Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

VISUALIZACION DE ECUACIONES IMPLICITAS UTILIZANDO WEBGL

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central de Venezuela
Por el Bachiller
Juan Andrés González Trejo
para optar al título de

Licenciado en Computación

Tutor: Prof. Rhadamés Carmona

<Caracas, 27/10/2015>

Dedicatoria

A mi padre, a mi madre, a mi hermana, a mis amigos, quienes me han brindado todo para que pudiera alcanzar esta meta y culminar una etapa más en la vida. Sin su apoyo y dedicación no me hubiese sido posible llegar al punto en el que me encuentro hoy en día.

Agradecimientos

A mi hermana quien ha sido un apoyo constante a lo largo de mi vida y un símbolo de que querer es poder.

A mi madre y padre por ser constantes y darme un empujón de seguir adelante.

A mis amigos, los cuales me han apoyado en todo momento y siempre han estado ahí cuando ha sido necesario.

A mi tutor, Prof. Rhadamés Carmona, por asistirme mi formación como profesional y durante la elaboración de este trabajo.

Resumen

Las Ecuaciones implícitas son ampliamente utilizadas en el área de la matemática y la física, por lo que en el mundo científico han formado parte de nuestra vida cotidiana, y han ayudado a estudiar distintos fenómenos. Típicamente, para visualizar estas ecuaciones en tres dimensiones se utilizan algoritmos que reconstruyen la superficie 3D buscando los ceros de la función en una malla regular, y conectando los puntos resultantes para generar dicha superficie. Entre estos algoritmos se encuentran Marching Cubes y Tetra Cubes. Con la popularidad que ha adquirido la visualización de volúmenes en los últimos años, hoy en día es posible visualizar la ecuación implícita sin la necesidad de realizar la reconstrucción tridimensional. En este caso, basta evaluar la ecuación implícita en una malla regular 3D que forma un volumen, y configurar la función de transferencia para visualizar únicamente la superficie deseada (los ceros de la función). Basados en esta idea, podemos visualizar fácilmente cualquier otra capa de la función implícita; es decir, visualizar otra superficie que no coincida con los ceros de la función, sino con cualquier otro valor. Por ejemplo, con la ecuación de una esfera $x^2+y^2+z^2-r^2=u$ es posible visualizar las superficies generadas para distintos valores de u sin necesidad de reconstruir cada una de estas superficies, lo cual genera costos importantes de tiempo y espacio. En los últimos años han surgido tecnologías que permiten la utilización del hardware gráfico en el despliegue tridimensional en navegadores Web; entre estas tecnologías se encuentra WebGL. En este trabajo se elaboró un prototipo de aplicación Web basado en WebGL que implementa el algoritmo de visualización de volúmenes llamado ray casting para el despliegue de ecuaciones implícitas.

Palabras claves: ecuaciones implícitas, ray casting, Web Open Graphics Library (WebGL), visualización de volúmenes.

Índice

Dedicatoria 3
Agradecimientos 4
Resumen 5
Índice 6
Introducción 8
Capítulo 1 - Descripción del problema 10
1.1 - Planteamiento del problema 10
1.2 Objetivo General 11
1.3 Objetivos Específicos 11
1.4 Solución propuesta 11
1.4.1. Metodología de desarrollo 11
1.4.3. Módulos a desarrollar 12
- Análisis sintáctico de la ecuación implícita 12
- Rendering de volúmenes: 12
- Interfaz gráfica 12
1.4.4. Pruebas a realizar 13
Capítulo 2 – Marco teórico 14
2.1 Imagen 14
2.2 Pixel 14
2.3 Voxel 14
2.4 Polígono 15
2.5 Rendering 15
2.6 Volume Rendering (Rendering de Volúmenes) 18
2.6.1 Rendering Directo de Volúmenes 19
2.6.2 Clasificación 20
2.7 - Ecuaciones Implícitas 21
2.8 Ceros de funciones 22
2.8.1 Método de Bisección 22
2.8.2 Método de Newton 23
2.8.3 Método de la Secante 24
2.8.4 Método de Posición Falsa o Regula Falsi 25
2.8.5 Método de Müller 26

2.9 Evaluadores sintácticos 26
2.9.1 Análisis Léxico 27
2.9.2 Análisis Sintáctico 27
2.9.3 Top-Down 28
2.9.4 Botton-Up 29
2.10 Trabajos anteriores 30
Capítulo 3 – Diseño e Implementación 36
3.1 Funcionamiento general del sistema 36
3.2 Implementación 37
Contenido externo utilizado 37
Capítulo 4 Pruebas y Resultados 45
4.1 Ambiente de trabajo 45
4.2 Funciones de prueba 45
4.3 Pruebas de calidad y rendimiento 45
Capítulo 5 – Conclusiones y Trabajos Futuros 49
Referencias 50

Introducción

Las Ecuaciones implícitas son ampliamente utilizadas en el área de la matemática y la física, por lo que en el mundo científico han formado parte de nuestra vida cotidiana, y han ayudado a estudiar distintos fenómenos (Steffen, 2003). Típicamente, para visualizar estas ecuaciones en tres dimensiones se utilizan algoritmos que reconstruyen la superficie 3D buscando los ceros de la función en una malla regular, y conectando los puntos resultantes para generar dicha superficie. Entre estos algoritmos se encuentran Marching Cubes y Tetra Cubes ((Lorensen & Cline, 1987) y (Hansen & Johnson, 2004)). La idea es primero evaluar las función implícita en los puntos (x,y,z) de una malla regular uniforme, y luego seleccionar el umbral a reconstruir (típicamente el valor cero). El resultado es un mallado de triángulos que puede ser visualizado en cualquier computador que tenga soporte para visualización 3D.

Con la popularidad que ha adquirido la visualización de volúmenes en los últimos años, hoy en día es posible visualizar la ecuación implícita sin la necesidad de realizar la reconstrucción tridimensional. En este caso, basta evaluar la ecuación implícita en una malla regular 3D que forma un volumen, y configurar la función de transferencia para visualizar únicamente la superficie deseada (los ceros de la función). Basados en esta idea, podemos visualizar fácilmente cualquier otra capa de la función implícita; es decir, visualizar otra superficie que no coincida con los ceros de la función, sino con cualquier otro valor. Por ejemplo, con la ecuación de una esfera $x^2+y^2+z^2-r^2=u$ es posible visualizar las superficies generadas para distintos valores de u sin necesidad de reconstruir cada una de estas superficies, lo cual genera costos importantes de tiempo y espacio.

En los últimos años han surgido tecnologías que permiten la utilización del hardware gráfico en el despliegue tridimensional en navegadores Web; entre estas tecnologías se encuentra WebGL (acrónimo de Web Graphics Library). En este trabajo se elaboró un prototipo de aplicación Web basado en WebGL que implementa el algoritmo de visualización de volúmenes llamado ray casting para el despliegue de ecuaciones implícitas. Esto le permite al usuario escribir la ecuación implícita, definir el área a visualizar (subdominio de la ecuación), y el conjunto de umbrales con sus respectivos niveles de transparencia.

En este documento se describen las técnicas y decisiones de diseño tomadas para poder crear un prototipo de aplicación Web para generar el volumen a partir de la ecuación implícita, y desplegarlo con ray casting. EL volumen es almacenado en un atlas de textura 2D, pues la textura 3D no era soportada por WebGL al momento de iniciar la implementación.

El documento está dividido en capítulos. En el capítulo 1 se describe el problema a resolver, sus objetivos y alcances. El capítulo 2 comprende el marco teórico que soporta esta investigación. En el capítulo 3 se describe el diseño y la implementación del sistema, incluyendo las funciones necesarias

para la correcta actualización de la función de transferencia y re dibujo de la ecuación implícita. Se implementa una interfaz gráfica para controlar cada aspecto del despliegue. Haciendo uso de esta interfaz, en el capítulo 4 se discute las pruebas realizadas para demostrar el funcionamiento del sistema como un todo y la viabilidad de la utilización WebGL en este campo. Finalmente, se tienen las conclusiones y trabajos a futuro de esta investigación en el capítulo 5.

Capítulo 1 - Descripción del problema

Las ecuaciones implícitas están involucradas en diversos procesos e investigaciones. Estas nos ayudan a representar comportamientos físicos, entre otros fenómenos (Schulz, 2003). Comúnmente estas funciones son bidimensionales, y visualizadas en un plano 2D; esto ayuda tener una mejor idea de cómo determinar los límites, puntos interés y características propias de la función.

Al pasar de un plano de 2d al espacio 3d aumenta la dificultad de ver la representación gráfica. Se han diseñado distintas tecnologías para facilitar el trabajo a la hora de visualizar dichas ecuaciones, hasta llegar al punto de tener aplicaciones por computador donde se escribe la función implícita, y esta es desplegada en 3D.

El rendering o despliegue de volúmenes muestra una nueva forma de ver el conjunto de datos estableciendo dos vías para su despliegue por pantalla. El primer método, rendering directo de volúmenes, consiste en la asignación de un color y una opacidad a cada muestra (voxel), por medio de una función de transferencia. El volumen como tal es representado por medio de cortes igualmente espaciados de la ecuación implícita. Luego, mediante trazado rayos (*ray casting*), el volumen es proyectado en el plano imagen para su visualización. El segundo método, el rendering indirecto de volúmenes, requiere de la generación de una de iso-superficie intermedia, típicamente reconstruida por algoritmos como *Marching Cubes* y *Tetra Cubes* (Lorensen & Cline, 1987) y (Hansen & Johnson, 2004) .Para poder observar las distintas capas de estos volúmenes, se utiliza un umbral por cada capa, al cual se le asigna un color y una opacidad a través de una función de transferencia. Este tipo de rendering es usado para estudiar datos médicos, piezas automotrices y funciones implícitas.

Se han utilizado métodos indirectos como *Marching Cubes* para visualizar Ecuaciones Implícitas. Visualizar decenas de capas con este método es poco práctico, puesto que requiere de espacio para almacenar todos los mallados reconstruidos, así como tiempo para su generación. Es así como el método de rendering directo de volúmenes es atractivo, pues podemos visualizar gran cantidad de capas del volumen, sin detrimento del tiempo de respuesta ni de requerimientos extras de memoria, más que el volumen en sí.

1.1 - Planteamiento del problema

Se propone desarrollar un sistema prototipo de visualizador de ecuaciones implícitas aplicando la técnica de *rendering* directo de volúmenes, que permita visualizar varias valores de superficie asociadas a la ecuación implícita. Por ejemplo, no solo se desea visualizar los ceros de la función implícita, sino cualquier otro iso-valor que nos genere información de interés, de manera similar a como se visualizan los distintos tejidos en una resonancia magnética o tomografía computarizada. Adicionalmente se desea que este sistema esté disponible en la web, y no requiera de instalación.

1.2 Objetivo General

Desarrollar un sistema de visualización de ecuaciones implícitas en la Web aplicando la técnica de *rendering* de volúmenes.

1.3 Objetivos Específicos

- Crear una interfaz Web que permita al usuario introducir la ecuación implícita, y seleccionar las valores de superficie que desea visualizar.
- Crear un módulo de análisis sintáctico de la ecuación.
- Crear un módulo para evaluar la función implícita, de manera de poder generar un volumen para su posterior visualización.
- Crear un módulo de visualización utilizando la técnica de *ray casting*.
- Evaluar el rendimiento de la aplicación.

1.4 Solución propuesta

A continuación, se presentan algunos detalles de la propuesta de solución, incluyendo metodología de desarrollo a emplease, la plataforma de desarrollo, módulos a desarrollar, y pruebas a realizar.

1.4.1. Metodología de desarrollo

Le metodología de desarrollo a utilizar es AdHoc. En esta se planificarán una serie de reuniones en donde se mostrará el desarrollo del sistema, conforme se avance en el proyecto, y se realizarán los cambios y sugerencias hechas por el tutor.

Se utilizará la versión de sublime text (Skinner, 2015) como ambiente de desarrollo, usando como API librerías graficas OpenGL ES 2.0 y librería de shaders GLSL 1 para esta aplicación, los cuales están incluidas dentro de WebGL. Asimismo, nos apoyaremos en plantillas de Bootstrap para el diseño de interfaces, todo bajo el lenguaje de javascript/HTML/css3 para implementar el sistema.
1.4.2. Plataforma de desarrollo y pruebas

Plataforma de hardware

CPU Intel Core i7Q740 1.73GHz

RAM 8GB

NVIDIA geforce 310m (mobile graphics) 512mb ddr3

Plataforma de software

OpenGL ES 2.0.

Windows 8.1

Javascript/HTML5/ccs3

1.4.3. Módulos a desarrollar

Para solucionar el problema planteado, se desean desarrollar los siguientes módulos:

- Análisis sintáctico de la ecuación implícita

Se debe crear un módulo que verifique si la cadena de caracteres introducida es una función valida. El analizador a usar es un *framework* de la librería de *Boost* c++ (Guzman, 2010), llamado *Spirit*. Este analizador sintáctico posee gran flexibilidad que brindan las plantillas y la sobrecarga de operadores de c++. Esto hace que sea sencillo de construir, además de poseer una adecuada documentación. Se utilizará regex de boost para comprobar que la cadena introducida es cumple con las reglas léxicas correctas, y se definirá una gramática que evaluará la expresión dada.

- Rendering de volúmenes:

Se utilizará *ray casting* basado en GPU. Según el trabajo de (Klaus Engel, 2001), se utilizará la post-clasificación y la clasificación pre-integrada. La razón de utilizar la clasificación pre-integrada es que permite capturar los detalles finos de la función de transferencia, que con la post-clasificación requeriría aumentar significativamente la taza de muestreo del volumen, acarreando una reducción en el tiempo de respuesta.

- Interfaz gráfica

La interfaz se diseñará con el fin de cumplir con los requisitos mínimos para la visualización de la ecuación implícita. El usuario debe poder introducir los valores de superficie a visualizar, y tener la posibilidad de asignarle color, transparencia y demás propiedades de rendering a cada iso-valor. El usuario debe poder especificar el dominio en el cuál se evaluará la función implícita mediante una caja alineada a los ejes (i.e. x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}). Debido a que en principio el usuario podría no conocer el rango función (el conjunto de valores que puede tomar [f_{min},f_{max}]), la aplicación podría guiar al usuario. Para ello, la función implícita puede ser primero evaluada en el dominio establecido para conocer el rango [*fmin*, *fmax*] de la función, y una vez conocido dicho rango, solicitarle al usuario dónde estarían ubicados los valores de superficie que desea visualizar, siempre dentro de dicho rango de valores. En esta dirección, se puede crear un control especial que permita al usuario introducir los valores de superficie y cambiarlos en forma sencilla, como se muestra en la Fig. 1.1. En la imagen a continuación podemos observar un prototipo de la interfaz gráfica a realizar. El primer elemento contiene los campos para establecer el sub volumen de la ecuación a visualizar, dados por los puntos extremos min(x,y,z) y max(x,y,z) de una caja alineada a los ejes. En este dialogo se introducen valores flotantes (sean positivos o negativos) de cada uno de los limites (x,y,z) de la ecuación. Se agrega un campo para definir el número de subdivisiones el cada eje. Por ejemplo, si colocamos 32 subdivisiones significa que el volumen será discretizado en 32^3 muestras.

En el siguiente campo se insertará la ecuación de la forma descrita, en el que los símbolos representarán las operaciones asociadas, como suma (+), resta (-), multiplicación (*), división (/), potenciación (^) y agrupación de operadores(+,-,*,^,) y operandos (X,Y,Z) con paréntesis.

En el tercer campo el usuario podrá seleccionar los diferentes iso valores, y moverlos libremente en el rango de valores [*fmin*, *fmax*] determinado por el sistema. El iso valor y su color asociado serán mostrados con un simple objeto de color (triángulo o cuadrado), y el valor flotante asociado podrá visualizarse en otro control. A su vez se mostrará una ventana informativa donde se pueden editar los datos de la función de transferencia para cada iso-valor, como el color (R,G,B) y absorción (A).

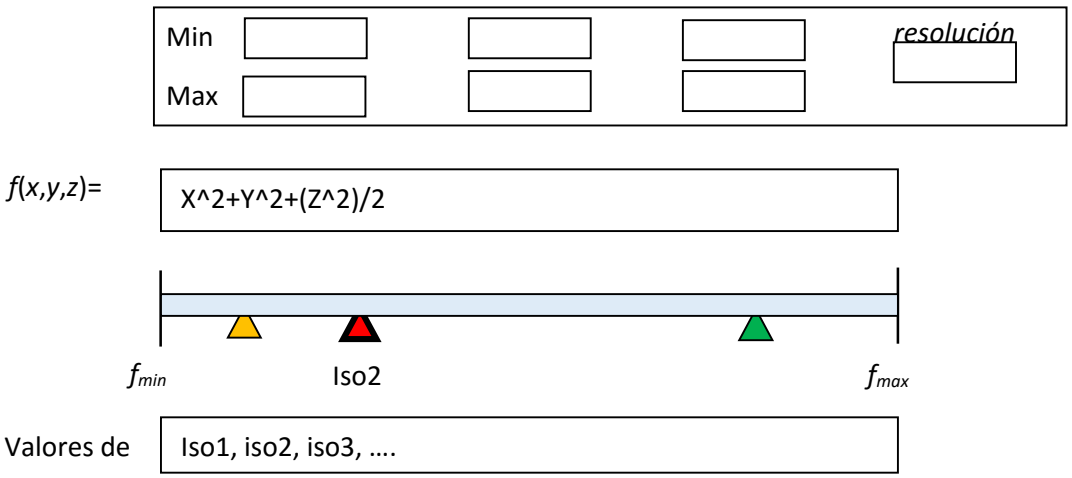


Fig. 1.1: prototipo de interfaz para seleccionar valores de superficie

El modelo de iluminación a utilizar es Phong Shading (Foley, van Dam, Feiner, & Hughes, 1996), el cual se puede habilitar o deshabilitar. Los colores difusos y especular de una iso-superficie están definidos por el mismo color RGB de la función de transferencia para dicha iso-superficie.

1.4.4. Pruebas a realizar

Las diferentes pruebas a realizar buscan medir el tiempo de reconstrucción y el tiempo de rendering para diferentes ecuaciones implícitas, variando la cantidad de valores de superficie, y considerando distintas resoluciones del volumen.

Capítulo 2 – Marco teórico

En este capítulo se introducen los conceptos básicos de imagen, pixel, voxel, entre otros, necesarios para describir brevemente el proceso de rendering, y particularmente el rendering de volúmenes, del cual se describen el método a implementar (ray casting).

2.1 Imagen

La palabra imagen proviene del latín imago. La imagen es un artefacto que representa o registra la percepción visual, por ejemplo, una foto en dos dimensiones. Algunas de las características de la imagen son:

- Pueden ser de dos dimensiones, como una fotografía, o las que aparecen en un monitor. También pueden ser de tres dimensiones, como una estatua o un holograma. Estos últimos son capturados por dispositivos ópticos, como cámaras, espejos, lentes, telescopios, microscópicos, etc. Otros objetos naturales y fenómenos, como el ojo humano o superficies de agua también los capturan.

- La palabra imagen es usada para figuras bidimensionales como un mapa, un grafo, un gráfico de torta o una pintura. En un amplio espectro, las imágenes pueden ser generadas manualmente (como es el caso de los dibujos y una pieza de arte), o mediante un computador, en donde se pueden generar imágenes sintéticas (generadas con técnicas de computación gráfica), se pueden obtener mediante digitalización de imágenes existentes, o simplemente se pueden reproducir o imprimir.

2.2 Pixel

En una imagen digital, un pixel, es simplemente un punto de la imagen, el cual puede ser direccionado mediante un par de coordenadas enteras. En un dispositivo de despliegue, es el elemento más pequeño controlable por el hardware del dispositivo. Cada pixel es una muestra de una imagen real; lógicamente más muestras proveen una mejor representación de la imagen original. La intensidad de cada pixel es variable. Su color es típicamente representado por tres o cuatro componentes de intensidad ya sean rojo, verde y azul, o los componentes cyan, magenta, amarillo y negro en el caso de las impresoras. La palabra pixel proviene de la contracción de "picture" (pix) y "element" (el). Encontramos construcciones similares para las palabras voxel (volume element) y texel (texture element).

2.3 Voxel

Así como el pixel es la unidad mínima discreta direccionable en una imagen digital, el voxel es similarmente una unidad discreta que representa un punto de un volumen. En un voxel pueden almacenarse distintas propiedades, como una densidad, una opacidad, un color, un gradiente, etc. El

valor de un voxel puede representar diversas propiedades. En cada aparato de captura (por ejemplo, CT, MRI y ultrasonido) tiene una interpretación distinta.

Para volúmenes representados por una malla regular, los voxels no tienen por lo general su posición (sus coordenadas) codificada explícitamente junto con sus valores. La posición de un voxel se infiere basándose en su posición relativa con respecto a otros voxels, es decir, su posición en la estructura de datos que define el volumen discretizado. Los voxels se utilizan con frecuencia en la visualización y el análisis de datos médicos y científicos.

Por lo general, se hace abstracción de un voxel como un punto de la malla regular, y no un sub volumen del volumen original. Dependiendo del tipo de datos y el uso previsto para el conjunto de datos, el espacio que hay entre voxels puede ser reconstruido y/o aproximado, por ejemplo, mediante interpolación.

Entre los usos más frecuentes de los voxels está la medicina para representar data volumétrica de pacientes, la representación de datos geológicos para explotación petrolera, la representación de la tierra en los juegos y la simulación. Los terrenos de voxels suelen ser utilizados en vez de un mapa de altura gracias a que se pueden representar las salientes, cuevas, arcos, y otras características del terreno en 3d como es el caso de C4 engine [LLC, 2001]. Estas características cóncavas no pueden ser representadas en un mapa de altura dado que solo la capa superior del terreno (una altura por cada punto) puede ser representada.

2.4 Polígono

En geometría un polígono es una figura plana que está dada por una cadena finita de segmentos de rectas, formando un ciclo o un circuito. Estos segmentos son llamados bordes o lados; los puntos donde dos bordes se encuentran son los vértices o esquinas. El interior de un polígono es llamado cuerpo. Un n-ágono es un polígono de n lados. La palabra polígono proviene del griego *polús* "muchos", y *gōnía* "esquina" o "ángulo".

Los polígonos son usados en computación gráfica para componer imágenes o modelos por lo general en tres dimensiones. No siempre son triangulares. Estos surgen cuando se modela la superficie del objeto.

En contraste a los pixeles y voxels, los polígonos se representan a menudo explícitamente por las coordenadas de sus vértices. Una consecuencia directa de esta diferencia es que los polígonos son capaces de representar de manera eficaz las estructuras 3D simples con espacios homogéneos, mientras que los voxels son adecuados para la representación de espacios no homogéneos.

2.5 Rendering

Rendering es el proceso de transformación de una escena en una imagen [STEVENS, 1993]. En el rendering, se suele emplear un modelo de iluminación para generar una simulación realista del comportamiento de luces, texturas y materiales (agua, madera, metal, plástico, tela, etcétera). También se pueden considerar los comportamientos físicos como en el caso de las colisiones y fluidos (ver Fig. 2.1).

Figura 2.1: Ejemplo del resultado final del proceso de rendering, donde se recrea una escena con distintos materiales, texturas e iluminación

Como se puede apreciar en la Fig. 2.1, una escena contiene objetos, los cuales pueden estar definidos por un lenguaje o estructura de datos. Dicha estructura contiene la geometría, el punto de vista, las texturas, la iluminación e información del sombreado (shading), es decir, la descripción de la escena. Los datos contenidos en la escena son transferidos a un programa de rendering, cuyo resultado final es una imagen digital o gráfico raster. Durante el rendering 3D se suelen utilizar distintos sistemas de coordenadas, como coordenadas de mundo, de modelo, de vista, de recorte o clipping, etc. Las transformaciones que llevan un sistema de coordenadas a otro suelen representarse por una matriz.

Varios de los usos del rendering están ligados a la arquitectura, video juegos, simulación, películas o efectos visuales de televisión y diseño asistido por computador (Computer-Aided Design; CAD). Cada uno emplea diferentes balances de elementos y técnicas. Como producto, hay una gran variedad de motores de rendering disponibles; algunos de estos están integrados en grandes paquetes de modelaje y animación, otros son de uso dedicado, como proyectos de código libre, juegos, etc.

Para visualizar la escena en tiempo real, se suele utilizar un pipeline gráfico, el cual convierte primitivas gráficas del espacio de modelo en primitivas dentro del espacio del dispositivo, es decir, en píxeles. Partes de este pipeline gráfico son implementadas en el hardware gráfico para tener mejores tiempos de respuestas. Con el avance de la tecnología, algunas etapas de este pipeline gráfico tienen la facilidad de ser programables. Así, algunos de sus estados son de hecho unidades de programación; esto quiere decir que podemos implementar nuestros modelos de luz o transformaciones geométricas, teniendo libertad en la transformación de cada vértice y de la

asignación de color a cada fragmento. Los aceleradores gráficos programables, son también conocidos como GPU (*Graphics Processor Unit*, o unidad de procesamiento gráfico).

La Unidad de Procesamiento Gráfico (Graphic Processing Unit; GPU) es un dispositivo diseñado con el propósito de acelerar la creación de imágenes en el frame buffer (memoria dedicada de video que contiene los datos completos de cada cuadro de imagen). Las GPUs se encuentran embebidas en sistemas, teléfonos móviles, computadores personales, estaciones de trabajo y consolas de videojuegos. La GPU moderna es eficiente manipulando los gráficos del computador, y gracias a su estructura de alto paralelismo es más eficiente que el CPU.

La GPU se encarga de hacer el cálculo del modelo de iluminación, rasterización, texturización y clipping (recortes); este último consiste la eliminación de porciones de objetos extendidos más allá de una región predeterminada de la escena (e.g. un clippling plane), quedando así un subconjunto de los objetos a ser desplegados (STEVENS, 1993).

El proceso de rendering se puede describir mediante la Ecuación 1.1. Esta ecuación no abarca todo el fenómeno luminoso, pero es parte del modelo de iluminación para generar imágenes por medio de la computadora.

$$L_o(x,\vec{w}) = L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w}) L_i(x,\vec{w}')(\vec{w}' \cdot \vec{n}) \mathrm{d}\vec{w}'$$

Ecuación 1.1 (Ecuación de rendering o cálculo del modelo de iluminación)

La ecuación 1.1 describe como en una posición *x* y dirección *w* particular, la luz de salida(L_o) es la suma de la luz emitida (L_e) y el reflejo de la luz. El reflejo de la luz por otra parte se modela como una integral donde intervienen la luz de entrada(L_i) de todas las direcciones, multiplicada por el reflejo de la superficie $f_r(x,\vec{w},\vec{w})$, y el ángulo de entrada de la luz $(\vec{w} \cdot \vec{n})$. Esto define el transporte de la luz a través de la escena. La función de distribución bidireccional de reflectancia (BRDF) $f_r(x,\vec{w},\vec{w})$, expresa un modelo de interacción de luz con las superficies, en cual se destacan 2 tipos: la reflexión difusa y la reflexión especular.

2.5.1 Características del Rendering

El motor de rendering posee una abundante cantidad de características visuales para dar un resultado realista en lo posible. El desarrollo y la investigación en el rendering se han enfocado en encontrar formas de simular estas características visuales de manera eficiente. Algunas de estas características están directamente relacionadas con algoritmos y técnicas, mientras que otras características son producto de la unión de varias técnicas. Algunas de estas técnicas son:

- Shading (sombreado): define cómo el color y el brillo de una superficie varían según la iluminación
- Texture-mapping (aplicación de textura): es un método para aplicar detalle de textura sobre superficies

- Bump-mapping: es un método para simular a escala pequeña protuberancias o hendiduras sobre superficies
- Neblina: define cómo la luz obscurece cuando pasa por medio de una atmósfera densa
- Sombras: muestra el efecto de obstrucción de la luz
- Sombras suaves: variación de oscuridad causada por fuentes de luz parcialmente ocultas
- Reflexión: efecto espejo, un rayo de luz incide sobre una superficie y es reflejado
- Transparencia: transmisión continua de la luz a través de objetos sólidos
- Refracción: curvatura de la luz asociada con la transparencia
- Difracción: desviación de la luz al encontrar un obstáculo o al atravesar una rendija
- Iluminación indirecta: superficies iluminadas por la luz reflejada de otras superficies, en lugar de ser iluminada por una fuente de luz directa
- Cáustica: es la combinación de los rayos de luz reflejada o refractada por una superficie u objeto curvo, o la proyección de esa combinación de rayos en otra superficie
- Depth of field (profundidad de campo): efecto que hace que objetos aparezcan borrosos o fuera de foco cuando se está muy lejos, en frente o detrás de un objeto enfocado
- Motion blur (borrosidad por movimiento): los objetos aparecen borrosos dado el movimiento de alta velocidad o movimiento de la cámara
- Rendering no foto realístico: rendering de escenas en un estilo artístico, destinado a verse como una pintura o un dibujo

2.6 Volume Rendering (Rendering de Volúmenes)

En visualización científica y computación gráfica, es una técnica utilizada para generar una imagen a partir de la proyección de un volumen, simulando cómo se propaga la luz a través de dicho volumen (B & D, 2007). Típicamente este volumen viene dado por un conjunto de imágenes seriadas 2D, igualmente espaciadas, adquiridas por CT, MRI, entre otros (ver Fig. 2.2). La distancia entre muestras (voxels) de un mismo corte también es constante, generando un patrón regular llamado mallado volumétrico regular.

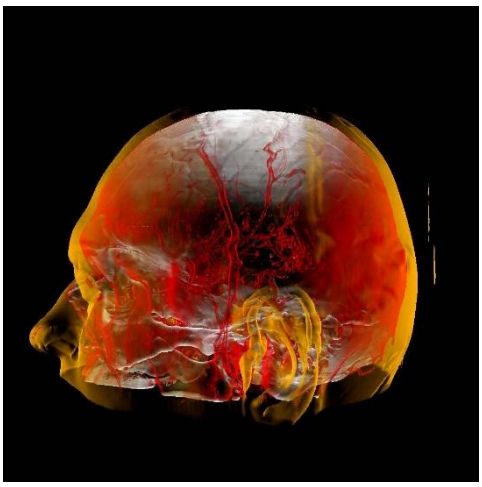


Figura 2.2: rendering de un volumen capturado por un escáner CT

Hay dos vertientes para visualizar el volumen: el despliegue indirecto de volúmenes, el cual requiere de la generación de una superficie antes de su visualización, y el despliegue directo de volúmenes, el cual visualiza el volumen partir de la proyección de sus muestras. En el caso de despliegue indirecto de volúmenes, vale mencionar el algoritmo de *Marching Cubes* (B & D, 2007), que es una técnica común utilizada para la extracción de valores de superficie, en el cual toma 8 muestras adyacentes a la vez (formando un cubo imaginario), en donde se determinan los polígonos que atraviesan ese cubo mediante interpolación lineal. Para el caso de despliegue directo de volúmenes, estudiaremos la técnica de ray casting, la cual será implementada en este trabajo.

2.6.1 Rendering Directo de Volúmenes

Para hacer una proyección 2D del conjunto de voxels 3D, primero hay que definir una cámara en el espacio en relación con el volumen. También, hay que definir la opacidad y el color de cada voxel [Levoy, 1988]. Esto se define por lo general con una función de transferencia unidimensional [Infogrames, 2003]. Con dicha función de transferencia se define el valor RGBA (Red, Green, Blue, Alpha; las siglas para cada color y trasparencia) para cada valor posible de voxel. Esta función suele ser una función lineal a trozos, por cada componente de la tupla RGBA. El proceso de asignar un color y una opacidad a un voxel se llama clasificación.

El pipeline del rendering de volúmenes sigue una serie de pasos que constan del muestreo (sampling), clasificación y composición.

El muestreo consiste en reconstruir muestras del volumen por medio de la interpolación de voxels. Esto se realiza debido a que, al proyectar el volumen, las muestras originales del mismo no coinciden con la ubicación de los pixeles, por lo que en cambio se reconstruyen los voxels que están a lo largo de un rayo que pasa por el pixel desde la posición del observador.

La clasificación e iluminación consiste en darle color y opacidad al voxel reconstruido. Mediante una función de transferencia se le otorga un color y opacidad inicial al voxel, el cual es usado en conjunto con un modelo de iluminación para calcular su contribución lumínica.

Luego de muestrear, clasificar e iluminar cada voxel al largo de la discretización de un rayo, estos deben ser compuestos de acuerdo al modelo físico del volumen.

El orden en que se realiza el proceso de clasificación y muestreo determina el tipo de clasificación. Se realiza post-clasificación cuando se reconstruye el voxel y luego es clasificado, mientras que se realiza pre-clasificación cuando primero se clasifican las muestras del volumen original, y luego se realiza la reconstrucción de las muestras (B & D, 2007).

La técnica de ray casting se deduce directamente de la ecuación de rendering [Kajiya, 1986]. Esta provee resultados de alta calidad. En esta técnica se genera un rayo por cada pixel deseado de la imagen. Usando un modelo simple de cámara, el rayo comienza desde el centro de proyección de la cámara (normalmente la ubicación del ojo) y pasa a través de un pixel en el plano imagen imaginario ubicado entre la cámara y el volumen. En la intersección de rayo con el volumen, el volumen es muestreado en intervalos regulares o adaptativos para, a través de la composición,

obtener el valor de un pixel en la imagen. El proceso se realiza por cada pixel en la imagen (ver Fig. 2.3).

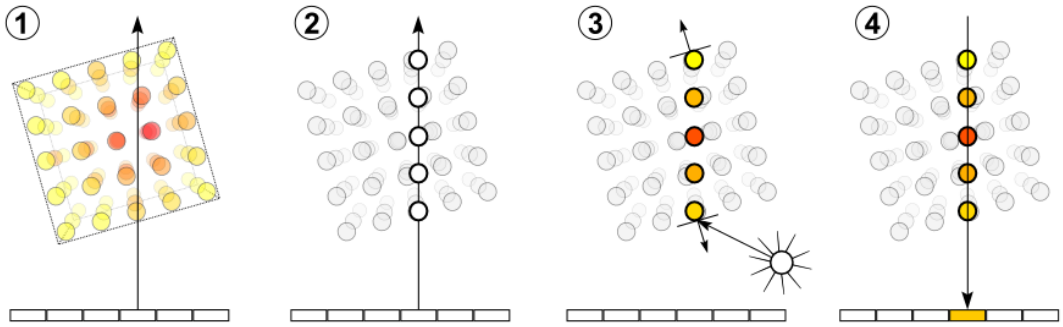


Figura 2.3: pasos de ray casting para un pixel de la imagen. (1) Se lanza un rayo desde el ojo, pasando por un píxel de la imagen. (2) Se reconstruyen muestras del volumen a lo largo del rayo. (3) Se clasifican e iluminan dichas muestras. (4) Se componen las muestras para obtener el color del pixel

2.6.2 Clasificación

En el despliegue de volumen existen diversos métodos para procesar dicho volumen y ser mostrado por pantalla, estos métodos de clasificación de volumen ayudan a tener una mejor visualización de este (K. & T., 2001).

Clasificación en este caso es la asignación de un color una opacidad a ese valor del conjunto datos. Dentro de los métodos de clasificación tenemos:

PRE-CLASIFICACIÓN:

Indica la aplicación de la función de transferencia a los puntos de muestreo discretizados antes de la etapa de interpolación de datos. En otras palabras, el color y la absorción son calculados en una etapa de pre procesamiento para cada punto de muestreo y luego se usa para interpolar el color y la opacidad con el fin de calcular la representación integral del volumen

POST-CLASIFICACIÓN:

El orden de las operaciones se invierte, con respecto a la Pre-clasificación. Este tipo de clasificación se caracteriza por la aplicación de la función de transferencia después de la interpolación de la opacidad a partir de los valores escalares de los puntos de muestreo discretizados.

PRE-INTEGRACIÓN:

La idea principal de la clasificación pre-integrada es dividir el proceso de integración numérica. La integración por separado del campo escalar continúo y las funciones de transferencia se lleva a cabo para hacer frente a la problemática de la frecuencia de Nyquist. Mientras el campo escalar del volumen puede ser suave, la función de transferencia entre dos muestras del campo escalar podría requerir una alta tasa de muestreo para capturar todos los detalles.

Para evitar el muestreo excesivo de la función de transferencia durante el rendering, una alternativa es pre calcular la integral entre cada par posible de muestras cuantizadas de un rayo, y en tiempo de rendering, obtener la integral pre-calculada entre cada par de muestras consecutivas en O(1).

En la Fig 2.4 podemos apreciar el rendering utilizando las diferentes técnicas de clasificación.

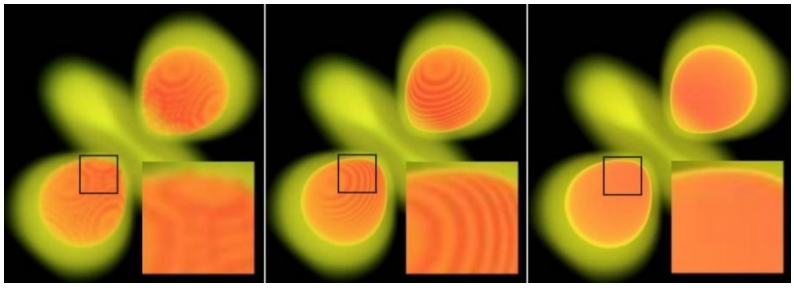


Figura 2.4: Ejemplo de las diferentes clasificaciones: de izquierda a derecha (a)pre-clasificacion (b)pos-clasificacion (c)pre-integracion

2.7 - Ecuaciones Implícitas

Una función explicita, es una función que es dada en términos de una o más variables independientes. Por ejemplo, en la siguiente función $y = f(x) = x^2 + 3x - 8$, y es la variable dependiente y es dada en términos una variable independiente x.

Se llama variable independiente a los valores que pueden tomar los elementos del dominio de la función. Generalmente se denota por *x* para funciones bidimensionales. En el ejemplo anterior, $x \in Z$ ya que así fue especificada. Se llama variable dependiente a los valores que pueden tomar la imagen o rango de la función. Generalmente se denota por la letra y para funciones bidimensionales, dónde $y = f(x)$. En el ejemplo anterior,

$$y \in \{-2, -53, -73, -43, -83, -1, \dots\}$$

Los valores de *x* que anulan la función (i.e. *x*:*f*(*x*)=0) son las raíces de la función. También son llamados los ceros de la función.

Las funciones implícitas, por otro lado, son usualmente dadas en término de ambas variables (dependientes e independientes), como por ejemplo: $f(x, y) = 2y + sen(-x)$.

La ecuación de una función, es la expresión algebraica que resume cómo se obtienen los valores del conjunto final a partir de los valores del conjunto inicial función, tal y como se presenta la función explícita: $y = f(x) = x^2 + 3x - 8$. Similarmente, la ecuación implícita no es más que el resumen de los valores obtenidos a partir de una Función Implícita. Por ejemplo,

$\{ (x,y) \in \Re^2 : f(x,y) = y + x^2 - 3x + 8 = 0 \}$

Los ceros de la función *f*(*x*,*y*) son en este caso los puntos (*x*,*y*) que pertenecen a la ecuación implícita.

2.8 Ceros de funciones

Existen diversos métodos para hallar las raíces de funciones; la idea es obtener una solución de la ecuación $f(x) = 0$, para una función dada f. Entre los métodos se encuentran los métodos cerrados (Bisección, Regula Falsi, Müller) los cuales parten de un intervalo dado para hallar la raíz de dicha ecuación. Por otro lado se encuentran los métodos abiertos (Newton, Secante) que establecen distintas condiciones para hallar la raíz.

Varios de estos métodos para hallar raíces requieren ser comprobada si la función preserva el orden (monótona creciente) o es reverso (monótona decreciente), ya que existen funciones que no son monótonas y puede que no exista la raíz en ese intervalo seleccionado.

Una función f que define un subconjunto de números reales se llama monótona creciente, si para cada x e y tales que $x \leq y$ se tiene que $f(x) \leq f(y)$. Del mismo modo una función se llama monótona decreciente, si para cada x e y tales que $y \leq x$ se cumple que $f(x) \geq f(y)$. La monotonicidad puede estar presente en un sub dominio de la función (por ejemplo, en un sub intervalo), o es todo su dominio.

A continuación, estudiaremos varios métodos clásicos de ceros de funciones, a saber: Bisección, Newton, Secante, Regula Falsi y Müller.

2.8.1 Método de Bisección

Supongamos que f es una función continúa definida en el intervalo $[a,b]$ con $f(a)$ y $f(b)$ de signos diferentes. De acuerdo con el teorema del valor intermedio, este teorema nos indica que si u es un numero entre $f(a)$ y $f(b)$ con $f(a) > u > f(b)$ o $f(a) < u < f(b)$, entonces existe un $c \in [a,b]$, tal que $f(c) = u$. Si bien el procedimiento se aplica, aunque exista más de una raíz en el intervalo (a,b), por razones de simplicidad suponemos que la raíz de este intervalo es única. El método requiere dividir varias veces a la mitad los sub intervalos de $[a,b]$ y, en cada paso, localizar la mitad que contenga a c.

Para empezar, supongamos que $a_1 = a$ y $b_1 = b$, y sea m_1, el punto medio de $[a,b]$; es decir:

$$m_1 = a_1 + \frac{b_1 - a_1}{2} = \frac{a_1 + b_1}{2}.$$

Si $f(m_1) = 0$, entonces $c = m_1$; de no ser así entonces $f(m_1)$ tiene el mismo signo que $f(a_1)$ o $f(b_1)$. Si $f(a_1)$ y $f(m_1)$ tienen el mismo signo, entonces c pertenece (m_1, b_1) y tomamos $a_2 = m_1$ y $b_2 = b_1$. Si $f(a_1)$ y $f(m_1)$ tienen signos opuestos, entonces c pertenece (a_1, m_1) y tomamos $a_2 = a_1$ y $b_2 = m_1$. Después volvemos a aplicar el proceso al intervalo $[a_2, b_2]$ tomando el punto medio m_2, de forma iterativa hasta obtener la raíz de dicha función con ciertos decimales de

precisión. Haciendo un análisis del método se garantiza la convergencia de la raíz de *f* si *f* es una función continua dentro del intervalo establecido $[a, b]$ y tanto *f*(*a*) como *f*(*b*) tienen signos opuestos. El error absoluto se reduce a la mitad por cada paso, por lo que el método converge de manera lineal (ver Fig. 2.5). El error luego de *n* iteraciones es acotado por $|m_n - c| <= |b\text{-}a|/2^n$. Esta fórmula es usada para determinar el número de iteraciones para converger a la raíz deseada con una cierta tolerancia *t*. Así, $n >= \log_2(b\text{-}a)\text{-}t$.

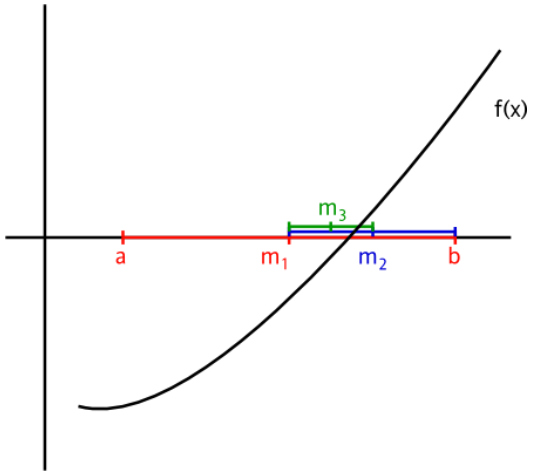


Figura 2.5: Ejemplo de aplicación del Método de Bisección [Ziegler, 2004]

2.8.2 Método de Newton

Parte de una aproximación inicial x_0 y obtiene una aproximación mejorada, x_1, dada por la fórmula:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad (1)$$

La ecuación 1 anterior puede derivarse del desarrollo en serie de Taylor. Para un entorno del punto x_n: $f(x) = f(x_n) + f'(x_n)(x - x_n) + (x - x_n)^2 \frac{f''(x_n)}{2!} + \ldots$ si se trunca el desarrollo a partir del termino de grado 2, y evaluamos en x_{n+1}:$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$, y si además se acepta que x_{n+1} tiende a la raíz, se ha de cumplir que $f(x_{n+1}) = 0$. Luego sustituyendo en la expresión anterior, obtenemos 0 = $f(x_n) + f'(x_n)(x_{n+1} - x_n)$. Finalmente, despejamos x_{n+1} y obtenemos la ecuación (1).

El método de Newton tiene una interpretación geométrica sencilla, como se puede apreciar del análisis de la Fig. 2.6. De hecho, el método de Newton consiste en una linealización de la función, es decir, f se reemplaza por una recta tal que contiene al punto $(x_0, f(x_0))$ y cuya pendiente coincide con la derivada de la función del punto, $f(x_0)$. La nueva aproximación a la raíz, x_1, se obtiene de la intersección de la función lineal con el eje x.

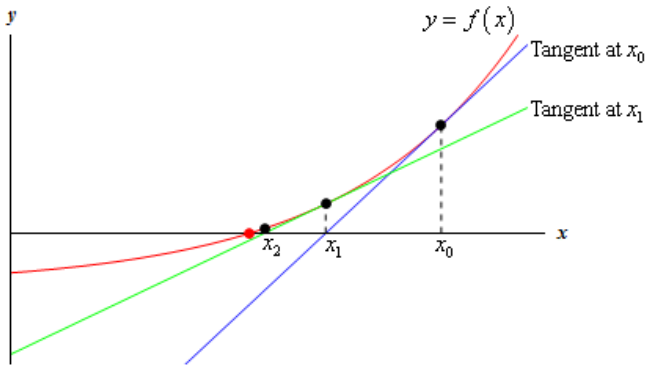


Figura 2.6: Ejemplo de la aplicación del Metodo de Newton [Pauls, 2011]

Otra forma de derivar la ecuación (1) es utilizando la ecuación de la recta que pasa por el punto $(x_0, f(x_0))$ y de pendiente $y - f(x_0) = f(x_0)(x - x_0)$. Haciendo $y = 0$ y despejando x obtenemos la ecuacion (1). La convergencia de este método es cuadrática.

2.8.3 Método de la Secante

Requiere dos puntos iniciales, los cuales pueden ser arbitrarios. Consiste en trazar rectas secantes a la curva de la ecuación que se está analizando, y verificar la intersección de dichas rectas con el eje de las *x* para determinar si es la raíz que se busca (ver Fig 2.7).

Al ser un método abierto, converge con la raíz con una velocidad semejante a la de Newton, aunque similarmente al método de Newton, la convergencia no está garantizada si el iterado inicial está lejos de la raíz. Su principal diferencia con el método de Newton es que no se requiere obtener la derivada de la función para realizar las aproximaciones, lo cual facilita las cosas al momento de crear un código para encontrar raíces por medio de este método. La convergencia de este método es de 1.62 (súper lineal), pero no llega a ser cuadrática.

El método de la secante se basa en la fórmula de Newton, pero evita el cálculo de la derivada usando la siguiente aproximación: $f'(x_i) \approx \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}$. Sustituyendo en la fórmula de newton, obtenemos:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \approx x_i - \frac{f(x_i)}{\frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}}, x_{i+1} \approx x_i - \frac{f(x_i)(x_{i+1} - x_i)}{f(x_{i+1}) - f(x_i)}$$

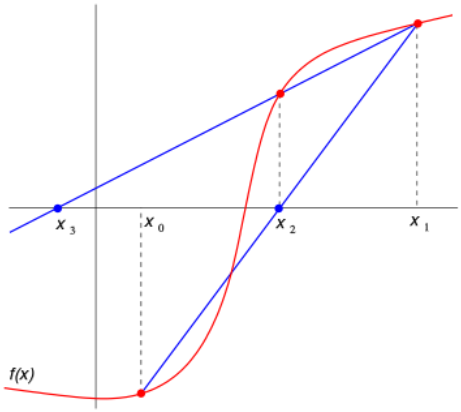


Figura 2.7 Ejemplo de la aplicación del Método de la Secante

2.8.4 Método de Posición Falsa o Regula Falsi

El método de la posición falsa usa lo mejor de la bisección y del método de la secante. Este método, como en el método de la bisección, parte de dos puntos que rodean a la raíz $f(x) = 0$, es decir, dos puntos x_0 y x_1 tales que $f(x_0)f(x_1) < 0$. La siguiente aproximación, x_2, se calcula como la intersección con el eje x de la recta que une ambos puntos (empleando la ecuación $x_2 = x_0 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_0)$ del método de la secante). La asignación del nuevo intervalo de búsqueda se realiza como en el método de la bisección: entre ambos intervalos $[x_0, x_2]$ y $[x_2, x_1]$, se toma aquel que cumpla $f(x_{min})f(x_{max}) < 0$. En la Fig. 2.8 se representa geométricamente el método. Similar a Bisección, la convergencia de este método es lineal, e igualmente la convergencia está garantizada.

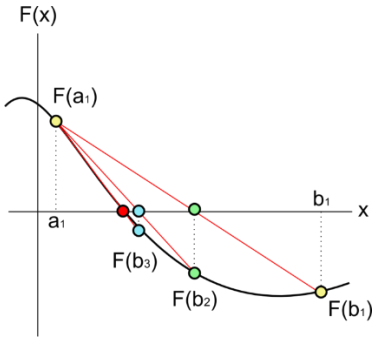


Figura 2.8 Ejemplo de la aplicación del método de Regula Falsi

2.8.5 Método de Müller

Consiste en utilizar tres aproximaciones iniciales x_0, x_1 y x_2 a la raíz de $f(x) = 0$, y determinar la siguiente aproximación al considerar la ecuación de la parábola que pasa por los puntos $(x_0, f(x_0))$; $(x_1, f(x_1))$ y $(x_2, f(x_2))$. La intersección con el eje x en el punto (x_3, O) define la aproximación a la raíz de f. Para hallar x_3, primero se encuentra los coeficientes de la ecuación de la parábola.

$$y(x) = a_0(x - x_2)^2 + a_1(x - x_2)^2 + a_2,$$

donde:

$$a_0 = \frac{(x_1 - x_2)[f(x_0) - f(x_2)] - (x_0 - x_2)[f(x_1) - f(x_2)]}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)},$$

$$a_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} + (x_2 - x_1)a_0,$$

$$a2 = f(x_2).$$

La aproximación a la raíz x_3 del polinomio es obtenida de la siguiente forma,

$$x_3 = x_2 - \frac{2\,a_2}{a_1 + sign(a_1)\sqrt{a_1^2 - 4\,a_0\,a_2}}$$

.

Para continuar con el proceso, se eligen de las tres aproximaciones iniciales las dos más próximas a x_3, y luego se renombran como x_0, x_1 y x_2, y se repite el proceso tanto como desee (Ver Fig. 2.9). La velocidad de convergencia del método es 1.84.

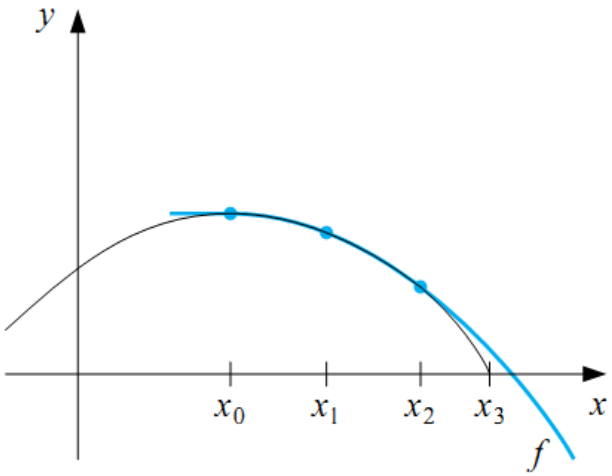


Figura 2.9: Ejemplo de la aplicación del método de Müller (DaFeda, 2010)

2.9 Evaluadores sintácticos

Antes de explicar los evaluadores sintácticos tenemos que tener en cuenta que existe un proceso de traducción por parte de la computadora en donde, en nuestro caso, la entrada de datos

es la ecuación (Aho, Lam, & Sethi, 2007) (Northwood, 2009). La traducción está compuesta por dos etapas:

- Análisis léxico: el flujo de caracteres de entrada se convierte en un flujo de lexemas. Un lexema es una unidad gramatical mínima reconocida por el lenguaje. Por ejemplo, un lexema puede ser un identificador, un operador, una palabra reservada del lenguaje o una constante (números, strings, etc.)
- Análisis sintáctico: a este flujo de lexemas se le aplican reglas de una gramática que define al lenguaje que queremos reconocer. Estas reglas generalmente forman una gramática libre de contexto, aunque depende del lenguaje que se quiere reconocer.

2.9.1 Análisis Léxico

El análisis léxico es la extracción de palabras individuales o lexemas a partir de una secuencia de símbolos. Otros roles del analizador léxico incluyen remover los espacios en blanco y los comentarios.

El proceso de transformación a lexemas, consiste en la construcción de un autómata finito que agrupa los símbolos de la entrada. Este autómata se construye una sola vez y es usado cada vez que se desea hacer el análisis léxico de un nuevo conjunto de símbolos. Generalmente un lexema puede ser un identificador, operadores del lenguaje, palabras reservadas y constantes (numéricas y cadenas de caracteres).

Así, una entrada como la siguiente: “if (velocidad > 50)” se traduce a “<if> <paréntesis abierto> <identificador> <operador relacional> <constante> <paréntesis cerrado>”, obteniendo a partir de un flujo de símbolos un flujo de lexemas.

El autómata se construye de forma tal que se reconozca siempre el lexema más largo posible. Así, por ejemplo, el lexema “abc” podría identificarse como tres identificadores seguidos (a, b y c) o como un solo identificador. Esta regla garantiza que el identificador más largo es reconocido.

Para construir un analizador léxico pueden definirse los lexemas del lenguaje con una gramática lineal (usando una expresión regular), y aplicando el algoritmo de construcción de Thompson es posible crear el autómata asociado. Pueden también usarse programas como flex los cuales permiten definir los lexemas del lenguaje y generan código C para manejar el autómata y reconocer los distintos lexemas. El autómata se representa de forma muy compacta usando una tabla de transición de estados.

En el caso de las ecuaciones es necesario definir las expresiones regulares que definen a cada lexema del lenguaje: nombres de funciones, operadores y constantes, y luego generar un analizador sintáctico usando algún generador de analizadores léxicos.

2.9.2 Análisis Sintáctico

Una vez que le flujo de símbolos es convertido es un flujo de lexemas, este debe ser procesado por el analizador sintáctico. Debe existir una gramática libre de contexto que represente

al lenguaje que se desea reconocer. Esa gramática especifica mediante reglas, cuáles son las construcciones válidas del lenguaje, esto es, de qué forma es posible combinar los lexemas para reconocer frases del lenguaje.

El resultado del análisis sintáctico es un árbol que representa el análisis que se le hizo a la entrada. Este árbol almacena en cada nodo un lexema, y su estructura indica que operación de debe aplicar. Por ejemplo, en el caso de ecuaciones, un operador binario tendrá siempre dos nodos hijos (las dos expresiones que se desean sumar). Un nombre de una función tendrá tantos hijos como parámetros tenga la función. Identificadores y constantes numéricas aparecerán siempre como hojas de éste árbol (ver Fig 2.10).

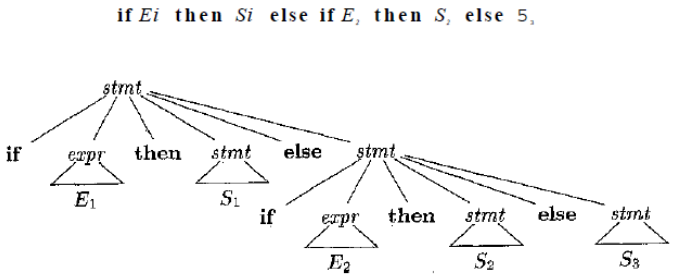


Figura 2.10: Ejemplo de un árbol de análisis

Un analizador sintáctico necesita un autómata de pila para poder representarse. Existen varios tipos de algoritmos para construir analizadores sintácticos, pero los dos tipos más usados son los analizadores descendientes (top-down) y ascendiente (bottom-up).

2.9.3 Top-Down

El análisis top-down puede ser desglosado en dos clases: analizadores en backtracking, en el cual intenta aplicar la regla de la gramática y si falla retrocede un paso; y los analizadores predictivos, que consisten en tratar de predecir el próximo símbolo no terminal en la entrada usando uno o más tokens de búsqueda hacia adelante. Los analizadores en backtracking pueden manejar gramáticas complejas, pero los analizadores predictivos son más rápidos.

2.9.3.1 Descendiente Recursivo

Los analizadores sintácticos predictivos permiten reconocer un conjunto de las gramáticas libres de contexto, en las que las reglas gramaticales permiten decidir sin ambigüedad cual regla aplicar. Sin embargo, existen gramáticas libres de contexto en donde esto no es posible, y es necesario utilizar un analizador sintáctico más poderoso.

Los analizadores recursivos descendientes son predictivos, pueden identificar cuál regla aplicar examinando uno o más lexemas (K). De ahí, este grupo de analizadores sintácticos son llamados LL(K), en donde K es el número de lexemas que deben examinar para saber que regla aplicar. Los más sencillos son los LL(1) que al examinar el siguiente lexema pueden reconocer cuál regla de la gramática aplicar. Su implementación es bastante sencilla, ya que generalmente cada no terminal de

la gramática se corresponde con una función que toma decisiones dependiendo del siguiente lexema leído. Cuando la gramática no permite utilizar este tipo de analizadores sintácticos es necesario utilizar analizadores bottom up.

2.9.3.2 Análisis tipo LL(1)

El LL(1) es un analizador tipo top-down que usa una pila como memoria. En el inicio, el símbolo inicial es puesto dentro de la pila, y próximo a esto tiene 2 acciones disponibles: *Generar*, que consiste en remplazar un símbolo no terminal A en el tope de la pila por una cadena de caracteres α usando la regla de la gramática A → α; y la otra acción *Match*, la cual verifica que el token en el tope de la pila y el próximo token de entrada coincide (y, en el caso de acierto, desapila ambos). La acción es seleccionada usando la tabla de análisis.

2.9.4 Botton-Up

El análisis de forma top-down trabaja trazando las derivaciones por la izquierda, mientras que el análisis de forma bottom-up trabaja haciendo una derivación por la derecha de forma inversa. La forma de frase derecha, son las derivaciones de la extremidad derecha que se le pueden hacer a una cadena. Y un análisis de una cadena de caracteres consume tokens de izquierda a derecha hasta la derivación extrema derecha inversa. Una forma de manejarlo es usando una cadena de caracteres definida por una expansión de un símbolo no terminal en la forma de frase derecha.

Este método de análisis trabaja comenzando con una pila vacía y teniendo dos operaciones: *shift* (salto), el cual coloca el próximo token de entrada dentro de la pila; y el otro método *reduce* (reducción), remplazando el lado derecho de la regla de una gramática con su lado izquierdo. La pila de análisis se mantiene con tokens que son desplazados en el hasta que se tiene un manejador en el tope de la pila, con lo cual vamos a reducirlo mediante la inversión de expansión.

2.9.4.1 Análisis tipo LR(0)

Un objeto tipo LR(0) es una forma de monitorear el progreso hacia un manejador. Este es representado por unas reglas de producción en conjunto con un punto. El lado derecho de la gramática tiene una parte detrás del punto y la otra parte en frente del punto. Esto nos dice que el análisis ha igualado una sub cadena de caracteres derivada por el componente que está a la izquierda del punto y que ahora nos sirve para hacer match a lo que esté en el flujo de entrada, definido por un componente a la derecha del punto.

El análisis LR(0) pertenece a una clase general de analizadores, llamada analizadores tipo LR. Estos pueden tomar ventaja de los símbolos que están más adelantes, muy parecido a los analizadores de top-down donde el lado izquierdo se expande en el lado derecho basado en un símbolo que está más adelante (si es que hay). Los principales métodos tipo LR son:

- SLR(1) – el lado izquierdo solo se remplaza con el de la derecha si el símbolo de más adelante está en el conjunto Follow del lado izquierdo.
- LR(1) – este utiliza un subconjunto del Follow set del lado izquierdo que tiene en cuenta el contexto (el árbol a lo anterior e izquierda del lado izquierdo).

- LALR(1) – esto reduce el número de estados comparado con LR(1) y (si tenemos suerte), usa un subconjunto apropiado del Follow set del lado izquierdo.

2.9.5 Recuperación de Errores

Similar al análisis LL(1), existen tres posibles acciones para el manejo de errores:

- Desapilar un estado de la pila
- Desapilar un token de la entrada hasta tener uno aceptable (el cual reanudará el análisis)
- Apilar un Nuevo estado dentro de la pila

2.10 Trabajos anteriores

El trabajo de Luiz H. de Figueiredo en el 1992 nos presenta métodos discretos de base física para generar aproximaciones poligonales de funciones implícitas e incluso superficies implícitas. Estos procedimientos no solo generan una aproximación de la superficie, sino también producen una estructura adecuada para simulación numérica y la modelación de base física y sistemas de animación. Al mismo tiempo describen cómo son estas ecuaciones y cuál es el uso que se quiere dar en la modelación física, y establecen que cada superficie se compone de estructura poligonal. También ofrecen dos sistemas para construir las aproximaciones poligonales, que incluyen el sistema discreto físico, el cual se abstrae de la materia ensamblada por partículas unidas a otras fuerzas. Varias de estas fuerzas físicas pueden ser naturalmente modeladas usando sistemas discretos. Los sistemas de partículas consisten en tener un conjunto finito de partículas el cual tiene una posición inicial en el espacio y el comportamiento a través del tiempo es gobernado por una serie de reglas algorítmicas. El segundo sistema son los de *Spring-Mass*, "resortes-masa", que es un tipo de partícula física en el sistema estructurado por la unión de pares de partículas con resortes. Los resortes imponen una fuerza interna que depende de la distancia entre esas partículas y es gobernada por el comportamiento global del sistema. Luego de tener estos dos sistemas, los autores establecen un sistema dinámico de poligonización usando un sistema de partículas. En el primer caso hacen un muestreo de cada una de las partículas (ver Fig 2.11) y las estructuran creando una malla (ver Fig 2.12) y de esta forma modelan la superficie deseada (ver Fig 2.13), bajo la simulación de física de partículas. Para el caso de la poligonización por medio del sistema de Spring-Mass, se hace una subordinación triangular, donde los elementos de estos sistemas están asociados con la triangulación Freudental del espacio.

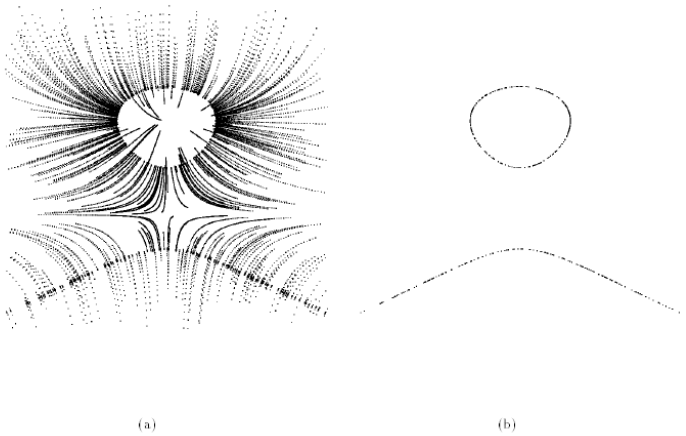


Figura 2.11: Ejemplo de (a)(izquierda) trayectorias y (b)(derecha) posicionamiento de las partículas en 2D

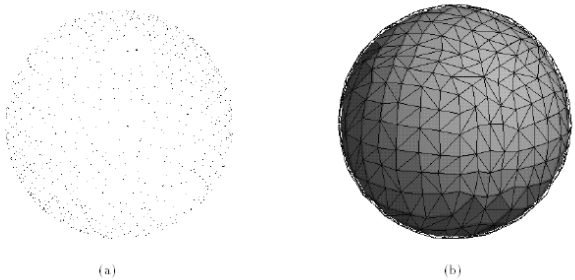


Figura 12: Ejemplo de Puntos de Control de la esfera(a) y creación del mallado de esfera (b)

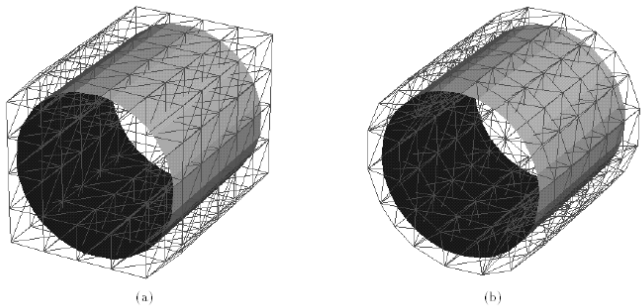


Figura 13: Ejemplo de mayado 3D antes (a) y después (b) de la deformación usando Spring Mass, como se puede apreciar el mallado cambia ajustándose a la figura.

Estos sistemas están sujetos a fuerzas de deformación derivadas de la gradiente de la ecuación principal. Su posición de equilibrio da la triangulación de una región del espacio que contiene la ecuación M y tiene las siguientes propiedades:

a.- M es una transversal a la triangulación

b.- Las divisiones son cuasi-regulares

c.- Por cada enésimo simplex σ que intersecta M existe un punto perteneciente a M cercano al baricentro de σ cuya tangente del espacio M en el punto p es cercano al soporte del hiperplano de una de las caras de σ. (ver Fig. 2.14)

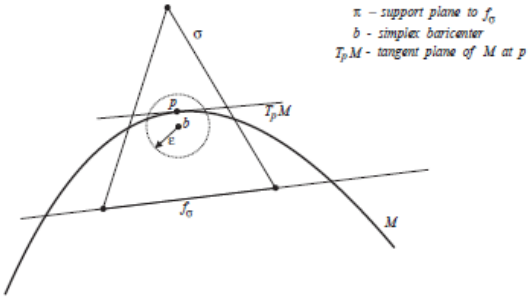


Figura 2.14: Triangulación Subordinada, proceso previo antes de crear el mallado

La generación del mallado por Spring-Mass consiste en la triangulación Freudental, que es creada a partir de la ecuación principal, usando como límite un volumen que delimita la ecuación implícita. Cada simplex (triángulo) que intersecta la ecuación implícita es identificada. En conjunto, estos forman una intersección compleja simplicial. El sistema Spring-Mass es creado por la asociación de los nodos de masa y los resortes a los vértices y aristas de cada intersección compleja. Luego de esta generación del mallado se usa un enfoque físico para la obtención de la triangulación final que será usada para realizar la poligonalización de M (Luiz, Gomez, Demitri, & Luiz, 1992).

Años más tarde, en 1994, el trabajo realizado por Jules Bloomenthal, de la universidad de Calgary, de *An Implicit Surface Polygonizer*, nos describe cómo a partir de una ecuación implícita, puede construirse un mallado de triángulos para ser visualizado en pantalla. Esto permite que una superficie sea desplegada de la forma convencional a partir de polígonos. La poligonalización se realiza mediante el uso de la técnica de tetra cubes, combinado con Regula Falsi para mejorar la precisión en la búsqueda de ceros de funciones (ver fig. 2.15a y 2.15b). A partir de la ecuación implícita se genera un volumen discreto (malla regular), evaluando la función implícita en un conjunto de puntos (x,y,z) igualmente espaciados. Luego, cada conjunto de 8 muestras conectadas (cubo), es dividido en tetraedros, en donde se evalúa si la superficie corta al tetraedro. Para ello se tiene una tabla de 16 casos, de las 16 formas en que la superficie puede cortar al tetraedro, ya que la función puede ser negativa o positiva en cada vértice del tetraedro. Originalmente el corte de la superficie (los ceros de la función) con el tetraedro se aproxima por interpolación lineal, pero dado que se conoce la ecuación implícita, se utiliza Regula Falsi para obtener una mejor aproximación a la raíz en cada arista intersecada (Bloomenthal, 1994).

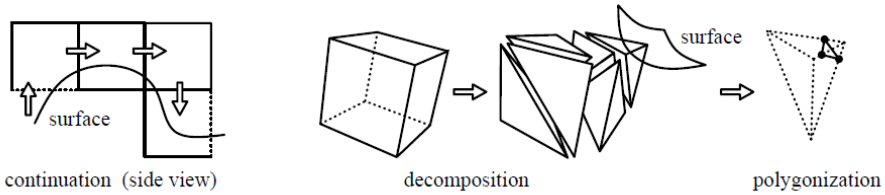


Fig 15a proceso de poligonización

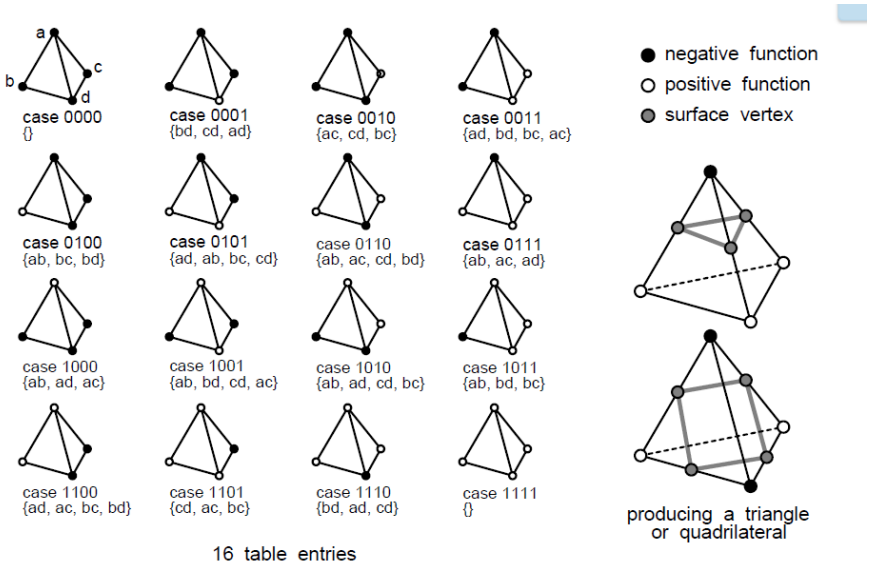


Fig 15b configuración de entradas del poligonizador usando tetra cubes.

En el año 2001, el trabajo de Klaus Engel et al. exploran el uso rendering de volúmenes para visualizar resonancias magnéticas, tomografías computarizadas, e incluso volúmenes sintéticos provenientes de ecuaciones implícitas. La técnica de rendering utilizada se basa en el uso de texturas 2D y texturas 3D, con planos alineados. Utilizan la técnica de pre-integración para mejorar la calidad visual del rendering.

En el año 2001, el trabajo de R. Carmona et al., reconstruye superficies, pero combinando Marching Cubes con los métodos de ceros de funciones como Regula Falsi y Bisección. El sistema RenderAll permite adicionalmente realizar el rendering del volumen a la par que la superficie reconstruida (ver Fig.2.16). El rendering del volumen se realiza con texturas 3D y planos alineados al viewport. Para combinar el rendering de volúmenes con el rendering de una superficie reconstruida, se despliega un plano alineado al viewport, seguido de los polígonos de la superficie que se encuentran entre dicho plano y el próximo plano. La idea se repite hasta haber desplegado todos los planos alineados al viewport. En cada par de planos consecutivos, se despliegan los triángulos de la superficie que estén total o parcialmente entre estos en coordenadas de ojo. Se usa una actualización de una lista de triángulos activos, que ordena los triángulos desde el más lejano al más cercano. Esto permite determinar los triángulos activos para el próximo par de planos con pocos chequeos. Los fragmentos remanentes de triángulos (áreas de triángulos parcialmente fuera del par de cortes)

deben ser cortados para no ser desplegados dos veces. En este caso se utilizan los planos de corte soportados por el hardware gráfico (Carmona., 2001).

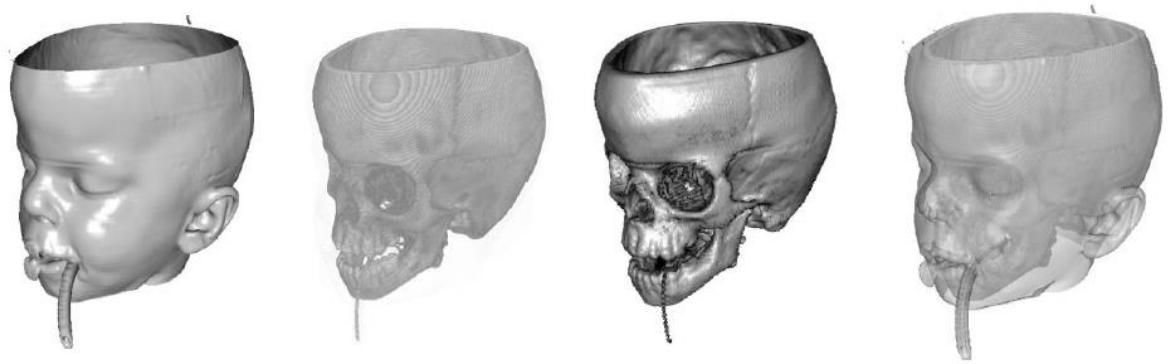


Figura 2.16: RenderAll. La primera imagen de izquierda a derecha muestra el despliegue de la superficie con phong shading. La segunda y tercera imagen de izquierda a derecha muestran el rendering del volumen sin iluminación y con iluminación Phong. La última imagen muestra la mezcla de superficie y volumen intercalando polígonos del volumen con triángulos de la superficie.

A partir de los años siguientes al 2004 se exploran diferentes técnicas para mejorar el rendimiento de estas aplicaciones, evitando el uso excesivo de recursos. Tal es el caso del trabajo de Bruno Rodrigues de Araujo y Joaquim Armando Pires Jorge, titulado *Curvature Dependent Polygonization of Implicit Surface,* en el cual nos explican cómo el enfoque de Triangle Marching reduce esta carga de recursos en la máquina. En sus trabajos previos estudian métodos basados en partición de celdas en los cuales se destacan el Marching Cubes y la variante Marching Tetrahedra. Esto algoritmos ofrece buen rendimiento, pero en ocasiones carece de la calidad adecuada. El segundo método es el Surface Tracking, donde destaca el Marching Triangle y el algoritmo de Hartmann. Básicamente estos métodos generan mallados del mismo tamaño con triángulos cuasi equiláteros. Pero estos no se adaptan a las propiedades de la superficie, tomando muchos triángulos pequeños aproximados a las superficies con pocas variaciones en la curvatura. El último de los métodos son los Mallados Adaptativos. Estos comienzan desde un mallado de alta calidad, aplica operaciones división de aristas y colapso de aristas para simplificar y optimizar el mallado en tiempo real. Al final, se obtienen más triángulos en donde la curvatura es más pronunciada, requiriendo así más detalle (Ara´ujo & Jorge, 2004).

En otro trabajo, realizado por Andrew Corrigan y H. Quynh Dinh, en el 2005, titulado *Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU*, se exploran métodos de funciones con base radial. Esto autores muestran cómo se comporta este método en la GPU, ya que todo es almacenado en una textura 3D y no en una forma convencional (Corrigan & Dinh, 2005).

Un ejemplo de colaboradores de mozilla developer network, el trabajo de Malgoratza Jatczyk, en el 2013, muestra tecnologías Web (usando el API de OpenGL ES) para crear una aplicación demo para triangular y visualizar ecuaciones implícitas en WebGL (Jatczyk, 2013).

Como podemos apreciar los métodos para visualizar ecuaciones implícitas se basan primordialmente en la reconstrucción de una iso-superficie, y mejorar el rendimiento y/o calidad del rendering. Pocos trabajos han utilizado el rendering directo de volúmenes, el cual permite la visualización del volumen sin requerir de la reconstrucción de una superficie intermedia. Una de las ventajas que vemos de utilizar el rendering de volúmenes es que se pueden visualizar distintas "capas" de una ecuación implícita, sin generar un impacto significativo en el tiempo de respuesta de la aplicación, y sin generar mayores retos en el almacenamiento, como sí sucede en el caso de querer reconstruir todas estas capas (que podrían ser decenas), una por una, utilizando mallas de triángulos.

Capítulo 3 – Diseño e Implementación

En este capítulo se explica cómo se realizó el diseño y la implementación del sistema, se hace una breve explicación con diagramas de flujo e imágenes de la interfaz.

Se muestran como los módulos están compuestos y como engranan para generar el rendering. Se hace una descripción con los algoritmos de todo el proceso.

3.1 Funcionamiento general del sistema

El prototipo de sistema se puede describir globalmente en la Fig. 3.1. Al iniciar el sistema, se va a mostrar la ecuación de una esfera la cual es la que se tiene por defecto. Se comienza con la inicialización del sistema con valores por defecto como el tamaño de la caja en los máximos y mínimos (-2,-2,-2) y (2,2,2), el número de voxels en cada dimensión (64), el tamaño de la función de transferencia (1024), 2 valores de superficie (una de color amarillo y otra roja, de grosor 30 píxeles), y como función a visualizar, una esfera unitaria. Durante la interacción del usuario con la interfaz, el mismo puede cambiar cualquiera de los parámetros, en cualquier momento, para definir la entrada de datos.

Al recibir los datos de entrada, estos valores pasan por el generador de la textura del volumen (ver diagrama de flujo Fig. 3.2). El generador de la textura simplemente evalúa la función dentro de la caja, con el espaciado de voxels fijado en cada dimensión. Luego se discretiza la función de transferencia en una paleta de colores o textura unidimensional de tipo RGBA. Inicialmente la función de transferencia contiene dos valores de superficie, aplicados a las posiciones 0 y 222 del rango [0,1023]. Luego se prepara el contexto de render colocando los parámetros adecuados para ejecutar el algoritmo de dos pasadas del *ray casting*, haciendo así el despliegue de la función implícita.

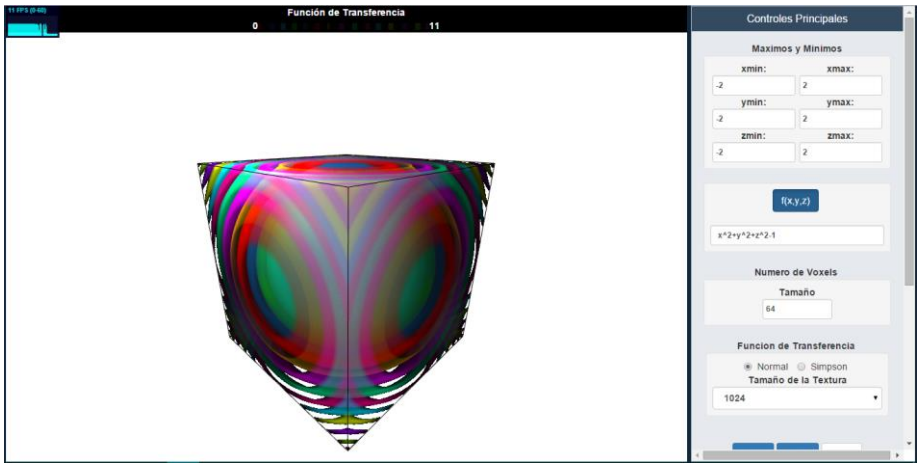


Figura 3.1: interfaz + contenedor Canvas WebGL en una grilla de 9x3 de bootstrap

3.2 Implementación

En la implementación de los módulos de despliegue se hizo con las especificaciones del grupo Khronos group de WebGL 1.0 y la librería de THREEjs que facilita la creación de la escena y del despliegue total del volumen.

A continuación, se introducen las diferentes librerías y plugins usador para la implementación:

Contenido externo utilizado

Para crear el Visualizador de Volúmenes en WebGL, se utilizaron diversos recursos externos entre ellos librerías y otros como plugins de javasript, los cuales son nombrados a continuación:

- THREEjs: librería que crea primitivas de WebGL con mayor facilidad que la versión estándar de WebGL.
- OrbitControls: librería externa a THREEjs permite colocar controles sobre la cámara.
- Bootstrap: agrega estilos al diseño de la página.
- JQuery: librería con el fin de mejora desarrollo y uso de las variables de javascript.
- JQuery-UI: es extensión de jQuery que permite agregar *widgets* u objetos de interfaz gráfica.
- Math: librería que agrega más funciones a la librería básica de matemáticas de javascript, permite compilar, analizar expresiones matemáticas.
- Detector: plugin que informa si ocurrió algún error en la instancia de WebGL sobre un objeto canvas.
- Stats: plugin que visualiza información sobre cuadros por segundo/tiempo de muestreo/ memoria usada

En la figura 3.2 se muestra el diagrama de flujo de cada uno de los módulos y como se comunican entre sí hasta general la imagen de salida por pantalla

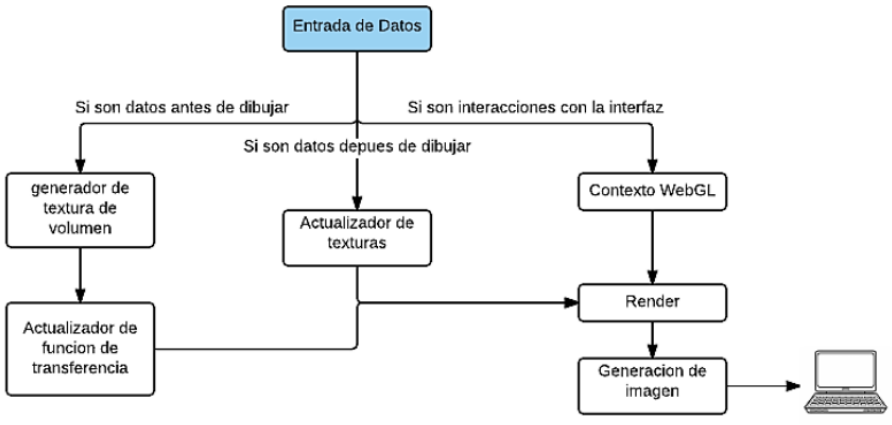


Fig 3.2 diagrama de la aplicación general

A continuación, se explicará cómo fue implementado cada módulo, usando como guía el diagrama de flujo

Se tomaron en consideración limitantes de la especificación de WebGL 1.0 en el que nos indica que no se puede hacer uso de texturas 3D y texturas que no sean en potencia de 2, pero existen métodos para hacer creer al programa que está leyendo una textura como si fuera 3D. En los programas de shaders se usa un algoritmo que hace procesamiento de un Atlas de textura 2D como si fuera una textura 3D.

3.2.1 Generador de la textura del volumen

Haciendo seguimiento al diagrama de flujo por medio de la entrada de datos, si el usuario introdujo o hizo cambios en el tamaño de la caja contenedora del volumen (Mínimos y Máximos), o en el Número de Voxels, o en la ecuación implícita, este módulo crea el volumen.

Para la creación del atlas de textura se usó una librería de JavaScript *mathjs* para poder evaluar con exactitud si la expresión escrita, es válida. Esta librería ofrece una extensa cantidad de funciones que pueden ser usadas en conjunto con expresiones matemáticas en las que podemos listar: +,-,*,/,agrupación (),pre multiplicación <numero>(). Una vez evaluada es almacenada en una variable para su uso en la construcción del volumen. En el proceso de creación del atlas se usa un arreglo que representa la matriz del volumen 4 componentes (RGBA). En cada componente se almacena un valor de la gradiente de la función (RGB) y en el 4to componente (A) se guarda el valor de la función $f(x,y,z)$. El vector gradiente se almacena con el fin de aplicar un modelo de iluminación local durante el rendering. El volumen en enviado al GPU como un atlas de textura 2D, representando lógicamente una textura 3D.

3.2.2 Actualizador de función de transferencia

Dentro de esta etapa, se construye la función de transferencia la cual especifica en qué posición sobre el volumen se encuentra cada iso-valor, de qué color es, y qué opacidad tiene. La función se almacena en una textura 1D de tipo RGBA que por defecto abarca el rango de [0,1023]. Al finalizar esta etapa, la textura se carga en una unidad de textura, y recibe como nombre "transferencia" en el programa de shader.

3.2.3 actualizador de las texturas

Cada vez que el usuario introduce datos de la función de transferencia (inserción, eliminación, modificación de un iso-valor y sus atributos), se actualiza la textura correspondiente a la función de transferencia. Si el usuario realiza alguna modificación sobre los parámetros de la función implícita (caja, número de voxels o la función en sí), se crea un nuevo volumen RGBA, e igualmente se envía a la unidad de textura correspondiente.

3.2.4 contexto webGl

En este módulo se crea el ambiente para utilizar openGL en el browser, en el cual valores como tamaño de la caja, geometría de despliegue de volumen y los algoritmos involucrados en el *ray casting* conviven con el entorno HTML/CSS/JavaScript de la aplicación.

3.2.5 Módulo de Render

El módulo de rendering realiza un ray casting sobre el volumen. Para poder describir en más detalle cómo se procesa la textura de volumen, se colocan algunos de los algoritmos clave para el desarrollo de la aplicación, donde se describen aspectos importantes y procesos intermediarios antes de hacer el despliegue del volumen e integración con la interfaz. Toda interacción con el resultado del despliegue resulta en una llamada a actualizar el próximo cuadro a dibujar como lo son la rotación, la translación, el zoom in, el zoom out, y llamadas de la interfaz también pasa por este proceso.

```
//Acts like a texture3D using Z slices and trilinear filtering.
//Actua como una textura3D, usa cortes en z y filtro trilinear
vec4 sampleAs3DTexture(sampler2D texturec, vec3 texCoord, float size) {
    float sliceSize = 1.0 / size;                       // space of 1 slice
    float slicePixelSize = sliceSize / size;            // space of 1 pixel
    float sliceInnerSize = slicePixelSize * (size - 1.0); // space of size pixels (interpolacion del primer corte)
    float zSlice0 = min(floor(texCoord.z * size), size - 1.0);
    float zSlice1 = min(zSlice0 + 1.0, size - 1.0);
    float xOffset = slicePixelSize * 0.5 + texCoord.x * sliceInnerSize;
    float s0 = xOffset + (zSlice0 * sliceSize);
    float s1 = xOffset + (zSlice1 * sliceSize);
    vec4 slice0Color = texture2D(texturec, vec2(s0, texCoord.y));
    vec4 slice1Color = texture2D(texturec, vec2(s1, texCoord.y));


    float zOffset = mod(texCoord.z * size, 1.0);
    return mix(slice0Color, slice1Color, zOffset);
}
```

Figura 3.4: Función de muestreos del Atlas 2D

La función presentada en la Fig 3.4 ayuda al raycasting a obtener las muestras de la textura 2D pero haciendo interpolación tri lineal. Una textura 2D de tamaño (n,m,k) es almacenada en una textura 2D de tamaño (n*k,m). Sabiendo esto, se puede realizar una fórmula de acceso para determinar sobre qué par de sub texturas 2D se debe hacer el muestreo para simular la interpolación tri lineal. Una vez determinado el par de sub texturas 2D (zSlices0, zSlices1), se calcula un desplazamiento dentro de cada sub textura 2D para hallar la posición del voxel a muestrear en cada corte. Este desplazamiento es xOffset. Luego de muestrear en ambas sub texturas, se realiza una interpolación lineal entre ambas muestras mediante la función mix.

El algoritmo de la Fig. 3.5 corresponde a la función principal del ray casting. En este se toman una muestra de la textura, luego se extraen valores que conforman las normales de la muestra y es realizado el proceso de iluminación sobre cada una de las muestras, después se extrae el color y se realiza el proceso de pos clasificación, y retorna el acumulado al color fragmento.

Continuando con el algoritmo de la Fig. 3.5 calcula la distancia que hay entre las dos caras de entrada y salida del rayo (*front and back*), se fija un tamaño del paso (*step*) que es dividido entre la diagonal de la caja, se crea un delta que nos indica el tamaño o longitud del rayo y un acumulador de color, se procede a hacer muestreos progresivos según la longitud del rayo y la cantidad de absorción acumulada. La absorción se calcula por la ecuación de *rendering* en donde se evalúa sobre el coeficiente de absorción exp(-0.7 * ColorSample.a), luego, se hace la obtención de una muestra del

volumen dentro del contenedor, se hace la acumulación de color sobre la función de transferencia y este resultado se muestra por fragmento de shader.

```
for(int i = 0; i < 4096 ; i++)
{
    //Get the voxel intensity value from the 3D texture.
    colorSampleA = sampleAs3DTexture( volumetex, currentPosition, maxZ);
    colorSampleB = sampleAs3DTexture( volumetex, currentPosition+deltaDirection, maxZ);
    vec4 SampleNormal =mvMatrix * vec4(normalize((colorSampleA.xyz+colorSampleB.xyz)/2.0),0.0);
    vec3 N = normalize(SampleNormal.zyx);
    vec3 L = normalize(ligDir);

    lambertian = max(dot(N,L),0.0);
    specular = 0.0;

    if(lambertian > 0.0)
    {
        //Especular
        vec3 E = normalize(-eyeVec);
        vec3 H = normalize(L+E);
        specular = specular = pow(max(dot(H, N), 0.0), 16.0);
    }

    colorSample  = texture2D(transferTex, vec2(colorSampleA.a,colorSampleB.b));
    accumulatedLight =  vec4(colorSample.rgb* (ambientLightColor+ lambertian+ specular),1.0);
    alphaSample = 1.0 - exp(-0.7 * colorSample.a);
    //Perform the composition.
    accumulatedColor += accumulatedAlpha * alphaSample * accumulatedLight;
    //Store the alpha accumulated so far.
    accumulatedAlpha *= 1.0-alphaSample;
    //Advance the ray.
    currentPosition += deltaDirection;
    //currentbackpostion-=deltaDirection;
    accumulatedLength += deltaDirectionLength;
    //If the length traversed is more than the ray length, or if the alpha accumulated reaches 1.0 then exit.
    if ((accumulatedLength >= rayLength) || (1.0-accumulatedAlpha >= 0.9))
        break;
}
gl_FragColor  = accumulatedColor;
if(gl_FragColor.a < 0.1)
    discard;
```

Figura 3.5: Algoritmo de RayCasting + BlingPhong

```
materialFirstPass = new THREE.ShaderMaterial( {
        vertexShader: document.getElementById( 'vertexShaderFirstPass' ).textContent,
        fragmentShader: document.getElementById( 'fragmentShaderFirstPass' ).textContent,
        side: THREE.BackSide,
        uniforms: {
          xdistance: {type: "f", value: parseFloat(xmax-xmin)},
          ydistance: {type: "f", value: parseFloat(ymax-ymin)},
          zdistance: {type: "f", value: parseFloat(zmax-zmin)}
        }
      } );

//luego se gurda como textura y es pasada al segudno material para calcular la parte frontal
materialSecondPass = new THREE.ShaderMaterial({
        uniforms: THREE.UniformsUtils.merge([THREE.UniformsLib['lights'],
        {  backside: {type: "t", value: rtTexture},
           maxY: {type: "f", value: parseFloat((m))},
           maxZ: {type: "f", value: parseFloat((k))},
           maxX: {type: "f", value: parseFloat(n)},
           volumetex: {type: "t", value: null},
           transferTex: {type: "t", value: null},
           transferpalette: {type: "t", value: null},
           steps: {type: "1f", value: parseFloat(math.eval("sqrt(64^2+64^2+64^2)"))},
           alphaFixed :{type: "1f", value: parseFloat(1.0)},
           xdistance: {type: "f", value: parseFloat(xmax-xmin)},
           ydistance: {type: "f", value: parseFloat(ymax-ymin)},
           zdistance: {type: "f", value: parseFloat(zmax-zmin)}

         }]),
        vertexShader: document.getElementById('vertexShaderSecondPass').textContent,
        fragmentShader: document.getElementById('fragmentShaderSecondPass').textContent,
        side : THREE.FrontSide,
        lights : true
      });
```

Figura 3.6: Definición de las ShaderMaterial

El algoritmo mostrado en la Fig. 3.6 muestra como es configurados los dos *ShaderMaterial* que conforman los dos pasos del algoritmo de ray casting. El primero se utiliza para desplegar las caras traseras del volumen, representando la salida de los rayos del volumen. Mientras que la segunda pasada, despliega las caras frontales del volumen, y conociendo las caras traseras del mismo, permite calcular la longitud de cada rayo por fragmento.

Luego de pasar por estos módulos se genera una imagen como resultado que es desplegado en el dispositivo de salida (monitor) del computador.

Para el desarrollo de la interfaz (Fig 3.7, 3.8, 3.9), se hace uso de tecnologías estándares de HTML5/CSS/Javascript y de librerías que facilitan la creación, como JQuery para rápida actualización de variables, BootStrap para el diseño y estilo de la página, plugins, como Bootstrap colorpicker y Jquery UI para agregar más características que complementan el diseño de la interfaz.

Mediante la interfaz gráfica, el usuario puede variar los parámetros de la función implícita, y los parámetros de visualización. Dentro de estos controles existen dos etapas, antes de dibujar y después de dibujar.

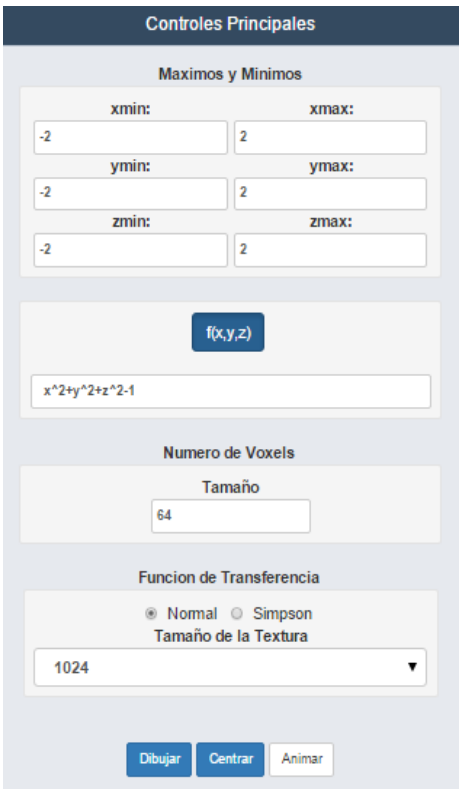


Figura 3.7: Interfaz gráfica para definir la función a visualizar

Figura 3.8: Definiendo los valores de superficie a considerar en la función de transferencia

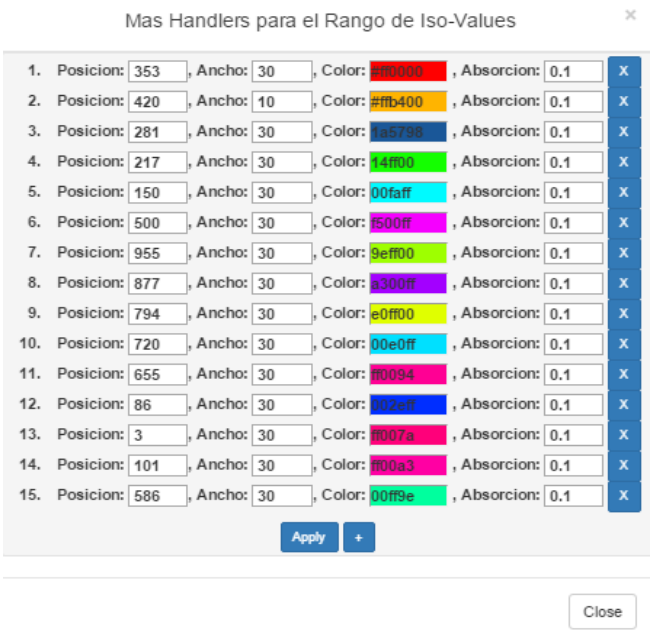


Figura 3.9: Diálogo modal para agregar más Valores de superficie

Como podemos observar en la Fig. 3.7, se encuentran los elementos de la interfaz gráfica en los que tenemos:

- Máximos y Mínimos: son los componentes que definen el tamaño máximo y mínimo de la caja en los ejes x,y,z.
- Ecuación Implícita (f(x,y,z)): en este campo se escribe la función.
- Número de Voxels: es la cantidad de muestras por eje del volumen. En este caso se observa 64, lo que corresponde a 64 en el eje *X*, 64 en *Y*, y 64 en *Z*.
- Funciones de Transferencia: dos opciones Normal (pos-clasificación) y Simpson (pre-integración). Este define de qué forma se construye la función de transferencia y se especifica la resolución que posee la misma (256, 512 o 1024). La función de transferencia pre-integrada queda como trabajo a futuro.
- Dibujar: toma todos los parámetros principales (Máximos y Mínimos, la función y número de voxels) y hace un despliegue de dicha función.
- Centrar: coloca la figura en el centro de la pantalla.

En las figuras 3.8 y 3.9 contienen la interfaz necesaria para definir la función de transferencia, es decir, el conjunto de valores de superficie a visualizar, junto a sus colores y opacidades.

- Rango de Valores de superficie: estos componentes están a lo largo del rango de la función. Su función es indicar en qué parte del volumen están ubicados (posición), su ancho o grosor (Ancho), color y Absorción.
- Iso-Valor Seleccionado: esta ventana muestra el valor seleccionado actualmente con sus características, pueden ser cambiadas en cualquier momento.

- Por último, se encuentra el Botón de agregar más Valores de superficie (Fig. 3.8). Este despliega un diálogo modal en el cual se muestran todos los valores de superficie actuales, sus características si se desean agregar más o eliminar (Fig 3.9).

Capítulo 4 Pruebas y Resultados

Previo a definir las pruebas realizadas junto con sus resultados, es importante establecer los parámetros bajo los cuales se llevaron a cabo estas pruebas, así como los recursos que se han utilizado para poder completar el Visualizador de Volúmenes en WebGL.

4.1 Ambiente de trabajo

Las pruebas se realizaron bajo un ambiente con las siguientes especificaciones:

- Sistema operativo: Windows 10 - 64bits
- Procesador: Intel Core i7 740qm – 1.73Ghz
- Memoria RAM: 8GB DDR3
- Tarjeta Gráfica: Geforce 310m, con 512MB DDR3.

Como herramienta principal para el desarrollo de este trabajo se utilizó Sublime Text editor (Skinner, 2015), debido a la flexibilidad que posee en cuanto al desarrollo web y a la familiaridad que se tiene con la utilización de esta herramienta.

4.2 Funciones de prueba

Se seleccionaron las siguientes 3 funciones:

- Esfera: f(x,y,z) = x^2+y^2+z^2-1
- Toroide: f(x,y,z) = ((X*X)+(Y*Y)+(Z*Z))^2-4*Z*((X*X)+(Y*Y)+(Z*Z))-8*X*X-8*(Y*Y) + 12*(Z*Z) - 16*Z +16
- Jacky: f(x,y,z) = 1/(X*X/9+4*Y*Y+4*Z*Z)^4+1/(Y*Y/9+4*X*X+4*Z*Z)^4+1/(Z*Z/9+4*Y*Y+4*X*X)^4+1/((4*X/3-4)^2+16.0*Y*Y/9.0+16.0*Z*Z/9.0)^4.0+1.0/((4*X/3+4)^2+16.0*Y*Y/9+16*Z*Z/9)^4 + 1/((4*Y/3-4)^2+16*X*X/9+16*Z*Z/9)^4+1/((4*Y/3+4)^2+16*X*X/9+16*Z*Z/9)^4)^-0.25-1

Cada una de estas funciones fue desplegada con 3 valores de superficie de 2% de grosor en la función de transferencia. Como referencia el valor de superficie cuyo color es el azul tiene un valor de opacidad de 1.0, mientras que para los valores de superficie restantes se colocó 0.01 como valor de opacidad (ver Fig. 4.1).

4.3 Pruebas de calidad y rendimiento

El tamaño del volumen medido en voxels y el tamaño de la función de transferencia discretizada influyen tanto en la calidad del despliegue como en el tiempo de respuesta. En esta prueba se pretende realizar una comparación tanto de calidad como de tiempo de respuesta conforme se varían estos parámetros.

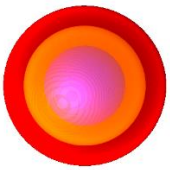
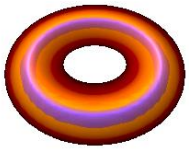
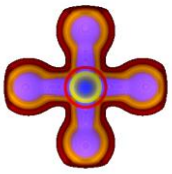
 Esfera	 Toroide	 Jacky

Figura 4.1: ecuaciones utilizadas. En los 3 casos se utilizó una función de transferencia discretizada en 1024 entradas, y se construyó un volumen de 64^3 voxels.

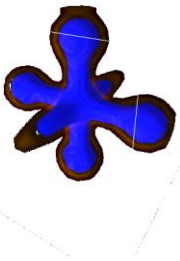
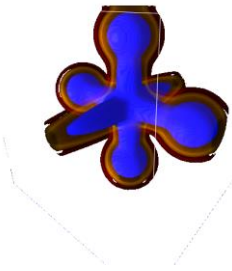
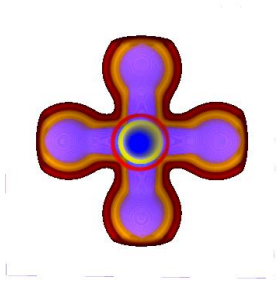
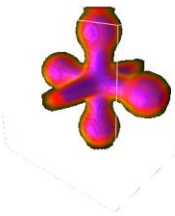
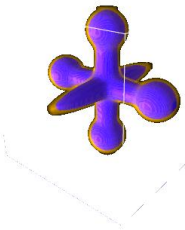
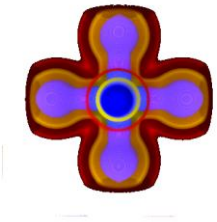
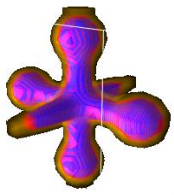
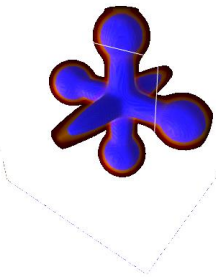
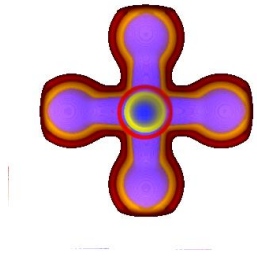
 Función de Transferencia 256 N. de voxels 16x16x16	 Función de Transferencia 256 N. de voxels 32x32x32	 Función de Transferencia 256 N. de voxels 64x64x64
 Función de Transferencia 512 N. de voxels 16x16x16	 Función de Transferencia 512 N. de voxels 32x32x32	 Función de Transferencia 512 N. de voxels 64x64x64
 Función de Transferencia 1024 N. de voxels 16x16x16	 Función de Transferencia 1024 N. de voxels 32x32x32	 Función de Transferencia 1024 N. de voxels 64x64x64

Tabla 4.2 Funciones de Transferencia versus Numero de Voxels

En la Fig. 4.2 podemos apreciar que a medida que el número de voxels aumenta, se pueden apreciar mejor los detalles, y se reducen los artefactos visuales. El mismo fenómeno sucede al aumentar la resolución de la función de transferencia, al discretizarse con más muestras. Los mejores resultados visuales se obtienen con 64^3 voxels y 1024 entradas en la función de transferencia.

En la tabla 4.1 se hizo la prueba de tiempo de respuesta con diferentes tamaños de función de trasferencia y diferentes números de voxels, fijando el paso entre muestras en 1 voxel usando la ecuación del jacky. Muestreos por voxel indica la cantidad de muestreos que se hace por cada ciclo del *raycasting*. Este es definido por 1/diagonal de la caja que es previamente calculado. Si tomamos como medida principal la tasa de cuadros por segundo (fps o *frames per second*), tenemos que el rendimiento disminuye a medida que aumenta la resolución del volumen.

Función de Transferencia	Número de Voxels	Muestreos por voxel	Rata de cuadros por segundo
256	64x64x64	1	35fps
256	32x32x32	1	45fps
256	16x16x16	1	60fps
512	64x64x64	1	35fps
512	32x32x32	1	45fps
512	16x16x16	1	60fps
1024	64x64x64	1	35fps
1024	32x32x32	1	45fps
1024	16x16x16	1	60fps

Tabla 4.1: Función de Transferencia variable vs Numero de voxels variable

Como prueba siguiente, queremos estudiar cómo afecta el paso de rendering o muestreos por voxel en la calidad del rendering. Si este número aumenta (ver Fig. 4.3) podríamos perder detalles en el rendering, pues es posible no capturar todos los valores de superficie adecuadamente, o contar con pocas muestras del volumen para ciertos valores de superficie. El el paso es más corto, se recolecta mejores detalles, y la calidad del rendering es superior.

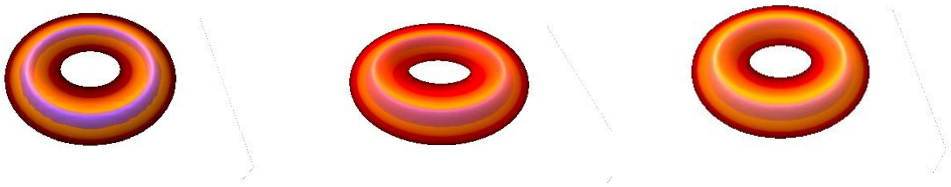


Figura 4.3: diferentes longitudes del paso (1/diagonal, 0.5/diagonal, 0.25/diagonal) con función de transferencia de (1024), y la ecuación del toroide

Como podemos observar en este ejemplo del toroide, si fijamos el paso a 1 podemos tener menor cantidad de muestras lo que traduce a perdida de detalles (ver Fig. 4.3 izquierda) y una rata

de cuadros más estable (ver tabla 4.2). Si fijamos el paso a 0.25 tenemos mayor detalle, pero hay una caída de rata de cuadros por segundo, que, en el peor caso, se tienen apenas 6fps.

Número de Voxels	Cantidad del Paso/ 1 Voxel	Función de Transferencia	Rata de cuadros por segundo
64x64x64	1	256	35fps
64x64x64	1	512	35fps
64x64x64	1	1024	35fps
64x64x64	0.5	256	29fps
64x64x64	0.5	512	29fps
64x64x64	0.5	1024	29fps
64x64x64	0.25	256	6fps
64x64x64	0.25	512	6fps
64x64x64	0.25	1024	6fps
32x32x32	1	256	45fps
32x32x32	1	512	45fps
32x32x32	1	1024	45fps
32x32x32	0.5	256	39fps
32x32x32	0.5	512	39fps
32x32x32	0.5	1024	39fps
32x32x32	0.25	256	18fps
32x32x32	0.25	512	18fps
32x32x32	0.25	1024	18fps
16x16x16	1	256	60fps
16x16x16	1	512	60fps
16x16x16	1	1024	60fps
16x16x16	0.5	256	55fps
16x16x16	0.5	512	55fps
16x16x16	0.5	1024	55fps
16x16x16	0.25	256	39fps
16x16x16	0.25	512	39fps
16x16x16	0.25	1024	39fps

Tabla 4.2: Funciones de Transferencia versus Número de Voxels del volumen y longitud del Paso de rendering

Capítulo 5 – Conclusiones y Trabajos Futuros

En este trabajo se desarrolló un prototipo de sistema web usando el estándar de *WebGL* para despliegue de ecuaciones implícitas mediante la técnica de visualización directa de volúmenes. El sistema permite definir y desplegar múltiples valores de superficie (f(x,y,z) = c) de la función implícita al mismo tiempo, asignando color, opacidad, y grosor por cada superficie, con una interfaz sencilla.

Debido a que es una aplicación web con el estándar *WebGL*, la calidad u velocidad del despliegue dependerá del tipo de navegador web y del hardware utilizado, lo cual puede producir insatisfacción en algunos sistemas de computación de bajo rendimiento gráfico.

Por limitaciones del *framework* actual de *WebGL* no se permite el uso de texturas 3D, por lo que en este trabajo se simularon las texturas 3D mediante un atlas de textura 2D, realizando el muestreo del atlas mediante una función propia.

En cuanto a los resultados obtenidos, se puede apreciar a que mayor resolución del volumen y de la función de transferencia, la calidad del despliegue es superior, pero el tiempo requerido para generar un cuadro de imagen aumenta. Igualmente, si reducimos la longitud del paso en el algoritmo de ray casting, la calidad del despliegue aumenta, pero la rata de cuadros por segundo puede bajar drásticamente.

Como trabajos a futuro se propone rediseñar la aplicación con el estándar de *WebGL* 2.0, el cual aplica soporte completo de *OpenGL* ES 3.0 que permite el uso de texturas que no potencias de 2 y texturas 3D, y utiliza de GLSL 3.0. Igualmente se propone terminar el módulo de visualización con pre-integración, para obtener un despliegue de gran calidad, sin necesidad de requerir reducir el paso de *ray casting*, ni de asignarle un grosor al valor de superficie en la función de transferencia.

Cabe destacar que a esta aplicación es posible agregarle un módulo de visualización de datos médicos, puesto que solo habría que cambiar la fuente del volumen. En vez de generar un volumen a partir de una ecuación implícita, este podría cargarse de un archivo. bastaría agregar como mínimo un manejador de archivos tipo RAW, especificar el tamaño y el número de bits por muestra del volumen.

Finalmente, se desea que está aplicación esté disponible en la página Web del Centro de Computación Gráfica, bajo el dominio ccg.ciens.ucv.ve.

Referencias

Aho, A. V., Lam, M. S., & Sethi, R. (2007). *Compilers principles, techniques and tools.* Boston: Pearson Addyson Wesley.

Ara´ujo, B. R., & Jorge, J. A. (2004). Curvature Dependent Polygonization of Implicit Surfaces. *Computer Graphics and Image Processing, 2004. Proceedings. 17th Brazilian Symposium on Formal Methods* (págs. 266 - 273). Maceio - Alagoas: IEEE.

B, P., & D, B. (2007). *Visualization in Medicine.* Burlington: Morgan Kaufmann Publisher.

Bhalerao, A., Pfister, H., Halle, M., & Kikinis, R. (2000). Fast Re-Rendering of Volume and Surface Graphics by Depth, Color and Opacity Buffering. *Journal of Medical Image Analysis, Vol 4, Issue 3*, 235-251.

Bloomenthal, J. A. (1994). *Graphics Gems IV.* San Diego, CA, USA: Academic Press Professional, Inc.

Burden, R. y. (2009). *Análisis Numérico 7ma Edicion.* Youngstone: CENGAGE Learning.

Carmona., R. (2001). Triangulación de Ecuaciones Implícitas Combinando Cubos Marchantes con Algoritmos de Ceros de Funciones. *LI Convención Anual AsoVAC*.

Corrigan, A., & Dinh, H. Q. (2005). Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU. *International Workshop on Volume Graphics*, 187-195.

DaFeda. (2010). *Mathematics; ranting & learning*. Obtenido de DaFeda's Blog: http://dafeda.wordpress.com/2010/09/09/mullers-method-deriving-of-and-code/

Fleming, W. (1965). *Functions of several Variables.* New York: Addison-Wesley.

Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). Computer Graphics: Principles and Practice (2nd ed. in C). Addison-Wesley Publishing Company.

Foundation, T. j. (2015). *JQuery*. Obtenido de jQuery: The Write Less, Do More, JavaScript Library.: https://jquery.com/

Guzman, J. d. (2010). *Boost.Spirit*. Obtenido de boost-spirit: http://boost-spirit.com/home/

Hansen, C. D., & Johnson, C. R. (2004). *Visualization Handbook.* Salt Lake City: Elsevier.

Jatczyk, M. (15 de Junio de 2013). *Desing concept - Graphics art desing*. Obtenido de http://designconcept.webdev20.pl/: http://www.webdev20.pl/skins/default/js/demos/implicit-equation-3d-grapher/index.html

Jong, J. d. (2015). *mathjs*. Obtenido de math.js | an extensive math library for JavaScript and Node.js: http://mathjs.org/

K., E., & T., K. M. (2001). High Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *ACM Press*, 9-16.

Kajiya, J. T. (1986). THE RENDERING EQUATION. *SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (págs. 143-150). California Institute of Technology: SIGGRAPH.

Kaplan, W. (1949). *Advanced Calculus.* Michigan: Addison-Wesley 5th edition.

Klaus Engel, M. K. (2001). High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *Visualization and Interactive Systems Group, University of Stuttgart, Germany*.

Levoy, M. (1988). Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 29-37.

LLC, T. S. (2001). *C4 Game Engine* . Obtenido de Terathon: http://www.terathon.com

Lorensen, W. E., & Cline, H. E. (6 de 11 de 1987). Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics 21 (4)*, págs. 163–169.

Lorensen, W. E., & Cline, H. E. (1987). MARCHING CUBES: A HIGH RESOLUTION 3D SURFACE CONSTRUCTION ALGORITHM. *Computer Graphics*, 163-169.

M., W., J., N., & Davis, T. (1997). *OpenGL, Programming Guide.* Addison-Wesley Developers Press, 2da. Edición.

Mark Otto, j. (18 de August de 2011). *Bootstrap*. Obtenido de Bootstrap The world's most popular mobile-fisrt and responsive front-end framework: http://getbootstrap.com/

mrdoob. (23 de April de 2010). *threejs dat org*. Obtenido de three.js A Javascript 3D library: http://threejs.org/

Northwood, C. (2009). *Lexical and Syntax Analysis of Programming Languages*. Obtenido de http://www.pling.org.uk/: http://www.pling.org.uk/cs/lsa.html

Pauls. (2011). *Lamar University.* Obtenido de Paul's Online Math Notes: http://tutorial.math.lamar.edu/Classes/CalcI/NewtonsMethod.aspx

Philippe Lacroute, M. L. (July de 1994). Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Computer Graphics Procedings, Anual Conference Series, 1994* (págs. 451-458). Stanford: SIGGRAPH.

qiao, m. a. (2011). *threejs OrbitControl*. Obtenido de three.js A Javascript 3D library: http://threejs.org/

R. Carmona, O. R. (1997). Aspectos de Implementación para el algoritmo de Cubos Marchantes. *47o Convención Anual AsoVAC.*

R. Carmona, O. R. (1999). Cubos Marchantes: una implementación eficiente. In Proc. *XXV Conferencia Latinoamericana de Informática. Paraguay, La Asunción, Agosto*.

Schulz, S. (2003). Four Lectures on Differential-Algebraic Equations. *Institut für Mathematik*.

Skinner, J. (2015). *Sublime Text: The text editor you'ill fall in love*. Obtenido de Sublime Text: http://www.sublimetext.com/

Steffen, S. (2003). *Four Lectures on Differential-Algebraic Equations.* Berlin: Humboldt Universit¨at zu Berlin.

STEVENS, R. T. (1993). *Quick Reference to Computer Graphics Terms.* Academic Press, Inc.

Westover, L. A. (1991). *SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm.* North Carolina: The University of North Carolina.

Willian E. Lorensen, H. E. (27-31 de Julio de 1987). Marching Cubes: A high resolution 3D surface construction algorithm. *Computer Graphics, Volumen 21, Número 4*, págs. 163-1699.

WolframMathWorks. (2014). *WolframMathWorks*. Obtenido de WolframMathWorks: http://mathworld.wolfram.com/SecantMethod.html

Ziegler, J. (2004). *The LEDA Tutorial*. Obtenido de The LEDA Tutorial: http://www.leda-tutorial.org/en/discussion/ch06s03.html