Programa Oficial de Postgrado en Ingenierías Transversales

Máster en Ingeniería Biomédica

Proyecto Fin de Máster

GENERACIÓN DE IMÁGENES VOLUMÉTRICAS DE DATOS BIOMÉDICOS EN TIEMPO REAL

Autor: D. David Anaya Treviño

Director: D. Francisco José Serón Arbeloa

Septiembre 2009

Agradecimientos

A Francisco Serón Arbeloa, catedrático de la Universidad de Zaragoza, por haber dirigido este proyecto y por toda la ayuda prestada a lo largo del mismo.

Al doctor Juan José Herrero, del servicio de radiología del hospital Militar de Zaragoza, por su implicación en este proyecto.

A Jorge López y a todos los compañeros del laboratorio por aconsejar y responder a todas las dudas y preguntas que les he planteado.

GENERACIÓN DE IMÁGENES VOLUMÉTRICAS DE DATOS BIOMÉDICOS EN TIEMPO REAL

Resumen

La visualización de campos escalares 3D como los obtenidos del interior del cuerpo humano a partir de CT (Computerized Tomography) o de MRI (Magnetic Resonance Imaging), cubre un amplio abanico de técnicas que persiguen presentar de la forma más clara y eficiente la información contenida en esos datos. El volumen de datos generado por estos dispositivos es considerable y provoca que los actuales sistemas sean lentos en el procesamiento y visualización de estos datos.

En este proyecto se presentará un visualizador capaz de procesar y mostrar datos escalares con distribución espacial en tiempo real, utilizando CUDA la última tecnología para trabajar con GPGPUs (General-Purpose Computing on Graphics Processing Unit) y ray cast, el algoritmo más adecuado para ser utilizado con esta tecnología y conseguir la mayor calidad de imagen. El sistema será integrado en un motor gráfico más completo, que podrá ser utilizado para mostrar los resultados de un estudio radiológico de una forma muy realista e intuitiva al cirujano o médico especialista o en clases de medicina en la Universidad.

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Cuadros

Abreviaturas

GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Unit
CUDA	Compute Unified Device Architecture
GLSL	OpenGL Shading Language
MIP	Maximum Intensity Projection
MRI	Magnetic Resonance Imaging
CT	Computed Tomography
PET	Positron Emission Tomography
CPU	Central Processing Unit
HLSL	High Level Shader Language
OpenCL	Open Computing Language
SM	Streaming Multiprocessor
SP	Streaming Processor
MT IU	MultiThreaded Instruction Unit
SFU	Special Function Units
DRAM	Dynamic Random Access Memory
API	Aplication Program Interface
VTK	Visualization ToolKit
GDCM	Grassroots DICOM library

Capítulo 1

Introducción

1.1 Contexto

La adquisición de imágenes médicas es uno de los medios más frecuentes, utilizados en medicina, para el estudio y diagnóstico de las distintas patologías o lesiones del cuerpo humano.

Usando diferentes técnicas de escaneo como por ejemplo MRI, CT, PET o ultrasonidos se adquieren del interior del cuerpo humano datos escalares con distribución espacial que, a partir de ahora y para utilizar la terminología anglosajona, los denominaré "datos volumétricos".

La visualización de estos datos volumétricos denominada "renderizado de volumen" es una rama de la informática gráfica que puede utilizarse para dar color y textura a datos escalares y visualizar diferentes estructuras transparentes, semi-transparentes u opacas, recreando una imagen del cuerpo más cercana a la real, de donde se pueden obtener indicios útiles de anomalías. Múltiples aplicaciones utilizan esta técnica para la detección de tumores, visualización de aneurismas, planificación de una operación quirúrgica o incluso monitorización en tiempo real de una operación.

Pero no solo la medicina es la única área en la que se generan estos tipos de datos volumétricos. En campos como el diseño o el análisis, así como el testeo de la calidad de materiales y prototipos, se generan datos volumétricos a través de escáneres industriales o ultrasonidos. En análisis microscópicos a partir de microscopios confocales es posible conseguir rodajas ópticas microscópicas de alta resolución sin tener que perturbar el espécimen. Otra gran fuente de datos volumétricos es la simulación física donde se simula la dinámica de fluidos. A menudo se lleva a cabo usando partículas o puntos de muestreo que se mueven alrededor siguiendo leyes físicas dando a lugar puntos sin estructura. Estos puntos pueden visualizarse directamente o ser muestreados en una estructura mallada sacrificando posiblemente la calidad.

La desventaja de los dispositivos de adquisición mencionados es la pérdida de la información del color, la cual necesita ser añadida durante el

proceso de visualización, puesto que las técnicas de adquisición generan valores escalares representando densidades en unidades Hounsfield (CT), oscilaciones (MRI), ecos (ultrasonidos) y otros más.

Los estudios geosísmicos es probablemente una de las fuentes que más cantidad de datos genera. Normalmente se generan al menos 1024^3 vóxeles[1] (1 GByte o más) que necesitan ser visualizados. El campo de aplicación más común es la extracción petrolífera donde los costes se pueden reducir ampliamente encontrando la correcta localización del punto donde realizar la perforación.

Este es el principal problema al que se tiene que enfrentar el renderizado de volumen, "la gran cantidad de datos que tiene que manejar". Esto conlleva que las aplicaciones de visualización 3D de estos datos volumétricos sean lentas a la hora de mostrar el resultado, dificultando la interactividad del usuario con la aplicación y obligando a este a trabajar con secciones reducidas de los datos o con sistemas de visualización 2D.

Existen principalmente tres metodologías para realizar el renderizado de volumen. La primera está basada en utilizar hardware específico dedicado a este tipo de renderizado [1, 2, 3]. La segunda está basada en trabajar principalmente sobre CPU [4], delegando únicamente en la tarjeta gráfica el dibujado en pantalla, y la tercera, que es en la que se basa este proyecto, combina la utilización de CPU y de tarjetas gráficas comunes [5] tanto para realizar el cálculo del algoritmo como para mostrar las imágenes.

Las soluciones basadas únicamente en hardware aportan gran calidad de imagen y permiten trabajar con los datos en tiempo real, sin embargo, están limitadas en las funcionalidades que ofrecen.

Los sistemas que funcionan principalmente sobre CPU tienen la ventaja de que son muy flexibles para añadir nuevas funcionalidades así como de perfeccionar las existentes, pero no trabajan muy bien con grandes cantidades de datos y buscan técnicas para reducir la visualización a una parte de ellos.

Los sistemas que combinan las dos soluciones anteriores consiguen sus virtudes, flexibilidad y velocidad, pero tienen el inconveniente de que el intercambio de datos entre la tarjeta gráfica y la CPU es lento, por lo que tienen que buscar siempre la forma de reducir este tráfico.

[1] **Vóxel** (la palabra proviene de la contracción del término en inglés "volumetric pixel") es la unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional y es, por tanto, el equivalente del píxel en un objeto 2D.

1.1.1 Nuestro caso particular

Durante las prácticas realizadas en el Hospital General de la Defensa de Zaragoza se pudo comprobar "in situ" la manera de trabajar del radiólogo Dr. Juan José Herrero.

A partir de los datos procedentes de un escáner realizado a un paciente, el radiólogo segmentaba los datos que le interesaban dependiendo de la patología buscada, trabajando con los datos corte a corte (rodajas). Una vez que ya tenía la segmentación concluida podía extraer una reconstrucción 3D gracias al software utilizado. Pero debido a la complejidad de manejo del visualizador 3D, y al bajo frame rate (imágenes/segundo) soportado que impedía trabajar en tiempo real con el modelo 3D, el Dr. Herrero optaba por realizar diferentes capturas de los planos más representativos de la anomalía y éstas eran impresas en negativo, para enviarlo posteriormente al cirujano o médico especialista. La figura 1 muestra el proceso completo.

Figura 1. Pasos de un examen radiológico. 1 realizar el escáner, 2 segmentar a partir de cortes 2D, 3 generar modelo 3D, 4 imprimir en negativo los planos representativos.

La lentitud del visualizador 3D y el hecho de haber estudiado y trabajado siempre con cortes 2D provocaba que el Dr. Herrero se sintiera más cómodo trabajando en este formato.

1.2 Objetivos

En este proyecto se pretende desarrollar un sistema de visualización de datos biomédicos que permita al usuario trabajar con los datos en tiempo real, así como realizar operaciones simples de segmentación. Este sistema se integrará en un entorno de visualización más completo que aporta un interfaz intuitivo para el usuario, que permitirá la visualización de los datos médicos en estereoscopía, aumentando de esa forma la sensación 3D y permitirá

realizar operaciones más complejas como utilizar planos de corte, cropping u operaciones de segmentación más elaboradas.

El Dr. Herrero pretende introducirlo en sus clases en la Facultad de Medicina y desea que el sistema pueda cargar los datos resultado de su trabajo en el hospital, así como datos procedentes de otros sistemas de captura de imágenes médicas de forma que pueda utilizarlos durante sus clases, por ello el sistema deberá ser capaz de cargar datos en formato DICOM. Con esto se acercará a los estudiantes universitarios los beneficios del 3D en la visualización de datos médicos.

Para ello el sistema debe ser capaz de funcionar en tiempo real. Para conseguir esta capacidad se utilizará la tecnología CUDA (Compute Unified Device Architecture) [6] para acelerar el renderizado de volumen.

En Febrero de 2007 NVIDIA publicó el primer SDK para CUDA, un conjunto formado por un compilador, varias interfaces de programación (una de alto nivel y otra de bajo), drivers y un conjunto de herramientas de desarrollo que permiten a los programadores usar una variación de diferentes tipos de lenguaje de programación (C, python, Fortran, Java) para codificar algoritmos en GPUs, utilizando el controlador de NVIDIA a partir de la serie G8X en adelante. CUDA intenta explotar las ventajas de las GPUs frente a las CPUs de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos. Por ello, si una aplicación se puede diseñar de forma que numerosos hilos realicen tareas independientes, una GPU podrá ofrecer un rendimiento superior al de un procesador convencional.

La tecnología ofrecida por NVIDIA con CUDA es muy prometedora y muchos auguran que desbancará a otros lenguajes de programación sobre GPUs como GLSL [7], OpenCL [8] o HLSL [9].

1.3 Estructura de la memoria

Esta memoria está dividida en 6 capítulos, incluido este capítulo a modo de introducción. A partir de aquí se realiza una revisión de los principales algoritmos empleados para el renderizado de volumen (capítulo 2) y se describe la arquitectura de la tarjeta gráfica utilizada, así como la metodología de trabajo aportada por CUDA (capítulo 3). En el capítulo 4 se describe el trabajo realizado en este proyecto (elección del algoritmo, método de interpolación, carga de datos DICOM, estrategias para acelerar el renderizado y el interfaz de usuario para el manejo de la función de transferencia). Los resultados obtenidos se presentan en el capítulo 5. Finalmente las líneas de trabajo futuras y las conclusiones alcanzadas son tratadas en el capítulo 6.

Capítulo 2

Algoritmos de renderizado de volumen

El objetivo que se pretende alcanzar con la lectura de este capítulo es el conocimiento de los principales algoritmos de renderizado de volumen. La mayoría de estos algoritmos siguen principalmente dos estrategias:

- Espacio imagen: Desde cada píxel del plano de la imagen se lanzan rayos a través del volumen para calcular el color del píxel (isosuperficies, ray casting).

- Espacio objeto: El volumen es atravesado, de delante atrás o viceversa, y se procesa cada vóxel para determinar su contribución a la imagen (splatting).

- También hay algunos que utilizan una combinación de ambas (shear-warp) y otros emplean estrategias diferentes basándose en las ventajas aportadas por determinados módulos del hardware (Texturas 2D/3D).

A continuación se describirán los fundamentos generales de cada método así como sus ventajas y desventajas.

2.1 Isosuperficies

Una isosuperficie es una superficie que representa puntos de un valor constante (isovalor) dentro de un volumen. En el caso de tratarse de datos volumétricos procedentes de un TAC una isosuperficie representa superficies del cuerpo con valores de intensidad constante en unidades Hounsfield (p. e. piel, huesos, órganos), como muestra la Figura 2.

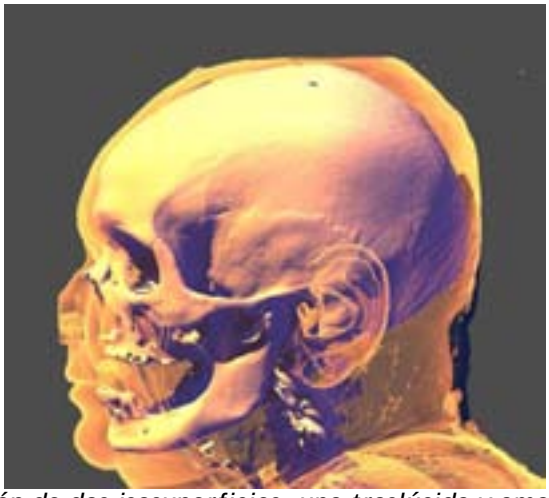


Figura 2. Visualización de dos isosuperficies, una traslúcida y amarilla para la piel y otra opaca y blanca para los huesos [10].

El método más utilizado para la construcción de isosuperficies a partir de datos volumétricos es el algoritmo de marching cubes.

2.1.1 Marching cubes

El algoritmo de marching cubes se publicó en 1987 por Lorensen y Cline [11] y su objetivo es la extracción de mallas poligonales de una isosuperficie a partir de datos escalares tridimensionales (vóxeles). Es equivalente a la versión en 3D del algoritmo 2D de marching squares.

Principalmente se basa en dividir el espacio en vóxeles (cubos) formados por los valores de intensidad de cada una de las 8 esquinas del vóxel que se corresponden con puntos obtenidos de los datos volumétricos procedentes del TAC.

El algoritmo recorre cada uno de estos vóxeles y si una o más esquinas del vóxel tienen valores menores que el isovalor especificado, y una o más esquinas tienen valores mayores que el isovalor, sabemos que el vóxel forma parte de la isosuperficie.

Determinando que lados del vóxel son intersectados por la isosuperficie, podemos crear polígonos triangulares los cuales dividen el vóxel en regiones que están dentro de la isosuperficie y otras que están fuera.

Como el cubo está formado por 8 lados, existen 2^8 = 256 maneras diferentes en las que la isosuperficie puede atravesar el vóxel, que se pueden reducir a 15 casos topológicamente distintos (ver Figura 3) por razones de rotación y simetrías. A estos 15 se le añaden otros 6 casos para solucionar ciertos casos en los que puede producirse ambigüedades o huecos en la superficie. Inicialmente se crea un vector con cada una de estos casos y se asigna un bit a cada una de las 8 esquinas del vóxel. Si el valor escalar de una

esquina es mayor que el isovalor (está dentro de la superficie) entonces su bit correspondiente se pone a 1, mientras que en caso contrario (está fuera de la superficie) se pone a 0. El valor final de los 8 bits se utiliza como índice para acceder al vector de configuraciones y obtener la correcta.

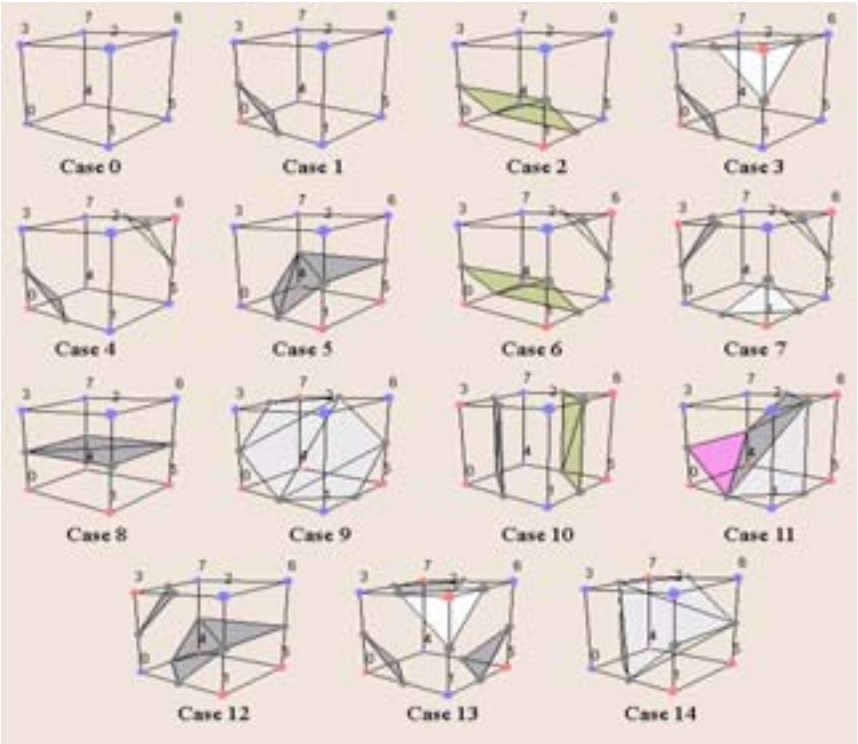


Figura 3. Las 15 posibles topologías de intersección del vóxel por la isosuperficie [12].

Una vez que se conoce la topología de la isosuperficie dentro del vóxel, cada vértice del polígono que forma el contorno de esa isosuperficie y que intersecta en algún lado del vóxel se calcula por interpolación lineal de los valores de intensidad de los vértices que forman ese lado del vóxel.

El gradiente del campo escalar en cada punto de la malla de datos volumétricos es también la normal de una isosuperficie que hipotéticamente pasara por ese punto. Por lo tanto, interpolando estas normales a lo largo de los lados del vóxel atravesado por la isosuperficie se obtienen las normales de los vértices del polígono triangular generado. Estos gradientes son esenciales para el sombreado e iluminación de la malla resultante.

Por último, conectando los polígonos triangulares de cada uno de los vóxeles que se encuentran en la frontera de la isosuperficie, se obtiene una representación mallada de la isosuperficie con el isovalor buscado.

La ventaja de este algoritmo es que es rápido, pero tiene el inconveniente de que no muestra el interior del objeto, por lo que la posibilidad de realizar cortes o disecciones no aporta ninguna información adicional.

2.2 Shear-warp

La idea que persigue el algoritmo de renderizado shear-warp es transformar previamente los datos con los que se va a trabajar, dándoles una orientación concreta a partir de la cual el renderizado pueda ser más rápido. Aunque la técnica fue presentada por Cameron y Undrill [13], fueron Lacroute y Levoy [14] quienes popularizaron este algoritmo por su desarrollo más efectivo.

En lugar de proyectar los vóxeles sobre el plano imagen en un ángulo variable, se utiliza una transformación shear (cizalla) de forma que se traslada cada rodaja (slice) y se muestrea a lo largo de la dirección del rayo (ver figura 4).

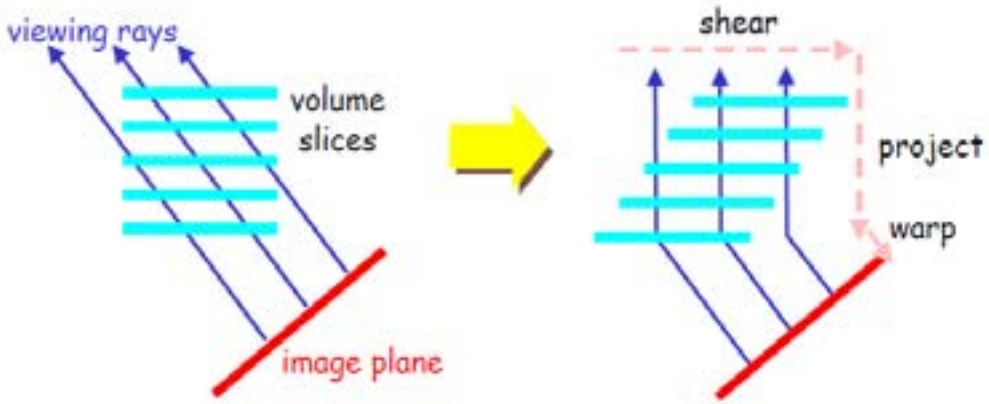


Figura 4. Esquema del algoritmo shear-warp.

La proyección es ahora trivial. Las rodajas se componen en orden de delante atrás, creando una imagen intermedia (ver figura 4), pero en esta ocasión el cálculo es más simplificado puesto que los rayos son perpendiculares a las rodajas del volumen.

Esta imagen intermedia es entonces corregida utilizando una transformación warp (deformación) (ver figura 5).

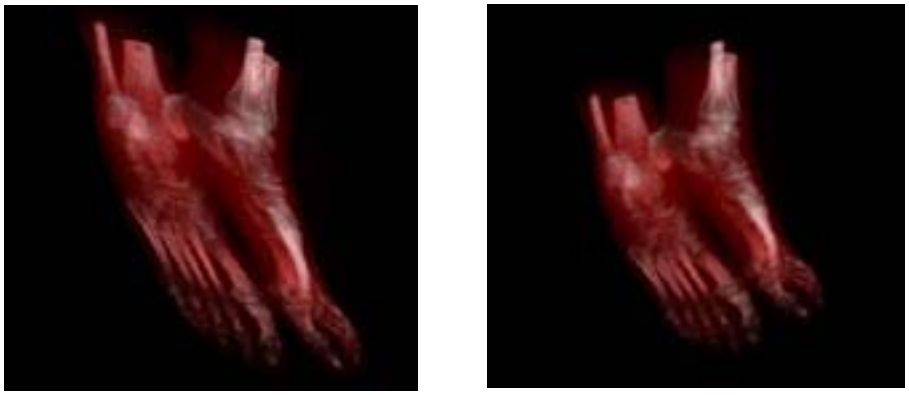


Figura 5. La imagen de la izquierda es el resultado de la proyección de las rodajas "sheared" sobre la imagen intermedia. La imagen de la derecha es el resultado final obtenido de la transformación warp de la imagen intermedia [15].

El algoritmo explota la coherencia en los datos del volumen usando una codificación run-length. Esto aumenta la velocidad del algoritmo al reducir la sobrecarga en memoria que se produce al almacenar múltiples copias del volumen, debido a la habilidad de tener estos volúmenes alineados con los ejes.

A pesar de ser un algoritmo bastante rápido se han hecho mejoras para conseguir la paralelización del algoritmo [16] y así poder usar de ese modo multiprocesadores, pero sigue teniendo la desventaja de que la calidad de imagen conseguida es inferior a la obtenida con métodos como el ray casting.

2.3 Splatting

Splatting, propuesto por primera vez por Westover [17, 18], es una aproximación orientada al objeto, lo opuesto al ray casting que es orientado a la imagen. Cada vóxel se proyecta sobre el plano imagen o como definía Westover, "splatted", como estrellar una bola de nieve, solapando la proyección de los anteriores.

Esta proyección se aproxima por un núcleo Gaussiano invariante a la orientación con una amplitud escalada de acuerdo con el valor del vóxel (ver Figura 6 a). La imagen es generada al proyectar las funciones base sobre la pantalla. La proyección de estas funciones base con simetría radial puede llevarse a cabo eficientemente mediante el barrido de tablas de huellas precalculadas. Cada una de las entradas de la tabla contiene el núcleo de la función integrada analíticamente a lo largo del rayo atravesador.

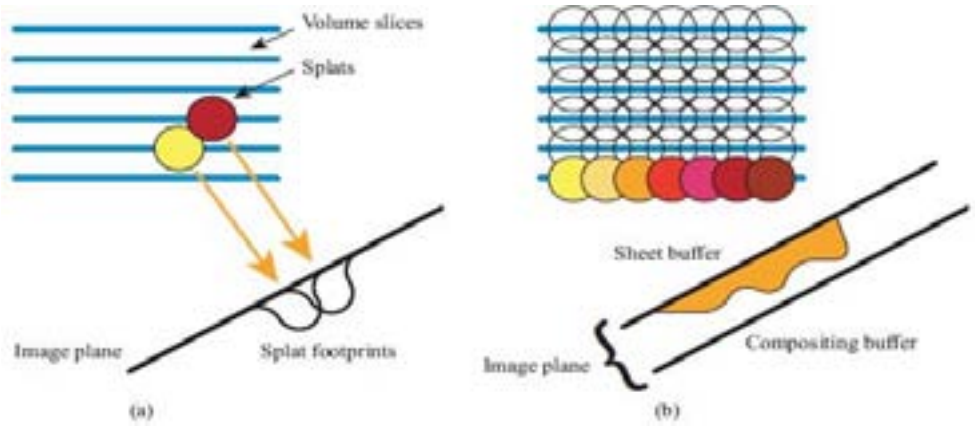


Figura 6. Splatting: (a) huellas de las splats. (b) Sheet buffer compuesto de splats [4].

Con el objetivo de obtener composiciones correctas, el volumen tiene que ser procesado en el orden visible correcto. El método más común es usar un sheet buffer (ver Figura 6 b). Los núcleos de los vóxeles son procesados dentro de slabs que están alineadas paralelamente al plano imagen. Todos los núcleos de los vóxeles que superponen una slab son sumados en el sheet

buffer. Este método reduce significativamente la severa aparición de errores que pueden ocurrir durante el splatting, sin embargo, dependiendo del factor de zoom, cada splat puede cubrir hasta cientos de píxeles que necesitan ser procesados.

El método splatting que provee mejor calidad utiliza como función base o función de huella una combinación de un núcleo de reconstrucción elíptica Gaussiana con un filtro paso bajo Gaussiano, por lo tanto la calidad de la imagen es mejorada significantemente.

La mayor ventaja del splatting es que los vóxeles que, debido a que son transparentes, no contribuyen a la imagen final no tienen que ser procesados. Esto reduce enormemente la cantidad de datos que se tienen que procesar.

2.4 Ray Casting

Ray casting es un algoritmo orientado al espacio imagen. Es una técnica que ofrece resultados de muy alta calidad. En este método se genera un rayo por cada un de los píxeles de la imagen. El rayo sale del centro de proyección de la cámara (punto de visión) y pasa a través del píxel del plano imagen situado entre la cámara y el volumen que se desea renderizar, como muestra la Figura 7.

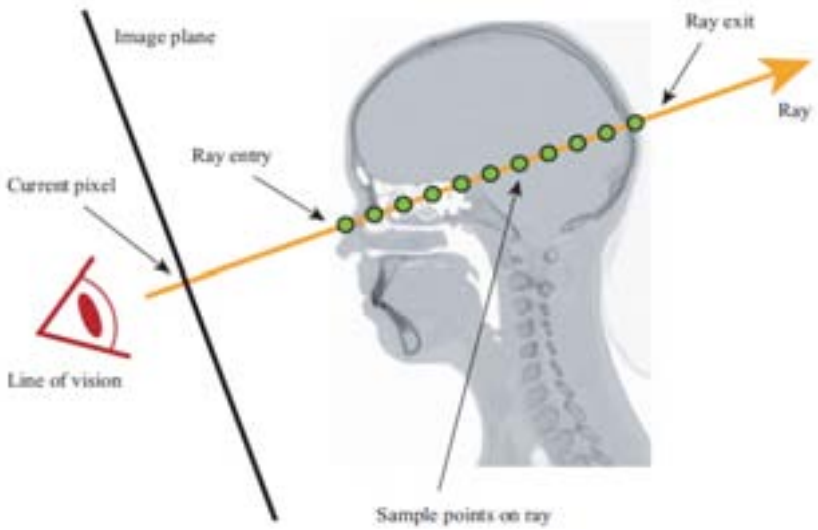


Figura 7. Esquema del proceso de ray casting [4].

Este rayo es muestreado en intervalos regulares o adaptativos a través del volumen, en orden de delante atrás o al revés. El valor de cada punto muestreado es obtenido por interpolación a partir de los valores de los puntos que forman las esquinas del vóxel en el que el rayo está siendo muestreado. A partir del valor resultado de la interpolación, una función de transferencia le asigna al punto muestreado sus propiedades de color y transparencia (RGBA).

La función de transferencia se encarga de mapear el valor de un punto del volumen (intensidad en unidades Hounsfield) a un valor de color y transparencia.

Este valor se compone junto con el valor RGBA acumulado por el rayo en su paso por los puntos de muestreo anteriores.

Esta composición se expresa como una aproximación de la integral de Low-Albedo [19,20] de la siguiente forma:

Si el orden en el que se realiza la composición es de atrás a delante:

$$C_{i+1} = c_i \left(1 - \alpha_i\right) + C_i \alpha_i \qquad (2.1)$$

C_i es el color compuesto actual, y c_i y α_i son el color y el valor alfa (transparencia) de la muestra actual que nos devuelve la función de transferencia.

Si el orden en el que se realiza la composición es de delante atrás:

$$C_{i+1} = \alpha_i c_i + \left(1 - \alpha_i\right) A_i C_i$$
$$A_{i+1} = \alpha_i + \left(1 - \alpha_i\right) A_i \qquad (2.2)$$

C_i y A_i son el color y la transparencia compuesta actual, y c_i y α_i son el color y el valor alfa (transparencia) de la muestra actual.

El valor α es la medida del grado de transparencia de un objeto. Varía entre 0 y 1, donde alpha = 0 = transparencia total y alpha = 1 = opacidad.

La mayoría de los métodos de aceleración aplicados al ray casting se centran en evitar cálculos innecesarios, tales como el método de terminación temprana del rayo, en el que el muestreo a lo largo del rayo termina una vez que ha sido alcanzado el máximo de opacidad acumulado, o el salto de espacios, donde no se procesan áreas totalmente transparentes del volumen de acuerdo a un intervalo de muestreo adaptativo.

Hay técnicas que se aprovechan de la coherencia entre píxeles y la coherencia entre vóxeles. La técnica de supersampling se basa en aprovechar la coherencia del píxel. En un conjunto de píxeles se trazan los rayos y en el resto se interpola el valor de los píxeles calculados. Para las zonas en las que se ve grandes cambios de gradiente se trazan nuevos rayos.

La coherencia entre vóxeles permite aumentar la distancia de muestreo en zonas homogéneas. En zonas con cambios de gradiente o ambiguas se reduce la distancia de muestreo.

Algunos algoritmos utilizan un muestreo del rayo en distancias equivalentes o equidistantes. Mientras que otros ajustan esta distancia o la varían para eliminar errores de patrones de muestreo o aplicar la técnica de salto de espacios para acelerar el paso por regiones vacías.

Las ventajas del ray casting es que aporta la mayor calidad de imagen a costa de un procesamiento más lento pero su esquema de cálculo permite explotar cálculos paralelos así como estructuras de memoria eficientes para ganar velocidad [21, 22].

2.4.1 MIP

El algoritmo Maximum Intensity Projection es una versión simplificada del ray casting, en el que en el plano imagen únicamente se proyectan los vóxeles con máxima intensidad, que aparecen en el camino de los rayos paralelos trazados desde el punto de vista. Esto implica que dos renders desde puntos opuestos de vista son imágenes simétricas.

Esta técnica es muy rápida pero los resultados no dan sensación de profundidad. Para mejorar la sensación 3D se suelen utilizar animaciones de renders de algunas imágenes MIP con el punto de vista rotado.

2.5 Texture mapping

Los algoritmos de mapeado de texturas (texture mapping) se basan en aprovechar el hardware de las tarjetas gráficas para trabajar con texturas.

2.5.1 2D Texture mapping

Con el objetivo de explotar las capacidades de las texturas 2D para renderizar volúmenes, se representa el conjunto de datos volumétricos mediante tres pilas de texturas alineadas con el objeto (Ver Figura 8).

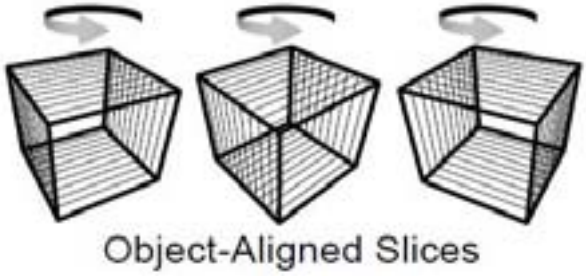


Figura 8. Rodajas alineadas con el objeto [23].

De las tres pilas disponibles, el renderizado se realiza sobre aquella cuyas texturas son "más paralelas" a la dirección de visión y así se evita la aparición de polígonos deformados durante la proyección. Los polígonos texturizados se renderizan de atrás hacia delante y los fragmentos generados se van componiendo con los píxeles que se encuentran en el frame buffer. La variación en la dirección de visión provoca que la distancia entre los puntos de muestreo, que compondrán el valor del píxel procesado, también varíe. Consecuentemente, la opacidad de las rodajas se debe adaptar con respecto a esta distancia con el fin de ajustarse al grosor que el muestreo representa. Además, únicamente se realiza interpolación lineal dentro de las rodajas originales lo que provoca unos resultados de menor calidad. Para evitar estas desventajas, las texturas 2D se pueden combinar con registros de múltiples texturas provistos por la mayor parte de los procesadores gráficos.

2.5.2 3D Texture mapping

La aparición del soporte de texturas 3D en las tarjetas gráficas ha permitido su aplicación al renderizado de volumen. La idea básica es interpretar el vector de vóxeles como una textura 3D y entender el mapeo de texturas 3D como la interpolación trilineal del conjunto de datos volumétricos en un punto arbitrario dentro del dominio.

Los datos se muestrean sobre planos de corte, que están orientados paralelamente al plano de visión, con los píxeles del plano de corte interpolados trilinealmente a partir de la textura escalar 3D. Esta operación se realiza sucesivamente para múltiples planos que tienen que ser cortados contra el dominio de texturas paramétrico (Figura 9). Los polígonos son renderizados de atrás adelante y las texturas resultantes se componen apropiadamente en el frame buffer aproximando de ese modo la integral continua de Low-Albedo.

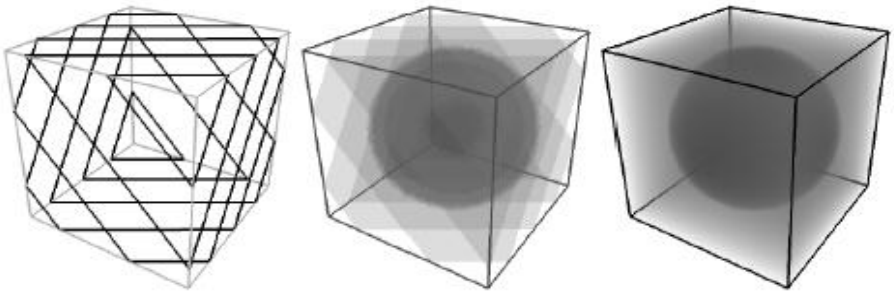


Figura 9. Ejemplo simple de renderizado de volumen usando mapeado de texturas 3D [23].

Existen unidades de hardware gráfico dedicadas exclusivamente para la interpolación trilineal dentro de la textura y para la composición de los fragmentos por píxel. Sin embargo, la principal ventaja del renderizado de volumen usando texturas 3D es la utilización de tablas de texturas previas al dibujado. Las muestras escalares reconstruidas a partir de las texturas 3D son convertidas en píxeles RGBA usando estas tablas. Esto permite manipular directamente la función de transferencia, necesaria para mapear el valor escalar a un valor RGBA, sin la necesidad de recargar la textura entera proporcionándole al usuario una mayor velocidad de interacción. De esta forma se pueden resaltar o eliminar partes arbitrarias de los datos así como utilizar diferentes colores y transparencias.

2.6 Resumen del capítulo.

Al llegar a este punto hemos visto las principales características de los principales algoritmos de renderizado de volumen.

En primer lugar la técnica de isosuperficies, basada en generar superficies malladas de un mismo valor escalar y aplicar trazado de rayos sobre ellas para la composición del color. Es una técnica rápida y la calidad de imagen es buena pero no permite ver el interior de los objetos al tratarse de superficies.

Por otro lado la técnica de shear-warp aplica transformaciones a los datos para pasar del espacio imagen al espacio objeto y viceversa con la intención de facilitar el cálculo del trazado de rayos. Es de las más rápidas pero con baja calidad de imagen.

El algoritmo de splatting se basa en proyectar cada vóxel sobre el plano imagen y realizar la composición del color a partir de funciones huella precalculadas. La calidad de la imagen es buena pero es el más lento de todos puesto que tiene que procesar un gran número de vóxeles.

El método ray casting se basa en trazar un rayo por cada píxel de la imagen y muestrearlo a lo largo del volumen, realizando la composición del color aplicando la integral de Low-Albedo. La calidad de imagen es buena y es

lento, pero el algoritmo se puede paralelizar fácilmente, puesto que el cálculo que realiza cada rayo es el mismo para todos e independiente de los demás.

Por último, el método de texture mapping se basa en descomponer los datos en texturas 2D o 3D y utilizar las técnicas aportadas por las tarjetas gráficas para trabajar con texturas. Son rápidas y de calidad buena pero ya no pueden acelerarse más puesto que dependen de la capacidad de la GPU.

Capítulo 3

GPGPU (CUDA)

En este capítulo pretendo dar a conocer las bases de la tecnología CUDA proporcionada por las tarjetas gráficas NVIDIA. CUDA ofrece un modelo de programación de propósito general sobre las GPU (GPGPU) [24] de NVIDIA. Es decir, no sólo está pensado para apoyar el desarrollo de aplicaciones gráficas 3D, como era habitual hasta ahora, sino que también se puede utilizar para otras áreas como compresión de audio y video, aceleración de ecuaciones matemáticas o desencriptación. En el capítulo veremos la estructura y metodología de trabajo sobre las GPU de NVIDIA, la distribución del hardware y la forma de programar las aplicaciones con CUDA.

3.1 Estructura y metodología de la GPU

La idea es que las partes dedicadas a trabajar con datos en paralelo, de una aplicación que se está procesando en la CPU, son ejecutadas en la tarjeta como núcleos (kernels) que corren en paralelo distribuidos entre muchos hilos (threads). La GPU se puede ver como un coprocesador para la CPU, que además tiene su propia memoria DRAM y es capaz de ejecutar muchos hilos en paralelo.

Los hilos de la GPU son extremadamente ligeros (la sobrecarga necesaria para su creación es muy pequeña), todo lo contrario que los hilos que se utilizan en la CPU, además de que la GPU necesita miles de hilos para trabajar a pleno rendimiento, mientras que una CPU de múltiples núcleos necesita unos pocos.

Las partes de la aplicación que se ejecutan sobre la GPU (kernels) lo hacen como una malla (grid) de bloques de hilos (Ver figura 10). Estos bloques son conjuntos de hilos que pueden cooperar entre si: Compartiendo eficientemente datos a través de una memoria compartida de baja latencia y sincronizando su ejecución de forma que se controla el acceso a esta memoria. Dos hilos de dos bloques diferentes no pueden cooperar.

Cada hilo y cada bloque tienen su propio identificador, el cual se utiliza para designar los datos sobre los que van a trabajar. Estos identificadores

también se utilizan para indexar el acceso a la memoria cuando se procesan datos de múltiples dimensiones.

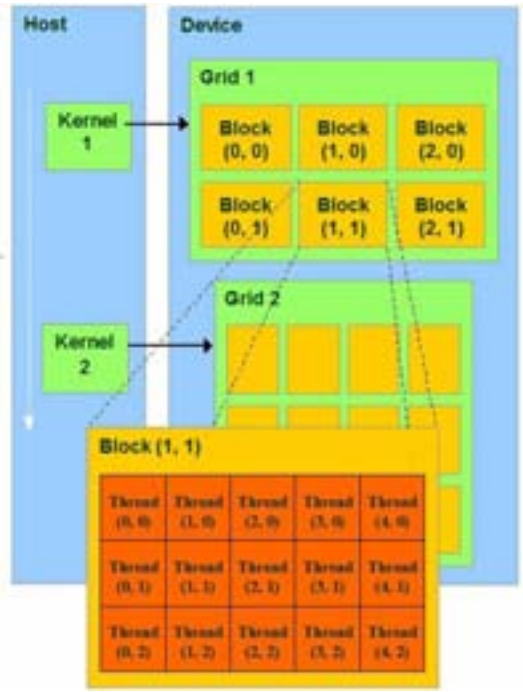


Figura 10. Subdivisión de una aplicación en mallas, bloques e hilos (threads) [25].

La memoria de la GPU es repartida de tal forma que hay una memoria local propia para cada hilo, una memoria compartida por todos los hilos de un bloque y tres memorias (global, textura, constante) compartidas por todos los bloques e hilos a las que tiene acceso la CPU (host) (ver Figura 11).

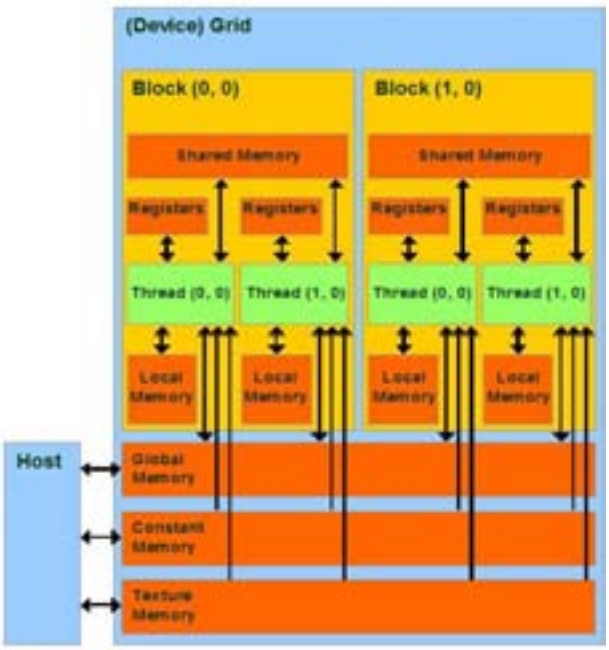


Figura 11. Reparto de la memoria de la GPU [25].

Las memorias local y global residen en la memoria DRAM de la tarjeta, mucho más lentas de acceder que la memoria compartida, por lo que una manera adecuada de realizar cálculos en la GPU es dividir los datos en subconjuntos, los cuales se ajustan al tamaño de la memoria compartida, de acceso mucho más rápido que las otras. Cada subconjunto de datos es utilizado por un bloque de hilos, cargándolo desde la memoria global a la memoria compartida. Al utilizar múltiples hilos se puede explotar el paralelismo del nivel de memoria. Cada hilo puede acceder eficientemente sobre cualquier elemento de los datos. Finalmente se copian los resultados desde la memoria compartida a la memoria global.

Las memorias constante y de textura también residen en la memoria DRAM del dispositivo y son mucho más lentas que la memoria compartida, pero están cacheadas, por lo que tienen una velocidad de acceso de sólo lectura altamente eficiente (Ver tabla 1).

Por todo esto, a la hora de distribuir un programa hay que dividir cuidadosamente los datos de acuerdo con los patrones de acceso:

Para solo lectura, si los datos no están estructurados, sería adecuado guardarlos en la memoria constante, y si los datos están indexados en vectores, sería bueno colocarlos en la memoria de texturas.

Para lectura y escritura compartida entre hilos los datos se guardan en la memoria compartida. Para leer y escribir los datos de entrada y los resultados estos se deben guardar en la memoria global.

Memoria	Tipo	Acceso	Visibilidad	Tiempo de acceso
Registro	Hardware	Lectura/Escritura	Un único hilo	1 ciclo
Local	DRAM	Lectura/Escritura	Un único hilo	Lenta
Compartida	Hardware	Lectura/Escritura	Threads del bloque	1 ciclo
Global	DRAM	Lectura/Escritura	Todos los hilos + CPU	Lenta
Constante	DRAM	Lectura	Todos los hilos + CPU	Variable (cacheada)
Textura	DRAM	Lectura	Todos los hilos + CPU	Variable (cacheada)

Tabla 1. Propiedades de cada tipo de memoria.

3.2 Distribución hardware de la GPU:

La serie 8 de GPUs de NVIDIA y en particular la GeForce 8800 GTX, utilizada en este proyecto, está formada por un conjunto de 16 multiprocesadores (SM) compuestos por un conjunto de 8 procesadores (SP) de 32-bit con una unidad de instrucciones compartida (MT IU). Cada multiprocesador tiene un conjunto de registros de 32 bits por procesador, una memoria compartida on-chip y dos memorias cache de solo lectura para acelerar el acceso a la memoria constante y a la memoria textura (Ver figura 12).

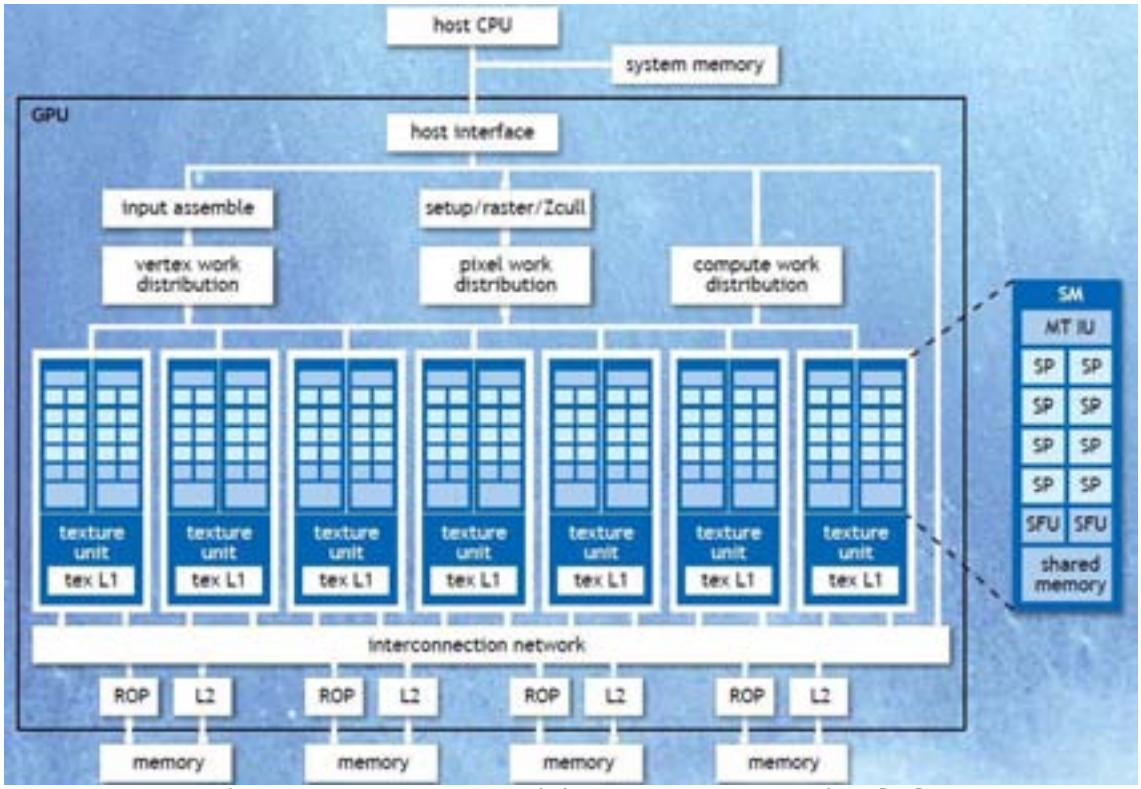


Figura 12. Esquema general de una GPU NVIDIA serie 8 [24].

La GPU procesa solo una malla a la vez. Cada bloque de hilos de una malla es ejecutado por un solo multiprocesador, aunque un multiprocesador puede ejecutar múltiples bloques concurrentemente. La memoria compartida y los registros son repartidos entre los hilos de todos los bloques concurrentes, por lo que decreciendo el uso de memoria compartida y registros por bloque incrementa el número de bloques que se pueden ejecutar concurrentemente.

Las mallas se dividen en bloques de hilos y los hilos se agrupan en "warps". Hay hasta 32 hilos por warp. Únicamente es menor de 32 si el nº de hilos totales no llega a esa cifra. En cada ciclo de reloj cada multiprocesador ejecuta la misma instrucción en los hilos del warp. Hay hasta 16 warps por bloque. Cada bloque, y por lo tanto cada warp, se ejecuta en un único multiprocesador.

La GPU tiene 16 multiprocesadores por lo que al menos 16 bloques se requieren para llenar el dispositivo, aunque más es mejor, porque si los recursos (memoria compartida y registros) lo permiten, más de un bloque se puede ejecutar por multiprocesador.

El siguiente cuadro muestra las especificaciones técnicas de la tarjeta gráfica GeForce 8800 GTX utilizada para este proyecto:

Máximo número de hilos por multiprocesador: 768 Máximo número de hilos por bloque: 512 Máximo tamaño de cada una de las dimensiones de una malla: 65,535 Número de multiprocesadores (Streaming Multiprocessors): 16 @ 675 MHz

Memoria de la tarjeta (DRAM): 768 MB
Memoria compartida por multiprocesador: 16KB dividida en 16 bancos
Memoria constante: 64 KB
Tamaño del warp: 32 hilos (16 Warps/Bloque)
367 GFLOPS
86 GB/s Ancho de banda de la memoria, 4GB/s Ancho de banda hacia la CPU

Cuadro 1. Especificaciones técnicas de la tarjeta utilizada (GeForce 8800 GTX).

3.3 Programación sobre la GPU

La forma de programar una aplicación que se ejecute sobre la GPU dista mucho de ser complicada. La aplicación se programa de forma normal y para aquellas funciones o partes del código (kernels) que se quiere ejecutar sobre la GPU, estas se programan utilizando la interfaz aportada por CUDA.

Obviamente el código de estas funciones debe ser paralelizable de forma que miles de hilos realicen el mismo cálculo independientemente de los demás.

Los métodos y variables aportados por la interfaz que permiten programar el cálculo que va a ejecutar cada hilo, así como la definición de los datos que van a utilizar y el tipo de memoria en el que van a residir esos datos (Ver tabla 2), no son complicados conceptualmente y utilizan una sintaxis más sencilla que otras lenguajes de programación sobre GPU (p. e. GLSL o HLSL).

	Memoria	Alcance	Tiempo de vida
device _local_ int LocalVar;	Local	Hilo	Hilo
device _shared_ int SharedVar;	Compartida	Bloque	Bloque
device int GlobalVar;	Global	Malla	Aplicación
device _constant_ int ConstantVar;	Constante	Malla	Aplicación

Tabla 2. Sintaxis de la definición de variables. Tipo de memoria en el que reside, Visibilidad para el resto de hilos y tiempo de vida.

De hecho lo difícil no es escribir el código en lenguaje CUDA, sino organizar y ajustar bien los recursos que va a consumir cada hilo, reservar memoria para los datos en el tipo de memoria adecuada y establecer los puntos de sincronización en el punto adecuado [26]. Una mala organización puede provocar mucha pérdida de tiempo en el acceso a memoria para leer o escribir datos, en excesivos movimiento de datos entre las memorias cache, o entre la memoria global de la GPU y la memoria compartida por los hilos de un bloque.

Los siguientes cuadros muestran la sintaxis de los tipos de datos y funciones más características de CUDA:

[u]char[1..4] , [u]short[1..4] , [u]int[1..4] , [u]long[1..4] , float[1..4], dim3
Estructuras accedidas con los campos x , y , z , w: uint4 param; int y = param.y;
dim3: tipo de dato basado en uint3. Usado para especificar dimensiones.

Cuadro 2. Tipos de datos permitidos por CUDA y estructuras que se pueden formas con ellos.

dim3 gridDim; Dimensiones de la malla en bloques (gridDim.z sin uso).
dim3 blockDim; Dimensiones del bloque en hilos.
dim3 blockIdx; Índice del bloque dentro de la malla.
dim3 threadIdx; Índice del hilo dentro del bloque.

Cuadro 3. Variables de CUDA utilizadas para establecer las dimensiones de las mallas y los bloques y para acceder a los hilos.

Reserva de memoria en la GPU cudaMalloc(void** devPtr, size_t count) , cudaFree(void* devPtr) Copia de datos entre memoria. cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol() cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind) kind indica el flujo de los datos: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, o cudaMemcpyDeviceToDevice Direccionamiento de memoria cudaGetSymbolAddress()

Cuadro 4. Funciones para reserva de memoria y transferencia de datos entre ellas.

void __syncthreads(); Sincroniza todos los hilos de un bloque Una vez que todos los hilos han alcanzado este punto, la ejecución se reanuda normalmente. Utilizado para evitar problemas de acceso (RAW/WAR/WAW) a la memoria compartida o a la memoria global. Invocación de una función para su ejecución en la GPU: kernel_function<< dimGrid, dimBlock >>(...parameters....);

Cuadro 5. Método de sincronización de los hilos e invocación de una función para su ejecución en la GPU.

El siguiente cuadro muestra un ejemplo simple de una función programada para ejecutarse sobre la CPU y la misma función programada para la GPU:

```
//Cálculo de la serie y=ax +y sobre CPU (ejecución secuencial)
void axpy_secuencial(int n, float alpha, float *x, floar *y)
{
        for (int i=0; i<n; ++i)
                y[i] = alpha*x[i] + y[i];
}

//Invocación de la función axpy
axpy_secuencial(n,2.0,x,y);

//Cálculo de la serie y=ax +y sobre GPU (ejecución paralela)
__global__ void axpy_paralelo(int n, float alpha, float *x, floar *y)
{
        int i= blockIdx.x*blockDim.x + threadIdx.x;

        if(i<n) y[i] = alpha*x[i] + y[i];
}

//Invocación del núcleo axpy (256 hilos por bloque)
dim3 blockDim (256);
dim3 gridDim ((n + 255)/256);
axpy_paralelo<< gridDim, blockDim >>(n,2.0,x,y);
```

Cuadro 6. Ejemplo de algoritmo secuencial y su versión paralela implementada con CUDA.

3.4 Resumen del capítulo.

En este capítulo hemos podido conocer las peculiaridades de la tecnología CUDA y de las GPUs de NVIDIA.

En primer lugar hemos visto que para ejecutar una aplicación o una parte del código sobre la GPU esta debe ser descompuesta en bloques de hilos, los cuales se distribuirán entre los multiprocesadores de los que dispone la tarjeta. Los datos utilizados en el cálculo deben ser repartidos entre las diferentes memorias de la tarjeta (global, texturas, constante o compartida). Dependiendo del tipo de acceso y de la frecuencia en la que se realiza se debe utilizar una memoria u otra. La forma en la que se repartan los recursos de la tarjeta entre los hilos determinará la velocidad del cálculo de estos siendo este el punto más crítico.

Por último hemos visto que las APIs soportadas por CUDA para programar los hilos permiten utilizar lenguajes de programación general como por ejemplo C o C++. Esto es una de las principales ventajas que ofrece CUDA frente a otras metodologías de programación sobre GPUs. Estas interfaces aportan funciones propias, con una sintaxis muy clara, para manejar cada bloque y cada hilo y acceder a cada una de las memorias.

Capítulo 4

Nuestra elección: Ray Cast + CUDA

El objetivo que se pretende conseguir con este capítulo es mostrar el grupo de tecnologías y algoritmos utilizados para desarrollar el visualizador.

Como algoritmo de renderizado de volumen nos hemos decantado por la técnica del ray casting por dos motivos principalmente: porque es uno de los métodos que ofrece mejor calidad de imagen en sus resultados y porque, a pesar de no ser de los más rápidos, la técnica de lanzar un rayo por píxel de la imagen es altamente paralelizable, lo que permite programar el algoritmo con CUDA para acelerarlo con la GPU, consiguiendo así la velocidad de la que carece hasta ahora su implementación sobre CPU.

Como base para construir el visualizador se ha utilizado la herramienta VTK (Visualization Toolkit) [27]. VTK es un conjunto de librerías de código abierto desarrolladas en C++ que te permiten desarrollar un motor gráfico para la visualización y el tratamiento de imágenes. Este sistema ya dispone de librerías dedicadas al renderizado de volumen, utilizando concretamente los algoritmos de isosuperficies y ray cast, pero trabajando sobre CPU. El objetivo será sustituir estas librerías por nuestra versión ray cast con CUDA. A su vez nos permitirá comparar los resultados con la versión CPU.

4.1 Estructura y carga de los datos volumétricos

Nuestro visualizador debe ser capaz de cargar los datos generados por el escáner con el que trabaja el Dr. Herrero en el Hospital Militar, los cuales vienen en el formato DICOM. A pesar de que la clase vtkDICOMImageReader implementada en VTK es capaz de leer archivos en este formato, esta es bastante rudimentaria y no soporta conjuntos de datos DICOM multi-frame, por lo que hemos añadido al visualizador la librería GDCM [28] que es mucho más completa y nos permite ampliar los formatos de los datos de entrada soportados.

Dependiendo de la fuente de origen los datos pueden venir dados en una malla rectilínea cartesiana, una malla lineal curva o completamente desestructurada. Mientras que los dispositivos de escaneo generan mayoritariamente estructuras malladas rectilíneas (isotrópicas o anisotrópicas), las simulaciones físicas generan mayormente datos no estructurados. La figura 13 Ilustra estos diferentes tipos de malla para el caso 2D. Para las diferentes estructuras de malla diferentes algoritmos pueden ser usados para visualizar los datos volumétricos. Los datos obtenidos del Hospital Militar son mallas rectilíneas y por lo tanto nuestro algoritmo se basa en este tipo de estructura.

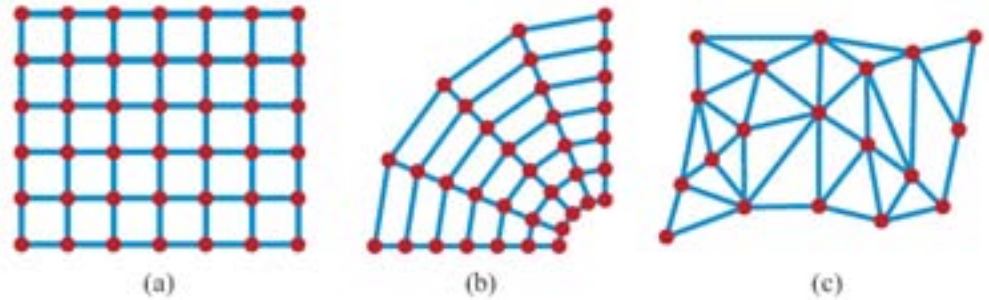


Figura 13. Diferentes estructuras de mallas: (a) Malla rectilínea. (b) Malla curvilínea. (c) Malla no estructurada [4].

Para organizar los datos en la memoria GPU hemos utilizado las librerías desarrolladas por Slicer3 [29], un sistema de visualización de datos médicos de código abierto, que tiene implementada una versión del algoritmo de isosuperficies acelerada por CUDA. Aunque el algoritmo es diferente, las librerías desarrolladas para cargar los datos en la GPU usando las funciones de CUDA es reutilizable. La forma en la que tratan los datos está basada en el trabajo de Grimm et al. [30] que se basa en dividir los datos en bloques o ”bricks” para que, en el caso de que el volumen de datos sea mayor al tamaño de la memoria, únicamente se carguen los bricks necesarios bajo demanda.

4.2 Interfaz de la función de transferencia

El algoritmo de renderizado necesita una función de transferencia que permita asignar unas propiedades de color y opacidad al punto de muestreo dependiendo de su valor escalar. Esto permite diferenciar por colores unos tejidos de otros dependiendo de su valor de intensidad en unidades Hounsfield. Para permitir que el usuario pueda cambiar esta función de transferencia de forma interactiva en tiempo real hemos utilizado KWWidgets [31], una herramienta de código abierto que permite crear gran cantidad de interfaces para usuarios.

La Figura 14 muestra el interfaz creado para nuestro visualizador en el que se puede determinar tanto numéricamente como a través de una barra deslizante el valor escalar y su color asociado (Scalar color mapping). Otra barra permite asociar diferentes valores de opacidad a un valor escalar concreto (Scalar opaciy mapping) y otra barra nos permite utilizar el valor del gradiente en lugar del valor escalar al cual se le asigna una opacidad

(Gradient opacity mapping). El panel Volume Appearance Settings permite elegir de forma rápida los valores del color en formato RGB.

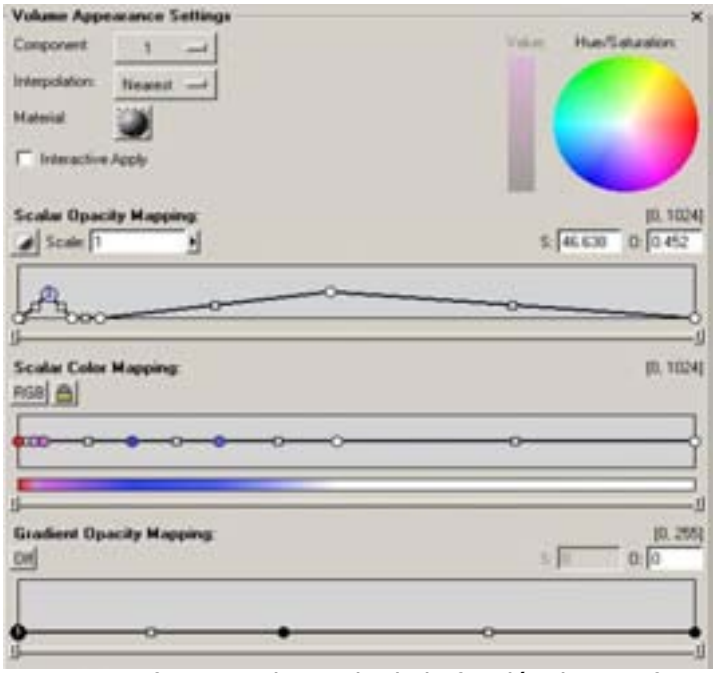


Figura 14. Interfaz para el manejo de la función de transferencia.

4.3 Pasos del algoritmo ray cast

Una vez que se han cargado los datos y se ha establecido la función de transferencia y los parámetros de la cámara, es turno del renderizado, es decir, de ejecutar el cálculo del algoritmo ray cast.

El siguiente cuadro muestra los principales pasos que sigue el algoritmo para cada rayo, o lo que es lo mismo, para cada píxel de la imagen final.

```
Cálculo de la dirección del rayo.
Cálculo del punto de intersección con el volumen y del punto de salida.
Comenzar a partir del punto de intersección.
while no se alcanza el punto de salida then
        Interpolación trilineal
        Obtener el valor de transparencia "alpha" (función de transferencia).
        if alpha > 0 then
                Cálculo del gradiente
                Composición de color
                if alcanzado el valor máximo de opacidad then
                        exit
        end if
        end if
        Calcular el nuevo punto de muestreo (sumar distancia de muestreo)
end while
Guardar el valor del píxel resultado
```

Cuadro 7. Algoritmo de ray cast ejecutado por cada hilo.

Para calcular el punto de intersección con el volumen de datos y el punto de salida, se utilizan los métodos implementados en la VTK para el cálculo de la intersección de un rayo con una bounding box. Una bounding box es el volumen que contiene al objeto de forma más ajustada. En este caso, como los datos de entrada tienen estructura rectilínea, la bounding box son los lados más externos del volumen. Esto nos permite comenzar a realizar el muestreo desde el punto de intersección con el volumen y no tener que muestrear el rayo desde el origen.

El orden en el que se realiza la composición del color a partir de los puntos de muestreo a lo largo del rayo es de delante a atrás, por lo que se aplica la ecuación (2.2) vista en el apartado 2.4. Esto nos permite, tal y como se puede ver en el algoritmo, realizar la técnica de finalización temprana del rayo y es que cuando se alcanza el valor máximo de opacidad, se da por concluido el muestreo a lo largo del rayo.

Siempre que se cambie alguno de los datos de entrada como pueden ser la función de transferencia, el ángulo de la cámara o los propios datos volumétricos, se tiene que ejecutar de nuevo el algoritmo de renderizado ray cast.

En los siguientes apartados se describe el método seguido para realizar la interpolación y el cálculo del gradiente.

4.4 Interpolación

A lo largo del recorrido del rayo los puntos de muestreo pocas veces coinciden con las posiciones actuales de la malla y requieren la interpolación del valor basándose en los valores de los puntos vecinos en la malla.

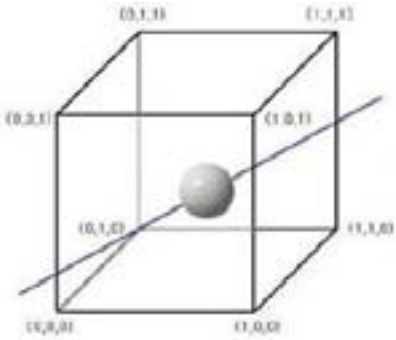
Los métodos de interpolación más usados son:

El vecino más cercano. Es el más simple y más rápido aunque el que peor resultado da. El valor del vóxel más cercano de todos los vóxel vecinos es asignado al punto de muestreo. La calidad de la imagen es bastante baja y a grandes resoluciones aparece muy pixelada.

Interpolación trilineal. Este método asume una relación linear entre los vóxeles vecinos, por lo tanto, puede descomponerse en 7 interpolaciones lineales (ver figura 15). La calidad de la imagen es mayor que con la del vecino más cercano aunque también aumenta el coste computacional. Sin embargo, a grandes resoluciones se pueden ver que aparecen como estructuras tridimensionales de diamante o aspas debido a la naturaleza de la máscara trilineal.

Mayor calidad que con los anteriores se puede conseguir utilizando métodos de interpolación de un orden mayor como la convolución cúbica o la interpolación b-spline. Sin embargo se busca un equilibrio entre calidad y coste computacional así como consumo de ancho de banda de memoria. Estos

filtros requieren un vecindario de 64 vóxeles y una gran cantidad de cálculos en comparación con la interpolación trilineal.



$$V_{xyz} = V_{000}(1-x)(1-y)(1-z) + V_{100}x(1-y)(1-z) +$$
$$V_{010}(1-x)y(1-z) + V_{001}(1-x)(1-y)z + V_{101}x(1-y)z +$$
$$V_{011}(1-x)yz + V_{110}xy(1-z) + V_{111}xyz$$

Figura 15. Método de interpolación trilineal.

Un aspecto importante a tener en cuenta, al que hacen referencia Meibner et al. [23], es que no se obtiene el mismo resultado si se interpolan los valores escalares y al resultado se le aplica un color y opacidad (esto provoca efectos de aliasing), o si a cada punto vecino se le aplica el color y opacidad correspondiente a su valor escalar, y se interpola con esos valores de color y opacidad en lugar de con el escalar (esto provoca efectos de mezcla de colores). Tampoco da el mismo resultado si se interpola por separado el color y la opacidad o si se interpola el color compuesto con la opacidad. Este último método es el más conveniente.

En este proyecto se han implementado los dos primeros métodos de interpolación.

4.5 Cálculo del gradiente

Para cada punto muestreado se calcula también su gradiente que podrá usarse para aplicar sobre él las técnicas de iluminación y sombreado.

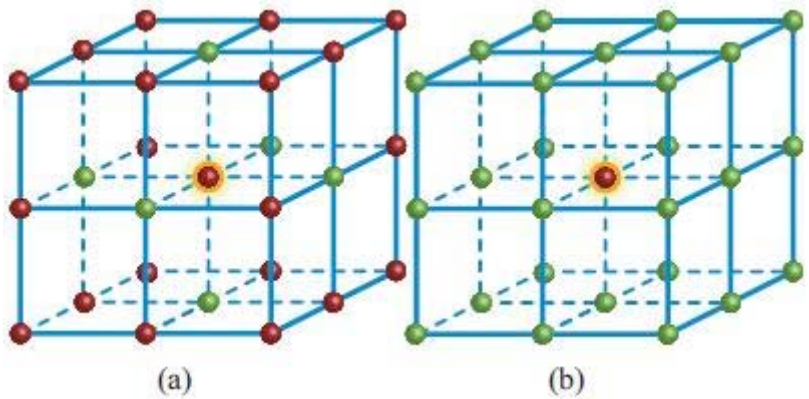


Figura 16. (a) Gradiente de diferencias centrales. (b) Gradiente de Neumann. Las esferas verdes son los puntos de entrada para los diferentes métodos de estimación del gradiente [4].

El método utilizado es el de gradiente de diferencias centrales (ver figura 16 a) a partir de los valores de los 6 vóxeles vecinos, tal y como indica la siguiente ecuación:

$$\begin{aligned} \nabla_x &= V_{x+1,y,z} - V_{x-1,y,z} \\ \nabla_y &= V_{x,y+1,z} - V_{x,y-1,z} \\ \nabla_z &= V_{x,y,z+1} - V_{x,y,z-1} \end{aligned} \qquad (4.1)$$

Existen otros métodos más precisos como el de Neumann et al. [32] el cual se basa en coger los valores de 26 vóxeles vecinos (ver figura 16 b) y aplicar una técnica lineal de regresión, pero al igual que con el método de interpolación empleado, se busca un compromiso entre calidad y velocidad por lo que hemos elegido el de diferencias centrales.

En este proyecto no se aplican todavía técnicas de iluminación al resultado final, pero el hecho de realizar estos cálculos nos permiten poder realizar comparaciones de tiempo con respecto a la versión CPU que también realiza el cálculo del gradiente.

4.6 Resumen del capítulo.

Al llegar a este punto hemos visto todos los algoritmos y tecnologías empleadas para desarrollar el visualizador.

La librería VTK nos sirve como soporte para crear el visualizador a la que se añaden la librería GDCM para cargar archivos DICOM, de forma que el Dr. Herrero pueda cargar los resultados de su trabajo en el sistema, y la librería KWWidgets que aporta el interfaz para manejar la función de transferencia y permite segmentar los datos por su valor de intensidad.

El algoritmo de renderizado elegido ha sido ray cast, porque la calidad de la imagen es buena y porque CUDA nos permite acelerar el cálculo realizado por cada rayo. Parte de este cálculo es el método de interpolación trilineal, usado para calcular el valor escalar del punto de muestreo del rayo, y el método de diferencias centrales, utilizado para calcular el gradiente, que aunque no se está aplicando ninguna técnica de iluminación, nos permite poder comparar los tiempos de ejecución de nuestro algoritmo con el resto que también calcula el gradiente.

Capítulo 5

Resultados y comparativa

Este capítulo pretende mostrar los resultados obtenidos en lo referente al aspecto visual del sistema, así como en la velocidad del cálculo de una imagen. Para esto último se presentan los tiempos obtenidos de una comparación con otros algoritmos de renderizado. Dichos resultados permiten ver las limitaciones del sistema y se citarán las técnicas disponibles para superarlas.

5.1 Resultados

Juntando todos los algoritmos y arquitecturas descritos en los apartados anteriores hemos desarrollado un módulo independiente de visualización de datos volumétricos, el cual permite aplicar diferentes funciones de transferencia para mostrar los datos tal y como se puede ver en las siguientes imágenes.

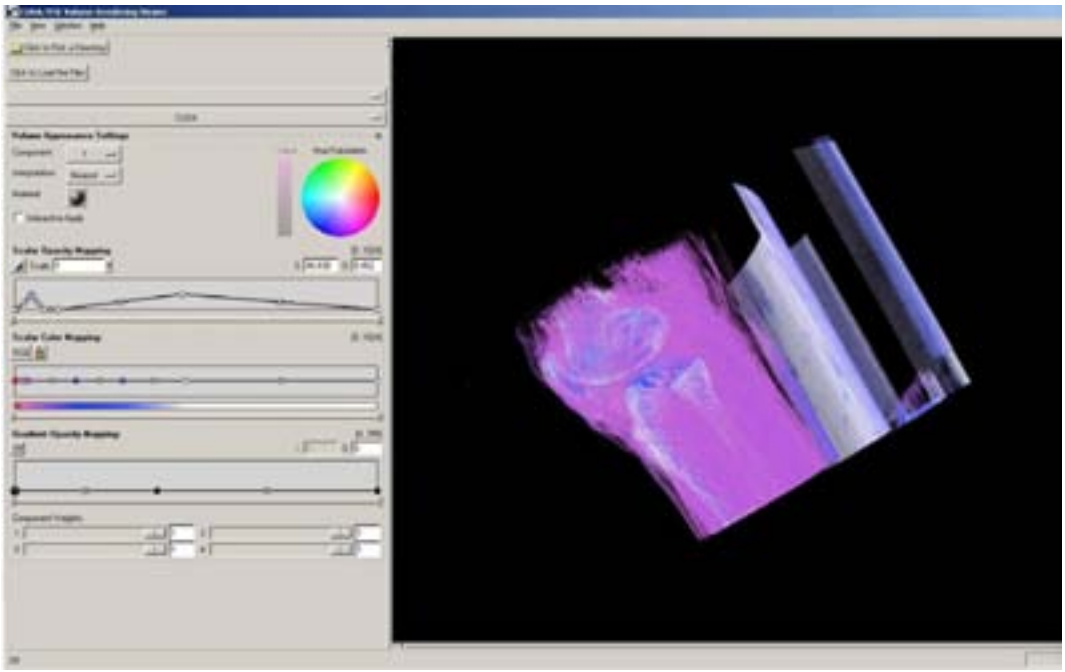


Figura 17. Captura del visualizador mostrando el tejido óseo y el tejido muscular de un CT de una rodilla de un paciente.

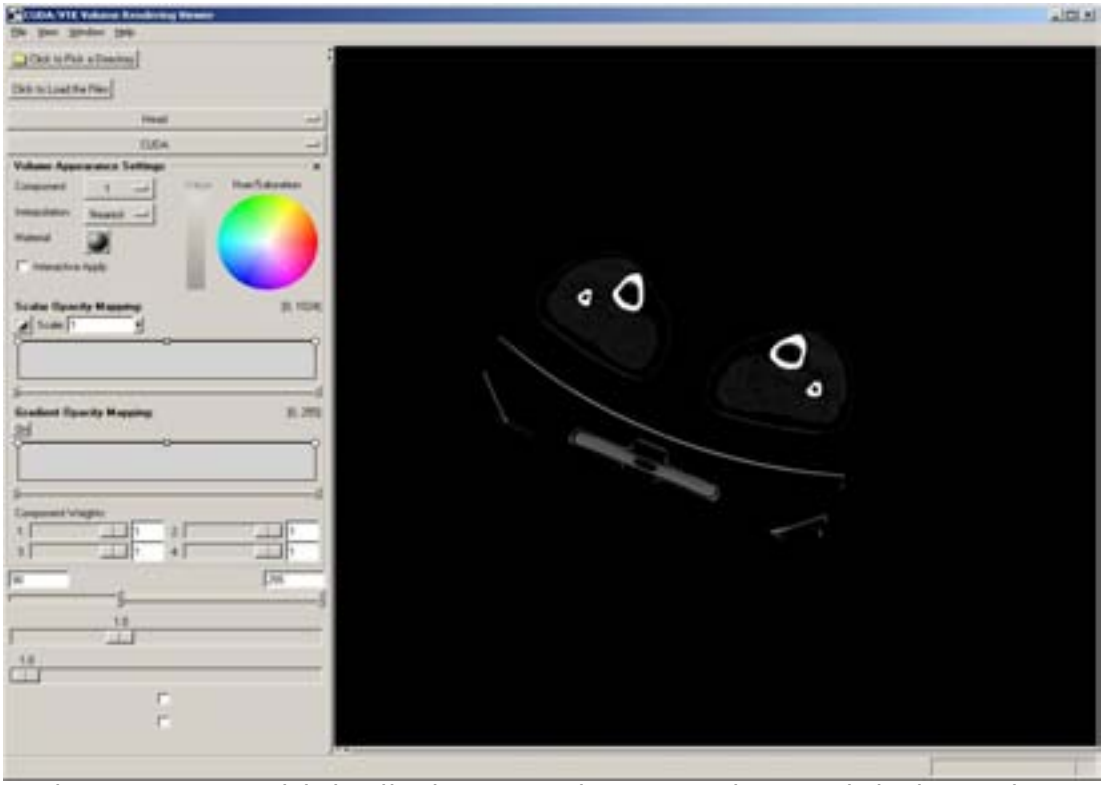


Figura 18. Captura del visualizador mostrando un corte de un CT de los brazos de un paciente.

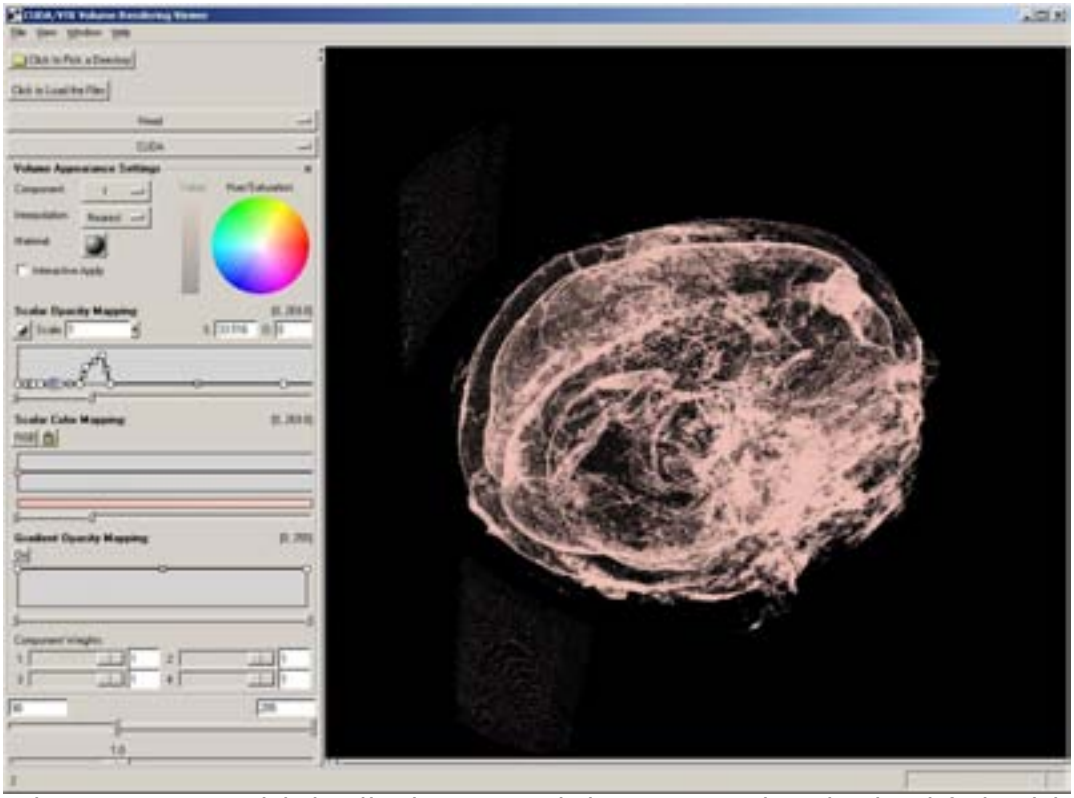


Figura 19. Captura del visualizador mostrando los contornos de ambos hemisferios del cerebro de un paciente.

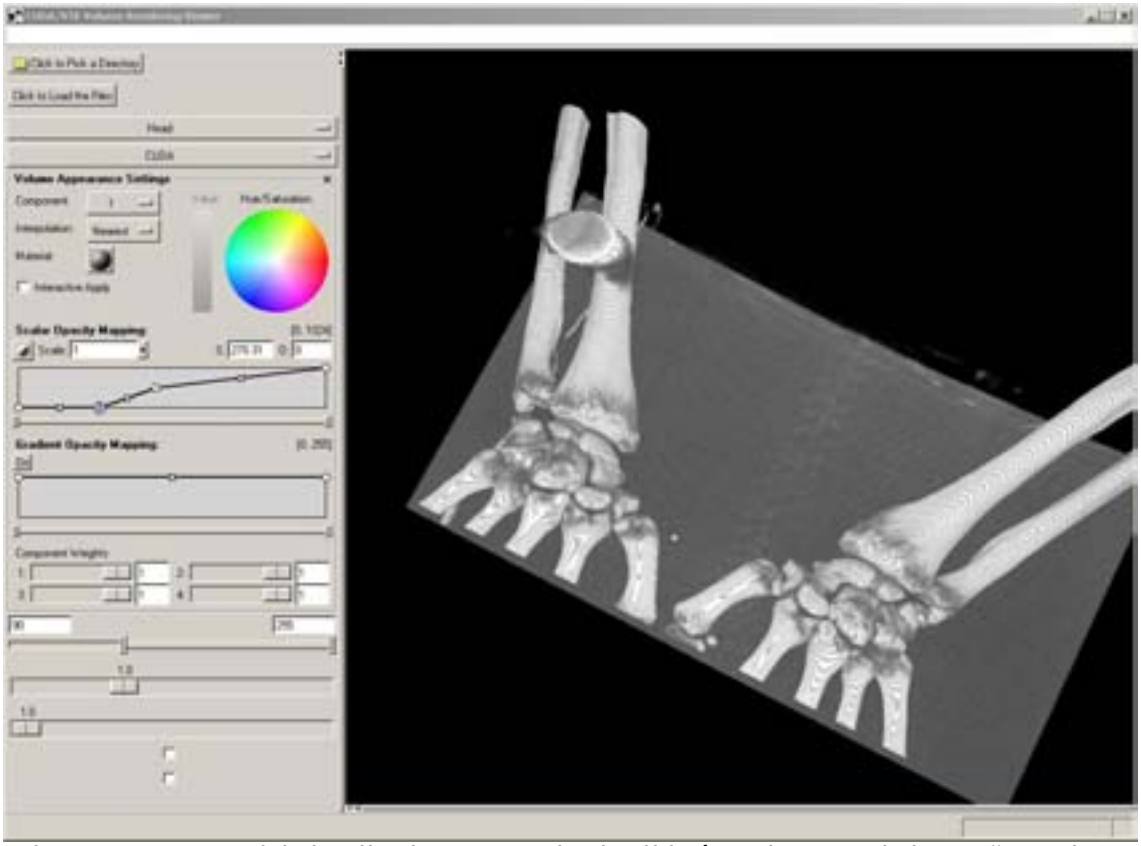


Figura 20. Captura del visualizador mostrando el tejido óseo de un CT de las muñecas de un paciente.

5.2 Comparativa

Para comprobar la eficacia de la aceleración con CUDA hemos elegido una serie de algoritmos contra los que poder comparar tiempos. Lógicamente el primero debía ser obligadamente la versión del algoritmo ray cast para CPU implementado por VTK. En segundo lugar hemos realizado mediciones de tiempo con el algoritmo de isosuperficies para CPU implementado también por la VTK. Por último, otro método interesante con el que hemos hecho comparaciones es el algoritmo de isosuperficies acelerado con CUDA implementado en Slicer3.

Para cada uno de estos métodos se han realizado mediciones del tiempo de cálculo de un frame por el algoritmo de renderizado. Aunque los algoritmos de ray cast son diferentes, realizan básicamente los mismos pasos, y para aquellos cálculos que no realiza nuestro algoritmo, como aplicar sombreado e iluminación, se ha restado el tiempo dedicado a estos aspectos del tiempo global del ciclo. Las mediciones de tiempos de los algoritmos de isosuperficies nos permiten hacernos una idea de las diferencias de tiempo con el resto de algoritmos de renderizado de volumen.

Las siguientes tablas muestran los resultados de las mediciones de tiempos llevadas a cabo para cada uno de los algoritmos usando diferentes

tamaños del volumen de datos y diferentes resoluciones. Con resolución nos referimos al número de píxeles de la pantalla que muestran una parte del volumen.

Algoritmo	Implementación	Tiempo (seg.)	Imágenes/seg.	Aceleración
ray cast	CPU (OpenGL y C++) VTK	1,393+-0,07	0,718	1,0
isosuperficies	CPU (OpenGL y C++) VTK	0,311+-0,012	3,215	4,5
ray cast	GPU (CUDA)	0,168+-0,015	5,952	8,3
isosuperficies	GPU (CUDA) Slicer3	0,126+-0,011	7,936	11,0

Tabla 3. Mediciones de tiempos resolución 425x400 píxeles, tamaño de los datos 160MB.

La tabla 3 muestra los tiempos obtenidos con una resolución y un volumen de datos pequeño. Por los tiempos obtenidos se puede ver que nuestro algoritmo CUDA de ray cast es 8 veces más rápido que el algoritmo implementado para trabajar sobre CPU. Así mismo, también se puede comprobar que los algoritmos de isosuperficies, tanto para CPU como GPU, son más rápidos que el algoritmo de ray cast, tal y como ya se apuntaba en el apartado 2.1.

Algoritmo	Implementación	Tiempo (seg.)	Imágenes/seg.	Aceleración
ray cast	CPU (OpenGL y C++) VTK	4,392+-0,5	0,227	1,0
isosuperficies	CPU (OpenGL y C++) VTK	1,098+-0,2	0,910	4.0
ray cast	GPU (CUDA)	0,665+-0,28	1,503	6,6
isosuperficies	GPU (CUDA) Slicer3	0,395+-0,11	2,531	11,1

Tabla 4. Mediciones de tiempos resolución: 820x750 píxeles, tamaño de los datos: 160MB.

La tabla 4 muestra los tiempos utilizando el mismo volumen de datos pero con una resolución mayor que la anterior (4 veces más grande). Al aumentar el número de píxeles se aumenta el número de rayos lanzados y con ello el cálculo necesario para procesar una imagen. Puede observarse cómo se produce un aumento del tiempo de cálculo de un frame para todos los algoritmos prácticamente en la misma proporción (ver Aceleración).

Algoritmo	Implementación	Tiempo (seg.)	Imágenes/seg.	Aceleración
ray cast	CPU (OpenGL y C++) VTK	No Soportado	No Soportado	0,0
isosuperficies	CPU (OpenGL y C++) VTK	No Soportado	No Soportado	0,0
ray cast	GPU (CUDA)	0,711+-0.3	1,977	1,0
isosuperficies	GPU (CUDA) Slicer3	0,371+-0.2	2,695	1,3

Tabla 5. Mediciones de tiempos resolución: 475x370 píxeles, tamaño de los datos: 501MB.

En esta nueva prueba se ha utilizado de nuevo una resolución pequeña pero aumentando por 4 el tamaño del volumen de los datos. Los algoritmos implementados por la VTK, tanto de ray cast como de isosuperficies, son incapaces de manejar tal cantidad de datos. En cambio, nuestro algoritmo de ray cast para CUDA está preparado para cargar cualquier volumen de datos

independientemente de su tamaño, pero también se ve mermada su velocidad debido a que durante el cálculo aumentan los fallos de acceso a memoria cache, obligando a realizar costosas transferencias de datos entre las diferentes memorias de la GPU, tal y como se ha visto en el apartado 3.1.

Algoritmo	Implementación	Tiempo (seg.)	Imágenes/seg.	Aceleración
ray cast	CPU (OpenGL y C++) VTK	No Soportado	No Soportado	0,0
isosuperficies	CPU (OpenGL y C++) VTK	No Soportado	No Soportado	0,0
ray cast	GPU (CUDA)	3,496+-0.5	0,286	1,0
isosuperficies	GPU (CUDA) Slicer3	1.412+-0.15	0,708	2,4

Tabla 6. Mediciones de tiempos resolución: 880x630 píxeles, tamaño de los datos: 501MB.

La tabla 6 muestra el peor escenario posible de pruebas con una resolución y un volumen de datos grande. Un tiempo de 3.5 segundos para calcular una imagen es excesivo. Nuestro método es en el peor de los casos hasta 6 veces más rápido que la versión CPU, pero aun así, todavía se necesita reducir el tiempo de ciclo, tal y como reflejan los resultados de las tablas 5 y 6, para aumentar el frame rate hasta un valor que permita la interactividad (p. e. 10 imágenes/segundo).

Resumiendo lo visto en las tablas anteriores, el hecho de aumentar la resolución, provoca que haya un mayor número de hilos que intersectan los datos volumétricos y que deben calcular un valor de color para el píxel, y eso explica que para resoluciones mayores el tiempo de ciclo aumente. Tanto si aumenta el número de hilos que tienen que acceder a los datos, como si aumenta el volumen de los datos, se produce un aumento del tiempo de cálculo. Esto es debido a que aumenta el número de fallos de acceso a los datos en cache y el consiguiente acceso lento a la memoria de texturas o memoria global de la GPU.

Una forma de reducir el tiempo de ciclo es evitar estos continuos fallos de acceso a la memoria cache. Para ello, el método que se pondrá en práctica es el indicado por Hadwiger et al. [33], que propone la división del volumen de datos en bricks, que ya se está utilizando para la carga de los datos en memoria, pero en esta ocasión reduciremos todavía más el tamaño de estos bricks, de forma que todos los hilos de un bloque tengan los datos del brick que se está procesando disponibles en la memoria cache de la memoria de texturas. En el momento en el que estos hilos necesitan acceder a un nuevo brick, este es cargado desde la memoria global. Hay que ajustar el tamaño de los bricks de forma que sólo se necesite hacer un número pequeño de transferencias de datos entre la memoria global y la memoria de texturas o constante. Aunque en principio esta transferencia es lenta, compensa si se permite reducir drásticamente los fallos de acceso a cache de la memoria de texturas o de la memoria constante.

Otra forma de aumentar la sensación de velocidad es reducir la calidad de la imagen mostrada mientras que se realiza un movimiento de cámara, y

cuando ésta se para, se genera la imagen con la mayor calidad posible. Algunas de estas técnicas para aumentar la velocidad reduciendo la calidad se basan en lanzar un hilo para calcular el valor de un píxel, mientras que el valor de los píxeles vecinos se calcula por interpolación. Otra técnica consiste en aumentar la distancia de muestreo del rayo, reduciendo así el tiempo de procesamiento de cada hilo.

Estos métodos no dejan de ser trucos para mejorar la sensación de interactividad. Lo ideal sería que, tanto cuando se está moviendo la cámara, como cuando está fija, la calidad de la imagen sea siempre la máxima posible y eso lo que se ha buscado en este proyecto.

Capítulo 6

Conclusiones, coste temporal y trabajo futuro

6.1 Conclusiones

El objetivo principal que nos planteábamos al principio de este proyecto era desarrollar un sistema de visualización de datos biomédicos que permitiese al usuario trabajar con los datos en tiempo real, así como realizar operaciones simples de segmentación. Este sistema debía ser capaz además de cargar los archivos obtenidos del trabajo realizado por el Dr. Herrero en el hospital militar.

Nuestra elección del ray cast como algoritmo de renderizado frente al resto, por su calidad de imagen y por su método de trabajo con rayos altamente paralelizable, juntándolo con el uso de la tecnología CUDA y las librerías VTK y KWWidgets ha resultado muy acertada, puesto que se ha desarrollado un visualizador que llega a multiplicar en el peor de los casos hasta por 6 la velocidad de renderizado de las actuales aplicaciones software. El visualizador dispone además de un interfaz que permite modificar la función de transferencia en tiempo real de forma que el usuario puede elegir que datos mostrar o no y con que color.

Aunque el sistema puede cargar diferentes tipos de datos, está especialmente preparado para cargar datos volumétricos cuyo formato de entrada sea en archivos DICOM. Esto permitirá cargar los datos originales procedentes de CT y MRI, o estos mismos datos una vez que han sido segmentados y modificados por un radiólogo o especialista. Esto permite cubrir el último de los objetivos propuestos por lo que puedo afirmar que se han alcanzado plenamente cada uno de ellos.

Este sistema podrá ser utilizado tanto para presentar los resultados del escáner al médico o cirujano así como para mostrar imágenes en clases de medicina. Para esto último se va a trabajar con el Dr. Herrero que introducirá el sistema en sus clases de la facultad de medicina de la Universidad de Zaragoza, donde podrá mostrar datos que extrae de su propio trabajo como

radiólogo en el Hospital Militar. Con esto se pretende familiarizar a los estudiantes con los beneficios del 3D, presentarles los datos de una manera más realista e interactiva y hacer las clases más amenas y entretenidas.

6.2 Coste temporal

El periodo de realización de este proyecto comprenden los meses que van de Febrero del 2008 a Agosto del 2009. En la figura 21 se muestra un gráfico con la distribución temporal de cada una de las fases de desarrollo del proyecto.

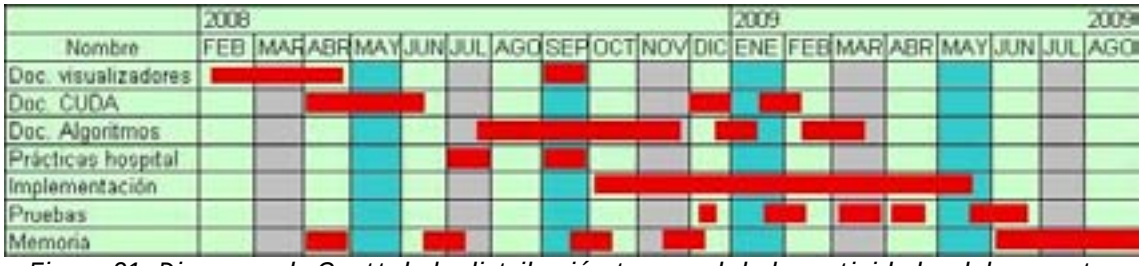


Figura 21. Diagrama de Gantt de la distribución temporal de las actividades del proyecto.

Durante los primeros meses y coincidiendo con las últimas clases del master se revisaron los visualizadores disponibles en el mercado en aquel momento y comenzó el aprendizaje de la tecnología CUDA. De todos los sistemas la VTK era el más prometedor, pero puesto que su algoritmo era lento se tomó la decisión de cambiarlo y comenzó el estudio de todos los posibles algoritmos de renderizado de volumen.

En los meses de Julio y Septiembre se realizaron las prácticas en el hospital con el Dr. Herrero de donde se sacaron los requisitos funcionales que debía tener nuestro visualizador. Una vez elegido ray cast como algoritmo de renderizado comenzó la implementación del mismo.

Este periodo de implementación ha sido el más largo de todos y el más complejo. Fue el momento de programar con CUDA el algoritmo y esto exigió continuas pruebas y nuevas búsquedas de documentación acerca de métodos que permitiesen ahorrar tiempo en el muestreo de los rayos, la interpolación, el cálculo del gradiente o la función de transferencia entre otros. Programar en CUDA es sencillo pero ajustar el uso optimizado de los recursos de la tarjeta es un trabajo minucioso que exige realizar cuantiosos ensayos.

Al finalizar cada fase de documentación se iban redactando notas y resúmenes para ir completando esta memoria, aunque han sido los últimos meses donde se le ha dado forma a cada uno de los capítulos.

6.3 Trabajo futuro

A pesar de que se ha llegado a superar la velocidad de renderizado de

otros sistemas, se va a seguir trabajando en intentar reducir todavía más el tiempo de ciclo del algoritmo de renderizado, para permitir aumentar la interactividad con el sistema. Así mismo se trabajará para mejorar la carga y el manejo de los datos para que al utilizar volúmenes de datos más pesados no se vean reducidas considerablemente las prestaciones del sistema.

Como trabajo futuro el visualizador se integrará en un motor gráfico más potente que permitirá añadir técnicas de iluminación, cropping, clipping planes, sombreado y estereoscopía. Lo que permitirá mejorar el aspecto visual y la calidad de la imagen, así como explotar los beneficios del 3D frente al 2D. Además, dicho motor está preparado para hacer el seguimiento de la orientación y posición de la cabeza y el reconocimiento de órdenes gestuales, lo que permitirá utilizar diferentes interfaces más intuitivos.

Referencias

[1] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer and L. Seiler. "The VolumePro real-time ray-casting system". In SIGGRAPH, pp. 251–260, 1999.

[2] M. Meissner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Strasser, M. Doggett, P. Forthmann and R. Proksa. "Vizard II, a reconfigurable interactive volume rendering system". In Eurographics Workshop on Graphics Hardware, pp. 137–146, 2002.

[3] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl W. and Strasser. "Smart hardware-accelerated volume rendering". In Symposium on Data Visualization, pp. 231–238, 2003.

[4] S. Grimm, "Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware". PhD thesis, Institute of Computer Graphics and Algorithms, University of Vienna. April 2005.

[5] R. Shen, P. Boulanger and M. Noga, "MedVis: A Real-Time Immersive Visualization Environment for the Exploration of Medical Volumetric Data". IEEE Fifth International Conference BioMedical Visualization. Volume 9-11, p. 63 – 68. July 2008.

[6] CUDA Zone, 2007, http://www.nvidia.es/object/cuda_home_es.html.

[7] GLSL, http://www.opengl.org/documentation/glsl/.

[8] OpenCL, http://www.khronos.org/.

[9] HLSL, http://msdn.microsoft.com/en-us/library.

[10] Steven G. Parker, PhD http://www.cs.utah.edu/~sparker/images.html.

[11] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". Computer Graphics (SIGGRAPH 87 Proceedings) Volume 21 (Nº 4) p. 163-170, July 1987.

[12] D. Lingrand and A. Charnoz, courses from ESSI (Ecole Supérieure en Sciences Informatiques) http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html.

[13] G. G. Cameron, y P. E. Undrill, "Rendering volumetric medical image data on a SIMD architecture computer". Eurographics Workshop on Rendering (1992), pp. 135–145.

[14] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation". SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques, July 1994.

[15] Course of the Institute of Computer Graphics and Algorithms. Vienna University of Technology http://artoolkit.org/courses/Visualisierung/2002-2003/Beispiel2/2-BrucknerSt-SeemannR/index.htm.

[16] C. Lin, D. Yang and Y. Chung, "Parallel Shear-Warp Factorization Volume Rendering Using Efficient 1-D and 2-D Partitioning Schemes for Distributed Memory Multicomputers". The Journal of Supercomputing, Volume 22, Nº 3, P. 277-302, July 2002.

[17] L. Westover, "Footprint evaluation for volume rendering". SIGGRAPH Computer Graphics 24, 4, 367–376, 1990.

[18] L. Westover, "SPLATTING: A Parallel Feed-Forward Volume Rendering Algorithm". PhD thesis, University of North Carolina at Chapel Hill, 1991.

[19] W. Krueger, "The application of transport theory to visualization of 3D scalar data fields". In IEEE Visualization, pp. 273–280, 1990.

[20] N. Max, "Optical models for direct volume rendering". IEEE Transactions on Visualization and Computer Graphics 1, 2, 99–108, 1995.

[21] G. Knittel, "The Ultravis system". In IEEE Symposium on Volume visualization, pp. 71–79, 2000.

[22] M. Sramek, A. Kaufman, "Fast ray-tracing of rectilinear volume data using distance transforms". IEEE Transactions on Visualization and Computer Graphics 6, 3, 236–252, 2000.

[23] M. Meißner, H. Pfister, R. Westermann, and C.M. Wittenbrink "Volume Visualization and Volume Rendering Techniques". EUROGRAPHICS 2000.

[24] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable parallel programming with CUDA". SIGGRAPH 2008 classes, August 2008.

[25] D. Kirk (NVIDIA) and Wen-mei W.Hwu, CUDA course. University of Illinois at Urbana-Champaign, 2007.

[26] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA". ACM, February, 2008.

[27] VTK "Visualization Toolkit". Main page: http://www.vtk.org/.

[28] GDCM Grassroots DICOM library. Main page: http://sourceforge.net/apps/mediawiki/gdcm/index.php.

[29] 3D Slicer http://www.slicer.org/.

[30] S. Grimm, S. Bruckner, A. Kanitsar and E. Gröller, "A Refined Data Addressing and Processing Scheme to Accelerate Volume Raycasting", Computers & Graphics, 28(5), pp. 719-729, 2004.

[31] KWWidgets http://www.kwwidgets.org.

[32] L. Neumann, B. Csebfalvi, A. König, and E. Gröller, "Gradient estimation in volume data using 4D linear regression". Computer Graphics Forum 19, 3, p. 351–358, 2000.

[33] M. Hadwiger, P. Ljung, C. Salama and T. Ropinski "Advanced Illumination Techniques for GPU-Based Volume Raycasting". ACM Siggraph Courses, 2009.