



Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

MODELO DE CORTES VOLUMÉTRICOS PARA LA SIMULACIÓN DE CIRUGÍA DE HUESOS

Br. Carlos Guía

Héctor Navarro, Tutor

Caracas, 27 de junio del 2011



Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

MODELO DE CORTES VOLUMÉTRICOS PARA LA SIMULACIÓN DE CIRUGÍA DE HUESOS

Br. Carlos Guía

Héctor Navarro, Tutor

Caracas, 27 de junio del 2011

MODELO DE CORTES VOLUMÉTRICOS PARA LA SIMULACIÓN DE CIRUGÍA DE HUESOS

Br. Carlos Guía

*Trabajo Especial de Grado presentado
ante la ilustre Universidad Central de Venezuela
como requisito parcial para optar al título de
Licenciado en Computación.*

Quienes suscriben, miembros del Jurado que examinó el trabajo presentado por el **Br. Carlos Guía**, titulado: “**Modelo de Cortes Volumétricos para la Simulación de Cirugía de Huesos**” para optar al título de Licenciado en Computación, consideramos que dicho trabajo cumple con los requisitos exigidos por los reglamentos respectivos y por lo tanto lo declaramos APROBADO en nombre de la Universidad Central de Venezuela.

Héctor Navarro, Tutor

Fecha

Ernesto Coto

Fecha

Jaime Blanco

Fecha

Caracas, 27 de junio del 2011

A mi abuela Sila, de lo único que me arrepiento es que no puedas ver este momento

Agradecimientos

Le agradezco a mi mamá que siempre haya estado cerca para ayudarme y apoyarme cuando la he necesitado, y más de lo que puedo expresar en un párrafo.

Le agradezco a mi papá el haberme enseñado a programar desde niño y su disposición para discutir problemas, sin duda la mayor influencia en el camino que he tomado.

Le agradezco a mis hermanos, Rafa y Auri, toda su ayuda, apoyo e interés desde el comienzo en mi incursión en el mundo de las computadoras.

Le agradezco a mi novia Days su cariño, ayuda y apoyo constante, sin los cuales no podría estar presentado este trabajo en este momento.

Le agradezco a Héctor, primer *coach* y tutor de este trabajo, todo lo que me ha enseñado desde los comienzos de la carrera.

Gracias a Ernesto por todo su apoyo durante la realización de este trabajo y por no explotar al leer “*onion.in*” en donde no debía estar.

Gracias a William, segundo *coach*, por su confianza y apoyo en uno de los momentos más importantes.

Gracias a Rhadamés, tercer *coach*, por toda su gran amistad y todas las horas compartidas en *q3ctf1.bsp*.

Gracias a Robinson por su invaluable ayuda a lo largo de la carrera, sin ti seguro que aún estaría cursando sistemas de información.

Gracias a Omaira, Walter y el resto del Centro de Computación Gráfica por su confianza y apoyo durante la carrera.

Gracias a Juan Carlos e Ivens por su apoyo y amistad, sin ustedes la carrera no habría tenido ningún sentido.

Gracias a Jorge por su amistad y continua ayuda en el desarrollo de mis habilidades, has sido una pieza clave en mucho de lo que he logrado.

Gracias a Karima por estar siempre pendiente y ayudarme en todo lo que la he necesitado, incluso antes de yo enterarme.

Gracias a “*root ? destroy : 0*” por los años de sana competencia y obligarme a mejorar en el proceso.

Gracias a Trino por todo su esfuerzo en mejorar la calidad e importancia de los maratones de programación en la universidad y el país.

Lamentablemente es imposible nombrarlos a todos, sin embargo, le agradezco a todos los profesores, compañeros y familiares que siempre me han brindado su ayuda y apoyo a lo largo de la carrera.

Finalmente, le agradezco a Shigeru Miyamoto, John Carmack, Altaïr Ibn-La’Ahad, Ezio Auditore, Nathan Drake, y muchos más; las invaluables horas de diversión y por mostrarme el camino que deseo seguir.

Resumen

Modelo de Cortes Volumétricos para la Simulación de Cirugía de Huesos

Carlos Guía

Héctor Navarro, Tutor

Universidad Central de Venezuela

Los costos involucrados en los cursos de taladrado de hueso utilizando cadáveres disecados han generado una demanda para métodos alternos. Las simulaciones computarizadas ofrecen la oportunidad de reducir los costos de aprendizaje, práctica y planificación de cirugías. Sin embargo, las técnicas de modelado de sólidos tradicionales no ofrecen el grado de realismo necesario en el área de simulaciones médicas. El presente Trabajo Especial de Grado propone un modelo para representar, desplegar y manipular cortes arbitrarios sobre sólidos volumétricos. Además muestra la capacidad de utilizar dicho modelo para la simulación realista de cirugías de hueso.

Los resultados obtenidos muestran que el modelo es capaz de generar imágenes de alta calidad en tiempo real, capacidades requeridas en los sistemas de simulaciones quirúrgicas.

Héctor Navarro
Tutor

Índice General

Resumen	iv
Índice General	v
Introducción	1
1. Propuesta de Trabajo Especial de Grado	4
1.1. Planteamiento del problema	4
1.2. Objetivos	5
1.2.1. Objetivo General	5
1.2.2. Objetivos Específicos	5
2. Marco Teórico	6
2.1. CUDA	6
2.1.1. Modelo de Programación Escalable	7
2.1.2. Jerarquía de Hilos	8
2.1.3. Jerarquía de Memoria	10
2.1.4. Programación Heterogénea	10
2.1.5. Consideraciones de Rendimiento	11
2.1.5.1. Arquitectura SIMT	13
2.1.5.2. Maximizando la Utilización	15
2.1.5.2.1. Nivel de Aplicación	15
2.1.5.2.2. Nivel de Dispositivo	16
2.1.5.2.3. Nivel de Multiprocesador	16
2.1.5.3. Maximizar la Tasa de Transferencias Efectivas . .	17

2.1.5.3.1.	Memoria Global	18
2.1.5.3.2.	Memoria Compartida	19
2.1.5.3.3.	Memoria Constante	19
2.1.5.3.4.	Memoria de Textura	20
2.1.5.4.	Maximizar la Tasa de Instrucciones Ejecutadas .	20
2.2.	Despliegue Volumétrico	21
2.2.1.	Emisión de Rayos	22
2.2.1.1.	Modelo Blinn-Kajiya	22
2.2.1.2.	Algoritmo Básico	24
2.2.1.3.	Despliegue de Isosuperficies	27
2.3.	Simulaciones médicas y trabajo de referencia	28
2.3.1.	Representación Multivolumétrica	29
2.3.2.	Corte de Volumen	29
2.3.3.	Corte Progresivo	30
2.3.4.	Visualización Multivolumétrica	31
3.	Diseño	33
3.1.	BoneSurgeryLib	33
3.2.	BoneSurgeryDemo	37
3.3.	BoneSurgeryLibWrapper	38
3.4.	BoneSurgeryTester	38
3.4.1.	Medición Correcta de Tiempo para Operaciones de CUDA	39
3.4.2.	Reloj de Alta Precisión	40
4.	Implementación	42
4.1.	Corte de Volumen	42
4.1.1.	Voxelización del Instrumento	42
4.1.1.1.	Combinación de los Resultados	47
4.2.	Despliegue Volumétrico	48
4.2.1.	Emisión de Rayos en Paralelo	48
4.2.2.	Detección de Isosuperficies con un Volumen de Corte . .	49

4.2.3.	Estimación del Gradiente en la Frontera de las Superficies de Corte	53
4.3.	Detección y Manipulación de Fragmentos Separados	58
4.3.1.	Abstracciones Lógicas	59
4.3.2.	Detección de Componentes Conexas	60
5.	Análisis de Resultados de Pruebas	63
5.1.	Resultados Cuantitativos	63
5.1.1.	Sistema General	64
5.1.2.	Despliegue Volumétrico	66
5.1.3.	Manipulación de Fragmentos Separados	70
5.2.	Resultados Cualitativos	71
5.2.1.	Comparación con las Técnicas Tradicionales	72
5.2.2.	Comparación de las Diferentes Propuestas para Estimar el Vector Gradiente	73
5.2.3.	Calidad Visual de los Bordes Agudos	78
	Conclusiones	79
	Trabajos a Futuro	82
	Bibliografía	85

Introducción

Por muchos años la única forma de aprender, practicar y reforzar los diferentes procedimientos quirúrgicos, era utilizando cadáveres disecados. Como se puede imaginar esto tenía ciertas limitaciones, principalmente el tiempo y costo que toma la disección del mismo. Estos problemas crearon una creciente necesidad de métodos alternos, aunque la utilización de cadáveres disecados sigue siendo la herramienta de aprendizaje principal [1].

Una de las mejores opciones que ha surgido es la utilización de modelos sintéticos, la cual sigue siendo una buena alternativa por su bajo costo y alto grado de realismo. Pero también sufre de una gran limitación, modelar la gran variedad patológica del mundo real utilizando modelos generales estilizados es prácticamente imposible.

Estas limitaciones pueden ser resueltas incorporando simulaciones computarizadas en el proceso de capacitación médica. Es posible modelar la variedad patológica utilizando datos reales, capturados de una gran cantidad de personas, y el costo de repetir una cirugía en un sistema de simulación es despreciable. Por supuesto, no se puede ignorar el incremento en la velocidad de procesamiento, el aumento en la capacidad de almacenamiento y las grandes mejoras en los algoritmos de computación gráfica durante las últimas décadas, puesto que sin estos avances simplemente no se podría ofrecer una alternativa interactiva con el nivel de realismo necesario [21].

Uno de los avances tecnológicos más importantes ha sido la inclusión y evolución de la Unidad de Procesamiento Gráfico (GPU - Graphics Processing Unit), en computadoras personales. Las cuales, debido a la demanda insaciable en el mercado

por gráficos de alta fidelidad en tiempo real, se han vuelto procesadores altamente paralelos, multihilos, de muchos núcleos con un gran potencial de cómputo y alto ancho de banda de memoria [20].

El GPU ha sido especializado para cómputo paralelo de alta intensidad aritmética (exactamente lo que el despliegue de gráficos necesita), por lo tanto, más transistores son dedicados al procesamiento de datos en vez de control de flujo y cacheo de datos. Específicamente, el GPU se adecua especialmente bien a problemas que pueden ser expresados como cálculos paralelos sobre datos (el mismo programa se ejecuta sobre muchos datos en paralelo) de alta intensidad aritmética. Desde finales del año 2006, la corporación NVIDIA introdujo la arquitectura paralela de propósito general CUDA, la cual permite desarrollar aplicaciones que aprovechen al máximo las capacidades de cómputo paralelo del GPU [20].

Debido a la gran complejidad de la anatomía humana, los grupos que trabajan en simulaciones computarizadas para la capacitación médica suelen dividir el trabajo en módulos especializados. Por ello, el Centro de Computación Gráfica de la Universidad Central de Venezuela ha dividido este desarrollo en componentes específicos, algunos de los módulos ya implementados estudian el diseño de interfaces [23], la detección de colisiones [19] y la deformación de tejidos blandos [25], entre otros.

Este trabajo se enfoca en la creación de un nuevo componente, el cual servirá como fundamento para la simulación de cirugía de huesos. Este tipo de cirugías es altamente utilizada por los cirujanos para arreglar problemas tanto funcionales como estéticos. Parte de la cirugía de huesos involucra taladrar y remover astillas, lo cual puede ser simulado utilizando sustracciones volumétricas de los huesos y las herramientas [21]. Sin embargo, la mayoría de estas cirugías requiere cortar un hueso para obtener nuevos pedazos del mismo, los cuales pueden ser reposicionados o removidos, para alojar una prótesis o normalizar la morfología del esqueleto [22]. Por esta razón es necesario analizar la intersección de la herramienta de corte con el hueso y determinar si esta produce nuevos objetos. También es posible que un corte solo genere una brecha, pero

que varios cortes subsecuentes se intersecten generando nuevos fragmentos de hueso separados por completo. Sin embargo, cualquiera que sea caso, estas simulaciones tienen requerimientos muy diferentes a los utilizados para modelar cortes en la piel o membranas, ya que estos últimos requieren predicciones acertadas sobre deformaciones en el tiempo de superficies blandas [17].

El campo predominante en la visualización de datos médicos es el despliegue volumétrico, sin embargo, los modelos tradicionales de despliegue y manipulación de volúmenes sufren de artefactos visuales cuando se desea modelar cortes arbitrarios sobre los mismos [21]. Por lo tanto, es necesario extender tanto los esquemas de representación como los algoritmos de manipulación y despliegue para obtener el realismo deseado.

En el primer capítulo se realiza la propuesta del trabajo especial de grado, planteando de forma concreta el problema que se estudia y los objetivos que se desean alcanzar. El segundo capítulo establece el marco teórico utilizado para el resto del documento, en éste se incluye un análisis detallado de la arquitectura CUDA, un modelo general de despliegue volumétrico y el análisis de un trabajo con resultados muy prometedores en el área de simulación de cirugía de huesos. El tercer capítulo muestra el diseño general del sistema, incluyendo todas las bibliotecas y aplicaciones desarrolladas. El cuarto capítulo profundiza en los aspectos más importantes del sistema de simulación, abarcando los detalles de implementación de las funcionalidades más importantes. El quinto capítulo muestra el análisis de los resultados de las pruebas, el cual incluye resultados cuantitativos como tiempos de ejecución y despliegue; y resultados cualitativos como la calidad visual. Para finalizar, se resumen las conclusiones obtenidas durante la investigación y desarrollo, tomando en cuenta los resultados de las pruebas y se proponen posibles trabajos a futuro para la continuación de la investigación y desarrollo en el área.

Capítulo 1

Propuesta de Trabajo Especial de Grado

1.1. Planteamiento del problema

Actualmente el aprendizaje, práctica y planificación de intervenciones quirúrgicas de hueso consume recursos costosos, tales como horas de laboratorio y cadáveres disecados. La escasez de dichos recursos genera la necesidad de métodos alternos, los modelos sintéticos han sido de gran utilidad para reducir estos costos, pero tienen limitaciones a la hora de modelar la diferencias que existen en el mundo real.

Últimamente las simulaciones computarizadas han adquirido un gran valor en dicha área, ya que tienen la capacidad de reducir los costos del aprendizaje y sobrellevar la limitación de los modelos sintéticos. Pero aún es un área en desarrollo que tiene sus propias limitaciones, es por ello que las investigaciones científicas tienen aún mucho que aportar. Una limitación importante es el modelado de cortes arbitrarios sobre volúmenes médicos, los métodos tradicionales se encuentran limitados por la resolución del volumen o introducen artefactos visuales, los cuales son suficientemente pequeños para ser aceptados en muchas áreas, pero suficientemente notables para ser rechazados en el área de simulación médica.

Tomando en cuenta lo expuesto anteriormente se propone el diseño e implementación de un modelo para la simulación de cortes arbitrarios sobre sólidos volumétricos, dicho modelo ha de incluir esquemas de representación, algoritmos de manipulación y

algoritmos de despliegue que sirvan como fundamento para la simulación computarizada de cirugía de huesos.

1.2. Objetivos

1.2.1. Objetivo General

Diseñar e implementar un modelo para la simulación de cortes de hueso en un sistema de simulaciones computarizadas, el modelo ha de permitir representar, manipular y desplegar cortes de hueso de forma realista y en tiempo real.

1.2.2. Objetivos Específicos

- Diseñar un esquema de representación para modelar cortes arbitrarios sobre sólidos volumétricos.
- Diseñar algoritmos de despliegue que permitan la visualización, libre de artefactos, de cortes arbitrarios sobre sólidos volumétricos.
- Diseñar algoritmos de manipulación que permitan la detección y manipulación de fragmentos de hueso.
- Implementar la estrategia propuesta utilizando el lenguaje de programación C++, la arquitectura CUDA y el estándar gráfico OpenGL.
- Implementar una biblioteca que muestre el uso del modelo desarrollado para la simulación de cortes de hueso en tiempo real.
- Implementar una aplicación para demostrar el uso de la biblioteca desarrollada.
- Utilizar la metodología de desarrollo de programación extrema y UML para la especificación de las clases.

Capítulo 2

Marco Teórico

2.1. CUDA

En Noviembre de 2006, NVIDIA introdujo una arquitectura de propósito general para cómputo paralelo denominada CUDATM. En ella se incluyó un nuevo modelo de programación y conjunto de instrucciones, que permiten utilizar las capacidades de paralelismo de los GPUs de NVIDIA para resolver problemas computacionales complejos de forma eficiente [20].

CUDA viene con un ambiente de desarrollo que permite utilizar C como lenguaje de programación de alto nivel, aunque como se muestra en la Figura 2.1, también existen otras interfaces de programación de aplicaciones.

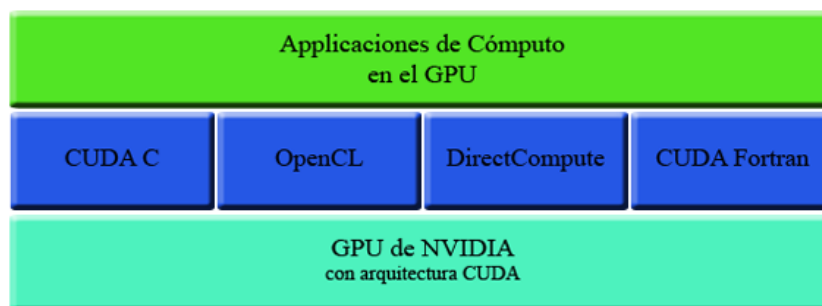


Figura 2.1: Soporte de diferentes lenguajes e interfaces de programación por CUDA.

2.1.1. Modelo de Programación Escalable

La llegada de los CPUs de múltiples núcleos y los GPUs de muchos núcleos significa que la mayoría de los procesadores de hoy en día son sistemas paralelos. Más aún, su paralelismo continúa creciendo según la ley de Moore. Para poder explotar dicho crecimiento al máximo, la arquitectura CUDA se ha diseñado de forma que aprovechar el aumento de núcleos sea transparente para la aplicación [20].

Esta transparencia se logró con tres simples abstracciones: una jerarquía de grupos de hilos, memoria compartida y barreras de sincronización. Estas abstracciones proveen paralelismo de granularidad fina para datos e hilos, anidado dentro de paralelismo de granularidad gruesa para datos y tareas. Ellas guían a los programadores a dividir el problema en sub-problemas grandes que pueden ser resueltos independientemente (bloques de hilos), y cada sub-problema en piezas más pequeñas que pueden ser realizadas cooperativamente en paralelo por los hilos de un mismo bloque [20]. Por ejemplo, si se quiere implementar la multiplicación de dos matrices de orden $n \times n$, podemos dividir el problema en n productos matriz vector independientes y cada producto matriz vector se puede resolver en paralelo por hilos cooperativos. Por lo tanto, ignorando por un momento las limitaciones de tamaño, se podría dividir la tarea en n bloques de hilos independientes encargados de calcular un producto matriz vector; y cada bloque en n hilos, donde cada hilo se encarga de calcular una componente del vector resultante.

Requerir independencia entre los bloques de hilos permite que ellos puedan ser planificados en cualquiera de los núcleos disponibles y en cualquier orden. Por lo tanto, un programa CUDA puede ejecutarse sobre cualquier cantidad de núcleos de ejecución y automáticamente proveer la escalabilidad deseada, en la Figura 2.2 se muestra como un programa CUDA se escala automáticamente al cambiar la cantidad de núcleos de ejecución. Debido al aumento del cómputo paralelo al aumentar la cantidad de núcleos, teóricamente se espera que el tiempo de ejecución total se reduzca proporcionalmente [20].

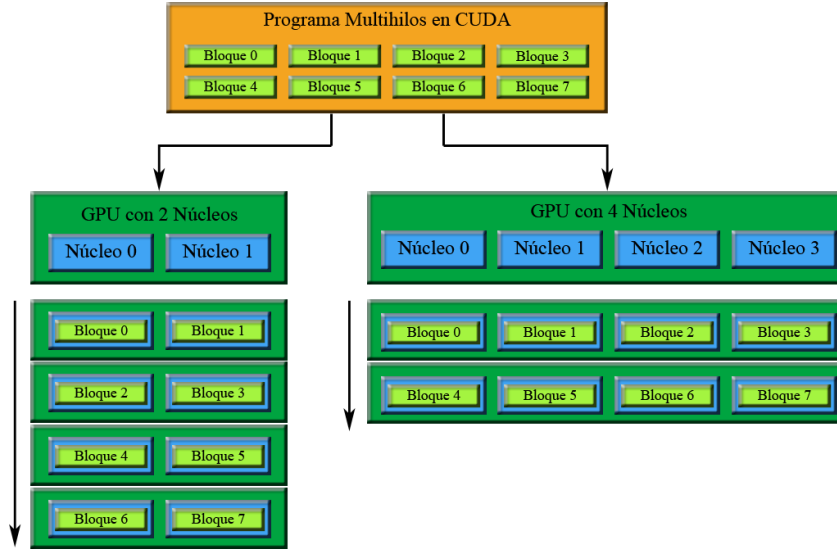


Figura 2.2: Escalabilidad automática de CUDA.

2.1.2. Jerarquía de Hilos

CUDA C extiende al lenguaje de programación *C* permitiendo que el programador defina funciones, llamadas *kernels*¹, que, cuando son llamadas, se ejecutan en paralelo por N hilos de CUDA, a diferencia de sólo una vez como en una función tradicional de *C*. La definición de un *kernel* se especifica con el modificador `--global--` y la invocación de los mismos utiliza una nueva sintaxis de *configuración de ejecución*, en la que se especifica la cantidad de bloques de hilos y la cantidad de hilos por bloque [20].

Cada hilo que ejecuta el *kernel* recibe un *identificador de hilo* y un *índice de hilo*, únicos dentro del bloque; el índice de hilo es accesible dentro del *kernel* a través de la variable integrada `threadIdx`, la cual es un vector de tres dimensiones. Por lo tanto, los hilos pueden ser organizados en bloques de hilos de una, dos o tres dimensiones. Lo cual provee una forma natural de invocar cálculos para los elementos de un dominio, tales como vectores, matrices o volúmenes [20].

¹*Kernel* en español significa núcleo, se decidió utilizar el término en inglés para evitar posibles confusiones con los núcleos de ejecución (*execution core*), manteniendo a su vez, la mayor consistencia posible con la documentación existente.

El índice de hilo y su identificador de hilo se relacionan de la siguiente forma:

- Para un bloque de una dimensión el índice y el identificador es el mismo.
- Para un bloque bidimensional de tamaño (D_x, D_y) , el identificador del hilo con índice (x, y) es $x + yD_x$.
- Para un bloque tridimensional de tamaño (D_x, D_y, D_z) , el identificador del hilo con índice (x, y, z) es $x + yD_x + zD_xD_y$.

La cantidad de hilos por bloque es limitada, ya que todos los hilos de un mismo bloque deben residir en el mismo núcleo de ejecución y compartir la memoria limitada de dicho núcleo. Sin embargo, los *kernels* pueden ser ejecutados por múltiples bloques de hilos homogéneos, por lo que el número total de hilos que ejecutan el *kernel* es igual al número de hilos por bloque multiplicado por el número de bloques de hilos. Al invocar un *kernel*, se crea una *mallada de bloques* que contiene todos los bloques de hilos especificados en la configuración de ejecución. Esta mallada puede ser de una o dos dimensiones dependiendo del dominio del problema y su tamaño es usualmente determinado por la cantidad de datos a procesar o la cantidad de procesadores en el sistema [20]. La Figura 2.3 muestra una mallada bidimensional de 3×2 de bloques bidimensionales de 2×3 .

Los hilos dentro de un bloque pueden cooperar utilizando *memoria compartida* y sincronizando su ejecución para coordinar los accesos a memoria. Más precisamente, se pueden especificar puntos de sincronización dentro del *kernel* llamando a la función `__syncthreads()`; los cuales actúan como una barrera que todos los hilos del bloque deben alcanzar antes de que alguno pueda proseguir. Por motivos de eficiencia, la memoria compartida debe ser una memoria de latencia baja dentro del chip (como una caché de 1^{er} nivel) y `__syncthreads()` debe ser ligera [20].

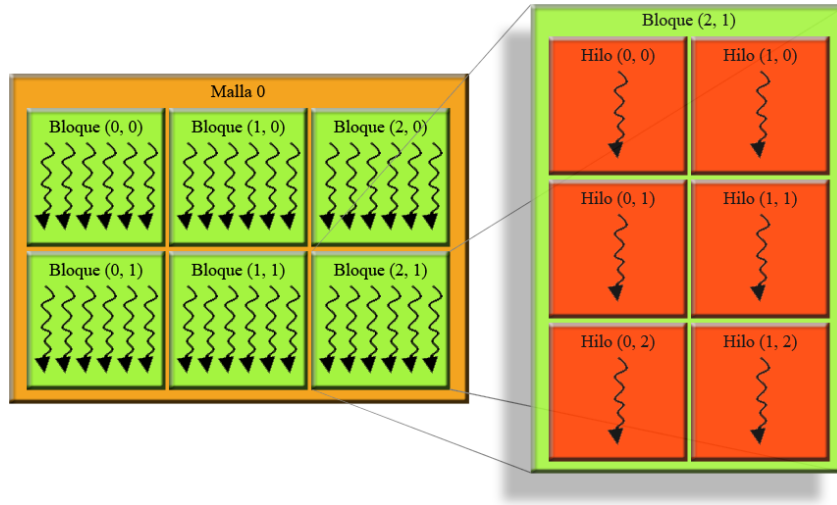


Figura 2.3: Malla de bloques de hilos.

2.1.3. Jerarquía de Memoria

Como se puede ver en la Figura 2.4, los hilos de CUDA pueden acceder a datos en diferentes espacios de memoria durante su ejecución. Cada hilo tiene una memoria local privada, todos los hilos de un bloque tienen acceso a una memoria compartida, la cual perdura durante la ejecución del bloque completo; y todos los hilos del *kernel* tienen acceso a la memoria global. Además, todos los hilos tienen acceso a dos espacios de memoria de sólo lectura: los espacios de memoria constante y de texturas. Los datos en memoria global, constante y de textura persisten a lo largo del tiempo de vida de la aplicación [20].

2.1.4. Programación Heterogénea

El modelo de programación CUDA asume que los hilos de CUDA se ejecutan en un *dispositivo* físicamente separado que opera como un coprocesador para el *anfitrión* que ejecuta el programa *C*. Este es el caso, por ejemplo, cuando los *kernels* se ejecutan en un GPU y el resto del programa se ejecuta en el CPU. Además, el modelo asume que tanto el anfitrión como el dispositivo mantienen espacios de memoria separados

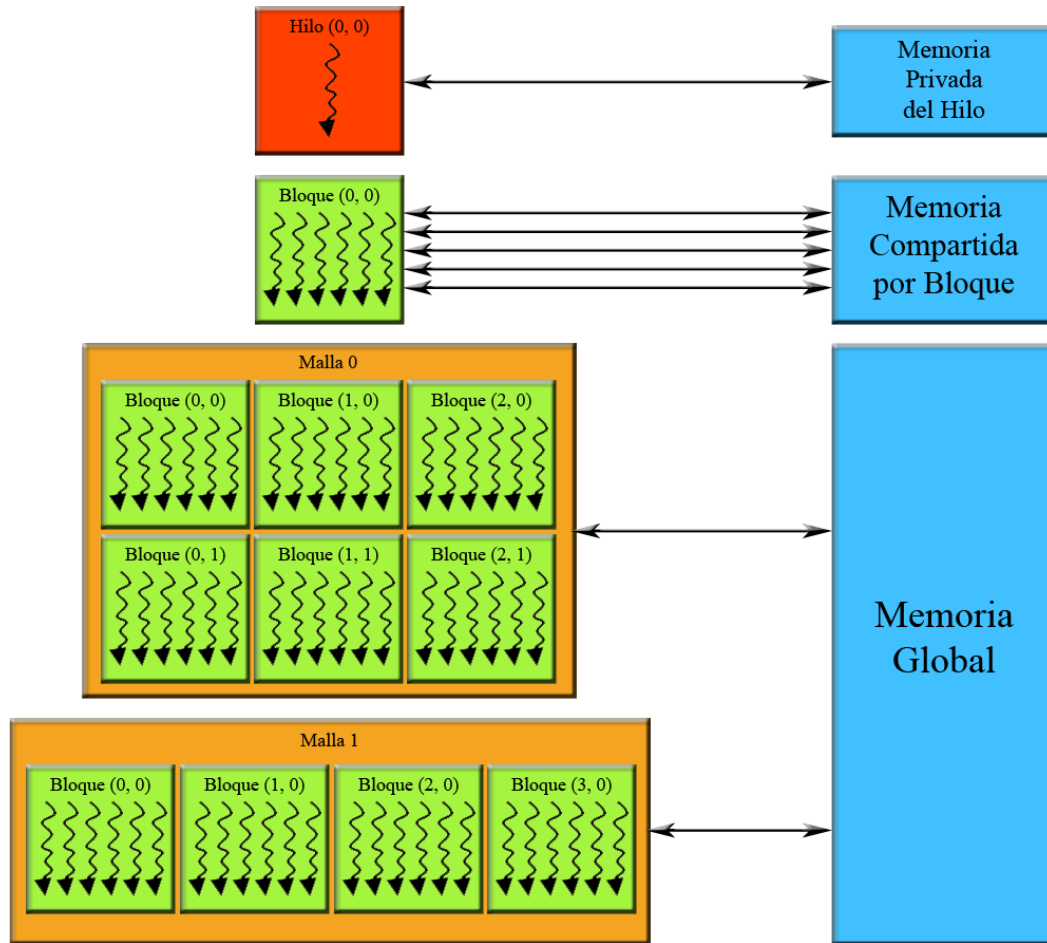


Figura 2.4: Diferentes espacios de memoria accedidos por los hilos de CUDA.

en DRAM. Por lo tanto, el programa debe manejar los espacios de memoria global, constante y de textura visibles a los *kernels* a través de llamadas al ambiente de ejecución de CUDA. Esto incluye reservar y liberar memoria así como transferir datos entre los espacios de memoria del anfitrión y el dispositivo [20]. En la Figura 2.5 se muestra la ejecución común de un programa, en la que el anfitrión ejecuta código serial y el dispositivo *kernels* paralelos.

2.1.5. Consideraciones de Rendimiento

Las estrategias de optimización de rendimiento para programas CUDA circulan alrededor de las siguientes tres estrategias básicas:

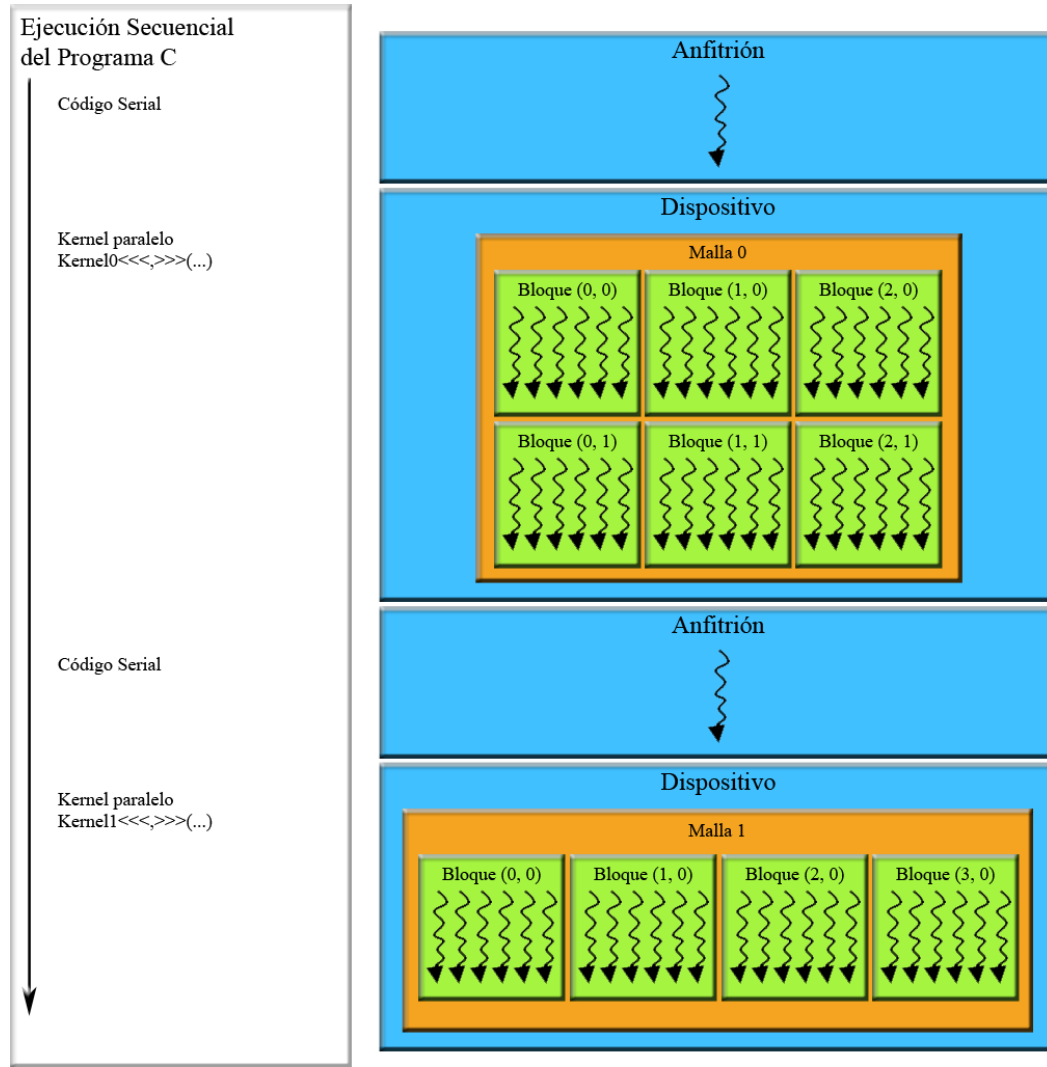


Figura 2.5: Programación heterogénea.

- Maximizar la ejecución paralela para maximizar la utilización.
- Optimizar el uso de memoria para maximizar la tasa de transferencias efectivas.
- Optimizar el uso de instrucciones para maximizar la tasa de instrucciones ejecutadas.

Que estrategia produce mayores ganancias de rendimiento para una porción particular de una aplicación depende de las limitantes de rendimiento de dicha porción. Optimizar el rendimiento de las instrucciones de un *kernel* que está principalmente

limitado por accesos a memoria no producirá grandes ganancias de rendimiento, por ejemplo. Por lo tanto, los esfuerzos de optimización deben ser constantemente guiados a través de mediciones y seguimiento de las limitantes de rendimiento del *kernel* [20].

Antes de analizar las estrategias de optimización, se debe estudiar el funcionamiento de la arquitectura en mayor detalle. La arquitectura CUDA está construida sobre un arreglo escalable de *multiprocesadores de flujo*² multihilos. Cuando un programa CUDA en el anfitrión invoca un *kernel*, los bloques de la malla son enumerados y distribuidos entre los multiprocesadores con capacidad de ejecución disponible. Los hilos de un bloque se ejecutan concurrentemente en un multiprocesador, y múltiples bloques de hilos pueden ser ejecutados concurrentemente por el multiprocesador. A medida que los bloques de hilos terminan su ejecución, bloques nuevos son iniciados en los multiprocesadores liberados [20].

2.1.5.1. Arquitectura SIMT

Un multiprocesador está diseñado para ejecutar cientos de hilos concurrentes, para manejar tal cantidad de hilos éste utiliza una arquitectura llamada *SIMT*³ (una instrucción, múltiples hilos). Para maximizar la utilización, el hardware aprovecha el paralelismo a nivel de hilos en vez del paralelismo a nivel de instrucciones dentro de un mismo hilo, es decir, las instrucciones son ejecutadas en orden, sin predicción de salto ni ejecución especulativa [20].

El multiprocesador crea, maneja, planifica y ejecuta hilos en grupos de 32 hilos paralelos llamados *warps*⁴. Los hilos que componen un *warp* comienzan su ejecución al mismo tiempo en la misma dirección de programa, pero tienen su propio conta-

²Streaming Multiprocessors.

³El término SIMT es un acrónimo de *Single-Instruction Multiple-Thread*.

⁴El término *warp* (urdimbre) recibe el nombre de la tejeduría, considerada la primera tecnología paralela de hilos. Se decidió utilizar el término en inglés por su amplia aceptación en la documentación, mientras que el término urdimbre no es muy utilizado en el área de cómputo paralelo.

dor de programa y estado de registros, por lo que son libres de divergir y ejecutar independientemente [20].

Cuando un multiprocesador recibe uno o más bloques de hilos, este los particiona en *warps* que son planificados por un planificador de *warps*. La forma en la que los bloques son particionados es siempre la misma; cada *warp* contiene hilos con identificadores consecutivos y crecientes, donde el primer *warp* recibe el hilo con identificador 0 [20].

Un *warp* ejecuta una instrucción común a la vez, por lo que la mayor eficiencia es alcanzada cuando los 32 hilos del *warp* tienen el mismo camino de ejecución. Si los hilos de un *warp* divergen por medio de ramificaciones condicionales, el *warp* ejecuta cada rama de forma serial, deshabilitando los hilos que no pertenecen a dicha rama. Cuando todos los caminos terminan, los hilos convergen de nuevo a la misma ruta de ejecución. La divergencia de ramas sólo ocurre dentro de un *warp*, ya que, los diferentes *warps* de un mismo bloque siempre se ejecutan de forma independiente sin importar si están ejecutando la misma ruta de ejecución o no [20].

La arquitectura SIMT es similar a la organización vectorial SIMD⁵ (una instrucción, múltiples datos), en la que una instrucción controla múltiples elementos de procesamiento. Una diferencia clave es que las organizaciones vectoriales SIMD exponen el tamaño de la operación SIMD, mientras que las instrucciones SIMT especifican la ejecución y comportamiento de ramificación para un hilo. El programador puede ignorar el comportamiento SIMT y desarrollar aplicaciones correctas, sin embargo, grandes mejoras de rendimiento pueden ser obtenidas al minimizar la divergencia de ramas [20].

⁵El término SIMD es un acrónimo de *Single-Instruction Multiple-Data*.

2.1.5.2. Maximizando la Utilización

Para maximizar la utilización de las unidades del sistema, la aplicación debe ser estructurada de forma que exponga el mayor nivel de paralelismo posible y eficientemente proyecte este paralelismo a los diferentes componentes del sistema, de manera que se mantengan ocupados la mayor parte del tiempo [20].

2.1.5.2.1. Nivel de Aplicación

A un alto nivel, la aplicación debe maximizar el paralelismo entre el anfitrión, el dispositivo y el bus que los conecta; utilizando llamadas y transferencias asíncronas. Además, debe asignar a cada procesador el tipo de trabajo que hace mejor: carga serial al anfitrión y carga paralela a los dispositivos. Para cargas de trabajo paralelas, existen dos casos donde el paralelismo puede romperse debido a que los hilos necesitan sincronizarse para compartir datos entre ellos:

1. Los hilos pertenecen al mismo bloque, en este caso utilizan la función intrínseca `__syncthreads` y comparten los datos utilizando la memoria compartida desde el *kernel*.
2. Los hilos pertenecen a bloques diferentes, en este caso ellos deben utilizar la memoria global para compartir datos y hacer dos llamadas a *kernels* diferentes, una para escribir y una para leer.

El segundo caso es mucho más lento que el primero debido al costo de la invocación del *kernel* y el uso de memoria global, por lo tanto, su ocurrencia debe ser minimizada implementado el algoritmo con el modelo de programación CUDA en mente [20].

2.1.5.2.2. Nivel de Dispositivo

En un nivel más bajo, la aplicación debe maximizar el paralelismo entre los multiprocesadores del dispositivo. Para dispositivos con versión de CUDA 1.x, sólo se puede ejecutar un *kernel* a la vez, por lo que siempre se debe crear la cantidad de bloques necesarios (o la mayor cantidad posible) durante la invocación. Los dispositivos con versión 2.0, pueden ejecutar múltiples *kernels* al mismo tiempo, de forma que se puede obtener mayor utilización con diferentes flujos para permitir que existan suficientes *kernels* en el dispositivo de forma concurrente [20].

2.1.5.2.3. Nivel de Multiprocesador

A un nivel aún más bajo, la aplicación debe maximizar el paralelismo entre las diferentes unidades funcionales de un multiprocesador. Como se explicó en la Sección 2.1.5.1, el multiprocesador del GPU se basa en el paralelismo a nivel de hilos para maximizar la utilización de sus unidades funcionales. La utilización está, por lo tanto, directamente ligada con la cantidad de *warps* residentes en el multiprocesador. En cada ciclo del reloj, el planificador de *warps* escoge un *warp* que esté listo para ejecutar, si existe, y emite la próxima instrucción a todos los hilos activos del *warp*. La cantidad de ciclos de reloj que un *warp* necesita para estar listo para ejecutar la próxima instrucción se denomina *latencia*; la utilización máxima se consigue cuando en cada ciclo de reloj el planificador siempre tiene alguna instrucción que emitir a algún *warp*, es decir, cuando la latencia de todos los *warp* se encuentra completamente “oculta” por otros *warps* [20].

La razón más común para que un *warp* no esté listo para ejecutar la próxima instrucción es que los operandos de entrada no están listos aún. Si todos los operandos de entrada son registros la latencia es causada por dependencia de registros, por ejemplo, algunos de los operandos son calculados por instrucciones anteriores que no han terminado de ejecutarse; en este caso, la latencia es igual al tiempo que toma

ejecutar la instrucción anterior. El tiempo de ejecución de una instrucción depende de la instrucción, pero es típicamente alrededor de 22 ciclos de reloj. Por otro lado, si alguno de los operandos reside en memoria fuera del chip, la latencia es mucho mayor: de 400 a 800 ciclos aproximadamente. El número de *warps* necesarios para mantener al planificador ocupado durante ese tiempo depende del código del *kernel*, aunque en general, mientras menor sea la tasa de instrucciones sin operandos en memoria con respecto a instrucciones con operandos en memoria, mayor será la cantidad de *warps* necesarios [20].

Otra razón por la que un *warp* puede no estar listo para ejecutar su próxima instrucción, es porque se encuentra en una barrera de sincronización. Un punto de sincronización puede forzar al multiprocesador a esperar, mientras más y más *warps* esperan que los otros *warps* del mismo bloque terminen de ejecutar las instrucciones previas al punto de sincronización. Tener múltiples bloques en el multiprocesador ayuda a aliviar este problema, ya que los *warps* de bloques diferentes no necesitan esperar [20].

2.1.5.3. Maximizar la Tasa de Transferencias Efectivas

El primer paso para mejorar la tasa de transferencias es minimizar el uso de operaciones con bajo ancho de banda, es decir, minimizar las transferencias entre las memorias del anfitrión y el dispositivo. Luego, se debe minimizar la transferencia de datos de memoria global al chip, lo cual se logra maximizando el uso de la memoria que se encuentra dentro del chip [20].

La memoria compartida es como una caché manejada por el usuario: la aplicación la reserva y la maneja directamente. Un patrón de programación típico es traer datos de memoria global a la memoria compartida, realizar algún procesamiento y luego devolver el resultado a memoria global, es decir, hacer que cada hilo del bloque:

1. Cargue datos de memoria global en memoria compartida.

2. Se sincronice con los otros hilos del bloque para que todos los hilos puedan leer de memoria compartida de forma segura.
3. Procese los datos en memoria compartida.
4. Se sincronice nuevamente, si es necesario, para asegurar que la memoria compartida tenga el resultado completo.
5. Escriba los resultados en memoria global.

Una instrucción que acceda a memoria direccionable (global, local, compartida, constante o de textura) puede necesitar ser emitida nuevamente múltiples veces dependiendo de la distribución de memoria a lo largo de los hilos de un *warp*. Cómo la distribución afecta la tasa de transferencias depende del tipo de memoria accedida [20].

2.1.5.3.1. Memoria Global

La memoria global reside en la memoria del dispositivo y es accedida a través de transacciones de memoria de 32, 64 o 128 bytes, las cuales deben estar naturalmente alineadas. Cuando un *warp* ejecuta una instrucción que accede a memoria global, éste agrupa los accesos de los diferentes hilos en una o más transacciones dependiendo del tamaño de la palabra utilizada por los hilos y la dispersión de las direcciones de memoria. En general, mientras más transacciones sean necesarias, mayor será la cantidad de palabras no utilizadas que serán transferidas; reduciendo proporcionalmente la tasa de transferencias efectivas. Cuantas transacciones son necesarias es finalmente determinado por la versión de CUDA del dispositivo y el patrón de dispersión de los accesos, por ello se invita al lector a ver los apéndices expuestos en [20] para mayor información.

2.1.5.3.2. Memoria Compartida

La memoria compartida se encuentra dentro del chip y es mucho más rápida que la memoria del dispositivo. De hecho, para todos los hilos de un *warp*, el costo de acceder a memoria compartida es casi despreciable siempre y cuando no existan *conflictos de bancos*. Para alcanzar un gran ancho de banda, la memoria compartida se divide en módulos de igual tamaño, llamados bancos, los cuales pueden ser accedidos simultáneamente. Cualquier petición de lectura o escritura a n direcciones que se encuentren en n bancos diferentes pueden ser servidas simultáneamente, generando un ancho de banda general n veces mayor [20].

Sin embargo, si dos direcciones de una petición de memoria caen en el mismo banco, se genera un *conflicto de banco* y los accesos deben ser serializados. El hardware separa la petición en una cantidad de peticiones libres de conflictos, reduciendo la tasa efectiva por la cantidad de peticiones separadas. Por lo tanto, para maximizar el ancho de banda efectivo, es necesario estudiar la forma en la que las direcciones de memoria son traducidas a bancos. De nuevo se invita al lector a ver los apéndices de [20] para los detalles de las traducciones para las diferentes versiones de CUDA.

2.1.5.3.3. Memoria Constante

El espacio de memoria constante se encuentra en memoria de dispositivo, pero sus datos son guardados en caché. Una petición a memoria constante es dividida en tantos accesos como direcciones diferentes existan en dicha petición, disminuyendo la tasa efectiva por un factor igual al número de accesos separados. Luego, el ancho de banda de cada acceso es igual al ancho de banda de la memoria compartida, si el dato se encontraba en la memoria caché; o a el de la memoria global en caso contrario [20].

2.1.5.3.4. Memoria de Textura

Al igual que la memoria constante, la memoria de textura reside en memoria de dispositivo y tiene una caché de textura; por lo que cada acceso cuesta un acceso a memoria de dispositivo sólo cuando éste no se encuentra en la caché. La caché de textura está optimizada para localidad espacial en dos dimensiones, por lo que los hilos de un *warp* que accedan a datos que están cercanos en dos dimensiones obtendrán el mayor rendimiento. Leer memoria de dispositivo a través de muestreos de texturas puede ser una alternativa ventajosa para algunas aplicaciones debido a:

- Los accesos a memoria no siguen los patrones necesarios para ser agrupados pero existe proximidad espacial⁶.
- Los cálculos de direccionamiento son realizados fuera del *kernel*.
- Datos empaquetados pueden ser transferidos a múltiples variables en una sola operación.
- Datos enteros de 8 o 16 bits, son opcionalmente transformados en valores reales de 32 bits en los rangos $[0, 1]$ o $[-1, 1]$.

Sin embargo, durante la ejecución de un *kernel* no se mantiene la consistencia de la caché de textura, las lecturas por muestreo de textura a una dirección que ha sido modificada durante la ejecución del *kernel* retornan un valor indefinido. Por lo tanto, los hilos pueden leer de forma segura, si y sólo si, no existen escrituras a la misma dirección de memoria por ningún hilo del *kernel* [20].

2.1.5.4. Maximizar la Tasa de Instrucciones Ejecutadas

Para maximizar la tasa de instrucciones ejecutadas, la aplicación debe:

⁶Los dispositivos con versión CUDA 2.0 incluyen caché tradicional para memoria global, por lo tanto, la memoria de textura no ofrece ninguna ventaja para explotar localidad espacial en dichos dispositivos.

- Minimizar la cantidad de instrucciones lentas, lo que significa sacrificar precisión por velocidad cuando sea posible.
- Minimizar la divergencia de ramas dentro de los *warps*, como se explicó anteriormente.
- Reducir la cantidad de instrucciones, optimizando la cantidad de puntos de sincronización, por ejemplo.

Para mayor detalle sobre la cantidad de ciclos que cada operación requiere se refiere al lector a [20].

2.2. Despliegue Volumétrico

El término, despliegue volumétrico (*volume rendering*), se refiere a un conjunto de técnicas que permiten generar imágenes bidimensionales a partir de datos tridimensionales, llamados *volúmenes*. De forma teórica, un volumen es un campo escalar sobre el espacio euclídeo tridimensional, es decir, una función $f : \mathbb{E}^3 \rightarrow \mathbb{R}$ [4]. Sin embargo, el incentivo original del despliegue volumétrico fue crear métodos eficientes para visualizar imágenes médicas, campo que sigue siendo el más importante. Los datos generados por Tomografías Computarizadas (CT) o Imágenes por Resonancia Magnética (MRI) son los de mayor interés, este tipo de técnicas generan mallas tridimensionales basadas en muestras discretas regularmente espaciadas a lo largo de cada eje⁷ [26].

Los métodos de despliegue volumétrico se pueden dividir en dos categorías. La primera categoría, denominada *despliegue volumétrico indirecto*, consiste en detectar superficies en el volumen, ajustar primitivas geométricas a dichas superficies y luego desplegarlas utilizando técnicas de despliegue tradicionales. Una de las técnicas más utilizadas en esta categoría se conoce como *cubeos marchantes*, presentado por primera

⁷El espaciado de las muestras no tiene que ser el mismo en todos los ejes. De hecho, es común que tengan el mismo espaciado en los ejes x e y , pero un espaciado mayor en el eje z .

vez en [16]. La segunda categoría, llamada *despliegue volumétrico directo*⁸, consiste en generar la imagen directamente de los datos sin imponer ninguna restricción geométrica sobre ellos. Estas técnicas tienen el potencial de producir imágenes de mayor calidad a cambio de un costo computacional más elevado [14].

2.2.1. Emisión de Rayos

La técnica de despliegue volumétrico directo más importante y más documentada se conoce como emisión de rayos (*ray casting*). Existen muchas formas de implementar el algoritmo de emisión de rayos, en este trabajo se estudia el algoritmo propuesto por Marc Levoy en varias publicaciones [14][13], basado en el modelo Blinn-Kajiya [2][11]; ésta es una de las combinaciones más comunes en programas comerciales.

2.2.1.1. Modelo Blinn-Kajiya

El modelo Blinn-Kajiya propone una forma analítica de calcular la acumulación de intensidades de luz reflejadas desde las partículas dentro de un volumen a lo largo de un rayo. Para ésto se asume que se tiene un rayo $R = (x(t), y(t), z(t))$, un volumen definido por una función de densidad $\rho : \mathbb{E}^3 \rightarrow \mathbb{R}$ y una función de fase P . La función de fase en su forma general, recibe dos vectores u y u' y $P(u, u')$ indica la cantidad de luz que es reflejada desde la dirección u en la dirección u' . Sin embargo, cuando el medio es isotrópico⁹, la función de fase depende únicamente del *ángulo de fase* $\Theta = \angle uu'$ [11]. En este trabajo sólo se consideran medios isotrópicos y se utiliza $\cos(\Theta)$ como parámetro de la función de fase.

El modelo divide el cómputo en dos etapas, la primera etapa determina la radiación desde la i -ésima fuente de luz a través del volumen y la almacena en el arreglo

⁸Originalmente el despliegue volumétrico directo se llamaba simplemente despliegue volumétrico, y el indirecto no se consideraba como una técnica de despliegue volumétrico.

⁹Uniforme en todas las direcciones.

$I_i(x, y, z)$, el cual contiene la contribución de cada fuente de luz al brillo de cada punto en el espacio como se muestra en la Figura 2.6. Esto se realiza resolviendo la integral de línea

$$I_i(x, y, z) = e^{-\tau \int_{T_{x,y,z}} \rho(x(t), y(t), z(t)) dt}, \quad (2.1)$$

donde τ es una constante que transforma densidad en atenuación, para cada trayectoria $T_{x,y,z} = (x(t), y(t), z(t))$ desde la fuente de luz a través del volumen [10].

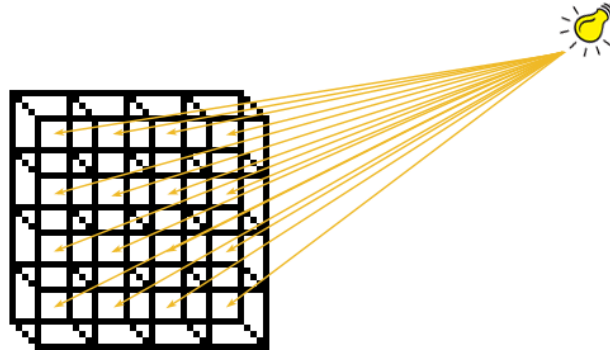


Figura 2.6: Trayectorias desde la luz.

La segunda etapa calcula la intensidad acumulada de un rayo R como la suma de las contribuciones de los elementos volumétricos en el intervalo $[t_0, t_1]$. Donde t_0 representa el punto en el rayo donde éste entra en el volumen y t_1 el punto por el que sale, la Figura 2.7 ilustra el proceso para el volumen encontrado antes del punto t . El cálculo de dicha acumulación se obtiene resolviendo la integral de brillo mostrada en (2.2).

$$B = \int_{t_0}^{t_1} e^{-\tau \int_{t_0}^t \rho(x(s), y(s), z(s)) ds} \sum_i \{I_i(x(t), y(t), z(t)) P(\cos(\Theta_i))\} \rho(x(t), y(t), z(t)) dt \quad (2.2)$$

La exponencial al principio de (2.2) calcula la atenuación por absorción y dispersión del material visible al observador y la sumatoria determina el brillo de un punto como la acumulación de la contribución de brillo de cada fuente de luz [10]. Para ciertas aplicaciones es preferible asumir que cada punto del volumen es iluminado uniformemente por cada fuente de luz, es decir, sin absorción, dispersión u oclusión de

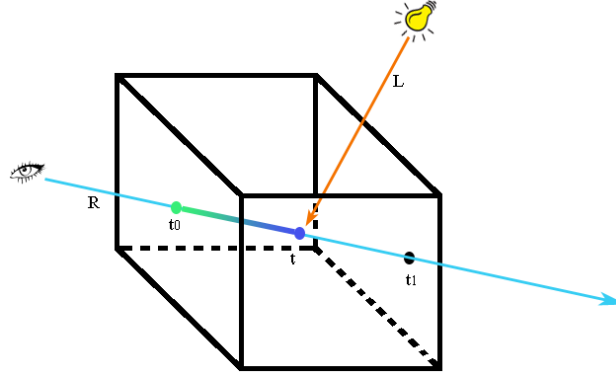


Figura 2.7: Interpretación de la integral de brillo.

ningún tipo. Este es el caso cuando se quiere ver el interior de un cuerpo al visualizar imágenes médicas. Esta simplificación elimina la necesidad de calcular la integral de línea (2.1) y permite simplificar (2.2) como

$$B = \int_{t_0}^{t_1} e^{-\tau \int_{t_0}^t \rho(x(s), y(s), z(s)) ds} \sum_i P(\cos(\Theta_i)) \rho(x(t), y(t), z(t)) dt$$

2.2.1.2. Algoritmo Básico

El algoritmo de emisión de rayo propuesto por Levoy [14][13], permite aproximar la integral de brillo para cada píxel de la imagen final. Más aún, permite incorporar la asignación de color y opacidad a diferentes valores de densidad y mezclarlos acordemente a lo largo del rayo. El algoritmo propone emitir un rayo desde el ojo del observador hacia el centro de cada píxel de la imagen, estos rayos atraviesan el volumen hasta llegar a un fondo completamente opaco que se coloca por detrás del volumen. El segmento de rayo que se encuentra dentro del volumen es muestreado en intervalos equivalentes y se determina un color y opacidad para la muestra, los cuales son acumulados a lo largo del rayo para obtener el color del píxel [14]. La Figura 2.8 ilustra el proceso mientras que el Algoritmo 2.1 muestra los pasos en mayor detalle.

Uno de los mejores métodos para estimar el vector gradiente en el Paso 8 del Algoritmo 2.1, es el método de sombreado de niveles de gris [26][14][13][8][24][21]. El método ofrece varias ventajas como estimaciones de alta precisión, habilidad para

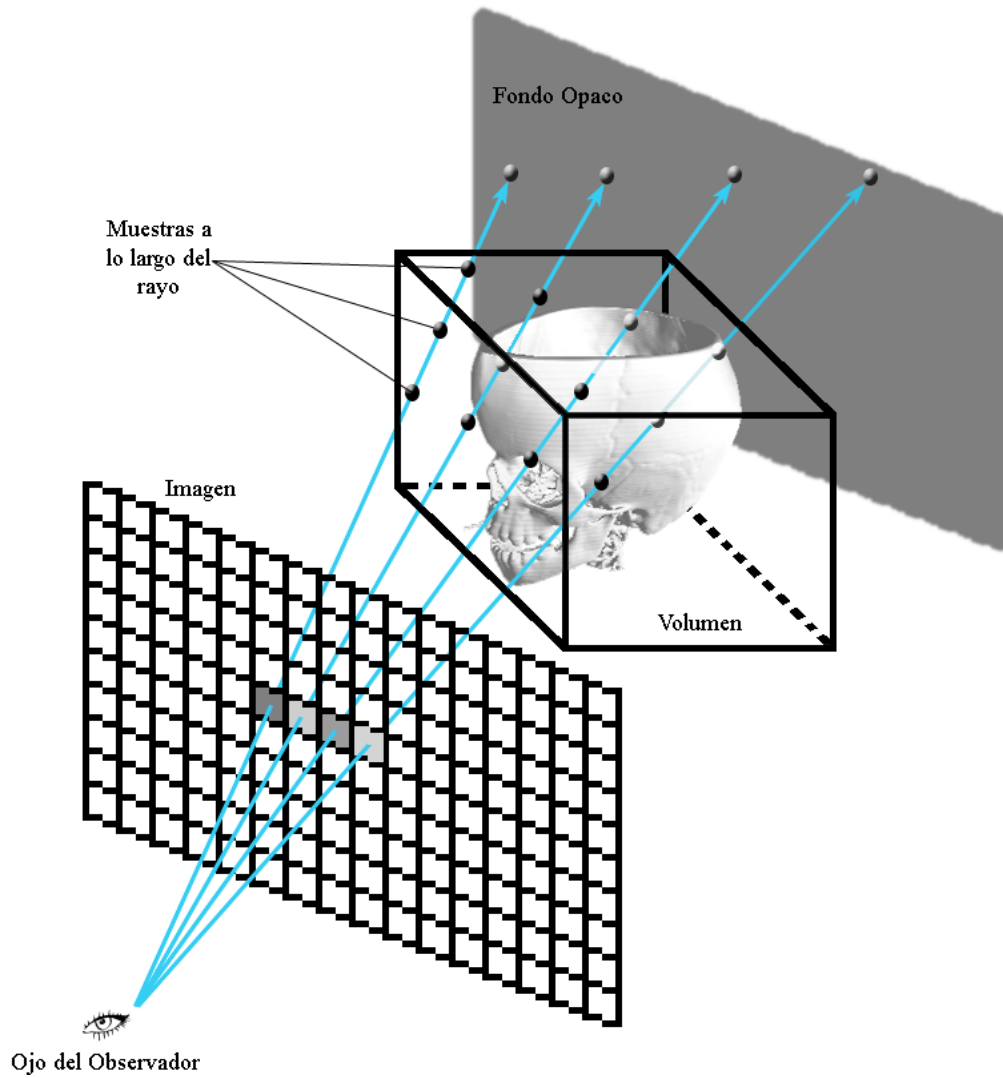


Figura 2.8: Emisión de rayos.

calcular normales para superficies semitransparentes e independencia del ángulo de visión. La normal de un punto (x, y, z) en un volumen V , es la derivada parcial de V con respecto a los ejes x , y , y z :

$$\Delta V(x, y, z) = \left(\frac{\delta V}{\delta x}, \frac{\delta V}{\delta y}, \frac{\delta V}{\delta z} \right) \quad (2.3)$$

Un método común con resultados satisfactorios para aproximar las derivadas de (2.3), es el *método de diferencias finitas*, el cual genera tres ecuaciones para estimar la derivada de una función discreta utilizando los valores de sus vecinos. Las

-
1. Para cada píxel (i, j) de la imagen, **hacer**:
 2. | **Crear** un rayo R cuyo origen es la posición del ojo y con dirección hacia el píxel (i, j) .
 3. | **Inicializar** el color y la opacidad del píxel como $c_R = (0, 0, 0)$ y $\alpha_R = 0$.
 4. | **Calcular** t_0 y t_1 , los puntos de entrada y salida del rayo R .
 5. | Para t desde t_0 hasta t_1 incrementado en *paso*, **hacer**:
 6. | | **Asignar** $p_t = R.\text{origen} + R.\text{dirección} * t$.
 7. | | **Muestrear** el volumen en el punto p_t y asignar el valor a u_t .
 8. | | **Aproximar** el vector gradiente $\Delta V(p_t)$ y el vector normal $N = \frac{\Delta V(p_t)}{\|\Delta V(p_t)\|}$.
 9. | | **Calcular** el color y la opacidad del punto, $(c_t, \alpha_t) = f(u_t, p_t, N)$.
 10. | | **Actualizar** el color del píxel, $c_R = c_R + c_t * (1 - \alpha_R)$.
 11. | | **Actualizar** la opacidad del píxel, $\alpha_R = \alpha_R + \alpha_t * (1 - \alpha_R)$.
 12. | **Mezclar** el píxel con el fondo opaco, $c_R = c_R + \text{fondo.color} * (1 - \alpha_R)$.
-

Algoritmo 2.1: Emisión de Rayos

ecuaciones para estimar la derivada con respecto al eje x , mostradas en (2.4), (2.5) y (2.6); se conocen como *diferencia hacia adelante*, *diferencia hacia atrás* y *diferencia central*, respectivamente [26][8][24].

$$\frac{\delta V(x, y, z)}{\delta x} \approx \frac{V(x + 1, y, z) - V(x, y, z)}{D_x} \quad (2.4)$$

$$\frac{\delta V(x, y, z)}{\delta x} \approx \frac{V(x, y, z) - V(x - 1, y, z)}{D_x} \quad (2.5)$$

$$\frac{\delta V(x, y, z)}{\delta x} \approx \frac{V(x + 1, y, z) - V(x - 1, y, z)}{2D_x} \quad (2.6)$$

Las ecuaciones (2.4) y (2.5) son utilizadas para estimar la derivada en los bordes del volumen, mientras que (2.6) es utilizada para los puntos internos debido a que ofrece una mejor aproximación. Este método puede ser extendido para obtener una aproximación más precisa, utilizando 64 vecinos en lugar de tan sólo ocho [26][13].

La función f utilizada en el Paso 9 del Algoritmo 2.1 calcula un color y opacidad para el punto, a partir de la densidad, posición y vector normal de la muestra. Una de las formas más comunes es utilizar una función, llamada *función de transferencia*, que a partir de la densidad de la muestra genera un color y una opacidad; y luego, dicho color es sombreado utilizando el modelo de iluminación de Phong o Blinn-Phong [13].

2.2.1.3. Despliegue de Isosuperficies

El Algoritmo 2.1 puede ser modificado un poco para que, en vez de mezclar muestras semitransparentes, despliegue isosuperficies opacas dentro del volumen [26][13]. En este caso, para cada rayo es necesario determinar el punto donde éste toca la isosuperficie. Sea α el isovalor asociado a la isosuperficie deseada y p_s el punto donde el rayo toca dicha isosuperficie, es decir, $V(p_s) = \alpha$. El primer paso es determinar si la superficie se encuentra entre dos puntos de muestreo consecutivos p_{t-1} y p_t , como se ilustra en la Figura 2.9. Sin perder la generalidad, se asume que $V(p_{t-1}) \leq V(p_t)$ y utilizando el teorema de los valores intermedios se concluye que:

$$p_s \in (p_{t-1}, p_t], \text{ sii, } \alpha \in (V(p_{t-1}), V(p_t)]$$

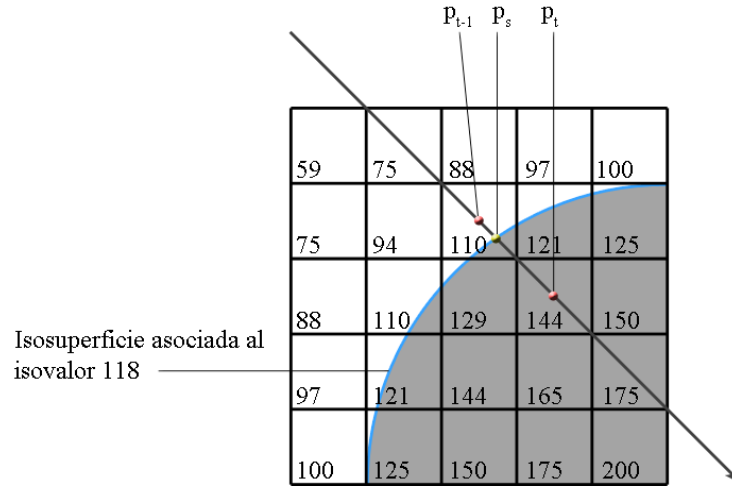


Figura 2.9: Detección de isosuperficies con emisión de rayos.

Luego de encontrar un rango sobre el rayo que contiene a la isosuperficie deseada, se necesita estimar un punto $\tilde{p}_s$ que aproxime a p_s . Una opción es utilizar uno de los extremos del rango, dependiendo de cual tiene una densidad más cercana al isovalor, como se muestra en (2.7). Otra opción, con mejores resultados, es interpolar linealmente los extremos del rango tomando en cuenta las densidades en ambos puntos,

éste método se muestra en (2.8) [26]. Una opción más costosa pero más precisa, es utilizar un método de bisección para reducir iterativamente el rango que contiene la isosuperficie. Esto se logra muestreando en el punto medio $p_m = p_{t-1} + \frac{1}{2}(p_t - p_{t-1})$ y utilizando la ecuación (2.7) para determinar si $p_s \in (p_{t-1}, p_m]$ o $p_s \in (p_m, p_t]$, este proceso puede repetirse mientras que el tamaño del rango sea superior a una tolerancia prescrita o una cantidad de iteraciones sea alcanzada [15].

$$\tilde{p}_s = \begin{cases} p_{t-1} & \text{si } |V(p_{t-1}) - \alpha| < |V(p_t) - \alpha|, \\ p_t & \text{si } |V(p_{t-1}) - \alpha| \geq |V(p_t) - \alpha| \end{cases} \quad (2.7)$$

$$\begin{aligned} \bar{s} &= \frac{\alpha - V(p_{t-1})}{V(p_t) - V(p_{t-1})} \\ \tilde{p}_s &= p_{t-1} + (p_t - p_{t-1}) * \bar{s} \end{aligned} \quad (2.8)$$

2.3. Simulaciones médicas y trabajo de referencia

La simulación de cirugías médicas es un área en crecimiento y su aceptación por la comunidad de cirujanos es cada día más amplia. Entre ellos se destacan los sistemas de realidad virtual, debido en gran parte a los avances en las técnicas de realidad virtual y a la disponibilidad de modelos anatómicos virtuales [21].

Debido a sus prometedores resultados, este trabajo profundiza en las soluciones propuestas en [21], el cual fue presentado por el Instituto de Matemáticas y Ciencias de la Computación en Medicina del Hospital Universitario Hamburg-Eppendorf de Alemania. Este trabajo tenía como objetivo ofrecer una alternativa al estudio tradicional de las zonas destacadas en el hueso temporal, ya que actualmente dicho estudio se realiza por medio de cursos limitados de taladrado de hueso. El trabajo propone un sistema de cirugía virtual de la parte petrosa (peñasco) del hueso temporal que permite la simulación realista de abordajes quirúrgicos [21].

2.3.1. Representación Multivolumétrica

Se creó un modelo del hueso temporal basado en tomografías computarizadas, en el cual se determinan una serie de estructuras anatómicas. Algunas utilizando un esquema de segmentación semiautomático basado en umbrales, mientras que otras utilizando un editor de volúmenes con el cual el usuario puede separar objetos manualmente. Dichas estructuras son representados por una malla rectilínea tridimensional de elementos volumétricos, donde cada vóxel tiene asociado un valor de densidad y un conjunto de atributos, como la región anatómica a la que pertenece y un color para el despliegue. Este nivel es equivalente al modelo de vóxel generalizado presentado en [9].

Para lograr la funcionalidad de especificar y representar de forma interactiva regiones de corte, se extendió el esquema a una representación de múltiples volúmenes, donde cada región de corte no es representada únicamente por atributos en el volumen original sino que tiene su propia representación espacial en un volumen separado. Esto permite que el volumen original se encuentre disponible en todo momento y que todas las operaciones puedan ser revertidas fácilmente [21].

2.3.2. Corte de Volumen

Utilizar atributos a nivel de vóxel esta limitado a la resolución del volumen y no provee ningún medio para representar superficies de corte de forma apropiada. Por lo tanto, es necesario representar las superficies de corte irregulares que resultan por cortes graduales, de forma que permita determinar la posición e inclinación de cualquier punto en la superficie. En teoría, esto se puede lograr utilizando la descripción geométrica del instrumento de corte, pero como una superficie de corte puede estar compuesta por cientos de cortes individuales, determinar la superficie de esta forma sería computacionalmente muy costoso para una aplicación interactiva. Es por ello, que este trabajo propone transformar la representación geométrica del instrumento a una representación volumétrica (voxelización) [21].

El proceso de voxelización asemeja el efecto de volumen parcial como lo generaría un dispositivo de captura de imágenes, este efecto es necesario para la estimación precisa de las normales de la superficie. Para desplegar superficies en volúmenes tomográficos el método de sombreado de niveles de gris ha probado ser bastante preciso, por lo tanto, al utilizar un método de voxelización este método puede ser aplicado para desplegar superficies de corte arbitrarias. La voxelización se realiza mediante una técnica de filtro pesado como se muestra en la Figura 2.10 (b), en la que la punta del instrumento es muestreada a nivel subvolumétrico y un nivel de gris es determinado [21].

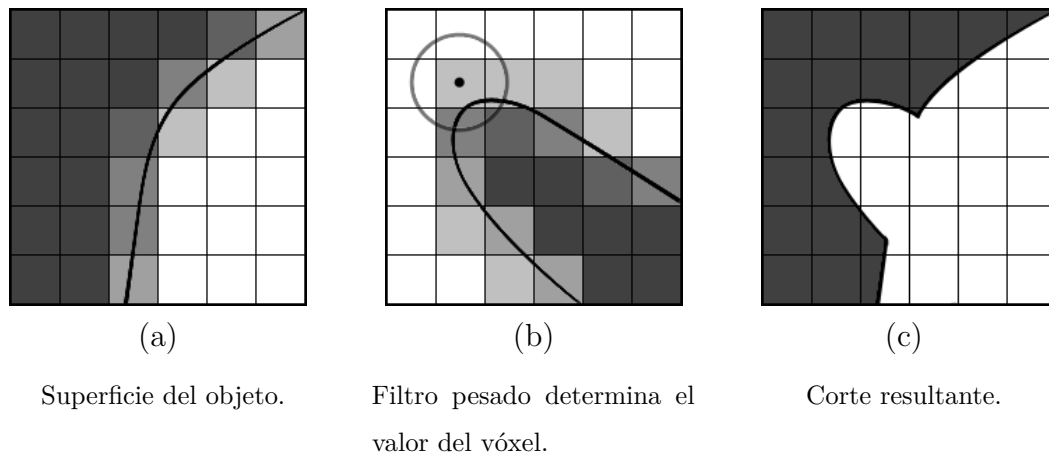
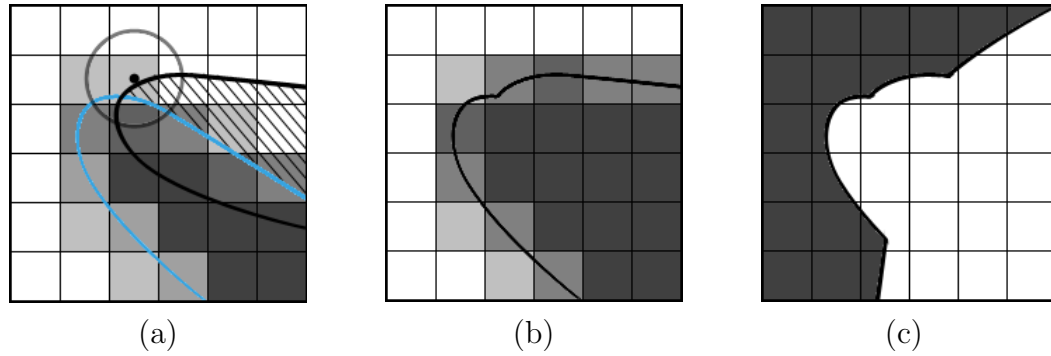


Figura 2.10: Voxelización con filtro pesado.

2.3.3. Corte Progresivo

A diferencia de otras técnicas de voxelización, en las que el propósito es transformar una representación geométrica en su contraparte volumétrica, la voxelización es utilizada para simular el corte progresivo. Por lo tanto, es necesario preservar las superficies de corte existentes y no basta con voxelizar la punta del instrumento, ya que se debe determinar la cantidad de la región de corte existente que no es afectada por el nuevo corte. Más aún, no basta con voxelizar el instrumento y combinar el nuevo valor del voxel con el valor de la superficie de corte existente. La cantidad correcta

de la región de corte existente que no es afectada por el corte nuevo sólo puede ser calculada voxelizando el instrumento y muestreando la superficie existente al mismo tiempo a nivel subvolumétrico [21], este proceso se ilustra en la Figura 2.11.



Sólo la región rayada debe
ser tomada en cuenta.

Nueva superficie de corte.

Corte resultante.

Figura 2.11: Determinación de la nueva superficie de corte.

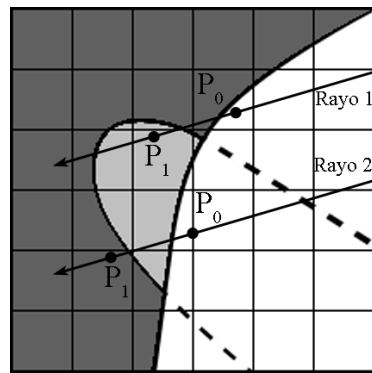
El punto a resaltar aquí, es que la intersección de una superficie de corte y una superficie de objeto genera bordes agudos, los cuales no pueden ser representados y desplegados por modelos volumétricos tradicionales sin artefactos visuales. Mas aún, no se pueden aplicar técnicas de suavizado para eliminar dichos artefactos, ya que éstas también suavizan los bordes agudos; los cuales, en el área de simulaciones médicas, deben ser modelados con la mayor precisión posible. Es por ello, que el trabajo propone una nueva técnica de visualización que combina los volúmenes independientes del modelo anatómico y las superficies de corte, que permite el despliegue libre de artefactos de la intersección de superficies [21].

2.3.4. Visualización Multivolumétrica

El despliegue de múltiples volúmenes requiere que los mismos sean combinados de forma volumétrica. Lo cual se puede conseguir mezclando los volúmenes en un solo volumen antes del despliegue [6][18] o combinando los datos de los diferentes volúmenes durante el proceso de emisión de rayos [12][3]. Todas estas técnicas se

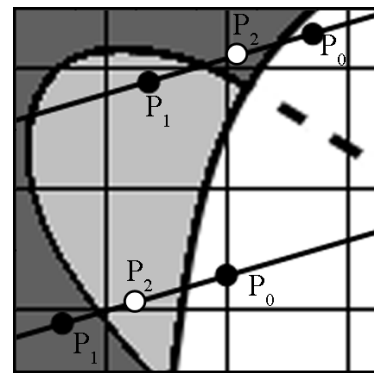
basan en operaciones de vóxel con vóxel, en las que un nuevo valor es obtenido por una regla de mezcla. De esta forma, la ubicación exacta de los puntos en la superficie se pierde y en el caso de intersección de objetos, las superficies son sólo aproximaciones. Particularmente, los bordes agudos no pueden ser representados y desplegados por estos métodos [21].

El método propuesto combina los datos de los diferentes volúmenes a nivel sub-volumétrico durante el proceso de emisión de rayos. Cuando se detecta una superficie entre puntos de muestreo sucesivos, se determina la posición de dicha superficie con una técnica de bisección. Sin embargo, cuando se despliegan múltiples volúmenes es común que más de una superficie se encuentre entre puntos sucesivos, en estos casos es necesario determinar cuál superficie debe ser desplegada. Para ello se generan puntos de muestreo entre las superficies, los cuales son clasificados y se decide que superficie debe ser mostrada. La Figura 2.12 (a) ilustra dos situaciones en las que dos superficies se encuentran entre los puntos de muestreo sucesivos (P_0 y P_1). Por lo tanto, como se muestra en la Figura 2.12 (b), se generan y clasifican nuevos puntos de muestreo (P_2) entre las superficies [21].



(a)

Situaciones con dos superficies entre puntos de muestreo.



(b)

Se generan nuevos puntos de muestreo entre las superficies.

Figura 2.12: Muestreo adaptativo determina la superficie correcta.

Capítulo 3

Diseño

Para cumplir con la propuesta realizada, se desarrolló una biblioteca dinámica (*BoneSurgeryLib*) en el lenguaje de programación C++, la cual contiene los métodos para la simulación de cirugía de huesos implementados. Luego, para mostrar dichos métodos en un ambiente interactivo, se creó una aplicación (*BoneSurgeryDemo*) utilizando el lenguaje de programación C#. Además, para facilitar la comunicación entre la biblioteca dinámica y la aplicación, se creó una biblioteca dinámica intermedia (*BoneSurgeryLibWrapper*) en el lenguaje de programación C++/CLI. Por último, para realizar las pruebas de rendimiento se desarrolló una aplicación (*BoneSurgeryTester*) en el lenguaje de programación C++, la cual utiliza la biblioteca *BoneSurgeryLib* directamente. Dicha división se encuentra ilustrada en la Figura 3.1.

3.1. BoneSurgeryLib

Como es de esperarse, el componente más importante que se desarrolló es la biblioteca dinámica que implementa las funcionalidades para la simulación de cirugía de huesos. Por lo tanto, a continuación se profundiza un poco en el diseño de la misma. En la Figura 3.2 se muestra parte del diagrama de clases de la biblioteca, sólo se muestran las clases más importantes y algunos de los métodos relevantes. Como se puede apreciar, la biblioteca exporta los métodos públicos de las clases *Config*, *MacroAction* y *SimulationSystem*. La clase *Config* se utiliza para manejar la configuración

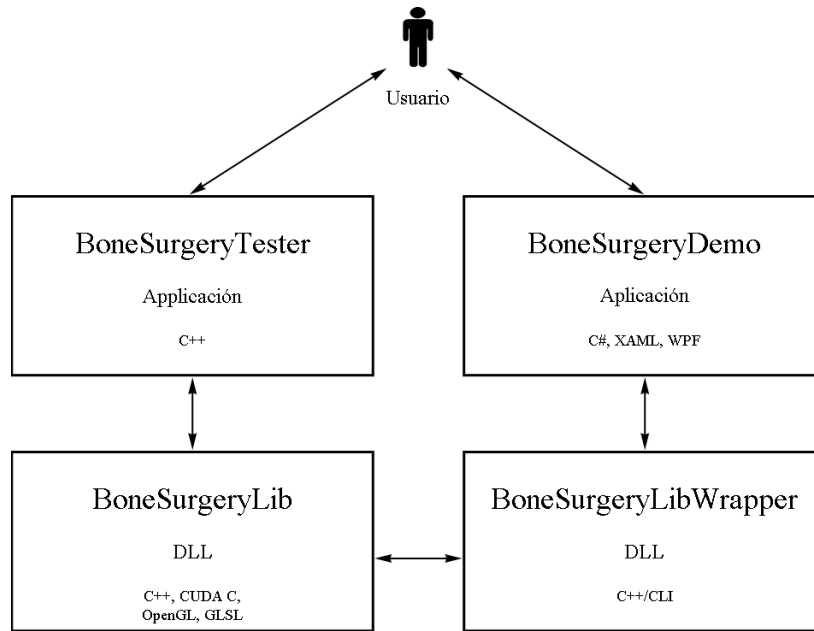


Figura 3.1: Diseño general.

del sistema, entre los valores que se pueden modificar se encuentran la sensibilidad del ratón, el valor de la isosuperficie a mostrar y el modelo de iluminación a utilizar. La clase *MacroAction* representa una acción realizable por el usuario, como cargar un volumen, mover el instrumento y eliminar un fragmento de hueso; esto permite almacenar una secuencia de acciones para reproducirlas de forma idéntica en otro momento. Finalmente, la clase *SimulationSystem* es la encargada de comunicarse con el resto del sistema de simulación y representa el punto de comunicación más importante con el sistema.

Las clases *Config* y *SimulationSystem* utilizan el patrón de diseño *Singleton* para garantizar que exista una única instancia, accesible para todo el sistema a través del método estático *Instance*. Antes de poder utilizar el sistema, éste debe ser iniciado a través del método *InitializeSystem*. Una vez iniciado, se puede cargar un volumen utilizando el método *OnLoadVolume* para ser manipulado. Se puede cambiar el instrumento de corte llamando al método *SetInstrument*, el cual carga un instrumento descrito en un archivo en formato XML, para leer estos archivos se utilizó la biblioteca

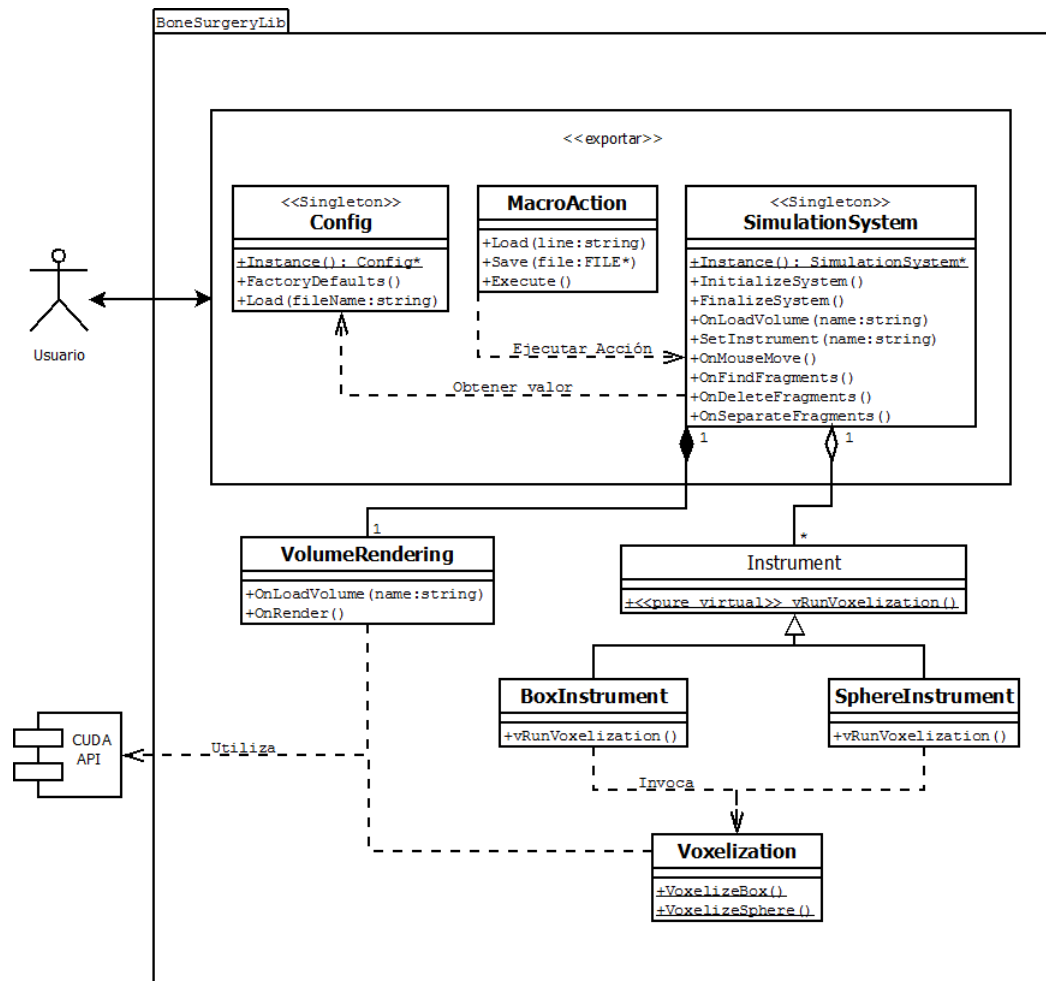


Figura 3.2: Diagrama de clases simplificado de la biblioteca BoneSurgeryLib.

de código abierto **pugixml**¹. La posición y orientación del instrumento es modificado llamando al método *OnMouseMove*, el cual asume la utilización de un ratón tradicional. Sin embargo, tanto la voxelización como el despliegue son independientes del movimiento del ratón, únicamente requieren la información de la transformación. De esta forma la integración de un dispositivo diferente es trivial y transparente para el sistema de simulación.

También es posible detectar fragmentos de hueso separados utilizando el método *OnFindFragments*, para luego eliminar o separar algunos utilizando los métodos

¹Disponible en la página <http://pugixml.org>

OnDeleteFragments y *OnSeparateFragments* respectivamente. Finalmente, se utiliza el método *FinalizeSystem* para detener la simulación y liberar los recursos, la Figura 3.3 muestra un diagrama que ilustra una secuencia de acciones típica al utilizar el sistema.

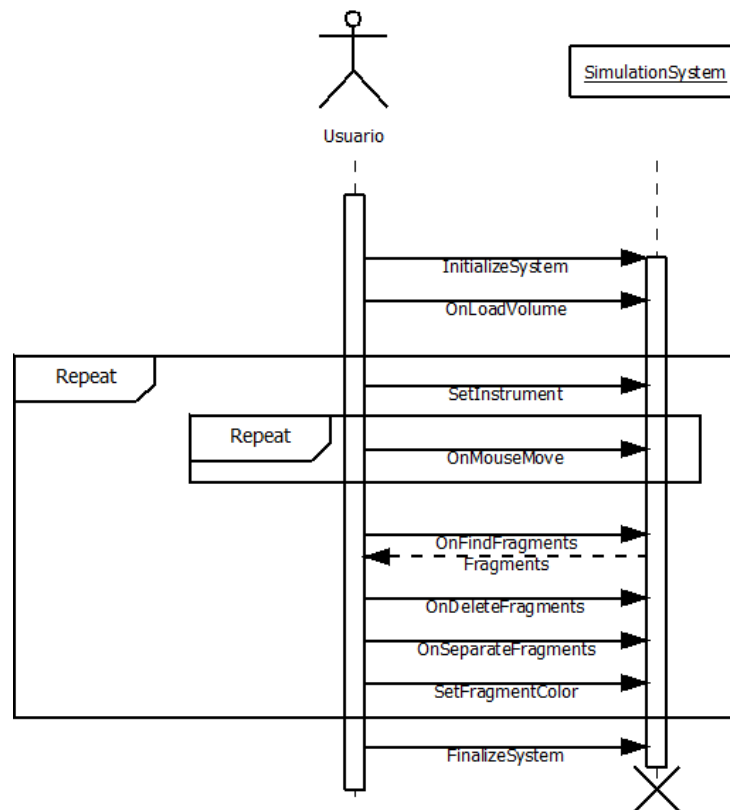


Figura 3.3: Posible secuencia de acciones.

Uno de los objetivos en el diseño de la biblioteca era soportar una gran variedad de instrumentos de corte sin modificar el código de la misma. Para lograr dicho objetivo, la clase *Instrument* se diseñó utilizando el patrón de diseño *Composite*, el cual permite definir objetos compuestos por otros objetos del mismo tipo (los cuales pueden ser simples o compuestos). De esta forma, los instrumentos son composiciones de primitivas u otros instrumentos compuestos pero el código cliente de la clase *Instrument* no requiere conocimiento sobre la composición del mismo. Las primitivas soportadas actualmente son la esfera y el ortoedro (paralelepípedo ortogonal), aunque

agregar nuevas primitivas al utilizar el patrón de diseño *Composite* no ofrece mayor complicación.

3.2. BoneSurgeryDemo

La aplicación interactiva tiene como propósito mostrar las funcionalidades implementadas. Como se muestra en la Figura 3.4, ésta posee una interfaz gráfica sencilla e intuitiva que permite al usuario interactuar con la biblioteca de simulación sin conocer los detalles de implementación. Se utilizó el subsistema gráfico WPF (*Windows Presentation Foundation*) para el desarrollo de la interfaz gráfica, el cual utiliza un lenguaje derivado de XML llamado XAML (*Extensible Application Markup Language*) para definir la interfaz gráfica. Al desarrollar aplicaciones que utilizan WPF, se puede escoger C++/CLI o C# como lenguaje de programación. Sin embargo, si se utiliza C++/CLI existe una serie de restricciones para incorporar XAML, particularmente, no se puede utilizar XAML compilado; por lo tanto, se decidió utilizar C# para la aplicación y obtener todas las ventajas posibles de XAML.

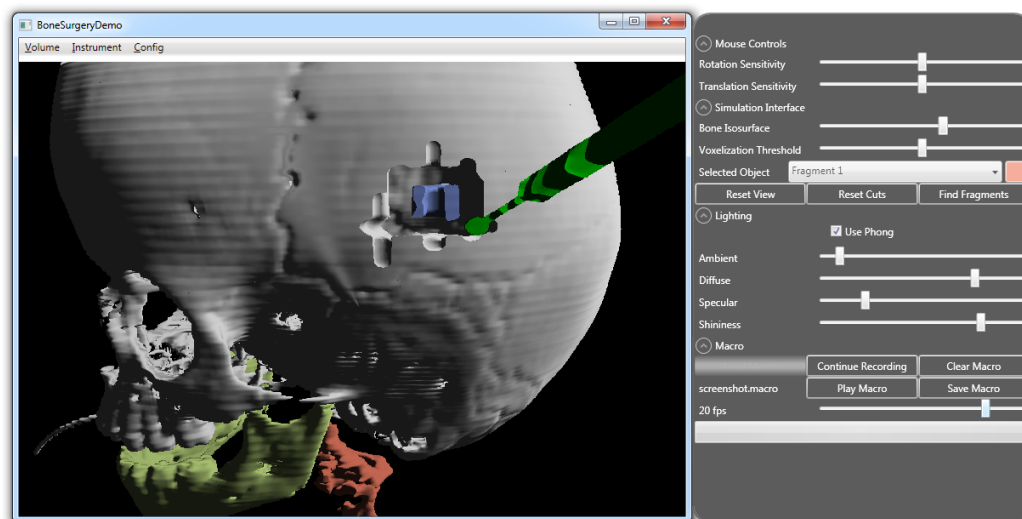


Figura 3.4: Interfaz gráfica utilizada por la aplicación de demostración

Al utilizar la aplicación, ésta se comunica con la biblioteca de simulación, permi-

tiendo al usuario cargar un volumen, cambiar el instrumento, mover la herramienta y borrar o separar fragmentos de hueso. Además, permite asignar diferentes colores a los fragmentos separados, ajustar la sensibilidad de movimiento, modificar el modelo de iluminación y grabar una secuencia de acciones (macros), las cuales pueden ser cargadas y reproducidas en otro momento.

3.3. BoneSurgeryLibWrapper

Para facilitar la comunicación entre la biblioteca de simulación y la aplicación de muestra, se desarrolló una biblioteca intermedia en C++/CLI. Teóricamente era posible utilizar la biblioteca de simulación directamente, pero el código para incorporar una biblioteca en código no manejado en una aplicación de C# es engorroso y restrictivo. En cambio, una biblioteca escrita en C++/CLI puede comunicarse libremente tanto con la aplicación en C# como la biblioteca en C++.

El diseño de esta biblioteca es bastante simple, para cada método de la biblioteca de simulación que la aplicación de demostración necesita llamar, existe un método en la biblioteca intermedia. Dicho método de la biblioteca intermedia, simplemente transforma los datos de entrada al formato esperado por la biblioteca de simulación, llama al método correspondiente en la biblioteca nativa y transforma los resultados al formato esperado por la aplicación de demostración.

3.4. BoneSurgeryTester

Generalmente, al realizar mediciones precisas acerca del tiempo de ejecución de un algoritmo, cierta sobrecarga de cómputo es introducida. Lo que significa que el algoritmo podría ejecutarse más rápido si no es necesario realizar las mediciones de tiempo, situación que se agrava aún más para algoritmos paralelos. Por lo tanto, el sistema de mediciones de tiempo en la biblioteca de simulación es activado y desactivado durante la compilación por medio de definiciones del preprocesador de C++. De

esta forma, al desactivar la compilación de mediciones de tiempo se garantiza que el sistema de simulación se ejecuta con el mayor rendimiento posible.

Para realizar las mediciones de tiempo es necesario comparar la diferencia del momento en el que ocurren dos eventos, el inicio y el final de la ejecución de una sección, para simplificar la situación se considera que la sección que se desea medir esta compuesta por una única llamada a función. De esta forma, un método común para tomar el tiempo requerido por una función se muestra en el Algoritmo 3.1, en el Paso 4 se utiliza una función que transforma de la unidad utilizada por el CPU a la unidad de tiempo deseada, por ejemplo, de ciclos de reloj a milisegundos.

-
1. inicio = marca de tiempo actual del CPU.
 2. **Invocar** la función a la que se le desea medir el tiempo.
 3. final = marca de tiempo actual del CPU.
 4. tiempo = Transformar(final - inicio).
-

Algoritmo 3.1: Medición de tiempo común

3.4.1. Medición Correcta de Tiempo para Operaciones de CUDA

El método mostrado anteriormente para medir el tiempo utilizado por una función es simple, preciso y poco costoso para la mayoría de los casos. Sin embargo, no se puede utilizar para realizar mediciones de tiempo relacionadas a la ejecución de los *kernels* o transferencias de memoria al utilizar la arquitectura CUDA. Como se mencionó en la Sección 2.1, la invocación de un *kernel* y las peticiones de transferencias de memoria son asíncronas por defecto, lo que significa que el anfitrión recibe control del procesador inmediatamente después de realizar la petición. Por lo tanto, si la función invocada es un *kernel* de CUDA, la marca de tiempo final obtenida en el Paso 3 no tiene relación alguna con el tiempo en el que la ejecución del *kernel* finalizó, es altamente probable

que el mismo se encuentre aún en ejecución dentro del dispositivo. Más aún, es posible que la marca de tiempo inicial tomada en el Paso 1 sea irrelevante, ya que es posible que algún *kernel* o transferencia de memoria anterior aún se encuentre activa en ese momento; de ser éste el caso, al invocar el *kernel* en el Paso 2 el anfitrión debe esperar que el dispositivo se libere antes de realizar la nueva petición [20].

Un método preciso es utilizar los mecanismos de sincronización provistos por la arquitectura CUDA, específicamente, es necesario colocar un punto de sincronización antes de tomar cada una de las marcas de tiempo [20]. El Algoritmo 3.2 muestra la manera correcta de realizar mediciones de tiempo para operaciones asíncronas en CUDA.

-
1. **Esperar** a que el dispositivo finalice cualquier petición anterior invocando a `cudaThreadsSynchronize()`.
 2. inicio = marca de tiempo actual del CPU.
 3. **Invocar** un *kernel* o una transferencia de memoria.
 4. **Esperar** a que el dispositivo finalice la petición realizada invocando a `cudaThreadsSynchronize()`.
 5. final = marca de tiempo actual del CPU.
 6. tiempo = Transformar(final - inicio).
-

Algoritmo 3.2: Medición de tiempo en la arquitectra CUDA

3.4.2. Reloj de Alta Precisión

Los Pasos 1 y 3 del Algoritmo 3.1 requieren obtener las marcas de tiempo actuales del CPU, mientras que el Paso 4 requiere transformar las marcas en una unidad de tiempo apropiada. En la actualidad la mayoría de los procesadores poseen un reloj de alta precisión incorporado que puede ser utilizado con este propósito. Obtener la marca de tiempo del reloj de alta precisión es relativamente sencillo desde un punto de vista de programación, sin embargo, existen dos problemas de suma importancia que deben ser tomadas en cuenta. El primero es que cada núcleo de un procesador de

múltiples núcleos posee su propio reloj, además, el tiempo exacto en cada reloj suele desviarse a medida que el tiempo transcurre [7]. El segundo problema viene dado por un error en el diseño de ciertos fabricantes de tarjetas madres², por el cual es posible que el valor del reloj del procesador se adelante de forma inesperada, incluso varios segundos.

Con la intención de garantizar la mayor fiabilidad posible en las mediciones de tiempo, se decidió utilizar la implementación del reloj de alta precisión provista por la biblioteca de código abierto **Ogre3D**³, dicha implementación contempla y maneja correctamente los problemas mencionados anteriormente, además de proveer facilidades para transformar las mediciones a diferentes unidades de tiempo.

²Error *KB274323*: La descripción del problema y una alternativa para remediarlo se encuentran disponibles en <http://support.microsoft.com/kb/274323>

³Disponible en la página <http://www.ogre3d.org>

Capítulo 4

Implementación

Las funcionalidades relevantes a la propuesta de Trabajo Especial de Grado desarrolladas en la biblioteca de simulación son el corte de volumen, el despliegue volumétrico y la detección y manipulación de fragmentos separados. Por lo tanto, este capítulo muestra los detalles de implementación de dichas funcionalidades.

4.1. Corte de Volumen

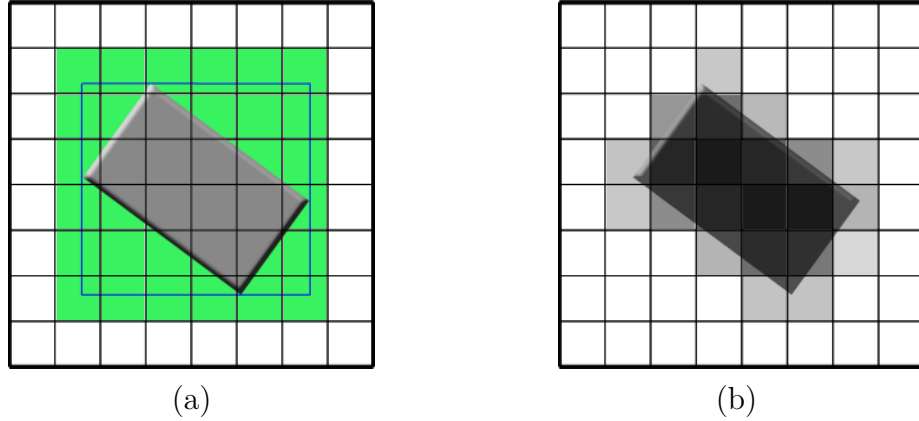
La base del sistema requiere realizar cortes sobre el volumen médico utilizando primitivas geométricas. Este proceso puede dividirse en dos grandes componentes, la primera debe voxelizar la representación geométrica del instrumento, mientras que la segunda debe combinar los resultados de la voxelización con las superficies de corte existentes.

4.1.1. Voxelización del Instrumento

Para el proceso de voxelización del instrumento se utilizó una estrategia similar a la propuesta en [21], en la que la punta del instrumento era voxelizado utilizando una técnica de filtro pesado para determinar el nivel de gris del vóxel correspondiente. La implementación realizada posee las mismas características teóricas, pero en vez de

utilizar un filtro pesado, se toman muestras a nivel subvolumétrico y se explota el paralelismo inherente en los cálculos para determinar el nivel de gris del vóxel.

Como se muestra en la Figura 4.1, la técnica se implementó utilizando dos etapas. Primero se calcula el AABB (ortopedro delimitador alineado a los ejes) del instrumento de corte en coordenadas de volumen¹ y se determinan todos los vóxeles que lo intersectan. Luego, para cada uno de estos vóxeles se realiza un muestreo pesado para determinar un valor de gris adecuado para el mismo.



Se determina los vóxeles que tocan el AABB.

A cada vóxel se le asigna un nivel de gris dependiendo del volumen de vóxel que se encuentra dentro del instrumento.

Figura 4.1: Voxelización del instrumento.

La primera etapa se realiza construyendo un ortopedro delimitador orientado en coordenadas de objeto, el cual es llevado a coordenadas de volumen utilizando la matriz de transformación del instrumento. Finalmente, tomando los valores mínimos y máximos en los tres ejes del ortopedro transformado se obtienen las esquinas del AABB. Dado que el AABB se encuentra alineado a los ejes del volumen, obtener la lista de vóxeles que se encuentran en contacto con el mismo se realiza de forma trivial, las ecuaciones (4.1) y (4.2) muestran el cálculo para obtener los índices del primer y

¹En este trabajo se considera que en coordenadas de volumen, el volumen es un cubo centrado en el origen cuyo lado mide dos unidades.

último vóxel en el eje x , los cálculos para los otros ejes son equivalentes.

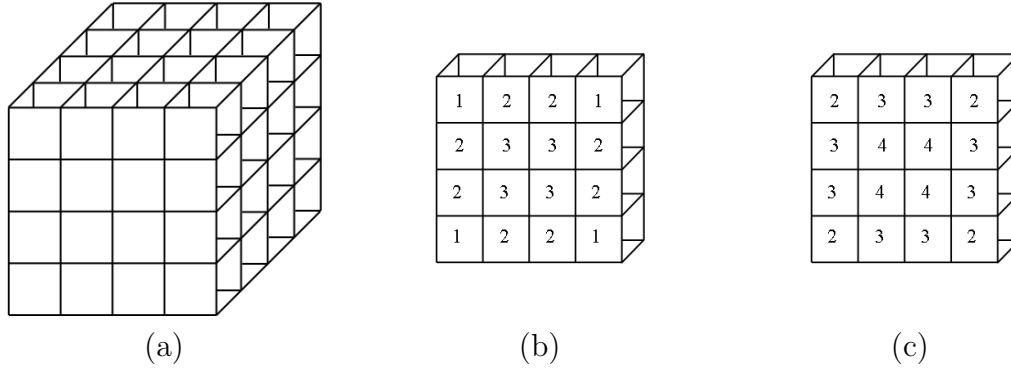
$$k_{\min} = \left\lfloor \frac{(\min_x + 1) D_x - 1}{2} - \epsilon \right\rfloor \quad (4.1)$$

$$k_{\max} = \left\lfloor \frac{(\max_x + 1) D_x - 1}{2} + \epsilon \right\rfloor + 1 \quad (4.2)$$

Donde $\min_x$ representa la coordenada mínima en el eje x del AABB y D_x es la cantidad de vóxeles en el eje x . Para evitar que los errores de redondeo descarten vóxeles en los bordes, se utiliza un valor pequeño ϵ para agrandar el AABB un poco. De esta forma, en el peor caso se consideran vóxeles que se encuentran cerca del AABB sin intersectarlo pero nunca se descartan vóxeles potenciales.

En la segunda etapa es necesario determinar la cantidad de volumen de cada vóxel dentro del instrumento de corte, esto se realiza tomando 64 muestras regularmente espaciadas dentro del vóxel y determinando si dicha muestra se encuentra dentro del instrumento. Las muestras son pesadas de forma inversa a su distancia al centro, es decir, las muestras cercanas al centro tienen mayor peso; la Figura 4.2 muestra la subdivisión del vóxel y los pesos de las primeras 32 muestras, las otras 32 muestras son equivalentes. El peso de las muestras que se encuentran dentro del instrumento son acumulados y el resultado es normalizado (dividiendo por la acumulación de todos los pesos posibles), este proceso produce un valor $g \in [0, 1]$ para cada vóxel el cual es utilizado para determinar el nivel de gris del mismo.

Entrando en mayor detalle, la implementación genera N bloques de 32 hilos CUDA, donde N es el número de vóxeles que intersectan el AABB. Esta organización sugiere que el nivel de gris de cada vóxel es calculado cooperativamente por 32 hilos, el número 32 se utilizó para aprovechar por completo el paralelismo que ofrecen los *warp* de CUDA, si se utilizan más hilos por vóxel es necesario incluir puntos de sincronización y si se utilizan menos se producen mayores niveles de divergencia de ramas. Cada uno de los hilos toma dos muestras, de esta forma se consigue tomar 64 muestras por vóxel. Durante la ejecución del *kernel* cada hilo utiliza su índice de hilo



El vóxel es dividido en 64 regiones iguales, las muestras son tomadas en el centro de estas regiones.

Pesos de las muestras de las capas más lejanas al centro.

Pesos de las muestras de las capas más cercanas al centro.

Figura 4.2: Pesos de las muestras

para determinar las coordenadas de las muestras que debe probar, para cada muestra su posición es transformada a coordenadas de objeto ya que calcular si un punto se encuentra dentro de la primitiva es extremadamente eficiente en coordenadas de objeto.

Cada bloque de hilos utiliza un arreglo de 32 posiciones en la eficiente memoria compartida del dispositivo, en donde cada vóxel almacena la suma de los pesos de las dos muestras que debe calcular. Luego, se realiza un proceso de reducción para acumular los valores y obtener un sólo valor como resultado. Este proceso puede realizarse en $O(\log_2 N)$, donde N es el número de valores a reducir, notando que el problema puede realizarse cooperativamente entre los hilos del *kernel*. La idea principal es disminuir la cantidad de números a acumular en aproximadamente la mitad en cada paso, lo que significa que la cantidad de pasos a realizar es $O(\log_2 N)$. Por lo tanto, para lograr que el algoritmo completo mantenga el mismo orden, es necesario que cada paso sea realizado en $O(1)$; para ello es necesario realizar todas las sumas de un paso en paralelo, el Algoritmo 4.1 muestra el código para realizar la reducción logarítmica mientras que la Figura 4.3 ilustra gráficamente el proceso.

1. Sea N el número de valores a reducir.
2. Sea A el arreglo de N valores a reducir.
3. Mientras $N > 1$, hacer:
 4. | Calcular el número valores a reducir en este paso $V = N \text{ div } 2$.
 5. | Para cada hilo con índice $i < V$, hacer:
 6. | | Acumular $A[i] = A[i] + A[i + V]$.
 7. | | Actualizar $N = N - V$.
8. Retornar el resultado de la reducción, el cual se encuentra en $A[0]$.

Algoritmo 4.1: Reducción Logarítmica

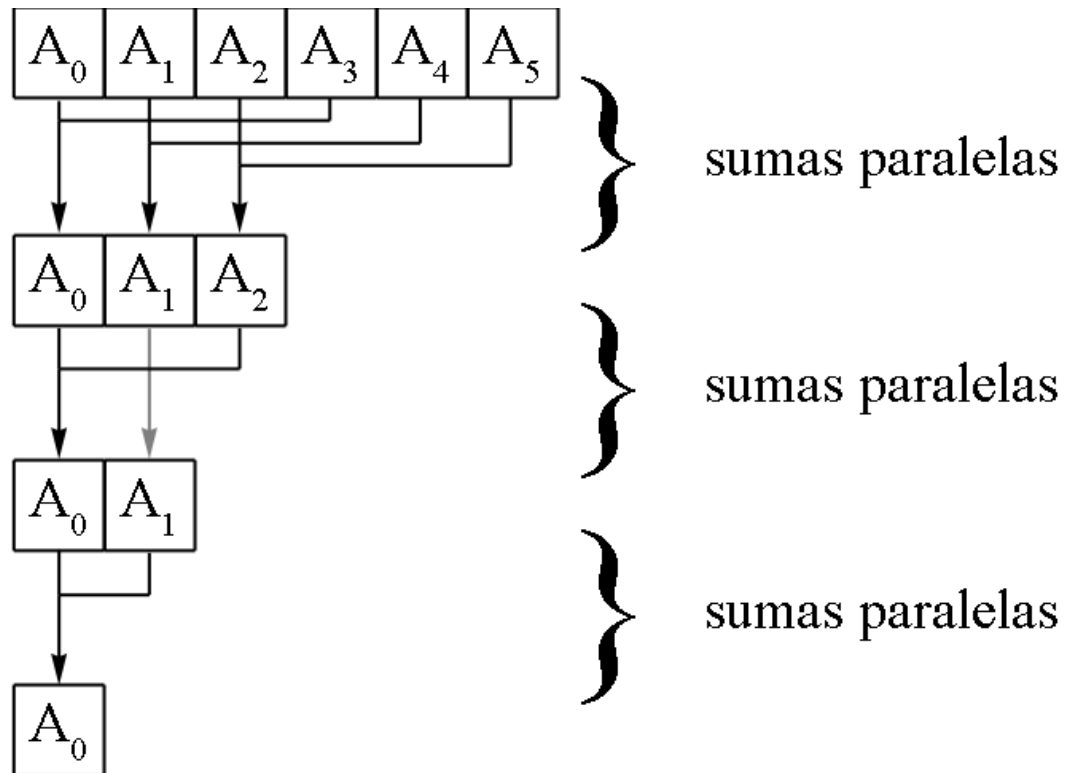


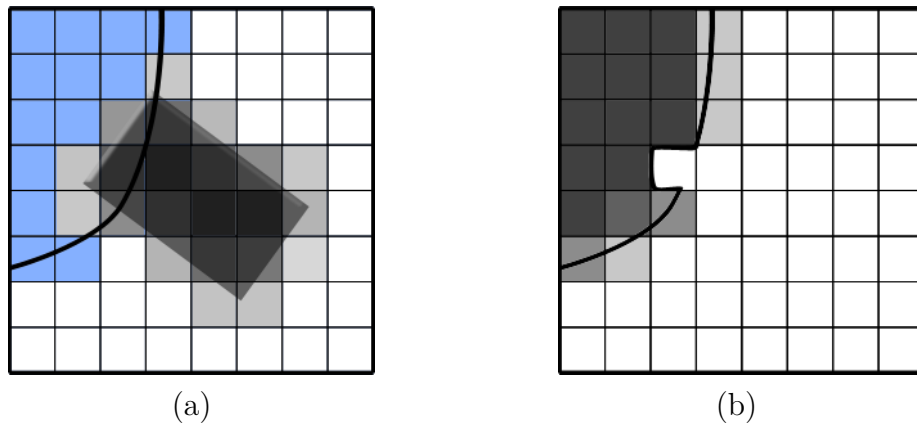
Figura 4.3: Reducción Logarítmica

Todos los hilos son capaces de probar la condición del Paso 5 en paralelo, más aún, aquellos que pasan la prueba realizan la suma del Paso 6 en paralelo. Realizar dicha reducción dentro del *kernel* de voxelización es extremadamente eficiente, ya que se utilizan los mismos hilos que tomaron las muestras. Por lo tanto, no hay transferencias

de datos entre el anfitrión y el dispositivo ni requerimientos de planificación de *warps* adicionales.

4.1.1.1. Combinación de los Resultados

La primera estrategia que se implementó utilizaba una regla muy sencilla para combinar los resultados de la voxelización, si la cantidad de volumen de un vóxel que se encontraba dentro del instrumento de corte superaba cierto umbral el vóxel del volumen médico era eliminado. Aunque dicha estrategia ofrecía buen rendimiento, tanto en tiempo como en espacio, generaba artefactos visuales notables. Esto se debe a que la resolución de dicho enfoque se encuentra limitada por la resolución del volumen y no es posible representar las intersecciones arbitrarias entre el volumen y el instrumento. Esta estrategia se encuentra ilustrada en la Figura 4.4.



Niveles de gris generados por la voxelización, como referencia se muestra el volumen médico en azul.

Los vóxeles para los que el nivel de gris supera el umbral son eliminados del volumen médico.

Figura 4.4: Primera estrategia de corte de volumen

Debido a los problemas mencionados se decidió utilizar una versión simplificada a la propuesta en [21], la idea básica es representar las regiones de corte con un modelo volumétrico separado y trasladar el problema de la visualización sin artefactos

al algoritmo de despliegue. La simplificación que se realizó durante la combinación de los resultados es que el nivel de gris de un vóxel en el volumen de corte es actualizado con la sencilla regla mostrada en (4.3), donde $C(x, y, z)$ es el nivel de gris del vóxel en el volumen de corte y $G(x, y, z)$ es el nivel de gris determinado por la voxelización actual.

$$C(x, y, z) = \text{máx}(C(x, y, z), G(x, y, z)) \quad (4.3)$$

4.2. Despliegue Volumétrico

Originalmente se decidió adaptar alguno de los algoritmos de despliegue volumétrico ya implementados y optimizados en el Centro de Computación Gráfica de la Universidad Central de Venezuela. Sin embargo, cuando se adoptó la estrategia de combinación de resultados de la voxelización mencionada anteriormente, la necesidad de un algoritmo de despliegue nuevo y específico al problema se hizo evidente.

Para poder realizar el despliegue de la representación multivolumétrica adoptada, se implementó un algoritmo de despliegue volumétrico basado en emisión de rayos, modificado para determinar la posición correcta de las superficies del objeto que no han sido cortadas. Por lo tanto, el algoritmo resultante no es un algoritmo de despliegue volumétrico de propósito general. Sin embargo, las diferencias del algoritmo ocurren únicamente durante la determinación del color de un rayo particular, es decir, al igual que en el Algoritmo 2.1 se debe emitir un rayo desde el ojo del observador a través del centro de cada píxel de la imagen y determinar un color para cada uno de estos rayos.

4.2.1. Emisión de Rayos en Paralelo

Como se mencionó anteriormente, la lógica para la emisión de los rayos es igual que la propuesta en el Algoritmo 2.1, sin embargo, la implementación utiliza las capacidades de cómputo paralelo del GPU para obtener un mayor rendimiento. Observando el algoritmo básico, es fácil notar que el color de un píxel depende únicamente de un rayo

y que los rayos realizan su trabajo independientemente de los demás. Por lo general, cuando un problema exhibe dichas características, se puede desarrollar un algoritmo paralelo sin mayores dificultades. De esta forma, el algoritmo implementado utiliza un hilo de CUDA para cada píxel de la imagen, dispuestos en un bloque bidimensional para obtener una correspondencia lógica directa con las dimensiones de la pantalla. El bloque de hilos utilizados es de 16×16 hilos, la decisión que llevó a dicho tamaño se puede resumir en tres razones:

1. Los múltiplos de medio *warp* (16) ofrecen buenos patrones de acceso a memoria, este se debe a la unificación de muchas de las peticiones.
2. Un bloque de 32×32 hilos, necesita ejecutar 1024 hilos residentes en el mismo microprocesador. Aunque los dispositivos de CUDA más modernos soportan dicha cantidad de hilos, la mayoría de los chips actuales soportan máximo 512.
3. Bloques de 32×16 hilos generaban mayor divergencia de ramas dentro del *kernel*.

4.2.2. Detección de Isosuperficies con un Volumen de Corte

El algoritmo de emisión de rayos para la detección de isosuperficies sobre un único volumen sólo necesita determinar en cada paso si el rayo atravesó la isosuperficie deseada. Sin embargo, este enfoque no es suficiente al incorporar el volumen de corte. El análisis debe tomar en cuenta la ubicación de las superficies de corte, si el rayo atraviesa la isosuperficie deseada debe asegurarse que ésta no ha sido removida por el instrumento de corte. Por lo tanto, para simplificar el análisis, se define el estado del rayo en el tiempo t como un par ordenado $S_t = \langle O, C \rangle$, donde O y C son valores *booleanos* que representan si la muestra se encuentra dentro del objeto y dentro de un corte respectivamente. Al tomar la muestra en el tiempo t , se analiza el cambio de estado del rayo con respecto al estado anterior (S_{t-1}) para determinar si se encontró una isosuperficie que debe ser desplegada. Como el rayo puede estar en uno de cuatro estados en el tiempo $t - 1$ y en uno de cuatro estados en el tiempo t , existen

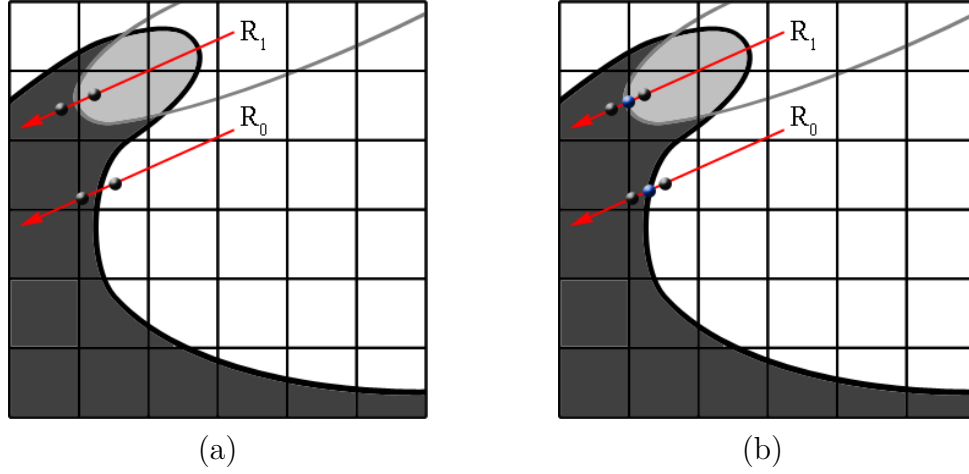
Caso	$S_{t-1} \langle O, C \rangle$	$S_t \langle O, C \rangle$	Condición para detener el rayo	Superficie a mostrar
1	$\langle F, F \rangle$	$\langle V, F \rangle$	Siempre	Objeto
2	$\langle V, V \rangle$	$\langle V, F \rangle$	Siempre	Corte
3	$\langle F, F \rangle$	$\langle V, V \rangle$	Si el objeto es encontrado primero	Objeto
4	$\langle V, V \rangle$	$\langle F, F \rangle$	Si el corte es encontrado primero	Corte
5	$\langle F, V \rangle$	$\langle V, F \rangle$	Siempre	La última que es atravesada

Tabla 4.1: Casos en los que rayo debe detenerse

16 casos potenciales. Sin embargo, sólo cinco de ellos deben ser analizados, los cuales se pueden ver en la Tabla 4.1, que detalla el estado del rayo en la muestra anterior, el estado en la muestra actual, la condición para detener el rayo y la superficie que debe ser sombreada.

Los primeros dos casos son triviales y se caracterizan por encontrar una sola superficie entre dos puntos de muestreo consecutivos. El primer caso ocurre cuando el rayo se encontraba fuera del objeto y del corte en el punto anterior, pero el punto nuevo determina que sólo el objeto fue encontrado. En este caso el rayo simplemente determina la posición de la isosuperficie del objeto y sombrea en la intersección del rayo con dicha superficie. El segundo caso ocurre cuando el rayo se encontraba dentro del objeto y de la superficie de corte en el punto anterior, pero la nueva muestra determina que el rayo salió del corte sin salir del objeto. Cuando esto ocurre, el rayo determina el punto donde el rayo intersecta la superficie de corte y sombrea dicho punto. Ambos casos se encuentran ilustrados en la Figura 4.5 por los rayos R_0 y R_1 respectivamente.

El tercer caso a tomar en cuenta ocurre cuando el rayo se encuentra fuera del



Los rayos R_0 y R_1 se encuentran en los casos 1 y 2 respectivamente. R_0 :

$$\langle F, F \rangle \rightarrow \langle V, F \rangle$$

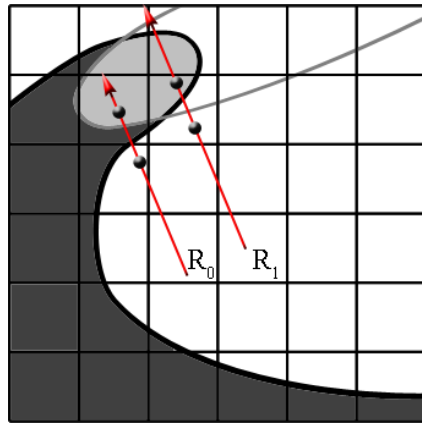
$$R_1: \langle V, V \rangle \rightarrow \langle V, F \rangle$$

El rayo debe detenerse sobre la única superficie que se encuentra entre los puntos de muestreo.

Figura 4.5: Casos simples en los que el rayo debe detenerse.

objeto y del corte en la muestra anterior, pero encuentra ambas superficies antes de la muestra actual. Como se puede observar en la Figura 4.6, existen dos posibilidades en este caso. La primera posibilidad, ilustrada por el rayo R_0 , es entrar primero en el objeto y luego en el corte; en este caso el rayo debe sombrear el punto en el que encuentra la superficie del objeto. Sin embargo, como lo muestra el rayo R_1 , es posible entrar primero en el corte y luego en el objeto; cuando esto ocurre, la superficie del objeto encontrada no debe ser mostrada, por lo que el rayo continúa su camino.

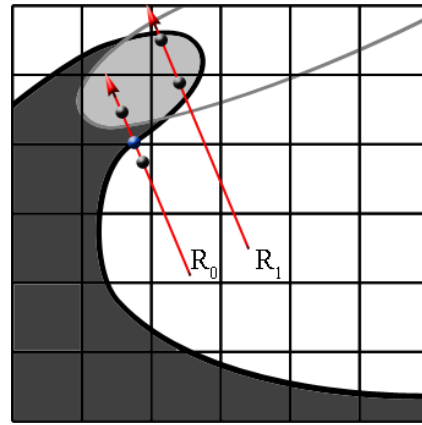
El cuarto caso ocurre cuando el rayo sale tanto del objeto como del corte entre dos muestras consecutivas. Este caso, ya que es el caso inverso, debe tratarse de forma contraria al anterior. Si el rayo sale primero del corte que del objeto, la intersección con la superficie de corte es sombreada y el rayo se detiene. Por el contrario, si la primera superficie encontrada es la del objeto, el rayo debe continuar su camino. Ambas situaciones se encuentran ilustradas en la Figura 4.7 por los rayos R_0 y R_1 respectivamente.



(a)

Ambos rayos se encuentran en el caso 3.

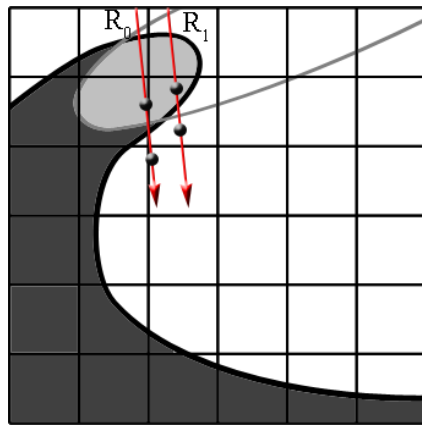
$$\langle F, F \rangle \rightarrow \langle V, V \rangle$$



(b)

El rayo R_0 debe detenerse al cruzar la superficie del objeto, mientras que el rayo R_1 debe continuar tomando muestras.

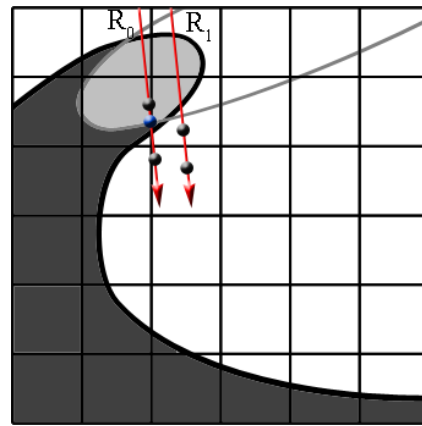
Figura 4.6: El rayo entra en el objeto y en el corte entre dos muestras consecutivas.



(a)

Ambos rayos se encuentran en el caso 4.

$$\langle V, V \rangle \rightarrow \langle F, F \rangle$$



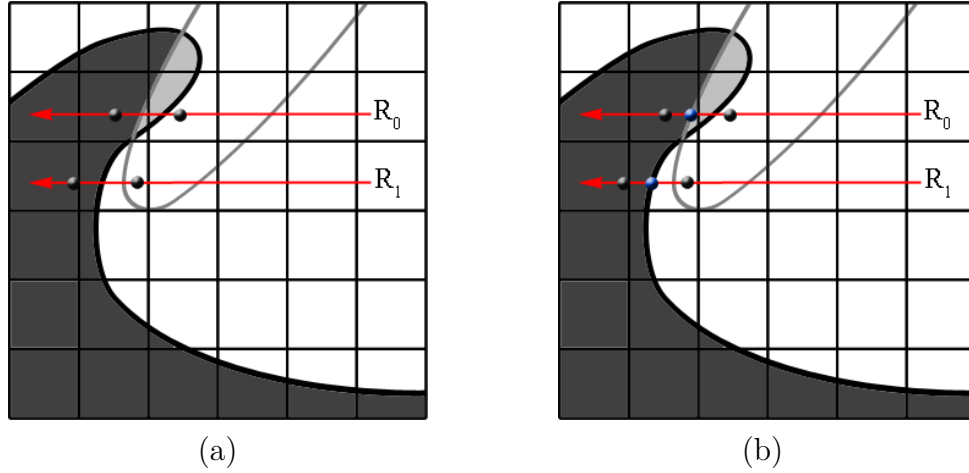
(b)

El rayo R_0 debe detenerse al cruzar la superficie de corte, mientras que el rayo R_1 debe continuar tomando muestras.

Figura 4.7: El rayo sale del objeto y del corte entre dos muestras consecutivas.

Por último, el quinto caso ocurre cuando el rayo se encuentra fuera del objeto y dentro del corte en la muestra previa, pero encuentra ambas superficies antes de la

muestra actual. Cuando esto ocurre el rayo siempre debe parar, ya que encontró una superficie que debe mostrar. Sin embargo, este caso debe determinar si mostrar la intersección con la superficie del objeto o con la superficie de corte, dependiendo de cual encuentra de último. Como se puede ver en la Figura 4.8, cuando el rayo atraviesa la superficie del objeto primero (R_0) este debe utilizar la posición de la superficie de corte, ya que una parte del objeto ha sido cortada. Por el contrario, cuando la primera que atraviesa el rayo es la superficie de corte (R_1), se debe sombrear la posición donde atraviesa la superficie del objeto.



Ambos rayos se encuentran en el caso 5.

$$\langle F, V \rangle \rightarrow \langle V, F \rangle$$

Ambos rayos se detienen sobre la segunda superficie que encuentran, R_0 debe detenerse al cruzar la superficie de corte, mientras que el rayo R_1 debe detenerse sobre la superficie del objeto.

Figura 4.8: El rayo sale del corte y entra al objeto entre dos muestras consecutivas.

4.2.3. Estimación del Gradiente en la Frontera de las Superficies de Corte

El modelo de iluminación utilizado en el algoritmo de despliegue volumétrico requiere la estimación del vector gradiente, como se mencionó en la Sección 2.2.1.2,

Símbolo	Significado
p	Punto en el espacio euclídeo de tres dimensiones, es decir, $p \in \mathbb{E}^3$
$V(p)$	Valor de densidad del volumen original en el punto p
$C(p)$	Valor de densidad del volumen de las superficies de corte en el punto p
$V'(p)$	Valor de densidad del volumen que resulta de aplicar los cortes al volumen original en el punto p
$\Delta V(p)$	Gradiente del volumen original en el punto p , estimado con diferencias finitas
$\Delta C(p)$	Gradiente del volumen de las superficies de corte en el punto p , estimado con diferencias finitas
$\Delta V'(p)$	Gradiente del volumen que resulta de aplicar los cortes al volumen original en el punto p

Tabla 4.2: Notación utilizada en las fórmulas de estimación del gradiente

esto se suele realizar utilizando diferencias finitas como método de aproximación a las derivadas parciales requeridas en (2.3). Para la mayoría de las aplicaciones es posible calcular el gradiente en cada centro de vóxel, almacenar dichos valores en una textura e interpolar los vectores durante el despliegue. Sin embargo, al modelar cortes sobre el volumen es necesario estimar el vector gradiente en la frontera de las superficies de corte. En lo que sigue se utiliza la notación mostrada en la Tabla 4.2.

En este trabajo se proponen e implementan varios métodos para estimar el vector gradiente en la frontera de la superficies de corte. Específicamente, se desea estimar el valor de $\Delta V'(p), \forall p \in \mathbb{E}^3$. El primer método considera el gradiente de la superficie del objeto y el gradiente de la superficie de corte igual de importantes, por lo tanto, se considera que el gradiente resultante podría encontrarse cerca del promedio de ambos.

$$\Delta V'(p) = \left(\frac{\Delta V(p) + \Delta C(p)}{2} \right) \quad (4.4)$$

En (4.4) se muestra analíticamente este método, sin embargo, al analizar la Figura 4.9 este método determina vectores gradientes totalmente incorrectos en la mayor parte de la frontera de corte. Específicamente, mientras más profundo el corte mayor será el error de la estimación.

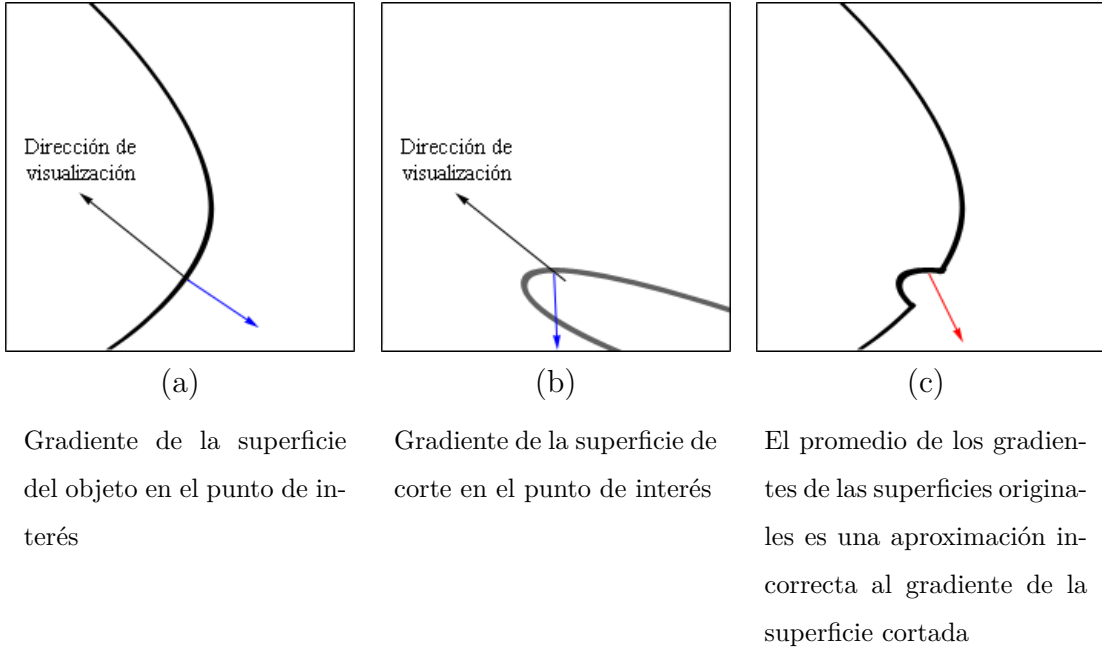


Figura 4.9: Estimación del gradiente como promedio de los gradientes individuales

Se puede notar que el gradiente correcto en la Figura 4.9 (c) es el mismo gradiente de la superficie de corte de la Figura 4.9 (b), por lo tanto, una alternativa sería utilizar el gradiente de la superficie que detuvo el rayo de visualización. La formula analítica (4.5) de esta propuesta sugiere que es necesario conocer el gradiente de la superficie del objeto y de la superficie de corte. Sin embargo, debido a que se conoce cual superficie se desea sombrear antes de calcular el gradiente, sólo el gradiente que realmente será utilizado es estimado, lo cual reduce un poco el costo computacional.

$$\Delta V'(p) = \begin{cases} \Delta V(p), & \text{si la superficie a sombrear es el objeto} \\ \Delta C(p), & \text{si la superficie a sombrear es el corte} \end{cases} \quad (4.5)$$

El problema de utilizar el gradiente de la superficie encontrada, como lo muestra la Figura 4.10, es el cambio brusco en el gradiente en la zona de la intersección de

ambas superficies. De forma concreta, cerca del punto de la intersección cualquiera de los dos vectores gradientes es inapropiado para la superficie cortada.

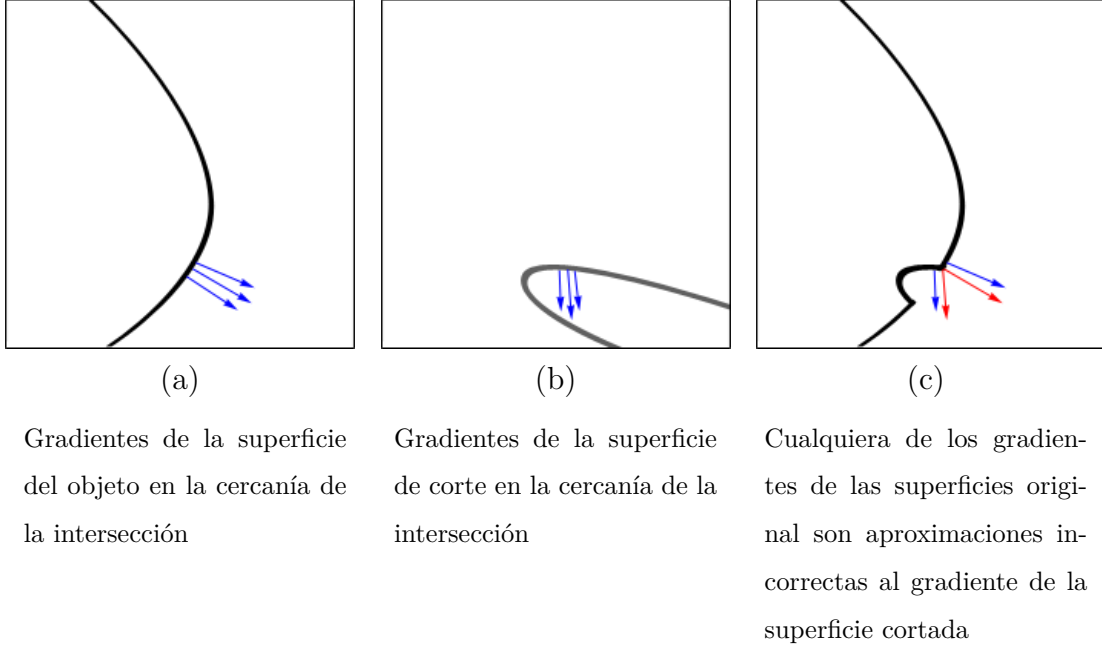


Figura 4.10: Utilización del gradiente de la superficie a sombrear

Tomando en cuenta las limitaciones de los métodos propuestos, en los que se utilizan los vectores gradientes de las superficies individuales para estimar el vector gradiente de la superficie resultante, se plantean enfoques basados en el método de diferencias finitas para aproximar el gradiente de la superficie resultante directamente. En lo que sigue se asume que se desea obtener una aproximación al componente x del gradiente por medio de diferencia central, es decir, se desea obtener una función $f : \mathbb{E}^3 \rightarrow \mathbb{R}$ tal que,

$$\frac{\delta V'(x, y, z)}{\delta x} \approx \frac{f(x+1, y, z) - f(x-1, y, z)}{2D_x}. \quad (4.6)$$

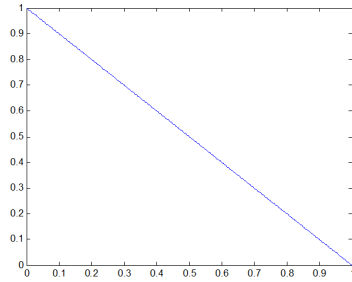
Comparando (4.6) con (2.6), es evidente que si $f(p) = V'(p)$, $\forall p \in \mathbb{E}^3$, el estimado del vector gradiente posee la misma calidad que el obtenido antes de realizar ningún corte. Sin embargo, el valor de $V'(p)$ es desconocido en la frontera de las superficies de corte y ha de ser estimado. Tomando en cuenta que el proceso de voxelización genera

niveles de grises que representan el volumen del vóxel que se encuentra dentro del corte, se decidió probar funciones que disminuyen el valor $V(p)$ basados en $C(p)$. La fórmula (4.7) utiliza una función lineal con respecto al nivel de gris del corte para disminuir la densidad del volumen original, la curva de la caída se muestra en la Figura 4.11 (a).

$$f(p) = V(p) \times (1 - C(p)) \quad (4.7)$$

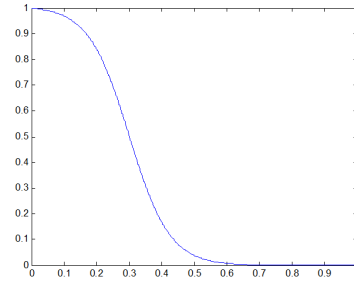
Con la intención de obtener una caída fuerte cerca del valor de la isosuperficie de corte, se utilizó una función sigmoide (4.8). Las funciones sigmoides, también conocidas como funciones $-S$ por su forma, permiten concentrar la mayor parte de la caída en un rango del intervalo. La Figura 4.11 (b) muestra la gráfica asociada a la función sigmoide mostrada en (4.8).

$$f(p) = V(p) \times \frac{1}{1 - 5^{-3} \times (1 - 5^{10 \times C(p)})} \quad (4.8)$$



(a)

Función lineal



(b)

Función sigmoide

Figura 4.11: Funciones utilizadas para disminuir la densidad del volumen

Por último, se propone una fórmula simple para la función f , cuyo objetivo es utilizar el promedio del valor de densidad del volumen original y el nivel de gris del volumen de corte, siempre y cuando este no sea menor a la densidad original. La función puede verse en forma analítica en (4.9), su efecto es aumentar la dirección del

gradiente en el sentido de la superficie de corte, conservando cierta información del volumen original.

$$f(p) = \max \left(V(p), \frac{V(p) + C(p)}{2} \right) \quad (4.9)$$

Para facilitar las pruebas y la posibilidad de agregar nuevas estimaciones al vector gradiente, el *kernel* de CUDA que implementa la emisión de rayos iluminada para el volumen médico recibe un objeto por parámetro que representa la función de estimación del gradiente. Dicho objeto debe implementar una interfaz sencilla que le permite al método de despliegue ignorar los detalles de la estimación del gradiente. De este modo, agregar nuevos métodos para estimar el gradiente se realiza sin modificaciones al algoritmo de despliegue volumétrico, únicamente es necesario crear una clase que implemente la interfaz mencionada y luego utilizar un objeto de ese tipo en la llamada al *kernel*.

4.3. Detección y Manipulación de Fragmentos Separados

Uno de los objetivos del trabajo era detectar fragmentos de hueso separados, los cuales podrían originalmente ser partes de un mismo hueso que fueron separados completamente por el instrumento de corte. Luego, algunos de estos fragmentos podrían eliminarse o manipularse independientemente, sin embargo, la representación y algoritmos utilizados para la simulación de cortes de huesos imponen ciertas restricciones en la capacidad de representar fragmentos independientes. Principalmente, ya que existe un único volumen para los datos médicos, si se desea trasladar o rotar un fragmento independiente la resolución se vería limitada por el volumen; más aún, dichas transformaciones generarían pérdidas de información. Por lo tanto, en este trabajo se tomó la decisión de no incorporar la capacidad de realizar transformaciones independientes a los fragmentos.

4.3.1. Abstracciones Lógicas

Para poder determinar los fragmentos de hueso separados se definió de manera lógica un grafo $G = \langle V, E \rangle$ sobre los datos médicos, de manera que, cada centro de vóxel (v_x, v_y, v_z) tiene un vértice $v \in V$ en el grafo siempre y cuando el valor de densidad asociado al vóxel sea mayor que la isosuperficie. Luego, para cada par de vértices $u, v \in V$ tales que

$$\max(|u_x - v_x|, |u_y - v_y|, |u_z - v_z|) = 1, \quad (4.10)$$

se agrega el arco no dirigido e_{uv} . La condición mostrada en (4.10), como muestra la Figura 4.12, implica que un vértice tiene un máximo de 26 vecinos.

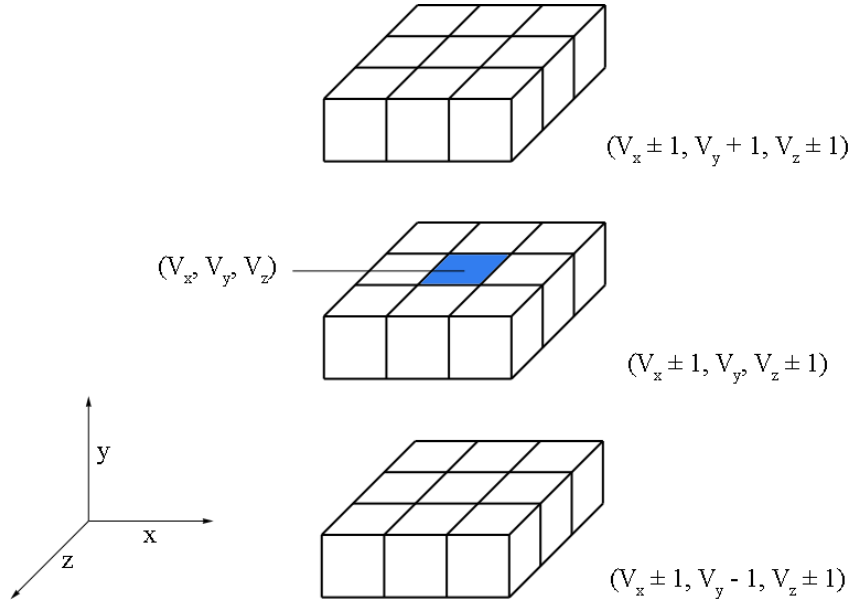


Figura 4.12: Un vóxel y sus 26 vecinos en el grafo.

Los diferentes fragmentos de hueso se definen como las componentes conexas sobre dicho grafo como se ilustra en la Figura 4.13. Sin embargo, este enfoque está limitado por la resolución del volumen y trae problemas con los vóxeles parcialmente rellenos. Cuando el fragmento es eliminado no hay problema, ya que la isosuperficie es eliminada y no se generan artefactos visuales. El problema ocurre cuando se le asignan colores

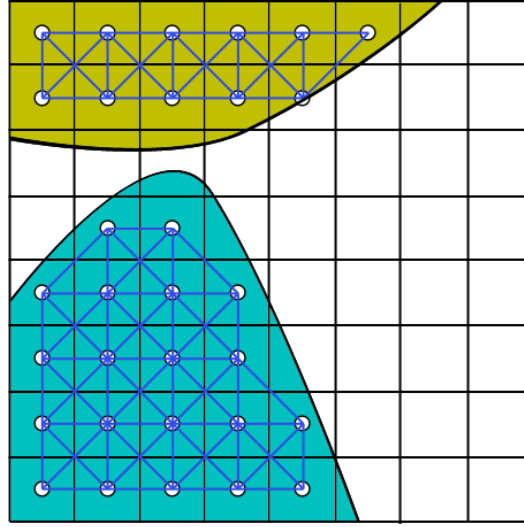
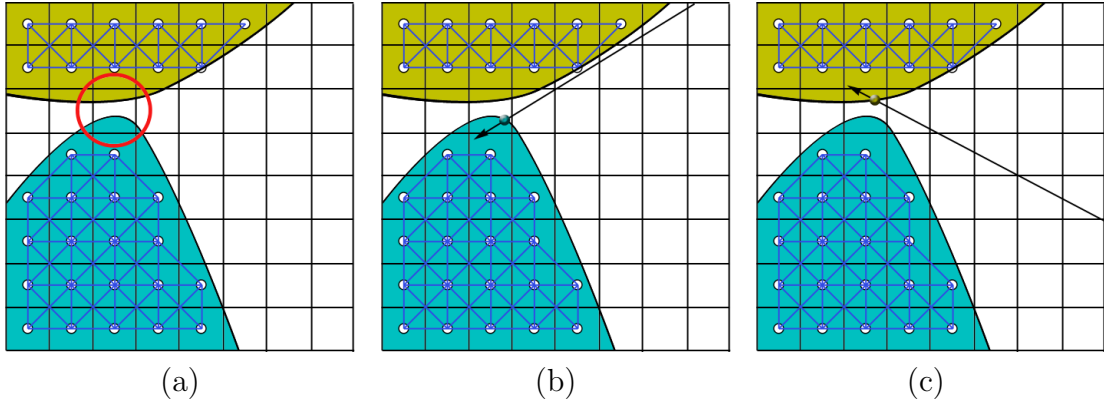


Figura 4.13: Diferentes componentes conexas representan diferentes fragmentos de hueso

diferentes a los distintos fragmentos de hueso, esto ocurre cuando un rayo toca la isosuperficie en un vóxel que no fue asignado al fragmento. Como se muestra en la Figura 4.14 reglas simples como utilizar el color del fragmento más cercano o el que contenga mayor volumen dentro del vóxel producen artefactos visuales, ya que, un mismo vóxel puede requerir dos colores diferentes dependiendo del rayo. La única forma de eliminar por completo los artefactos es extender el modelo para permitir un número arbitrario de volúmenes médicos y utilizar el método de muestreo adaptativo mostrado en [21].

4.3.2. Detección de Componentes Conexas

Para obtener las componentes conexas se implementó un algoritmo de búsqueda en anchura en grafos (BFS). Dado un grafo $\langle G, V \rangle$ y un nodo inicial $s \in V$, el algoritmo de búsqueda en anchura, mostrado en el Algoritmo 4.2, explora sistemáticamente los arcos de G para “descubrir” todos los vértices alcanzables desde s . El algoritmo recibe su nombre porque el proceso expande la frontera entre nodos descubiertos y nodos sin descubrir a lo ancho de la misma. Lo cual significa, que el algoritmo descubre todos los vértices con distancia k desde s antes de descubrir ningún vértice a distancia $k + 1$.



(a) El círculo señala un vóxel problemático.

(b) El rayo debe tomar el color del fragmento inferior.

(c) El rayo debe tomar el color del fragmento superior.

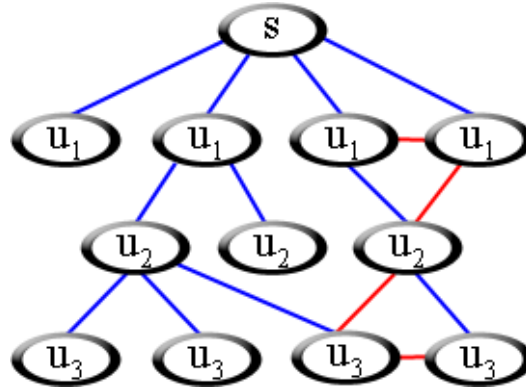
Figura 4.14: Problemas en el enfoque de las componentes conexas.

Este proceso, ilustrado en la Figura 4.15, genera un árbol cuya raíz es s y el camino desde la raíz a cada nodo u_i del árbol representa un camino de distancia mínima (i) de s a u_i en el grafo original [5].

-
1. Sea $G = \langle V, E \rangle$ el grafo de búsqueda.
 2. Sea $s \in V$ el nodo origen.
 3. Sea $W = \{s\}$ un conjunto de nodos visitados.
 4. Sea $Q = \{s\}$ una cola de nodos a explorar.
 5. Mientras Q no es vacía, hacer:
 6. | Desencolar u de Q , donde u es el primer nodo en la cola.
 7. | Para todo nodo $v \in V$, tal que $(u, v) \in E$ y $v \notin W$, hacer:
 8. | | Actualizar $W = W \cup \{v\}$.
 9. | | Encolar v en Q .
-

Algoritmo 4.2: Búsqueda en Anchura (BFS)

Ya que todos los nodos que pertenecen a la misma componente conexa del nodo origen son visitados durante el recorrido del algoritmo, es posible modificarlo para encontrar todas las componentes conexas. Para ello, simplemente se realiza una búsqueda desde cualquier nodo que no ha sido visitado antes, durante ésta búsqueda, todos los



Los arcos azules son agregados al árbol, mientras que los arcos rojos existen en el grafo pero no en el árbol.

Figura 4.15: Árbol de búsqueda en anchura

los nodos visitados son agregados a la misma componente conexa. El Algoritmo 4.3 muestra el algoritmo modificado para detectar todas las componentes conexas.

-
1. Sea $G = \langle V, E \rangle$ el grafo de búsqueda.
 2. Sea $W = \emptyset$ un conjunto de nodos visitados.
 3. Mientras $W \neq V$, hacer:
 4. | Sea $s \in V$ cualquier nodo tal que $s \notin W$.
 5. | Crear $C_s = \{s\}$ la componente conexa que contiene a s .
 6. | Actualizar $W = W \cup \{s\}$.
 7. | Sea $Q = \{s\}$ una cola de nodos a explorar.
 8. | Mientras Q no es vacía, hacer:
 9. | | Desencolar u de Q , donde u es el primer nodo en la cola.
 10. | | Para todo nodo $v \in V$, tal que $(u, v) \in E$ y $v \notin W$, hacer:
 11. | | | Actualizar $C_s = C_s \cup \{v\}$.
 12. | | | Actualizar $W = W \cup \{v\}$.
 13. | | | Encolar v en Q .
-

Algoritmo 4.3: Detección de Componentes Conexas Utilizando Búsqueda en Anchura

Capítulo 5

Análisis de Resultados de Pruebas

El modelo propuesto se sometió a una serie de pruebas para obtener resultados cuantitativos y cualitativos. Los resultados cuantitativos se enfocan en los tiempos de ejecución requeridos por los diferentes algoritmos del sistema, de esta forma se espera mostrar la capacidad del sistema de ofrecer simulaciones de alta calidad en tiempo real. Por otra parte, los resultados cualitativos se concentran en mostrar la calidad del resultado visual del modelo propuesto, comparándolos con los resultados poco satisfactorios de los métodos tradicionales que mantienen una separación lógica entre la representación de los cortes del algoritmo de despliegue volumétrico.

5.1. Resultados Cuantitativos

Con la intención de comprobar la capacidad del sistema para la simulación interactiva en tiempo real, se realizaron diferentes pruebas de tiempo utilizando tres computadoras con capacidades diferentes. Los aspectos más relevantes de cada plataforma de prueba se encuentran en la Tabla 5.1. Es importante mencionar que no se realizó ninguna modificación al código fuente al cambiar de plataforma, sin embargo, el mismo fue compilado especialmente para cada plataforma.

GPU	Núcleos de CUDA	Versión de CUDA soportada	CPU
GeForce GTX 285M	128	1.1	Core i7 Q740M
GeForce 8800 GTS	96	1.0	Core 2 Duo E6550
GeForce GTX 470	448	2.0	Core i3 540

Tabla 5.1: Detalles de las plataformas de prueba.

5.1.1. Sistema General

El primer estudio de interés es determinar que partes del sistema consumen la mayor cantidad de tiempo, el cual se puede realizar analizando la Figura 5.1 que muestra el porcentaje que cada etapa consume del tiempo total durante una ejecución común del sistema. Como se puede observar, más del 90 % del tiempo se está ejecutando el algoritmo de despliegue volumétrico, por lo que esta etapa domina por completo el tiempo de respuesta del sistema. Sin embargo, al ver la Figura 5.2 que muestra el tiempo en milisegundos que toma la ejecución de cada etapa en promedio, se nota inmediatamente que el manejo (detección, remoción y separación) de fragmentos de hueso separados es la etapa más lenta del sistema. Es importante tener en cuenta que esta es la única etapa del sistema que es ejecutada completamente en el CPU por la naturaleza serial inherente del algoritmo de búsqueda en anchura.

El motivo por el cual el manejo de fragmentos separados, a pesar de ser la etapa más lenta, no domina el tiempo total de ejecución es la frecuencia relativa con la que son ejecutadas ambas etapas. El sistema debe ejecutar el algoritmo de despliegue después de cualquier acción que pueda modificar el estado de los datos, es decir, después de cada una de las otras etapas. Por otra parte, el algoritmo de manejo de fragmentos es invocado voluntariamente por el usuario cuando se considera necesario, lo que ocurre con una frecuencia muy baja.

Debido a que el tiempo de ejecución del algoritmo de manejo de fragmentos separa-

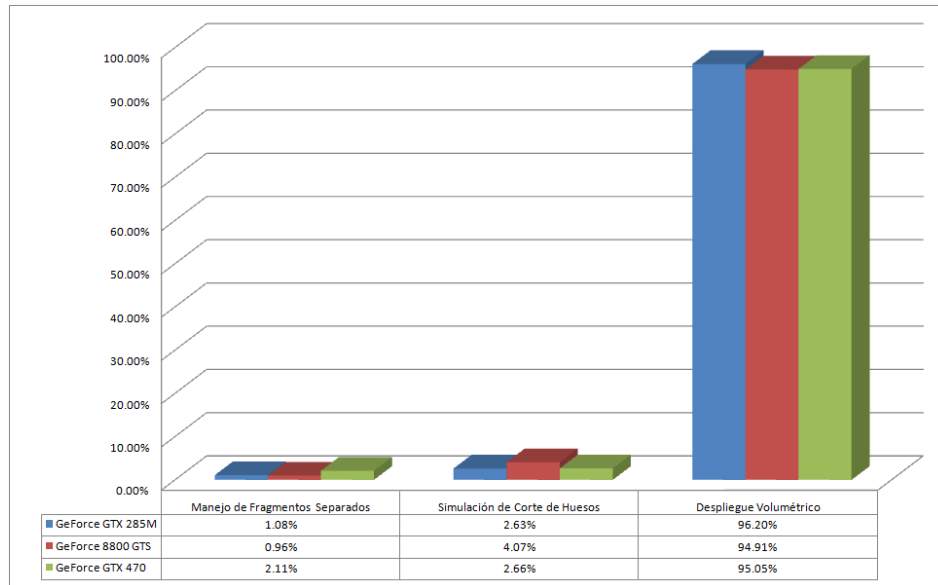


Figura 5.1: Porcentaje del tiempo que el sistema emplea en cada etapa durante una ejecución típica.

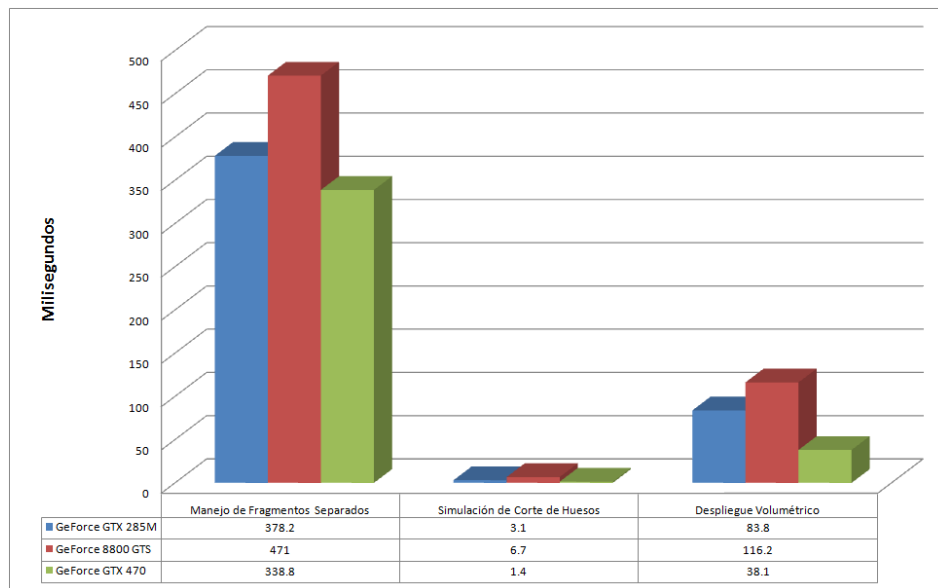


Figura 5.2: Tiempo en milisegundos que toma la ejecución de cada etapa en promedio.

dos es muy elevado para llevarse a cabo en cada cuadro, se debe analizar la posibilidad de mantener una tasa de cuadros por segundo aceptable considerando únicamente los otros aspectos del sistema. Al analizar la Figura 5.3, la cual muestra la tasa de cuadros por segundo con y sin voxelización, se observa que el efecto de voxelizar el instrumento de corte en cada cuadro es despreciable. Por lo tanto, el interés se reduce a analizar

el tiempo de ejecución del algoritmo de despliegue, además, se observa el efecto que tiene la tarjeta gráfica sobre la velocidad de ejecución del algoritmo.

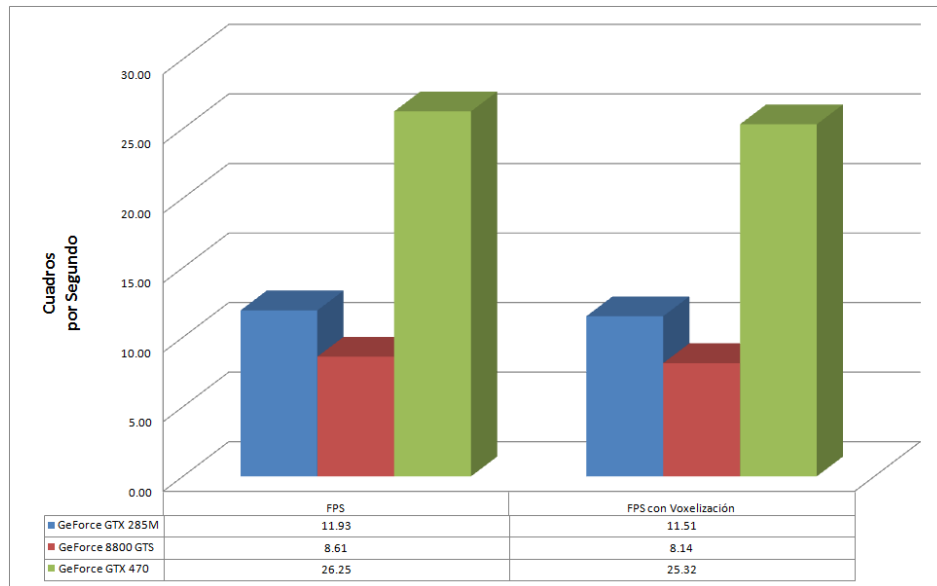


Figura 5.3: Cuadros por segundo.

A pesar de la desalentadora tasa de 8 cuadros por segundo ofrecida por la GeForce 8800 GTS, la tasa de 25 cuadros por segundo que ofrece la GeForce GTX 470 es más que aceptable en un ambiente de tiempo real. La tasa de 11 cuadros por segundo que obtiene la GeForce GTX 285M es relativamente baja, pero si se toma en cuenta que esta es una tarjeta gráfica para computadoras portátiles, el resultado es esperanzador y podría significar que en el futuro cercano las computadoras portátiles serán suficientes para realizar simulaciones de alta calidad en tiempo real.

5.1.2. Despliegue Volumétrico

Los resultados anteriores sugieren que la mayor ganancia en tiempo de respuesta del sistema sería generada si se optimiza el algoritmo de despliegue, por lo tanto, es preciso analizar con mayor detalle el efecto que tiene el tamaño de la imagen y las subetapas que componen el método. Para determinar los tamaños de imágenes

que el sistema puede manejar en tiempo real, se analizan las Figuras 5.4 y 5.5 que muestran, respectivamente, el aumento en el tiempo requerido y la caída de la tasa de cuadros por segundo cuando el tamaño de la imagen crece. Dichas imágenes sugieren que el crecimiento en el tiempo requerido por el algoritmo de despliegue es, en general, aproximadamente lineal con respecto al tamaño de la imagen de salida para todas las plataformas probadas. Sin embargo, es importante mencionar que tres muestras no son suficientes para obtener un resultado concluyente y certero sobre el orden de crecimiento del algoritmo.

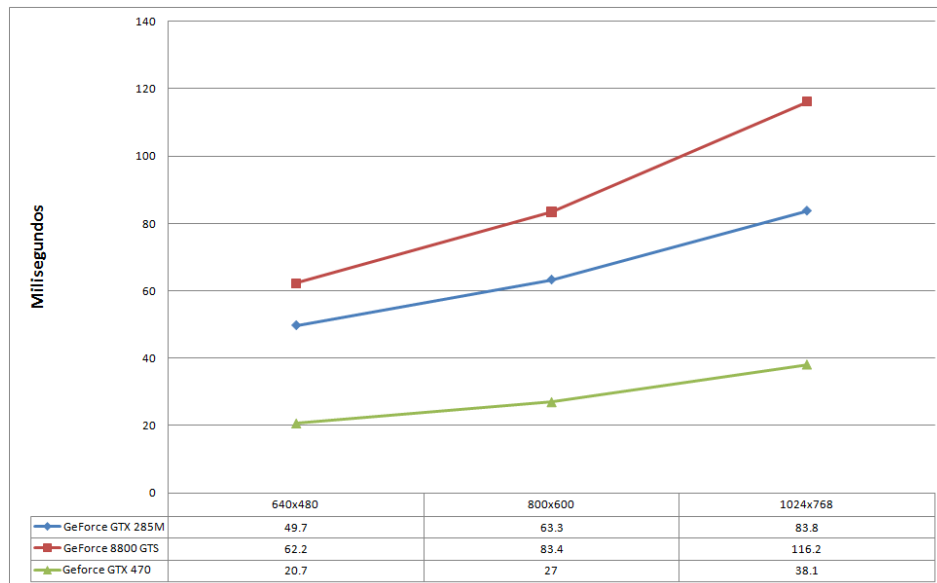


Figura 5.4: Crecimiento del tiempo de despliegue al aumentar el tamaño de la imagen.

Observando la Figura 5.6, la cual muestra los cuadros por segundo del algoritmo de emisión de rayos con y sin iluminación, se nota que todas las tarjetas son capaces de mantener una tasa adecuada sin iluminación. La tasa de cuadros por segundo cae dramáticamente cuando el fragmento es sombreado con un modelo de iluminación. Lamentablemente las restricciones de CUDA no permiten realizar medidas de tiempo acertadas a partes de un *kernel*, pero se sospecha que esta diferencia es dominada completamente por la determinación del gradiente. A diferencia de los modelos estáticos, el gradiente cambia constantemente y no puede ser calculado de antemano. Sin

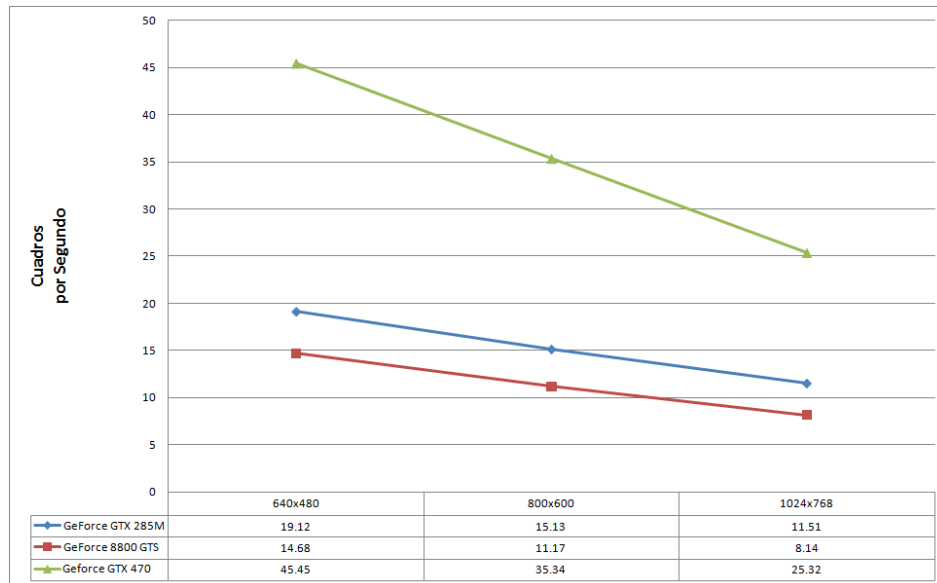


Figura 5.5: Disminución de los cuadros por segundo al aumentar el tamaño de la imagen.

embargo, una observación importante es que el gradiente cambia únicamente cerca de la frontera de la superficie de cada corte nuevo, por lo tanto, una posibilidad para mejorar el tiempo de respuesta sería mantener una textura de gradientes que es actualizada durante la voxelización. De esta forma la velocidad del algoritmo de despliegue se incrementaría sustancialmente, pero el impacto que tendría sobre el tiempo de voxelización no es trivial de estimar y debe ser medido apropiadamente.

Otro aspecto muy importante es determinar cuan efectivo es el algoritmo, es decir, que tanto del tiempo total es en efecto utilizado por el despliegue y no preparando los datos. Luego de realizar las mediciones apropiadas, se determinó que la única sección de preparación que tiene un impacto notable es la copia de datos en memoria de dispositivo. Esta copia de datos es necesaria por una limitación actual de CUDA, que no permite asociar memoria modificable a unidades de texturas tridimensionales. Por lo tanto, es necesario mantener la superficie de corte y el volumen del instrumento en secciones de memoria modificable y realizar copias a los arreglos estáticos de CUDA antes del despliegue. En la Figura 5.7 se puede observar el tiempo promedio que toma la ejecución de los *kernels* de despliegue y la copia de los datos dinámicos, lo cual

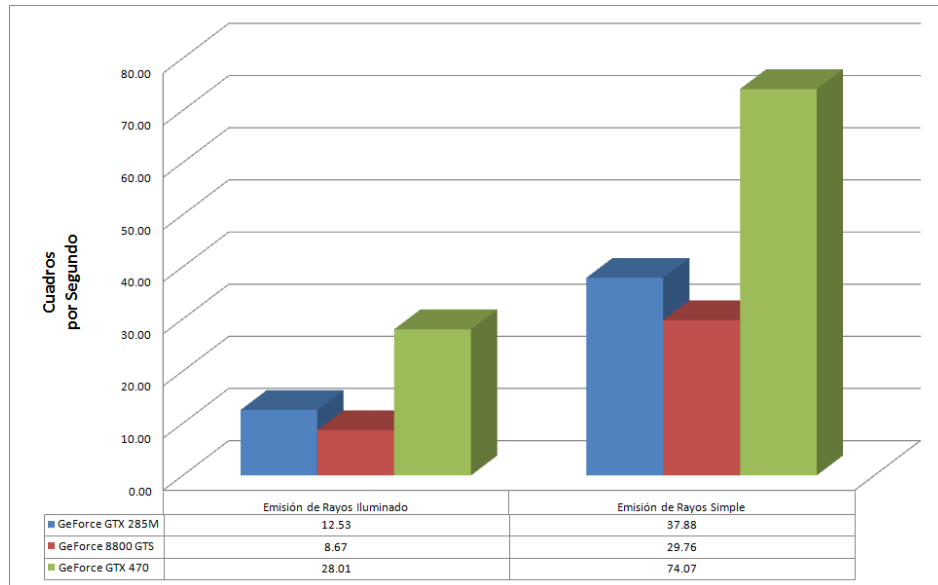


Figura 5.6: Cuadros por segundo con y sin iluminación.

indica que hay un desperdicio significativo al realizar dichas copias.

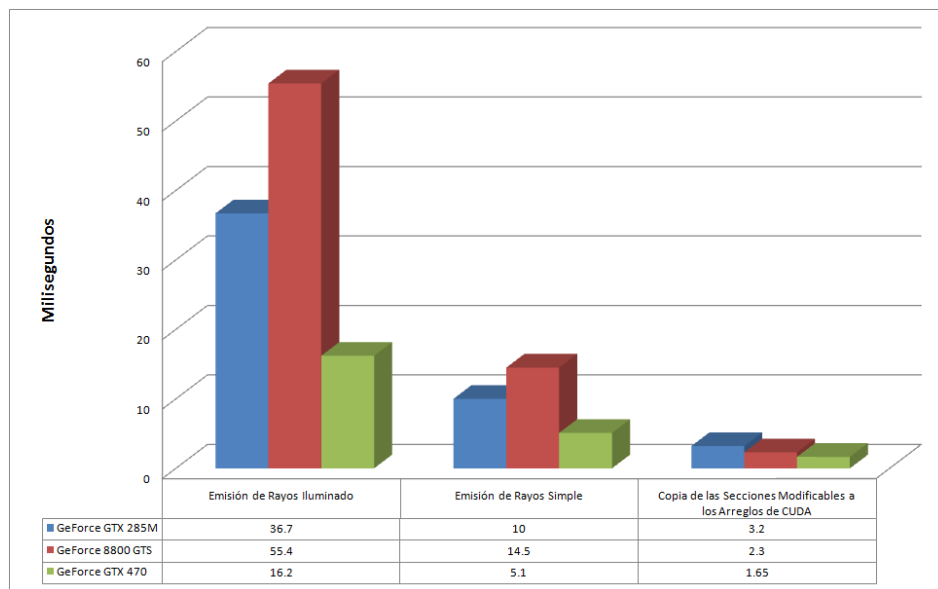


Figura 5.7: Tiempo en milisegundos de los diferentes componentes del despliegue volumétrico.

Para concretar el desperdicio generado por la copia de datos en memoria, se estudió el porcentaje del tiempo de ejecución que ésta utiliza. Dicho desperdicio se encuentra ilustrado en la Figura 5.8, donde se nota que el desperdicio es un poco

menos del 10 % para la emisión de rayos con iluminación y aproximadamente del 24 % para la emisión de rayos sin iluminación para las tarjetas más potentes. Cuando se utiliza la tarjeta menos eficiente, GeForce 8800 GTS, el desperdicio es mucho menor, sin embargo esto se debe principalmente a la lenta ejecución del *kernel* en dicha tarjeta. Estos resultados implican que una ganancia considerable se obtendrá gratuitamente cuando la arquitectura CUDA permita asociar memoria modificable a texturas tridimensionales. Sin embargo, también sugieren que una investigación razonable involucra utilizar capas de texturas bidimensionales para simular una textura tridimensional, ya que estas sí pueden ser asociadas a memoria modificable.

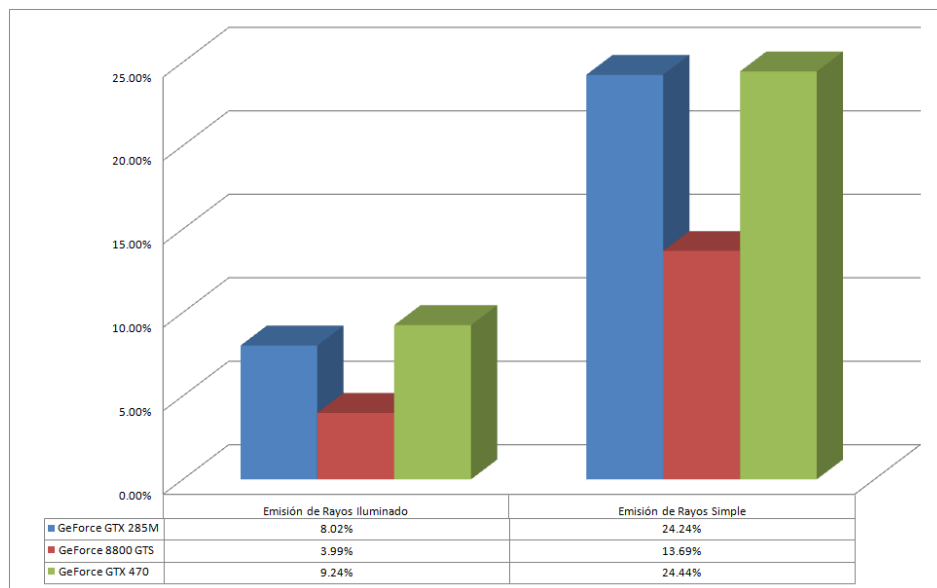


Figura 5.8: Porcentaje de tiempo desperdiciado moviendo datos en memoria de dispositivo.

5.1.3. Manipulación de Fragmentos Separados

El otro punto donde se puede obtener una ganancia considerable en tiempo de ejecución, es la manipulación de fragmentos separados. Para analizar las posibilidades se midió el tiempo que toma cada una de los componentes que lo conforman. La Figura 5.9 muestra que prácticamente todo el tiempo es empleado durante la búsqueda de fragmentos separados, es decir, ejecutando el algoritmo de búsqueda en anchura.

Sin embargo, el algoritmo se implementó con el deseo de ofrecer el mejor rendimiento posible y se sospecha que la única forma de generar una ganancia sustancial es utilizar un enfoque completamente diferente al propuesto.

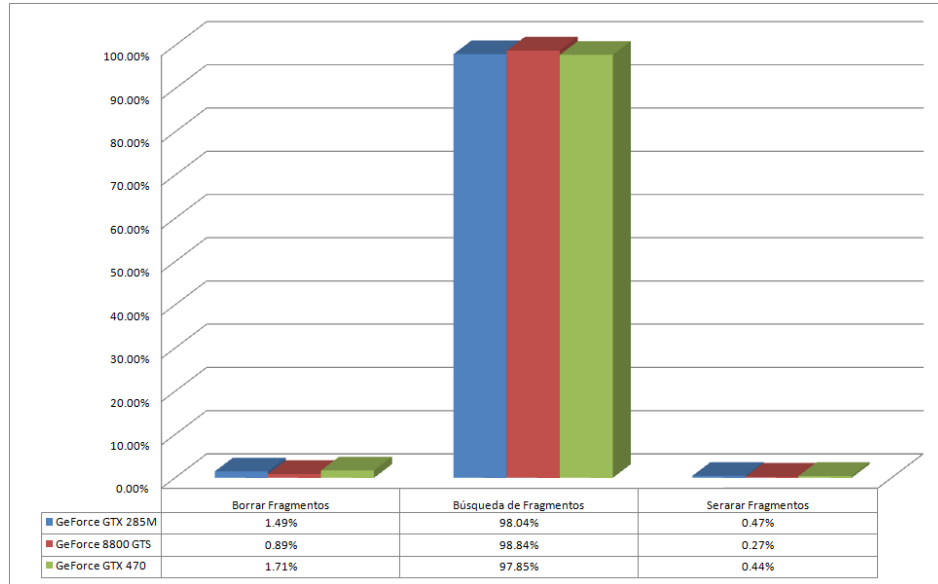


Figura 5.9: Tiempo en milisegundos de cada componente de la manipulación de fragmentos.

5.2. Resultados Cualitativos

Las pruebas cualitativas realizadas se pueden dividir en dos categorías. La primera consiste en comparar las imágenes producidas por el modelo propuesto con una técnica tradicional, mientras que la segunda analiza las diferentes propuestas realizadas para la estimación del gradiente en las fronteras de la superficie de corte.

En todos los casos se utilizó la capacidad de la biblioteca de simulación para almacenar un secuencia de pasos a ejecutar, de esta manera se garantiza que tanto la distancia y el ángulo de visión son exactamente iguales.

5.2.1. Comparación con las Técnicas Tradicionales

Para mostrar que el modelo de cortes de sólidos volumétricos propuesto provee una calidad visual superior a las técnicas tradicionales, es necesario realizar comparaciones sobre las imágenes resultantes en ambos casos. El modelo tradicional consiste en mezclar las superficies de corte directamente sobre el volumen original, mientras que el modelo propuesto mantiene las superficies de corte en un volumen separado y realiza la mezcla durante el algoritmo de emisión de rayos.

Se comienza analizando la Figura 5.10, la cual muestra dos imágenes generadas sin aplicar iluminación. Como se puede ver en la Figura 5.10 (a) el resultado de realizar la mezcla en el volumen del corte en el volumen original genera imágenes melladas, esto se debe a que la calidad del corte se encuentra limitada por la resolución del volumen. Por otra parte, la Figura 5.10 (b) muestra que realizar la mezcla durante la emisión de rayos produce cortes mucho más suaves. Es importante mencionar, que los pequeños puntos de color azul en la Figura 5.10 (a) no son artefactos generados por el algoritmo de mezcla, más aún, los mismos seguían apareciendo al utilizar el modelo propuesto. Lamentablemente, la solución a dicho problema se desarrolló después de cambiar al modelo actual, por lo tanto, no existe ninguna versión del programa que permita realizar la mezcla tradicional sin dichos puntos.

Al observar la Figura 5.11, en donde las mismas imágenes (con un acercamiento menor) son generadas aplicando el modelo de iluminación, el problema del mellado se hace aún más evidente. Este fenómeno es natural y se debe a que la pérdida de información provoca una mala estimación del gradiente. En cambio, la Figura 5.11 (b) sugiere que el método actual es capaz de generar una estimación mucho más acertada del vector gradiente.

En las imágenes anteriores, el instrumento de corte estaba compuesto únicamente por ortoedros. Sin embargo, la biblioteca de simulación permite utilizar esferas para la descripción de los instrumentos de corte. Las esferas son objetos muy simples desde

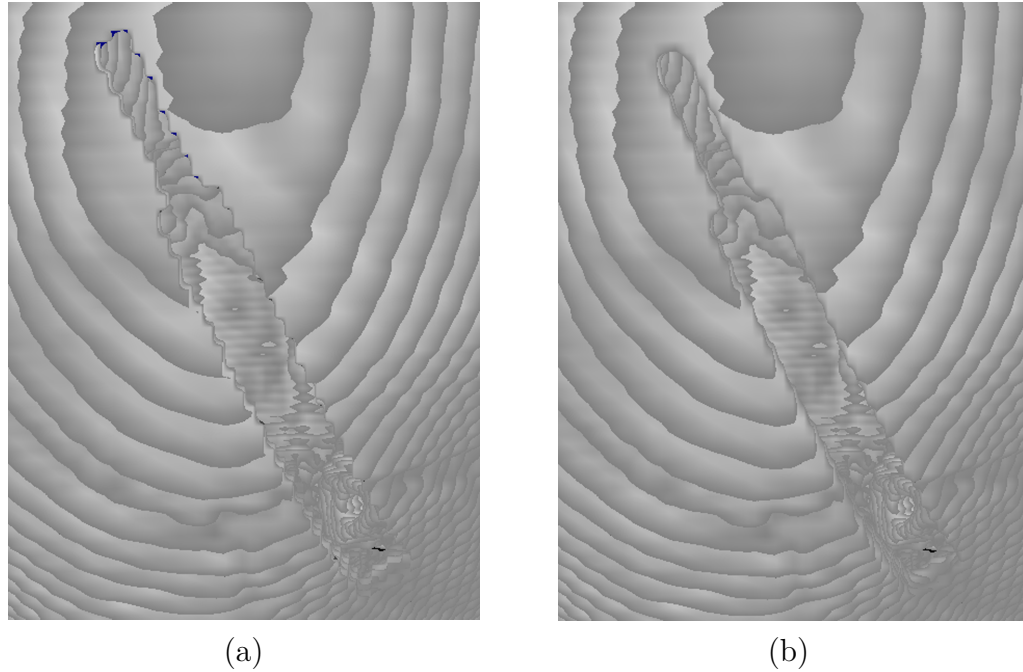
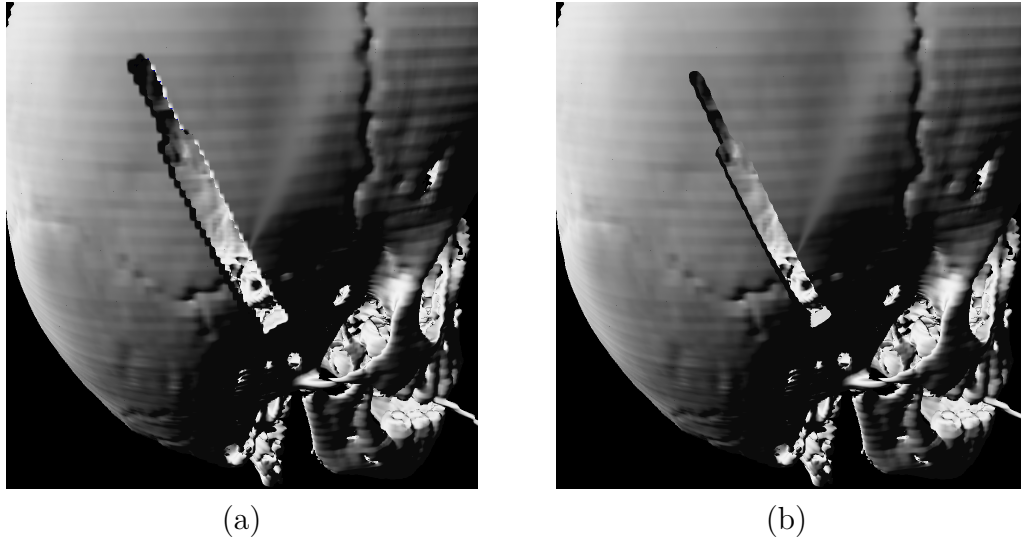


Figura 5.10: Diferencia en la calidad del corte sin iluminación

un punto de vista geométrico, pero son notoriamente difíciles de modelar en ambientes discretos. La Figura 5.12 muestra las imágenes generadas por un instrumento de corte compuesto por una única esfera. Como se puede observar en la Figura 5.12 (a) el mellado generado por la mezcla discreta ofrece la calidad típica de una imagen de baja resolución, mientras que la Figura 5.12 (b) muestra un resultado mucho más suave y natural.

5.2.2. Comparación de las Diferentes Propuestas para Estimar el Vector Gradiente

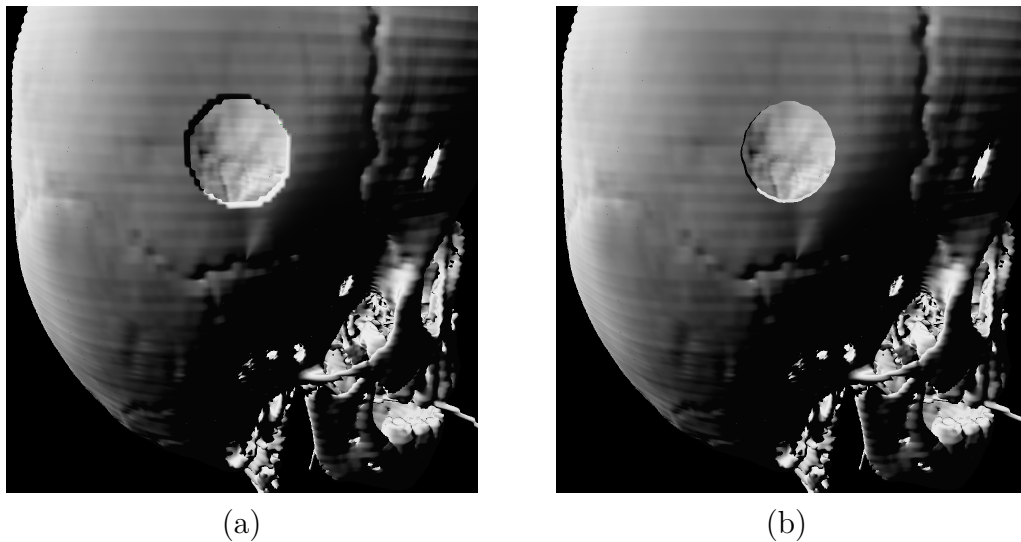
En la Sección 4.2.3 se muestran una serie de alternativas para estimar el vector gradiente en la frontera de las superficies de corte, sin embargo, se señala que todas las aproximaciones propuestas sufren de una u otra dificultad para realizar una estimación apropiada en todas las situaciones. Por lo tanto, es necesario generar imágenes



Mezcla de resultados de corte en el volumen original

Superficies de corte independiente es mezclada durante la emisión de rayos

Figura 5.11: Diferencia en la calidad del corte con iluminación



Mezcla de resultados de corte en el volumen original

Superficies de corte independiente es mezclada durante la emisión de rayos

Figura 5.12: Diferencia en la calidad del corte al voxelizar esferas

utilizando las diferentes alternativas, para estudiar objetivamente la calidad visual de las mismas. El análisis comienza observando la Figura 5.13 (a), la cual muestra

el resultado de la imagen cuando el estimado del gradiente ignora por completo las superficies de corte. Generar esta imagen es necesaria para mostrar la necesidad de una estimación del gradiente especializada para el modelo propuesto, lo cual se puede inferir de forma inmediata debido al poco realismo observado en la imagen.

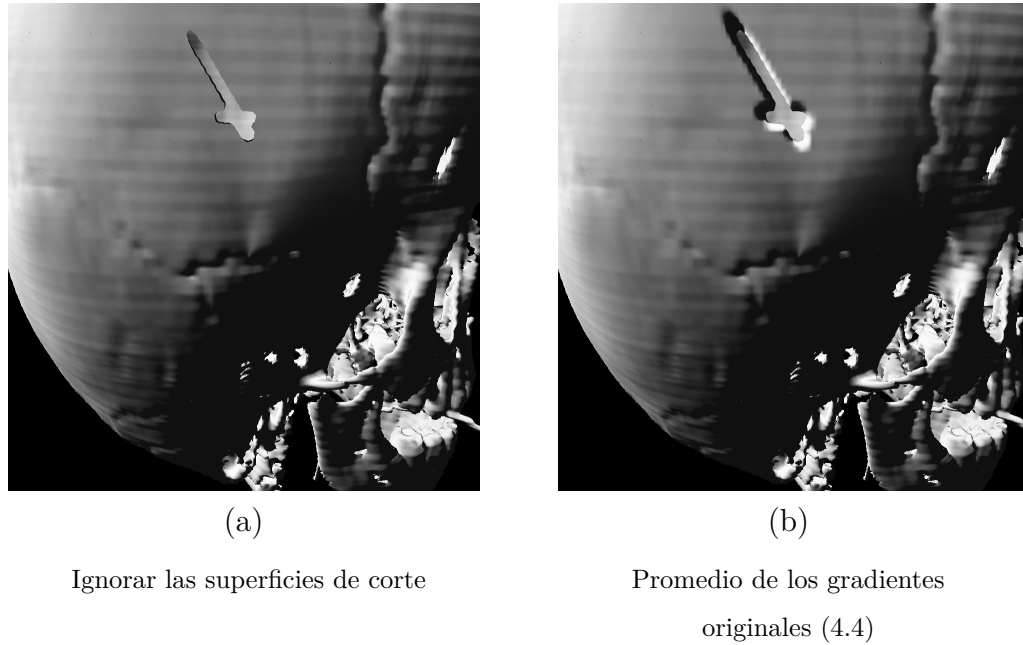
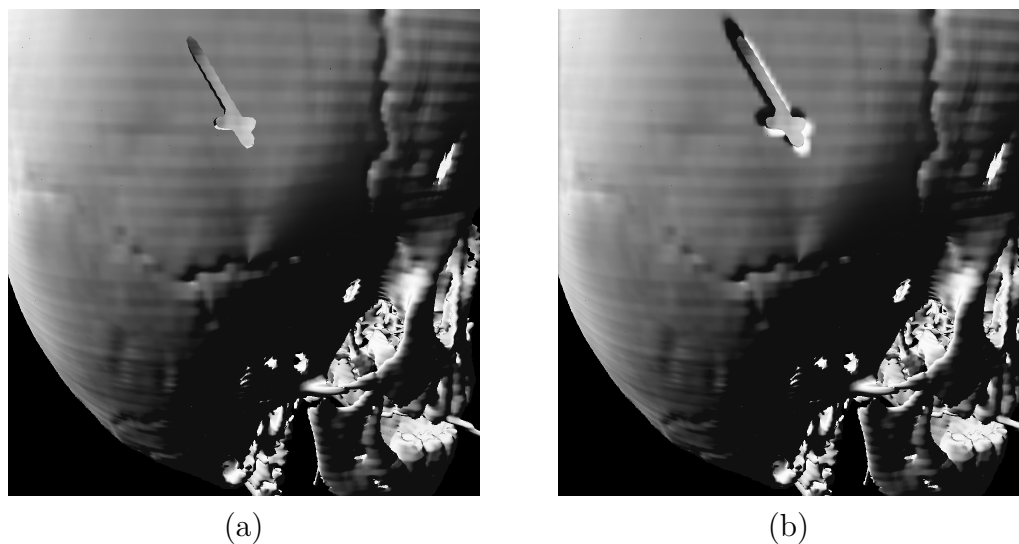


Figura 5.13: Diferentes estimados para el gradiente en las fronteras de las superficies de corte

La Figura 5.13 (b) muestra el resultado de utilizar el promedio de los gradientes originales de las superficies del hueso y de corte, calculados independiente el uno del otro, basado en (4.4). Como se puede observar, este método genera imágenes notablemente irreales. Por otra parte, la Figura 5.14 (a) utiliza la superficie encontrada por el rayo, como se propone en (4.5). La imagen generada es coherente con los resultados teóricos. El cambio en el gradiente utilizado en la intersección de las superficies genera un salto brusco y no permite modelar correctamente el gradiente en dicha zona. Sin embargo, la calidad de la imagen es altamente superior al promedio de los vectores gradientes independientes.

Las propuestas basadas en disminuir la densidad del volumen con una función que depende del nivel de gris del volumen de cortes tienen un fundamento teórico menos



Gradiente de la superficie a
sombrear (4.5)

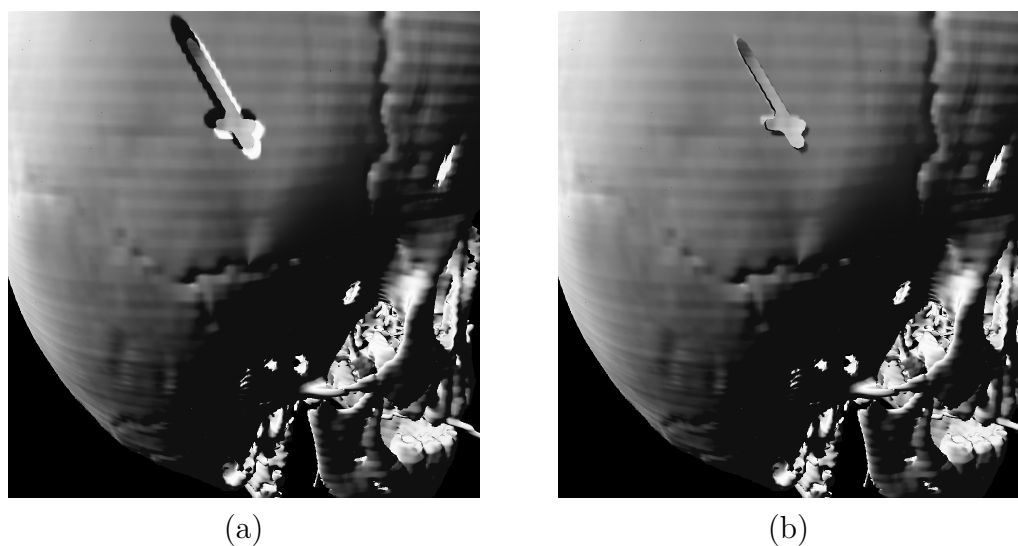
Disminución por función lineal (4.7)

Figura 5.14: Diferentes estimados para el gradiente en las fronteras de las superficies de corte

estricto, es decir, no se formalizaron razones matemáticas para garantizar que $f(p) \approx V'(p)$. Sin embargo, la Figura 5.14 (b) muestra que utilizar una función lineal para disminuir la densidad, como se muestra en (4.7), genera imágenes similares al método que promedia los gradientes individuales. Dicho resultado muestra que la estimación de la densidad V' no es apropiada, pero sugiere que existe la posibilidad de formular métodos basado en este enfoque.

Por otra parte, como lo muestra la Figura 5.15 (a), cuando se utiliza (4.8) para disminuir la densidad del volumen basado en una función sigmoide, la imagen generada es completamente irreal. Dicho resultado no es del todo inesperado, a pesar de lo interesante de las funciones sigmoides, el proceso de voxelización genera niveles de grises con un incremento lineal.

Finalmente, la Figura 5.15 (b) muestra el resultado de utilizar (4.9), esta imagen sugiere que el método es capaz de obtener resultados de calidad aceptable. Sin embargo, la Figura 5.16 compara la imagen creada por este método con una que utiliza el gradiente de la superficie intersectada (4.5) desde otro ángulo de visión. En este caso

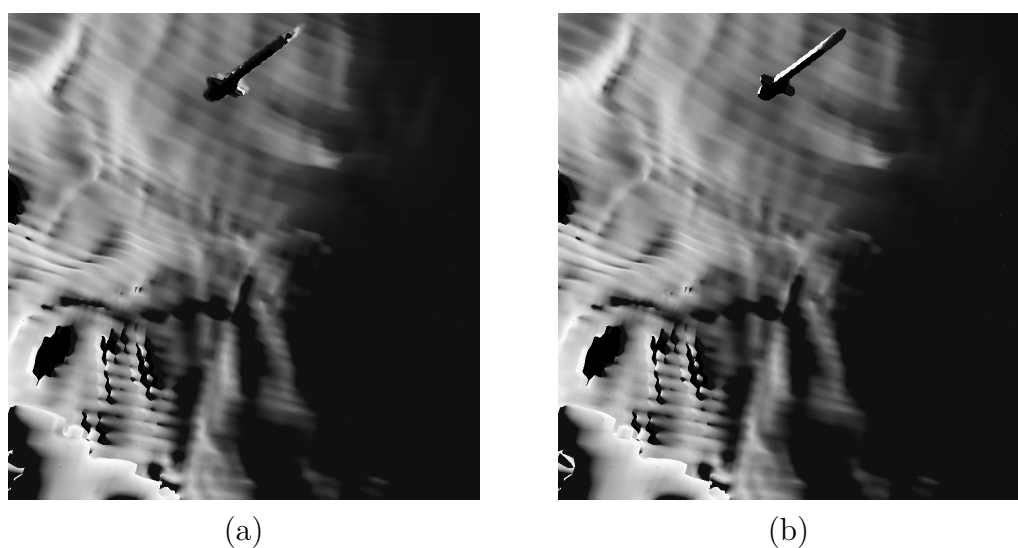


Disminución por función sigmoide (4.8)

Promedio de densidades (4.9)

Figura 5.15: Diferentes estimados para el gradiente en las fronteras de las superficies de corte

se puede observar que la calidad de la Figura 5.16 (a) es sumamente inferior a la Figura 5.16 (b).



Promedio de densidades (4.9)

Gradiente de la superficie a
sombrear (4.5)

Figura 5.16: Diferente ángulo para comparar estimados del gradiente en las fronteras de las superficies de corte

5.2.3. Calidad Visual de los Bordes Agudos

Finalmente, se debe estudiar la calidad de las imágenes producidas con respecto a los bordes agudos generados por la intersección de diferentes superficies de corte. Como se puede ver en las imágenes de la Sección 5.2.1, el modelo propuesto elimina muchos de los artefactos generados por el método tradicional. Sin embargo, los bordes son suavizados tanto que se pierde parte de la realidad, uno de estos bordes se encuentra resaltado en la Figura 5.17. El artefacto es producido por la regla simplificada para la combinación de resultados (4.3), el problema de dicha regla es que ignora por completo el estado de la superficie luego de aplicar varios cortes al volumen original.



Figura 5.17: El borde agudo se suaviza tanto que se pierde calidad

Conclusiones

En este trabajo se propone un modelo para la representación, manipulación y despliegue de cortes sobre sólidos volumétricos con instrumentos geométricos, con el objetivo de realizar simulaciones de corte de hueso de alta calidad en tiempo real. Para representar los cortes con resolución subvolumétrica se utiliza un modelo de múltiples volúmenes, en el que las superficies de cortes no son modeladas como atributos en el volumen original sino por un volumen separado. El despliegue volumétrico propuesto es una extensión del algoritmo de emisión de rayos basado en isosuperficies, el cual toma en cuenta la existencia de un volumen de superficies de corte para determinar la superficie que se debe sombrear y la estimación del gradiente sin limitarse a la resolución del volumen. Además, se presentó un método para determinar la separación de fragmentos de hueso basado en el algoritmo de búsqueda en anchura para la detección de componentes conexas.

El modelo propuesto mostró una tasa de cuadros por segundos aceptable para ambientes de tiempo real, al menos en las tarjetas gráficas más modernas, tanto para la simulación de cortes como para el despliegue volumétrico. Sin embargo, el algoritmo propuesto para la detección de fragmentos separados no puede ser ejecutado en tiempo real, para remediar esta situación el sistema se diseñó de forma que el mismo sea invocado por el usuario únicamente cuando este así lo desee y considere necesario.

La utilización de la arquitectura paralela CUDA permitió la implementación paralela y de alta intensidad aritmética requerida por la voxelización del instrumento de corte y el despliegue volumétrico de forma eficiente y sencilla. Desafortunadamente, no se tienen versiones seriales de los algoritmos para comprobar una ganancia en

rendimiento, sin embargo, se sospecha que la explotación del paralelismo inherente en dichos algoritmos ha sido beneficioso. Por otra parte, no se alega tener la implementación más eficiente de los algoritmos en dicha plataforma, ya que el proceso de desarrollo se encontró superpuesto al tiempo de aprendizaje de la plataforma. De esta forma, se sugiere que el modelo propuesto es más importante y relevante que la implementación exacta desarrollada.

Los resultados cualitativos muestran que el modelo propuesto es capaz de generar imágenes de alta calidad, muy superiores a las técnicas tradicionales en las que la representación de los cortes es realizada sobre el volumen original. Al utilizar el algoritmo de despliegue especializado, es posible determinar la posición y orientación de las superficies con una resolución subvolumétrica.

De las diferentes estrategias implementadas para realiza la estimación del vector gradiente, utilizar el gradiente de la superficie que se va a sombrear ofreció las imágenes de mayor calidad. Sin embargo, se cree que la calidad del resultado puede ser mejorada con diferentes aproximaciones. Por otra parte, la regla de combinación de resultados no permite modelar los bordes agudos, ya que los mismos son suavizados al eliminar el mellado generado por las técnicas clásicas.

Por otra parte, el algoritmo de despliegue sólo es capaz de manejar un volumen médico, por lo que los fragmentos separados no pueden ser transformados de forma independiente y no se permite la manipulación de dispositivos protéticos. Estas limitaciones sólo pueden sobrellevarse adaptando el algoritmo de despliegue a múltiples volúmenes médicos, aunque la idea subyacente no ha de ser modificada.

En conclusión, el modelo propuesto ofrece diversas ventajas cualitativas sobre los modelos tradicionales, pero aún se puede mejorar sustancialmente la calidad de las imágenes generadas. Más aún, el desempeño de los algoritmos permite que el modelo sea utilizado en ambientes de tiempo real. Se espera que el rendimiento pueda ser mejorado con optimizaciones sobre los algoritmos paralelos, pero también se espera

que gracias al modelo de programación escalable de CUDA y el avance constante de las tarjetas gráficas, el rendimiento mejore con el simple pasar del tiempo.

Trabajos a Futuro

El algoritmo de despliegue volumétrico propuesto no puede manejar más de un volumen médico, sin embargo, adaptarlo a múltiples volúmenes permitiría la transformación individual de los fragmentos de hueso y la manipulación de dispositivos protéticos. En la Sección 4.2.2 se muestra una tabla que analiza todos los casos que el algoritmo de emisión de rayos debe tener en cuenta cuando existe un único volumen médico, dicha solución es sencilla y eficiente para el caso presentado. Sin embargo, si se desea extender el modelo para soportar múltiples volúmenes médicos al mismo tiempo, intentar analizar todos los casos manualmente implica grandes costos y riesgos. Sería mejor implementar el algoritmo de muestreo adaptativo propuesto en [21], con el cual la cantidad de volúmenes médicos que se pueden manejar se vería limitado únicamente por la memoria disponible y el tiempo de ejecución del algoritmo.

La implementación desarrollada permite especificar para cada componente del instrumento de corte una bandera que especifica si éste realmente corta el volumen, sin embargo, se permite que los componentes no cortantes atraviesen el volumen. Debería implementarse un método de detección y manejo de colisiones para evitar que dichos componentes atraviesen el modelo, una opción sería la propuesta en [17].

El algoritmo de voxelización presentado no utiliza ninguna información sobre el volumen médico durante la creación de la superficie de corte, esto implica que el instrumento de corte es capaz de cortar cualquier parte del volumen médico con la misma facilidad. Sin embargo, para realizar simulaciones de cirugías de hueso realistas es necesario extender los atributos del modelo para así representar que tan resistente es cada vóxel a ser cortado. También es necesario que las componentes de corte del

instrumento especifiquen la facilidad con la que pueden combatir dicha resistencia. De esta forma, se pueden especificar tipos de instrumentos que cortan fácilmente ciertos materiales pero no aquellos que sean más fuertes.

Como se mencionó en la Sección 4.2.3 la estimación del gradiente en la frontera de las superficies de corte no es un cálculo trivial. Más aún, el análisis de resultados muestra que las imágenes generadas no son completamente correctas. Por lo tanto, es sumamente importante investigar diferentes alternativas para realizar dicha estimación, ya que es la única manera de obtener imágenes con la calidad necesaria.

Actualmente el algoritmo de emisión de rayos se encuentra basado en detección de isosuperficies. El problema de este enfoque es que no permite visualizar el interior de los huesos, incluso luego de cortar y eliminar un fragmento del mismo. Se proponen dos opciones simples, la primera es filtrar el volumen médico antes de comenzar la simulación para remover los vóxeles que se encuentren por fuera del hueso, luego se puede utilizar un algoritmo de emisión de rayos sin detección de isosuperficies. La segunda opción es modificar el algoritmo de despliegue de manera que el comportamiento sea híbrido, es decir, si el rayo consigue la isosuperficie deseada fuera de la superficie de corte, esta es mostrada; por otra parte, si la primera vez que se encuentra la isosuperficie ésta se encuentra dentro de la superficie de corte, el algoritmo comienza a acumular muestras basadas en una función de transferencia. Para ambas propuestas, el resultado sería que al remover un fragmento de hueso el interior del mismo se hace visible.

En la Sección 4.1.1.1 se muestra una regla muy simple para combinar los resultados de la voxelización con las superficies de corte existentes (4.3). Más aún, la Figura 5.17 muestra que dicha regla no permite modelar los bordes agudos que genera la intersección del instrumento de corte con las superficies de corte existentes. Para poder modelar dichos bordes, es necesario implementar una estrategia similar a la propuesta en [21], en la que la superficie de corte es muestreada a nivel subvolumétrico durante la voxelización del instrumento. De esta forma se puede estimar el volumen

del sólido original que es realmente afectado por el corte nuevo, permitiendo que la combinación de resultados genere la posición y orientación correcta para la superficie resultante.

Finalmente, en la Sección 4.3.2 se propone un algoritmo para la detección de fragmentos de hueso separados basado en búsqueda en anchura. El análisis de resultados muestra que dicho método es muy costoso para ser ejecutado en tiempo real. Una alternativa es utilizar una estrategia de llenado basado en semillas similar a la propuesta en [17], dicha estrategia podría ofrecer mejores tiempos de ejecución y permitir la ejecución del algoritmo en tiempo real.

Bibliografía

- [1] M. Agus, A. Giachetti, E. Gobbetti, G. Zanetti, y A. Zorcolo. “Adaptive techniques for real-time haptic and visual simulation of bone dissection.” En *IEEE Virtual Reality Conference*, pp. 102–109. IEEE Computer Society Press, Conference held in Los Angeles, CA, USA, marzo 2003.
- [2] J. F. Blinn. “Light reflection functions for simulation of clouds and dusty surfaces.” *SIGGRAPH Computer Graphics*, vol. 16, no. 3, pp. 21–29, julio 1982.
- [3] W. Cai y G. Sakas. “Data Intermixing and Multi-volume Rendering.” *Computer Graphics Forum*, vol. 18, pp. 359–368, 1999.
- [4] M. Chen, A. S. Winter, D. Rodgman, y S. Treuvett. “Enriching volume modelling with scalar fields.” En *Data Visualization: The State of the Art*, pp. 345–362. 2003.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, y C. Stein. *Introduction to Algorithms*. The MIT Press, 3^a edición, 2009.
- [6] R. A. Drebin, L. Carpenter, y P. Hanrahan. “Volume rendering.” *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 65–74, junio 1988.
- [7] J. Gregory, J. Lander, y M. Whiting. *Game Engine Architecture*. A K Peters, 2009.
- [8] K. H. Höehne y R. Bernstein. “Shading 3D-Images from CT Using Gray-Level Gradients.” *IEEE Transactions on Medical Imaging*, vol. 5, no. 1, pp. 45–47, marzo 1986.

- [9] K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, y G. Wiebecke. “3D-visualization of tomographic volume data using the generalized voxel-model.” *Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization*, pp. 51–57, 1989.
- [10] J. T. Kajiya. “The rendering equation.” *SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 143–150, agosto 1986.
- [11] J. T. Kajiya y B. P. Von Herzen. “Ray tracing volume densities.” *SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 165–174, enero 1984.
- [12] A. Leu y M. Chen. “Modelling and rendering graphics scenes composed of multiple volumetric datasets.” *Computer Graphics Forum*, vol. 18, pp. 159–171, 1999.
- [13] M. Levoy. “Display of Surfaces from Volume Data.” *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, mayo 1988.
- [14] M. Levoy. “Efficient ray tracing of volume data.” *ACM Transactions on Graphics*, vol. 9, no. 3, pp. 245–261, julio 1990.
- [15] M. Levoy. “Volume rendering by adaptive refinement.” *The Visual Computer: International Journal of Computer Graphics*, vol. 6, no. 1, pp. 2–7, febrero 1990.
- [16] W. E. Lorensen y H. E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm.” *SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 163–169, agosto 1987.
- [17] T. Ming-Dar y H. Ming-Shium. “Volume manipulations for Simulating bone and joint surgery.” *IEEE Transactions on Information Technology in Biomedicine*, vol. 9, no. 1, pp. 139–149, agosto 2005.
- [18] D. R. Nadeau. “Volume scene graphs.” *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pp. 49–56, 2000.

- [19] H. Navarro. *Implementación y evaluación comparativa de métodos de detección de colisiones para un ambiente de Realidad Virtual Inmersiva*. Tesis de Maestría, Universidad Central de Venezuela, 2005.
- [20] NVIDIA. “NVIDIA C Programming Guide Version 3.1.”, 2010.
- [21] B. Pflessner, A. Petersik, U. Tiede, H. K. Höhne, y R. Leuwer. “Volume cutting for virtual petrous bone surgery.” *Computer Aided Surgery*, vol. 7, no. 2, pp. 74–83, junio 2002.
- [22] H. Potter, J. Linklater, A. Allen, J. Hannafin, y S. Haas. “Magnetic resonance imaging of articular cartilage in the knee: An evaluation with use of fast-spin-echo imaging.” *Bone Joint Surgery*, vol. 80, pp. 1276–1284, 1998.
- [23] G. Rondón. *Quirófano Virtual y su prototipo de Interfaz para un sistema de mesa de Trabajo Virtual*. Trabajo Especial de Grado, Universidad Central de Venezuela, 2002.
- [24] U. Tiede, K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, y G. Wiebecke. “Surface Rendering.” *IEEE Computer Graphics and Applications*, vol. 10, no. 2, pp. 41–53, marzo 1990.
- [25] B. Urbina. *Implementación de un Modelo de Deformación de Objetos para la Simulación de Tejidos Blandos*. Trabajo Especial de Grado, Universidad Central de Venezuela, 2007.
- [26] A. S. Winter. *Field-Based Modelling and Rendering*. Tesis Doctoral, University of Wales, 2002.

