UNIVERSIDAD NACIONAL DE SAN AGUSTÍN
FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

VISUALIZACIÓN INTERACTIVA DE SUPERFICIES IMPLÍCITAS INMERSAS EN EL ESPACIO $\mathbb{R}^3$

Tesis presentada por el Bachiller:

HENRY GIOVANNY GALLEGOS VELGARA

Para optar el Título Profesional de:

INGENIERO DE SISTEMAS

AREQUIPA - PERÚ

2018

Visualización interactiva de superficies implícitas inmersas en el espacio $\mathbb{R}^3$

Esta versión corresponde a la redacción
final de la tesis debidamente corregida
y defendida por *Henry Giovanny Gallegos Velgara*
el día 09 de enero del 2018

Comisión del jurado:

- Prof. Eveling Castro Gutierrez - EPIS - UNSA
- Prof. Alfredo Paz Valderrama - EPIS - UNSA
- Prof. Pedro Rodriguez Gonzalez - EPIS - UNSA

Agradecimientos

Agradezco a Dios porque siempre está a mi lado acompañando y guiándome en mi vida.

A Jakelinne, mi esposa que siempre me apoyó e incentivó en todos los proyectos que realicé, por su gran amor y por traer al mundo a mi preciosa hija Élienai que es mi mayor motivación para terminar mi tesis.

A mis padres y hermanos que estuvieron a mi lado animándome y apoyándome.

Resumen

El principal objetivo de este trabajo es presentar un nuevo método para la visualización de superficies implícitas de dimensión 2 inmersas en el espacio $\mathbb{R}^3$. Ese método consiste primeramente de un pré-procesamiento en CPU usando una estructura *Octree*, y usando la Aritmética Intervalar para encontrar las regiones de dominio donde está presente la superficie implícita. La información del *Octree* es enviada a la GPU usando la estructura de *Bounding Volume Sphere*, dicha información es procesada en la GPU para efectuar la visualización y para eso fue utilizado una generalización de la técnica *Ray Tracing*.

Palabras clave: Visualización en $\mathbb{R}^3$, Visualización de superficies implícitas; *Ray Tracing*; Aritmética Intervalar; *Octree*; GPU-GLSL.

Abstract

The main objective of this work is to present a new method for the visualization of implicit surfaces of dimension 2 immersed in space $\mathbb{R}^3$. This method consists first of a pre-processing in CPU using an Octree structure, and using Interval Arithmetic to find the domain regions where the implicit surface is present. The Octree information is sent to the GPU using the structure of Bounding Volume Sphere, this information is processed in the GPU to perform the visualization and for that a generalization of the Ray Tracing technique was used.

Keywords: Visualization in $\mathbb{R}^3$, Visualization of implicit surfaces; Ray Tracing; Arithmetic Interval; Octree; GPU-GLSL.

Índice general

Lista de Abreviaturas

AA	Aritmética Afín
AI	Aritmética de Intervalos
BVS	Esfera de volumen envolvente (*Bounding Volume Sphere*)
FPS	Imágenes por segundo (*frames per second*), es la velocidad (tasa) a la cual un dispositivo muestra imágenes llamadas cuadros o fotogramas.
GLSL	Lenguaje de sombreado de OpenGL (*OpenGL Shading Language*), es un lenguaje de alto nivel de sombreado con una sintaxis basada en el lenguaje de programación C.
GPU	Unidad de Procesamiento Gráfico (*Graphic Processing Unit*)
RGB	Rojo, verde y azul (*Red, Green, Blue*)

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En la actualidad el desarrollo de la computación gráfica se ha vuelto muy importante para diversas áreas, como el diseño de videojuegos realistas, simulación de ambientes para entrenamiento de personal o toma de decisiones por los jefes de una empresa. En esos sistemas una parte fundamental del desarrollo es la visualización del terreno o la superficie donde se presentan los objetos o la información a ser visualizada. [Gomes et al., 2009]

Tradicionalmente esas superficies fueron diseñadas usando mallas triangulares o mallas rectangulares uniformes, porque usan el *Pipeline* gráfico de la GPU, la cual está especialmente diseñada para procesar triángulos, de una forma muy rápida. [Shreiner et al., 2013]

Poder conseguir un sistema de visualización interactiva de superficies implícitas, nos permite formar las bases para un motor gráfico, ya sea para la visualización de terrenos en videojuegos, o la simulación de exploración de túneles, por citar algunos ejemplos. En la literatura actual existen muchos trabajos que presentan nuevos métodos, nuevas estructuras de datos que cada vez van acelerando el tiempo de renderización ya sea que se esté ejecutando en CPU o en GPU.

Con el desarrollo de las GPU's, cada vez más potentes, más rápidas y con un mayor número de núcleos, hacen que la investigación en el procesamiento paralelo sea muy atractiva. Se eligió usar el algoritmo de *Ray Tracing* justamente para aprovechar la capacidad de procesamiento en paralelo de las GPU.

Se eligió usar OpenGL 4, en vez de las otras arquitecturas que también utilizan el procesamiento en paralelo de las GPU's, porque se desea aprovechar el *Pipeline* programable del OpenGL, especialmente la etapa del procesamiento de fragmentos (*fragment shader*), porque es allí donde se va a utilizar el algoritmo de *Ray Tracing* y el procesamiento en paralelo de las GPU's.

1.1. Objetivos

1.1.1. Objetivo general

Presentar un método para la visualización interactiva de superficies implícitas inmersas en el espacio $\mathbb{R}^3$.

1.1.2. Objetivos específicos

1. Determinar un método para acelerar los cálculos de procesamiento en la CPU.
2. Determinar un método para enviar la información del CPU a la GPU.
3. Determinar un método para conseguir la renderización en la GPU con tazas de interactividad.
4. Determinar un conjunto de superficies implícitas y parámetros de configuración del sistema para realizar las pruebas,
5. Analizar los resultados de las pruebas realizadas para determinar los parámetros óptimos de configuración del método.

1.2. Contribución

En el presente trabajo se propone un método para conseguir la visualización de superficies implícitas con renderización en tiempo real, lo que va a permitir la interactividad con el usuario, y una buena calidad de la imagen renderizada, usando una combinación de un pre-procesamiento en CPU y un procesamiento en GPU.

Los productos finales de este trabajo son:

- La descripción del método, paso por paso en forma resumida y extendida.
- Un prototipo de un sistema que usa el método propuesto.
- Las pruebas realizadas, los resultados obtenidos y conclusiones.

1.3. Organización del trabajo

Este trabajo está organizado de la siguiente manera:
En el Capítulo 2 se presenta los trabajos previos relacionados, luego en el Capítulo 3 se brinda

un marco teórico fundamental e introductorio sobre superficies implícitas, *Ray Tracing*, *Pipeline* programable de OpenGL, Aritmética Intervalar, estructura de datos espaciales, modelo de iluminación, luego en el Capítulo 4 se explica el método propuesto compuesto de dos etapas, la primera etapa que es un pré-procesamiento en CPU y la segunda etapa corresponde al procesamiento en GPU, en el Capítulo 5 se presenta los resultados obtenidos y finalmente en el Capítulo 6 concluye el trabajo describiendo las conclusiones y proponiendo trabajos futuros.

Capítulo 2

Estado del Arte

El interés en la visualización de objetos en el espacio $\mathbb{R}^3$ es muy antiguo, y han sido propuestos diferentes métodos para lograr la que la visualización sea rápida, exacta y con buena calidad de la imagen. A continuación se resume los principales trabajos encontrados, los que fueron agrupados por los métodos que usan, método de extracción de mallas, métodos basados en puntos y métodos de *Ray Tracing*.

2.1. Métodos de extracción de mallas

Dada la popularidad de la rasterización de la GPU, el método más común ha sido la extracción de una malla, usando la técnica de *Marching Cubes* [Wyvill et al., 1986] lo que hace esta técnica es subdividir el espacio en cubos pequeños y considerando unos cubos plantillas, van aproximando la superficie implícita, la desventaja es que omite detalles pequeños de la superficie implícita. Otro método usado es la técnica de poligonización Bloomenthal [Bloomenthal, 1994] que puede generar mallas interactivamente.

También existen técnicas que mejoran los resultados de la calidad de la imagen, pero con el costo de menor interactividad como el presentado en [Paiva et al., 2006].

Un estudio resumiendo muchas de las técnicas usando este método, puede ser encontrado en [de Araújo et al., 2015], allí se centra en la clasificación y comparación de los métodos existentes, para identificar las mejores estrategias según los requerimientos específicos como velocidad, exactitud, calidad o estilo.

2.2. Métodos basados en puntos

Otro método popular es usar una aproximación a la superficie implícita usando puntos, cuando la geometría discretizada de la superficie implícita es suficientemente densa, sería más eficiente aproximar la superficie con una geometría no poligonal, en esta geometría es común usar puntos. La renderización basada en puntos fue concebida por Levoy y Whitted [Levoy and Whitted, 1985] y probada con un gran conjunto de puntos por Rusinkiewics y Levoy [Rusinkiewicz and Levoy, 2000]. El proceso implica muestrear el espacio en puntos y aproximar esos puntos a la superficie y sombrear esos puntos como discos planos pequeños, ya sea visualizándolos en la CPU o en la GPU.

2.3. Métodos de Ray Tracing

Este método será empleado en el desarrollo del presente trabajo, a continuación se describe los trabajos similares encontrados. Este método es muy popular, la idea es que a partir de un pixel donde se va a renderizar la imagen, se lanza un rayo hacia la superficie implícita y se calcula el punto de intersección. Un método para calcular esta intersección es el muestreo de puntos usando la regla de signos de Descartes para aislar la raíz.

En el trabajo de [Hart, 1996] se propone un método robusto para *Ray Tracing* algebraico, para encontrar la raíz define funciones de distancia con signo desde un punto arbitrario a la superficie. En el trabajo de [Romeiro et al., 2006] se propone una técnica hibrida GPU/CPU para lanzar rayos a través de un árbol de superficies implícitas de geometría solida constructiva. En el artículo de [Fryazinov and Pasko, 2008] emplea métodos de regla de signos de intervalos en *Ray Tracing* de superficies implícitas generalizadas en la GPU.

En el trabajo de [Mitchell, 1990] propone un *Ray Tracing* implícito usando Aritmética de Intervalos (AI) con bisección recursiva para aislar intervalos de rayos, junto con la bisección estándar como un método de refinamiento.

En el artículo de [de Cusatis Junior et al., 1999] usaron la teoría de la Aritmética Afín (AA) estándar junto con la bisección recursiva para acelerar los cálculos de encontrar las raíces de la superficie implícita. En el trabajo [Knoll et al., 2007] para encontrar las raíces, implementó un algoritmo de bisección de intervalos interactivo, para renderización de formas implícitas arbitrarias en la CPU.

En la tesis de Knoll [Knoll, 2009] usa las teorías de Aritmética Intervalar (AI) y Aritmética Afín (AA) para acelerar el cálculo de intersección en *Ray Tracing*, también describen la implementación de AI y AA en la GPU. En la tesis de [Sigg, 2006] se describe diferentes enfoques para la representación de superficies implícitas y métodos de renderización de

superficies de esas representaciones altamente paralelizada.

En el artículo de [Liktor, 2008] examina métodos de renderización de superficies implícitas con *Ray Tracing* en GPU, el principal problema que aborda es como encontrar eficientemente el punto de intersección del rayo con la superficie, para ello usa un algoritmo de seguimiento de esfera, también aborda asuntos relacionados a la coloración y textura.

En el artículo de [Wald et al., 2005] implementan un *Ray Tracing* interactivo usando la estructura espacial *KD-Tree*, basándose en el *Ray Tracing* de escenas poligonales para mejorar en el *Ray Tracing* de una superficie implícita, prueban su método con superficies implícitas altamente complejas, también lo prueban con renderización interactiva hibrida de escenas poligonales y superficies implícitas, incluyen efectos de sombreado e iluminación global.

Capítulo 3

Conceptos Fundamentales

En este Capítulo van a ser descritos los conceptos teóricos básicos que permitirá tener un mejor entendimiento del método propuesto. Se presenta los conceptos de superficies implícitas, de *Ray Tracing*, del *Pipeline* programable de OpenGL, de Aritmética Intervalar, las estructura de datos usadas, y el modelo de iluminación.

3.1. Superficies Implícitas

Superficies implícitas son usadas en computación gráfica para el modelamiento de objetos geométricos, son útiles para representar deformaciones y *blending*.

En matemática se define una superficie S en forma implícita en $\mathbb{R}^n$ como el conjunto de soluciones de la función $F : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$ igual a cero: $F(p) = 0, p \in \mathbb{R}^n$, el dominio de la función F pertenece a $\mathbb{R}^n$ y su imagen a $\mathbb{R}$ [Knoll, 2009].

En la Figura 3.1 se puede ver un ejemplo de superficie implícita dada por la siguiente función: $x^4 + y^4 + z^4 - 1 = 0$

Este trabajo estudia la visualización de superficies implícitas de dimensión 2 inmersas en

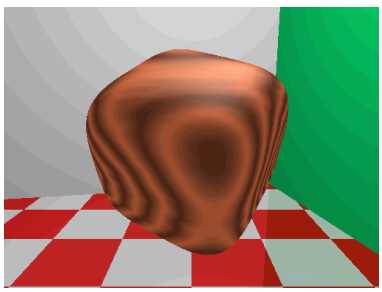

Figura 3.1: *Ejemplo de superficie implícita. Fuente: [Cordero, 2017].*

el espacio $\mathbb{R}^3$, determinadas por una única función implícita con tres variables ($n = 3$):

$$F(x, y, z) = 0. \tag{3.1}$$

Una función implícita F divide el espacio $\mathbb{R}^3$ en tres regiones: la primera corresponde al conjunto de puntos en que F tiene signo negativo, la segunda corresponde al conjunto de puntos en que F tiene signo positivo, y la tercera corresponde al conjunto de puntos donde F es igual a cero. En el primer caso, se dice que un punto que pertenece a esa región está dentro de la superficie implícita, en el segundo caso, se dice que el punto está fuera de la superficie implícita, y finalmente, en el tercer caso, se dice que el punto está sobre la superficie implícita.

Mas adelante, para calcular la iluminación en un punto, va se preciso calcular la normal de la superficie implícita en cada punto. La normal en un punto $\mathbf{p} = (x, y, z) \in F^{-1}(0) \subset \mathbb{R}^3$ es definida por la gradiente de la función:

$$\mathbf{n} = \nabla F(\mathbf{p}) = (\frac{\partial F}{\partial x}(\mathbf{p}), \frac{\partial F}{\partial y}(\mathbf{p}), \frac{\partial F}{\partial z}(\mathbf{p})) \tag{3.2}$$

3.2. Ray Tracing

La técnica de *Ray Tracing* está basada en la técnica de *Ray Casting*, la cual fue primero propuesta por [Appel, 1968], Ray Casting es una técnica de computación gráfica que crea imágenes lanzando rayos a partir de una cámara y guardando en la pantalla la intersección de la trayectoria de los rayos con los objetos de la escena. La técnica de *Ray Tracing* fue primero propuesta por [Whitted, 1980] y la principal diferencia entre *Ray Tracing* y *Ray Casting* es que la primera permite lanzar rayos secundarios para calcular sombras, reflexiones y refracciones.

La Figura 3.2, presenta el proceso de *Ray Tracing* en $3D$ con una cámara, una ventana con pixeles, una fuente de luz y dos rayos intersectando una esfera y un toroide.

Muchos trabajos en visualización de superficies implícitas están relacionados a las técnicas de *Ray Tracing*, en los cuales son utilizados rayos secundarios para obtener una iluminación realista y modelos de sombreado avanzados como la iluminación global usado en el trabajo de [Kajiya, 1986]. Mas recientemente, la comunidad viene concentrando esfuerzos en convertir el *Ray Tracing* más interactivo, a través de técnicas de paralelización y optimizaciones como en [Singh, 2013].

Este trabajo utiliza la técnica de *Ray Tracing*, usando pocos rayos secundarios, para no sobrecargar los cálculos, ya que será hecho uso de la programación en GPU. En las

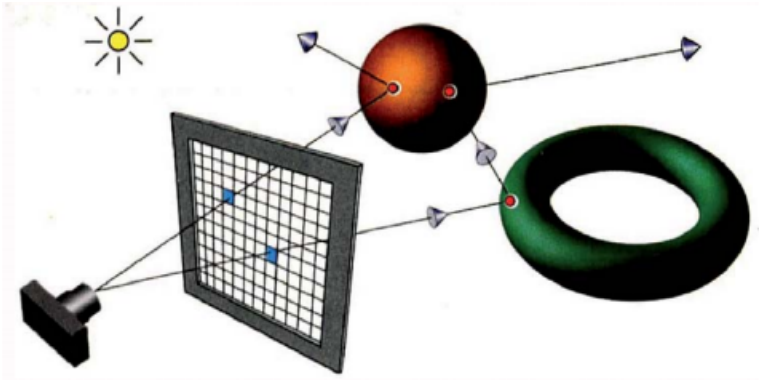

Figura 3.2: *El proceso de Ray Tracing. Fuente: [Suffern, 2007].*

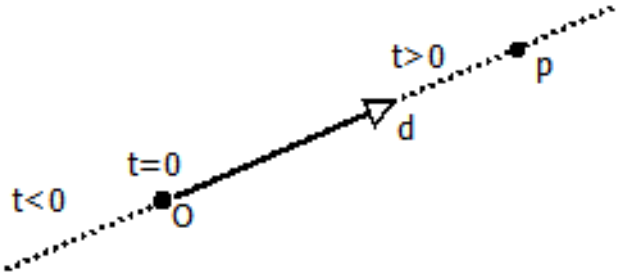

Figura 3.3: *Definición del rayo.*

siguientes subsecciones serán descritas los conceptos de Rayo (subsección 3.2.1), intersección del rayo con una esfera (subsección 3.2.2), y la intersección del rayo con una función implícita (subsección 3.2.3) en $\mathbb{R}^3$.

3.2.1. Rayo

Un rayo es una linea infinita que está definida por un punto $\mathbf{o}$, llamado origen, y un vector $\mathbf{d}$, llamado dirección. Un rayo $\mathbf{r} : \mathbb{R} \to \mathbb{R}^3$ es una función parametrizada por un número real t. Cuando $\mathbf{r}(0)$ corresponde al punto de origen del rayo, un punto arbitrario $\mathbf{r}(t)$ en el rayo es expresado por [Suffern, 2007]:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{3.3}$$

La Figura 3.3 ilustra la definición de un rayo.

3.2.2. Intersección del rayo con una esfera

Una esfera es definida por un conjunto de puntos $\mathbf{p} = (x, y, z) \in \mathbb{R}^3$ que distan R de un punto $\mathbf{c}$, donde R es el radio y $\mathbf{c} = (c_x, c_y, c_z)$ es el centro de la esfera. Ese objeto puede ser

representado a través de una ecuación implícita que es dada por:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0 \tag{3.4}$$

donde el operador $\cdot$ corresponde al producto interno usual en el $\mathbb{R}^3$. Esa ecuación puede ser re-escrita en la siguiente forma:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - R^2 = 0 \tag{3.5}$$

Para interceptar un rayo con una esfera, se deve substituir la Ecuación 3.3 en 3.4 para obtener:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0 \tag{3.6}$$

Para calcular las dos raices, se expande la Ecuación 3.6 de tal modo que se obtiene:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + [2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}]t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2 = 0 \tag{3.7}$$

La Ecuación 3.7 es una ecuación cuadrática en t, que puede ser escrita como:

$$at^2 + bt + c = 0, \tag{3.8}$$

donde

$$\begin{aligned} a &= \mathbf{d} \cdot \mathbf{d}, \\ b &= 2(\mathbf{o} - \mathbf{c}) \cdot d, \\ c &= (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 \end{aligned} \tag{3.9}$$

La solución de la Ecuación 3.8 es dada por la expresión:

$$t = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

Las ecuaciones cuadráticas pueden tener ninguna, una o dos raíces reales, dependiendo del valor del discriminante $b^2 - 4ac$.

En este trabajo se considera los casos que poseen dos raíces reales, y serán descartadas las otras dos situaciones.

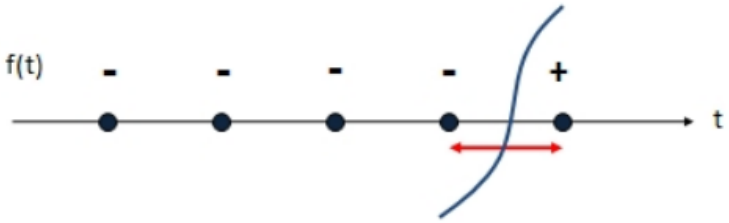

Figura 3.4: *Encontrando la raiz con muestreo de puntos con bisección.*

3.2.3. Intersección de un rayo con una función implícita

La intersección entre un rayo (Subsección 3.2.1) y una función implícita $F(x, y, z) = 0$, se obtiene substituyendo los parámetros de F por la Ecuación del rayo (3.3), con el cual se obtiene:

$$g(t) = F(o_x + td_x, o_y + td_y, o_z + td_z, o_w + td_w) \tag{3.10}$$

Calcular la intersección con un rayo significa encontrar las raíces de la función g dada por la Ecuación 3.10.

Existen muchas técnicas para encontrar esas raíces como lo explica en el trabajo de [Flórez Díaz, 2008]. El enfoque usado en este trabajo es del muestreo de puntos con bisección, un ejemplo está en la Figura 3.4.

Considerando un dominio t_{min} y t_{max} para los valores del parámetro $\mathbf{t}$ del rayo, se debe muestrear ese intervalo, en una cantidad finita de puntos, y para cada punto verificar el signo de la función $\mathbf{g}$. Si hubiera dos puntos con signos diferentes, entonces eso significa que en ese intervalo hay por lo menos una intersección, por el Teorema del Valor Intermedio, ya que $\mathbf{g}$ así como F son funciones contínuas (ver Figura 3.4).

3.3. *Pipeline* programable de OpenGL

En esta sección será presentado los conceptos básicos de la programación en GPU usando el *Pipeline* programable de OpenGL. Una explicación más detallada puede ser encontrada en los libros [Shreiner et al., 2013], [Engel et al., 2006] y en [Akenine-Möller et al., 2008].

La mayoría de computadores modernos están equipados con un procesador para gráficos 3D acelerados por *hardware*. Las unidades de procesamiento de gráficos (GPU, por sus siglas en ingles *graphics processing units*), están altamente optimizadas para el procesamiento de datos en paralelo. Las GPUs han substituido el tradicional *Pipeline* de función fija por un *Pipeline* programable, el cual permite al programador escribir pequeños programas para que sean ejecutados muy rápidos y eficientemente.

A continuación será descrito el *Pipeline* y sus estados programables.

3.3.1. El *Pipeline* programable

Para visualizar una escena virtual, primero la escena debe ser descompuesta en polígonos planos, generalmente triángulos. Al enviar esos polígonos a la GPU, la GPU genera imágenes *raster* de forma muy rápida y eficientemente. Ese proceso de conversión de un conjunto de primitivas poligonales en una imagen *raster* es llamado como *display transversal*.

Todos los procesadores gráficos 3D implementan el *display transversal* como un *Pipeline* consistiendo de una secuencia fija de estados de procesamiento. La Figura 3.5 presenta el orden de operaciones de un procesador gráfico moderno con el uso de OpenGL con versiones a partir de 3.2. Las cajas azules son los estados programables, las cajas blancas representan los procesamientos internos de la GPU, y los números en paréntesis indican cual versión de OpenGL es requerida. En las siguientes subsecciones cada estado será descrito separadamente.

3.3.2. Vertex Shader

El *vertex shader* opera en los vértices de forma individual, significa: un vértice cada vez. el *vertex shader* no tiene conocimiento de los otros vértices que forman la primitiva geométrica. El *vertex shader* calcula transformaciones lineales de los vértices de entrada, tales como escala, rotación y traslación en el espacio tridimensional. Este paso comprende la transformación de los vértices desde las coordenadas del modelo local dentro del espacio del mundo (matriz del modelo), luego dentro del espacio de la cámara (matriz de visión), hasta el espacio de la ventana (matriz de proyección).

3.3.3. Tessellation

Tessellation es un estado que recibe *patches* como entradas y genera nuevas primitivas geométricas que pueden ser puntos, lineas o triángulos. Un *patch* es dado por un cadena de vértices y sus atributos son calculados por el *vertex shader*.

3.3.4. Geometry Shader

Este estado es opcional, si está activo el *geometry shader* entonces recibe como entrada las primitivas geométricas construidas en el estado anterior. El *geometry shader* tiene acceso a todos los vértices de la primitiva en la cual se está trabajando, y si fue especificado, puede

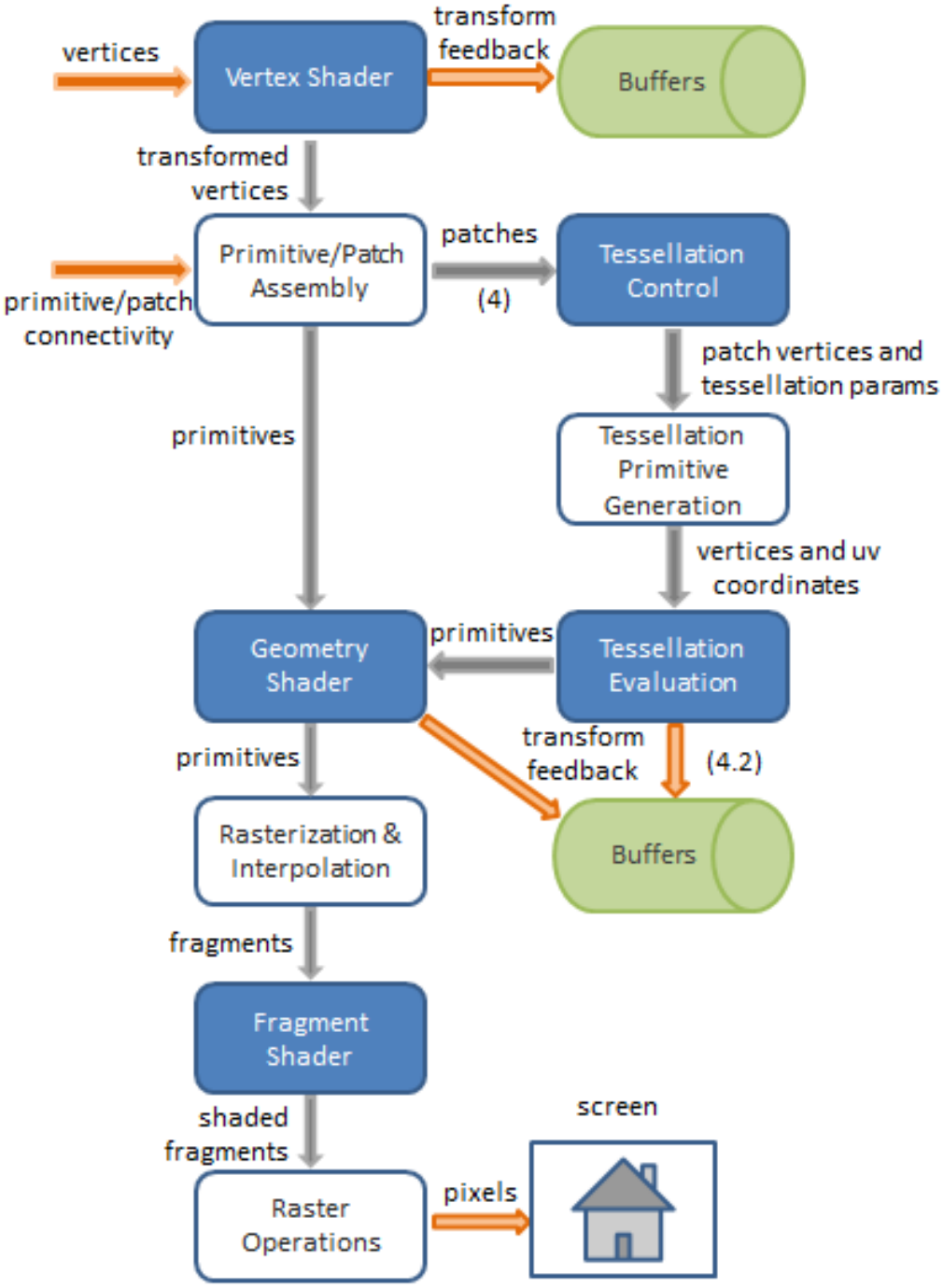

Figura 3.5: *El Pipeline gráfico programable. Fuente: [Lighthouse3d, 2014].*

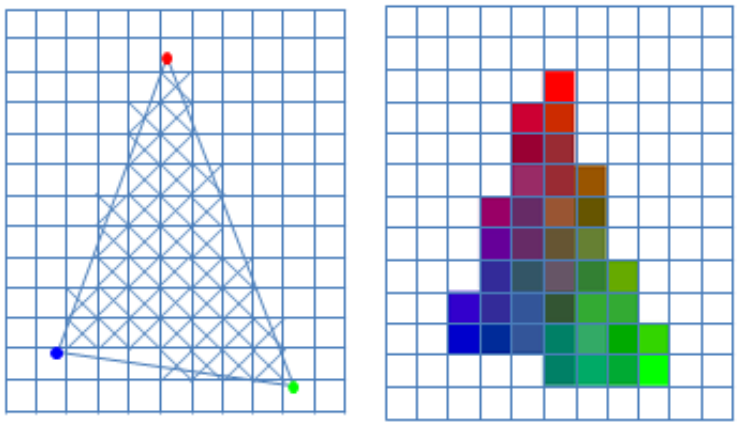

Figura 3.6: *Un ejemplo del estado de rasterización (izquierda) e interpolación (derecha) del Pipeline gráfico. Fuente: [Lighthouse3d, 2014].*

también tener acceso a la información de adyacencia. Los tipos de primitivas geométricas de entrada y de salida deben ser declarados, la salida puede tener cero o más primitivas geométricas.

3.3.5. Rasterization and Interpolation

Este estado no es programable, es un componente fijo del *Pipeline* gráfico, pero es importante describir lo que sucede aquí, para entender mejor el *Pipeline*.

Rasterization

Es el proceso de determinar el conjunto de pixeles en la imagen final, que son parte de la primitiva. Para cada pixel que tiene su centro dentro de los limites del triángulo serán adicionados al conjunto de pixeles para procesamiento futuro. En la Figura 3.6 en la parte izquierda, los puntos con colores representan las posiciones de los vértices en el espacio de la pantalla.

Interpolation

El siguiente paso es calcular los atributos para cada pixel basado en los atributos y en la distancia del pixel a cada vértice en la posición de la pantalla. En la Figura 3.6 en la parte derecha, se puede observar los colores de los pixeles interpolados.

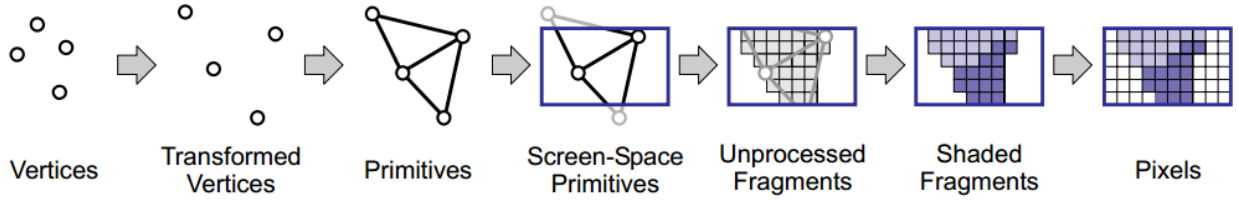

Figura 3.7: *Una descripción visual de los estados del Pipeline gráfico, desde los vértices hasta los pixeles. Fuente: [Engel et al., 2006].*

3.3.6. Fragment Shader

En este estado cada fragmento tiene sus propios atributos interpolados del estado anterior, no es posible acceder atributos de fragmentos vecinos. El procesador de fragmento es capaz de efectuar varias texturas y usar operaciones de filtrado para cada fragmento. El programa calcula el color final del fragmento a partir de los atributos del vértice interpolados, las muestras de texturas filtrados y el código implementado.

Un *fragment shader* solo puede tener una variable de salida, la cual debe ser el color del fragmento. Algunas variables propias de cada fragmento son:

- gl_FragCoord: contiene las coordenadas del fragmento (xf, yf, zf, wf), donde (xf, yf) es la posición del pixel en la pantalla, zf es la profundidad, y wf es $1/wc$, donde wc es el componente del fragmento en el espacio de *clipping*.
- gl_FrontFacing: dice la orientación de la primitiva que origino el pixel.
- gl_FragDepth: contiene la profundidad del pixel, si está activo el buffer de profundidad.

En la Figura 3.7 se puede observar una descripción visual de los estados anteriormente descritos, desde los vértices hasta los pixeles.

3.4. Aritmética Intervalar

El objetivo de esta sección es definir las notaciones y los conceptos básicos de *Aritmética Intervalar* (AI). Los siguientes conceptos fueron extraídos del trabajo de [Moore, 1966], para más detalles sobre la teoría de AI y demostraciones puede consultar dicho trabajo.

Moore en [Moore, 1966] definió el conjunto de todos los intervalos reales compactos sea denotado por $\mathbb{IR}$. Un elemento $[x]$ en $\mathbb{IR}$ corresponde a un intervalo $[\underline{x}, \overline{x}]$, donde $\underline{x}$ y $\overline{x}$ son dos números reales tal que $\underline{x} \leq \overline{x}$.

Para un intervalo $[x] = [\underline{x}, \overline{x}] \in \mathbb{IR}$, sus dos puntos extremos $\underline{x}$ y $\overline{x}$ son llamados, respectivamente, *ínfimo* y *supremo* de $[x]$ y son denotados por ínf$([x])$ y sup$([x])$.

[Moore, 1966] demostró que la operación aritmética binaria básica $\circ \in \{+, -, \times, \div\}$ para números reales puede ser extendida para dos intervalos $[x]$ y $[y] \in \mathbb{IR}$ en la siguiente manera: $[x] \circ [y] := \{x \circ y | x \in [x] \text{ and } y \in [y]\}$. El operador aritmético $\div$ solo puede ser aplicado cuando $0 \notin [y]$. El resultado de cualquier operación aritmética intervalar es un intervalo, y usando sus propiedades de monotonicidad, pueden ser expresados en términos de los puntos extremos de los operandos, por ejemplo, $[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. Todas estas operaciones aritméticas binarias $\circ \in \{+, -, \times, \div\}$ tienen *inclusión isotónica*, que significa: $[z] \subseteq [x]$ and $[w] \subseteq [y] \Rightarrow [z] \circ [w] \subseteq [x] \circ [y]$.

El *dominio* de una función real f es denotado por $Dom(f)$. La *imagen* de una función continua real f sobre todos los puntos x en el intervalo real compacto $[x] \subseteq Dom(f)$ es definido como siendo el intervalo $Im(f, [x])$ tal que $Im(f, [x]) := [\text{mín}\{f(x) | x \in [x]\}, \text{máx}\{f(x) | x \in [x]\}]$.

La *extensión intervalar de una función real elemental* $\phi \in \{\exp, \ln, \cos, \sin, \tan, \cos^{-1}, \sin^{-1}, \tan^{-1}, \cosh, \sinh, \tanh, \cosh^{-1}, \sinh^{-1}, \tanh^{-1}\}$ sobre un intervalo dado $[x] \subseteq Dom(\phi)$ es definida como siendo la imagen de ϕ sobre $[x]$, por ejemplo, $\phi([x]) := Im(\phi, [x])$. Las funciones potencia y racionales son definidas de manera semejante.

Una función intervalar γ se dice que tiene *inclusión isotónica* cuando para todos los pares de intervalos $[x]$ y $[y]$ contenido en su dominio, la proposición $[x] \subseteq [y] \Rightarrow \gamma([x]) \subseteq \gamma([y])$ es siempre verdadera. Por definición, la extensión intervalar de todas las funciones elementales citadas anteriormente tienen esa propiedad importante. Ya que cada función real elemental es continua y su extensión intervalar tienen inclusión isotónica, el intervalo $\phi([x])$ puede también ser expresado en términos de puntos de limite de $[x]$, por ejemplo: $\exp([x]) := [\exp(\text{ínf}([x])), \exp(\sup([x]))]$.

Sea f una función escalar de una variable real x definida por una expresión que contiene apenas las operaciones aritméticas y funciones elementales. La *extensión intervalar* de f, denotada por $f_{[]}$, sobre un intervalo real compacto $[x] \subseteq Dom(f)$ es definida como siendo la función intervalar construida a partir de la substitución de cada ocurrencia de la variable real x en la expresión de f por su correspondiente variable intervalar $[x]$. El intervalo $f_{[]}([x])$ es entonces obtenido por el uso de las correspondientes operaciones aritméticas intervalares y funciones intervalares elementales sobre la evaluación de su expresión. Cuando el intervalo $[x]$ es un único punto ($[x] = [a, a]$), la extensión intervalar $f_{[]}$ satisface $f_{[]}([a, a]) = f(a)$. Ya que todas las operaciones de intervalo y funciones elementales tienen inclusión isotónica, la extensión intervalar de una función real también tiene inclusión isotónica. [Moore, 1966].

El siguiente teorema es la principal herramienta utilizada en este trabajo. El teorema enlaza la AI a la verdadera imagen de la función de estudio.
Teorema. Sea f una función continua real de una variable y $f_{[]}$ su extensión intervalar. Si $[x] \subseteq Dom(f)$, entonces $Im(f,[x]) \subseteq f_{[]}([x])$. [Moore, 1966].

Cuando un punto $y \notin f_{[]}([x])$, este teorema garantiza que el punto y no está en el conjunto $Im(f,[x])$. Desafortunadamente, el intervalo $f_{[]}([x])$ es en general, una sobrestimación de la imagen de f sobre el intervalo $[x]$. Ese Teorema vale también para las funciones reales continuas en n variables, que son los principales objetos de estudio de este trabajo.

3.5. Estructura de Datos Espaciales

En este trabajo se usará la estructura de datos *Octree*, para la subdivisión espacial de la superficie implícita, los siguientes conceptos fueron extraídos del libro [Samet, 1990], para más detalle se puede consultar dicho libro.

El *Octree* está basado en la subdivisión sucesiva de un objeto en octantes, si el arreglo no consiste enteramente de 1s o enteramente de 0s, es subdividido en octantes, suboctantes, así hasta que los cubos que son obtenidos consistan de 1s o de 0s, debe existir un criterio para detener la subdivisión, como un limite de subdivisiones.

Este proceso de subdivisión es representado por un árbol de grado 8 en cual el nodo raíz representa la objeto entero y los nodos hojas corresponden a estos cubos del arreglo para los cuales no es necesaria mas subdivisiones. La Figura 3.8(a) es un ejemplo de un objeto tridimensional, para quien su descomposición en bloques es dado en la Figura 3.8(b) y su árbol de representación es dado en la Figura 3.8(c). En la Figura 3.9 se observa los bloques *Octree* de un modelo 3D.

También se usará la estructura BVS (esfera de volumen envolvente, *Bounding Volume Sphere*) la que permite definir el área envolvente de un objeto con una esfera, o con una jerarquía de esferas, más información se puede encontrar en [Arvo and Kirk, 1989] y en [Franquesa Niubó and Rodríguez González, 2005].

3.6. Modelo de iluminación

En esta sección es explicado el modelo de iluminación que fue implementado. Fue escogido el modelo de iluminación de Phong, porque es el modelo de iluminación más usado en

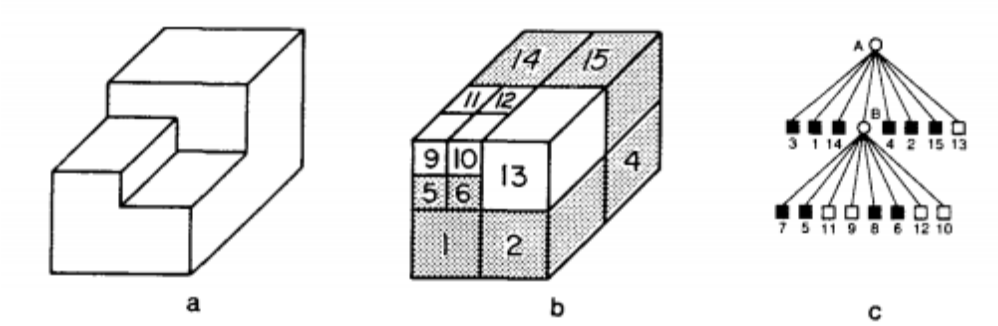

Figura 3.8: *(a) Ejemplo de objeto tridimensional; (b) La descomposición en bloques del Octree; (c) El árbol de representación. Fuente: [Samet, 1990].*

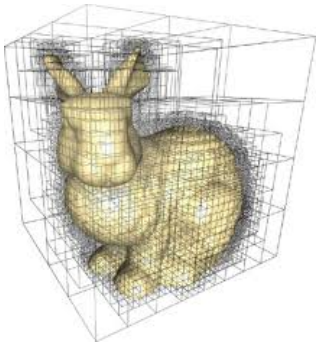

Figura 3.9: *Visualización de la estructura de datos Octree de un modelo 3D. Fuente: [Pharr and Fernando, 2005].*

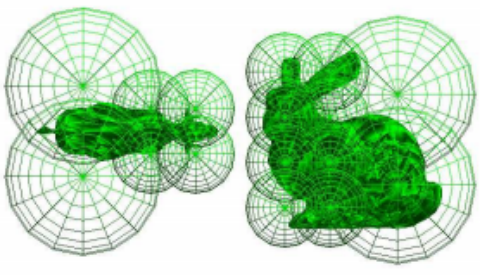

Figura 3.10: *Visualización de la estructura de datos Bounding Volume Sphere de un modelo 3D. Fuente: [Franquesa Niubó and Rodríguez González, 2005].*

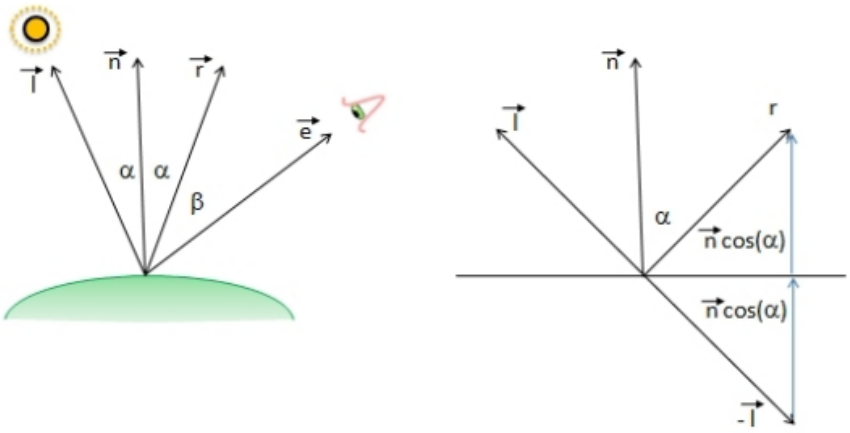

Figura 3.11: *Geometría del modelo de iluminación de Phong, a la izquierda los vectores que intervienen y a la derecha el cálculo del rayo de reflexión de la luz [Flórez Díaz, 2008].*

computación gráfica, para más detalle se puede consultar [Phong, 1975].

La iluminación es realizada en el espacio $\mathbb{R}^3$, considerando las posiciones de una o varias fuentes de luz. La Figura 3.11 presenta los vectores implicados en el proceso de iluminación:
$\vec{e}$: vector desde el punto de vista, en este trabajo este es el vector dirección del rayo desde el pixel,
$\vec{l}$: vector para la luz,
$\vec{n}$: normal de la superficie,
$\vec{r}$: rayo de reflexión.

El modelo de iluminación de Phong calcula la luz reflejada por un objeto como una combinación de tres diferentes términos: ambiente, difuso y especular [Engel et al., 2006], a través de la siguiente ecuación:

$$I_{Phong} = I_{ambiente} + I_{difuso} + I_{especular}$$

Los términos son escritos como valores RGB.

El termino ambiente es modelado como una constante de luz global multiplicado por el color ambiente y el coeficiente ambiente del material:

$$I_{ambiente} = k_a \; C_a \; I_a,$$

donde:
I_a representa el color y la intensidad de la luz ambiente global,
C_a representa el color ambiente del material,
k_a representa el coeficiente de la luz ambiente, que es una constante entre 0 y 1. Esa constante

controla cuanto de luz ambiente que llega a la superficie es reflejada.

El término difuso corresponde a la reflexión *Lambertiana*, donde la luz es reflejada igualmente en todas las direcciones:

$$I_{difuso} = k_d \; C_d \; I_d \; max(0, n \cdot l),$$

donde:
I_d representa el color y la intensidad emitida por la fuente de luz,
C_d representa el color difuso del material,
k_d representa el coeficiente difuso, constante entre 0 y 1.
Según el término difuso, el color de un punto en la superficie es proporcional al coseno del angulo (producto punto) entre la dirección de la luz $\vec{l}$ y la normal $\vec{n}$ en el punto. La función max es usada para los casos en que el producto punto es negativo, en esos puntos la luz no llega y quedan en la sombra.

El término especular modela la reflexión de superficies brillantes y depende del vector de visión $\vec{e}$:

$$I_{especular} = k_s \; C_s \; I_s \; max(0, r \cdot e)^n,$$

donde:
I_s representa la intensidad emitida por la fuente de luz,
C_s representa el color especular del material,
k_s representa el coeficiente especular, que determina la cantidad de reflexión especular del material,
n representa el exponente especular, llamado brillantes de la superficie, y es usado para controlar el tamaño de la iluminación resultante.

En Figura 3.12, se puede observar un ejemplo de la iluminación con el modelo de Phong.

3.6.1. Tipos de fuentes de luz

En computación gráfica existen muchas variedades de tipos de fuentes de luz, cada tipo crea un efecto visual único, y adiciona una cantidad de complejidad computacional a la escena. Mas informaciones pueden ser obtenidas en [Engel et al., 2006].

En este trabajo se ha utilizado las fuentes de luz puntual y direccional, por ser de menor carga computacional.

- *Fuentes de luz puntual* emiten luz a partir de un solo punto en el espacio, igualmente

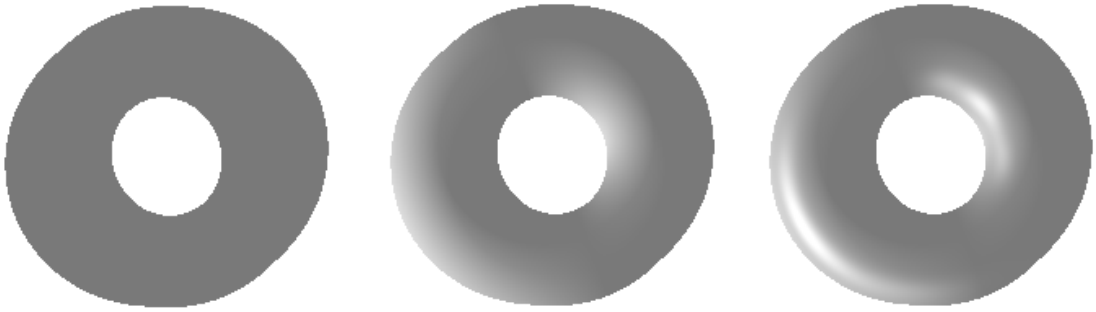

Figura 3.12: *Ejemplo de iluminación con el modelo de Phong. Izquierda, iluminando con el término ambiente. Centro, adicionando el término difuso. Derecha, imagen final con el término especular.*

para todas las direcciones.

- *Fuentes de luz direccional* son fuentes de luz puntual en el infinito. Son solamente descritos por la dirección de la luz, y todos los rayos emitidos son paralelos uno a otro.

Capítulo 4

Método Propuesto

4.1. Introducción

En este Capítulo en la primera sección será explicado una descripción general del método, y en las siguientes secciones serán explicados los pasos de la primera y segunda fase del método, que corresponden al pre-procesamiento hecho en CPU y al procesamiento hecho en GPU, respectivamente.

4.2. Descripción general del método

Considere $F : \Omega \subseteq \mathbb{R}^3 \to \mathbb{R}$ una función, donde 0 es un valor regular de F. El método propuesto realiza la visualización en tiempo real para la superficie implícita $\mathcal{S} = F^{-1}(0)$.

Considere un dominio $\Omega = [\mathbf{h}] \subseteq \mathbb{R}^3$ que es una caja dada por $[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$. La primera fase del método consiste en un algoritmo que subdivide $[\mathbf{h}]$ usando una estructura de datos *Octree*. Ese algoritmo realiza las subdivisiones del dominio de acuerdo con criterios basados en la Aritmética Intervalar, para descartar las cajas que no interceptan la superficie. Para cada caja de esa subdivisión que la superficie intercepta son calculados sus puntos centrales y el radio de la esfera envolvente. Esa estructura de datos es enviada para la GPU juntamente con la matriz de rotación para realizar la segunda fase del método.

En la GPU son usadas las informaciones de las esferas asociadas a las cajas de la subdivisión en conjunto con la matriz de rotación para ejecutar la técnica de *Ray Tracing* para visualizar la superficie implícita dada por la función F. Para eso, son calculadas los colores de cada pixel considerando la fuente de luz, dando destaque a los puntos que pertenecen a la silueta o al borde del dominio.

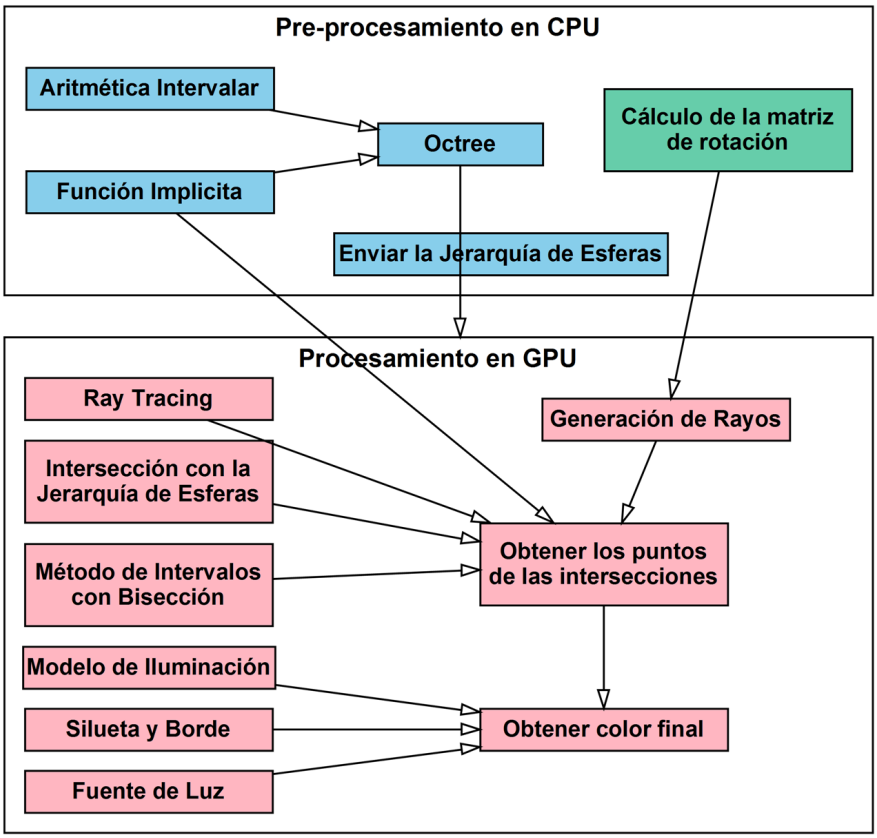

Figura 4.1: *Diagrama de la descripción general del método propuesto, las cajas azules son procesadas en CPU una sola vez, la caja verde es procesada en CPU en cada frame, y las cajas rojas son procesadas en GPU en cada frame.*

La Figura 4.1 presenta un diagrama conteniendo las etapas del método propuesto para la visualización de la superficie. En la próxima sección será descrito los detalles de cada etapa del pre-procesamiento en CPU, en la sección (4.4) es descrito el procesamiento en GPU.

4.3. Pre-procesamiento en CPU

En esta sección será explicado la primera fase del método propuesto, que corresponde al pré-procesamiento en CPU.

4.3.1. Construcción del *Octree*

El algoritmo de subdivisión es iniciado considerando el dominio [**h**] que representa la caja de la raíz del *Octree*. Se verifica si la caja actual debe ser subdividida, usando criterios basados en la Aritmética Intervalar. La subdivisión de un nodo de la *Octree* genera 8 nuevas cajas, y el proceso es repetido recursivamente para cada una de las cajas. Cada caja de la subdivisión

es marcada si contiene o no la superficie, y esa marcado sirve para procesar posteriormente solo las cajas que contienen la superficie (subsección 4.3.2).

Uso de la Aritmética Intervalar

Considere una función $F : \Omega \subseteq \mathbb{R}^3 \to \mathbb{R}$ y una caja rectangular $[\mathbf{h}] \subseteq \Omega$, donde sus dimensiones son representadas por los intervalos $[x]$, $[y]$ y $[z]$, usando los conceptos de Aritmética Intervalar definidos en la sección 3.4. Sea la función $F_{[]} : \mathbb{IR}^3 \to \mathbb{IR}$ la extensión intervalar de la función F, entonces es posible decir que:

$$F_{[]}([\mathbf{h}]) = F_{[]}([x], [y], [z]) \subseteq \mathbb{IR}$$

Así, $F_{[]}([\mathbf{h}])$ son intervalos que estiman el conjunto de valores resultantes de la evaluación intervalar de la función F en la caja $[\mathbf{h}]$. eso permite evaluar si la caja $[\mathbf{h}]$ no intercepta puntos de la superficie implícita $\mathcal{S} = F^{-1}(0)$, que significa:

$$0 \notin F_{[]}([\mathbf{h}]) \Rightarrow 0 \notin F([\mathbf{h}]).$$

Ese hecho es usado en el proceso de subdivisión del árbol *Octree*, como será descrito en la siguiente subsección.

Criterios de Subdivisión

Para que una caja sea subdividida debe satisfacer tres criterios: criterio de la componente conexa, criterio topológico y criterio de nivel máximo.

Criterio de la Componente Conexa

Este criterio evalúa si una caja puede contener puntos de la superficie $\mathcal{S} = F^{-1}(0)$. Para eso, se utiliza la evaluación intervalar $F_{[]}([\mathbf{h}_n])$ de la función F en la caja $[\mathbf{h}_n]$.
Si el valor $0 \notin F_{[]}([\mathbf{h}])$, entonces no existen puntos de la superficie en la caja $[\mathbf{h}_n]$. Caso contrario, si $0 \in F_{[]}([\mathbf{h}])$, posiblemente puede haber puntos interceptando la superficie dentro de esa caja, entonces se debe subdividir esa caja $[\mathbf{h}_n]$ para explorar mejor esa posibilidad.

Criterio Topológico

El criterio de la componente conexa no garantiza la preservación topológica de una superficie que contiene túneles o huecos. Para evitar eso se evalúa si el vector $(0, 0, 0)$ está dentro de la imagen del gradiente de la función F en la caja $[\mathbf{h}_n]$. Si la evaluación intervalar del gradiente de F contiene el origen, entonces se debe subdividir la caja $[\mathbf{h}_n]$.

Criterio del Nivel Máximo

Los criterios anteriores pueden hacer que sucedan un gran número de subdivisiones, hasta el infinito. Para evitar eso es establecido un nivel máximo de subdivisiones.

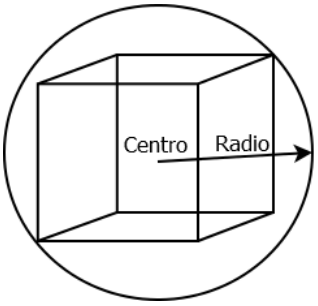

Figura 4.2: *Caja y su esfera envolvente.*

4.3.2. Envío de datos al GPU

Terminada la construcción del árbol *Octree*, el paso siguiente es el envío de los datos de ese árbol y de la matriz de rotación para el GPU. Se extrae del árbol una jerarquía de esferas envolventes, que es dada por:

- Los puntos centrales de cada caja que contiene la superficie, y el valor de la diagonal, que representa el radio de la esfera envolvente de esa caja. Ecuaciones 4.1 y 4.2.
- Referencias a las cajas hijas que contienen la superficie.

La Figura 4.2 representa esa extracción.

$$\mathbf{c} = ((\underline{x}+\overline{x})/2, (\underline{y}+\overline{y})/2, (\underline{z}+\overline{z})/2) \quad (4.1)$$

$$\mathbf{r} = \sqrt{(\overline{x}-\underline{x})^2 + (\overline{y}-\underline{y})^2 + (\overline{z}-\underline{z})^2} \quad (4.2)$$

La extracción de esa información es hecha solo una vez, y así que estuviera disponible es enviada a la GPU.

El cálculo de la matriz de rotación es hecha en cada *frame*, y enviada a la GPU en cada procesamiento del *frame*.

El proceso continua en la siguiente sección, donde serán presentados los cálculos para la visualización que son efectuados en la GPU.

4.4. Procesamiento en GPU

En esta sección será explicado la segunda fase del método propuesto, que corresponde al procesamiento en GPU del uso de *Ray Tracing*. También van a ser descritas algunas optimizaciones para disminuir los cálculos en GPU.

4.4.1. Descripción general del procesamiento en GPU

El procesamiento comienza con el envío de dos triángulos a la GPU. En el estado del *vertex shader*, los vértices permanecen inalterados, y son enviados para el siguiente estado. Los triángulos son rasterizados y para cada pixel se activa un *fragment shader*. En este estado es calculado la posición del pixel en $\mathbb{R}^3$, usando la matriz de rotación, y es lanzado un rayo a través del pixel. Para el rayo se busca la intersección con la superficie, usando los centros y el radio de cada esfera de la jerarquía dada por el árbol *Octree*. Se utiliza el método de bisección para calcular las posibles intersecciones. Esas intersecciones, caso existan, son ordenadas según la distancia al origen del rayo. Para cada punto de intersección es calculada la iluminación y es hecho el *blending*. También se analiza si el punto pertenece a la silueta o al borde del dominio.

En las siguientes subsecciones serán descritos los detalles de cada etapa del procesamiento en GPU.

4.4.2. Uso de dos triángulos

El *Ray Tracing* es hecho en el estado del *fragment shader*, pero para que se activen los fragmentos (pixeles) donde serán procesados los rayos, primero se debe activar el *vertex shader*. Como no existe ninguna primitiva geométrica, entonces se debe crearla de tal forma que cubra todo el dominio proyectado.

Una solución simples fue usar dos triángulos de un rectángulo que cubra todo el dominio, y así cada pixel solo es analizado una vez, ver Figura 4.3. Los vértices de esos dos triángulos serian, entonces enviados a la GPU. En el estado del *vertex shader* del *Pipeline* gráfico, los vértices son pasados sin modificaciones para el *fragment shader* (ver código listado en 4.1). De esa manera, son activados todos los pixeles dentro de los dos triángulos, esos pixeles serán procesados en el *fragment shader* como será descrito en la siguiente subsección.

Listing 4.1: *Vertex Shader, pasa para el siguiente estado los vértices sin modificaciones*

```
in vec3 position;
void main() {
  gl_Position = vec4(position, 1.0);
}
```

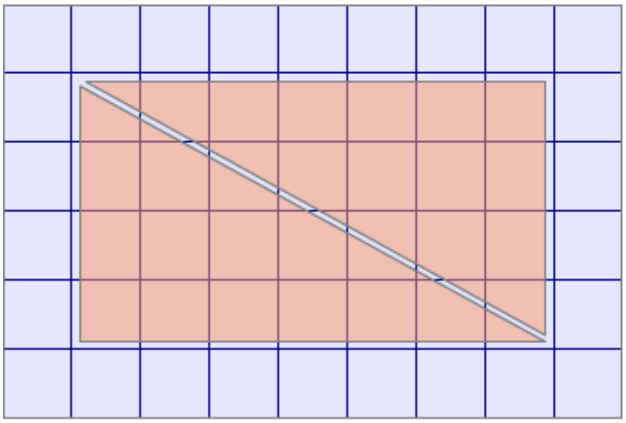

Figura 4.3: *Dos triángulos usados para activar los pixeles que serán procesados.*

4.4.3. Dentro del *fragment shader*

Una vez dentro del *fragment shader*, tenemos que determinar cual pixel va ser procesado, teniendo en consideración que el pixel $(0, 0)$ debe estar en el centro de la pantalla, para eso se utiliza el código listado en 4.2.

Listing 4.2: *Obteniendo las coordenadas del pixel*

```
void main() {
  vec2 pixel = uTamPixel*(gl_FragCoord.xy-0.5*(uRes.xy-1.0));
  ...
}
```

Donde: *uTamPixel* representa el tamaño del pixel; *gl_ FragCoord* son las coordenadas del pixel mantenidas por el *fragment shader*, cuyos valores van de 0 hasta el ancho y alto de la ventana; y *uRes* representa el valor del ancho y alto de la ventana de visualización.

Luego, se necesita determinar cual es el origen y dirección de cada rayo en el espacio $\mathbb{R}^3$. Ya que está siendo considerada la projección perspectiva, entonces el origen de cada rayo está en el observador y el rayo pasa por el pixel que se está evaluando. La posición del observador es enviada del CPU a la GPU.

La siguiente sección describirá sobre la generación de rayos.

4.4.4. Generación de rayos

Se generan dos tipos de rayos, rayos primarios y secundarios. Para los rayos primarios el origen del rayo está en la posición del observador, y la dirección del rayo tiene que pasar por cada pixel, como se puede observar en la Figura 4.4.

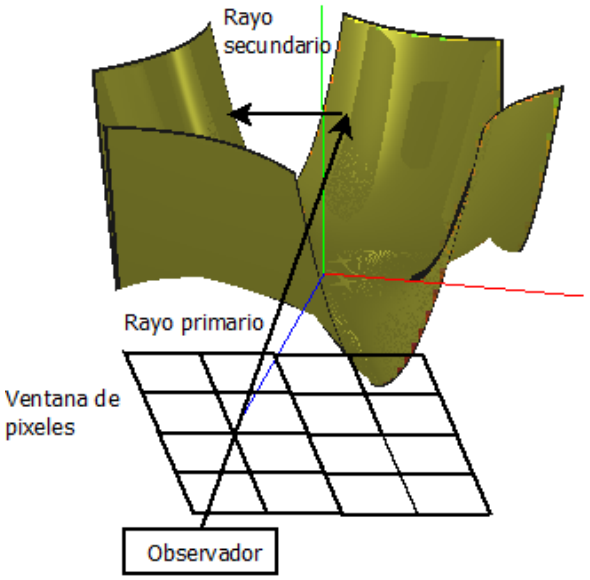

Figura 4.4: *Partes del Ray Tracing usado en el presente trabajo.*

Para calcular los rayos secundarios, el origen del rayo está en el punto de intersección con la superficie, y la dirección del rayo es calculado mediante la ecuación de reflexión, el cuál utiliza la normal del punto de intersección, y la normal es calculado mediante la gradiente de la función en ese punto, se puede observar en la Figura 4.4 la ubicación del rayo secundario y en la Figura 4.5 se observa como se cálcula la dirección del rayo secundario.

Si el rayo secundario tiene intersección con la superficie, entonces se calcula la iluminación y el color en dicho punto. En la siguiente subsección 4.4.5 será descrito como encontrar las raíces de una función implícita, y en la subsección 4.4.6 se explica como se obtiene el color final.

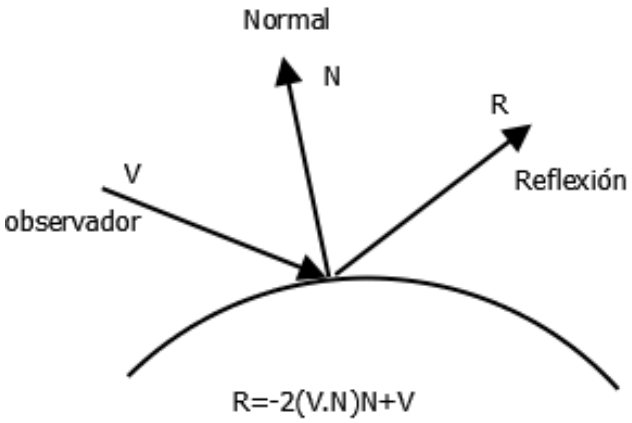

Figura 4.5: *Cálculo del Rayo de Reflexión, usando la normal del punto de intersección.*

4.4.5. Encontrando las raíces

Para cada rayo lanzado se debe encontrar las intersecciones con la superficie implícta. Para eso se usa los conceptos descritos en la sección 3.2, donde es descrito el método de la bisección para determinar las raíces de una función. Vale observar que en el lenguage del *shader* no es permitido escribir funciones recursivas, y, por eso, el método de la bisección tuvo que ser implementado en su versión iterativa, ver Algoritmo 1.

Algorithm 1 Algoritmo de bisección iterativo, implementado en la GPU.

Require: $s_1 \neq s_2$, $numIter > 0$, $Approximation > 0$
Ensure: p_m está muy cerca de la superficie F.

```
1:  function BisectionIterative(p1, p2, s2)        ▷ s: Signo del valor de F(p).
2:      for i ← 0 to numIter do
3:          pm ← (p1 + p2)/2
4:          fm ← F(pm)
5:          if abs(fm) < Approximation then
6:              break                              ▷ Salir porque se encontró una raíz.
7:          end if
8:          sm ← Sinal(fm)
9:          if sm ≠ s2 then
10:             p1 ← pm
11:         else
12:             s2 ← sm
13:             p2 ← pm
14:         end if
15:     end for
16:     return pm                                  ▷ pm es una raíz.
17: end function
```

El algoritmo recorre la jerarquía de las esferas representadas por el árbol *Octree*. Para cada esfera, primeramente se determina cuales son los valores de t_{min} y t_{max} a través del cálculo de intersección analítico entre un rayo y una esfera, ver subsección 3.2.2. Ese cálculo es hecho muy rápidamente, y con eso se consigue buscar las raíces solo en las esferas que ese rayo intercepta. Se puede observar en la Figura 4.6 la intersección entre un rayo y una esfera para determinar los valores de t_{min} y t_{max}. Si la esfera tiene hijos, osea no es hoja, buscar por t_{min} y t_{max} en las esferas hijas. Si la esfera es una hoja, entonces con los valores de t_{min} y t_{max} se puede encontrar la intersección del rayo con la superficie implícita usando el método descrito en la subsección 3.2.3, donde es utilizado el método de intervalos compuesto con el algoritmo de la bisección para encontrar las raíces, observar la Figura 4.7.

Si existen intersecciones, entonces son guardados los valores de t y los respectivos puntos de intersección. Puede suceder que tengan muchas intersecciones, y por eso, solo serán

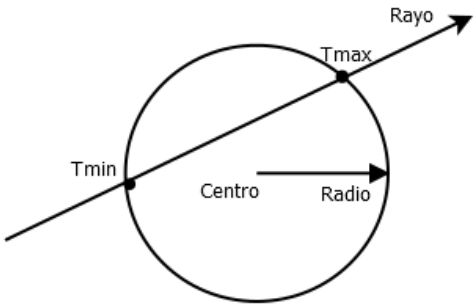

Figura 4.6: *Calculando los valores t_{min} y t_{max}.*

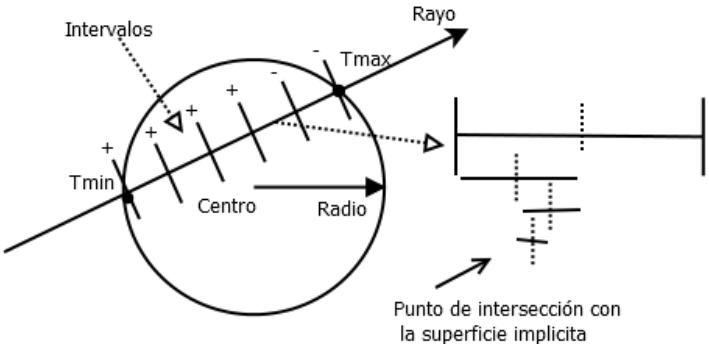

Figura 4.7: *Método de intervalos compuesto con el algoritmo de bisección.*

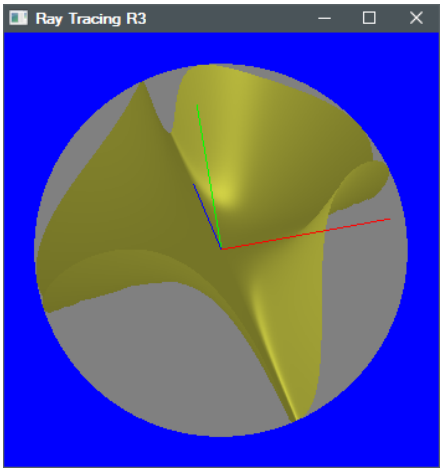

Figura 4.8: *Iluminación de la función* $F(x, y, z) = x^2 * z^2 - y$

guardados los n menores valores de t con sus respectivos puntos.

El paso siguiente es obtener los colores con la iluminación y hacer el mezclado (*blending*) de esos puntos, eso será descrito en la siguiente sección

4.4.6. Iluminación

Para hacer la iluminación es utilizado el modelo de Phong, cuyos conceptos básicos fueron descritos en la sección 3.6.

Con el punto de intersección del rayo primario, se calcula la normal de la superficie en ese punto, y con el modelo de iluminación es obtenido el color en ese punto, si se ha lanzado un rayo secundario, de ese rayo secundario también se obtiene un color, de la misma forma como se obtiene el color del rayo primario, y luego dicho color es mezclado con el color del rayo primario según la ecuación del *blending*: $color = color_n(1{,}0 - \alpha) + color_{n-1} * \alpha$ con $\alpha = 0{,}7$.

En la Figura 4.8, se puede observar un ejemplo usando el modelo de iluminación con una fuente de luz. También en dicha imagen se observa la esfera envolvente dentro de la cual es evaluada la intersección del rayo, el área azul representa el área evitada de evaluación de intersección del rayo.

4.4.7. Silueta y borde del dominio

Con el propósito de mejorar la visualización son adicionados dos atributos: la silueta y el borde del dominio.

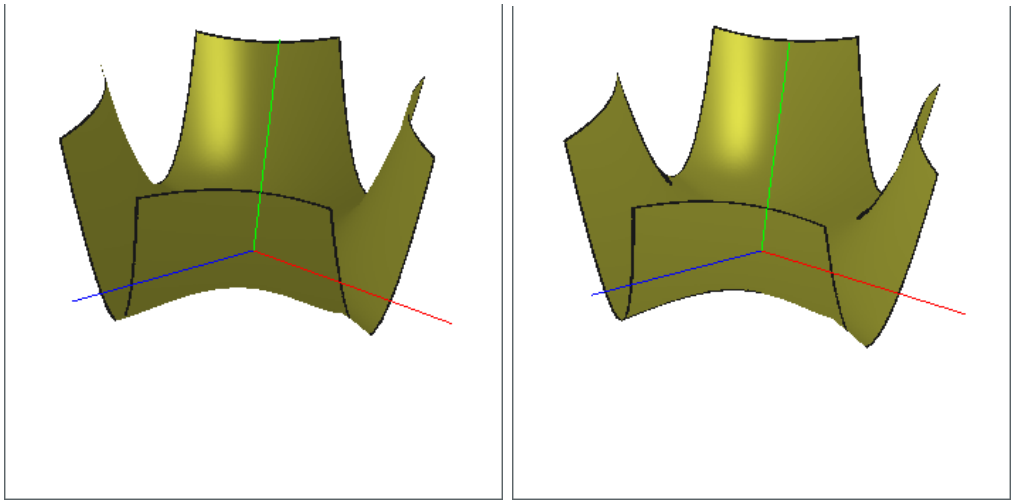

Figura 4.9: *Borde y silueta del dominio de la función* $F(x, y, z) = x^2 * z^2 - y$.

Borde del dominio

Las superficies se pueden extender al infinito, pero las superficies son calculadas y analizadas en un dominio dado por el usuario que corresponde a la caja [**h**]. De esa forma, la superficie implícita puede interceptar el borde de la caja y esos puntos de intersección serán realzados en las imágenes generadas. Para eso, se define una distancia mínima al borde. Los puntos que distan menos que esa tolerancia al borde también serán realzados. En la Figura 4.9 al lado izquierdo se puede observar los puntos del borde del dominio.

Silueta

La silueta es definida como los puntos de la superficie donde su normal es perpendicular al vector dirección del rayo, esto es:

$$\nabla F(\mathbf{p}) \cdot \mathbf{dir} = 0 \tag{4.3}$$

En la Figura 4.9 al lado derecho se puede observar los puntos de la silueta.

Capítulo 5

Pruebas y Resultados

En este Capítulo se presenta el prototipo implementado, las funciones utilizadas, las pruebas realizadas y los resultados obtenidos.

La sección 5.1 presenta una descripción de las funciones implementadas en el prototipo del sistema, luego en la sección 5.2 se presenta las funciones implícitas usadas como pruebas, y en la sección 5.3 se presenta los análisis de *frames* por segundo observando la variación del nivel del árbol *Octree* y la cantidad de intervalos del método.

5.1. Prototipo Implementado

Se ha programado un prototipo de un sistema de visualización utilizando el método propuesto, la implementación fue hecha en Visual Studio C++ 2015. Las pruebas fueron hechas en un computador Intel Core i5 2.5 GHz y con placa gráfica Intel HD Graphics 3000.

Las opciones implementadas incluyen:

- Selección dinámica de la función implícita a visualizar.
- Interactividad usando el *mouse* para girar la visualización.
- Aumento y disminución de la intensidad luminosa de la luz direccional.
- Aumento y disminución de la intensidad luminosa da la luz puntual.
- Activar y desactivar el realce de los puntos del borde del dominio.
- Activar y desactivar el realce de los puntos de la silueta.
- Aumento y disminución dinámica de la precisión en la aproximación a la raíz.

- Aumento y disminución del nivel del árbol *Octree*, por consiguiente afecta al nivel del *bounding volume sphere*.
- Aumento y disminución de la cantidad de intervalos, para encontrar la raíz.

5.2. Funciones Implícitas de Prueba

Las imágenes presentadas en este Capítulo, corresponden a las funciones implícitas definidas en la Tabla 5.1.

Función	$f(x, y, z) =$
1	Canal en Cruz $x^2 + z^2 - y$
2	Cúbica de Cayley $x^2y + x^2z + y^2x + y^2z + z^2y + z^2x - (2/5)(xy + xz + yz)$
3	Cuboide $x^4 + y^4 + z^4 - 1$
4	Peninsula $x^2 + y^3 + z^5 - 1$
5	Cuártica de Klein $x^3y + y^3z + z^3x$
6	Silla Cuártica $x^4 - z^4 - y$
7	Superficie Racor $x^2y^2 + y^2z^2 + z^2x^2 - 1$

Cuadro 5.1: *Funciones implícitas de prueba.*

La Figura 5.1 presenta la visualización de las superficies implícitas de la Tabla 5.1.

5.3. Evaluación

En esta sección serán presentadas figuras y tablas resumiendo las diferentes pruebas realizadas. Sigue una evaluación de *frames* por segundo variando el número de intervalos y el número de niveles del árbol *Octree* 5.3.1, y variando los niveles del árbol *Octree* y el valor de aproximación a la raíz 5.3.2.

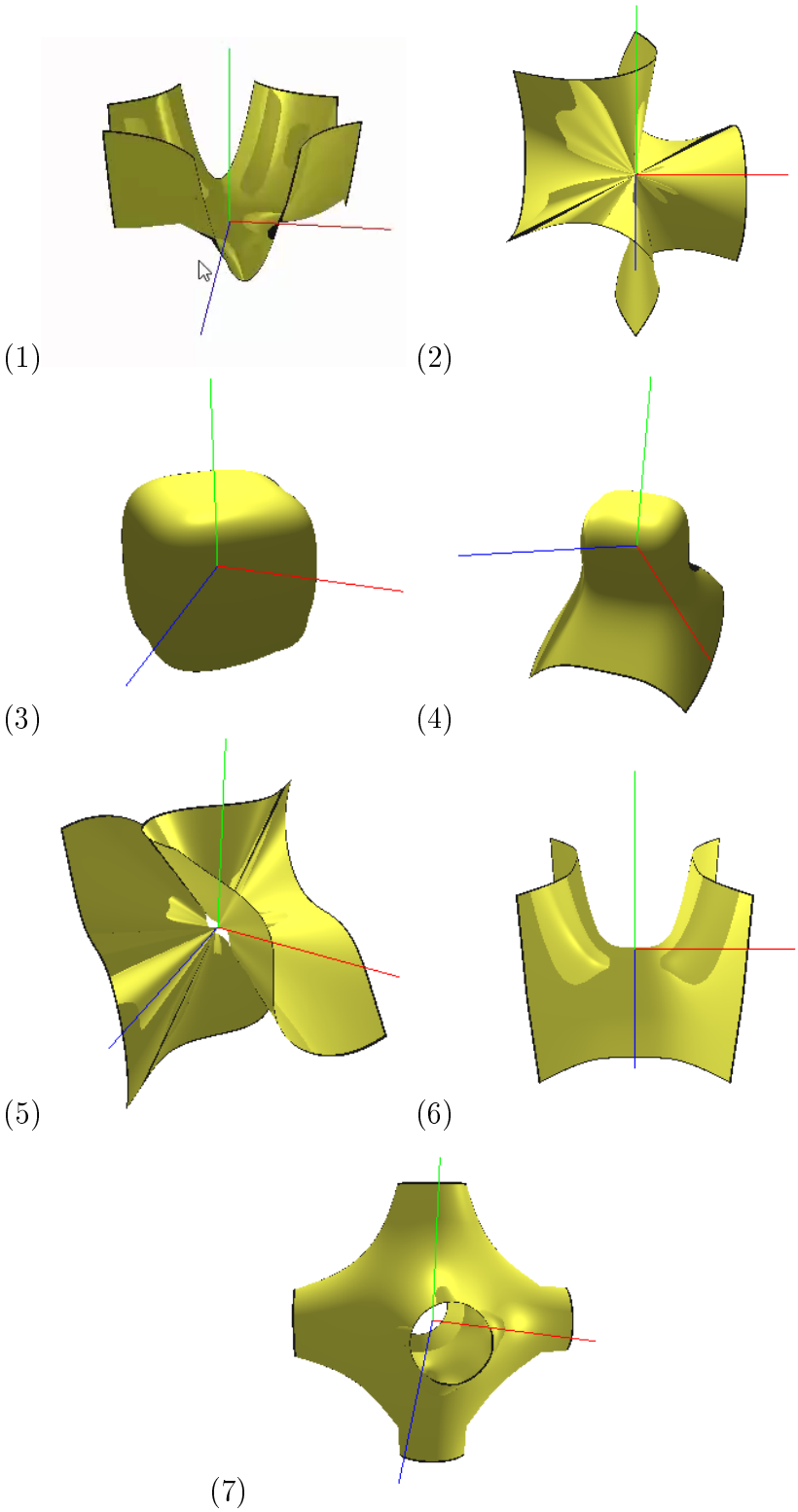

Figura 5.1: *Visualización de las superficies implícitas de la Tabla 5.1, presentadas en diferentes ángulos.*

5.3.1. Variando la cantidad de intervalos y el número de niveles del árbol

La Tabla 5.2 presenta los resultados obtenidos de los cálculo de *frames* por segundo, variando el número de intervalos en (10, 15, 20) y variando los niveles del árbol (0, 1, 2), considerando una aproximación de 0.001 para encontrar la intersección con la superficie implícita. En las Figuras 5.2, 5.3, 5.4 y 5.5 se puede observar los resultados, las imágenes tienen buena calidad, variar el nivel del árbol y/o variar el número de intervalos no afecta demasiado a la calidad, a lo que afecta es a la renderización lo que se nota una disminución en los *frames* por segundo, la mejor relación de calidad e interactividad se encuentra con los valores de 10 intervalos, y con el primer nivel del árbol.

Función	Nivel del árbol	fps			Imágenes
		10 intervalos	15 intervalos	20 intervalos	
1	0	21	16	14	Figura 5.2
	1	22	18	13	
	2	20	15	13	
2	0	20	15	11	
	1	21	16	10	
	2	19	14	10	
3	0	27	22	20	Figura 5.3
	1	28	23	20	
	2	25	19	16	
4	0	27	19	15	
	1	28	20	15	
	2	27	19	14	
5	0	17	12	11	Figura 5.4
	1	18	13	10	
	2	14	10	8	
6	0	22	17	13	
	1	23	18	12	
	2	19	14	10	
7	0	24	19	17	Figura 5.5
	1	25	20	18	
	2	20	16	12	

Cuadro 5.2: *Frames por segundo obtenidos variando la cantidad de intervalos y el nivel del árbol Octree.*

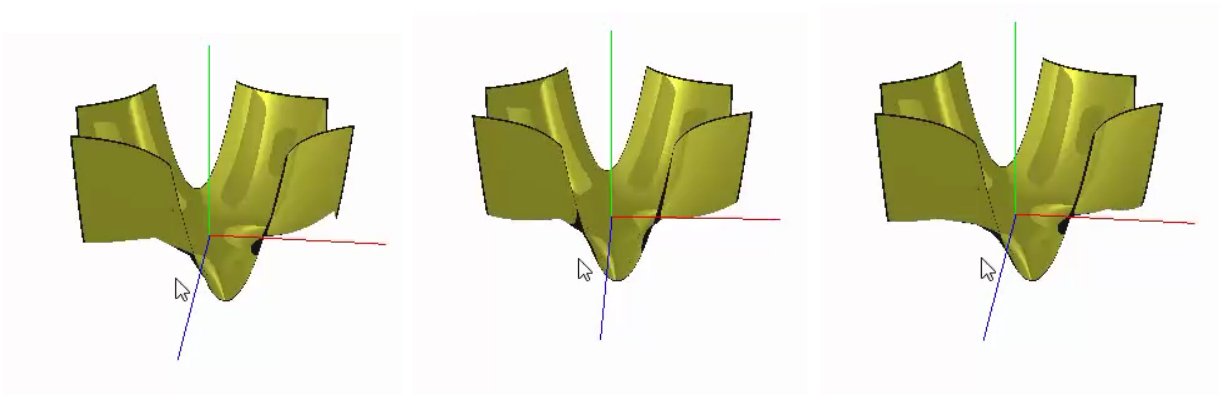

Figura 5.2: *Calidad de la función 1, variando la cantidad de intervalos (10,15,20), para el nivel 1 del árbol.*

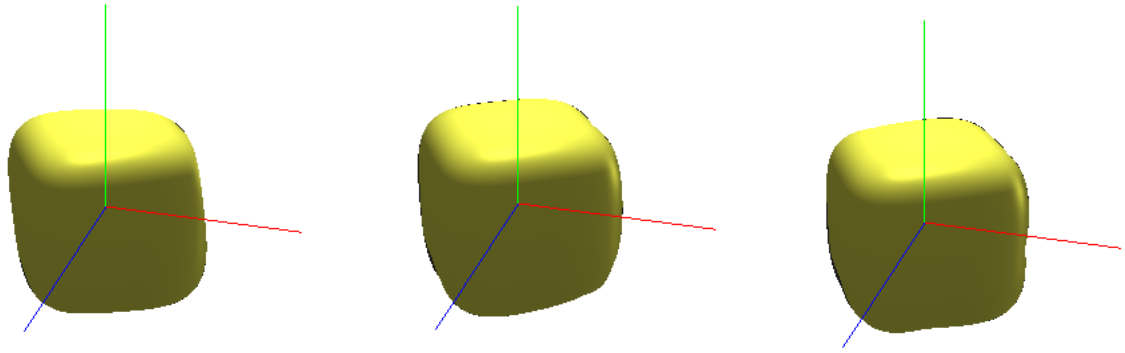

Figura 5.3: *Calidad de la función 3, variando la cantidad de intervalos (10,15,20), para el nivel 0 del árbol.*

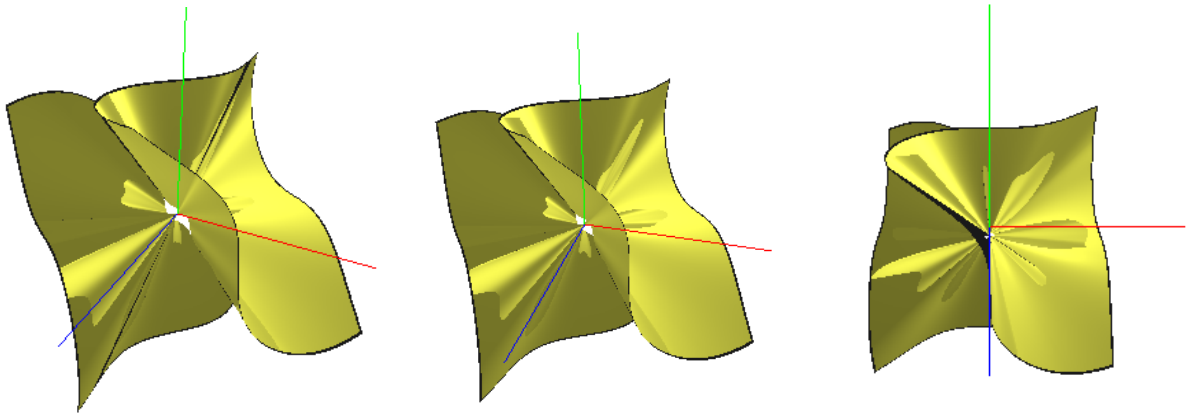

Figura 5.4: *Calidad de la función 5, variando la cantidad de intervalos (10,15,20), para el nivel 2 del árbol.*

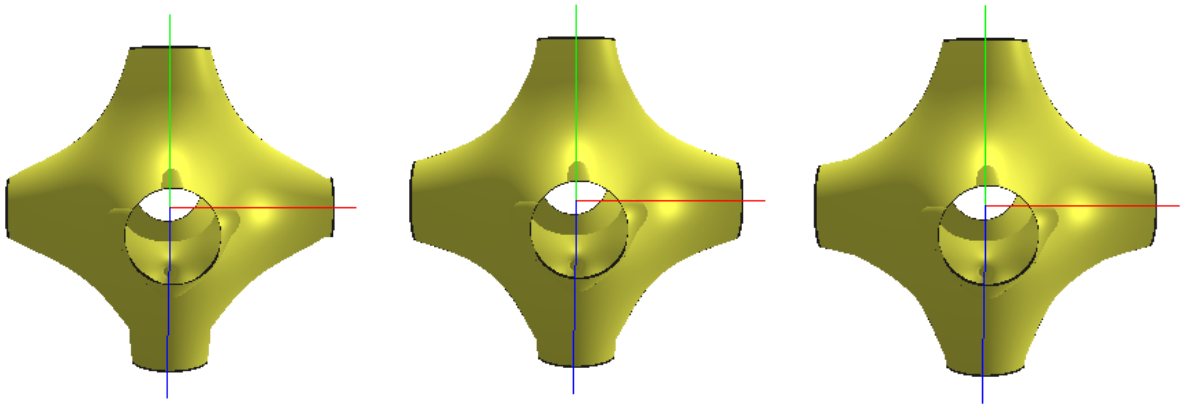

Figura 5.5: *Calidad de la función 7, variando la cantidad de intervalos (10,15,20), para el nivel 1 del árbol.*

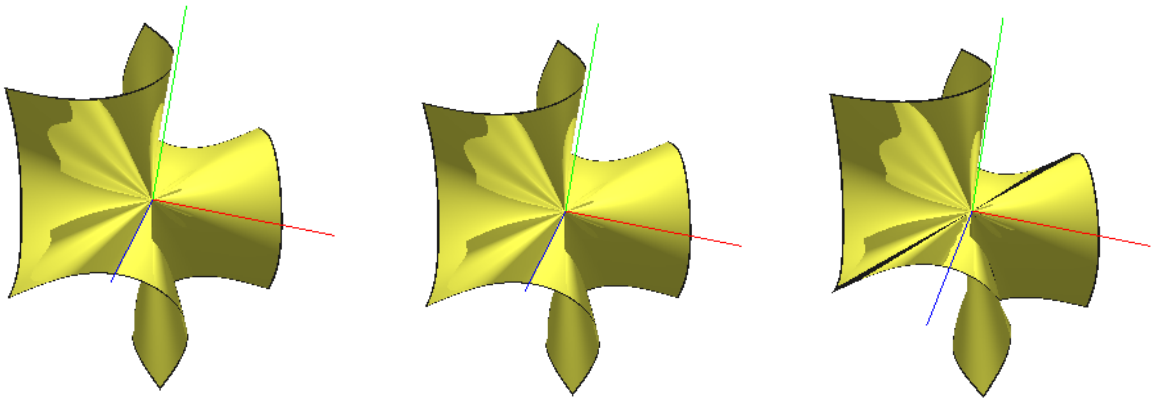

Figura 5.6: *Calidad de la función 2, variando el nivel máximo del árbol (nivel 0, 1 y 2), para un valor de aproximación de 0.001*

5.3.2. Variando el nivel del *Octree* y el valor de la aproximación

La Tabla 5.3 presenta los resultados del análisis hecho variando el nivel máximo del árbol y variando el valor de la aproximación a la raíz de la función implícita, considerando los niveles de árbol (0, 1, y 2) y los valores de la aproximación como (0.01, 0.001, 0.0001), considerando la cantidad de 10 intervalos. en las Figuras 5.6, 5.7 y 5.8 se puede observar los resultados. Variar el valor de aproximación a la raíz, no afecta demasiado a la calidad de la imagen resultante, lo que si afecta es a la renderización que se nota con la disminución de los *frames* por segundo, la mejor relación de calidad e interactividad.se da con el valor de aproximación de 0.01 y con el primer nivel del árbol.

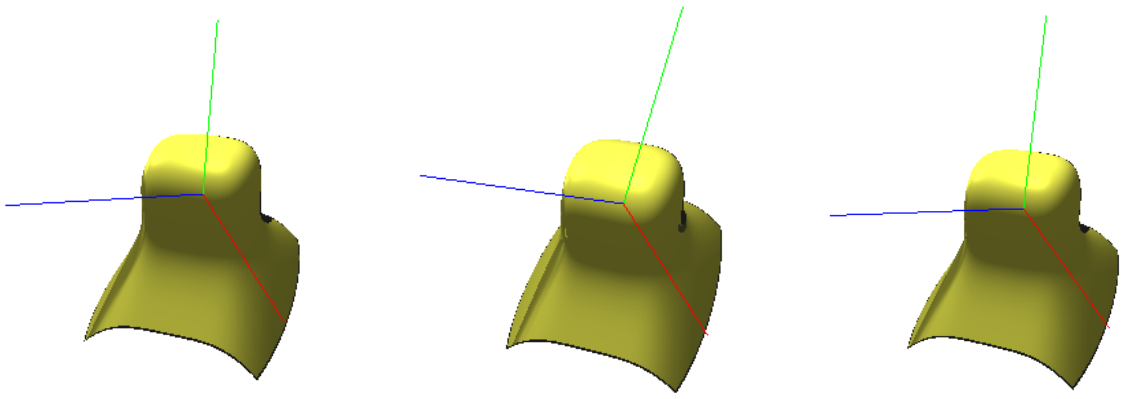

Figura 5.7: *Calidad de la función 4, variando el valor de aproximación a la raíz (0.01, 0.001, 0.0001), para el nivel 1 del árbol.*

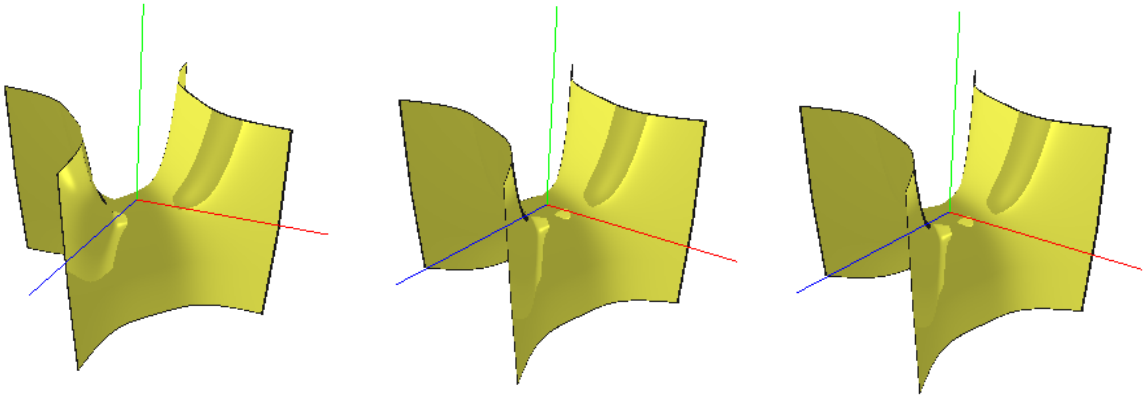

Figura 5.8: *Calidad de la función 6, variando el valor de aproximación a la raíz (0.01, 0.001, 0.0001), para el nivel 1 del árbol.*

Función	Nivel del árbol	fps			Imágenes
		aprox. 0.01	aprox. 0.001	aprox. 0.0001	
1	0	21	15	13	
	1	22	16	12	
	2	18	14	10	
2	0	20	18	16	
	1	21	19	18	
	2	19	15	13	Figura 5.6
3	0	27	24	22	
	1	28	25	22	
	2	25	22	18	
4	0	27	24	23	
	1	28	26	24	
	2	27	25	22	Figura 5.7
5	0	17	16	14	
	1	18	16	14	
	2	14	13	12	
6	0	22	20	18	
	1	23	21	19	
	2	19	17	16	Figura 5.8
7	0	24	21	18	
	1	25	22	20	
	2	20	18	16	

Cuadro 5.3: *Frames por segundo obtenidos variando el nivel del árbol Octree y el valor de aproximación a la raíz.*

5.3.3. Resultados

Por las pruebas realizadas, se puede observar cuales son los parámetros óptimos para la visualización de la superficie implícita, para obtener una buena relación entre calidad de la imagen e interactividad. La calidad de la imagen no se vio muy afectada con la variación de los parámetros, en lo que si afectó fue en la velocidad de renderización, que se nota en la disminución de los *frames* por segundo.

Observando las imágenes y las tablas, se puede decir que los parámetro más adecuados para la visualización es el nivel 1 del árbol, la cantidad de 10 intervalos y con un valor de aproximación a la raíz de 0.01.

Capítulo 6

Conclusiones y Recomendaciones

En este trabajo se presentó un nuevo método para la visualización de superficies implícitas de dimensión 2 inmersas en $\mathbb{R}^3$. El método hace uso de un pré-procesamiento en CPU, y de un procesamiento en GPU.

Para el pré-procesamiento en CPU se ha usado la Aritmética Intervalar y la estructura de datos espacial *Octree*; para enviar la información del CPU al GPU se ha usado la estructura de BVS (esfera de volumen envolvente); para el procesamiento en GPU se ha usado la técnica de *Ray Tracing* y él método de intervalos con bisección.

Las pruebas fueron realizadas sobre un conjunto conocido de funciones implícitas, se implementó un prototipo de sistema con el método propuesto, y se utilizaron los parámetros de niveles del árbol de *Octree*, la cantidad de intervalos para encontrar la raíz de la función implícita, y el valor de la aproximación a la raíz.

La calidad de la imagen no se vio muy afectada con la variación de los parámetros, en lo que si afectó la variación de los parámetros fue en la velocidad de renderización, que se nota en la disminución de los *frames* por segundo, cuando se intentaba obtener más detalle en la visualización.

Se puede decir que los parámetros más adecuados para la visualización es el nivel 1 del árbol, la cantidad de 10 intervalos y con un valor de aproximación a la raíz de 0.01. Con respecto al uso del árbol *Octree*, los *frames* por segundo disminuyen mucho con dos niveles del árbol. Solo es interactivo con los niveles 0 y 1.

Hay aun muchos trabajos que pueden ser hechos como extensión de este, por ejemplo:

1. Investigar otros métodos para acelerar los cálculos en GPU, que puede ser:
    - Otra estructura de datos con jerarquía,
    - Otra manera de enviar los datos a la GPU, tal vez usando texturas.
2. Un camino muy interesante es el de mejorar la visualización con la transparencia.
3. También se puede utilizar técnicas antialiasing para mejorar la calidad final de la imagen.

Bibliografía

[Akenine-Möller et al., 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA. 11

[Appel, 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA. ACM. 8

[Arvo and Kirk, 1989] Arvo, J. and Kirk, D. (1989). A survey of ray tracing acceleration techniques. *An introduction to ray tracing*, pages 201–262. 17

[Bloomenthal, 1994] Bloomenthal, J. (1994). An implicit surface polygonizer. In *Graphics Gems IV*, pages 324–349. Academic Press. 4

[Cordero, 2017] Cordero, L. A. (2017). Superficies definidas en forma implicita. VII, 7

[de Araújo et al., 2015] de Araújo, B. R., Lopes, D. S., Jepp, P., Jorge, J. A., and Wyvill, B. (2015). A survey on implicit surface polygonization. *ACM Comput. Surv.*, 47(4):60:1–60:39. 4

[de Cusatis Junior et al., 1999] de Cusatis Junior, A., de Figueiredo, L. H., and Gattas, M. (1999). Interval methods for raycasting implicit surfaces with affine arithmetic. *Proceedings of XII SIBGRAPHI*. 5

[Engel et al., 2006] Engel, K., Hadwiger, M., Kniss, J., Rezk-salama, C., and Weiskopf, D. (2006). *Real-Time Volume Graphics*. Ak Peters Series. A. K. Peters, Ltd., Natick, MA, USA. VII, 11, 15, 19, 20

[Flórez Díaz, 2008] Flórez Díaz, J. E. (2008). *Improvements in the Ray Tracing of Implicit Surfaces Based on Interval Arithmetic*. PhD thesis, Universitat de Firona. VII, 11, 19

[Franquesa Niubó and Rodríguez González, 2005] Franquesa Niubó, M. and Rodríguez González, O. (2005). Sphere-trees generation as needed in real time to speed up collision detection. VII, 17, 18

[Fryazinov and Pasko, 2008] Fryazinov, O. and Pasko, A. (2008). Interactive ray shading of frep objects. *16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, WSCG'2008 - In Co-operation with EUROGRAPHICS, Full Papers*. 5

[Gomes et al., 2009] Gomes, A., Voiculescu, I., and; Brian Wyvill, J. J., and Galbraith., C. (2009). *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms.* Springer. 1

[Hart, 1996] Hart, J. C. (1996). Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545. 5

[Kajiya, 1986] Kajiya, J. T. (1986). The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150. 8

[Knoll, 2009] Knoll, A. (2009). *Ray Tracing Implicit Surfaces for Interactive Visualization.* The University of Utah, Salt Lake City, Utah. 5, 7

[Knoll et al., 2007] Knoll, A., Hijazi, Y., Hansen, C., Wald, I., and Hagen, H. (2007). Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 11–18. IEEE. 5

[Levoy and Whitted, 1985] Levoy, M. and Whitted, T. (1985). *The use of points as a display primitive.* University of North Carolina, Department of Computer Science. 5

[Lighthouse3d, 2014] Lighthouse3d (2014). Pipeline opengl 4.2. Accessed: 2014-08-11. VII, VII, 13, 14

[Liktor, 2008] Liktor, G. (2008). Ray tracing implicit surfaces on the gpu. In *12th Central European Seminar on Computer Graphics (CESCG 2008 Proceedings). Technische Universität Wien, Institut für Computergraphik und Algorithmen.* 6

[Mitchell, 1990] Mitchell, D. P. (1990). Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface*, volume 90, pages 68–74. 5

[Moore, 1966] Moore, R. (1966). *Interval analysis.* Prentice-Hall series in automatic computation. Prentice-Hall, Upper Saddle River, NJ 07458, USA. 15, 16, 17

[Paiva et al., 2006] Paiva, A., Lopes, H., Lewiner, T., and De Figueiredo, L. H. (2006). Robust adaptive meshes for implicit surfaces. In *19th Brazilian Symposium on Computer Graphics and Image Processing, 2006. SIBGRAPI'06.*, pages 205–212, Manaus (AM), Brazil. IEEE. 4

[Pharr and Fernando, 2005] Pharr, M. and Fernando, R. (2005). *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional. VII, 18

[Phong, 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317. 19

[Romeiro et al., 2006] Romeiro, F., Velho, L., and De Figueiredo, L. H. (2006). Hardware-assisted rendering of csg models. In *Computer Graphics and Image Processing, 2006. SIBGRAPI'06. 19th Brazilian Symposium on*, pages 139–146. IEEE. 5

[Rusinkiewicz and Levoy, 2000] Rusinkiewicz, S. and Levoy, M. (2000). Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co. 5

[Samet, 1990] Samet, H. (1990). *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA. VII, 17, 18

[Shreiner et al., 2013] Shreiner, D., Sellers, G., Kessenich, J., Licea-Kane, B., and Group, K. O. A. W. (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Graphics programming. Addison-Wesley, Boston, Mass. 1, 11

[Sigg, 2006] Sigg, C. (2006). *Representation and Rendering of Implicit Surfaces*. PhD thesis, ETH Zurich. 5

[Singh, 2013] Singh, J. M. (2013). Real-time approximate and exact csg of implicit surfaces on the gpu. In *2013 Fourth National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG)*, pages 1–3, Jodhpur, India. IEEE. 8

[Suffern, 2007] Suffern, K. (2007). *Ray tracing from the ground up*. Ak Peters Series. A K Peters, Natick, MA, USA. VII, 9

[Wald et al., 2005] Wald, I., Friedrich, H., Marmitt, G., Slusallek, P., and Seidel, H.-P. (2005). Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572. 6

[Whitted, 1980] Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349. 8

[Wyvill et al., 1986] Wyvill, G., McPheeters, C., and Wyvill, B. (1986). Data structure for soft objects. *The visual computer*, 2(4):227–234. 4