UNIVERSIDAD CENTRAL DE VENEZUELA

FACULTAD DE CIENCIAS

ESCUELA DE COMPUTACIÓN

Visualización Volumétrica

Utilizando Mallado Tetraédrico

Trabajo Especial de Grado presentado ante la ilustre
Universidad Central de Venezuela por el bachiller
Danny R. Monsalve Páez
Para optar al título de Licenciado en Computación

Tutor
Prof. Rhadamés Carmona

Octubre 2016

Caracas, 24 de Octubre de 2016
Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Danny Rafael Monsalve Páez C.I.: V-10.542.220, con el título "Visualización Volumétrica utilizando Mallado Tetraédrico", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 24 de Octubre de 2016, a las 9am, para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, lo cual este realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo con una calificación de diecinueve (19) puntos.

En fe de lo cual se levanta la presente acta, en Caracas el lunes veinticuatro (24) de octubre de 2016 dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Tutor Rhadamés Carmona.

Prof. Rhadamés Carmona
(Tutor)

Prof. Luis Hernández
(Jurado Principal)

Prof. Francisco Moreno
(Jurado Principal)

RESUMEN

Los dos métodos principales para la visualización de datos volumétricos consisten en la visualización directa de volúmenes, y la visualización indirecta de volúmenes. El primer método ofrece la ventaja de mostrar todo el volumen en contexto mediante la proyección de las muestras del mismo, mientras que el segundo método sólo muestra una parte del volumen fuera de contexto, y requiere de una etapa previa de reconstrucción de isosuperficies.

En el CCG se propuso e implementó un algoritmo basado en la adaptación de cubos marchantes para extraer intervalos de volumen en lugar de isosuperficies, el cual procesa cada celda directamente mediante una tabla de conectividad, sin necesidad de dividir cada celda en tetraedros ni de recurrir a algoritmos complejos durante el procesamiento de cada celda. El resultado del algoritmo fue la generación de un mallado tetraédrico opaco que mostraba la superficie del subintervalo del volumen en cuestión.

Estos aportes al área de la visualización de volúmenes utilizando como unidad de volumen un tetraedro, atrajo el interés en realizar este trabajo de grado, el cual va a tener por objetivo implementar la visualización volumétrica de un mallado tetraédrico utilizando el hardware gráfico convencional para evaluar la calidad y el desempeño del *rendering*. Específicamente se implementó y evaluó la técnica visualización directa de volúmenes basados en el algoritmo de Tetraedro Proyectado (PT). El método que se presenta aproxima las celdas tetraédricas de un volumen con triángulos semi-transparentes, desplegados con el hardware gráfico moderno.

Palabras claves: reconstrucción 3D, mallas tetraédricas, vóxel, rendering.

TABLA DE CONTENIDOS

Introducción iv

Capítulo 1. Marco Teórico 1

1. Voxel y Volumen 2

2. Despliegue Directo de Volúmenes 5

3. Pipeline de Visualización de Volúmenes 7

4. Discretización de la Ecuación de Visualización de Volúmenes .. 9

5. Algoritmos para el Despliegue de Volúmenes 11

6. Volume Rendering Basado en Texturas 2D 12

7. Proyección de Celdas 16

Capítulo 2. Algoritmo Tetraedro Proyectado (PT) 18

Integración y Rendering 21

Capítulo 3. Diseño e Implementación 24

1. Detalles de Diseño 24

1.1. Abrir Archivo de Volumen 26

1.2. Procesamiento de los datos 28

1.3. Desplegar Volumen 39

2. Diagrama del Programa 41

3. Programa sin la función optimizada ‘glMultiDrawElements’ 46

3.1. Pipeline Gráfico y Nueva Solución 47

Capítulo 4. PRUEBAS 52

1. Descripción del ambiente de pruebas 52

2. Resultados cuantitativos 53

3. Resultados cualitativos 57

Capítulo 5. Conclusiones y Trabajos futuros 59

Referencias 62

INTRODUCCIÓN

En computación gráfica, *rendering* es el proceso de producción de imagen en la pantalla a partir de la descripción de un modelo u objeto.

La visualización de volúmenes o *volume rendering,* es el proceso mediante el cual es posible hacer una representación visual bidimensional de datos discretos que pertenecen a un espacio tridimensional. Cada uno de estos datos representa información escalar o vectorial de un fenómeno, proceso u objeto que se quiere visualizar [6] [1]. Las aplicaciones principales de esta técnica varían desde la visualización de datos médicos, datos geológicos y dinámica de fluidos.

La carga computacional de *volume rendering* depende directamente del tamaño del volumen de datos de entrada. Un volumen de datos puede llegar a ocupar desde unas pocas decenas de megabytes – como volúmenes de datos médicos – hasta unos cuantos gigabytes – volúmenes para la exploración de datos geológicos – [6]. Muchas técnicas se han desarrollado para disminuir el problema del espacio y el procesamiento, con enfoques basados en el cambio del dominio de los datos, representación basada en datos comprimidos, cómputo paralelo, operaciones que reducen elementos de un volumen en una etapa previa al procesamiento y el uso de memoria de texturas para el procesamiento de los datos.

Hasta hace un pocos más de una década, el poder de cómputo de los ordenadores de consumo masivo no era suficiente para hacer *volume rendering* en tiempo real. Con la reducción en los costos de

producción y avances en el desarrollo de nuevas tecnologías ya es posible lograr este cometido en tiempos muy razonables y con un alto nivel de calidad.

Un volumen discreto por lo general es un conjunto de muestras dispuestas en una malla estructurada tridimensional adquiridas de un objeto físico. Los mallados estructurados son topológicamente equivalentes al enrejado de números enteros, y como tal, fácilmente pueden ser representados por un arreglo o matriz 3D. El mapeo desde los elementos del arreglo a puntos de muestra y la relación de conectividad entre las celdas son implícitos. En este tipo de mallas, los datos vienen dado como un conjunto de muestras de una función escalar continua, representado por una malla regular, y almacenado en un arreglo tridimensional de escalares [4, 32]. Por otra parte, la distribución de los puntos de muestreo no sigue un patrón regular en las mallas no estructuradas y pueden existir vacíos en el mallado. En los datos no estructurados, los datos de volumen se pueden localizar escasamente, es decir, los valores escalares, vectoriales como la velocidad, el calor, etc. pueden estar en cualquier punto en el espacio. Los mallados no estructurados también se llaman mallados de *celdas-orientadas* debido a que estas mallas están representadas por una lista de celdas en la cual cada celda contiene punteros a los puntos de muestra en la celda. Debido a la naturaleza de las celdas-orientadas y la irregularidad de los mallados no estructurados, la información de conectividad se proporciona de forma explícita [35]. Los mallados irregulares (o no estructurados) se utilizan normalmente en la

visualización científica como la inspección geológica, la simulación de fluidos, entre otras aplicaciones.

Para visualizar un volumen, hay dos categorías. Métodos directos e indirectos. El primero consiste en proyectar directamente el conjunto de muestras o celdas hacia el plano imagen, para así visualizar el volumen sin necesidad de requerir de reconstrucciones intermedias [2]. El segundo requiere de la reconstrucción de una superficie intermedia (ej. Marching Cubes [15]), o de la reconstrucción de tetraedros (ej. reconstrucción de intervalos de volumen [34, 32]).

En el Centro de Computación Gráfica (CCG) se propuso e implementó un algoritmo basado en la adaptación de cubos marchantes para extraer intervalos de volumen en lugar de isosuperficies. Este algoritmo requiere de un mallado estructurado como entrada. Procesa cada celda del volumen directamente mediante una tabla de conectividad [32]. La salida del algoritmo es un mallado tetraédrico; sin embargo, no se cuenta en el CCG con un visualizador de mallas tetraédricas, lo que limitó la visualización a simplemente el despliegue de los tetraedros como un conjunto de triángulos. El resultado es que podemos ver la superficie opaca del volumen generado, y no sus estructuras internas, pues se ignoran las propiedades de absorción y emisión de los datos volumétrico. Resulta así de interés implementar un visualizador de volúmenes que soporte este tipo de mallas.

En este trabajo nos enfocamos en la visualización de mallas tetraédricas. Existen métodos que realizan una visualización directa de volumen proyectando cada celda o unidad de volumen como polígonos, los cuales son desplegados por el hardware gráfico [21, 22,

36]. Por tanto el volumen puede representarse como un conjunto de tetraedros, bien sea por la subdivisión de las celdas en tetraedros (cuando el mallado es estructurado), o conectando las muestras formando tetraedros (cuando el mallado no es estructurado). Para la visualización de estas mallas tetraédricas se utiliza típicamente la técnica de tetraedros proyectados (*PT*). Esta técnica consiste en proyectar los tetraedros al plano de imagen y hacer la composición en un orden de visibilidad.

Este algoritmo se puede aplicar a tanto a datos estructurados como no estructurados, ya que las *unidades de volumen* de tamaño uniforme pueden considerarse un caso especial de mallados irregulares más generales. En la práctica la visualización de volúmenes de datos no estructurados es aproximada generalmente por algoritmos de tetraedros proyectados [37].

Planteamiento del Problema

Diseñar e implementar un algoritmo para la visualización de volúmenes representado en tetraedros. Para volúmenes estructurados, las celdas serán subdivididas previamente en tetraedros para poder ser visualizados.

Propuesta de solución

Hace aprox. 20 años, la visualización de volúmenes directa de mallados tetraédricos no estructurados se aceleró de manera significativa por el algoritmo de Tetraedros proyectados (PT) de Shirley y Tushman [21]. De aquí pues, que se plantea presentar una

implementación del algoritmo PT propuesto por (Shirley y Tushman, 1990), utilizando las características programables de las tarjetas gráficas actuales, ya que este algoritmo opera con cualquier conjunto de datos tridimensionales que haya sido tetraedrizado.

Objetivo General

Presentar una implementación práctica del algoritmo de Tetraedros Proyectados para la visualización interactiva de datos volumétricos utilizando tarjetas gráficas programables.

Objetivos Específicos

1. Implementar el ordenamiento de primitivas tetraédricas de acuerdo a su visibilidad con respecto al punto de vista, típicamente de atrás hacia delante (*back-to-front)*, utilizando alguna característica de multihilo *(multithreading)* que se pueda implementar en C++ 11 en conjunción con una función *de ordenamiento de datos*.
2. Implementar la clasificación de cada tetraedro de acuerdo a su perfil de proyección para descomponerlos en triángulos, haciendo uso de la GPU y los recursos del hardware gráfico programable disponible.
3. Visualizar o desplegar los triángulos obtenidos utilizando los métodos que la librería OPENGL ofrezca, considerando la eficiencia en tiempo y espacio.
4. Realizar pruebas cuantitativas y cualitativas en la generación de las imágenes.

Alcance de este trabajo

- Debido a que la visualización está basado en el despliegue de primitivas, el tamaño del volumen y de los datos generados no deben sobrepasar las capacidades de memoria de la GPU.
- La Implementación se basará en sistemas con hardware gráfico Nvidia Geforce 9600 GT o superior con volúmenes de 8-bits por dato, y de a lo sumo 256^3 muestras.
- La plataforma de desarrollo y pruebas se basara en el sistema operativo Windows 7, utilizando el entorno de desarrollo (IDE) proporcionado por Microsoft, *Visual Studio 2013*, para la creación de la aplicación.
- Para el desarrollo se probarán distintas técnicas que exploten el hardware gráfico actual y la capacidad de procesamiento de CPU y GPU, específicamente en los problemas de *Ordenamiento de tetraedros y rendering.*

El trabajo es presentado en cinco capítulos. En el Capítulo I se estudian los conceptos básicos sobre la visualización de volúmenes. Seguidamente en el Capítulo II se explican los conceptos relacionados con el algoritmo de tetraedros proyectados (*PT*). En el Capítulo III se describe detalladamente la implementación del algoritmo de tetraedros proyectados (*PT*), donde se indican las estructuras de datos y el algoritmo general utilizado. En el Capítulo IV se realizan las pruebas pertinentes del sistema realizado, describiéndose los resultados obtenidos. Por último en el Capítulo V se presentan las conclusiones de la investigación y se proponen algunos trabajos futuros.

Capítulo I. Marco Teórico

En la medicina existen herramientas que permiten la adquisición de datos físicos del cuerpo para su diagnóstico. Estas herramientas (tomografía computarizada o CT, Resonancia magnética o MRI, entre otros) capturan, por medio de procesos físicos, las densidades de los materiales que componen el cuerpo.

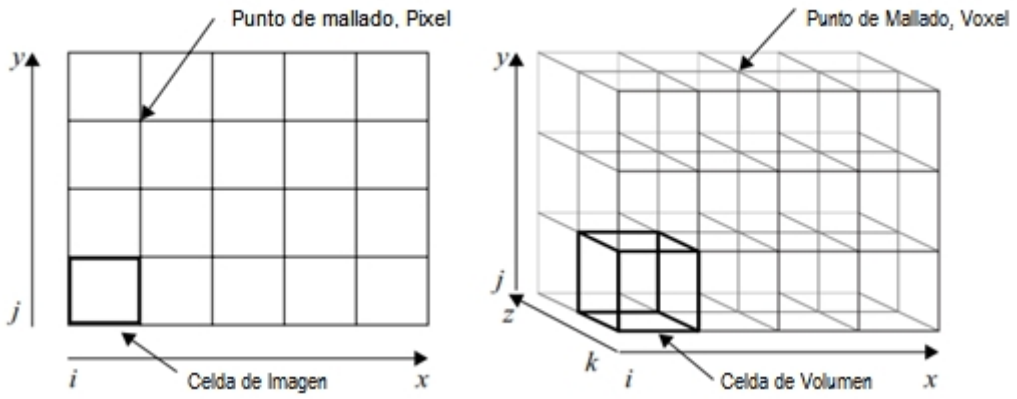

Figura 1: **A la izquierda**: Un mallado 2D, donde todos los píxeles de una imagen se arreglan en los puntos del mallado. **Derecha**: En los conjuntos de datos de un volumen, los vóxeles están dispuestos en una cuadrícula 3D.

Los datos médicos se suelen representar como una pila de imágenes individuales. Cada imagen representa un corte delgado de la parte del cuerpo escaneada y se compone de píxeles individuales (elementos de imagen). Estos píxeles están dispuestos en un mallado de dos dimensiones, donde la distancia entre dos píxeles es típicamente constante en cada dirección. Con esos datos, se forman un conjunto

de muestras, típicamente representados por una malla regular conocida como volumen [4]. Los datos volumétricos combinan imágenes individuales en una representación 3D sobre un mallado 3D (ver Figura 1).

La visualización de volumen se refiere a la generación de una representación visual de los datos volumétricos. La visualización de volumen tiene por objeto generar una representación visual completa del conjunto de datos, y por lo tanto de todas las imágenes al mismo tiempo. Un conjunto de datos 3D típico está constituido por un grupo de cortes o imágenes 2D adquiridas por un Tomógrafo, Resonador Magnético o escáner MicroCT. Por lo tanto, para poder generar una sola imagen, los vóxeles individuales del conjunto de datos deben ser seleccionados, ponderados, combinados y proyectados sobre el plano de imagen. Técnicas, como el despliegue directo de volúmenes, permiten observar el volumen en su totalidad, y eliminar únicamente los materiales no relevantes para su estudio.

Específicamente, el despliegue directo de volumen se refiere a las técnicas que producen una imagen proyectando directamente los datos del volumen sobre el plano imagen [1]. Estas técnicas requieren un modelo óptico para simular la interacción de los datos del volumen con la luz; esto es, cómo el volumen de datos genera, refleja, dispersa y ocluye la luz.

1. Vóxel y Volumen

El **vóxel** (del inglés *volumetric pixel*) es la unidad cúbica que compone un objeto tridimensional y es, por tanto, el equivalente del píxel en un

objeto 3D. Los datos volumétricos están compuestos por un gran número de vóxeles individuales. Por tanto el vóxel no es otra cosa que una unidad de volumen y es el equivalente del píxel en un objeto o imagen 2D. Al igual que los píxeles, los vóxeles no contienen su posición (x,y,z) en el espacio 3D, sino que esta se deduce por la posición del vóxel dentro del archivo de datos. Las imágenes con vóxeles se usan generalmente en el campo de la medicina y se obtienen, por ejemplo, de la Tomografía Axial Computarizada o de la Resonancia Magnética.

Las propiedades de los datos pueden estar representadas por un simple escalar, o por un conjunto de valores; en este último caso el valor de la muestra es multivaluada, incluyendo por ejemplo, color, intensidad, calor o presión. Incluso el valor puede ser un vector, representando, por ejemplo, la velocidad de cada muestra del volumen [5]. En consecuencia, un volumen es una colección de vóxeles generados mediante alguna medición o simulación.

La manera como está estructurado internamente el volumen es importante, pues de esto depende su acceso y almacenamiento. Un volumen es representado por una malla, que de acuerdo a la topología fundamental presentada por Jonathan Shewchuck[8], estas mallas pueden ser estructuradas y no estructuradas.

Las mallas estructuradas, se consideran todas las que se pueden representar como un arreglo tridimensional de escalares y cuya conectividad queda implícita. Las mallas no estructuradas son más generales; las mismas pueden tener concavidades y requieren conectividad explícita entre celdas [9][3][6].

Existe dos formas de interpretar un vóxel (ver Figura 2); una es considerarlo como un cubo en cuyo caso el valor de muestra corresponde al centro del mismo, o verlo como una celda en el cual el valor de muestra V corresponde a un vértice de la malla de la celda [7].

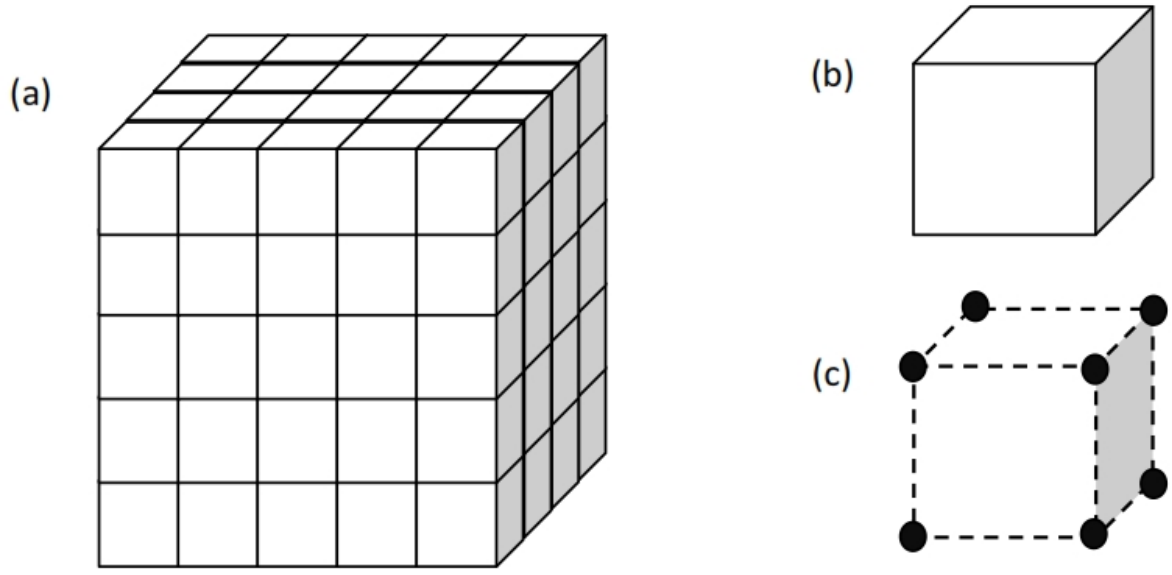

Figura 2: (a) una composición típica de volumen representado por un arreglo tri-dimensional, mientras que (b) y (c) son las dos interpretaciones de vóxel; en (b) la interpretación como cubo y en (c) su interpretación como celda.

Los volúmenes también pueden ser generados sintéticamente por medio de técnicas procedurales [7], que son usadas por los científicos y programadores para desplegar objetos tales como: fluidos o gases, fenómenos naturales (la niebla, nubes y fuego), visualización molecular, explosiones y cualquier otro efecto utilizado en los juegos 3D de computadora.

2. Despliegue Directo de Volúmenes

El Despliegue Directo de Volúmenes (Direct Volume Rendering) [1][7] se describe como una técnica que permite visualizar un volumen discreto mediante la proyección directa de sus muestras, y sin necesidad de reconstruir geometrías intermedias. Durante la proyección de las muestras se simula la interacción de la luz con un medio semi-transparente representado por el volumen.

El despliegue directo de volúmenes es usado en distintas áreas de las ciencias y en el entretenimiento, principalmente como herramienta de diagnóstico en la medicina; igualmente en simulaciones en las ciencias naturales o simplemente para lograr gran realismo en las escenas de los videojuegos [10]. En el proceso de despliegue, el volumen es un arreglo de datos escalares extraídos de un espacio tridimensional y la interacción luz-volumen produce varios fenómenos.

La interacción de la luz con los objetos y el medio ambiente, que en la práctica se descompone en la interacción de partículas y medio circundante, es por lo general lo que se conoce como Rendering; este se describe típicamente con absorción, emisión y la dispersión (scattering), y tiene su fundamento físico en *la teoría de transporte de la luz* [11].

El objetivo del despliegue directo de volúmenes no es simular todos estos fenómenos; por lo general, sólo se modela la propagación y el avance de la luz por el volumen. De esta manera, solo nos queda un modelo óptico en donde se toma en cuenta la emisión y absorción de la luz [11][12].

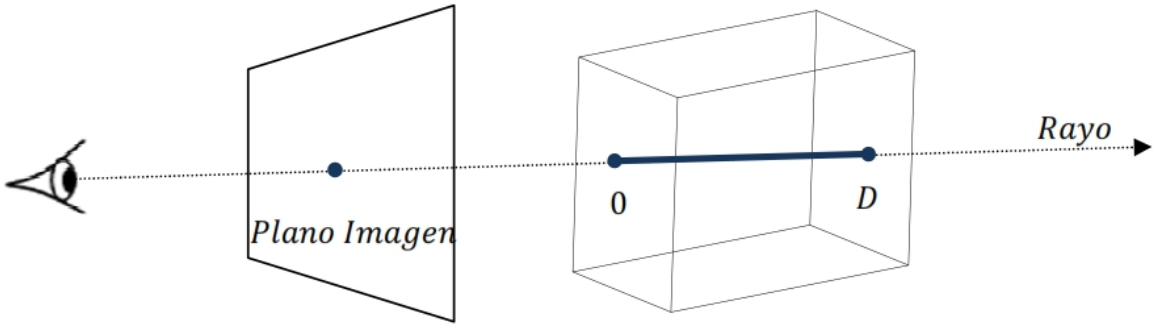

Figura 3: Para determinar el valor de un píxel en la imagen, se calcula la acumulación de color y opacidad a lo largo del rayo, desde que entra al volumen hasta que sale; barriendo λ desde 0 hasta D.

En el trabajo presentado por Carmona[13] se presenta detalladamente la derivación del modelo óptico Emisión-Absorción presentado por Sabella[14] y Williams et al.[15]. En este modelo el vector de visualización que atraviesa al volumen es parametrizado con λ en [0, D], y el color C resultante se obtiene mediante la siguiente ecuación:

$$C = \int_0^D c(\lambda)\tau(\lambda)e^{-\int_0^\lambda \tau(\lambda')d\lambda'}\, d\lambda, \qquad \text{ECUACIÓN 1}$$

en donde $C(\lambda)$, corresponde a la emisión, y $\tau(\lambda)$ a la absorción del volumen a una distancia λ (ver Figura 3). El factor $e^{-\int_0^\lambda \tau(\lambda')d\lambda'}$ correspondiente a la extinción de la luz, se puede interpretar como la transparencia $T(\lambda)$ del volumen a una profundidad o distancia λ. Basada en la transparencia, se puede calcular la

opacidad α acumulada en la travesía del rayo a cualquier distancia λ, como:

$$\alpha(\lambda) = 1 - T(\lambda)$$
$$\Rightarrow \alpha(\lambda) = 1 - e^{-\int_0^{\lambda} t(\lambda')d\lambda'}. \quad \text{ECUACIÓN 2}$$

3. Pipeline de Visualización de Volúmenes

Levoy [1] y Drebin et al.[16] en su modelo de visualización de volumen, proponen una serie de etapas (pipeline) que permiten el despliegue del volumen. Por ser los modelos más usados en la visualización gráfica y científica, resumiremos las etapas del pipeline presentado en Kniss [17] y Carmona[13]:

a. **Pre-procesamiento:** el volumen adquirido puede presentar ruido producto de las técnicas de muestreo empleadas por los sistemas de captura. Además, el volumen puede estar compuesto por una colección de imágenes independientes; por ejemplo, una colección de imágenes en formato Windows Bitmap (.bmp), o formato DICOM (Digital Imaging and Communications in Medicine). Es en esta etapa donde se realizan las conversiones de formato o se aplican filtros correctivos para reducir el ruido. Adicionalmente, se puede particionar el volumen, pre-calcular el vector gradiente por vóxel, etc., logrando entonces obtener una representación correcta del

volumen e información adicional, que luego se utiliza en las siguientes etapas del pipeline.

b. **Reconstrucción:** Básicamente se trata de la interpolación entre las muestras que ocurre cuando se desea estimar los valores desconocidos entre los vóxeles o muestras conocidas. Comúnmente el hardware está optimizado para estimar las muestras inexistentes con filtros lineales para texturas 1D, bi-lineales para texturas 2D y tri-lineales para texturas 3D [18].
c. **Clasificación:** Se refiere a la asignación de las propiedades ópticas a los vóxeles que componen el volumen. Es una de las etapas más importantes del pipeline de visualización de volúmenes, en donde se enfatiza las características de interés del volumen. Para valorar las características del volumen se acopla una función transferencia, y es por su manipulación donde se fijan las propiedades ópticas del volumen.
d. **Shading:** El shading consiste en modular el color y el brillo del vóxel al considerar su interacción con una luz externa [13]. El modelo Blinn-Phong es muy utilizado en la visualización de volúmenes. Este modelo es estrictamente local y tiene tres componentes, el ambiental que representa la luz proveniente en todas las direcciones, el difuso que representa la luz reflejada (scattered) isotrópicamente en todas las direcciones, y el especular que representa la luz reflejada en dirección al visor para simular superficies pulidas.
e. **Composición:** Es la etapa final del pipeline y consiste en mezclar por medio de los operadores under u over [1] [16], los

colores modulados producto de la etapa de clasificación y shading.

4. Discretización de la Ecuación de Visualización de Volúmenes

En la ecuación 1 no se tiene en cuenta el proceso de muestreo y clasificación del volumen. De acuerdo con Carmona [13], y centrando el análisis en la post-clasificación (ver Capítulo II), la integral de visualización de volumen se resume numéricamente a:

$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} e^{-h\tau(S_j)}\, c(S_i)\left(1 - e^{-h\tau(S_i)}\right). \qquad \text{ECUACIÓN 3}$$

En donde h es la longitud fija entre muestras y $\boldsymbol{n}$ la cantidad de muestras.

Sea el color $c_i = c(S_i)$, y la opacidad $\alpha_i = 1 - e^{-h\tau(S_i)}$ (Figura 4); la ecuación queda simplificada a:

$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} (1 - \alpha_j)\alpha_i c_i \qquad \text{ECUACIÓN 4}$$

$$\Rightarrow \quad C \approx \alpha_0 c_0 + (1 - \alpha_0)\alpha_1 c_1 + (1 - \alpha_0)(1 - \alpha_1)\alpha_2 c_2 + \cdots$$
$$+ (1 - \alpha_0) \cdots (1 - \alpha_{n-2})\alpha_{n-1} c_{n-1}.$$

La derivación de esta ecuación se detalla en [15].

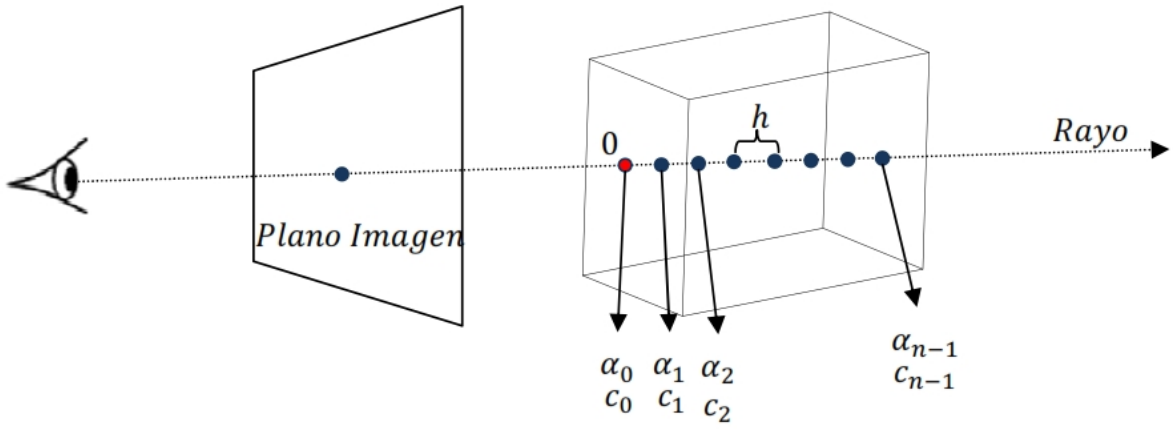

Figura 4: Para determinar el valor de un píxel, se componen la contribución de color y opacidad de las muestras equidistantes a lo largo de la travesía del rayo al pasar por el volumen. La distancia h generalmente es constante, y se asume constante también la emisión y absorción de cada segmento de longitud h.

Adicionalmente es necesario resaltar que la ecuación 4 es evaluada iterativamente a través de los operadores *over/under*. Estos operadores son utilizados para componer las muestras interpoladas, clasificadas y sombreadas [1][16]. Con el operador *under* se componen las muestras desde la más cercana a la más lejana (*front to back*) con respecto a la posición del ojo. Este operador se define como:

$$c \coloneqq (1-\alpha)\alpha_i c_i + c,$$
$$\alpha \coloneqq (1-\alpha)\alpha_i + \alpha, \qquad \text{ECUACIÓN 5}$$

En cambio el operador *over* evalúa las muestras desde la más lejana a la más cercana con respecto a la posición del ojo (*back to front*), definiéndose como:

$$c \coloneqq \alpha_i c_i + (1-\alpha_i)c,$$
$$\alpha \coloneqq \alpha_i + (1-\alpha_i)\alpha, \qquad \text{ECUACIÓN 6}$$

En los dos casos el C y α son inicializados en 0. Note que en el uso del operador *over*, el valor acumulado de α no es utilizado en la composición del color [15].

5. Algoritmos para el Despliegue de Volúmenes

Existen diversos algoritmos que permiten lograr la visualización de un volumen, por lo general estos algoritmos se caracterizan en dos grupos según la manera como utiliza el hardware gráfico. Los algoritmos existentes de Volumen Rendering directo son clasificados generalmente en dos categorías: *Orden de Imagen y Orden de Objetos*. Los algoritmos de *orden de imagen* llamados también *ray casting* emiten rayos a través de cada pixel de la imagen siendo muestreados y compuestos a lo largo del volumen y los que hacen uso de polígonos texturizados por lanzar cortes en vez de rayos, son principalmente acelerados por hardware (Textura 2D y 3D). Los demás algoritmos son técnicas basadas en software y generalmente en la actualidad implementadas en hardware programable, siendo estas el ray casting, splatting, y shear-warp.

El pipeline gráfico solo soporta el rendering de primitivas poligonales. No podemos usar directamente primitivas volumétricas, tales como un tetraedro sólido o un hexaedro. Por tanto, nos vemos obligados a descomponer nuestro objeto volumétrico en primitivas soportadas por el pipeline gráfico.

En el Volume Rendering basado en el Orden del Objeto (object-order), se hace uso del hecho de que un campo escalar 3D discreto puede ser representado como una pila de cortes 2D. Por lo tanto, podemos visualizar un conjunto de datos 3D mostrando un gran número de cortes 2D semitransparentes extraídos de este. Los polígonos que se corresponden con los cortes son las primitivas geométricas utilizadas para el rendering. Es importante notar que estas primitivas geométricas solo representan una *geometría proxy*. Ellos sólo describen la forma del dominio de datos, por lo general el área delimitadora (*bounding box*), y no la forma del objeto contenida en los datos. A continuación vamos a ver un enfoque basados en textura que muestra principalmente la forma en que se extraen estas imágenes de los cortes.

6. Volume Rendering Basado en Texturas 2D

La geometría proxy en este caso es una pila de cortes alineado a objetos, como se muestra en la Figura 5. En la literatura, los *cortes alineados a objetos* son a veces denominados como *cortes alineados a los ejes*. Preferimos el término objetos alineados, para hacer hincapié en el hecho de que los cortes se definen con respecto al sistema de coordenadas local del objeto.

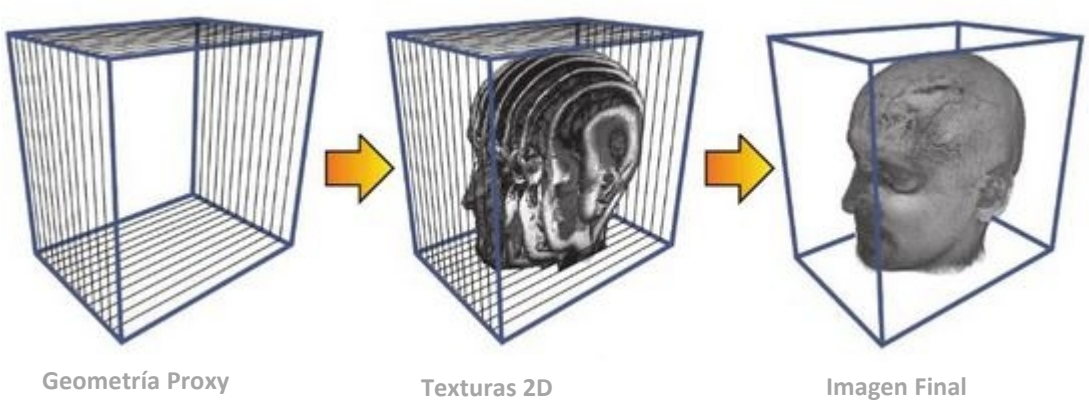

Figura 5: *cortes alineados a objetos* usados como geometría proxy en el mapeo de textura 2D.

Se requiere que todos los polígonos estén alineado con uno de los principales ejes en el espacio objeto (ya sea el ejes *x*, *y*, o *z*). La razón de este requisito es que las coordenadas de textura en 2D se utilizan para acceder a los datos de la textura. Por consiguiente, la tercera coordenada en el espacio debe ser constante. Para cada punto del espacio objeto 3D, una coordenada determina la imagen de textura a ser utilizada de la pila de cortes. Las dos componentes del vector restantes se convierten en las coordenadas reales de textura 2D. Los polígonos se mapean con la respectiva textura 2D, que, a su vez, se vuelve a muestrear por el filtro bilineal nativo en hardware.

6.1 Configuración de la Textura

Para permitir una rotación interactiva del data set, la dirección del corte debe ser elegido con respecto a la dirección de visión actual. El eje mayor debe ser seleccionado de una manera que minimiza el ángulo entre la normal del corte y la dirección de visión asumida. Esto elude efectivamente el problema de que los rayos de visualización pasen entre dos cortes sin intersectarlos. Como consecuencia, tres pilas de texturas son creadas, una pila corte para cada eje. Esto es

necesario para permitir el cambio entre diferentes pilas en tiempo de ejecución. La figura 6 ilustra esta idea en 2D, mostrando una incremental rotación de un volumen. Con un ángulo entre la dirección de visión y la normal del corte de 45° (en la figura 6 (d)), la dirección de corte se vuelve ambigua y puede ser elegido arbitrariamente. Con un ángulo mayor que 45°, las pilas deben ser conmutadas.

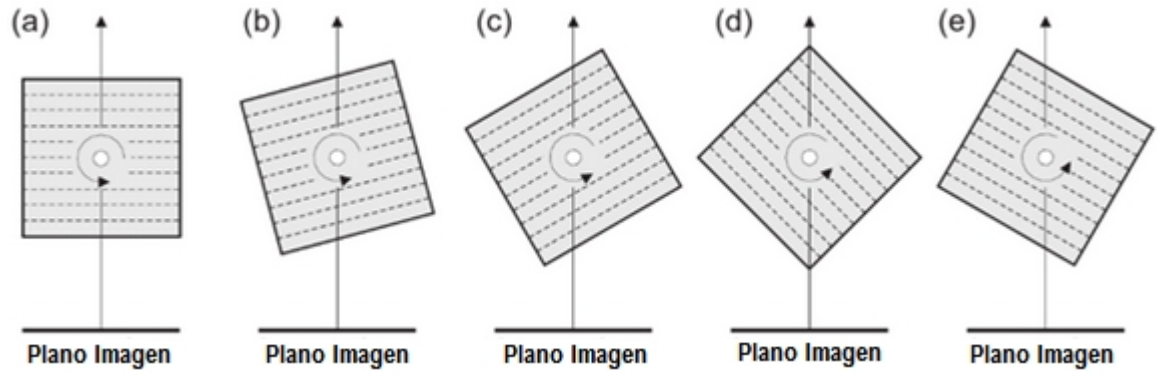

Figura 6: Conmutación de la pila de cortes de acuerdo con la dirección de visualización ilustrada en 2D. La pila de cortes utilizado para el rendering debe ser conmutados entre los frames (c) y (e) con el fin de minimizar el ángulo entre la normal del corte y la dirección de visualización. En el frame (d) la dirección de corte es ambigua, ya que ambas alternativas resultan en el mismo ángulo.

Durante la configuración de las texturas, debemos preparar las texturas para las tres pilas de cortes y cargarlo en la memoria gráfica local. Durante el rendering, en la configuración de la geometría se debe tener cuidado de que las texturas correctas se mapeen al polígono correspondiente.

6.2 Composición

De acuerdo con el modelo físico descrito en la *ecuación 3* la ecuación de transferencia de radiación puede ser resuelta de forma iterativa por discretización a lo largo del rayo de visualización. Como

se describió anteriormente, el formato interno para nuestras texturas 2D es RGBA, que significa que cada texel alberga cuatro valores fijos, un valor para las componentes rojo (R), verde (G), y color azul (B), respectivamente, más uno para el valor de opacidad (A). Para cada voxel, el valor de color (RGB) es el término fuente c_i de la ecuación 4.

El valor de opacidad A es la transparencia α_i de la Ecuación 4. Usando esta configuración, la radiación I resultado de una integración a lo largo de un rayo de visión, se puede aproximar por el uso de alpha blending.

La ecuación de *blending* especifica una combinación lineal de componente a componente del cuarteto RGBA del fragmento de entrada (source) con los valores que ya contiene el frame buffer (destination). Si el blending está desactivado, el valor de destino se sustituye por el valor fuente. Cuando el blending está activado, la fuente y la tupla RGBA destino se combinan mediante una suma ponderada, formando un nuevo valor de destino. Con el fin de calcular la solución iterativa de acuerdo con la ecuación 4, la opacidad α_i almacenados en la componente A del mapa de textura debe ser utilizado como blending factor. Para implementar el esquema de composición back-to-front un componente de color C ε {R, G, B} es calculado por la ecuación de blending como sigue:

$$C'_{\mathrm{dest}} = C_{\mathrm{src}} + C_{\mathrm{dest}} (1 - A_{\mathrm{src}}) . \qquad \text{ECUACIÓN 7}$$

Este esquema del blending corresponde a la configuración alpha-blending de OpenGL:

```
glEnable(GL_BLEND);
glAlphaFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

Nota: Se usa GL_ONE y 1-SRC_ALPHA cuando el color y la opacidad ya vienen premultiplicados en la textura es decir cuando el RGB es en realidad A*RGB.

7. Cell Projection

Es una técnica para la visualización directa de volúmenes de mallados no rectilíneos. Esta clase de algoritmos se basa en métodos que proyectan primitivas del espacio-objeto, por ejemplo, voxels, a la pantalla en un específico orden de profundidad. Es decir una celda en un volumen se proyecta sobre la pantalla, y su contribución a los píxeles bajo su punto de proyección es calculada, y se compone con las aportaciones de las primitivas previamente proyectadas. Los algoritmos de proyección de celdas necesitan obtener el correcto orden de visibilidad de las celdas para generar el correcto resultado de composición. Aquí, el orden la visibilidad de las celdas en sí no es trivial y puede consumir mucho tiempo.

A continuación, describiremos brevemente un método relacionado que proyecta celdas de volumen a un plano de imagen.

Shirley y Tushman [1990], introducen cell projection, que proyecta celdas enteras de volumen a la pantalla. Dado que las celdas de volumen tienen típicamente una proyección de pantalla simple (huella), esta proyección puede ser representada por pocos polígonos. Cada polígono entonces representa la contribución de color y opacidad de la

parte respectiva de la celda proyectada y es rasterizada por el hardware gráfico del ordenador en el frame buffer. Shirley y Tuchman proporcionan una solución para varios tipos de mallados al dividir las celdas en tetraedros. Estos tetraedros son entonces transformados en espacio-ojo, y la huella de su proyección es *teselada* o dividida en un máximo de cuatro triángulos.

El hardware gráficos normalmente rasteriza triángulos usando el *Gouraud shading*, en el que tanto el color y la opacidad se calculan en los vértices y se interpolan en el polígono. Esta interpolación espacio-imagen no suele reflejar las propiedades físicas de los datos del volumen, pero es sólo una simple aproximación. Si bien esto es apenas perceptible si las celdas de volumen, y por lo tanto los triángulos, son muy pequeños, rápidamente se hace evidente con elevados niveles de zoom, donde las celdas individuales son proyectadas a las áreas de pantalla más grande. En consecuencia, *Wilhelms y Van Geldern* [19] muestra métodos que mejoran la composición y la interpolación, los cuales están limitados a mallados cartesianos (rectilíneos) [7].

Capítulo II. Algoritmo Tetraedro Proyectado (PT)

El algoritmo PT visualiza una función escalar f(x, y, z) definida en una región del espacio tridimensional haciendo rendering de polígonos tales como triángulos, parcialmente transparentes, que puede procesar rápidamente el hardware gráficos especializado. El algoritmo Tetraedros proyectado opera con cualquier conjunto de datos tridimensionales que ha sido tetraedrizado, es decir la analogía que encontramos cuando datos en el plano son divididos en triángulos.

El algoritmo Tetraedros proyectados (PT) por Shirley y Tuchman [21] fue desarrollado en 1990 y se convirtió en la base de varias trabajos desde entonces. El algoritmo PT consiste en proyectar los tetraedros al plano de imagen y hacer la composición según el orden de visibilidad. La primera consecuencia es que los datos 3D deben ser divididos en tetraedros. En el caso de volúmenes regulares, esto se puede lograr mediante la subdivisión de cada celda hexaédrica en 5 tetraedros [21].

Una celda hexaédrica se refiere a la región rectilínea delimitada por ocho muestras de datos vecinas, las esquinas de la celda. Las celdas son modeladas conteniendo un material semi-transparente que ocluye emite luz en cantidades que dependen de los valores de datos escalares de las esquinas de las celdas y su asignación a la intensidad (rojo, verde, y azul) además de sus valores de opacidad utilizando funciones de transferencia.

El algoritmo de PT se puede resumir como sigue:

1. Descomponer el volumen en celdas tetraédricas. Los valores escalares se definen en cada vértice del mallado. Dentro de cada celda tetraédrica, f(x, y, z) se supone que es una combinación lineal de los valores de los vértices. (Figura 7)
2. Ordenar las celdas según su visibilidad.
3. Clasificar cada tetraedro de acuerdo con su proyectado contorno y descomponerlo en pequeños triángulos.
4. Encontrar los valores de color y opacidad en los vértices de los triángulos utilizando la integración a lo largo de los rayos de visualización.
5. Desplegar o hacer Render de los triángulos.

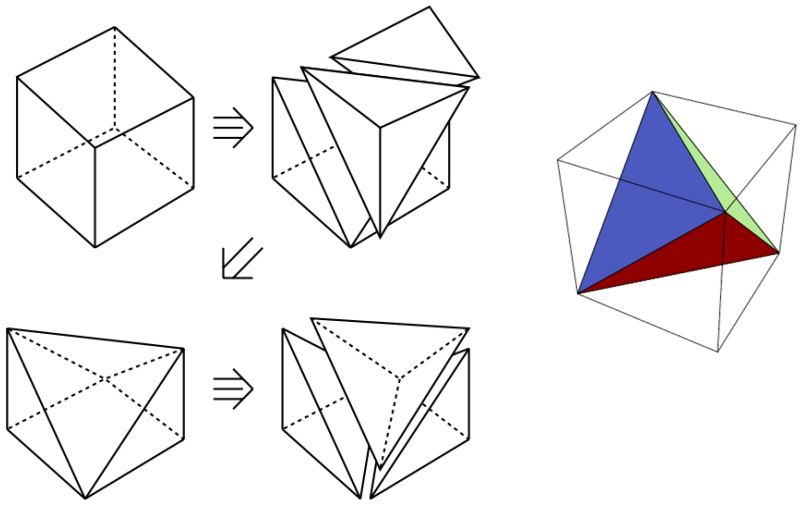

Figura 7: Descomposición de un cubo en 5 tetraedros.

La idea de tetraedros proyectado es que la imagen compuesta de los triángulos dibujados en el último paso sea de similar en apariencia a una completa visualización de volumen directa de los tetraedros de entrada. Debido a que los triángulos son semitransparentes, ellos deben ser renderizados en un orden de profundidad.

Los tetraedros proyectados se descomponen en 1 hasta 4 triángulos de acuerdo con un esquema de clasificación. Esta clasificación se basa en una comparación de las normales de las caras del tetraedro con el vector de visualización. Tres resultados diferentes se pueden esperar de este producto {+, -, 0} que indica si una cara esta de frente, de espaldas, u ortogonal al vector de visualización, respectivamente. Para cada una de las cuatro clases diferentes, un tetraedro se descompone en un número específico de triángulos (ver Figura 8).

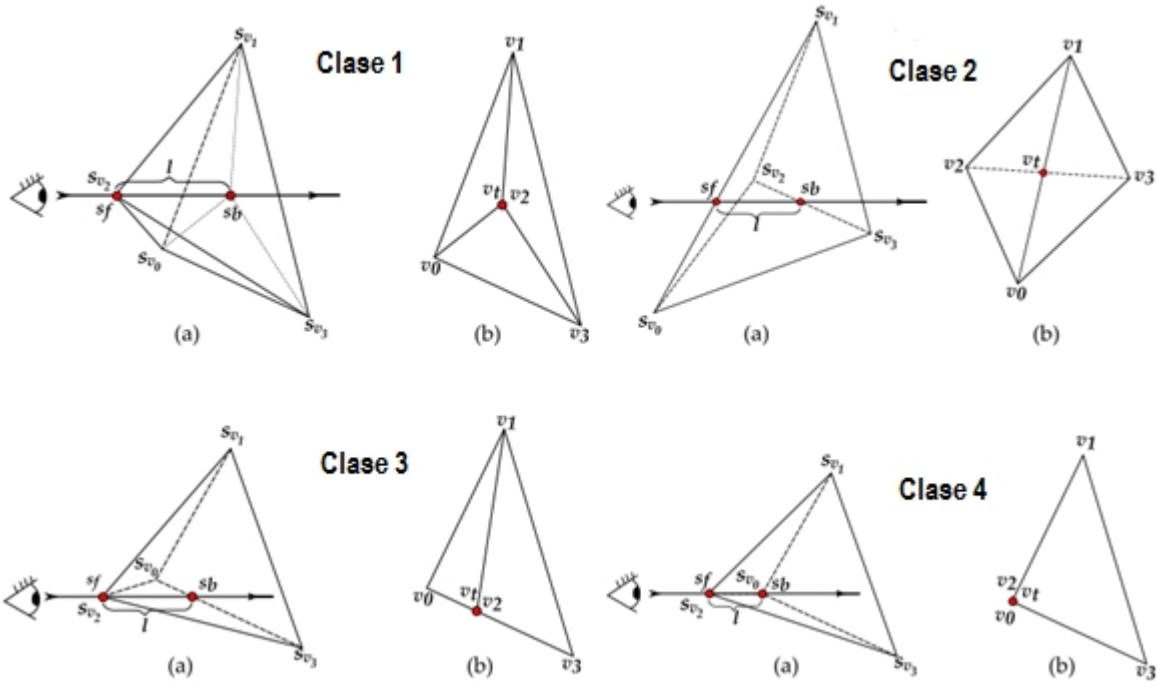

Figura 8: Un ejemplo de cada clase de proyección [20]. El dibujo ilustra el tetraedro y el vector de visualización (a) y el tetraedro proyectado (b).

Por ejemplo, las proyecciones Clase 1 se descomponen en tres triángulos, pero la Clase 2 se descompone en cuatro.

El vértice ***thick*** se define como el punto del segmento de rayo que atraviesa la distancia máxima a través del tetraedro. De forma análoga, los otros vértices se llaman vértices ***thin***, porque no hay una

distancia que ellos cubran. Para las proyecciones clase 2, el vértice *thick* se calcula a partir de la intersección entre la parte frontal y el posterior de los bordes proyectados, por eso se llama *vértice de intersección*, mientras que para las otras clases es uno de los vértices proyectados. Los vértices *thin* tienen los mismos valores s_f y s_b, mientras que para los vértices *thick* estos valores tienen que ser interpolados a partir de los de los vértices *thin*.

La Figura 8 muestra un ejemplo para cada clase de proyección. En el primer caso (clase 1), $s_f = s_{v2}$ y s_b se calcula mediante una interpolación trilineal de s_{v0}, s_{v1} y s_{v3}. En el segundo caso (clase 2), el *thick* vértice v_t, es la intersección de los dos segmentos interiores, y los valores escalares s_f y s_b se calculan por interpolación en los segmentos de v_0v_1 y v_2v_3, respectivamente. El tercer caso (clase 3) tiene $s_f = s_{v2}$. En el cuarto caso (de la clase 4) s_f y s_b son iguales a s_{v2} y s_{v0}, respectivamente.

Integración y Rendering

Antes de dibujar los triángulos proyectados, los valores de color se deben asignar a cada vértice. Cada celda se supone que consiste de un material semi-transparente que emite su propia luz, transmite algo de luz que viene de atrás, y ocluye un poco de luz, tanto desde atrás y desde el interior de la celda. La luminosidad y la oclusión se representan como la intensidad y la opacidad.

Para crear la imagen, los valores escalares a partir de los datos originales se convierten a valores de intensidad y opacidad utilizando funciones de transferencia en donde están almacenados las tablas de

color rojo, verde, azul, y opacidad. Una función de transferencia es utilizada para asignar los valores escalares de cromaticidad (RGB) y valores de opacidad (A). Los valores estimados de los datos para la intersección de bordes se encuentran por interpolación lineal entre los vértices de las esquinas adyacentes, y para los puntos de intersección en la cara por interpolación bilineal de los vértices de la cara; estos valores se utilizan luego para el rendering.

La contribución acumulada de intensidad y opacidad del medio entre la cara delantera y trasera de las celdas debe ser determinada. Este proceso se conoce como *Ray Integration (Integración de Rayo)*, e implica la solución de (aproximadamente) una ecuación diferencial lineal (28). Para un valor escalar s, una tabla (función de transferencia) se consulta con el fin de obtener los valores de cromaticidad ($C\ (s)$) y el coeficiente de extinción ($\tau_{(s)}$), que miden la cantidad de luz absorbida por la celda y se asocia directamente con el valor de opacidad. Los *Thin* vértices se dibujan con cero opacidad y con la cromaticidad original a partir de la función de transferencia. Por otro lado, el *Thick* vértice se hace con el color promedio entre los valores escalares frontal y posterior, y la opacidad se calcula por $\alpha = 1 - e^{-\tau_a\ l}$, donde τ_a es el promedio de $\tau_{(s\ )}$ y $\tau_{(s\ )}$.

Para finalizar, los triángulos visualizados son rasterizados en fragmentos mediante la interpolación de los valores de color y opacidad de los vértices *Thick* y *Thin*. Los fragmentos se componen en *back-to front* orden, y, para cada nuevo color añadido al *frame buffer*, el nuevo color del píxel se calcula como $C_{i+1} = \alpha_i Ci + (1- \alpha_i)C_{i-1}$

donde C_{i-1} es el previo color almacenado en el *frame buffer*, y Ci y α_i son los valores interpolados de color y opacidad. El nuevo valor de opacidad α_i se calcula mediante la acumulación de α_{i-1} con α_{i+1}. Un ejemplo de la composición de color se muestra en la Figura 9.

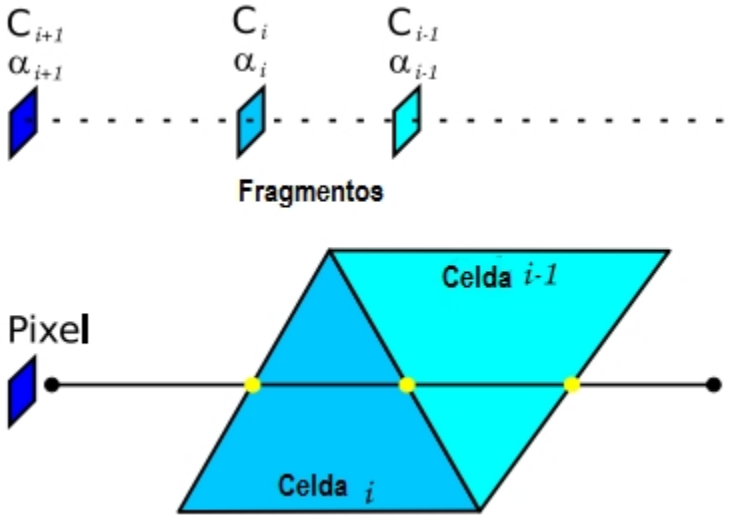

Figura 9: Fragmento composición para calcular el color final del pixel añadiendo la contribución de cada celda en orden back-to-front.

En este capítulo pudimos ver la teoría de un algoritmo que pertenece a la técnica de proyección de celdas conocido como algoritmo de tetraedros proyectados como método de visualización de un volumen, a continuación describiremos el proceso de implementación utilizado en este trabajo.

Capítulo III. Diseño e Implementación

La implementación se basa principalmente en el trabajo presentado por (Shirley and Tuchman, 1990) (presentado en el capítulo anterior) y en el de (Marroquim et al., 2006) que presentaremos en este capítulo. Se describen los requerimientos del sistema que se tomaron en cuenta en la etapa de Implementación, además, se describen los detalles de dicha implementación.

1.1 Detalles de Diseño

La etapa de diseño consistió de manera general, en modelar una aplicación que visualizara datos volumétricos utilizando el algoritmo Tetraedros proyectados (PT). Una vez procesada la visualización de los datos volumétricos, la aplicación debe ser capaz de permitir al usuario la interacción con la visualización. La interacción será una interacción básica, propia en aplicaciones de este tipo, las cuales consisten en la trasformación de la vista y la manipulación de los valores de opacidad y color de los valores de intensidad existente en los datos. Para esta última el usuario tendrá a su disposición proporcionada por la aplicación una interfaz con una función de transferencia, la cual tendrá la oportunidad de manipular.

La función de transferencia es unidimensional, ya que en ella el único valor característico es la opacidad representado por el eje vertical, el cual refleja la transparencia de cada valor de intensidad encontrado en el eje horizontal. A esta función también se agrega un color a cada

valor de intensidad el cual puede ser cambiado por el usuario. Esta función de transferencia muestra además de manera gráfica la distribución de frecuencia de los valores de intensidad encontrados en los datos cargados para visualizar. Esta gráfica es lo que se conoce como histograma de valores, y representa una fuente de información importante ya que muestra las densidades de intensidad presentes en los datos visualizados.

A continuación se describe las interacciones generales del usuario con los componentes del programa (ver figura 10). La descripción detallada de los elementos del programa se expone a continuación, en donde se indica las funcionalidades ofrecidas por cada uno.

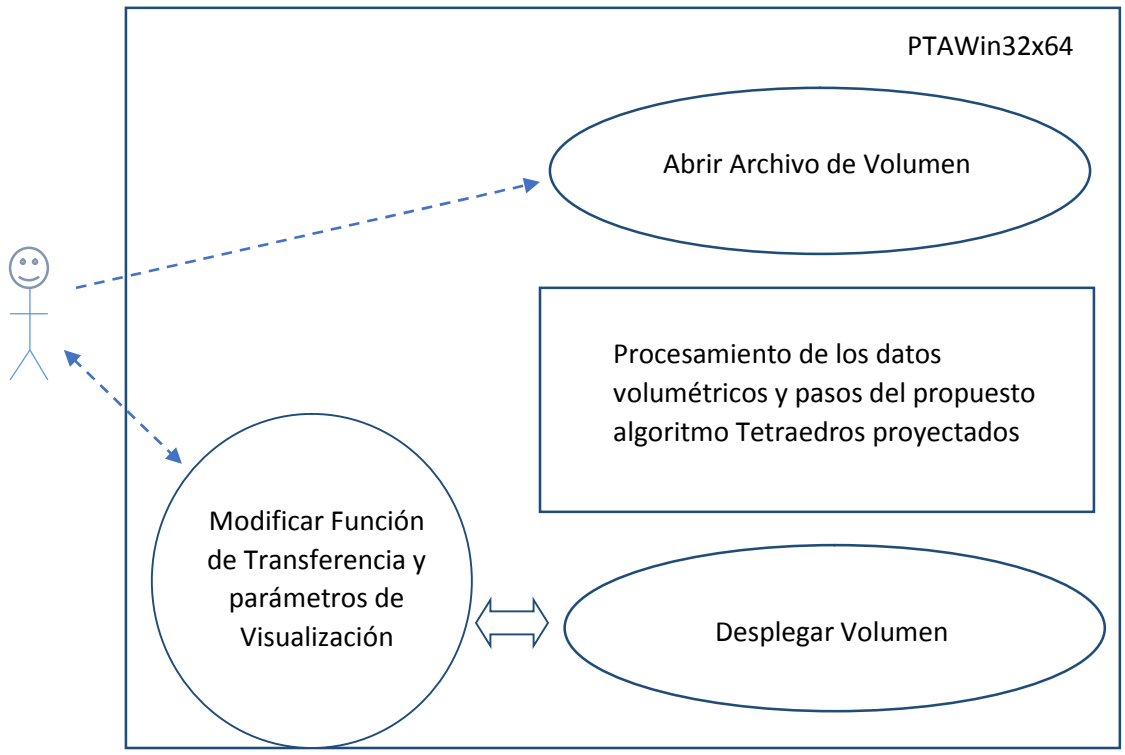

Figura 10: Representación general del sistema o programa que implementa el Algoritmo PT, así como las interacciones del usuario con el mismo

1.1.1 Abrir Archivo de Volumen

Parte del programa PTAWin32x64 (figura 10) que permite al usuario abrir los archivos compatibles necesarios para el uso del sistema. El usuario al ejecutar el sistema o programa, para poder utilizar las funcionalidades del mismo debe cargar el archivo que corresponde a un volumen. Esto se hace disponible para el usuario a través de una interfaz creada por el programa conocida en inglés como "*Common File Dialog*" [39], la cual se genera cuando el usuario interacciona con el programa. El diálogo generado se presenta en la figura 11.

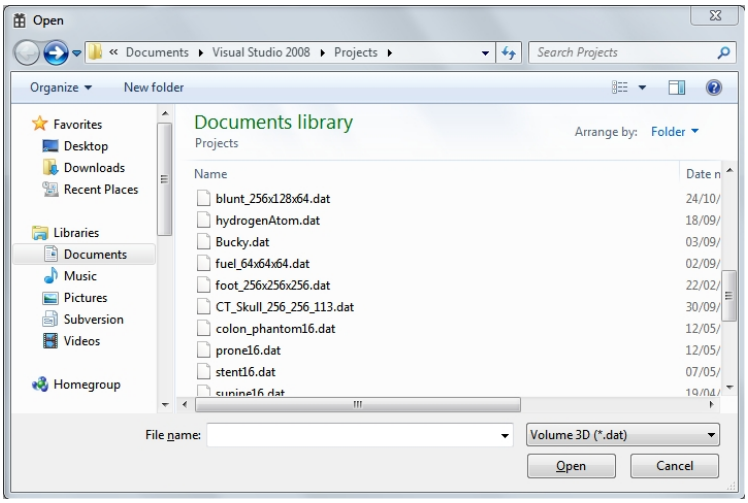

Figura 11: Cuadro de dialogo generado por el programa, el cual es común en SO Windows.

En esta figura podemos observar el tipo de archivos aceptados por el programa. Este tipo de archivo corresponde a un archivo con extensión "*.dat* " [38], el cual está definido como aparece en la figura 12 (conocido también como formato QVis). Este archivo es un archivo de texto en el cual, los parámetros desde *ObjectFileName* hasta *GridType* deben ser especificados en todos los archivos como aparece

en la figura por compatibilidad con el programa; es decir el archivo de volumen debe tener un byte por unidad de información, como especifica el parámetro *Format*, el resto es información para la implementación como lo indica el formato de volumen [38].

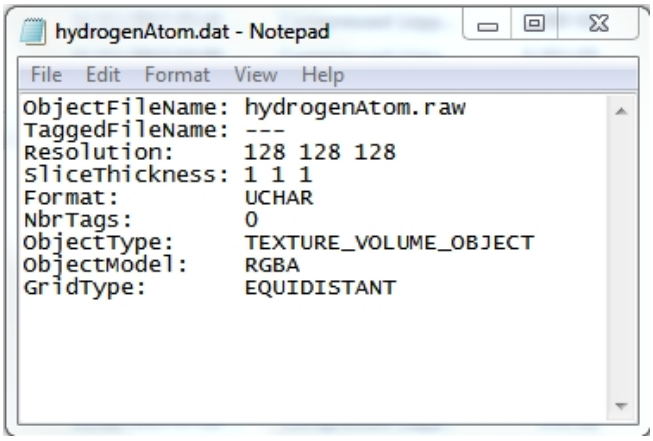

Figura 12: Archivo de texto que presenta el formato del volumen acepta el programa.

El parámetro *ObjectFileName* especifica el nombre del archivo binario que contiene la data volumétrica con los voxel en bruto (*raw*) que utiliza el programa para su funcionamiento. El parámetro *Resolution* especifica las dimensiones del volumen, es decir, largo, alto y ancho; es decir la resolución del conjunto de datos está dada por el número de voxels en las direcciones ***x-***, ***y-*** y ***z-***. Este parámetro nos da el tamaño en disco en bytes del archivo de volumen si multiplicamos las tres dimensiones, porque *Format* es de 1 byte. El parámetro *SliceThickness* nos especifica las dimensiones del *voxel volumétrico* (generalmente en milimetros), el *voxel volumétrico* se explicó en el capítulo anterior.

Entonces cada archivo de volumen está compuesto de un archivo de información del volumen en formato de texto (.dat) y un archivo con extensión ".raw", que es un archivo binario con la data volumétrica. Al

usuario se le solicitará a través de la interfaz de la figura 11 que seleccione un archivo existente; una vez seleccionado, el programa valida el archivo. De tratarse de un archivo válido se procede a la carga del archivo en el sistema, de lo contrario, se notifica al usuario que el archivo no es válido o no existe (según corresponda).

1.1.2 Procesamiento de los datos (Algoritmo Tetraedros Proyectados (PT))

El algoritmo se ejecuta en la CPU y en la GPU, aunque en su mayoría en la GPU y se divide en tres partes principales. El primer paso consiste en dividir la data volumétrica en tetraedros, en otras palabras dividir el volumen en un *mallado tetraédrico.* Este paso se ejecuta de la siguiente manera; debido a que tenemos a disposición volúmenes regulares o estructurados, esto lo logramos mediante la subdivisión de cada *celda hexaédrica* en 5 tetraedros.

Una celda hexaédrica del volumen la formamos con ocho datos, cuatro del corte *front,* y cuatro sobre el corte *back,* correspondientes en posición sobre cada corte. Un volumen de dimensiones L x A x G tiene (L-1) x (A-1) x (G-1) hexaedros y cada hexaedro está dividido en 5 tetraedros. Esta cuenta nos da el número total de tetraedros del mallado en que hemos dividido el volumen.

Todo este proceso es transparente porque los tetraedros resultantes de la división del volumen en esta primera etapa no son renderizados (ver figura 13), ya que lo único que le importa al programa es su información geométrica, la cual almacena en dos *buffer,* uno para la

información geométrica de los vértices de los tetraedros y en el otro los tetraedros en si los cuales están identificados por los índices de sus vértices. Por tanto la información del *buffer* de vértices contiene la información de una posición geométrica que hemos elaborado para cada dato volumétrico, por lo que podemos decir que el número de vértices corresponde a las dimensiones del volumen en términos de datos, es decir *Voxeles* o conjunto de *Voxeles*. El *buffer* de tetraedros tiene en cada unidad de información por así decirlo, los índices de los 4 vértices que forman el tetraedro, estos índices corresponden a la posición lógica de los vértices dentro del *buffer* de vértices.

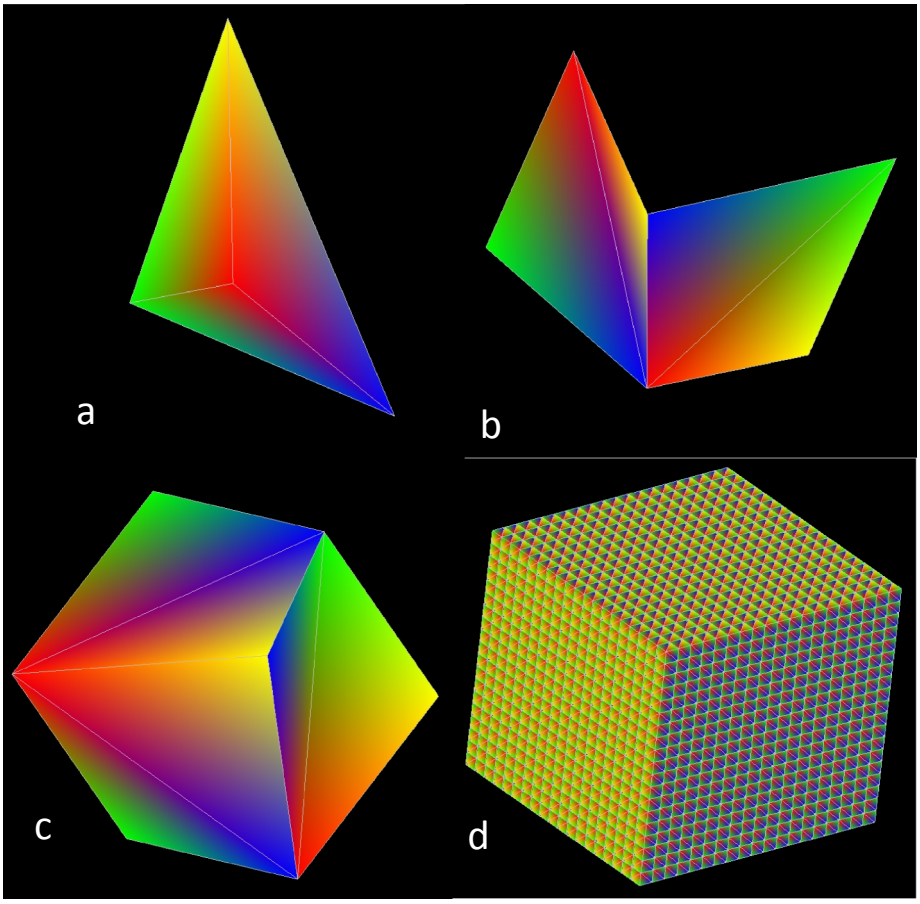

Figura 13: Parte del proceso de División de un *hexunit* en 5 tetraedros en *a*, *b* y *c*. En *d* nos muestra el mallado tetraédrico completo de un volumen.

El segundo paso del algoritmo PT implementado por el programa es clasificar los tetraedros generados en el paso anterior, de acuerdo a un esquema de clases. Un tetraedro puede tener cualquiera de cuatro siluetas dependiendo del punto de vista y la orientación del tetraedro. Dado que el objetivo es aproximar este elemento de volumen con triángulos, primero clasificamos el tetraedro basado en su forma proyectada. El proyectado tetraedro es entonces descompuesto en 1 a 4 triángulos de acuerdo con un esquema de clasificación como lo muestra la figura 14.

La figura 14 enumera las cuatro posibles proyecciones, que surgen de seis casos posibles. Esta clasificación se basa en una comparación de las normales de las caras del tetraedro con el vector de visualización. Tres resultados diferentes se pueden esperar de este producto {+, -, 0} que indica si una cara es visible, no es visible, u ortogonal al vector de visualización, respectivamente como se explicó en el capítulo anterior.

Para la clasificación de los tetraedros, el programa debe agrupar cinco tetraedros de una unidad *Hexaédrica* como la explicada anteriormente. Llamamos a este conjunto de tetraedros *hexaedro base* o *hexunit*, que es una abreviación que vamos a utilizar para denominar este conjunto de cinco tetraedros. De esta manera una observación clave es, que al hacer una *proyección paralela* (al vector de visualización) u ortográfica del volumen, todos los *hexunits* en que está compuesto el volumen se proyectan a la pantalla exactamente de la misma manera. Por lo tanto, para evitar cómputos redundantes, los parámetros de proyección se computan una sola vez para un *hexaedro base* que seleccionamos del volumen.

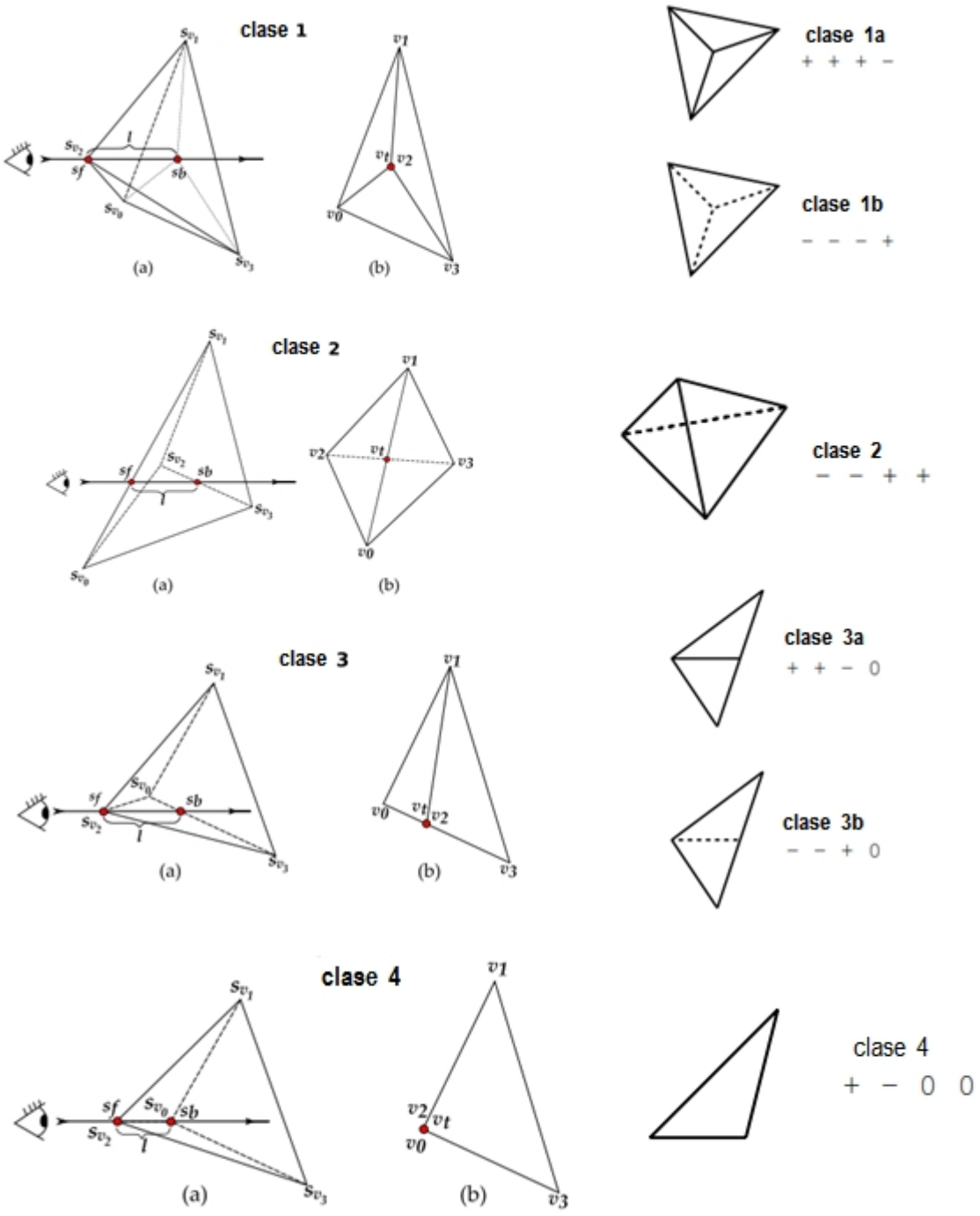

Figura 14: Un ejemplo de cada clase de proyección [20], [21]. El dibujo ilustra el tetraedro y el vector de visualización (a) y el tetraedro proyectado (b). Al lado los posibles casos para cada clase según la comparación de las normales de las caras del tetraedro con el vector de visualización.

Cada *tetraedro base* (tetraedros del *hexaedro base*) es proyectado de coordenadas objeto a coordenadas de pantalla (*Normalized Device Coordinates*); utilizando las respectivas matrices, y sus proyecciones se clasifican en la clase correspondiente por medio de cuatro test o pruebas utilizadas por (Marroquim et al., 2006). Cada prueba es una evaluación de un producto vectorial computada con los vértices proyectados v_0, v_1, v_2 y v_3, como se muestran en la Figura 15. El resultado de los test es utilizado para indexar una tabla que determina la clase de la proyección.

$$
\begin{aligned}
vec1 &= v_1 - v_0 \\
vec2 &= v_2 - v_0 \\
vec3 &= v_3 - v_0 \\
vec4 &= v_1 - v_2 \\
vec5 &= v_1 - v_3
\end{aligned}
$$

$$
\begin{aligned}
test1 &= sign((vec1 \times vec2).z) + 1 \\
test2 &= sign((vec1 \times vec3).z) + 1 \\
test3 &= sign((vec2 \times vec3).z) + 1 \\
test4 &= sign((vec4 \times vec5).z) + 1
\end{aligned}
$$

Figura 15: Pruebas realizadas en la clasificación de la proyección. La función *sign* devuelve -1, 0, 1 dependiendo si el argumento es menor que, igual o mayor que cero respectivamente.

Esta tabla es denominada *tabla ternaria* debido a que esta tabla ordena la permutación de los resultados de los cuatro test anteriores, y es ternaria debido a los posibles resultado de cada test {0, 1, 2}. Esta situación produce una tabla con 81 entradas, las cuales surgen de la permutación de los vértices para cada clase de proyección. Entonces esta tabla sirve para identificar la clase de proyección de los vértices, porque dependiendo del valor de la suma de los cuatro test se determina la clase de proyección del tetraedro, así como también las posibles permutaciones que pueden tener los vértices para cada clase.

Estas permutaciones de los vértices están relacionadas con los cálculos de la intersección del vector de visualización con los tetraedros, la cual está determinada por clase de proyección como se puede ver en la figura 14. El cálculo de las intersecciones se basa en las clase 1 y 2, ya que la 3 y la 4 se consideran degeneraciones de las dos primeras clases. Las ecuaciones utilizadas para los cálculos de las intersecciones se describen en algoritmos y ecuaciones de intersección, para la clase 2 es descrito en el libro *Graphics Gems III* (pg. 199-202) [40], en el cual el algoritmo determina si dos segmentos de línea en el espacio 2D intersectan o no y determina su punto de intersección, y para la clase 1 la ecuación de intercesión es descrita y explicada por los autores originales del Algoritmo de Tetraedros Proyectados (PT), Shirley y Tuchman [21], la cual de manera resumida calcula el punto de intersección de un segmento de recta con un plano.

Al diseñar la tabla se pudo observar que para cada permutación existe un resultado único relacionado con la posición de los vértices en el buffer de tetraedros del mallado, el cual puede llegar a ser de hasta seis vértices. Puede llegar a ser seis vértices porque para presentar en pantalla por ejemplo para la clase 2, la división del tetraedro en 4 triángulos, la manera más eficiente de realizar esto es utilizar la función optimizada de OpenGL *glMultiDrawElements* con el parámetro GL_TRIANGLE_FAN, los detalles del proceso no van a ser explicados pero de manera general lo que se hace es dibujar los triángulos en la forma de un abanico, y a para realizar esto en este caso se necesitan 6 vértices para dibujar los 4 triángulos. Por tanto el hecho de que

exista para cada entrada de la tabla anterior un resultado único en términos del orden de vértices a dibujar, se tiene entonces para el proceso de dividir cada tetraedro en triángulos, otras 81 entradas las cuales servirán para indexar el buffer de tetraedros del mallado en el momento de almacenar en un orden estipulado por estas entradas, los vértices de los tetraedros en memoria de video para ser utilizados por la función *glMultiDrawElements* para realizar el respectivo proceso de rendering. El proceso anterior expuesto se puede resumir en la figura 16 donde se muestra las características de las tablas utilizadas para la clasificación de los tetraedros.

```
const GLuint order_table[81][4] =
{ // (v1, v2, v3, v4) # Ternary - # Decimal
     { 0, 3, 2, 1 } // 0 0 0 0 - 0
    ,{ 0, 3, 2, 1 } // 0 0 0 1 - 1
    ,{ 0, 3, 2, 1 } // 0 0 0 2 - 2
    ,{ 1, 0, 3, 2 } // 0 0 1 0 - 3
    ,{ 2, 3, 1, 0 } // 0 0 1 1 - 4
          .              .      .
          .              .      .
          .              .      .
    ,{ 0, 1, 2, 3 } // 2 2 2 2 - 80
};
```

```
//--------- TRIANGLE FAN TABLA --------------
const GLint triangle_fan_order_table[81][6] =
{ // (v1, v2, v3, v4) # Ternary - # Decimal
 { -1,  0,  3,  2,  1,  0 } // 0 0 0 0 - 0
,{  2,  1,  0,  3,  9,  9 } // 0 0 0 1 - 1
,{  2,  0,  3,  1,  0,  9 } // 0 0 0 2 - 2
,{  3,  2,  1,  0,  9,  9 } // 0 0 1 0 - 3
,{  2,  1,  0,  9,  9,  9 } // 0 0 1 1 - 4
             .                    .      .
             .                    .      .
             .                    .      .
,{ -1,  0,  1,  2,  3,  0 } // 2 2 2 2 - 80
};
```

Figura 16: Algunas entradas de la tabla ternaria usada para la clasificación de la proyección de los tetraedros. En verde se puede apreciar las permutaciones de valores tomados por los test así como el índice que le corresponde en la tabla. Entre llaves a la izquierda el orden en que los vértices de los tetraedros deben ser almacenados para realizar los cálculos, a la derecha como deben ser dibujados los triángulos del tetraedro proyectado, los -1 indican clase 2, vértice de intersección que debe ser calculado.

Una vez clasificados los *tetraedros base* se procede a realizar la clasificación del *resto de los tetraedros del volumen*. La clasificación de los 5 *tetraedros base* producen valores de proyección los cuales

son guardados para clasificar el resto de los tetraedros del volumen, como son la *clase de proyección*, *coordenadas de los vértices proyectados (thick vertex)*, *parámetros de intersección* para el computo de los valores escalares front y back del *thick vertex* y por último el orden de rendering de los tetraedros. Los valores de proyección obtenidos del *hexunits* son los datos necesarios y suficientes para calcular la clasificación del resto del volumen, y para esta labor utilizaremos el alto grado de paralelismo que encontramos en el procesador gráfico GPU como el medio más eficiente.

La GPU comparada con la CPU no está fácilmente disponible para el programador para realizar cálculos de manera general, pero existe una técnica denominada algoritmo de propósito general en GPU (GPGPU) que nos permite aprovechar las capacidades de cómputo de la GPU para cálculos de este tipo. Una forma de hacerlo es a través de la librería OpenGL, la cual es la librería que utilizamos para la salida final en pantalla y en este momento haríamos lo mismo pero para realizar los cálculos que necesitamos. Entonces de manera general lo que se hace es renderizar una textura en un cuadrilátero del tamaño de la ventana de salida o *viewport*, de manera que el número de *texels* sea el mismo que el número de pixeles de la ventana, así al rasterizar el cuadrilátero, cada *texel* de la textura es un fragmento en la pantalla.

Para el procesamiento, pasamos a la GPU dos texturas, en la primera en cada *texel* queda*,* el centroide del tetraedro y su índice con respecto a la segunda textura, y en la segunda la información de los escalares de los vértices de cada tetraedro, quedando en cada *texel* de la imagen la información volumétrica de un tetraedro. En la figura

17 se escribe brevemente el proceso en el *fragment shader* de búsqueda de la información de los tetraedros y su desplazamiento al ser procesados.

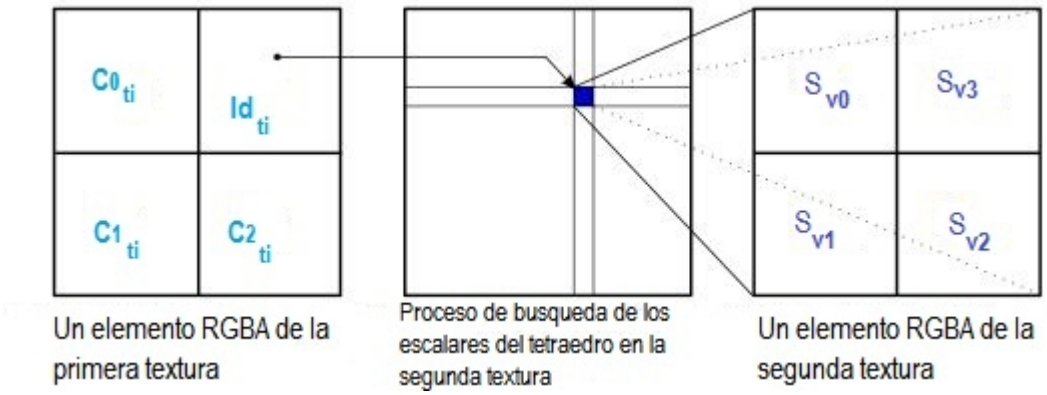

Figura 17: Búsqueda de los escalares en los vértices de cada tetraedro en el fragment shader. Cada pixel en la primera textura contiene un índice que corresponde con un tetraedro en la segunda.

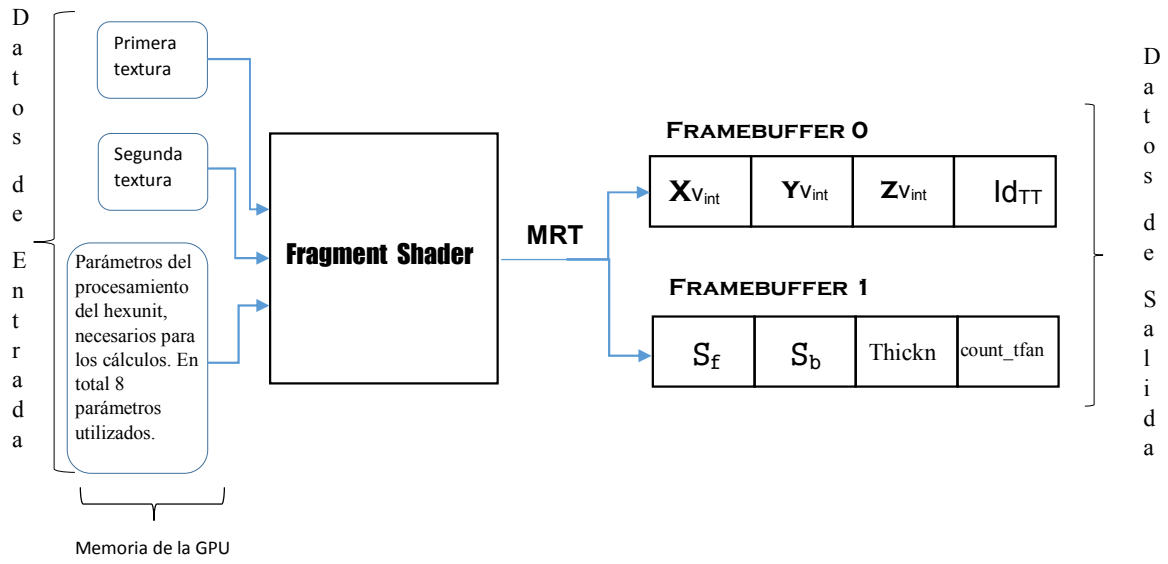

Figura 18: Entrada y salida en el fragment shader en el proceso de clasificación del resto del volumen, donde Id_{TT} es el índice en la tabla ternaria y MRT significa *Multiple Render Targets* con dos buffers de color adjuntos.

Los datos y parámetros obtenidos del proceso con los *hexunit* son pasados como variables uniformes al *fragment shader*, para el correcto desplazamiento de cada tetraedro al ser procesado. En la figura 18 describimos resumidamente el proceso total de esta clasificación así como el resultado que se espera. De esta manera clasificamos la proyección del volumen total.

Cuando el programa se encuentra en el proceso anterior de clasificación, en paralelo se realiza el ordenamiento de los tetraedros para la composición de la imagen final. Una desventaja del algoritmo PT es que las primitivas deben ser renderizadas de forma ordenada, por lo general back-to-front. Ordenar millones de tetraedros es una tarea que debe realizarse eficientemente para no perjudicar el rendimiento del programa. La manera que el programa realiza el ordenamiento es aprovechando el hecho de que el algoritmo PT es un algoritmo en el que se itera y se proyectan las celdas de volumen sobre el espacio de la pantalla y las compone en un orden de visibilidad determinado por una regla de recorrido del volumen. Por tanto la regla de recorrido más eficiente que se conoce es la de visualizaciones de cortes alineados a los ejes como se vio en el capítulo 1, en el cual se puede visualizar un conjunto de datos 3D mostrando un gran número de cortes 2D semitransparentes extraídos de este. En este caso no vamos a extraer cortes sino que vamos a utilizar la configuración de composición de esta técnica para ordenar los tetraedros de una manera más rápida con respecto al punto de visión. Dividimos entonces el mallado tetraédrico en cortes de *hexunit* en la dirección que indique la dirección de visión actual. La dirección

de los cortes de *hexunit* se toma de manera que minimiza el ángulo con el vector de visualización tomado. De esta manera al determinar la dirección de los cortes de *hexunit* en que se debe dividir el volumen, se puede utilizar entonces la función '*qsort*', de c++ 11 de una manera eficiente debido a que existe cierto orden por estar realizando el ordenamiento en la dirección de atravesamiento del volumen, de atrás hacia delante como indica la composición del algoritmo. La dirección en que el volumen debe dividirse en cortes de *hexunit* lo determina un algoritmo que al tener la transformación del volumen nos indica la dirección en que debe recorrerse el mismo, el resumen de lo anterior explicado se puede ver en la figura 19.

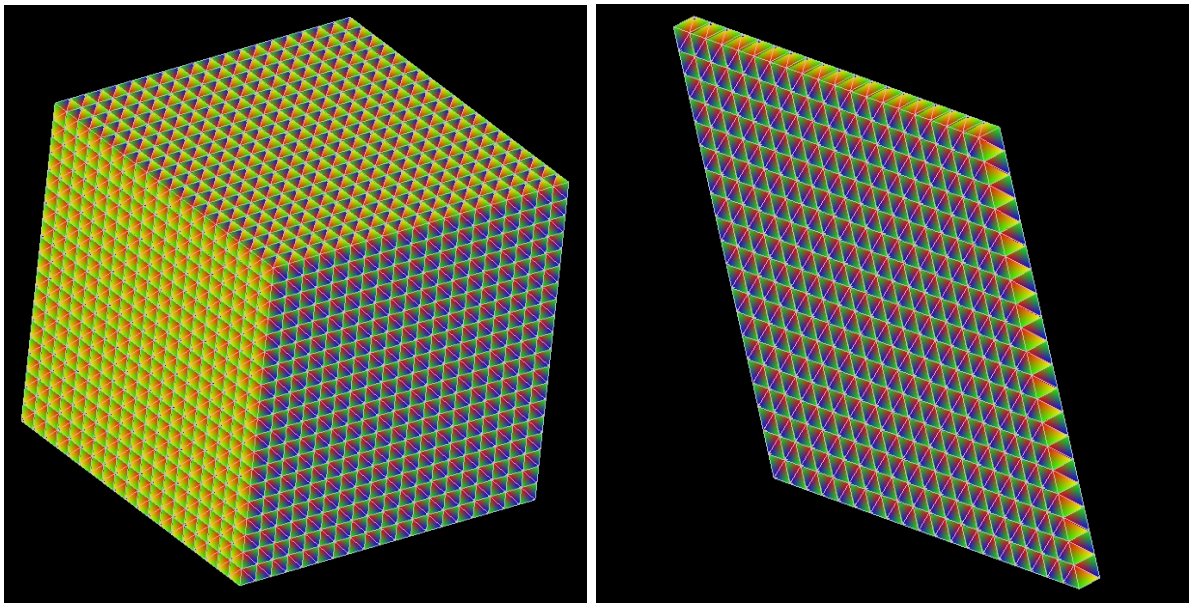

Figura 19: A la izquierda el mallado tetraédrico del volumen en el cual se puede ver como un conglomerado de *hexunit,* conglomerado que se puede dividir en capas como se puede ver a la derecha. Todo este proceso es transparente, la dirección de la división de los cortes es determinado por un algoritmo que decide si los cortes deben ser tomados paralelo a el eje *z*, *x* o *y.*

1.1.3 Desplegar Volumen

Para renderizar los datos resultantes del proceso anterior, cada tetraedro es desplegado o dibujado como un abanico de triángulos. Las primitivas se dibujan en el orden back-to-front y los fragmentos resultantes se componen de píxeles utilizando una función de mezcla o blending. Después de ordenar los tetraedros, la información obtenida se organiza para ser desplegada con la función optimizada de OpenGL *glMultiDrawElements*, cuyos argumentos referencia a los arreglos globales que almacenan la información de los vértices. En particular el programa hace uso de dos arreglos globales que almacenan las coordenadas de los vértices y los valores de los escalares de los vértices (colores) respectivamente.

Los dos arreglos globales contienen los tetraedros agrupados en cinco elementos: El *thick vertex* mas los cuatros vértices originales. Estos arreglos tienen las dimensiones de un *hexunit*, los cuales son actualizados constantemente al ir llenando la memoria de video con la información de salida del procesamiento anterior explicado. La memoria de video se hace disponible a través de los objetos de la librería OpenGL *Vertex Array Object (VAO)* y *Vertex Buffer Object (VBO).* Cada elemento del arreglo de vértices contiene la coordenadas $\{x, y, z, w\}$ del vértice. El arreglo de colores contiene los valores $\{s_f, s_b, l\}$ por cada vértice, en lugar de los colores reales, los cuales se calcularán *on-the-fly*, en el *fragment shader.* Hay que recordar que en los vértices *thin* (los vértices que no son *thick*) $s_f = s_b$ y $l = 0$.

Para gestionar el orden correcto de los vértices en el abanico de triángulos, se necesitan dos arreglos adicionales. El arreglo de índices se divide en grupos, cada uno con seis elementos que permiten almacenar el orden correcto de los vértices del abanico que sirve para dibujar un tetraedro (es decir igual a una matriz de dimensiones N°de tet x 6). El arreglo *count* contiene el número de vértices en cada abanico. Recordemos que el número máximo de vértices en un abanico es seis (caso de clase 2). Estos dos arreglos son los parámetros de la función *glMultiDrawElements,* la cual junto con los arreglos de vértices y colores explicados anteriormente, almacenados en memoria de video, producen la visualización del volumen, cumpliendo hasta aquí el programa con el objetivo planteado. En la figura 20 podemos ver lo explicado en esta sección correspondiente al rendering de la salida del proceso anterior. En cada transformación del volumen el proceso de clasificación actualiza todos los buffers mostrados y utilizados en esta sección para realizar la visualización del volumen, por lo que en la figura se describe el proceso de la indexación de los buffers cuando se trata de desplegar un tetraedro. Ver figura 20 para más detalles.

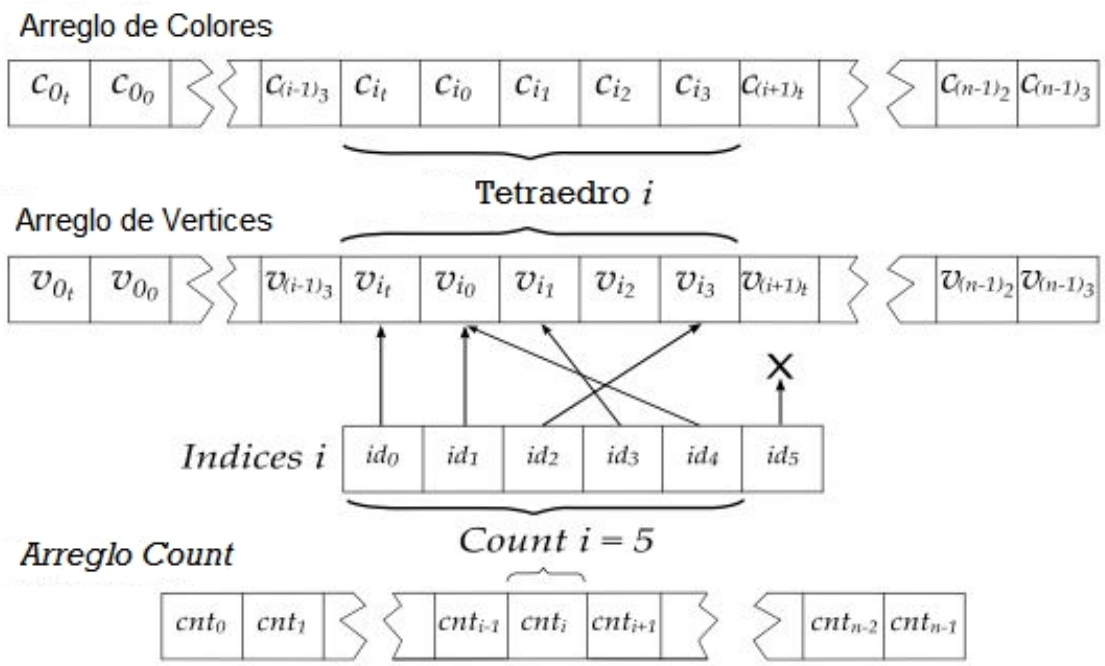

Figura 20: Arreglos de las estructuras de datos utilizados por la función *glMultiDrawElements*. Los índices ilustran el caso de desplegar un tetraedro clase 1, donde el correcto orden para dibujar el tetraedro i es Vi_t - Vi_0 – Vi_3 – Vi_1 – Vi_0. En este caso unos de los vértices del tetraedro es el vértice thick.

1.2 Diagrama del Programa

Luego de describir el funcionamiento del programa, se plantea un diagrama que resume la arquitectura de funcionamiento del sistema. El diagrama del sistema orientado a objetos utilizado para el diseño se presenta a continuación.

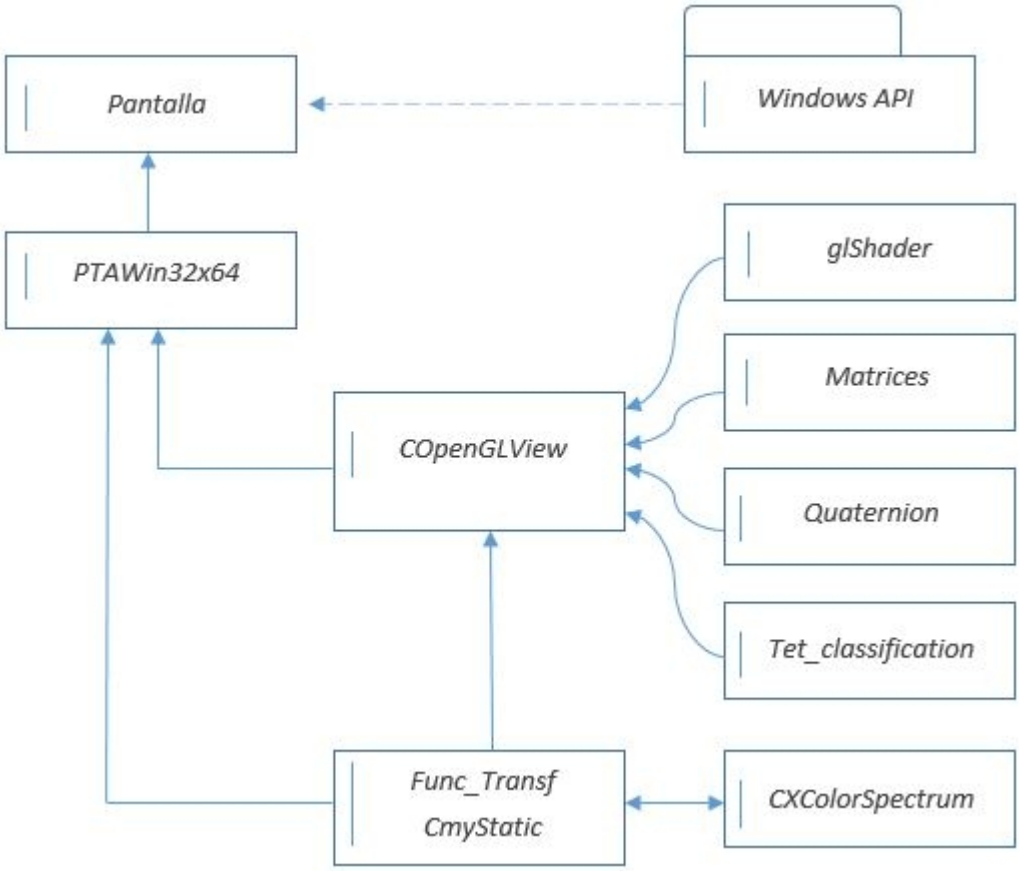

Figura 21: Diagrama de la aplicación con las clases que lo componen.

A continuación se describen las clases usadas por el sistema:

PTAWin32x64: Es la clase principal encargada del despliegue, además es la responsable de la lógica general del sistema. En esta clase se encuentran contenidas estructuras globales utilizadas en los diversos cálculos de transformación y clasificación del volumen. Pero la principal importancia de esta clase es que en ella se define el contexto de la librería OpenGL® (Open Graphics Library), que como se mencionó es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D, utilizado principalmente para generar la visualización del volumen en la aplicación.

COpenGLView: Es la clase que representa el corazón del programa y es creada solo cuando en la clase anterior se han cumplido los requisitos para crear el contexto de OpenGL. En esta clase se realizan los cálculos, transformaciones y actualizaciones necesarias para el funcionamiento del programa, además que es la encargada de manejar las funciones de la librería OpenGL para el despliegue del volumen. Muchas o la mayoría de las funciones realizada por esta clase las realiza gracias al soporte de otras clases que describiremos brevemente a continuación.

glShader: Es una clase de soporte de la clase anterior en la cual se define la extensión de OpenGL® GLSL (OpenGL® Shading Language), la cual nos permite ejecutar bloques de códigos directamente en la GPU llamados *shader*. Los *shaders* utilizados en el programa han sido el *shader* de vértices o *vertex shader* y el de fragmento *fragment shader*. Los *shaders* son muy importantes en este programa ya que nos permite programar y modificar el despliegue realizado a través de la librería OpenGL®.

En esta clase también se inicia la librería GLEW (OpenGL Extension Wrangler Library) [41] es una librería multiplataforma escrita en C/C++, destinada a ayudar en la carga y consulta de extensiones de OpenGL®; es utilizada en la aplicación para facilitar el manejo del OpenGL®.

Matrices: Clase de soporte en donde se implementa estructuras y funciones relacionadas con una Matriz algebraica. Las matrices son implementadas del modo como son utilizadas por la librería OpenGL® transpuesta a la forma normal de una matriz. Las matrices son la

matemática utilizada por OpenGL® para simular las trasformaciones de los objetos.

Quaternion: Clase de soporte que implementa la matemática de la transformación de rotación. La matemática se basa en Quaternion que son una extensión de los números reales, similar a la de los números complejos, pero su uso aquí en particular es para los cálculos que implican rotaciones tridimensionales. La clase permite generar matrices que son usadas por la clase anterior y la clase COpenGLView para simular la transformación del volumen.

Tet_Classification: Casi como su nombre lo indica, en esta clase se implementa la clasificación de los tetraedros que representan la unidad volumétrica del volumen a visualizar. La clasificación como se debe recordar es realizada sobre los cinco *tetraedros base* y es de aquí donde se obtiene los parámetros de clasificación necesarios para clasificar el resto del volumen. Esta clase representa el soporte principal del programa ya que en ella se encuentra implementado gran parte del algoritmo PT.

Func_Trans: Es la clase en donde esta implementada la interfaz y funcionamiento de la función de transferencia editable por el usuario. Esta clase se inicia cuando la clase principal del programa inicia la ventana interfaz de la función de transferencia y es solo accesible cuando se encuentra cargado un archivo de volumen válido por el programa. La siguiente imagen resume la función de esta clase.

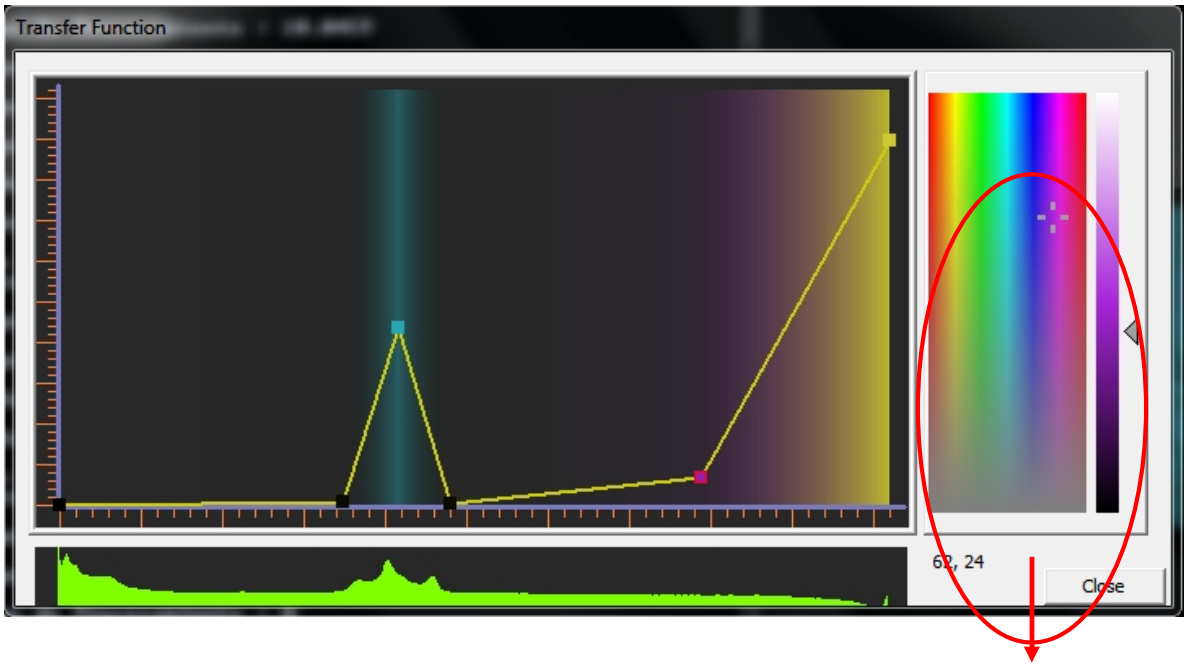

Figura 22: Interfaz de la función de transferencia unidimensional

Windows API: Este módulo en realidad no es una clase y está encargado de manejar la interfaz en el sistema. La **interfaz de programación de aplicaciones de Windows**, cuyo nombre en inglés es **Windows API** (*Windows application programming interface*), es un conjunto de funciones residentes en bibliotecas (generalmente dinámicas, también llamadas DLL por sus siglas en inglés, término usado para referirse a éstas en Windows) que permiten que una aplicación se ejecute bajo un determinado sistema operativo, en este caso sistemas operativos Microsoft Windows.

Pantalla: Este módulo tampoco es una clase, representa la correcta compilación de la interfaz del programa, que corresponde con la ventana principal del programa encargada de proporcionar la interfaz

al usuario manejada y proporcionada por el SO por medio del módulo anterior, siendo estas ventanas de aplicación, consolas o diálogos. En este caso están presente las dos primeras como podemos observar a continuación.

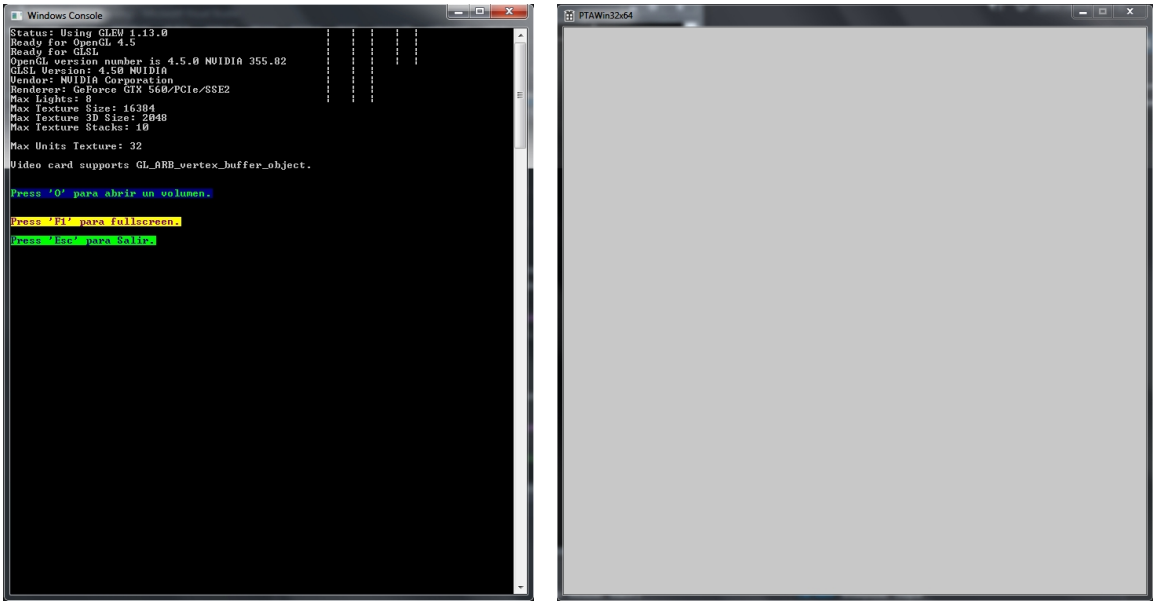

Figura 23: Interfaz del programa, a la izquierda consola y a la derecha ventana de salida.

La consola muestra información de estado, ayuda y cualquier información de relevancia de funcionamiento del programa. La ventana de despliegue es donde se dibuja el resultado y se visualiza el volumen manejado por el programa.

1.3 Programa sin la función optimizada '*glMultiDrawElements*'

La función '*glMultiDrawElements*' es una función que hasta el momento había permitido el despliegue de los tetraedros de una manera rápida ya que nos ayuda en la configuración de dibujado del

abanico de triángulos en que se dividen los tetraedros según su clasificación. Esta ayuda somete al programador y al programa a unos requerimientos que deben ser proporcionados para la ejecución correcta de la función. Estos requerimientos están compuesto de las estructuras de datos explicadas anteriormente en este capítulo en la sección 1.1.3, las cuales revisándolas y sacando cuenta de la cantidad de elementos que contiene, podemos ver la cantidad de recursos de memoria que consume, esto es debido a la limitación que además la función impone al programa la cual es la de dibujar o desplegar 1 tetraedro por vez. Entonces sumando todo esto dicho hasta ahora, la ayuda de la función es degradada por la condiciones de funcionamiento de la misma.

1.3.1 Pipeline Gráfico y Nueva Solución

Un pipeline, en términos de computación, se refiere a una serie de etapas de procesamiento en el que la salida de una etapa es el alimento de entrada de la siguiente etapa, similar a una línea de montaje de una fábrica o una tubería de agua / aceite. Con paralelismo masivo, el pipeline puede mejorar en gran medida el rendimiento general.

El pipeline de rendering gráfico 3D acepta la descripción de objetos 3D en términos de vértices de primitivas (como el triángulo, el punto, la línea y el cuadrilátero), y produce el color-valor de los píxeles en la pantalla.

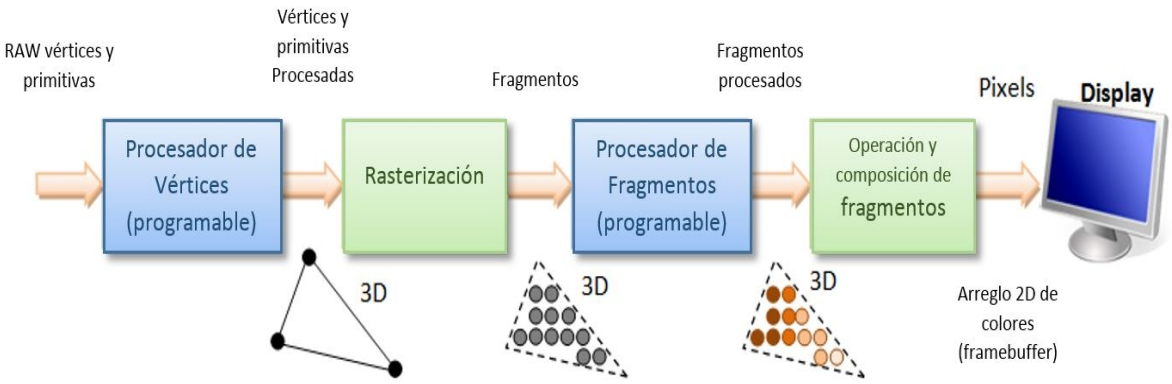

Figura 24: Pipeline gráfico utilizado hasta los momentos.

El pipeline de rendering gráfico 3D consta de las siguientes etapas principales:

1. Procesador de vértices: Procesa y transformar vértices individuales.
2. Rasterización: convierte cada primitiva (vértices conectados) en un conjunto de fragmentos. Un fragmento puede ser tratado como un píxel del espacio 3D, que está alineado con la cuadrícula de píxeles, con atributos como la posición, color, normal y textura.
3. Procesador de Fragmentos: Procesa fragmentos individuales.
4. Composición de salida: Combina los fragmentos de todas las primitivas (en el espacio 3D) en colores de píxeles 2D para la visualización.

En las GPU modernas, las etapa de procesamiento de vértices y procesamiento de fragmentos son programables. Se Puede escribir programas, conocidos como *vertex shader* y *fragment shader* para llevar a cabo transformación personalizada de vértices y fragmentos. Los programas shader están escritos en lenguajes de alto nivel iguales

a *C* como GLSL (OpenGL Shading Language), HLSL (lenguaje shader de alto nivel para Microsoft Direct3D), o CG (*C* para Gráficos de NVIDIA).

El uso de la función '*glMultiDrawElements*' junto con los módulos *vertex shader* y *fragment shader* del pipeline gráfico 3D, ha permitido mejorar el rendimiento del programa, al poder fusionar gracias a estos módulos, el proceso de clasificación con el de rendering. Por tanto se pudo eliminar los buffers de salida del proceso de clasificación (GPGPU), y con esto eliminar otros buffers de creación del mallado y generar los vértices en el proceso de rendering en el momento de ser utilizados. Pero a pesar de que mejoró notablemente el rendimiento del programa, seguía padeciendo consumo excesivo de memoria por el uso de la función '*glMultiDrawElements*'. Esto sería la segunda versión del programa y el final del uso de la función, ya que al estudiar el nuevo pipeline gráfico, se encontró un nuevo módulo programable que permite el rendering de los tetraedros disminuyendo principalmente la geometría pasada de manera explícita al hardware programable y por consiguiente el consumo excesivo de memoria por el proceso de rendering.

La función '*glMultiDrawElements*' como se dijo anteriormente impone al programa, pasar cada vértice de las primitivas a desplegar de manera explícita, y en el caso que nos concierne, de esto podemos deducir que también impone el poder dibujar o hacer rendering de un solo tetraedro a la vez. El nuevo módulo o estado del pipeline gráfico que nos va a permitir cambiar las cosas es conocido como *geometry shader.* Un *geometry shader* puede generar nuevas primitivas gráficas,

como los puntos, las líneas o los triángulos, estas primitivas creadas son enviadas al principio del pipeline gráfico. Los programas de geometry shader son ejecutados después de los vertex shaders. Toman como entrada toda la primitiva, a ser posible con información adjunta. Por ejemplo, cuando se operan triángulos, los tres vértices son la entrada del geometry shader. El shader puede emitir primitivas, que son rasterizadas y sus fragmentos al final son pasados al fragment shader. Un ejemplo típico y muy usado, de los beneficios que aporta los geometry shaders podría ser la modificación automática de la complejidad de una malla.

Un geometry shader (abreviado GS) es un modelo de programación de shader introducido con Shader Model 4.0 de DirectX 10. Las primeras tarjetas gráficas en soportar Geometry Shaders fueron las GPUs NVIDIA GeForce 8800.

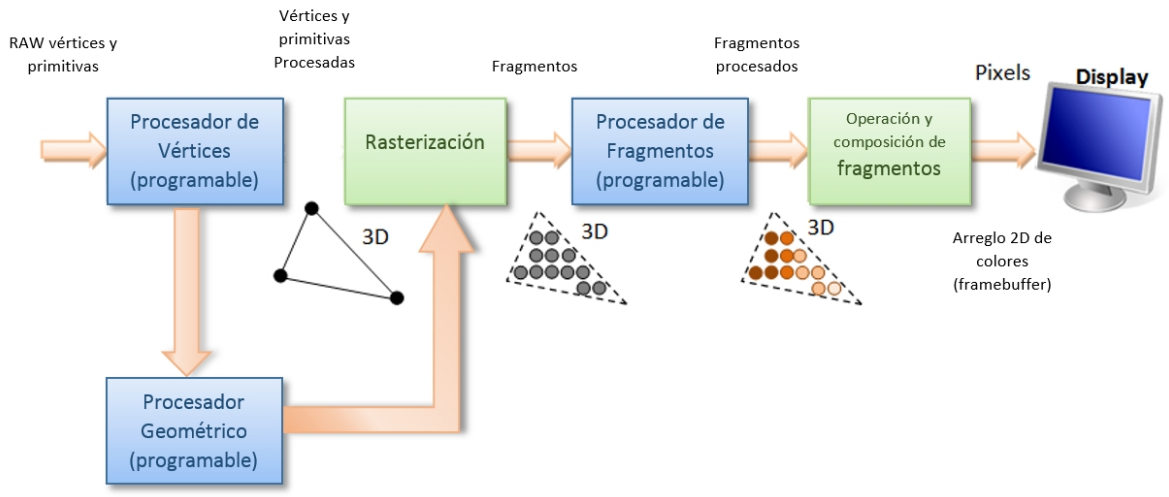

Figura 25: Pipeline gráfico con el nuevo módulo programable.

Con las etapa de procesamiento geométrico, es posible ahora hacer cambios considerable al rendimiento del programa tanto en términos

de consumo de memoria como en la interacción con el mismo, ya que el programa muestra una mejora considerable en los cuadros por segundos, como veremos en el capítulo siguiente.

Lo realmente resaltante en este momento a destacar es la mejora en el consumo de memoria, la etapa de procesamiento geométrico ha dado la libertad al programa de generar la geometría que necesita para poder renderizar los tetraedros en el momento de la ejecución del proceso de rendering en el pipeline gráfico. Es decir lo que se hacia fuera del pipeline gráfico para llenar los buffer que necesitaba la función '*glMultiDrawElements*' para realizar el rendering del volumen, ahora es realizado en el pipeline gráfico. Lo que se traduce en una eliminación de la geometría explicita pasada como atributos para realizar el rendering del volumen. Ahora con solo pasar un atributo por tetraedro o por un conjunto de tetraedros es suficiente.

La mejora en consumo la podemos resumir de esta manera, con la función '*glMultiDrawElements*' para hacer rendering de un volumen de dimensiones 256^3 era necesario pasar al hardware programable como mínimo alrededor de unos 6650 Mb de atributos, cosa que debería ser pasado en varias partes por obvias capacidades de memoria. Sin el uso de la función y utilizando el procesador geométrico del hardware programable el consumo para un volumen de 256^3 paso de la cantidad anterior a 663,25 Mb para hacer rendering de un tetraedro a la vez en el *geometry shader,* y de solo 132,65 Mb al hacer rendering de 5 tetraedros a la vez, pudiendo reducirse el consumo si se desea al hacer rendering de más tetraedros a la vez.

Capítulo IV. PRUEBAS

En este capítulo se muestran pruebas cualitativas y cuantitativas del programa implementado, evaluando así los objetivos propuestos, sobre una plataforma basada en Windows.

1. Descripción del ambiente de pruebas.

El sistema requiere de un hardware especializado para su ejecución; para realizar las pruebas se utilizó un PC con procesador Intel(R) Core(TM) i7 3770 3.40 GHz (4 núcleos/8 hilos), 8GB de memoria RAM y un sistema operativo Windows 7 de 64 bits. La tarjeta gráfica es basada en NVidia, con chip Nvidia GTX 660, 2GB DDR5 de memoria, y un bus interfaz PCI Express 16x. Los volúmenes utilizados en las pruebas tienen el formato descrito en el capítulo 3, sección 1.1.1; con precisiones de muestreo de 8 bits. Las características de los volúmenes son presentados en la Tabla 1.

Volumen	fuel	dti-fa	blunt	Baby
Dimensiones	64x64x64	128x128x58	256x128x64	256x256x98
# de Bits	8	8	8	8
Tamaño en MB	256 Kb	928 Kb	2048 Kb	6272 Kb
# de Tetraeds.	1.25 M	4.59 M	10.2 M	31.86 M

Tabla 1: Características de los volúmenes utilizados en el ambiente de pruebas.

Cada uno de estos volúmenes fue obtenido de “Volvis.org. (2011, Septiembre) The Volume Library. [Online]. http://www9.informatik.unierlangen.de/External/vollib/”

Para cada uno se realizó la visualización del volumen, utilizando el algoritmo implementado y evaluando tanto el tiempo requerido para el

procesamiento de los datos del mallado (sorting, llenado del buffer de rendering) así como el empleado para realizar la salida en pantalla.

2. Resultados cuantitativos

A continuación se mostrarán los resultados obtenidos de las pruebas realizadas sobre los volúmenes descritos en la Tabla 1, para finalmente realizar comparaciones cuantitativas de los resultados obtenidos. Todos los tiempos se tomaron con un viewport de 810^2, además de tomar en cuenta que el modelo está constantemente girando.

Los Volúmenes utilizados fueron: El *CT Scan* de un inyector de combustible, la tomografía de un cerebro, el Blunt Fin (blunt) de la NASA's NAS website y el CT Scan de la cabeza de un bebe.

Data set	# Verts	# Tet	spf	fps
fuel	262 K	1,25 M	0,2176	4,6
Dti-fa	950 K	4,59 M	0,714950	1,4
blunt	2 M	10,2 M	1,557990	0,64
baby	6,4 M	31,86 M	4,839150	0,21
	Usando el	**procesador**	**Geométrico**	
fuel	262 K	1,25 M	0,065	15
Dti-fa	950 K	4,59 M	0,178	5
blunt	2 M	10,2 M	0,417	2
baby	6,4 M	31,86 M	1,428	0,70

Tabla 2: Promedios de cuadros por segundos presentados por los distintos volúmenes cargados, usando la función '*glMultiDrawElements*' y el módulo *geometry shader* 1 tetraedro por vez.

El número de vértices (# Verts) y tetraedros (# Tet) depende de las dimensiones del conjunto de datos. La Medición de los resultados se

dan en cuadros por segundo (fps) y en segundos por cuadros (spf). Los valores presentados para *spf* y *fps* son el promedio de unos 100 cuadros mostrados por el programa por cada volumen.

De los resultados presentados en la tabla 2 se puede deducir que el tiempo de ejecución del algoritmo es directamente proporcional al tamaño del volumen presentado ya que a mayor número de voxeles o tetraedros presentes, mayor es el tiempo empleado para presentar la salida en pantalla.

Analizando con más detalle los tiempos de la tabla anterior tenemos a continuación los tiempos de procesamiento (*Setup*) (sorting y llenado del buffer de rendering) y el de Rendering utilizado por el programa, los cuales sumados cabe destacar equivalen de manera aproximada a los tiempos *spf* presentados en la tabla 2 para cada volumen.

Data set	*Setup*	*Render*	% Total
fuel	0,081985	0,134987	62,2
Dti-fa	0,310577	0,404332	56,56
blunt	0,703580	0,852750	54,8
baby	0,350444	4,493661	92,8
Usando	**el**	**Procesador**	**Geométrico**
fuel	0,024	0,065	63
Dti-fa	0,075	0,178	57
blunt	0,207	0,417	50
baby	0,477	1,428	67

Tabla 3: Tiempos de procesamiento y Rendering de los distintos volúmenes cargados, usando la función '*glMultiDrawElements*' y el módulo *geometry shader* 1 tetraedro por vez.

Los tiempos promedios de procesamiento y rendering de 100 frames del programa son mostrados en la tabla 3. El porcentaje mostrado en la última columna nos indica el porcentaje del tiempo total utilizado por cada volumen para hacer Render. En el algoritmo implementado, podemos observar que el tiempo promedio de rendering de los volúmenes mostrados es aproximadamente del 60% del tiempo total, al sacar el promedio de la última columna. Pero para finalizar el análisis, en el siguiente gráfico se muestra realmente el comportamiento del programa con los distintos volúmenes cargados. Mostraremos el comportamiento de los tiempos de procesamiento a medida que aumenta el número de tetraedros, ya que en el procesamiento radica el algoritmo implementado. Además mostramos el comportamiento de los tiempos de rendering a medida que aumenta el número de tetraedros.

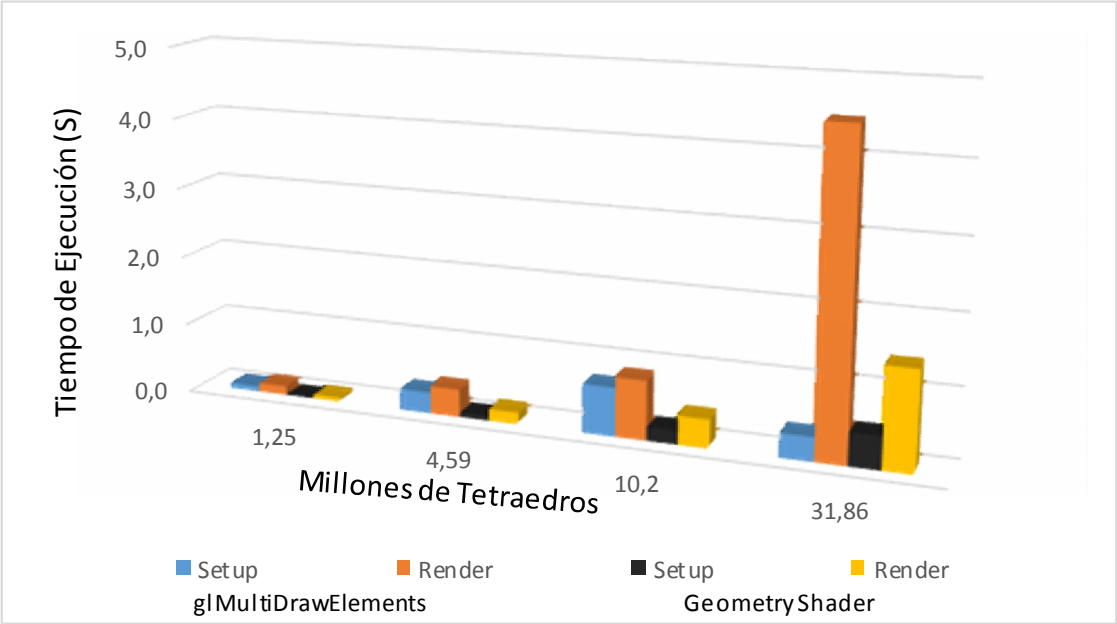

Gráfico 1: Comportamiento de los tiempos de procesamiento y Rendering de los distintos volúmenes cargados.

En el gráfico se puede observar que a medida que aumenta el número de tetraedros el algoritmo tiene un buen comportamiento, mientras que el tiempo de hacer *rendering* empeora significativamente sobre todo cuando el volumen a visualizar no cabe completamente ya sea en memoria de video o RAM. Por tanto es interesante observar una mejora en el procesamiento en el volumen de prueba tomado, cuyo tamaño no cabe completamente en memoria de despliegue. Este fenómeno realmente puede explicarse como un efecto producido en el desempeño de la prueba, el *overhead* del proceso de *rendering*, ya que al no caber completamente el volumen en memoria, ya no se puede contar en el procesamiento, el llenado del buffer de salida (con Geometry Shader si se incluye), y por tanto pasa a ser parte del proceso de *rendering* o despliegue. Lo que nos indica que a pesar de ser un proceso rápido para el computador, el llenar los buffer de salida, estos buffer tienen que ser llenados por tetraedros, los cuales tienen 5 vértices por cada uno y cada vértice tiene 4 componentes, lo que nos da unos números astronómicos de elementos contenidos por estos buffers, y por tanto a pesar de ser rápido consume un tiempo que considerar en cada rendering.

Por lo último comentado debemos pasar entonces a comparar los costos de consumo de recursos para terminar de puntualizar la mejora de la utilización del procesador geométrico con respecto a la ya obsoleta función '*glMultiDrawElements*', y sin más preámbulos pasamos a presentar la tabla que muestra la relevante mejora de la que se habló al final del capítulo pasado.

Consumo de Memoria		
Data set	*glMultiDrawElements*	*Procesador Geométrico*
fuel	100,1 Mb	10 Mb
Dti-fa	367,74 Mb	36,77 Mb
blunt	816,1 Mb	81,61 Mb
baby	2,52 Gb	252,3 Mb

Tabla 4: Comparación de los consumos de recursos entre la función glMultiDrawElements y el uso del *geometry shader*

Ya comparada la eficiencia del uso del procesador geométrico y la obsolescencia de la función optimizada de OpenGL, solo nos queda comparar el uso intensivo del procesador geométrico, al pasar de 1 tetraedro a la vez a 5 a la vez, es decir un hexaedro por primitiva pasada al hardware gráfico. Entonces sin más preámbulos pasamos a mostrar las tablas con algunos de los volúmenes usados en las pruebas.

Consumo de Memoria		
Data set	*1 Tetraedro a la vez*	*5 Tetraedros a la vez*
fuel	10 Mb	2 Mb
blunt	81,61 Mb	16,3 Mb
baby	252,3 Gb	50,4 Mb

Tabla 5: Comparación de los consumos de recursos al hacer rendering de 1 tetraedro a la vez y 5 tetraedros a la vez con el soporte del *geometry shader*

Rendimiento		
Data set	*1 Tetraedro a la vez*	*5 Tetraedros a la vez*
fuel	15 f/s	14 f/s
blunt	2 f/s	1 f/s
baby	0,80 f/s	0,45 f/s

Tabla 6: Comparación del rendimiento del programa al hacer rendering de 1 tetraedro a la vez y 5 tetraedros a la vez con el soporte del *geometry shader*

Para terminar podemos decir al observar los resultados obtenidos que es satisfactorio encontrar similitud con los trabajos revisados sobre el tema; también pudimos comprobar las mejoras que se han obtenido en el rendimiento y en el consumo de recursos gracias a la utilización de un módulo adicional hardware programable en este trabajo. Quedando como un hecho que la introducción de la programación del procesador geométrico es parte de la implementación de este algoritmo para su funcionamiento óptimo. Otro punto importante fue la última prueba realizada al usar el procesador geométrico de una manera más intensiva, ya que esta prueba muestra que los resultados son contrarios en términos de rendimiento, a lo que parecía ser mejor según la lógica. Los resultados muestran que a pesar del paralelismo, pasar más trabajo a la vez a los procesadores shader del hardware gráfico no representa ninguna ventaja.

Los tiempos observados también ponen en evidencia la mejora al algoritmo en si como al calcular la clasificación sobre un hexaedro base y los parámetros resultantes utilizarlos para calcular las interpolaciones de los colores de los vértices thick del resto del volumen en el mismo proceso de rendering, mejoras que han sido

predichas por los autores originales, gracias a las características programable del hardware gráfico. Cabe señalar que a medida que el tiempo empleado en hacer rendering del volumen se vuelve más cercano al 100%, los tiempos del algoritmo estarán sujetos al rendimiento de la tarjeta gráfica [30].

3. Resultados cualitativos

A continuación se presentan los resultados visuales de la aplicación con las distintas funciones de transferencias utilizadas en cada volumen al hacer las pruebas anteriormente mostradas.

En la figura 26 se puede apreciar el resultado de la visualización de los volúmenes de prueba, y se puede decir que la visualización es satisfactoriamente muy similar al de los trabajos estudiados sobre el tema, a pesar que esta implementación se limitó a presentar el volumen con una función de transferencia simple de opacidad y color, que por ser simple no utiliza texturas intermedias o tablas precalculadas que mejoren la calidad visual. Por tanto todo esto supone que el orden de visualización y la composición realizada en la implementación han sido correctas.

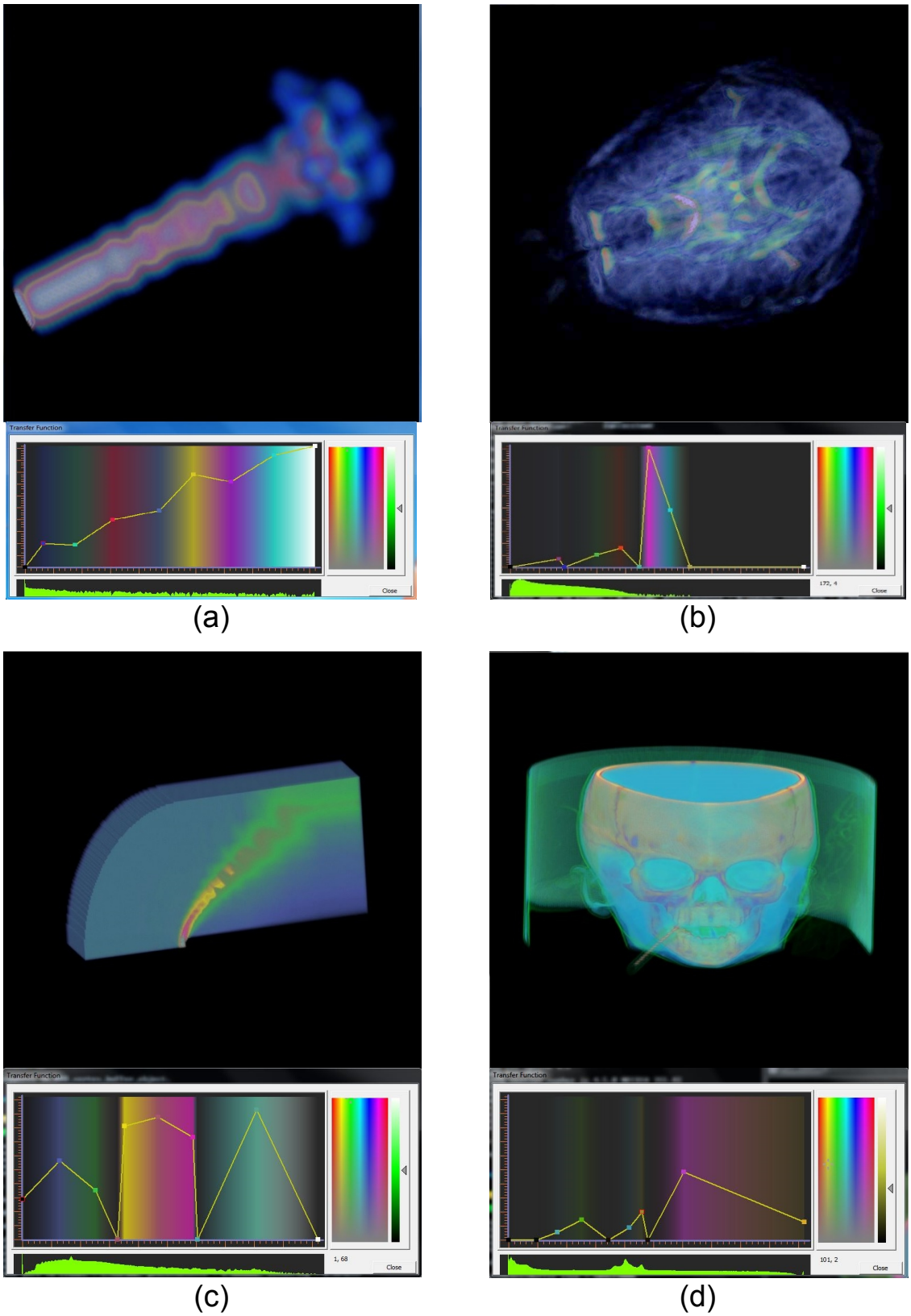

Figura 26: Volúmenes y funciones de transferencias utilizadas: fuel (a), Dti-fa (b), blunt (c) y baby (d).

Capítulo V

Conclusiones y Trabajos futuros

En este trabajo se diseñó e implementó un sistema de visualización directa de volumen, basado en la técnica de proyección de celdas, en este caso tetraédricas, que toma ventaja de la regularidad de los datos. El sistema cuenta con una interfaz que permite la interacción del usuario con el volumen, además la edición de la función de transferencia a través de la manipulación de puntos de control.

El algoritmo de Tetraedros proyectados presentado en este trabajo aproxima la visualización volumétrica de las celdas tetraédricas por el despliegue e interpolación de triángulos parcialmente transparentes. Esta aproximación se lleva a cabo mediante el despliegue de triángulos que dibuja la silueta de proyección de un tetraedro, y linealmente interpolando color y opacidad en los triángulos desde la silueta totalmente transparentes de los bordes a los valores en la parte más gruesa del tetraedro, como se ve desde el punto de vista. Actualmente, cada hexaedro se divide en cinco tetraedros, cada uno de los cuales se representa en el peor de los casos con 4 triángulos o un máximo de 20 triángulos por *hexunit*.

Los puntos fuertes del método son que los triángulos o primitivas tetraédricas son renderizados e interpolados directamente en la GPU de las tarjetas gráficas de la generación actual. Se pueden utilizar punto de vista ortográfico o perspectivos [37], y que la geometría del

mallado no estructurado es generado con un orden de profundidad conocido, permitiendo esto organizar más fácil el volumen en un orden de visibilidad. Pero el punto importante en términos de algoritmo en sí, es el hecho de que también tiene requisitos de memoria muy pequeños ya que los únicos datos necesarios son para el tetraedro que se está procesando actualmente. Pudiéndose entender entonces que los tetraedros se pueden procesar de forma independiente, por lo que el algoritmo de Tetraedros proyectados pueden ser implementado en paralelo, siendo esto la característica según mi punto de vista la más importante ya que aporta al método el interés necesario para su desarrollo. Además podemos decir que no hay estructuras de datos adicionales creadas aparte de los propios datos de volumen, es decir el algoritmo no requiere preprocesamiento y no requiere estructuras de datos adicionales.

La implementación con geometry shader reduce el costo de memoria en un factor de 10x, y acelera la rata de FPS entre 3x y 4x, al compararlo con la versión inicial sin geometry shaders (glMultiDrawElements). Además se pudo ver que sobrecargar la etapa de geometry shader para que genere 5 tetraedros (en vez de 1) reduce el consumo de memoria en un factor de 5x, pero aumenta el tiempo de respuesta en hasta 2x, debido a que el exceso de cómputo en la primera fase del pipeline reduce el rendimiento global del pipeline.

Las debilidades del método Tetraedros proyectados son diversos y se concentran en algunos puntos de su ejecución, por lo que podemos decir que estas debilidades forman parte de los trabajos a futuros.

Podemos empezar por el tiempo empleado para el rendering de visualización; este consume aproximadamente el 60% del tiempo usado por el método. Un trabajo a futuro era investigar ideas que permitiesen renderizar menos polígonos por *hexunit,* pero esto ya se realizó al utilizar el *geometry shader*, dando esto la posibilidad de almacenar los datos de un volumen de 256^3, en memoria de video haciendo el proceso de visualización realmente rápido para la GPU.

Como trabajos a futuro se propone realizar la clasificación de los tetraedros, *sorting* y transferencia de datos a los buffer de rendering en algún lenguaje paralelo provisto por el hardware gráfico (OpenCL o CUDA), en donde se permita reducir el tiempo de cálculo y transferencia de información.

Por último, el estudio de la insuficiente precisión utilizado para la composición debido al uso de un *framebuffer* de 32bits, el cual introduce artefactos en la interpolación de triángulos muy pequeños [36][26], así como técnicas de mejoramiento de la calidad de la visualización como por ejemplo la conocida pre-integración, aportará a este algoritmo ventajas importantes a medida que las GPU vayan siendo más rápidas y flexibles.

Referencias

[1]. Mark Levoy. “Display of surfaces from volume data”. IEEE Computer Graphics and Applications, 8(3):29–37, May, 1988.

[2]. Nelson Max, "Optical Models for Direct Volume Rendering", in Visualization in Scientific Computing, Springer, pp. 35-40, 1995.

[3]. Tom McReynolds and David Blythe. “Advanced Graphics Programming using OpenGL”. Morgan Kauffman Publishers, first edition, 2005.

[4]. Kenneth Moreland, "Fast High Accuracy Volume Rendering", University of New Mexico and Sandia National Laboratories, Tesis doctoral 2004.

[5]. Peter Shirley. **“**Fundamentals of Computer Graphics**”**, 2da. edición. Diciembre, 2002.

[6]. Ciro José Durán y Francisco Rafael Morillo. “Rendering Volumétrico Acelerado basado en Mecanismos de Manejo de Texturas” UNIVERSIDAD SIMÓN BOLÍVAR, Tesis de grado 2006.

[7]. Bernhard Preim y Dirk Bartz. **“**Visualization in Medicine Theory, Algorithms and Applications**”**. Morgan Kaufmann Publishers. 2007.

[8]. Jonathan Richard Shewchuck, "Delaunay Refinement Mesh Generation", Carnegie Mellon University, Tesis de doctorado 1997.

[9]. Nelson Max, Peter Williams, Claudio Silva, and Richard Cook, "Volume Rendering for Curvilinear and Unstructured Grids", in Proceedings of ACM SIGGRAPH 2000, pp. 53-61, 2000.

[10]. Sheng Guo, Cage Lu, and Xiaochang Wu, "Dynamic Volumetric Cloud Rendering for Games on MultiCore Platforms", Software and Services Group, Intel Corporation, 2010.

[11]. W. Kruger. “The application of transport theory to visualization of 3D scalar data fields”. In Computers in Physics, volume 5, pp. 397–406, 1990.

[12]. Peter Williams and Nelson Max, "A volume density optical model", in Proc. Workshop on Volume Visualization 92, Computer Graphics, pp. 61–68, 1992.

[13]. Rhadamés Carmona, "Visualización Multi-Resolución de Volúmenes de Gran Tamaño", Universidad Central de Venezuela, Caracas, Tesis Doctoral 2010.

[14]. Paolo Sabella, "A rendering algorithm for visualizing 3D scalar fields", in Proc. ACM SIGGRAPH 88, Computer Graphics, pp. 51–58, 1988.

[15]. Lorensen William y Cline Harvey. “Marching Cubes: A high resolution 3D surface construction algorithm”. Computer Graphics, Vol. 21- No. 4, Julio 1987.

[16]. R. A. Drebin, L Carpenter, and P Hanrahan, "Volume Rendering", in ACM Computer Graphics (SIGGRAPH 88 Proceedings), pp. 65-74, 1988.

[17]. Joe M Kniss, "Interactive Volume Rendering Techniques", Departamento de Ciencias de la Computación, Universidad de Utah, Tesis de Maestría 2002.

[18]. Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, “OpenGL® Programming Guide: The Official Guide to Learning OpenGL”, Version 1.2, 3rd ed.: Addison Wesley Professional, 1999.

[19]. J. WILHELMS and A. VAN GELDER. “Octrees for faster isosurface generation”. ACM Transactions on Graphics, 11(3):201–227, 1992.

[20]. Marroquim, R., Maximo, A., Farias, R., and Esperanca, C. “Gpu-based cell projection for interactive volume rendering”. SIBGRAPI 2006 paper, Manaus, AM, Brazil, 0:147–154, 2006.

[21]. Shirley, P. and Tuchman, A. A. “Polygonal approximation to direct scalar volume rendering”. In Proceedings San Diego Workshop on Volume Visualization, Computer Graphics, volume 24(5), pp. 63–70, 1990.

[22]. G. Frieder, D. Gordon, and A. Reynolds. “Back-to-front display of voxel-based objects”. In IEEE Computer Graphics and Applications, pp. 52–60. IEEE Press, 1985.

[23]. Wylie, B., Moreland, K., Fisk, L. A., and Crossno, P. “Tetrahedral projection using vertex shaders”. In VVS ’02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics, pp. 7–12, Piscataway, NJ, USA. IEEE Press, 2002.

[24]. Marroquim, R., Maximo, A., Farias, R., and Esperanca, C. “GPU-Based Cell Projection for Large Structured Data Sets”. SIBGRAPI 2007 Workshop Paper, Belo Horizonte, MG, Brazil, 2007.

[25]. Moreland, K. and Angel, E. “A fast high accuracy volume renderer for unstructured data”. In VVS ’04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics, pp.13–22, Piscataway, NJ, USA. IEEE Press, 2004.

[26]. Roettger, S. Kraus, M. Ertl, T. “Hardware-Accelerated Volume And Isosurface Rendering Based On Cell-Projection”. In VIS ’00: Proceedings Conf. on Visualization, pp109-116. IEEE Computer Society. Los Alamitos 2000.

[27]. Williams, P. L.. “Visibility-ordering meshed polyhedral”. ACMTrans. Graph., 11(2):103–126, 1992.

[28]. P. L. Williams and N. Max. “A Volume Density Optical Model”. In ACM Computer Graphics (1992 Workshop on Volume Visualization), pp.61-68, 1992.

[29]. N. MAX. “Optical models for direct volume rendering”. IEEE Transactions on Visualization and Computer Graphics, 1(2):99–108, 1995.

[30]. S. P. Callahan and J. L. D. Comba. “Hardware-assisted visibility sorting for unstructured volume rendering”. IEEE Transactions on Visualization and Computer Graphics, 11(3):285–295, Student Member-Milan Ikits and Member-Claudio T. Silva, 2005.

[31]. Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., and Strasser, W. “Smart hardware-accelerated volume rendering”. In *VISSYM ’03: Proceedings of the symposium on Data visualisation 2003*, pp 231–238, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association, 2003.

[32]. Jorge Luis Bernadas, “Tetraedrización de Intervalos de Volumen Mediante Modificación De Cubos Marchantes”. Proyecto de Grado, Biblioteca Alonso Gamero, Universidad Central de Venezuela, 2009.

[33]. T. T. Elvins, “A Survey of Algorithms for Volume Visualization”, Computer Graphics, vol. 26, num. 3, pp. 194-201, 1992.

[34]. B.P. Carneiro, C. Silva y A.E. Kaufman, “Tetra-Cubes: An algorithm to generate 3D isosurfaces based upon tetrahedra”, Anais do IX SIBGRAPI, pp. 205-210, 1995.

[35]. Hakan Berk and Cevdet Aykanat and Ugur Gudukbay, “Direct Volume Rendering of Unstructured Grids”, Computer & Graphics, vol. 27, pp. 387-406, 2003.

[36]. Wilhelms J. and Van Gelder A., “A Coherent Projection Approach for Direct Volume Rendering”. Computer Graphics, 25(4): 275–284, 1991.

[37]. *Kraus, M., Qiao, W., and Ebert, D. S.,* "Projecting Tetrahedra without Rendering Artifacts,*" Proceedings IEEE Visualization 2004, pp. 27-34, 2004.*

[38]. El proyecto OpenQVis http://openqvis.sourceforge.net/, http://openqvis.sourceforge.net/docu/fileformat.html

[39]. https://msdn.microsoft.com/en-us /library/ windows/ desktop/ dn742498(v=vs.85).aspx.

[40]. Graphics Gems III, *David Kirk (editor)*, Academic Press, ISBN: 0124096735, 1992.

[41]. OpenGL Extension Wrangler Library (GLEW), (http://glew. sourceforge.net/), 08-10-2015.