optimizaciones descritas en la sección 3.2, la métrica M ‘número de accesos a la memoria OsM’ del sistema de la figura 3.1 ¿sufrir alguna mejora?

3.4.2 Hipótesis Nula

La arquitectura propuesta en la sección 3.2 no presenta mejoras de la métrica M con respecto al sistema ingenuo de la sección 3.1.

3.4.3 Hipótesis Alternativa

La arquitectura propuesta en la sección 3.2 presenta mejoras de la métrica M con respecto al sistema ingenuo de la sección 3.1.

Metodología

“There are tumults of the mind, when, like the great convulsions of Nature, all seems anarchy and returning chaos; yet often, in those moments of vast disturbance, as in the strife of Nature itself, some new principle of order, or some new impulse of conduct, develops itself, and controls, and regulates, and brings to an harmonious consequence, passions and elements which seem only to threaten despair and subversion.”.

William Gibson, The differential
engine

A continuación se procedió a plantear una propuesta basada en la aplicación de diseño de experimento (DoE) para corroborar la veracidad de la hipótesis planteada en la sección 3.1. Diseño de experimentos (DoE) es una herramienta poderosa que se puede utilizar en una variedad de situaciones experimentales. DoE permite manipular múltiples factores de entrada para determinar su efecto sobre la salida deseada (la respuesta).

4.1 Factores y niveles

En el contexto de DoE un factor es aquel componente que tiene cierta influencia en las variables de respuesta. El objetivo de un experimento es determinar esta influencia. A su vez, cada factor cuenta con varios niveles posibles con los cuales experimentar. [29]

1. Profundidad del árbol

- (a) 5
- (b) 6
- (c) 7
- (d) 8
- (e) 9
- (f) 10

2. Tamaño de la partición sort-first

- (a) 2x2
- (b) 10x10
- (c) 20x20

4.2. Variables de respuesta

(d) 40x40

3. Número de líneas para cache

(a) 512 (8^3)

(b) 4096 (8^4)

(c) 32758 (8^5)

(d) 262144 (8^6)

Factores y Niveles						
Factor	Nivel 0	Nivel 1	Nivel 2	Nivel 3	Nivel 4	Nivel 5
Profundidad	5	6	7	8	9	10
Sort- First	2x2	10x10	20x20	40x40	–	–
Líneas- C1	–	512	4096	32768	262144	

4.2 Variables de respuesta

Dado que la hipótesis está dada en función del número de accesos a la OsM, se han identificado dos variables de respuesta a saber:

1. Tasa normalizada de aciertos en las memorias caché del sistema de la sección 3.2.
2. Número total de accesos a la OsM por corrida de experimento.

4.3 Recolección de datos

Como ya se mencionó, se creará un simulador que permita generar los datos para cada paso de la ejecución; mediante *scripts* estos datos serán ordenados de tal forma que

puedan ser usados como entrada para algún paquete estadístico que permita realizar un análisis de experimentos factoriales [26, p 561].

4.4 Análisis de varianza

El análisis de varianza (ANOVA, por sus siglas en inglés) permite asegurar que la variación en los resultados de un experimento no es mayor a la suma de variaciones de los factores y un cierto grado de error en sus medidas. Esto permite aceptar o rechazar la hipótesis con una probabilidad de error (de preferencia muy baja) con base en evidencia estadística [26, p 507].

El ANOVA tiene como objetivo analizar la relación entre una variable cuantitativa X y una variable cualitativa Y de k atributos. Cada atributo i define una población dada por la variable cuantitativa [20, p 247].

X_i : variable X restringida al atributo i .

Así, se tienen k poblaciones X_1, X_2, \dots, X_n (llamadas tratamientos) que se suponen normales, independientes, con variancias similares y con medias poblacionales $\mu_1, \mu_2, \dots, \mu_n$. Se desea determinar si X no varía según el atributo de Y , es decir, si las poblaciones son equivalentes y entonces los tratamientos son igualmente efectivos. Para ello se plantean y contrastan las hipótesis:

H_0 : X no varía según el atributo de Y (poblaciones equivalentes, es decir $\mu_1 = \mu_2 = \dots = \mu_k$).

H_1 : X varía según el atributo de Y (poblaciones no equivalentes), al menos dos de las medias no son iguales.

Un análisis de varianza permite saber si la diferencia en la media de varias poblaciones es significativa debido a la influencia de alguno de los factores. Una gran cualidad de ANOVA es que permite analizar varias poblaciones, mientras que otros métodos no permiten más de dos.

Existen varias herramientas de *software* que permiten hacer análisis de varianza y presentar los datos de manera gráfica. Inicialmente se ha pensado en el uso del paquete provisto por el proyecto R [16]; sin embargo, existen otras opciones como SPSS® y Minitab®.

Objetivos y alcance

“A los demonios no hay que creerles
ni cuando dicen la verdad”.

Gabriel García Márquez

5.1 Objetivos

5.1.1 Objetivo general

Diseñar una arquitectura de hardware para un renderizador tipo ray-cast empleando SVO, diseñado tomando consideraciones que disminuyan los accesos a la memoria que almacena los objetos de escena. La arquitectura deberá especificar los principales bloques funcionales del sistema, así como su funcionamiento en alto nivel y las interacciones principales entre estos.

5.1.2 Objetivos específicos

1. Especificar los principales bloques funcionales del sistema de render tipo ray-cast SVO propuesto, así como su funcionamiento en alto nivel y las interacciones principales entre estos.
2. Analizar el desempeño de los accesos a la memoria de almacenamiento de objetos de sistema propuesto, empleando como métrica el conteo de accesos de lectura en función de la profundidad del SVO.
3. Analizar el desempeño de los accesos a la memoria de almacenamiento de objetos de sistema propuesto, empleando como métrica la tasa de aciertos en sus memorias cache en función de las particiones de la pantalla por ordenamiento sort-first.

5.2 Alcances y limitaciones

El análisis estará limitado a la salida proporcionada por un simulador de software que será implementado como parte de los entregables de esta investigación. La investigación está enfocada únicamente en la arquitectura del sistema, estando fuera del alcance de esta investigación la síntesis del circuito, implementación de RTL, análisis de estático de tiempo, el consumo de energía del circuito, el trazado de rayos volumétrico, la

5.2. Alcances y limitaciones

representación interactiva o animación. Además, la investigación se centra únicamente en la teselación de los modelos, quedando por fuera la coloración o texturización de los mismos.

Arquitectura general del sistema simulado

“Todas las teorías son legítimas y ninguna tiene importancia. Lo que importa es lo que se hace con ellas.”

Jorge Luis Borges

6.1 Bloques e interacciones principales del sistema

La presente sección se dedica a describir más detalladamente los principales bloques del sistema. En términos generales el sistema propuesto consta de 5 bloques principales como se aprecia en la figura 6.1. Los bloques principales son:

- La memoria de almacenamiento de objetos OsM.
- La unidad de gestión de memoria.
- Las unidades de ejecución.
- Las memorias caché.
- El asignador de tareas.

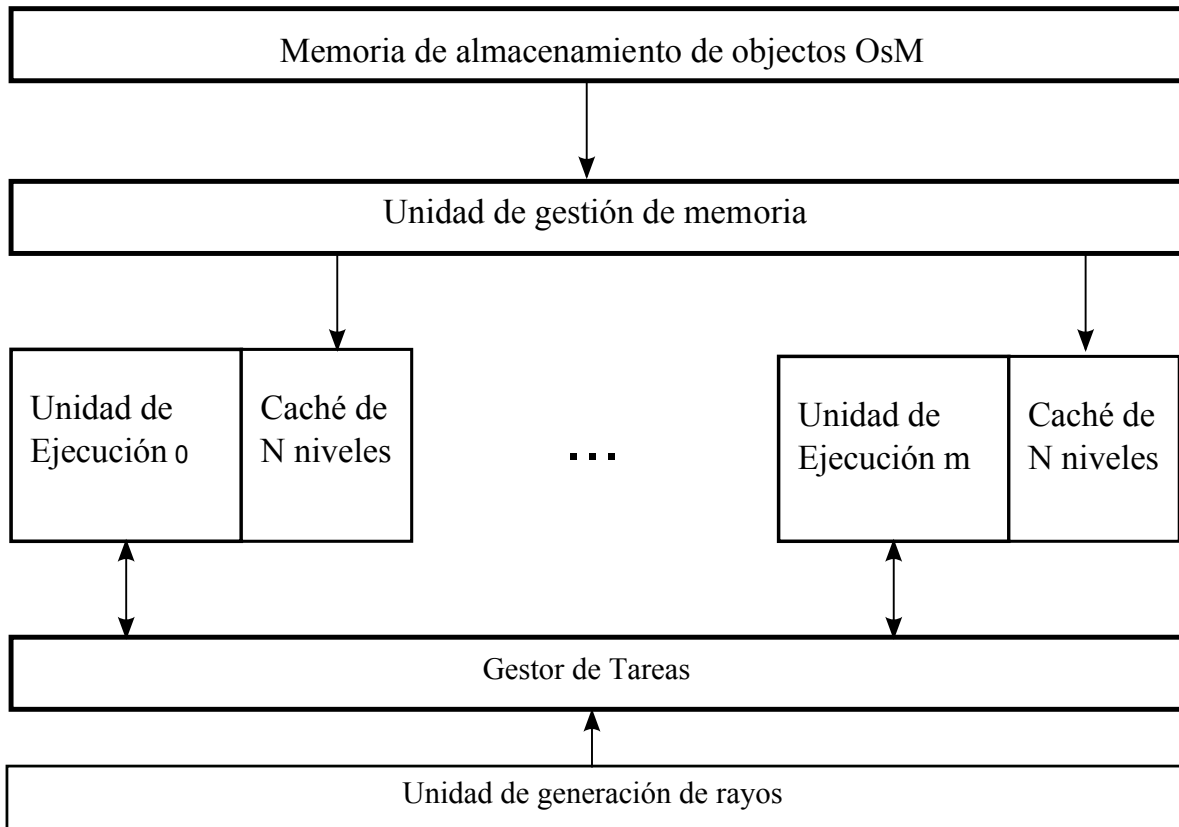


Figura 6.1: Arquitectura del sistema

Las siguientes secciones darán más detalle de cada uno los principales bloques de la arquitectura.

6.2 Memorias caché

Las memorias caché son una parte medular del sistema propuesto.

Wilhelmsen [27] sugiere en su tesis que dado que los caché de datos de un GPU que traza rayos utilizando SVO están siendo empleados en un algoritmo específico, debería en teoría ser posible implementar optimizaciones novedosas y específicas en los cachés que exploten los patrones de acceso de memoria del algoritmo. Por ejemplo, [27] encontró que los nodos que están más cerca de la raíz son visitados más a menudo que los nodos más profundos en el árbol. También encontró que es posible que sea ineficiente almacenar en el caché aquellos nodos que esten más profundo en el árbol ya que si se accede a ellos sólo una vez o dos veces, el beneficio del almacenamiento en caché es pequeño en comparación con la pena de desalojar a un nodo más cerca de la raíz.

El sistema propuesto tiene N memorias caché independientes para cada unidad del ejecución, donde N corresponde a la profundidad del SVO. Cada línea de una memoria caché almacena una estructura de datos similar a la de la figura 6.8. Las primeras 3 memorias caché tienen un número de líneas constante, asignadas de la siguiente manera.

- El caché de profundidad 0 tiene 8 líneas.
- El caché de profundidad 1 tiene $8*8 = 64$ líneas.
- El caché de profundidad 2 tiene $8*8*8 = 512$ líneas.

La razón para elegir este número de líneas para los 3 primeros cachés se debe a que corresponden directamente al número de octantes para los tres primeros niveles de profundidad del árbol. En otras palabras, el primer nivel de profundidad del SVO tiene 8 octantes, el segundo nivel de profundidad del SVO tiene 64 octantes y el tercer nivel

6.2. Memorias caché

de profundidad del árbol tiene 512 octantes, esto se ilustra en la figura 6.2. En el peor de los casos, cada rayo generado deberá evaluar su intersección con al menos un octante de cada uno de estos tres primeros niveles del SVO. Como se muestra posteriormente en la sección 7.7, el escenario real es mucho más favorable, ya que los tres primeros niveles del cache mantienen una tasa de aciertos consistentemente alta para las diferentes escenas y resoluciones.

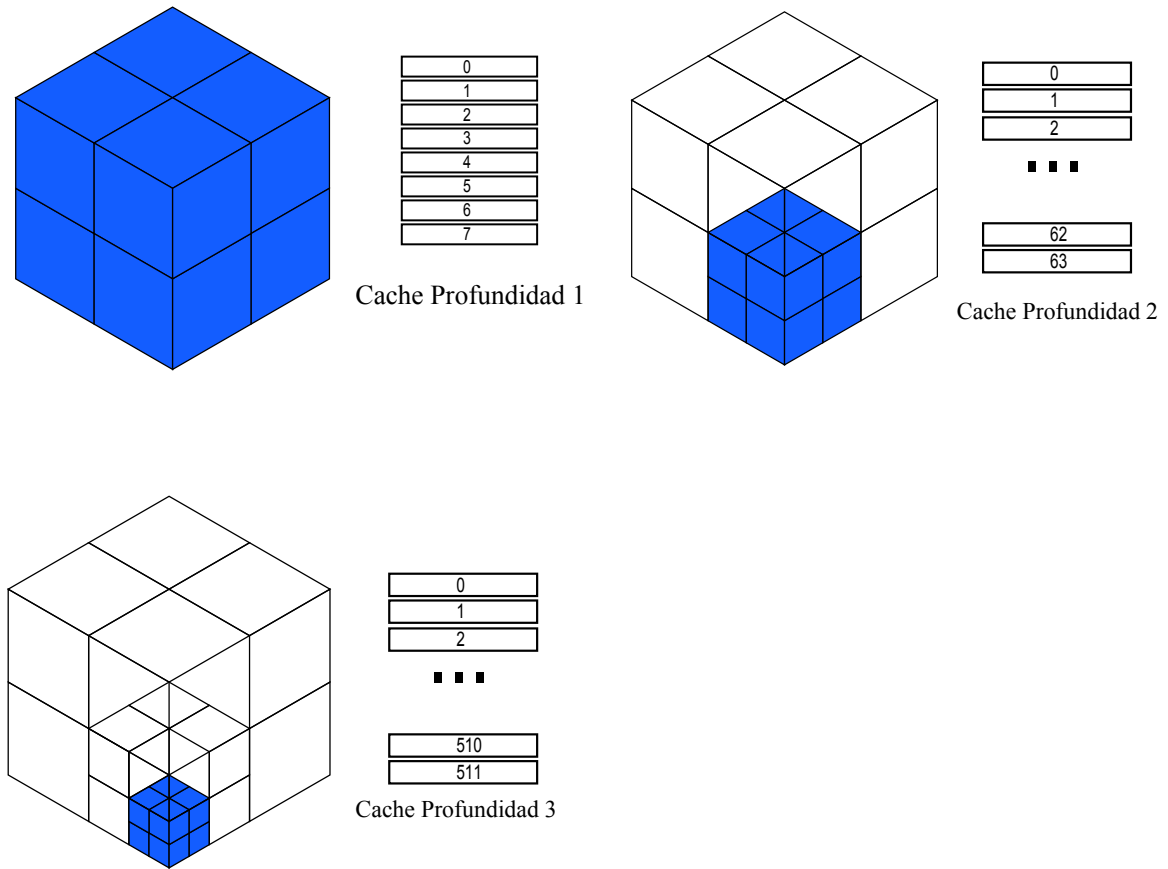


Figura 6.2: Líneas de cache para los tres primeros niveles de profundidad del SVO

Las memorias caché para las profundidades del SVO desde 3 hasta N tienen el mismo número de líneas al cual llamaremos L . Como se menciona en la sección 4.1, L es uno de los factores que se consideran en el diseño de experimento. Por el mismo razonamiento que con los tres primeros cachés, N corresponde a un número que es potencia de 8 para

que pueda al menos contener parte de la jerarquía inferior del árbol. La figura 6.3 ilustra esta idea.



Figura 6.3: Jerarquía de una memoria caché del sistema

6.3 Gestor de tareas global

El gestor de tareas es un módulo que se encarga de distribuir las tareas del sistema entre los núcleos de ejecución. Cada tarea es única y está asociada a un rayo. Para un rayo en particular, una tarea consiste en hallar la intersección de este rayo con el voxel más cercano al plano de proyección. El gestor de tareas adopta una política de asignación de tareas estática, la cual aprovecha la coherencia de los rayos dividiendo el plano de proyección en secciones rectangulares como se muestra en la figura 6.4.

Cada unidad de ejecución estará encargada de un único conjunto de particiones contiguas del plano de proyección. Esta idea se ilustra para 16 núcleos en la figura 6.6.

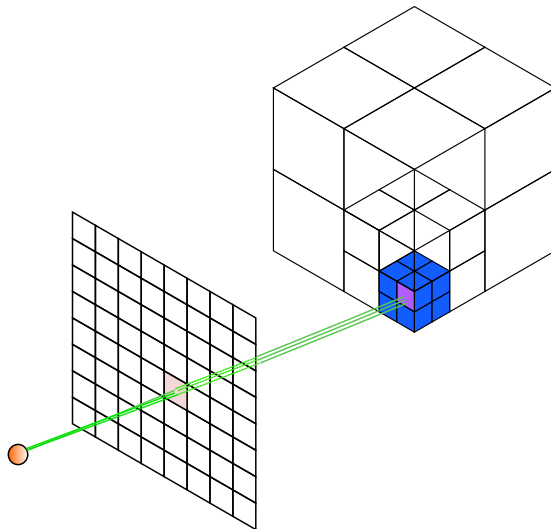


Figura 6.4: Rayos coherentes suelen intersectar los mismos voxels

La estrategia de usar particiones en el plano de proyección para aumentar la coherencia es la herramienta que nos permite aprovechar al máximo la jerarquía de caches de la sección 6.2. Entre más pequeñas las particiones del plano de proyección, mayor coherencia y menor número de lecturas fallidas en los caches. Esta idea de las particiones rectangulares se basa en el mismo principio de coherencia que los "Cone tracing" sugeridos por [2], con la particularidad de que realizar un barrido simple sobre regiones rectangulares es en general más sencillo de implementar en una arquitectura de hardware. Para una partición del plano de proyección particular, se eligió emplear un recorrido simple tipo raster como se ilustra en la figura 6.5. En este caso, el barrido da inicio generando un rayo que se envía hacia la esquina superior izquierda de la partición rectangular, desplazándose de izquierda a derecha en cada fila hasta finalmente alcanzar el último píxel correspondiente a la partición en la esquina inferior izquierda.

6.4 Unidades de ejecución

Para efectos de esta tesis cada unidad de ejecución se abrevia como GT (geometry traversal unit en inglés). Las GT son las encargadas de evaluar las intersecciones de los

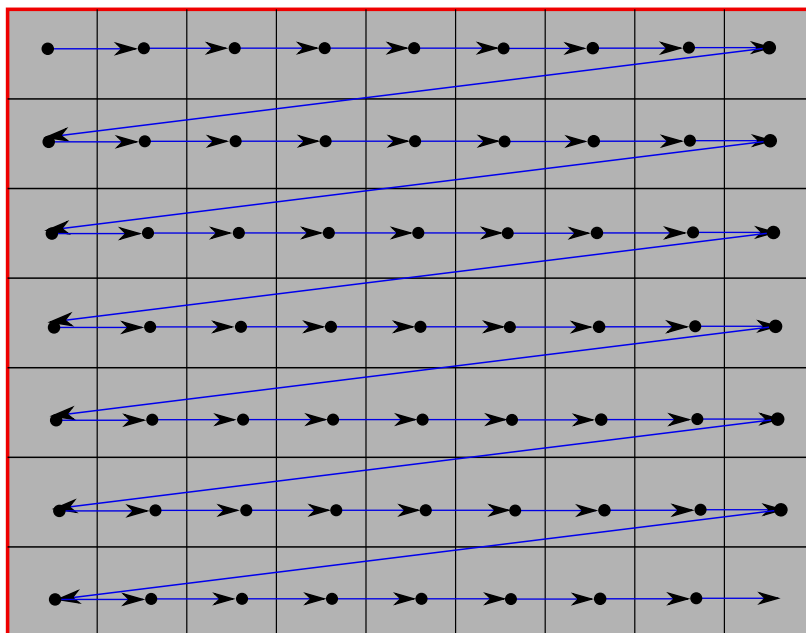


Figura 6.5: barrido de píxeles dentro de una partición rectangular del plano de proyección

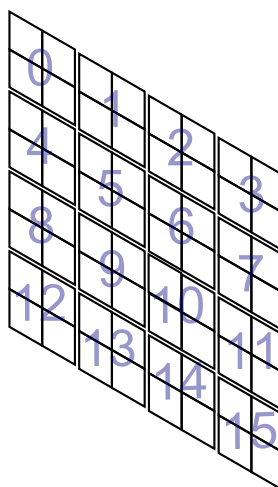


Figura 6.6: Ejemplo de asignación estática de tareas empleando secciones de plano de proyección para 16 núcleos de ejecución

rayos con el árbol de octantes. La evaluación de pruebas de intersección entre rayos y cajas es de las operaciones más costosas cuando se atraviesan BVH [10]. El algoritmo utilizado para por las GT para calcular la intersección del rayo con el *AABB* se basa en la sección 2.12, este algoritmo se muestra en el algoritmo 1.

Otra de las tareas de la GT es la generación de los índices de Morton, ya que el índice puede calcularse en tiempo real cuando se atraviesa el árbol a partir de la jerarquía del mismo. Esto último se explica con mayor detalle en la siguiente sección.

6.4.1 Propagación de tareas y recorrido de SVO

Si la intersección del rayo actual y el octante correspondiente al GT resulta positiva, entonces el GT genera 8 nuevas tareas, una para cada nodo hijo, de lo contrario el GT no genera nuevas tareas como se muestra en el algoritmo 2. Esto se ilustra en la figura 6.7.

La figura 6.7 ilustra el caso en el que la intersección del rayo con el nodo asignado al GT resulta positiva. Es importante recordar que el caché que se muestra en la figura 6.7 corresponde a un caché compuesto de N sub-caches, uno por cada nivel de profundidad del SVO tal y como se explicó en la sección 6.2.


```
 $tmin \leftarrow (min.x - r.orig.x)/r.dir.x$   
 $tmax \leftarrow (max.x - r.orig.x)/r.dir.x$   
if  $tmin > tmax$  then  
  |  $xchange(tmin, tmax)$   
end  
 $tymin \leftarrow (min.y - r.orig.y)/r.dir.y$   
 $tymax \leftarrow (max.y - r.orig.y)/r.dir.y$   
if  $tymin > tymax$  then  
  |  $xchange(tymin, tymax)$   
end  
if  $tmin > tymax || tymin > tmax$  then  
  | returnfalse  
end  
if  $tymin > tmin$  then  
  |  $tmin \leftarrow tymin$   
end  
if  $tymax < tmax$  then  
  |  $tmax \leftarrow tymax$   
end  
 $tzmin \leftarrow (min.z - r.orig.z)/r.dir.z$   
 $tzmax \leftarrow (max.z - r.orig.z)/r.dir.z$   
if  $tzmin > tzmax$  then  
  |  $xchange(tzmin, tzmax)$   
end  
if  $tmin > tzmax || tzmin > tmax$  then  
  | returnfalse  
end  
if  $tzmin > tmin$  then  
  |  $tmin \leftarrow tzmin$   
end  
if  $tzmax < tmax$  then  
  |  $tmax \leftarrow tzmax$   
end  
if  $tmin > r.tmax || tmax < r.tmin$  then  
  | returnfalse  
end  
if  $(r.tmin < tmin)$  then  
  |  $r.tmin \leftarrow tmin$   
end  
if  $r.tmax > tmax$  then  
  |  $r.tmax \leftarrow tmax$   
end  
returntrue
```

Algorithm 1: Intersección entre un rayo y un AABB

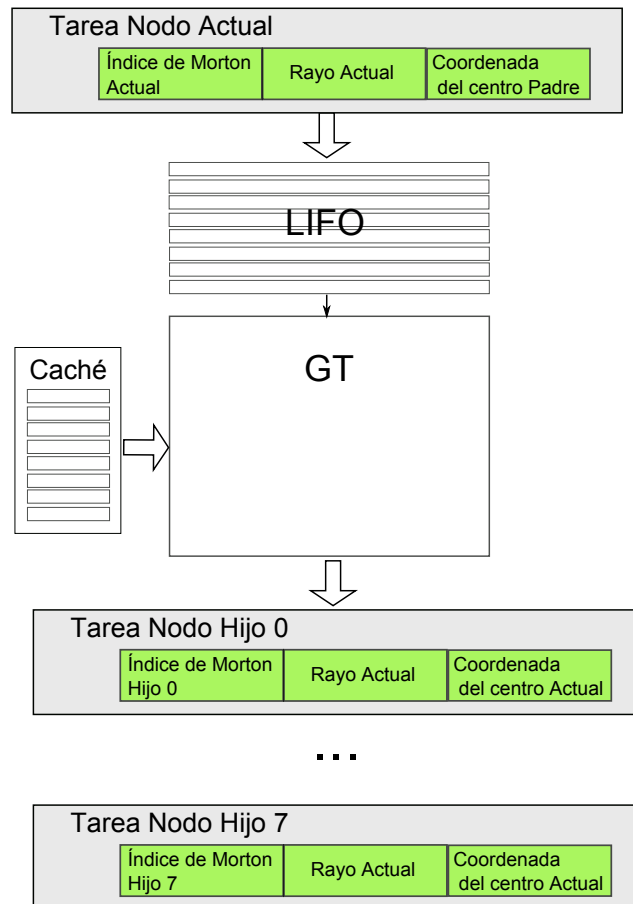


Figura 6.7: Entradas y salidas a un GT

```
Center ← ExtractOctantCenterFromParentCenter(ParentCenter )  
Hit ← RayAABBIntersection(Center )  
if Hit == true then  
    | i = 0 ;  
    | while i < 8 do  
    |     | ChildMortonCode ← (MortonCode << 3)|i  
    |     | PushJob( ChildMortonCode, Center, Ray )  
    |     | i ++  
    | end  
else  
  
end
```

Algorithm 2: Flujo de ejecución de un GT

6.5 Estructuras de datos

Se utiliza una estructura de datos de tipo árbol en la que cada nodo representa un octante. Esta estructura de datos esta diseñada de manera simple para minimizar la cantidad de información transmitida por el bus de datos que conecta el OsM. Al igual que [8] adoptamos un esquema donde la mayoría de los datos asociados con un voxel se almacenan en su nodo padre para de esta manera no necesitar nodos exclusivamente para guardar las hojas (voxeles sólidos). A diferencia de [8] no utilizamos punteros que referencian datos en bloques de la memoria, si no que apelamos únicamente al uso de índices de morton y a la codificación que se explica más adelante. Este esquema de almacenamiento permite representar la geometría de un voxel empleando únicamente 2 bits por voxel ¹.

Cada nodo del árbol almacena 8 elementos de dos bits para un total de 16 bits por

¹Es importante recordar que estos 2 bits corresponden únicamente a datos de geometría de modelo, tal y como se aclaró en la sección de alcances, para efectos de esta investigación no se toman en cuenta las texturas

6.5. Estructuras de datos

nodo como se muestra en la figura 6.8. Cada elemento codifica información sobre cada uno de los 8 hijos del nodo actual de la siguiente manera:

- 2'b00 : El nodo no tiene hijos (Octante vacío).
- 2'b01 : El nodo tiene hijos.
- 2'b10 : El nodo es una hoja (voxel sólido).
- 2'b11 : Valor reservado.

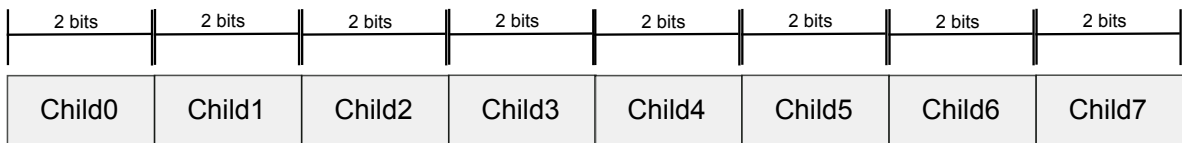


Figura 6.8: Estructura de datos para para un nodo del SVO

La unidad de generación de rayos genera rayos representados por la estructura de datos de la figura 6.9. La idea del sistema final es utilizar una representación numérica de punto fijo para esta estructura, sin embargo el análisis utilizando punto fijo se escapa de los alcances de este proyecto. Es importante notar de la figura 6.9 que los componentes de la dirección del rayo se guardan como una fracción entre 0 y 1. Esto último se hace con el fin de facilitar los cálculos de intersección para los GTs como aparecen en el algoritmo 1.

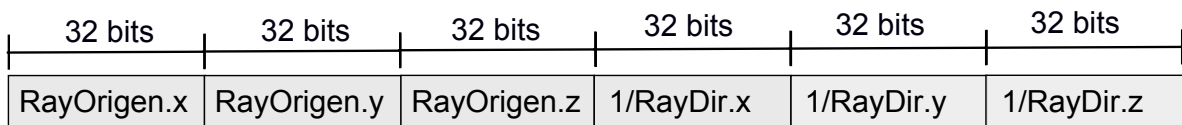


Figura 6.9: Estructura de datos para para un rayo

6.6 Algoritmo de recorrido del árbol

Como se mencionó en la sección 2.11 existen en la literatura una gran variedad de algoritmos para recorrer el árbol de octantes. Cada vez que el algoritmo visita un nodo se produce una solicitud de lectura a la memoria por lo que la elección del algoritmo de recorrido es importante.

La figura 6.10 muestra las 4 primeras iteraciones para el recorrido del SVO para el rayo R0. Inicialmente el RTU genera el rayo R0 representado por una estructura de datos similar a la mostrada en la figura 6.9.

- Iteración 1 :
 - La tarea asociada a $\{1,R0\}$ es asignada a GT0 por el gestor de tareas.
 - GT0 evalúa la intersección de R0 con el octante 1 a partir de la información del rayo R0 y de las coordenadas del centro del SVO disponibles de su elemento inferior en el LIFO de entrada.
 - La intersección del R0 contra el octante 1 resulta positiva.
 - GT0 genera 8 nuevos trabajos, uno para cada uno de sus 8 descendientes. Los nuevos trabajos se llaman $\{1000,R0\}$, $\{1001,R0\}$, $\{1010,R0\}$, $\{1011,R0\}$, $\{1100,R0\}$, $\{1101,R0\}$, $\{1110,R0\}$ y $\{1111,R0\}$ respectivamente.

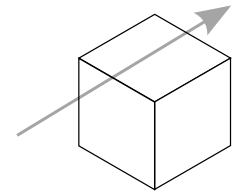
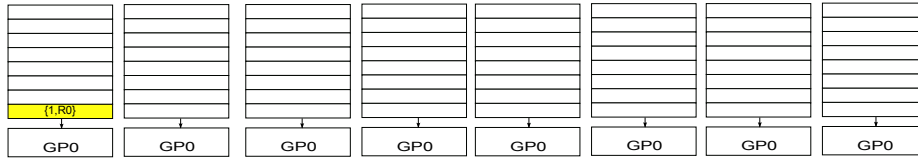
- Iteración 2 :
 - Los trabajos asociados a $\{1000,R0\}$, $\{1001,R0\}$, $\{1010,R0\}$, $\{1011,R0\}$, $\{1100,R0\}$, $\{1101,R0\}$, $\{1110,R0\}$ y $\{1111,R0\}$ son asignados a las unidades de ejecución GT0, GT1, GT2, GT3, GT4, GT5, GT6 y GT7 respectivamente. Cada una de las 8 unidades de ejecución de este ejemplo ejecuta sus instrucciones en paralelo con las demás.
 - Las pruebas de intersección del rayo actual resultan positivas únicamente para GT0 y GT1.

- GT0 genera 8 nuevos trabajos. Los nuevos trabajos se llaman $\{1000000,R0\}$, $\{1000001,R0\}$, $\{1000010,R0\}$, $\{100011,R0\}$, $\{1000100,R0\}$, $\{1000101,R0\}$, $\{1000110,R0\}$ y $\{1000111,R0\}$.
 - GT1 genera 8 nuevos trabajos. Los nuevos trabajos se llaman $\{1001000,R0\}$, $\{1001001,R0\}$, $\{1001010,R0\}$, $\{1001011,R0\}$, $\{1001100,R0\}$, $\{1001101,R0\}$, $\{1001110,R0\}$ y $\{1001111,R0\}$.
- Iteración 3 :
 - Las 8 nuevas tareas generadas por GT0 en la iteración anterior (tareas con prefijo $\{1000\}$) son asignadas a por el gestor de tareas a GT0, GT1, GT2, GT3, GT4, GT5, GT6 y GT7 respectivamente.
 - Las 8 nuevas tareas generadas por GT1 en la iteración anterior (tareas con prefijo $\{1001\}$)son asignadas a por el gestor de tareas a GT0, GT1, GT2, GT3, GT4, GT5, GT6 y GT7 con una prioridad secundaria, en otras palabras, las tareas asociadas al prefijo $\{1000\}$ deben ser atendidas de primero.
 - Las pruebas de intersección del rayo actual resultan positivas únicamente para GT0 y GT1.
 - GT0 genera 8 nuevos trabajos con el prefijo índice $\{1000000\}$.
 - GT1 genera 8 nuevos trabajos con el prefijo índice $\{1000001\}$.
- Iteración 4 :
 - Las 8 nuevas tareas generadas por GT0 en la iteración anterior (tareas con prefijo $\{100000\}$) son asignadas a por el gestor de tareas a GT0, GT1, GT2, GT3, GT4, GT5, GT6 y GT7 respectivamente.
 - Las 8 nuevas tareas generadas por GT1 en la iteración anterior (tareas con prefijo $\{100001\}$)son asignadas a por el gestor de tareas a GT0, GT1, GT2,

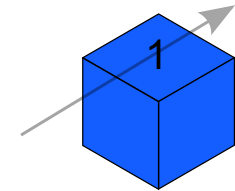
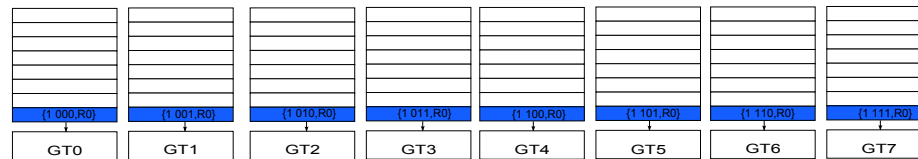
6.6. Algoritmo de recorrido del árbol

GT3, GT4, GT5, GT6 y GT7 con una prioridad secundaria, en otras palabras, las tareas asociadas al prefijo {100000} deben ser atendidas de primero.

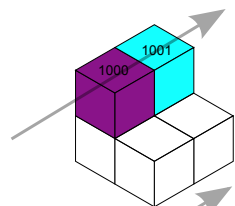
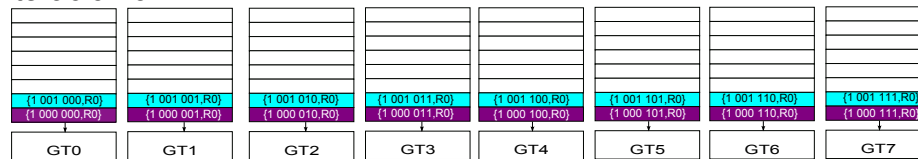
Iteración 1



Iteración 2



Iteración 3



Iteración 4

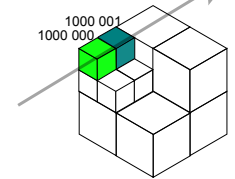
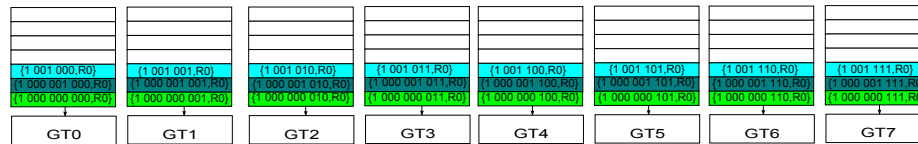


Figura 6.10: Distribución del trabajo de los octantes visitados del árbol en cada GT para 4 iteraciones del recorrido por profundidad, utilizando 8 GTs.

El efecto neto del sistema es el siguiente:

- **Solo un rayo se procesa a la vez:** No se comienzan a atender las tareas asociadas a un rayo hasta que aquellas asociadas al rayo anterior hayan concluido. Esto se hace con el fin de aprovechar al máximo la localidad espacial de los rayos, en otras palabras, si el rayo actual y el anterior tienen coherencia entonces el rayo actual podrá aprovechar la mayor parte de las entradas del caché escritas por el rayo anterior.
- **La mayoría de las entradas y salidas de los GT son generadas en tiempo de ejecución:** Como se explicó anteriormente, parte de la idea para reducir el

número de accesos a la OsM es que muchos de los parámetros tanto de entrada como de salida para procesar un rayo sean generados en tiempo de ejecución en lugar de ser leídos de la memoria. En otras palabras, dado que se ha identificado la lectura a memoria como el cuello de botella del sistema, entonces se invierten ciclos de procesamiento en las unidades de ejecución (esto puede ser llevado a cabo en paralelo como se muestra en la figura 6.10) para ahorrar de esta manera lecturas a la memoria principal. En resumidas cuentas, tanto los datos del rayo y las coordenadas del centro del octante requeridas para el algoritmo de intersección del listado 1, así como cada uno de los índices de Morton de los 8 hijos son calculados y propagados por el sistema sin requerir lecturas adicionales a la memoria. Las lecturas a la OsM son necesarias únicamente para decidir si un hijo está vacío o si se trata de una hoja, en otras palabras para detener el recorrido por profundidad, y como se mencionó en el párrafo anterior, gracias a que solo un rayo se procesa a la vez en el recorrido por profundidad, entonces la coherencia espacial (incrementada por el tamaño reducido de las particiones Sort-First) hace que se aprovechen las entradas que el rayo anterior escribió previamente al caché, reduciendo de esta manera las lecturas a memoria.

Análisis de Resultados

“...nothing contributes so much to
tranquelize the mind as a steady
purpose...”

Mary Shelley, Frankenstein

7.1 Descripción de los experimentos realizados, recolección de datos y diseño de banco de pruebas

7.1.1 Simulador arquitectónico

Con el fin de llevar a cabo los experimentos para esta investigación fue necesario desarrollar una herramienta de simulación. Esta herramienta de simulación se llama `theia_svo_sim` y consiste en un simulador a nivel arquitectónico desarrollado en C++ 2011. La herramienta de simulación arquitectónica es capaz de llevar a cabo todas las tareas necesarias para los experimentos incluyendo:

- Voxelización de modelos tridimensionales.
- Renderización rápida empleando un modelo simplificado de software.
- Renderización simulada utilizando el modelo arquitectónico de hardware de la sección 6.
- Modo interactivo de uso.
- Modo no interactivo de uso (empleado para correr los experimentos factoriales de DoE).

La herramienta modela el comportamiento del sistema propuesto incluyendo el recorrido del SVO por profundidad, la subdivisión de la pantalla empleando la política ‘sort-first’, el uso de códigos de Morton para apuntar a los datos de los octantes, así como la jerarquía de memoria de la sección 3.2. La herramienta de simulación es además capaz de modificar un gran número de parámetros estructurales y de operación del modelo de simulación. Estos parámetros incluyen desde luego las variables de respuesta, factores y niveles requeridos para el posterior análisis de varianza de cada experimento.

7.1. Descripción de los experimentos realizados, recolección de datos y diseño de banco de pruebas

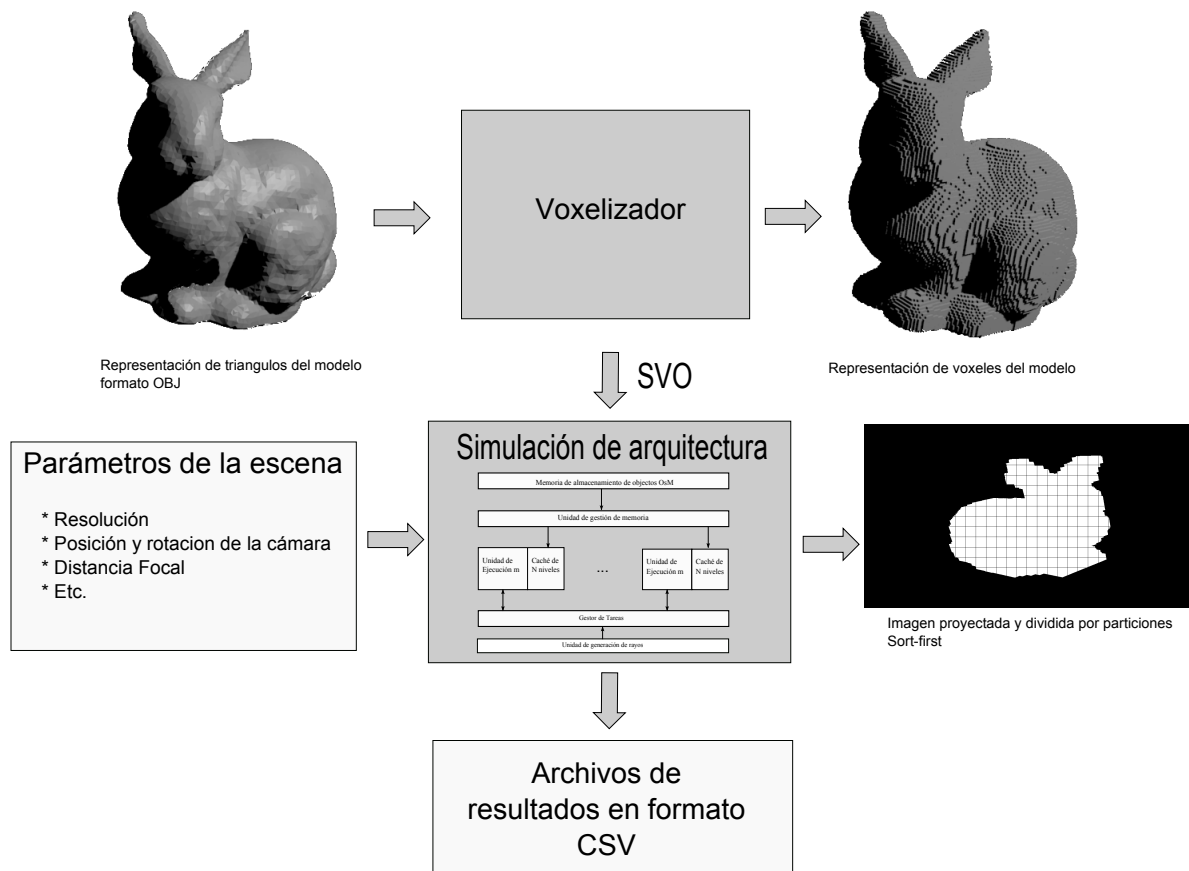


Figura 7.1: Flujo de ejecución de herramienta de simulación

7.1. Descripción de los experimentos realizados, recolección de datos y diseño de banco de pruebas

7.1.2 Banco de pruebas

Para esta investigación se simularon experimentos para cada una de las combinaciones factoriales de los factores y niveles de la sección 4.1.

Cada una de las pruebas se llevó a cabo para 4 escenas que se muestran en la figura 7.2 y que se enumeran a continuación: ¹.

- Utah Teapod : 992 triángulos.
- Stanford Bunny : 16214 triángulos.
- Happy Buddah : 100000 triángulos.
- Stanford Dragon : 100000 triángulos

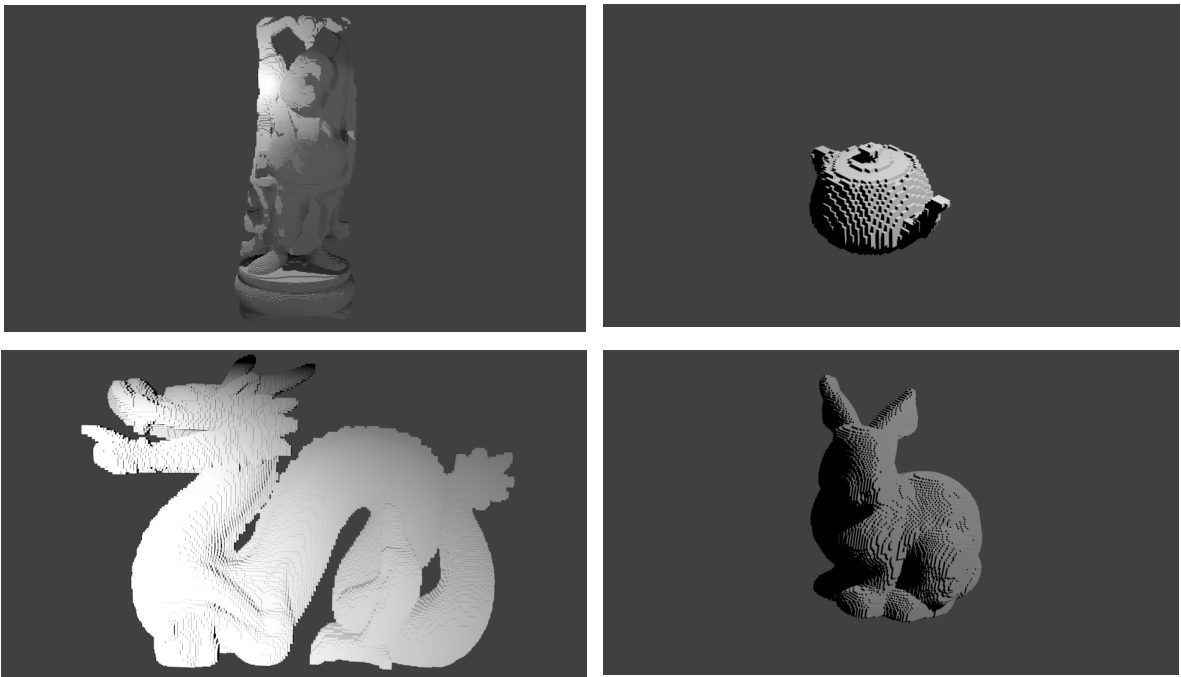


Figura 7.2: Ejemplo de escenas voxelizadas utilizadas en las pruebas. Superior izquierda : Happy Buddah, Superior derecha: Utah teapod. Inferior izquierda: Standord dragon. Inferior derecha: Standford Bunny.

¹ El simulador de la sección 7.1.1 no lleva a cabo iluminación. Los meshes voxelizados de la figura 7.2 fueron en efecto generados con la herramienta de simulación, pero la iluminación se agregó con una herramienta externa para efectos estéticos

7.1. Descripción de los experimentos realizados, recolección de datos y diseño de banco de pruebas

Cada una de las pruebas se corrió para las siguientes resoluciones (la resoluciones de los experimentos coinciden con el largo y ancho del plano de proyección)

- 640 x480 (VGA)
- 800x600 (SVGA)
- 1280x720 (720p)
- 1920x1080 (1080p)
- 3200x1800 (4k)

Los valores de las resoluciones se eligieron para representar la gama más común de resoluciones soportadas por dispositivos modernos.

Como se mencionó en la sección 7.1.1 se implementó una herramienta de simulación arquitectónica para validar el sistema propuesto. Luego de correr cada experimento factorial esta herramienta generó una serie de archivos de texto que contienen la información para cada uno de los factores y niveles. Debido a que los nombres se de estos factores y niveles en la gráficas de la secciones siguientes corresponden a los nombres de estas variables en el contexto de la herramienta, a continuación se procede a enumerar cada variable de salida como aparecen en la gráficas y su significado en el contexto del sistema propuesto.

7.1. Descripción de los experimentos realizados, recolección de datos y diseño de banco de pruebas

Parámetros de salida de simulador	
Nombre de parámetro de simulador	Descripción
scene.resolution	La resolución de la escena (corresponde a las dimensiones del plano de proyección)
octree.depth	Profundidad máxima del SVO
total.cache.size.mb	Tamaño total del caché en MB
gpu.grid.partition.size	Corresponde al tamaño de las particiones sort first del plano de proyección
gpu.memory.cache.lines.per.way	Número de líneas de las jerarquías de caché (recordar que esto aplica para los niveles de caché mayores a 3)
cache.l1.hit.rate	tasa de aciertos de la memoria caché
external.mem.read.count	número de accesos al OsM del sistema propuesto.
mem.total.reads	número de accesos al sistema de memoria, tanto para los datos que estaban en el caché así como datos que se encuentran en la OsM. Para efectos de comparación se considera además como el número de accesos del sistema ingenuo.

A continuación se procede a listar los resultados para cada una de las variables de respuesta del experimento.

7.2 Resultados y aplicabilidad de ANOVA

La figura 7.3 muestra una serie de 4 gráficos en relación a los residuos que se emplean para verificar los supuestos de ANOVA. En esencia, los residuos deben cumplir 2 condiciones: que sigan una distribución normal y que la varianza de los residuos sea igual para cada grupo y a todos los grupos. Mediante una exploración visual de la figura 7.3 es posible concluir que los datos generados por las corridas de los experimentos no parecen seguir una distribución normal. La gráfica Normal Q-Q plot de la figura 7.3 muestra que tan lejos están los residuos de una distribución normal. La figura sugiere que los residuos no siguen una distribución normal por la forma S de la curva en la figura. Por otro lado, la gráfica de los residuos vs. los valores ajustados de la figura 7.3 parece confirmar la sospecha de que los datos no siguen una distribución normal. Esta gráfica muestra las varianzas de los residuos y usualmente se espera que los puntos no deberían no seguir un patrón definido. En resumidas cuentas, la figura 7.3 sugiere que por su naturaleza, los datos de este experimento no son candidatos para un análisis utilizando la herramienta de ANOVA.

7.3 Test de Kruskal-Wallis para análisis de varianza

Tal y como se mencionó en la sección 7.2, dado que no se puede sustentar la suposición de normalidad, debemos utilizar un procedimiento alternativo al análisis de varianza el cual no dependa de esta normalidad. El procedimiento de Kruskal y Wallis es justamente la herramienta adecuada para este problema. El test de Kruskal-Wallis se utiliza para contrastar la hipótesis nula de que los n tratamientos son idénticos contra la hipótesis alternativa que dice que algunos tratamientos generan observaciones que son más grandes que otras [13]. A continuación se muestran los resultados del test de Kruskal-Wallis para todas las escenas y todas las resoluciones.

Kruskal-Wallis rank sum test

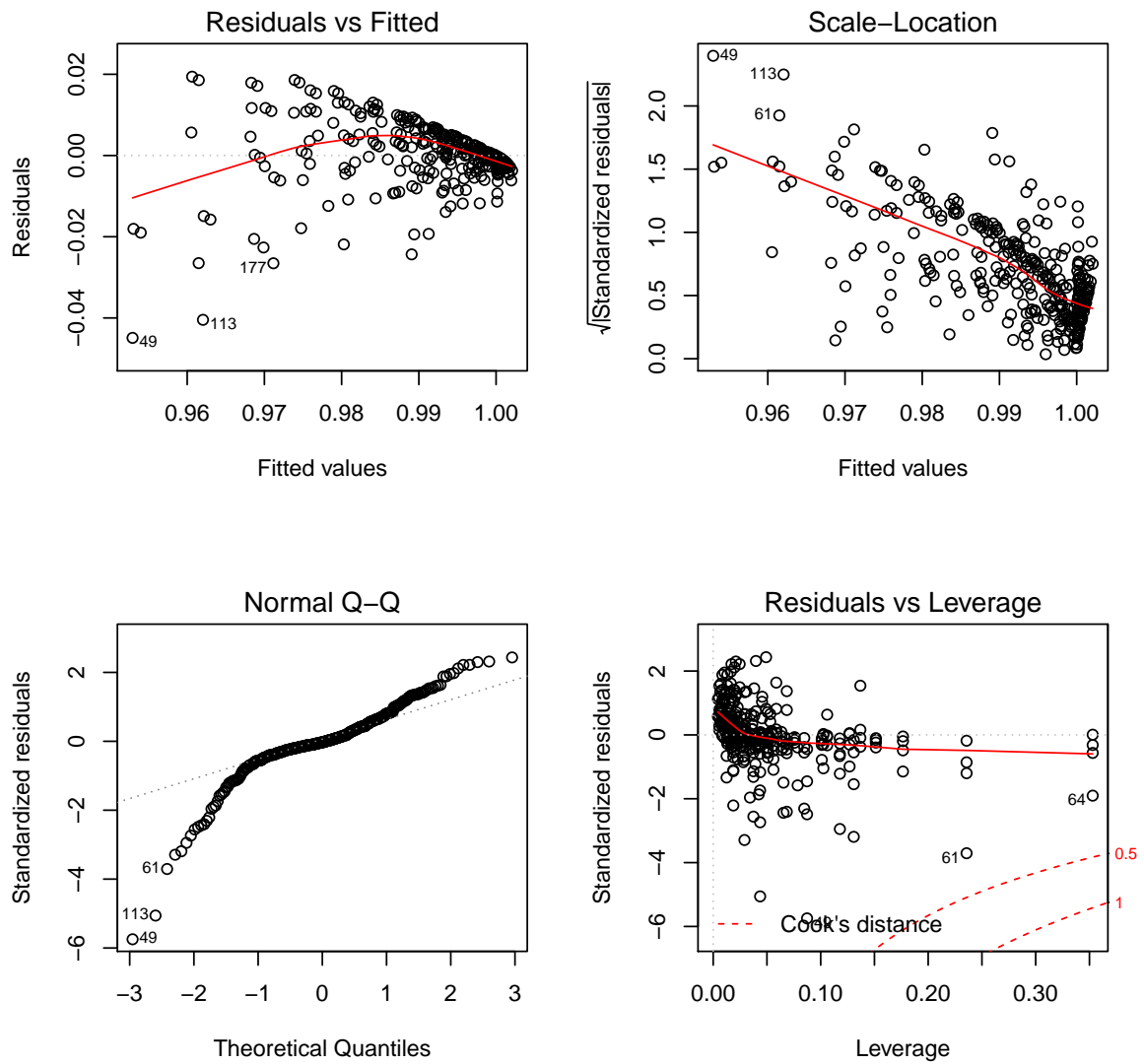


Figura 7.3: Resultados de ANOVA


```
data: cache_l1_hit_rate by
octree_depth by gpu_grid_partition_size by
resolution by gpu_memory_cache_lines_per_way
```

```
Kruskal-Wallis chi-squared = 1181.588, df = 5, p-value < 2.2e-16
```

Debido a que el $p - value$ es suficientemente pequeño, la hipótesis nula se rechaza y consecuentemente la hipótesis alternativa se acepta.

En otras palabras las características del sistema propuesto en efecto tiene un contribución positiva en la reducción de los accesos a la memoria OsM.

Las siguientes secciones se dedican a analizar el efecto sobre cada variable de respuesta, así como las contribuciones de cada uno de los factores del experimento sobre las mismas.

7.4 Resultados de la variable de respuesta 1 - Total de accesos a la OsM

La variable de respuesta número 1 de la sección 4.2 se llama "número total de accesos a la OsM por corrida de experimento". Tal y como se mencionó en las secciones anteriores, la idea de la arquitectura propuesta consiste esencialmente en minimizar el número de accesos a la memoria donde se almacena la jerarquía de la escena, la OsM. La variable de respuesta 1, es precisamente el número de accesos a esta memoria, por lo que su reducción confirma que el sistema propuesto cumple con su objetivo primordial. Las figuras 7.4, 7.5 y 7.6 muestran los resultados de esta variable para cada una de las escenas usadas. De estos resultados se pueden extraer las siguientes observaciones:

- **El número promedio de accesos al la OsM se ve dramáticamente reducido gracias a la prescencia del caché:** Esto se puede apreciar claramente de la figura 7.4, esta figura muestra el número de accesos a la OsM para el sistema ingenuo (filas azules) y el sistema propuesto (filas rojas) contrastando todas las escenas. En las figuras 7.5 y 7.6 las columnas color naranja claro representan el número de accesos del sistema ingenuo al SVO, mientras que las naranja oscuro representan el número de accesos del sistema SVO por el sistema propuesto. Se puede entonces apreciar que en el peor de los casos hay una mejora de casi un orden de magnitud con respecto al sistema que no cuenta con memorias caché (note que los ejes de las figuras están en escala logarítmica).
- **El número de accesos al OsM aumenta conforme aumenta la resolución:** Esto se aprecia claramente de las figuras 7.5 y 7.6. Esto se debe a que aumenta el número de rayos generados, un rayo por cada píxel del plano de proyección, cuyo largo y ancho es equivalente a la resolución de la escena para estos experimentos.
- **El sistema propuesto mantiene una muy buena disminución del trafico entre el OsM y los núcleos de ejecución aun para profundidades grandes y resoluciones grandes:** esto se aprecia de las figuras 7.5 y 7.6. Aun para resoluciones de 4k y profundidades de SVO de 10, el sistema mantiene un considerable disminución del promedio de lecturas.

7.4. Resultados de la variable de respuesta 1 - Total de accesos a la OsM

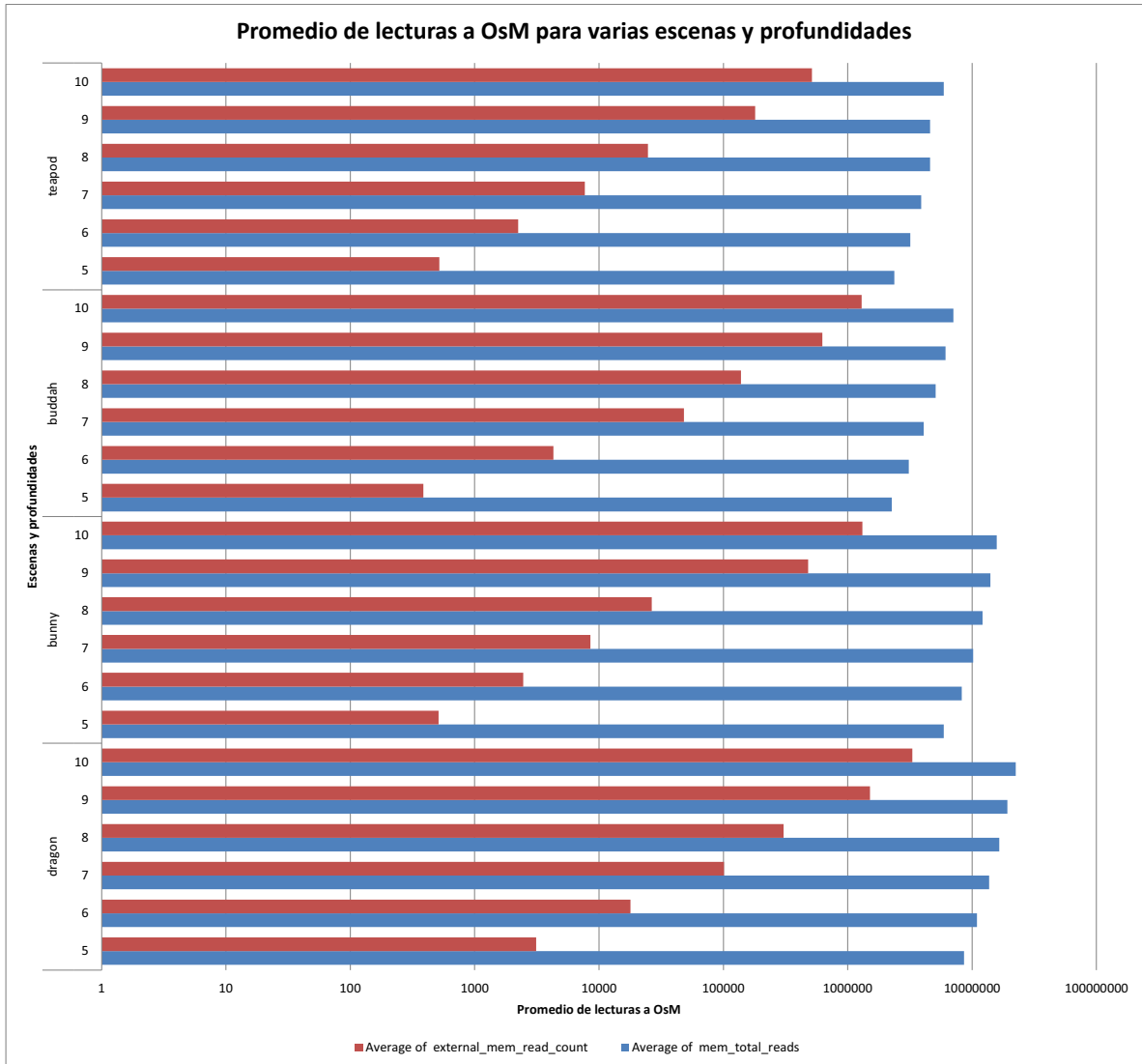


Figura 7.4: Resultados variable de respuesta 1. Comparación de todas las escenas

7.4. Resultados de la variable de respuesta 1 - Total de accesos a la OsM

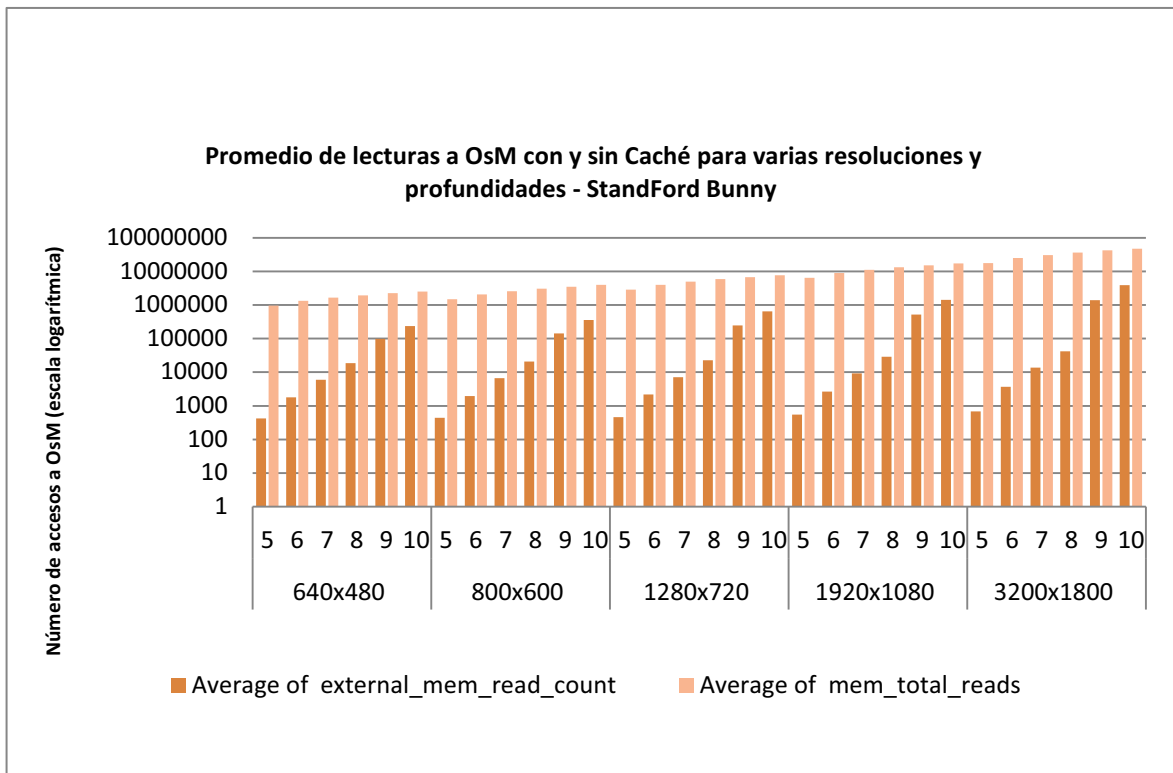
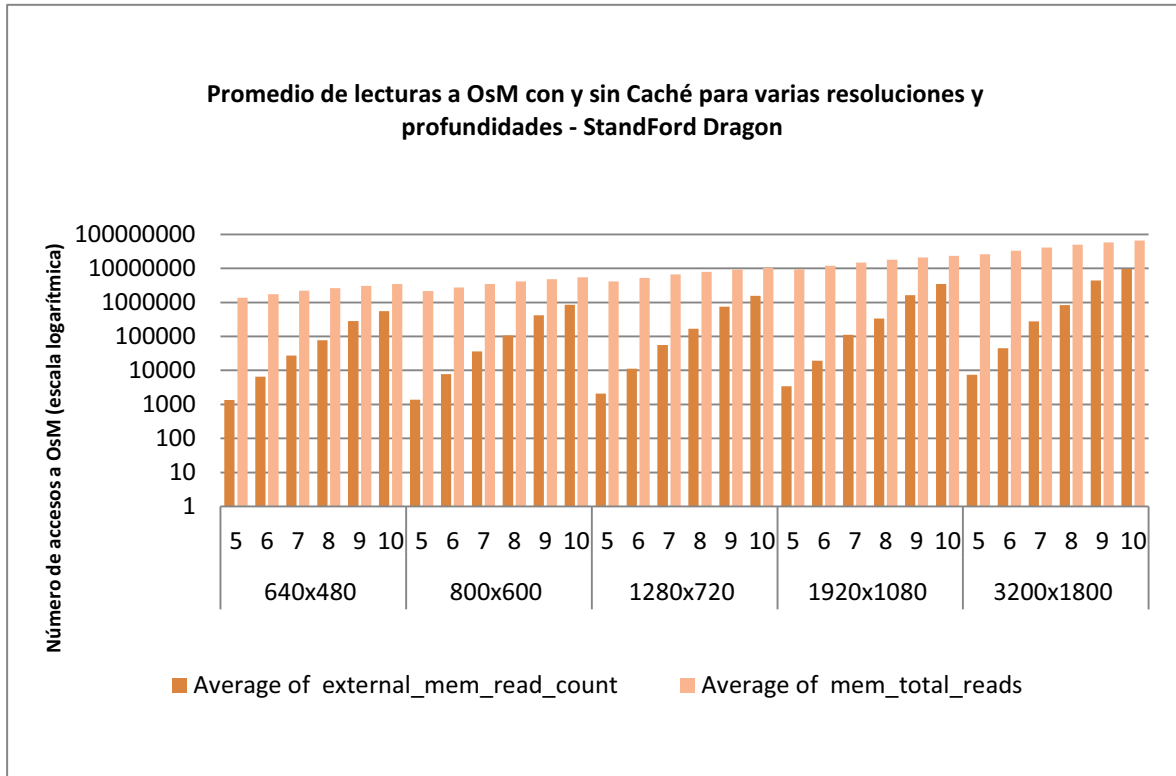


Figura 7.5: Resultados variable de salida 1

7.4. Resultados de la variable de respuesta 1 - Total de accesos a la OsM

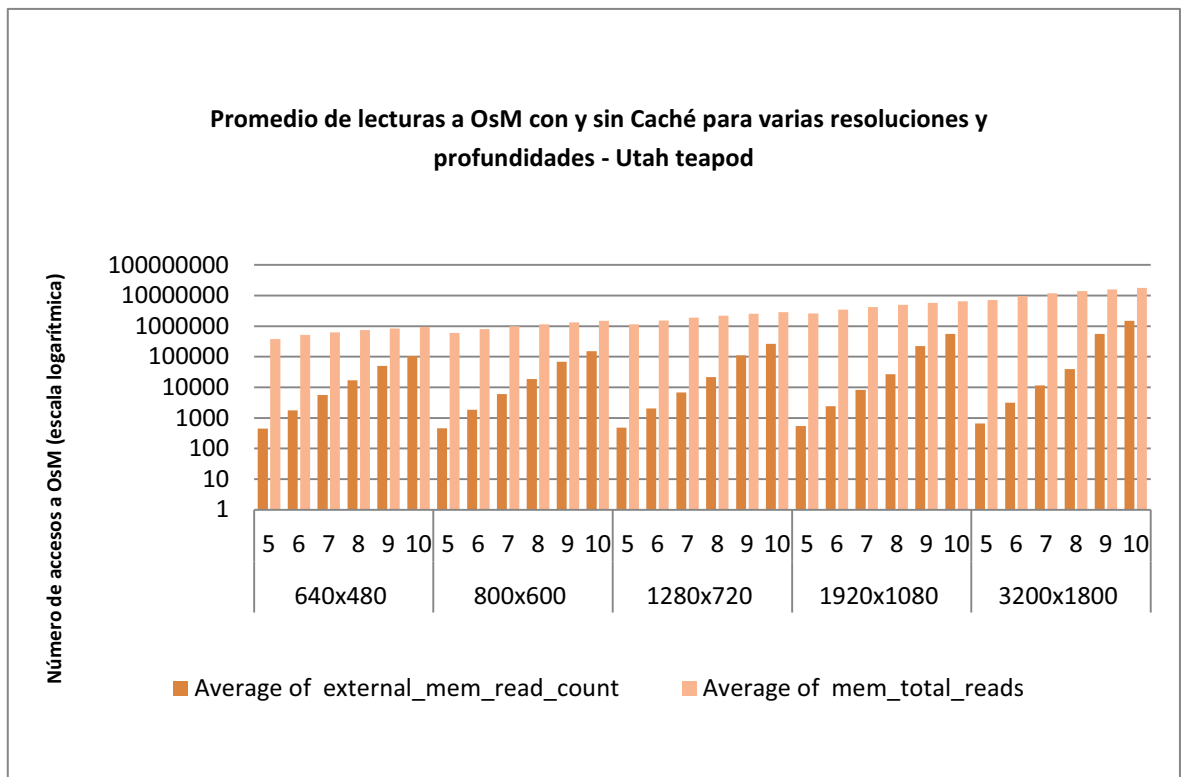
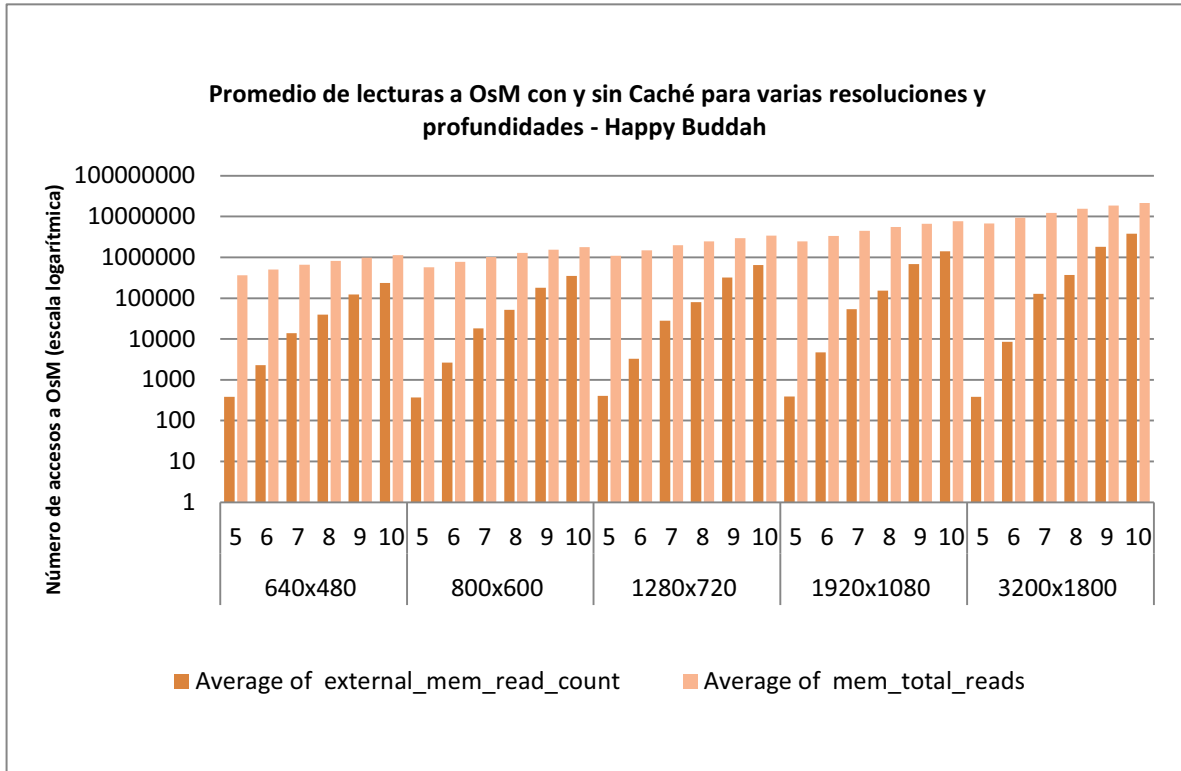


Figura 7.6: Resultados variable de respuesta 1 (continuación)

7.5 Resultados de la variable de respuesta 2 - Promedio de aciertos a la memoria caché

La variable de respuesta número 2 corresponde al promedio de aciertos en el caché. Para efectos del análisis los aciertos se representan como porcentajes en el rango entre 0 y 1. Se nota de las figura 7.7 que a diferencia del número total de accesos a la memoria, el cual aumenta conforme la resolución aumenta, para una profundidad de SVO dada, la media del porcentaje de aciertos a la memoria cache se mantiene en un número casi constante independientemente de la resolución. Esto último es de esperarse ya que los cachés únicamente almacenan información concerniente a los nodos del SVO lo cual es independiente de la resolución de la escena.

La figura 7.7 muestra los resultados para la variable de salida 2. Se aprecia de la figura que para profundidades del SVO inferiores o iguales a 8 el sistema presente un tasa de aciertos cercana al 95 %, mientras que para profundidades de 9 la tasa de aciertos es superior a 85 % y para una profundidad del 10 es siempre superior al 75 %. Es importante mencionar que un SVO con profundidad 10 puede llegar a ser un tanto exagerado. Recordemos de la figura 2.5 como para una profundidad de 8, el resultado de aproximar las superficies con voxels ya es lo bastante preciso y de requerir una mayor suavidad en la superficie se puede aplicar la técnica detallada en la sección 2.9, es decir el uso de la interpolación trilineal en lugar de incrementar la profundidad del SVO.

7.6 Análisis de tamaño de la ventana de partición sort first

La figuras 7.8 y 7.9 muestran los resultados de variar el tamaño de la partición sort-first en todas las escenas para las distintas resoluciones. Recordemos que los tamaños

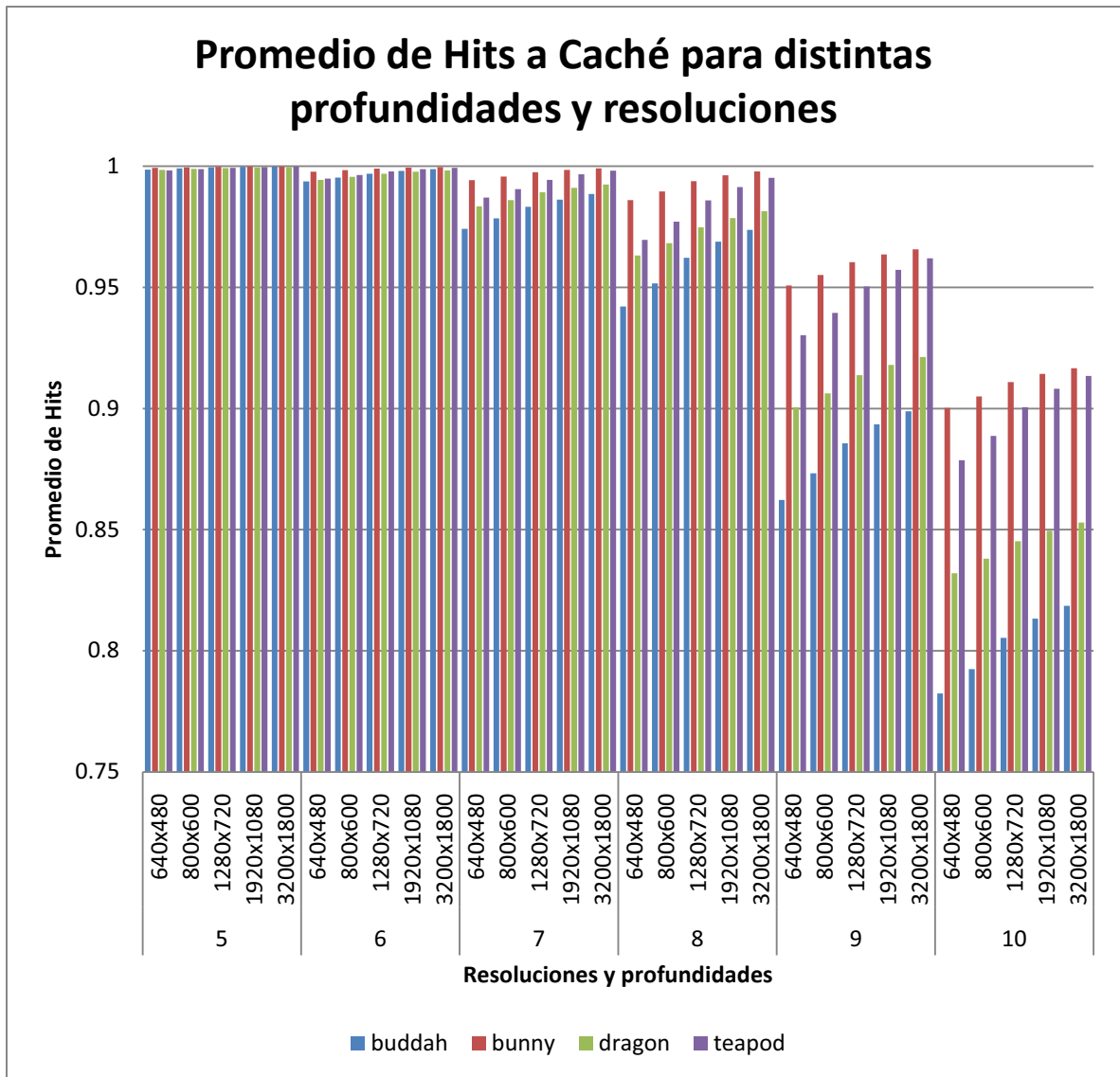


Figura 7.7: Resultados variable de salida 2

de ventana que se seleccionaron para los experimentos fueron:

- 2 píxeles x 2 píxeles
- 10 píxeles x 10 píxeles
- 20 píxeles x 20 píxeles
- 40 píxeles x 40 píxeles

En la figura 7.8 las columnas rojas representan el número de accesos al OsM del sistema ingenuo, mientras que las columnas azules representan el número de accesos al OsM del sistema propuesta para cada tamaño de ventana a las diferentes resoluciones. Es claro en la figura 7.8 que el tamaño de la ventana tiene un efecto en el número promedio de accesos, siendo el tamaño de ventana más pequeño (2x2) aquel que muestra mejores resultados. Lo anterior se debe que a menor tamaño de la ventana, la cantidad de datos almacenados en el caché entre un rayo y el siguiente es mayor, siendo el caso trivial el que la ventana es exactamente del mismo tamaño que el plano de proyección, en cuyo caso el caché no podrá ser lo suficientemente grande como para albergar la información común entre un rayo y el siguiente (tal y como se explicó en mayor detalle en la sección 6.3).

La figura 7.9 muestra el impacto en el promedio de Hits al caché cuando se varía el tamaño de la partición sort-first . La figura 7.9 promedia los resultados de las simulaciones para todas la escenas a distintas resoluciones. Resulta claro de esta figura que la variación porcentual del Hit Rate debido al tamaño de la ventana no es lo bastante significativo (inferior al 5 %), sin embargo el sistema mantiene un promedio de Hits al caché consistentemente superior al 90 %.

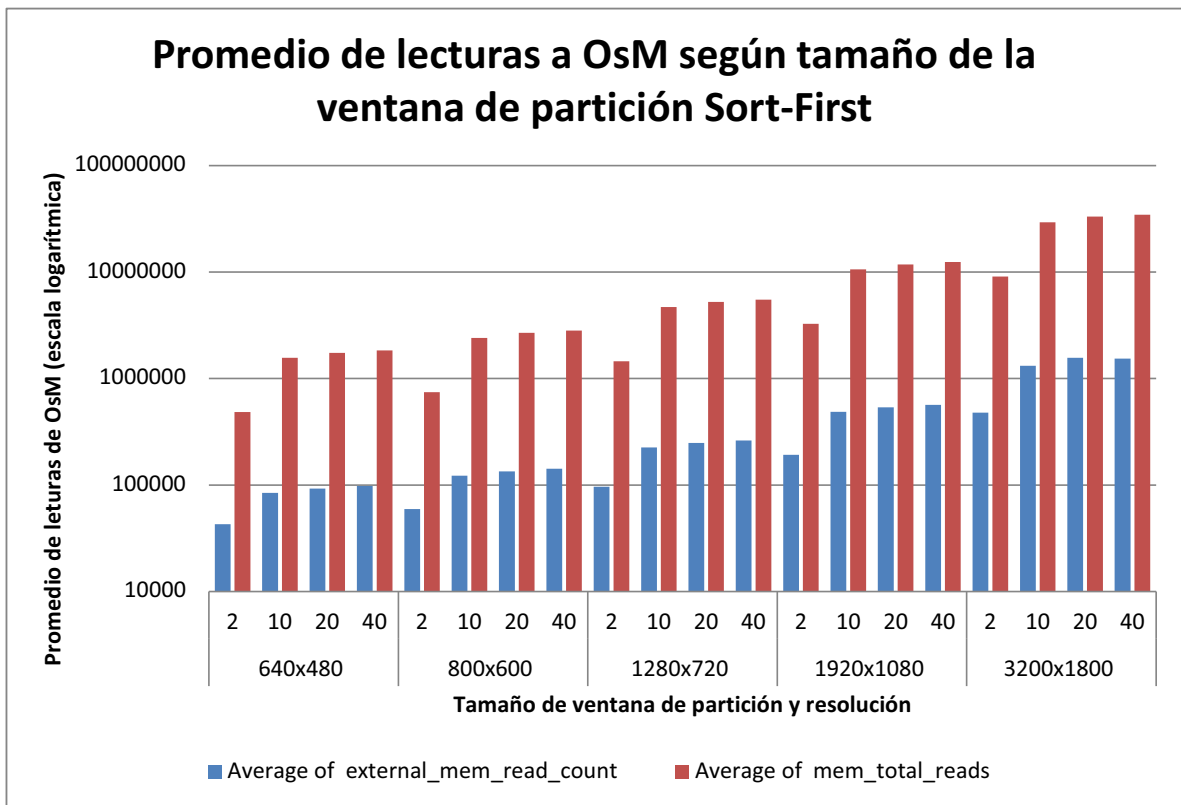


Figura 7.8: Promedio de lecturas a OsM para todas las escenas a distintas resoluciones

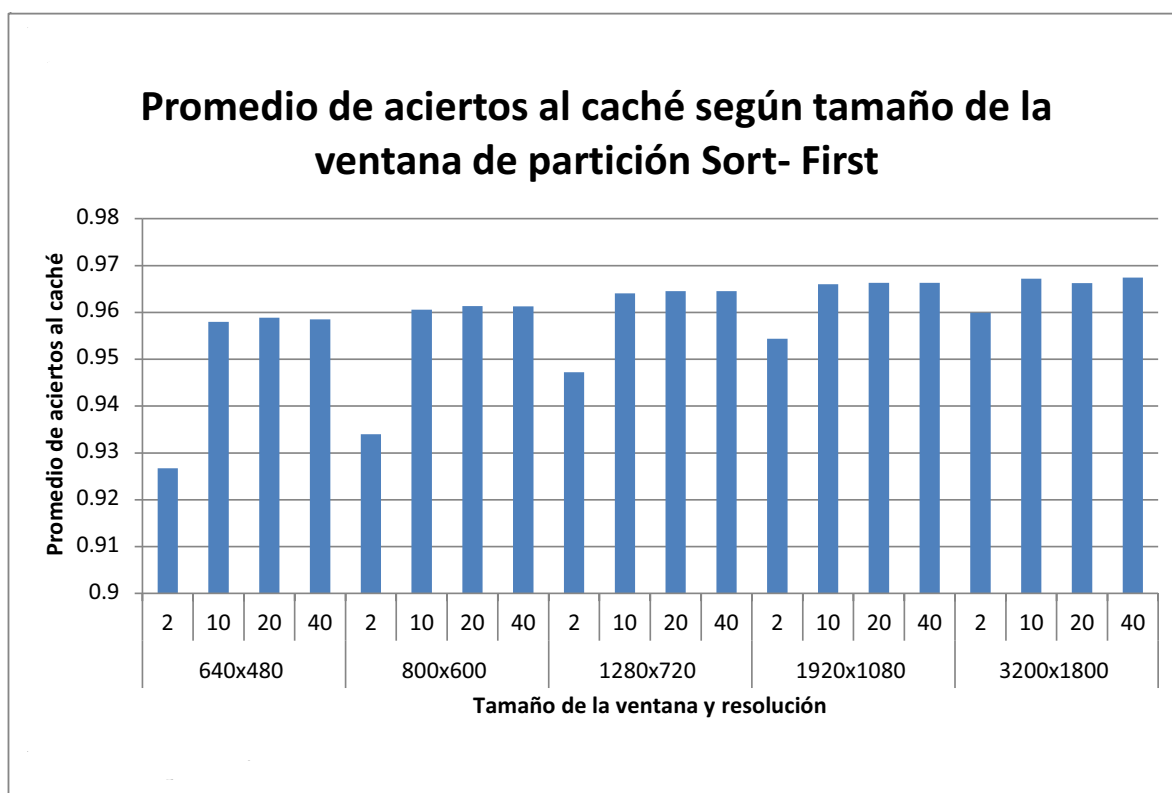


Figura 7.9: Promedio de Hits a caché para todas las escenas a distintas resoluciones

7.7 Análisis del despeño de los cachés jerárquicos.

Observando las figuras 7.10 y 7.11 es posible concluir varias cosas con respecto a la jerarquía de los cachés. Conforme aumenta la resolución el número de accesos promedio al cada uno de los caches es mayor, y de igual manera el porcentaje de aciertos aumenta. El caché correspondiente a la profundidad $N = 8$ del SVO es el que tiene menor porcentaje de aciertos y a la vez el que tiene menor número de accesos promedio, esto confirma las observaciones encontradas por [27] con respecto a que los niveles inferiores de un SVO tienen un menor número de accesos. Esto se puede apreciar claramente de la figura 7.13

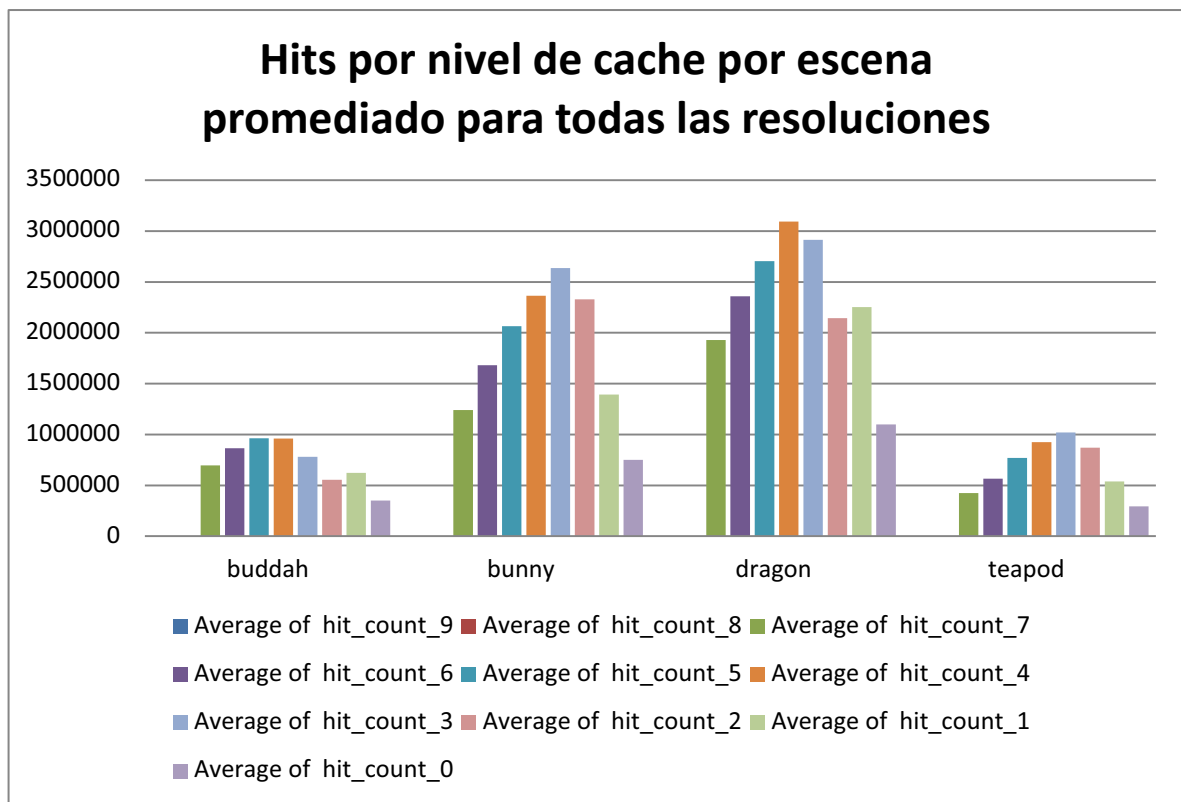


Figura 7.10: Promedio de aciertos al los diferentes niveles de caché para un SVO de profundidad (promedio por escena para todas las resoluciones).

La figura 7.12 muestra la tasa de Hits por nivel del caché, para cada escena y para cada una de las distintas resoluciones. Como es de esperarse los caché de nivel cero (franjaz azules) y de nivel 1 (franjaz rojas) tienen un hit rate promedio de casi 100 %

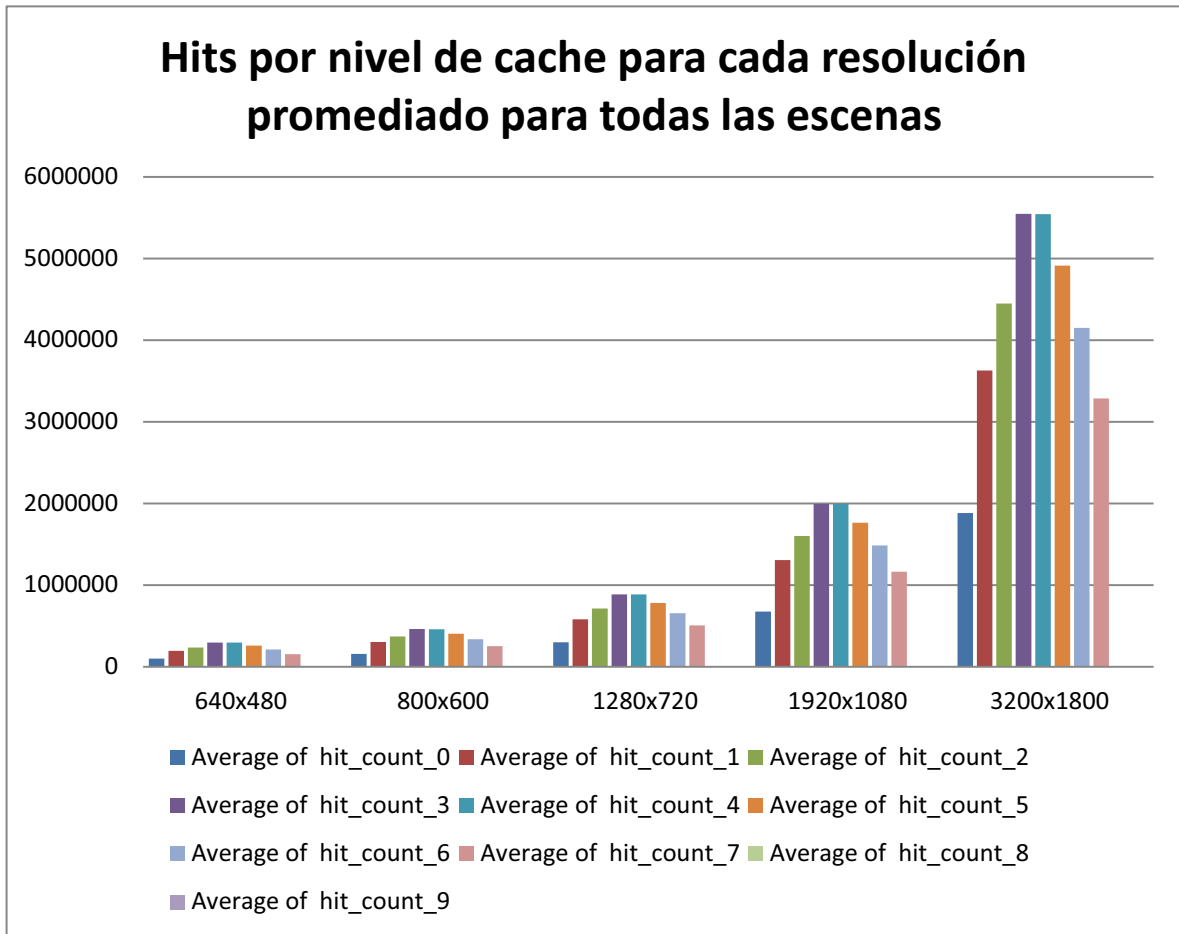


Figura 7.11: Promedio de aciertos al los diferentes niveles de caché para un SVO de profundidad 9 (promedio de Hits por resolución para todas las escenas).

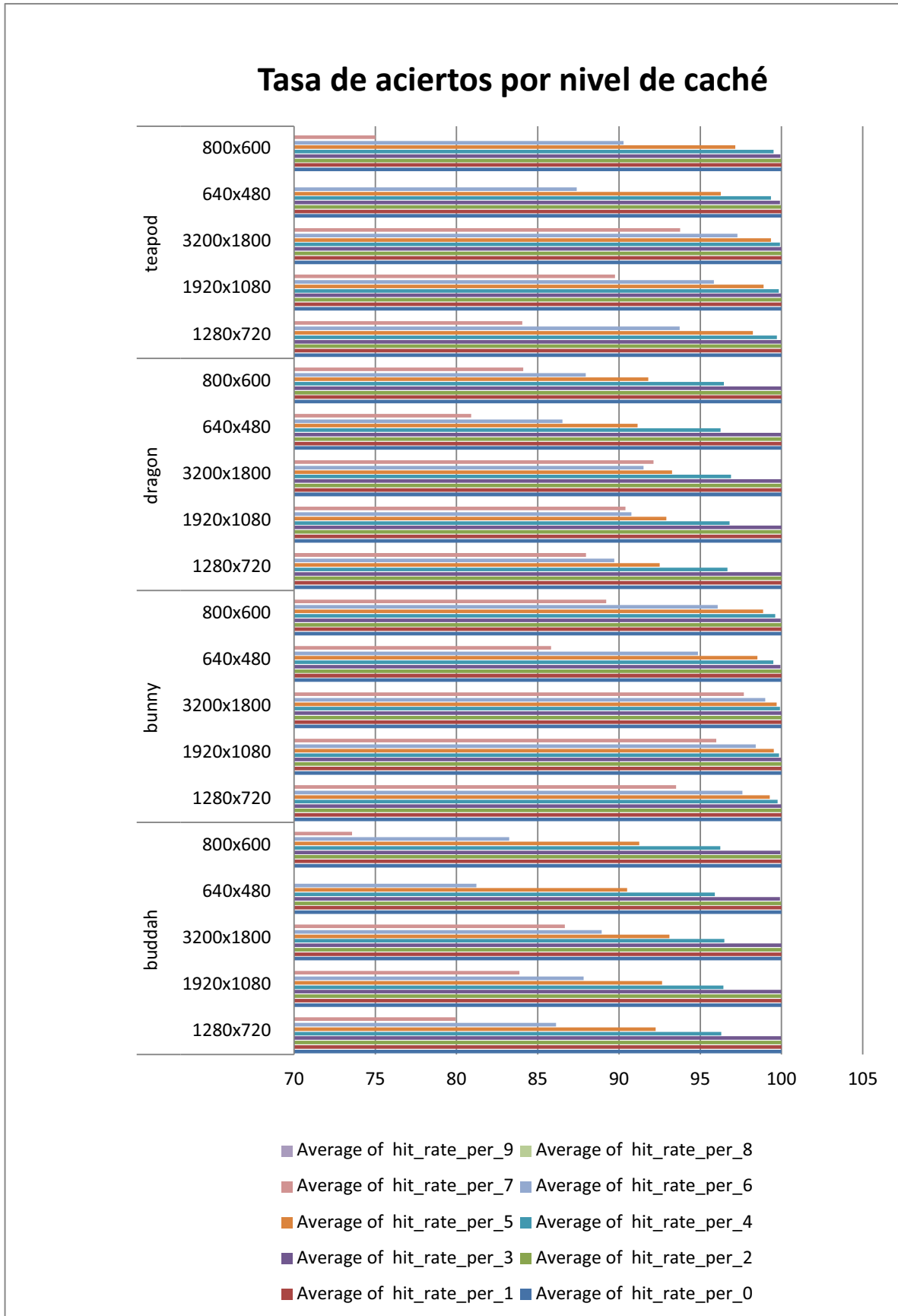


Figura 7.12: Tasa de aciertos al los diferentes niveles de caché para un SVO de profundidad 10

7.7. Análisis del despeño de los cachés jerárquicos.

para todas las escenas y resoluciones. Es interesante ver que los niveles de caché correspondientes a los niveles superiores del árbol hasta el nivel 5 tienen promedios de acierto iguales o superiores al 100%. Esto es importante ya que si se decide en el futuro implementar un esquema en el que el recorrido por profundidad se detenga después de cierto nivel de detalle, entonces se puede usar el hecho de que promedio de aciertos en el sistema solo se degrada después del nivel 5. Finalmente la figura 7.12 reafirma el hecho de que los niveles del caché más cercanos a las hojas del árbol tiene un promedio de aciertos cercano a cero.

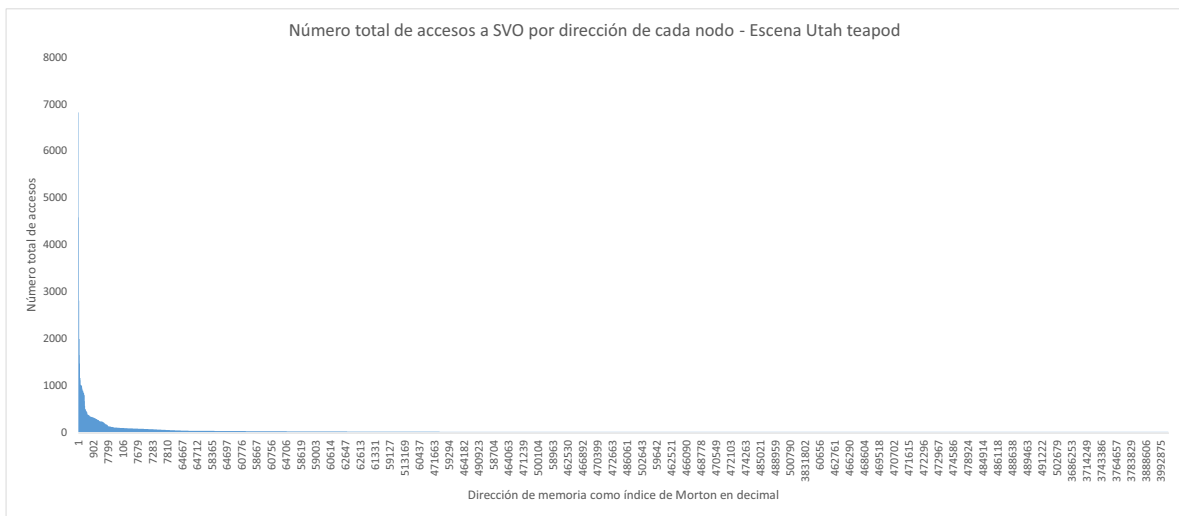


Figura 7.13: Número de accesos al SVO por nodo. Los nodos superiores (direcciones más bajas) son accedidos más veces

En la figura 7.13 el eje X tiene las direcciones de cada uno de los nodos del SVO que fueron accedidos durante la simulación. En esta codificación, las direcciones más altas representan los nodos superiores del árbol. Se puede notar como la dirección 1 tiene el mayor número de accesos. Esto se debe que la dirección 1 corresponde a la raíz del árbol, y por lo tanto cada rayo deberá solicitar una lectura a la raíz al menos una vez. Los direcciones más altas corresponden usualmente a las hojas del SVO por lo que el número de accesos es naturalmente menor.

7.7.1 Análisis del tamaño total de la caché

La definición de una arquitectura de caché toma en cuenta 3 parámetros de optimización [5]: el tamaño del caché, el tamaño de la línea de caché y el tipo de asociatividad.

El tamaño del caché se relaciona directamente con el conjunto de profundidades del árbol de octantes de la escena. Como se menciona en la sección 6.2 cada línea del caché guarda 16 bits de datos y 32 bits de direcciones. Para este análisis es necesario recordar de la sección 6.2 que los cachés asociados a los primeros tres niveles de profundidad del SVO tienen cada uno 8, 64 y 512 líneas correspondientemente.

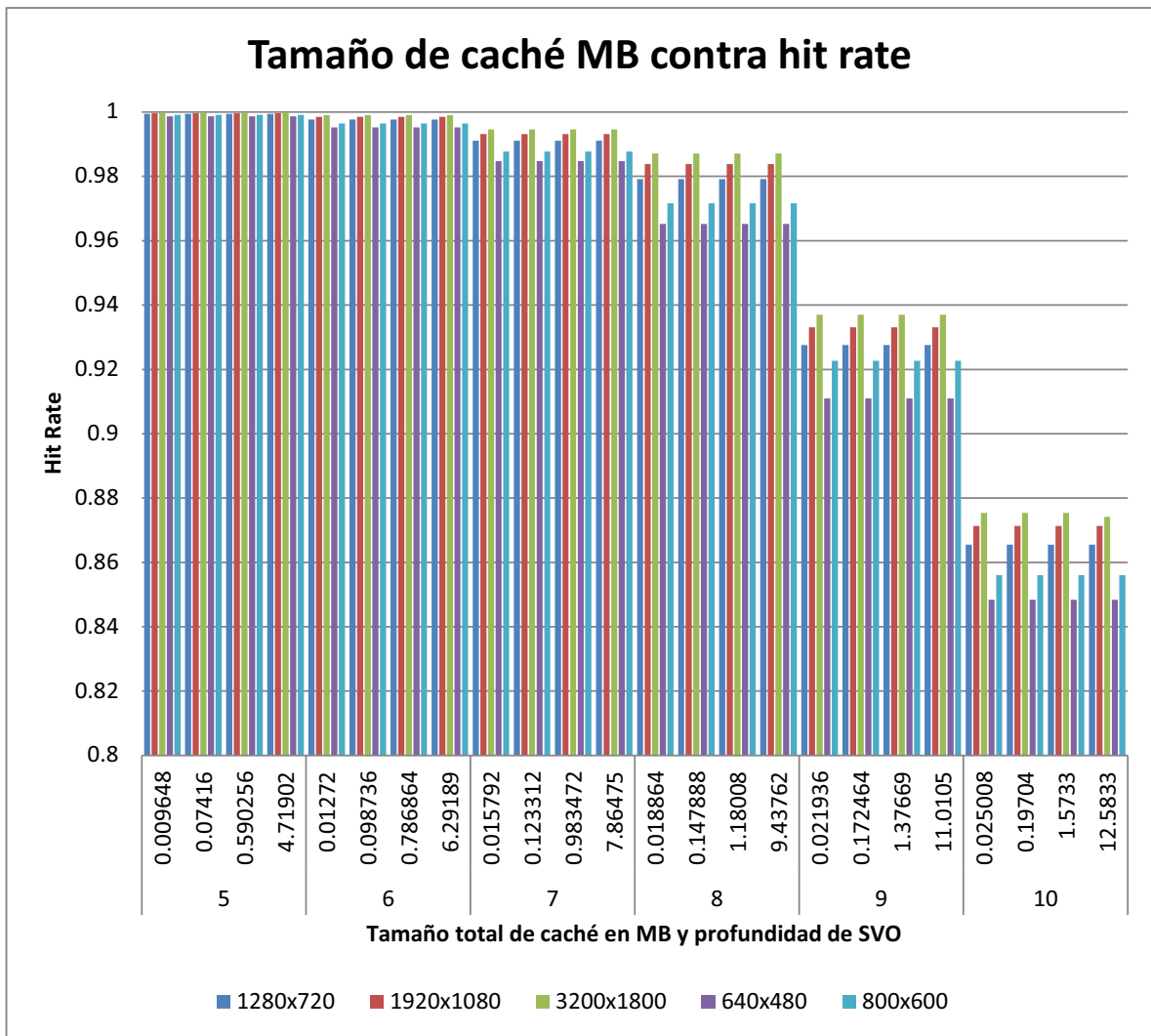


Figura 7.14: Promedio de Hits al caché y tamaños en MB de caché según profundidad del SvO por resolución (promedio para todas las escenas)

7.7. Análisis del despeño de los cachés jerárquicos.

La figura 7.14 analiza el tamaño total del sistema de caché (la suma del tamaño de los N sub-caches) para distintas profundidades del SVO y distintas resoluciones. Esta figura promedia los resultados para todas las escenas. Resulta claro de la figura 7.14 que existe una cota superior en tamaño del caché, luego la de cual no existe una mejora el tasa de aciertos. Esta cota superior depende de la profundidad del SVO, por ejemplo la figura 7.14 muestra como para un SVO de profundidad 10 sin importar cuanto aumente el tamaño en MB de los cachés, nunca se logra superar una tasa de aciertos promedio de aproximadamente 87%.

La tabla 7.7.1 es una resumen de los tamaños recomendados para el caché según los datos experimentales. En otras palabras los datos de la tabla 7.7.1 son los valores en kB luego de los cuales no hay mayor ganancia en hit rate para una profundidad de SVO dada.

Resumen tamaños recomendados de caché en kB		
Profundidad SVO	Tamaño aproximando MB	Promedio Hit Rate
5	9.6kB	99 %
6	12.72kB	99 %
7	15.72kB	98 %
8	18.8kB	96 %
9	21.9kB	91 %
10	25kB	85 %

7.7.2 Profundidad recomendada del SVO

Como pudimos observar a partir de los resultados, la profundidad del árbol SVO muestra ser el factor que tiene mayor influencia en el cantidad de datos transferidos desde el OsM hacia el resto del sistema. Es importante tomar en cuenta que la pro-

fundidad del SVO influencia además otra serie de parámetros del sistema, por ejemplo la codificación de Morton que crece en 3 bits por cada nivel de profundidad del SVO, provocando a su vez que los tamaños de los LIFOS y los buses de la arquitectura del sistema se vean afectados.

Desde luego, la herramienta de simulación de software permite convenientemente modificar todos estos parámetros internos, no obstante, dado el alto costo de verificación de un modelo de RTL, se recomienda que estos sean invariantes para un implementación final en Hardware.

Por lo anterior, y basados en los resultados de las simulaciones para las distintas escenas, tamaños de ventana y resoluciones, se procede a recomendar un valor de profundidad del SVO entre 8 y no mayor a 9. De esta manera se mantiene una alta tasa de aciertos en el caché y a la vez se tiene un buen nivel de detalle en las superficies aproximadas mediante voxels.

Conclusiones

“Scientific truth is beyond loyalty
and disloyalty.”

Isaac Asimov, Foundation

8.1 Conclusiones

En esta tesis se ha presentado el esbozo de una arquitectura práctica que permite el Ray-Casting para SVO disminuyendo considerablemente el tráfico de datos entre la memoria de almacenamiento de la escena y el GPU. Se han especificado los principales bloques funcionales del sistema, así como su funcionamiento en alto nivel y las interacciones principales entre estos. Se ha llevado a cabo un análisis del desempeño del sistema propuesto en cuanto a los accesos a la memoria de almacenamiento de objetos. Para este análisis se emplearon como métrica el conteo de accesos de lectura en función de parámetros tales como la resolución y la profundidad del SVO. Adicionalmente se ha analizado el desempeño de los accesos a la memoria de almacenamiento de objetos del sistema propuesto, empleando como métrica la tasa de aciertos en las memorias cache del sistema propuesto en función de las particiones de la pantalla por ordenamiento sort-first.

Se ha visto el valor de implementar un simulador en software del sistema propuesto. El uso de esta herramienta facilita el diseño de los algoritmos, la elección de patrones de diseño a nivel arquitectónico, así como estimaciones de desempeño de bloques de alto nivel e interacciones sin tener que preocuparse de los detalles específicos de una implementación a nivel de RTL tales como Glitches, carreras de estados, etc. Por otro lado, el modelo de software presentado en esta tesis puede ser empleado como un modelo de referencia para validar el comportamiento de algún otro modelo de RTL que se desee implementar en el futuro.

Se puede concluir de los resultados de esta investigación que el uso de un caché de datos es esencial para mejorar el rendimiento del sistema. Además en esta investigación se han presentado parámetros recomendados de tamaño de caché y profundidad de SVO, y tamaño de la partición Sort-First. Los valores de estos parámetros que se han obtenido luego de analizar los datos experimentales podrán posteriormente ser usados en una implementación de hardware o un tipo FPGA o un ASIC.

A continuación se resume puntualmente las conclusiones más relevantes de esta investigación:

- El número promedio de accesos al la OsM se ve dramáticamente reducido gracias a la presencia del caché
- El sistema propuesto comienza a presentar degradación en el número de lecturas para profundidades superiores a 8
- El tamaño de la ventana tiene un efecto en el número promedio de accesos, siendo el tamaño de ventana más pequeño (2x2) aquel que muestra mejores resultados
- En una jerarquía de cachés, donde cada nivel corresponde a un nivel de profundidad del SVO, los niveles inferiores tienen un número menor de accesos que los niveles superiores. En particular, el nivel correspondiente a las hojas tiene tan pocos accesos que en general vale la pena una memoria para él.
- Los niveles de cache correspondientes a los niveles superiores del árbol hasta el nivel 5 tienen promedios de acierto iguales o superiores al 100 %. Esto es importante ya que si se decide en el futuro implementar un esquema en el que el recorrido por profundidad se detenga después de cierto nivel de detalle, entonces se puede usar el hecho de que promedio de aciertos en el sistema solo se degrada después del nivel 5
- Existe una cota superior en tamaño del cache, luego la de cual no existe una mejora el tasa de aciertos. Esta cota superior depende de la profundidad del SVO, por ejemplo, para un SVO de profundidad 10 sin importar cuanto aumente el tamaño en MB de los caches, nunca se logra superar una tasa de aciertos promedio de aproximadamente 87 %

A continuación se resume puntualmente las recomendaciones para la arquitectura propuesta basados en los resultados experimentales:

- **Profundidad de SVO recomendada de 9:** Esto resulta un cache de 21.9kB con un tasa de aciertos promedio de 91
- **Tamaño de particion sort-first de 2x2:** Los experimentos indican que este tamaño de ventana da la mayor coherencia de los rayos (aproximación de un paquete de rayos)

Trabajo Futuro

“Past glories are poor feeding.”

Isaac Asimov, Foundation

9.1 Trabajo Futuro

Inclusive con la arquitectura presentada, aun queda mucho trabajo por hacer antes de que el diseño del sistema como un todo esté completo y sea óptimo para una futura implementación en hardware. Luego de esta investigación se reafirma el hecho de que uno de los mayores cuellos de botella en cualquier sistema basado en voxeles esta asociado a los requisitos de memoria. El presente trabajo se enfocaba en buscar una reducción del trafico de datos entre esta memoria y el GPU, pero es importante mencionar que este enfoque puede ser combinado con otras estrategias como la compresión de los datos. Con el uso de compresión se puede almacenar mayor detalle, entornos y escenas más grandes, además de un mejor rendimiento debido a un menor número de fallos de caché y menos datos que necesitan ser transferidos de una memoria a otra. Entonces el siguiente problema que este nuevo sistema debe solventar es encontrar una buena forma de comprimir eficientemente los datos volumétricos. Una opción interesante consiste en emplear una compresión basada en Hashes perfectos para datos espaciales como la propuesta en [11].

Otro tema que está dentro del trabajo futuro tiene que ver con cálculo de vectores normales e iluminación. De nuevo, el agregar iluminación al sistema implica que una cantidad considerablemente mayor de información deberá ser almacenada en la memoria de la escena. No obstante es posible emplear algún tipo de aproximación para calcular los valores de los vectores normales durante la ejecución del programa en lugar de mantener estos almacenados en la memoria.

Finalmente, el visualizar superficies voxelizadas que contengan información de texturas es uno de los aspectos más importantes que se deben investigar para este sistema. Para esto puede resultar interesante explorar la implementación en hardware de enfoques como el de Forestmann [3] que incluyen compresión de texturas RLE, e incluso un enfoque basado en Mip-Maps como en [2].

Bibliografía

- [1] Appel, A. (1968). 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. ACM.
- [2] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. (2009). Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, I3D '09*, pages 15–22, New York, NY, USA. ACM.
- [3] Forstmann, S. and Ohya, J. (2010). Efficient, high-quality, gpu-based visualization of voxelized surface data with fine and complicated structures. *IEICE Transactions*, 93-D(11):3088–3099.
- [4] Hardware, T. Next-gen 3d rendering technology: Voxel ray casting.
- [5] Hennessy, J. and Patterson, D. (2002). *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science.
- [6] Hollemeersch, C., Pieters, B., Lambert, P., and Van de Walle, R. (2009). Accelerating virtual texturing using cuda. In *GPU Technology Conference, Abstracts*.
- [7] Hollemeersch, C.-F., Pieters, B., Demeulemeester, A., Cornillie, F., Semmertier, B. V., Mannens, E., Lambert, P., Desmet, P., and de Walle, R. V. (2010). Infinitex: An interactive editing system for the production of large texture data sets. *Computers Graphics*, 34(6):643 – 654. Graphics for Serious Games Computer Graphics in Spain: a Selection of Papers from {CEIG} 2009 Selected Papers from the {SIGGRAPH} Asia Education Program.
- [8] Laine, S. and Karras, T. (2010). Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*.

- [9] Laine, S. and Karras, T. (2011a). Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059.
- [10] Laine, S. and Karras, T. (2011b). Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17:1048–1059.
- [11] Lefebvre, S. and Hoppe, H. (2006). Perfect spatial hashing. In *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06*, pages 579–588, New York, NY, USA. ACM.
- [12] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. (1994). A sorting classification of parallel rendering.
- [13] Montgomery, D. C. (2001). *Design and Analysis of Experiments*. John Wiley & Sons.
- [14] Morton., G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing.
- [15] Nickolls, J. and Dally, W. J. (2010). The gpu computing era. *IEEE Micro*, 30(2):56–69.
- [16] Project, R. (2013). What is r. [Online; accessed 25-May-2013].
- [17] R., M. A. (2009). Power mbx technology review. [Online; accessed 25-May-2014].
- [18] Revelles, J., Ureña, C., and Lastra, M. (2000). An efficient parametric algorithm for octree traversal. In *WSCG*.
- [19] Roth, S. D. (1982). Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144.
- [20] Sanabria Brenes, G. (2011). *Comprendiendo la estadística inferencial*. Editorial Tecnológica de Costa Rica.
- [21] Savitz, E. (2014). The blender project.
- [22] Schmittler, J., Wald, I., and Slusallek, P. (2002). Saarcor: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '02*, pages 27–36, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.
- [23] SinnDevelopment (2014). Unreal engine 4. [Online; accessed 25-Aug-2014].
- [24] Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. (1974). A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55.
- [25] Wald, I. and Slusallek, P. (2001). State of the art in interactive ray tracing.
- [26] Walpole, R., Myers, R., and Myers, S. (2010). *Probability and Statistics for Engineers and Scientists*. Pearson Education, Limited.
- [27] Wilhelmsen, A. (2012). Efficient ray tracing of sparse voxel octrees on an fpga.

Acrónimos

- CAD** Computer Asisted Design: es un software que se emplea para facilitar las labores de diseño gráfico.
- URL** Universal Resource Locator: es una referencia a un recurso en Internet.
- RTL** Register transfer level: Se refiere a la representación de HDL a nivel de transferencia de registros.
- SVO** Spare Voxel Octree: Tipo de árbol de octantes en el cual la mayoría de los octantes están vacíos
- HDL** Hardware Description Language: Se refiere a los lenguajes para descripción de sistemas de hardware.
- HW** Hardware: se refiere a todas las partes tangibles de un sistema informático; sus componentes son: eléctricos, electrónicos, electromecánicos y mecánicos
- SW** Software: equipamiento lógico o soporte lógico de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas, en contraposición a los componentes físicos
- API** Application Programming Interface: Interface de programación de Aplicaciones.
- AABB** Axis Alligned Bounding Box: Prisma con cada arista alineada a un eje cartesiano
- DoE** Design of Experiments: Técnica estadística que permite identificar y cuantificar las causas de un efecto dentro de un estudio experimental
- OsM** Object Storage Memory: se refiere a la memoria en el GPU que almacena los objetos de la escena
- DRAM** Dynamic Random Acces Memory: se refiere a una memoria de acceso aleatorio tipo dinámica
- SRAM** Static Random Access Memory: Memoria de acceso aleatorio de tipo estática

GPU Graphics Processing Unit: Unidad de procesamiento para gráficos por computadora

ROM Read Only Memory: Memoria de solo lectura

GUI Graphical User interface: Interfaz de usuario gráfica

LRU Least recently used: Política de remplazo para memorias caché en la cual el elemento mas viejo es remplazado

BVH Bounding Volume Hierarchy: Se refiere a un jerarquía de volúmenes como por ejemplo un jerarquía de AABB

VGA Vector Graphics Array: Se refiere a las resoluciones de 640x480.

SVGA Super VGA: Normalmente de refiere a una resolución de 800x600 píxeles.

RLE Run Length Encoding: Una forma muy simple de la compresión de datos en la que corridas de datos (runs) se almacenan como un valor y un conteo.

.1 Tablas de resultados de experimentos

Resultados de ANOVA escena Utah Teapod, para un solo núcleo

	Df
octree_depth	1
grid_partition_size	1
resolution	1
cache_lines_per_way	1
octree_depth:grid_partition_size	1
octree_depth:resolution	1
grid_partition_size:resolution	1
octree_depth:cache_lines_per_way	1
grid_partition_size:cache_lines_per_way	1
resolution:cache_lines_per_way	1
octree_depth:grid_partition_size:resolution	1
octree_depth:grid_partition_size:cache_lines_per_way	1
octree_depth:resolution:cache_lines_per_way	1
grid_partition_size:resolution:cache_lines_per_way	1
octree_depth:grid_partition_size:resolution:cache_lines_per_way	1
Residuals	304
	Sum Sq
octree_depth	0.014856
grid_partition_size	0.005248
resolution	0.005481
cache_lines_per_way	0.000629
octree_depth:grid_partition_size	0.002885
octree_depth:resolution	0.003497
grid_partition_size:resolution	0.001138
octree_depth:cache_lines_per_way	0.000223
grid_partition_size:cache_lines_per_way	0.000027
resolution:cache_lines_per_way	0.000011
octree_depth:grid_partition_size:resolution	0.000510

.1. Tablas de resultados de experimentos

octree_depth:grid_partition_size:cache_lines_per_way	0.000002
octree_depth:resolution:cache_lines_per_way	0.000002
grid_partition_size:resolution:cache_lines_per_way	0.000003
octree_depth:grid_partition_size:resolution:cache_lines_per_way	0.000000
Residuals	0.020333
	Mean Sq
octree_depth	0.014856
grid_partition_size	0.005248
resolution	0.005481
cache_lines_per_way	0.000629
octree_depth:grid_partition_size	0.002885
octree_depth:resolution	0.003497
grid_partition_size:resolution	0.001138
octree_depth:cache_lines_per_way	0.000223
grid_partition_size:cache_lines_per_way	0.000027
resolution:cache_lines_per_way	0.000011
octree_depth:grid_partition_size:resolution	0.000510
octree_depth:grid_partition_size:cache_lines_per_way	0.000002
octree_depth:resolution:cache_lines_per_way	0.000002
grid_partition_size:resolution:cache_lines_per_way	0.000003
octree_depth:grid_partition_size:resolution:cache_lines_per_way	0.000000
Residuals	0.000067
	F value
octree_depth	222.110
grid_partition_size	78.464
resolution	81.942
cache_lines_per_way	9.404
octree_depth:grid_partition_size	43.139
octree_depth:resolution	52.279
grid_partition_size:resolution	17.020
octree_depth:cache_lines_per_way	3.329
grid_partition_size:cache_lines_per_way	0.401
resolution:cache_lines_per_way	0.171
octree_depth:grid_partition_size:resolution	7.631
octree_depth:grid_partition_size:cache_lines_per_way	0.027
octree_depth:resolution:cache_lines_per_way	0.031
grid_partition_size:resolution:cache_lines_per_way	0.052
octree_depth:grid_partition_size:resolution:cache_lines_per_way	0.000
Residuals	
	Pr(>F)
octree_depth	< 2e-16
grid_partition_size	< 2e-16
resolution	< 2e-16
cache_lines_per_way	0.00236
octree_depth:grid_partition_size	2.21e-10

.1. Tablas de resultados de experimentos

octree_depth:resolution	3.92e-12
grid_partition_size:resolution	4.78e-05
octree_depth:cache_lines_per_way	0.06907
grid_partition_size:cache_lines_per_way	0.52727
resolution:cache_lines_per_way	0.67908
octree_depth:grid_partition_size:resolution	0.00609
octree_depth:grid_partition_size:cache_lines_per_way	0.86985
octree_depth:resolution:cache_lines_per_way	0.86048
grid_partition_size:resolution:cache_lines_per_way	0.81976
octree_depth:grid_partition_size:resolution:cache_lines_per_way	0.98883
Residuals	

octree_depth	***
grid_partition_size	***
resolution	***
cache_lines_per_way	**
octree_depth:grid_partition_size	***
octree_depth:resolution	***
grid_partition_size:resolution	***
octree_depth:cache_lines_per_way	.
grid_partition_size:cache_lines_per_way	
resolution:cache_lines_per_way	
octree_depth:grid_partition_size:resolution	**
octree_depth:grid_partition_size:cache_lines_per_way	
octree_depth:resolution:cache_lines_per_way	
grid_partition_size:resolution:cache_lines_per_way	
octree_depth:grid_partition_size:resolution:cache_lines_per_way	
Residuals	

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1