Dissertação apresentada para obtenção do título de Mestre em Matemática pelo

Instituto Nacional de Matemática Pura e Aplicada

Visualização de Superfícies Implícitas com Traçado de Raios na GPU

por

Francisco Ganacim

Orientador: Luiz Henrique de Figueiredo

Rio de Janeiro

28 de fevereiro de 2011

Agradecimentos

À minha família, pelo apoio ao longo dos anos. Aos colegas do Visgraf, por propiciarem um ambiente intelectualmente rico e estimulante. Ao meu orientador, Prof. Luiz Henrique de Figueiredo, pela paciência e pela ajuda ao longo de todo o trabalho. À Nara, por todo carinho e dedicação. Finalmente, agradeço ao CNPq pelo suporte financeiro durante o programa de mestrado.

Resumo

Superfícies implícitas são objetos de grande interesse em computação gráfica devido à sua simplicidade e às suas ricas propriedade matemáticas. Essas características as tornam ferramentas adequadas a várias áreas, e.g.: modelagem geométrica, visualização e reconstrução de dados, animação e simulação. Apesar disto, a adoção de superfícies implícitas esbarra na dificuldade de visualização, que pode ser computacionalmente custosa.

O ray-casting de superfícies implícitas nos permite renderizar imagens diretamente da função, o que reduz a quantidade de dados no processo e possibilita manter uma alta qualidade em diferentes escalas do modelo, mantendo suas propriedades geométricas e topológicas. Esses métodos se baseiam na solução de equações não lineares para cada raio usado para amostrar a cena. As equações do raio podem ter soluções analíticas nos casos em que a função é um polinômio de grau até quatro ou em outros casos muito simples. Para os outros casos são necessárias estratégias numéricas.

Em nosso trabalho abordamos métodos numéricos para isolar raízes baseados em amostragem pontual e em Aritmética Intervalar (AI). Os métodos por amostragem são de simples implementação, porém não são robustos, isto é, podem produzir resultados que não representem com exatidão as propriedades geométricas e a iluminação da superfície. Usando a AI, construímos algoritmos eficientes e robustos, como por exemplo o Algoritmo de Mitchell. Também usamos AI para implementar uma forma de beam-tracing, o que nos permite visualizar certos pontos especiais da superfície.

Para podermos visualizar as superfícies de forma interativa, implementamos os métodos discutidos em GPU usando a plataforma CUDA.

Abstract

Implicit surfaces are objects of great interest in computer graphics due to its simplicity and its rich mathematical properties. These characteristics make them suitable tools to various areas, eg: geometric modeling, visualization and data reconstruction, animation and simulation. Nevertheless, the adoption of implicit surfaces stumbles on the difficulty of the visualization process, which can be computationally expensive.

The ray-casting of implicit surfaces allows us to render images directly from the function, which reduces the amount of data and enables the process to maintain a high quality of the model at different scales, maintaining their topological and geometrical properties. These methods are based on the solution of nonlinear equations for each ray used to survey the scene. The equations of the ray may have analytical solutions in cases where the function is a polynomial of degree up to four or in other simple cases. For other cases, numerical strategies are needed.

In our work we deal with numerical methods for isolating roots based on point–sampling and Interval Arithmetic (IA). The sampling methods are simple to implement, but are not robust, they can produce results that do not accurately represent the geometric properties and the lighting of surface. Using AI, we build efficient and robust algorithms such as Mitchell's Algorithm. We also use AI to implement a form of beam-tracing, which allows us to view certain special points of the surface.

To be able to view the surfaces interactively, we implement the methods discussed in GPU using CUDA.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Superfícies implícitas são objetos de grande interesse em computação gráfica devido à sua simplicidade e às suas ricas propriedade matemáticas. Essas características as tornam ferramentas adequadas a várias áreas, e.g.: modelagem geométrica, visualização e reconstrução de dados, animação e simulação [2, 10]. Apesar disto, a adoção de superfícies implícitas esbarra na dificuldade de visualização, que pode ser computacionalmente custosa. Podemos dividir os métodos de visualização em duas categorias: métodos por *discretização* (ou poligonização) e *ray–casting*.

Nos métodos por discretização a superfície de nível (ou iso–superfície) é aproximada por polígonos, normalmente triângulos, antes de ser visualizada. Um método bem conhecido de poligonização é o *Marching Cubes* [27]. Esses métodos tem limitações e geralmente encontram problemas para discretizar superfícies com certas características topológicas. Versões robustas desse algoritmo podem usar métodos intervalares para garantir que a triangulação mantenha as propriedades topológicas [33]. Outra opção é decompor a superfície em *patches* e depois triangulá–los [45]. Esses métodos podem ser usados em aplicações em *tempo real*, porém seu desempenho em aplicações onde a função muda com o tempo é ruim. Mesmo superfícies simples como uma esfera precisam ser aproximadas por muitos triângulos para que o resultado da visualização seja bom. Isto nos faz abrir mão de umas das principais características das superfícies implícitas, que é a representação compacta.

O *ray–casting* de implícitas nos permite renderizar imagens diretamente da função, o que reduz a quantidade de dados no processo e possibilita manter uma alta qualidade em diferentes escalas do modelo, mantendo suas propriedades geométricas e topológicas. Esses métodos se baseiam na solução de equações não lineares para cada raio usado para amostrar a cena. As equações do raio podem ter soluções analíticas nos casos em que a função é um polinômio de grau até quatro [1, 17], ou em alguns casos muito simples. Para os outros casos, são necessárias estratégias numéricas.

Alguns trabalhos exploram características especiais das funções para resolver numericamente as equações dos raios. Por exemplo, para superfícies algébricas podem ser usados métodos robustos de aproximação baseados na regra de Descartes [12], para funções do tipo LG são usadas as constantes de Lipschitz da função e de seu gradiente [19]. Para classes mais gerais temos os métodos intervalares.

Métodos intervalares são formas robustas para isolar e encontrar raízes. Eles se baseiam na possibilidade de encontrar boas aproximações para a imagem de um intervalo por uma função real, e assim verificar se uma função tem raízes em um intervalo. O primeiro trabalho nessa linha foi apresentado por Mitchell [29], na qual ele usa *Aritmética Intervalar* [30] para isolar intervalos onde a equação é monótona, para depois aplicar um método de bisecção em cada intervalo. Outra forma de avaliação intervalar é a *Aritmética Afim* [4], que foi usada em [5] para acelerar o processo de visualização, e a *Aritmética Afim Modificada* [41] – que só serve para polinômios.

Um problema comum na visualização de superfícies implícitas é o *aliasing*. Ele aparece quando escolhemos tomar poucas amostras da superfície em troca de eficiência. Esse problema é tratado em [6], usando Aritmética Intervalar.

1.1 Estrutura do Trabalho

No Capítulo 2 discutimos os aspectos teóricos do problema de visualização de superfícies implícitas. Em seguida, no Capítulo 3 discutimos os métodos mais comumente utilizados. Abordamos a implementação desses métodos em CPU e em GPU no Capítulo 4. O problema de encontrar estratégias eficientes

para *anti–aliasing* é tratado no Capítulo 5. Os algoritmos anteriores e as estratégias abordadas nos levam, no Capítulo 6, a abordar o *beam–tracing* intervalar e a propor um algoritmo de subdivisão espacial que usa Aritmética Intervalar como forma de acelerar a visualização.

Capítulo 2

Ray Casting

Neste capítulo daremos uma conceituação matemática para o problema de visualização de superfícies implícitas por meio de traçado de raios. Começaremos caracterizando o fenômeno físico que inspira os métodos posteriormente adotados. Mostraremos como modelar esse fenômeno matematicamente e como esse modelo poderá ser discretizado, visando sua implementação. Caracterizaremos o problema de *aliasing*, que é resultado da discretização do modelo.

2.1 Um Modelo Físico

Visualizar uma superfície significa obter uma representação gráfica bidimensional, perceptualmente coerente, da geometria de um objeto existente em um espaço tridimensional. Esse processo é comumente resumido por analogia ao funcionamento de uma câmera fotográfica. A Figura 2.1 representa uma cena contendo uma câmera, uma superfície e uma fonte de luz. Nela, os raios luminosos saem da fonte de luz em direção aos pontos da superfície e depois refletem em direção a câmera que registra uma imagem. Esse registro ocorre quando a luz atinge um anteparo que contém material químico foto–sensível (no caso de filmes) ou sensores digitais.

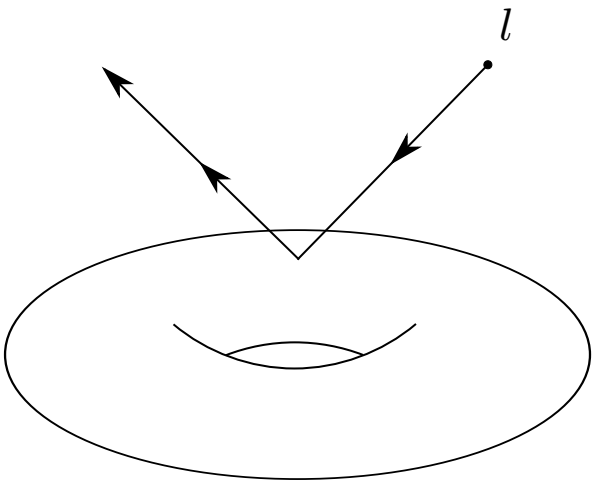
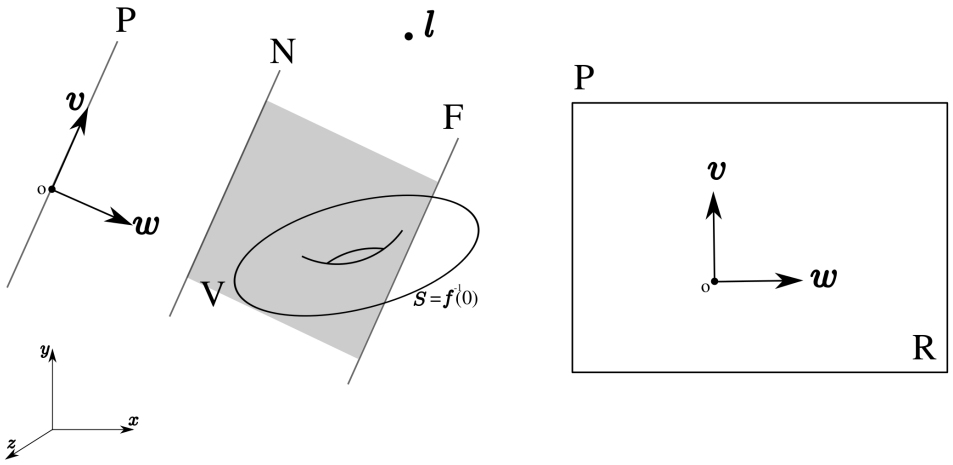


Figura 2.1: Representação de uma cena contendo uma superfície visível iluminada por uma fonte de luz pontual. A luz refletida pela superfície sensibiliza o anteparo da câmera, que registra uma imagem.

2.2 Um Modelo Matemático

Para caracterizar o fenômeno físico acima, precisamos de modelos matemáticos para cada item presente na cena, assim como precisamos descrever a forma como eles interagem. A caracterização completa desse processo foge ao escopo desse trabalho e pode ser encontrada em livros de computação gráfica como [11], [46].

Usaremos um modelo de câmera ortográfica, posicionada no ponto o cuja direção de visão é dada pelo vetor w que é normal ao *plano de imagem* P. Paralelos ao plano P, temos o plano *near* N e o plano *far* F, que definem os limites de visão da câmera, isto é, representaremos na imagem somente objetos que estão posicionados entre os planos N e F (Figura 2.2 a). Em P tomamos uma base ortonormal $\{u, v\}$ e escolhemos o retângulo R, cujos lados são paralelos a u e v e cujo centro é o ponto o, como *tela virtual* (Figura 2.2 b). A projeção de R sobre os planos N e F definem o volume V, que chamamos de *volume de visão*. A cena contém uma fonte de luz pontual posicionada em l.



(a) (b)

Figura 2.2: Em (a) temos uma cena e a representação dos parâmetros da câmera. Em (b) o retângulo R que, junto dos planos N e F, define o volume de visão V.

Uma visualização da superfície S é uma função $I\colon R \to Q$, onde Q é um espaço de distribuição de cores. Caracterizamos I da seguinte forma: para cada ponto $c \in R$ definimos o *raio* $r_{c,w}(t) := c + tw$ com $t \in [t_i, t_f]$, $r_{c,w}(t_i) \in N$ e $r_{c,w}(t_f) \in F$ (Figura 2.3). Quando o raio não intercepta a superfície S, isto é $r_{c,w}([t_i, t_f]) \cap S = \varnothing$, definimos $I(c) = q_0$, onde $q_0 \in Q$ é uma cor arbitrária chamada *cor de fundo*. Quando o raio intercepta S, tomamos o menor valor $t_0 \in [t_i, t_f]$ tal que $r_{c,w}(t_0) \in S$. Chamamos de p_c o ponto $r_{c,w}(t_0)$ e de n_v o vetor normal a S no ponto p_c. Podemos definir $I(c) = M(p_c, n_c, l, w)$, onde a função M é o *modelo de iluminação* adotado.

Ao longo deste trabalho assumiremos o modelo de iluminação de Blinn–Phong. O processo de associar a um ponto uma cor, através do modelo de iluminação é chamado de *shading*.

2.3 Superfícies Implícitas

Estamos interessados em um tipo particular de superfícies chamadas de *superfícies implícitas*. Superfícies implícitas são definidas como o conjunto de

pontos que são soluções de uma equação da forma

$$f(p) = 0,\ p \in \mathbb{R}^3 \tag{2.1}$$

onde $f : \mathbb{R}^3 \to \mathbb{R}$ é uma função real.

Com essa classe de superfícies podemos reinterpretar o método de visualização da seguinte forma: para cada raio $r_{c,w}(t)$ procuramos o menor valor $t_0 \in [t_i, t_f]$ que é a solução da equação

$$g_c(t) := f(r_{c,w}(t)) = 0. \tag{2.2}$$

Caso ela exista temos que

$$p_c = r_{c,w}(t_0)$$

e, nos casos onde p_c não é ponto crítico f,

$$n_c = \frac{\nabla f(r_{c,w}(t_0))}{\|\nabla f(r_{c,w}(t_0))\|}.$$

Portanto, para uma configuração de cena dada, o problema de visualizar uma superfície implícita se resume a encontrar a menor raiz da equação (2.2). Esse problema, em geral, tem várias abordagens e podemos classificá–las em dois grupos: métodos analíticos e métodos numéricos.

As soluções analíticas para a equação (2.2) são limitadas a uma pequena classes de funções. No caso em que f é um polinômio de grau até 4 podemos achar suas raízes por radicais [1, 17].

Para uma classe mais abrangente de funções, como por exemplo polinômios de maior grau e funções transcendentes, devemos usar métodos numéricos. Esses métodos, em geral, são compostos de duas partes. A primeira é chamada de *isolamento da raiz* e consiste em encontrar um intervalo $[t_a, t_b] \subset [t_i, t_f]$ no qual exista apenas uma raiz de g_c. A segunda, chamada de *busca pela raiz*, é o processo pelo qual o intervalo $[t_a, t_b]$ é *refinado*, isto é, dado um valor $\varepsilon > 0$, encontramos um intervalo $[t_{a'}, t_{b'}] \subset [t_a, t_b]$ com $t_{b'} - t_{a'} < \varepsilon$ e que contém a raiz de g_c.

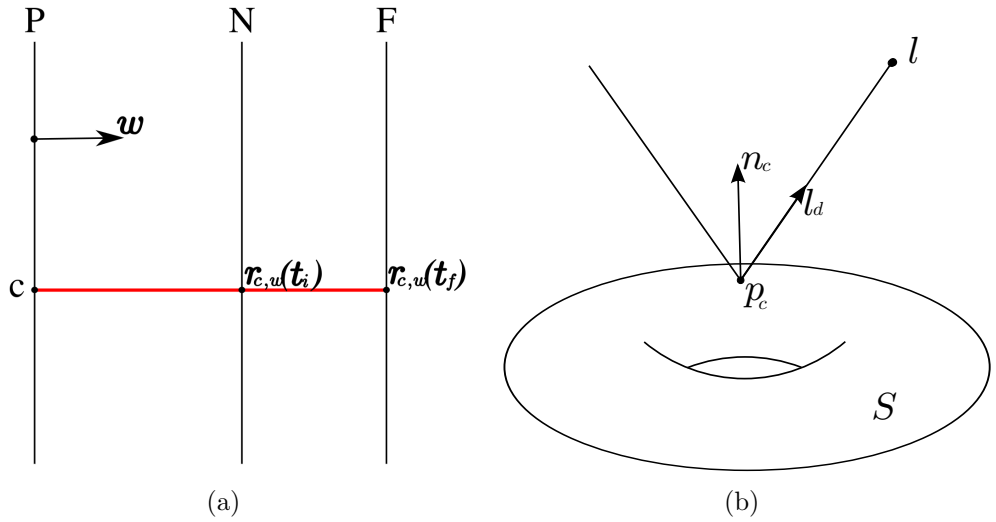


Figura 2.3: O item (a) mostra o raio relativo ao ponto c que parte do plano N na direção w e chega em F. No item (b) temos o ponto p_c em S e seu vetor normal n_c que são usados pelo modelo de iluminação para determinar a cor $I(c)$.

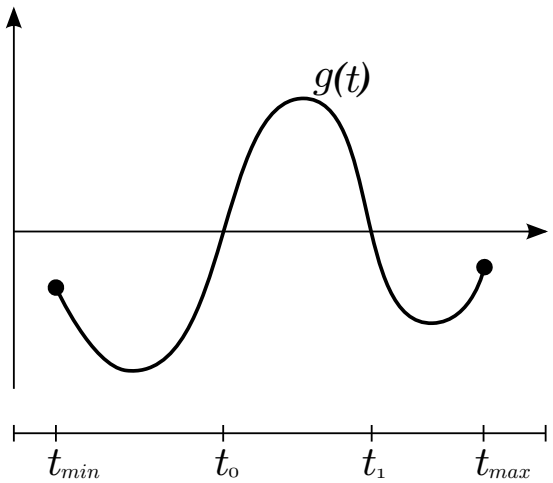


Figura 2.4: A função g_c é a composição do raio r e da função f. Procuramos encontrar a menor raiz de g_c, caso ela exista.

2.4 Discretização do Modelo

Nosso objetivo é implementar o modelo matemático descrito e, para uma dada cena, produzir uma imagem digital que representa I. Chamaremos esta imagem de $\hat{I}$ e denotaremos por $\hat{I}(i)$ o i-ésimo pixel de $\hat{I}$. Começamos supondo a existência de uma grade regular sobre R, onde cada elemento da grade é identificado com um pixel na imagem de saída. Como cada pixel é preenchido com uma única cor, precisamos escolher a cor que melhor representa os valores de I no elemento de R. Denotaremos por $r(i)$ a região em R correspondente ao pixel i. A hipótese de que cada região $r(i)$ corresponde a um pixel de $\hat{I}$ é feita para simplificar a exposição. Implicitamente estaremos assumindo que o processo de reconstrução de imagem a partir das amostras tomadas usa um núcleo do tipo caixa, ou núcleo de *Haar* [44].

Vamos usar o seguinte critério: se r é um elemento de R, então o pixel que representa r na imagem $\hat{I}$ deve ter a cor

$$q_r = \frac{\int_r I}{A(r)} \tag{2.3}$$

onde $A(r)$ é a área de r. Esse critério nos dá a *cor média* de I em r. Entretanto, para calcular o valor da equação (2.3), precisamos conhecer o valor de I em todos os pontos de r, o que é impossível do ponto de vista computacional, exceto nos casos mais simples. Portanto, precisamos adotar estratégias numéricas que nos permitam calcular o valor de q_r satisfatoriamente em um tempo finito.

Uma abordagem é a amostragem pontual: escolher um ponto c em r e usar $I(c)$ como valor de cor para o pixel. Como a escolha de c é arbitrária, podemos tomá–lo no centro da região r. Apesar de simples, essa forma de escolher o ponto produz resultados visualmente aceitáveis. Isso se justifica pelo fato de que, para uma função f suave, os valores para I não variam muito dentro de uma área pequena. Porém essa amostragem pontual produz um efeito conhecido como *aliasing*. Podemos encontrar uma descrição completa desse efeito em [9].

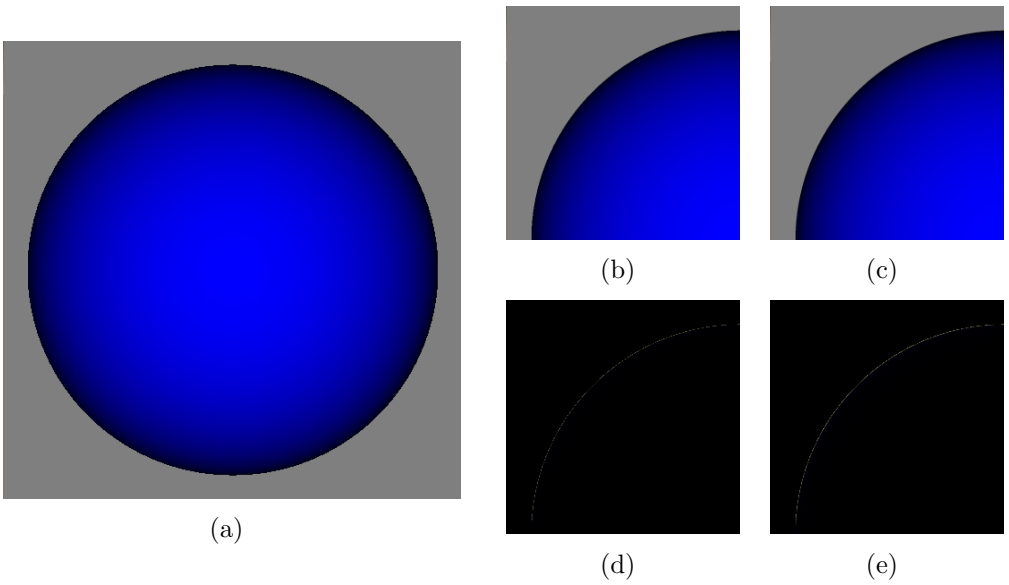
Na prática, uma imagem digital com *aliasing* apresenta artefatos que

evidenciam a discretização da imagem. Na Figura 2.5 temos a visualização de uma esfera e o efeito do *aliasing*. Estes artefatos são mais aparentes nas regiões mais próximas às bordas, já que nelas a variação da cor é maior.

A fim de atenuar o *aliasing* podemos tomar mais amostras da imagem I e combiná-las para formar o valor final da cor do pixel. De fato, ao subdividirmos r em partes menores $r = r_1 \cup r_2 \cup \cdots \cup r_k$ e em cada parte tomarmos um ponto $c_j \in r_j$ com $j = 1, \cdots, k$, podemos aproximar o valor da equação (2.3) por

$$q_r = \frac{I(c_1)A(r_1) + \cdots + I(c_k)A(r_k)}{A(r)}. \tag{2.4}$$

Esse processo é chamado de *super amostragem* e produz bons resultados: quanto mais amostras tomamos melhor representamos a cor média de r. A Figura 2.5 (b) a (d) mostra a mesma esfera com um número maior de amostras por pixel, o item (e) mostra a diferença entre o item (a) e (d).

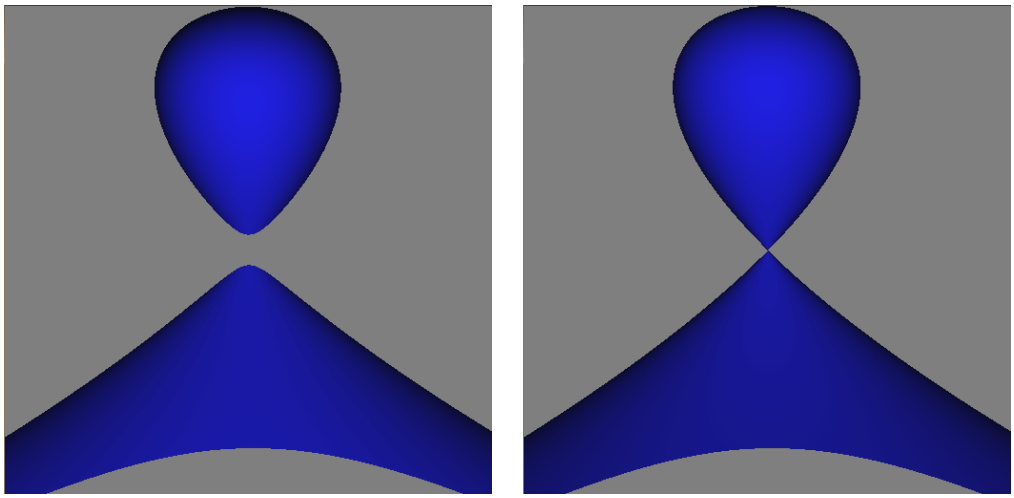


(a) (b) (c) (d) (e)

Figura 2.5: O item (a) mostra uma esfera em que cada pixel tem a cor de apenas um representante da região, é possível ver que nas bordas existem artefatos. No item (b) temos a parte superior esquerda usando 4 amostras, no item (c) com 16 amostras. Os itens (d) e (e) mostram a diferença entre os itens (b) e (c) para a mesma região no item (a).

2.5 Implementação

A implementação dos métodos de visualização de superfícies implícitas deve levar em conta os seguintes fatores: robustez, qualidade e eficiência. Um método *robusto* produz resultados que aproximam corretamente a geometria e a topologia da superfície. Na Figura 2.6 temos um exemplo de superfície visualizada usando um método não robusto. No quesito *qualidade*, procuramos produzir imagens com pouco *aliasing* e que sejam fiéis à iluminação da cena. Por fim, procuramos produzir resultados de forma eficiente, isto é, no menor tempo possível. No Capítulo 3 estudaremos vários métodos para visualizar superfícies implícitas e analisaremos a questão da robustez e da qualidade destes métodos. No Capítulo 4, veremos como esses métodos podem ser implementados de forma eficiente.



(a) (b)

Figura 2.6: No item (a) temos a superfície *Ding–Dong* visualizada por um método não robusto. No item (b) a mesma superfície visualizada por um método robusto.

Capítulo 3

Métodos

No capítulo anterior vimos que, para obter uma visualização de uma superfície implícita, é necessário resolver uma equação nos pontos da imagem virtual R de forma a calcular a cor das regiões r referentes aos pixels da imagem de saída $\hat{I}$. Neste capítulo estudaremos alguns métodos para resolver essas equações.

3.1 Introdução

O Algoritmo 3.1 mostra como obter a imagem $\hat{I}$ para uma dada função f. Para cada pixel i de $\hat{I}$ encontramos sua região $r(i)$ correspondente. Tomamos o ponto c, centro de $r(i)$, como ponto de amostragem e usamos um algoritmo, que chamaremos de *IsolaRaiz*, para encontrar o intervalo $J \subset [t_i, t_f]$ que isola a primeira raiz de g_c. Caso ele não exista (isto é, $J = \varnothing$) é porque o raio não atinge a superfície e definimos a cor de $\hat{I}(i)$ como a cor de fundo. Caso contrário, refinamos o intervalo J usando o algoritmo $RefinaRaiz$, de forma a encontrar uma aproximação para a menor raiz de g_c. Chamaremos essa aproximação da raiz de t_0. Por fim, definimos $\hat{I}(i)$ como a cor calculada no ponto da superfície correspondente a t_0, usando o modelo de iluminação.

Nas seções seguintes vamos procurar métodos para *IsolaRaiz* e *RefinaRaiz* e analisar suas propriedades. Estaremos particularmente interessados em métodos que possam ser implementados de forma eficiente. Analisaremos

esse aspecto no Capítulo 4.

Algoritmo 3.1 Algoritmo básico de visualização

```
para cada i faça
  c ← Centro(r(i))  ◇ Calcula o centro de r
  J ← IsolaRaiz(g, [t_i, t_f])
  se J ≠ ∅ então
    t_0 ← RefinaRaiz(g, J)
    Î(i) := Cor(t_0, r_{c,w}, f)  ◇ Calcula cor usando p_c, n_c
  se não
    Î(i) := q_0  ◇ Cor de Fundo
  fim se
fim para
```

3.2 Refinando Intervalos

Começaremos explorando uma situação simplificada. Suponhamos que para todos os pontos $c \in R$ a função $g_c(t) = f(r_{c,w}(t))$ é monótona. Nesse caso, para sabermos se g_c tem raiz, basta que comparemos seu sinal nos extremos do intervalo. Definimos $IsolaRaiz$ como no Algoritmo 3.2.

Algoritmo 3.2 *IsolaRaiz*

```
procedimento IsolaRaiz(g, [t_i, t_f])
  se g(t_i)g(t_f) ≤ 0 então
    retorne [t_i, t_f]
  se não
    retorne ∅
  fim se
fim procedimento
```

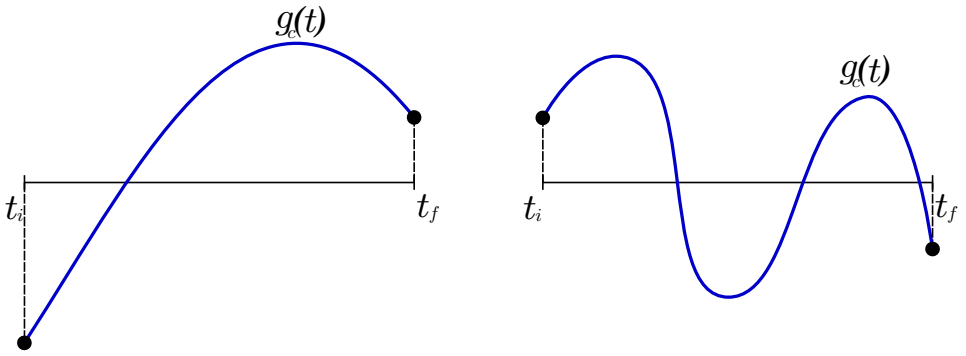
Para refinar o intervalo $[t_i, t_f]$, podemos usar o método de bisecção (Algoritmo 3.3). O algoritmo *Bisecção* recebe como argumento um intervalo $[t_i, t_f]$, uma função g_c e um valor de precisão ε. Podemos supor que $g_c(t_i) < 0$ e $g_c(t_f) > 0$, pois no caso contrário, poderíamos trocar os valores de t_i e de t_f. Essa possibilidade existe porque *Bisecção* não assume a ordem $t_i < t_f$. O algoritmo começa testando se o intervalo de entrada tem largura menor

que ε. Em caso a afirmativo, o intervalo de entrada é a solução procurada. Caso contrário, tomamos o ponto médio do intervalo e testamos g_c de forma a verificar em qual das metades do intervalo encontra–se a raiz. Fazemos isso verificando se g_c é nula ou troca de sinal nos extremos do intervalo $[t_i, m]$. Invocamos o algoritmo de forma recursiva para a metade do intervalo na qual sabemos existir a raiz. O algoritmo converge pois sempre recebe como entrada um intervalo que contém a raiz e para cada invocação a largura do intervalo é dividida pela metade. Portanto, após um número finito de passos a largura do intervalo testado será menor que ε.

Algoritmo 3.3 Algoritmo de Bisecção

procedimento *Bisecção*(g, $[t_i, t_f]$, ε)
  se $t_f - t_i < \varepsilon$ **então**
    retorne $\frac{t_i+t_f}{2}$ $\diamond$ *Ponto médio do intervalo solução.*
  fim se
  $m \leftarrow \frac{t_i+t_f}{2}$
  se $g(m) \geq 0$ **então**
    retorne *Bisecção*(g, $[t_i, m]$, ε)
  se não
    retorne *Bisecção*(g, $[m, t_f]$, ε)
  fim se
fim procedimento

É claro que a suposição inicial de que g_c é monótona é muito forte e poucas funções f satisfazem esses requisitos. Um exemplo, porém, são as funções afins que definem um plano. Podemos usar o método para funções mais gerais. De fato, não precisamos que g_c seja monótona em todo o intervalo e, em alguns casos, nem mesmo que tenha apenas uma raiz em $[t_i, t_f]$ (Figura 3.1 a e b). Esses exemplos ilustram dois casos onde *IsolaRaiz* reporta corretamente a existência de uma raiz no intervalo de entrada e, para os quais, o método de bisecção converge corretamente para a menor raiz. Na Figura 3.2 temos dois exemplos onde o método falha. No item (a), a função tem duas raízes e portanto *IsolaRaiz* falha ao testar a função usando somente seus valor nos extremos do intervalo. No item (b), temos uma função análoga à da Figura 3.1 (b) porém transladada. Nesse caso *IsolaRaiz* reporta corretamente a existência da raiz mas *Bisecção* converge para a maior raiz no intervalo.



(a) (b)

Figura 3.1: Dois casos onde *IsolaRaiz* reporta corretamente a existência de raízes e para os quais *Bisecção* converge para a menor raiz. Em (a) g_c não é monótona. Em (b) g_c tem 3 raízes.

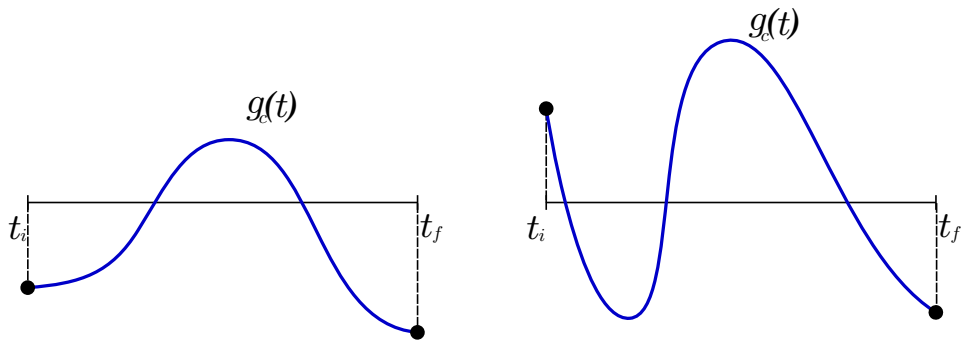
Precisamos melhorar *IsolaRaiz* de forma a detectar corretamente a existência de raízes em situações como as dos exemplos anteriores e, também, fazer com retorne intervalos onde possamos utilizar a *Bisecção* com garantias de que esta convirja para a menor raiz. Nas próximas seções veremos algumas maneiras de isolar raízes e analisaremos seus resultados.

3.3 Métodos por Amostragem

Nesta seção, veremos dois métodos usados para encontrar intervalos iniciais razoáveis. Esses métodos, apesar de não serem robustos, têm a vantagem de que podem ser facilmente implementados. E, como veremos no Capítulo 4, esses métodos se encaixam no modelo computacional das placas gráficas atuais.

3.3.1 Amostragem Uniforme

Consideremos o Algoritmo 3.4. Nele, o intervalo de entrada $[t_i, t_f]$ é sequencialmente amostrado em intervalos regulares, cada amostra é comparada com



(a) (b)

Figura 3.2: Em (a), *IsolaRaiz* não detecta as raízes. Em (b) g_c tem 3 raízes que são detectadas por *IsolaRaiz*, porém *Bisecção* converge para a raiz errada.

a amostra imediatamente anterior de forma a encontrar duas amostras consecutivas onde a função g_c muda de sinal (Figura 3.3). Caso nenhuma troca de sinais seja encontrada consideramos que não existem raízes sobre o raio. Caso contrário, tomamos o intervalo entre as duas amostras como base para a busca da raiz pelo método *Bisecção*.

Algoritmo 3.4 Amostragem Uniforme

procedimento *AmostragemUniforme*(g, $[t_i, t_f]$, ε)
  Calculamos o menor inteiro N tal que $\frac{1}{\varepsilon} < N$.
  Dividimos $[t_i, t_f]$ em N intervalos iguais $[a_i, b_i]$, $i = 1, \cdots, N$.
  para cada i de 1 a N **faça**
    se $g(a_i)g(b_i) \leq 0$ **então**
      retorne $[a_i, b_i]$
    fim se
  fim para
  retorne $\varnothing$
fim procedimento

É fácil encontrar exemplos onde o método falha (Figura 3.4). No primeiro caso (item a), se escolhermos um valor para ε muito grande corremos o risco de que entre duas amostras exista um número par de raízes de g_c, o que

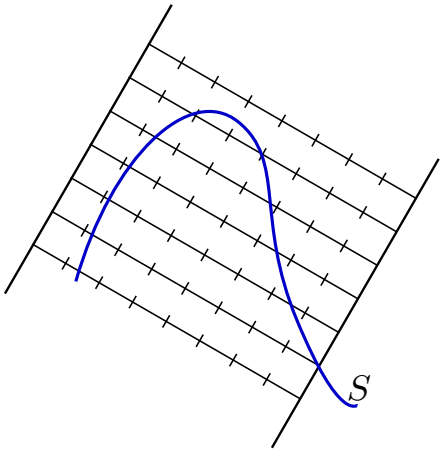
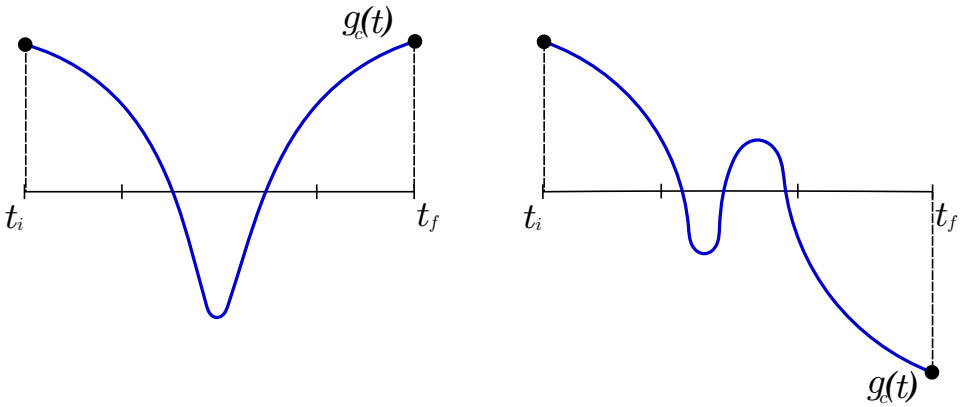


Figura 3.3: Amostragem Uniforme do raio.



(a) (b)

Figura 3.4: Dois casos de falha do método. Em (a) temos um número par de raízes. Em (b), não sabemos para qual raiz converge a bisecção.

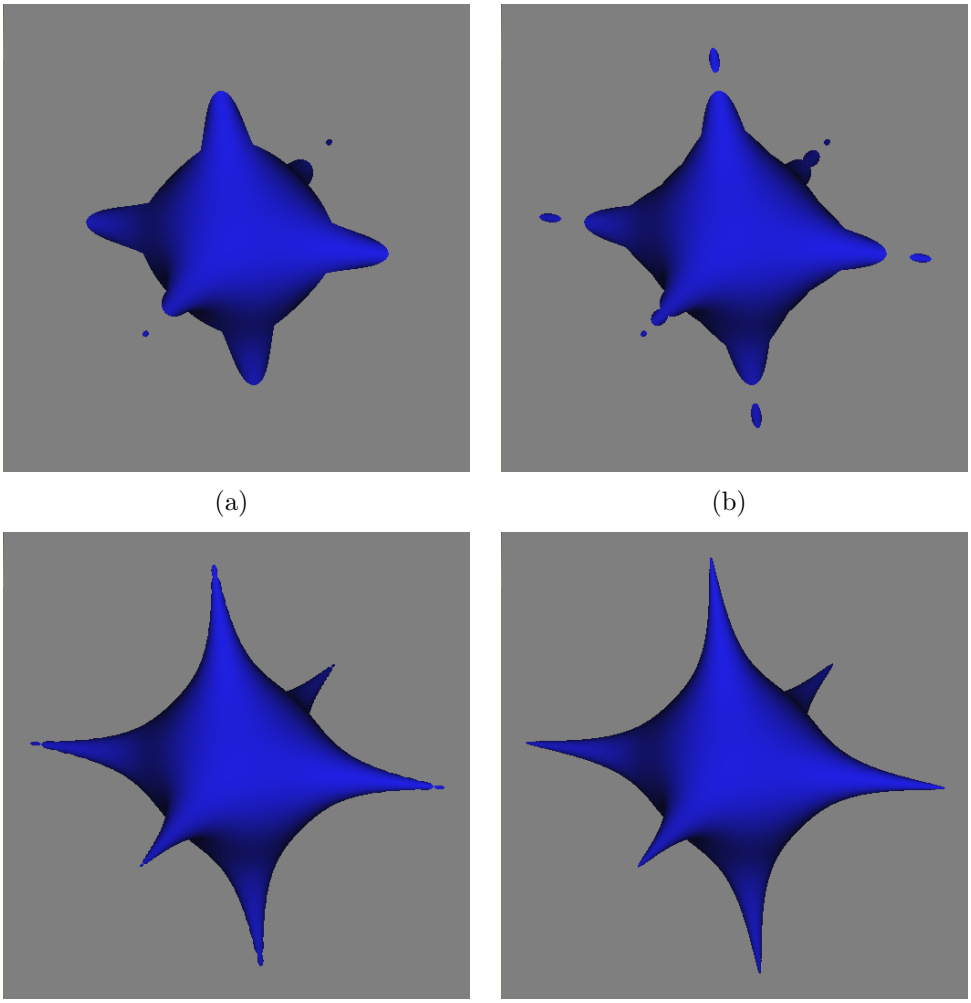
faria com que intervalo fosse descartado, mesmo contendo raízes. Também é possível que tenhamos um número impar de raízes (b), maior que ou igual a 3 e nesse caso não saberíamos para qual raiz o processo de bisecção irá convergir ([10]). Outra situação problemática é quando temos uma única raiz que não é detectada.

Uma forma de melhorar o método é diminuir a distância entre as amostras, de forma a evitar casos como os acima mencionados. De fato, quanto mais amostras são tomadas sobre o intervalo, melhor são preservadas as características da superfície a ser visualizada. Na Figura 3.5, podemos ver um exemplo de como o número de amostras altera a topologia e a forma da superfície visualizada. Essa melhora na qualidade e na robustez do método vem contra a eficiência. Veremos como melhorar esses resultados usando a informação da função f nos pontos de amostragem.

3.3.2 Amostragem Adaptativa

A amostragem uniforme utiliza uma distância fixa entre as amostras durante todo o processo e essa distância tem que ser pequena o suficiente para detectar todas as raízes. Os piores casos estão perto da silhueta da superfície (Figura 3.6). Podemos tomar distâncias maiores longe da superfície, mas gostaríamos de poder diminuir essa distância ao chegarmos mais perto das raízes de f. O Algoritmo de Amostragem Adaptativa toma distâncias que variam conforme a proximidade da superfície. Para isso precisamos de uma medida de proximidade entre a superfície e uma amostra sendo tomada sobre o raio g_c.

O Algoritmo 3.5 se baseia no valor de f na amostra para determinar a proximidade de uma raiz à superfície. Da mesma forma, podemos usar o valor $|\langle \nabla f(r_{c,w}), r'_{c,w} \rangle|$ como medida de proximidade da silhueta ([43]). Comparamos esses valores com as constantes pré–definidas τ_1, τ_2 e μ. O valor ótimo dessas constates é diferente para cada superfície; em nosso trabalho optamos por encontrar bons valores empiricamente. Trabalhos como [19] e [14] sugerem formas de encontrar bons valores, como por exemplo constantes de



(a) (b)

(c) (d)

Figura 3.5: Superfície *Distel* visualizada pelo método da Amostragem Uniforme. Conforme diminuímos o tamanho do passo, vemos que a detecção de raízes aumenta. O que melhora a qualidade da visualização. No item (a) foram tomadas 8 amostras por raio, no item (b) 16, no item (c) 64 e no item (d) 256.

Lipschitz de f.

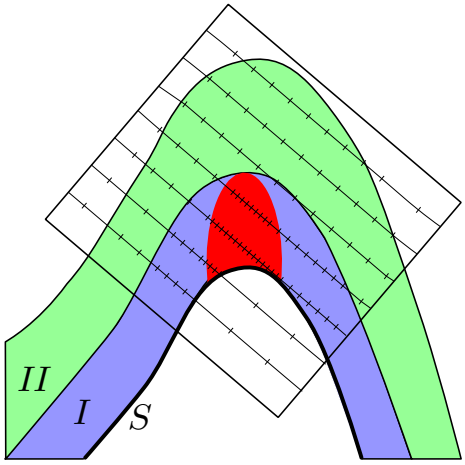


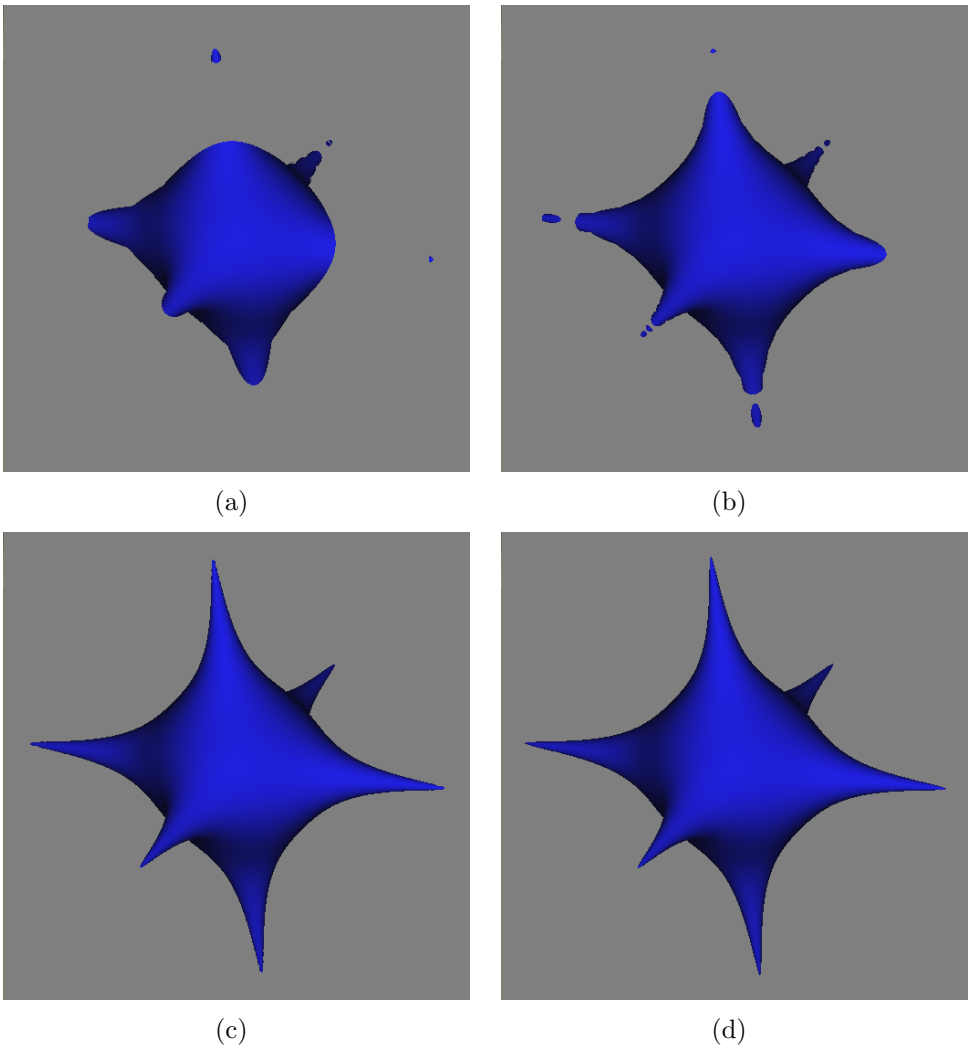
Figura 3.6: A distância entre as amostras muda conforme o valor da função g_c sobre a amostra sendo testada. Os pontos perto da silhueta são mais difíceis de serem detectados, por isso precisamos de uma menor distância entre as amostras.

O algoritmo começa com uma largura para intervalo entre as amostras de tamanho fixo $2h$ e, baseado nos valores de τ_1, τ_2 e μ, diminui a largura de forma a melhor amostrar áreas onde supomos encontrar raízes. Um resultado deste algoritmo pode ser visto na Figura 3.7.

3.4 Isolando Raízes com IA

Uma família de métodos robustos para isolar raízes são os métodos intervalares. A ideia fundamental por trás desses métodos é a de que podemos expressar variáveis numéricas como intervalos. A partir disto, podemos estender funções reais para *funções intervalares*. Funções intervalares recebem como argumento intervalos e retornam intervalos.

Idealmente, gostaríamos de caracterizar a *extensão intervalar* de uma função real f, como a função intervalar F que goza da seguinte propriedade: dado um intervalo $[a, b]$ contido no domínio de f, temos que $[A, B] := F([a, b]) = f([a, b])$, isto é, F calcula a imagem de $[a, b]$ por f. Dessa forma,



(a) (b)

(c) (d)

Figura 3.7: Superfície *Distel* visualizada pelo método da Amostragem Adaptativa. No item (a) foram tomadas 8 amostras por raio, no item (b) 16, no item (c) 64 a no item (d) 256.

Algoritmo 3.5 Amostragem Adaptativa

procedimento *AmostragemAdaptativa*(g, $[t_i, t_f]$, ε)
  Calculamos o menor inteiro N tal que $\frac{2}{\varepsilon} < N$;
  Definimos $h = \frac{1}{N}$ e $t = t_i$;
  enquanto $t < t_f$ **faça**

$$s = \begin{cases} h/4 & \text{se } |g(t)| < \tau_1 \text{ e } |g'(t)| < \mu \\ h/2 & \text{se } |g(t)| < \tau_1 \\ h & \text{se } |g(t)| < \tau_2 \\ 2h & \text{caso contrário} \end{cases}$$

    se $g(t)g(t+s) \leq 0$ **então**
      retorne $[t, t_s]$
    fim se
    $t \leftarrow t + s$
  fim enquanto
  retorne $\varnothing$
fim procedimento

para verificar se f tem raízes em $[a, b]$, bastaria verificar se $0 \in [A, B]$. Em [30] Moore introduz a *Aritmética Intervalar* (AI), como forma de criar extensões intervalares para funções racionais. Porém, como veremos adiante, essas extensões não serão exatas. O que os métodos da AI nos garantirão é a propriedade de inclusão, isto é, sob as hipóteses anteriores de f, teremos $f([a, b]) \subset F([a, b])$.

3.4.1 IA e Convergência

Começamos por definir uma extensão intervalar para o corpo dos números reais.

Definição 1. *Dados* $[a_1, b_1]$, $[a_2, b_2] \subset \mathbb{R}$ *e* $\alpha \in \mathbb{R}$, *definimos:*

- $[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$;
- $[a_1, b_1] * [a_2, b_2] = [\min S, \max S]$, *onde* $S = \{a_1a_2, a_1b_2, b_1a_2, b_1b_2\}$;
- $\alpha[a_1, b_1] = [\min\{\alpha a_1, \alpha b_1\}, \max\{\alpha a_1, \alpha b_1\}]$;

- *Se* $0 \notin [a_1, b_1]$, *então* $1/[a_1, b_1] = [1/b_1, 1/a_1]$.

Com essas operações podemos estender uma função racional $q(x) = \frac{p_1(x)}{p_2(x)}$, onde p_1 e p_2 são polinômios, para uma função intervalar $Q(X) = \frac{P_1(X)}{P_2(X)}$, onde $X = [x_1, x_2] \in \mathbb{R}$ e P_1, P_2 são extensões intervalares de p_1 e p_2 respectivamente. Note que para todo $X = [x_1, x_2] \subset \mathbb{R}$ vale a inclusão $q(X) \subset Q(X)$, porém podemos não ter a igualdade. Isto acontece porque a AI nos dá uma sobre–estimativa para $q(X)$. Por exemplo, tomando $q(x) = x^2$ e $X = [-1, 1]$ temos $q(X) = [0, 1]$ porém $Q(X) = X * X = [-1, 1]$ (Figura 3.8).

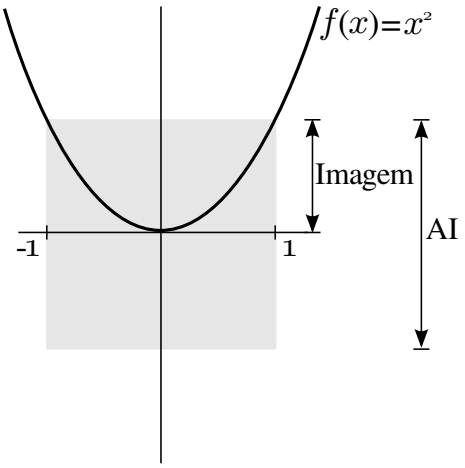


Figura 3.8: A extensão intervalar de um polinômio, e por consequência de uma função racional, pode nos dar estimativas muito maiores que o resultado exato.

Isto implica que a versão intervalar de uma função nos diz quando **não** há raízes em um intervalo. Isto é, dado $X \subset \mathbb{R}$, vale que $0 \notin Q(X) \implies q(x) \neq 0$, $\forall x \in X$. Porém a recíproca não é necessariamente verdadeira. Apesar disto, a extensão intervalar de uma função racional goza de uma propriedade importante: se uma função racional q está definida em um intervalo X_0, então existe uma constante L tal que para todo $X \subset X_0$ temos $\omega(Q(X)) \leq L\omega(X)$, onde $\omega(X)$ é a largura do intervalo X. Esta propriedade nos garante que, conforme diminuímos a largura de intervalo onde estamos tomando a estimativa para a imagem de f, temos estimativas mais precisas. Formalmente $X_n \to \{x_0\} \implies Q(X_n) \to \{q(x_0)\}$.

3.4.2 Algoritmo de Mitchell

Em [29] encontramos a descrição de um algoritmo recursivo que usa a extensão intervalar da função $g_c(t)$ e de sua derivada $g_c'(t)$ para isolar a menor raiz.

Algoritmo 3.6 Algoritmo de Mitchell

procedimento *Mitchell*(g, G, G', $[t_i, t_f]$, ε)
  se $0 \in G([t_i, t_f])$ **então**
    se $0 \notin G'([t_i, t_f])$ **então**
      se $g(a)g(b) \leq 0$ **então**
        retorne $[t_i, t_f]$
      se não
        retorne $\varnothing$ $\diamond$ *Não há raízes*
      fim se
    se não
      $J \leftarrow$ *Mitchell*$(g, G, G', [t_i, \frac{t_i+t_f}{2}])$
      se $J \neq \varnothing$ **então**
        retorne J
      se não
        retorne *Mitchell*$(g, G, G', [\frac{t_i+t_f}{2}, t_f])$
      fim se
    fim se
  se não
    retorne $\varnothing$
  fim se
fim procedimento

O Algoritmo 3.6 começa com o cálculo de G_c e G_c', que são as versões intervalares de g_c e g_c' respectivamente. Em seguida, tomamos o intervalo de interesse $[t_i, t_f]$ e testamos $G_c([t_i, t_f])$. Se $0 \notin G_c([t_i, t_f])$ certamente não temos raízes em $[t_i, t_f]$ e o algoritmo termina. Caso contrário g_c pode ter raízes em $[t_i, t_f]$. Em seguida verificamos se $0 \notin G_c'([t_i, t_f])$ pois nesse caso g_c é monótona em $[t_i, t_f]$ e um teste simples de mudança de sinal nos extremos do intervalo nos assegura a existência ou não existência de raízes (portanto $g(a)g(b) \leq 0$ nos dirá que existe e é única a raiz em $[a, b]$ ou $g(a)g(b) > 0$ que não existem raízes em $[a, b]$). Finalmente, no caso onde $0 \in G_c'([a, b])$ executamos o mesmo algoritmo de forma recursiva em $[t_i, \frac{a+b}{2}]$ e depois em

$[\frac{a+b}{2}, t_f]$. O algoritmo acaba no caso em que uma raiz é isolada ou quando todo o intervalo inicial foi testado. Note que, como estamos interessados na menor raiz, executamos a recursão primeiramente na metade inferior do intervalo atual. Outro ponto importante é o fato de que a subdivisão recursiva do intervalo nos leva a testar intervalos cuja largura converge a zero, o que garante que o algoritmo isola a menor raiz.

Como estamos interessados em algoritmos que possam ser implementados, precisamos estabelecer um limite para a recursão, de forma a impedir que o algoritmo execute por um tempo indeterminado. Para isso podemos arbitrar um valor $\varepsilon > 0$ e estabelecer que para intervalos com largura menor que ε executaremos diretamente o teste da troca de sinais. Uma observação interessante é que no pior caso o algoritmo de Mitchell degenera para uma amostragem regular de passo ε. Essa observação nos permitirá no Capítulo 4 comparar o desempenho dos métodos.

Antes de podermos usar esse algoritmo para detectar a intersecção de raios com superfícies implícitas precisamos entender como nossas funções $g_c(t) = f\big(r_{c,w}(t)\big)$ são estendidas para suas representações intervalares. Estas extensões estão diretamente relacionadas à função f e pela forma como esta é representada em nossa implementação. Por exemplo, se f é um polinômio em 3 variáveis, então $g_c(t)$ também é um polinômio e podemos estendê–lo usando as operações aritméticas intervalares. Outra possibilidade é que tenhamos f do tipo *caixa preta*, isto é, não conhecemos a implementação de f apenas podemos avaliá–la em pontos do espaço. Nesse caso precisamos contar com as versões intervalares F e ∇F, de f e ∇f respectivamente.

A exemplo de funções racionais em uma variável, as funções racionais de várias variáveis também contam com uma extensão intervalar. No primeiro caso estendemos números a intervalos, no segundo estendemos a noção de vetor pela noção de bloco (ou vetor intervalar). Por exemplo, se $f(x, y, z) = xy + z$ sua versão intervalar será $F(X, Y, Z) = XY + Z$ onde $X = [x_1, x_2]$, $Y = [y_1, y_2]$ e $Z = [z_1, z_2]$. Vale o esperado: $f(X, Y, Z) \subset F(X, Y, Z)$, ou seja, F nos dá limites para f em todos os pontos de um bloco $X \times Y \times Z \subset \mathbb{R}^3$.

Dessa forma para um raio $r_{c,w}(t)$, temos que $g_c(t) = f\big(r_{c,w}(t)\big)$ se estende para $G_c(T) = F\big(R_{c,w}(T)\big)$ onde $R_{c,w}(T)$ é o bloco que contém $r_{c,w}(t)$ $\forall t \in$

$[t_i, t_f]$ (Figura 3.9). De maneira análoga estendemos $g'_c(t) = \langle \nabla f\big(r_{c,w}(t)\big), w\rangle$ para $G'_{c,w}(T) = \langle \nabla F\big(R_{c,w}(t)\big), w\rangle$. Note que $\nabla F(X, Y, Z)$ nos dá um bloco que contém todos os valores de $\nabla f(x, y, z), (x, y, z) \in X \times Y \times Z$.

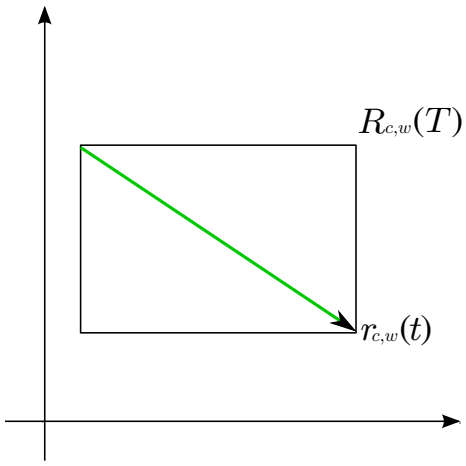


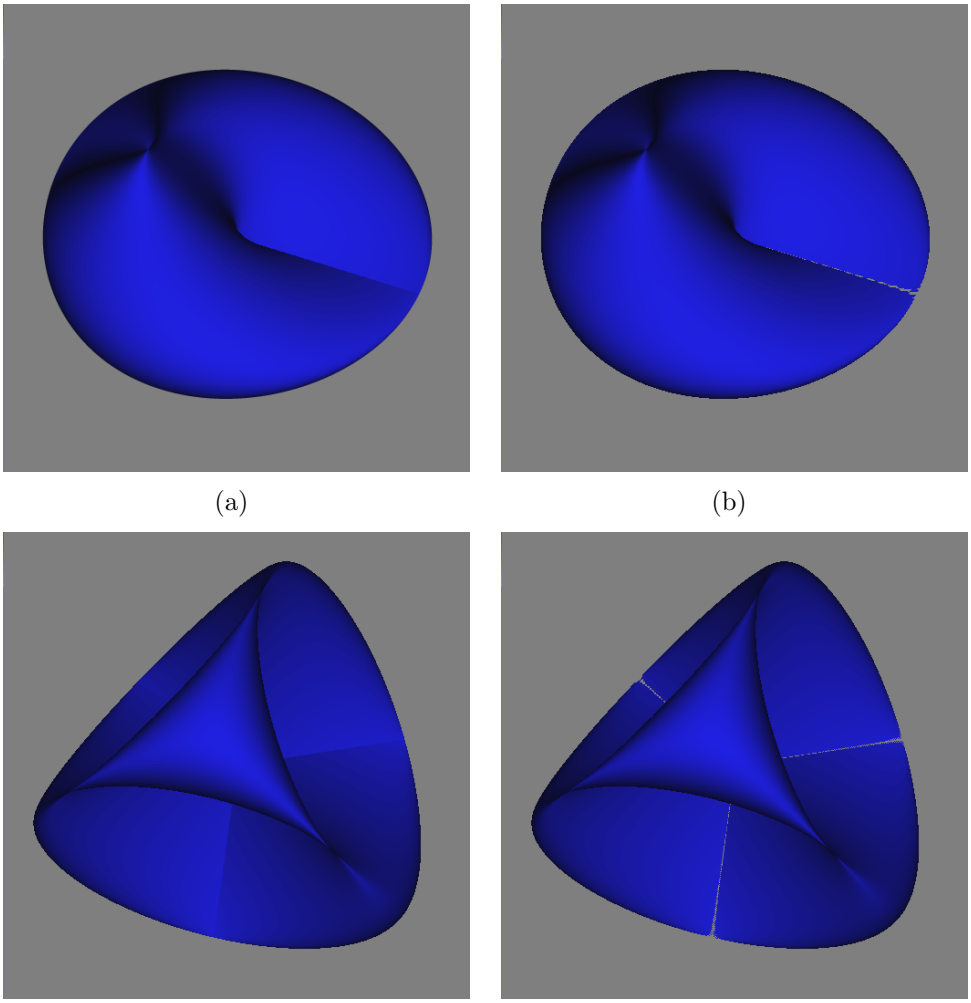
Figura 3.9: Versão intervalar do raio $r_{c,w}$.

O algoritmo de Mitchell, quando executado sem limites para a recursão, é robusto pois sempre converge para a primeira raiz. Quando adicionamos o limite para a recursão através do teste da largura do intervalo nós perdemos essa garantia, e o resultado final é o mesmo que o encontrado para o algoritmo de amostragem regular numa escala de largura equivalente. No Capítulo 4 discutiremos o desempenho de cada método e veremos que, para funções que exigem uma amostragem fina nos métodos por amostragem, o algoritmo de Mitchell tem desempenho superior para o mesmo resultado final.

Na Figura 3.10 vemos a comparação entre os resultados obtidos usando o Algoritmo de Mitchell e a Amostragem Uniforme.

3.5 Bisecção Intervalar

Como vimos anteriormente, ao diminuirmos a largura do intervalo no qual estamos testando f à procura de raízes, F nos dá estimativas melhores. De fato, nossas estimativas são proporcionais à largura do intervalo sendo



(a) (b)

(c) (d)

Figura 3.10: Exemplos de visualização usando o Algoritmo de Mitchell. No item (a) temos a superfície *Crosscap* com precisão total. No item (b) a mesma superfície com precisão $\varepsilon = 2^{-9}$. Analogamente, nos itens (c) e (d) vemos a superfície *Steiner*.

testado. Podemos usar esse fato para criar o Algoritmo 3.7 que é mais simples que o Algoritmo 3.6.

Algoritmo 3.7 Bisecção Intervalar

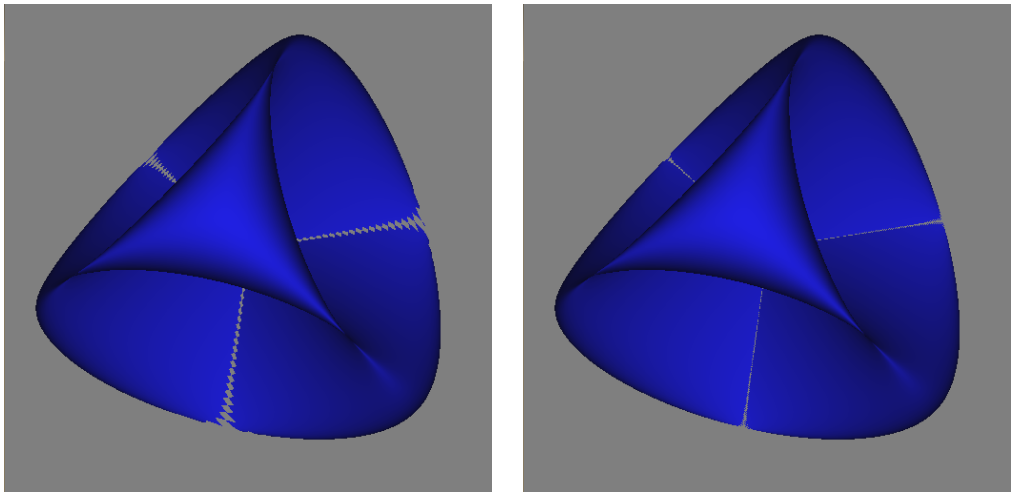
procedimento *BisecçãoIntervalar*(g, G, $[t_i, t_f]$, ε)
  se $0 \in G([a,b])$ **então**
    se $t_f - t_i < \varepsilon$ **então**
      retorne $[t_i, t_f]$
    se não
      $J \leftarrow$ *BisecçãoIntervalar*$(g, G, [t_i, \frac{t_i+t_f}{2}])$
      se $J \neq \varnothing$ **então**
        retorne J
      se não
        retorne *BisecçãoIntervalar*$(g, G, [\frac{t_i+t_f}{2}], t_f])$
      fim se
    fim se
  se não
    retorne $\varnothing$ $\diamond$ *Não há raiz em* $[t_i, t_f]$
  fim se
fim procedimento

Começamos por arbitrar que em um intervalo $[t_i, t_f]$ com largura menor que ε, $0 \in G_c([a,b]) \Leftrightarrow 0 \in g_c([a,b])$. Isto é, para intervalos pequenos assumimos que a extensão intervalar de g_c nos dá resultados suficientemente precisos. Caso este teste detecte raízes para um intervalo maior que ε, o algoritmo é executado recursivamente, primeiro para a metade inferior do intervalo de entrada e, caso nesse não haja raízes, para a metade superior. O algoritmo acaba ao encontrar um intervalo pequeno onde supomos existir raízes ou depois de testar todo o intervalo inicial.

Esse algoritmo pode ter duas saídas possíveis: a indicação de que não existem raízes ou um intervalo de largura menor que ε. No primeiro caso, temos garantias de que realmente não existem raízes em $[a,b]$. Já no segundo caso, devido à imprecisão da AI, o intervalo pode não conter raiz alguma, o que nos leva a um erro. Porém, como veremos nos exemplos (Figura 3.11), para valores de ε suficientemente pequenos, temos um resultado perceptualmente aceitável.

Como estamos arbitrando um valor pequeno para ε e não temos garantias de que o intervalo de saída contém uma raiz, podemos simplesmente não executar o método de bisecção e tomar o ponto médio do intervalo de saída como estimativa para a raiz. Tal escolha se justifica pelos seguintes fatos. Se o intervalo de saída não contém nenhuma raiz então a bisecção converge para o início do intervalo, o que é igualmente arbitrário. Todo método de bisecção tem uma precisão arbitrada e caso tenhamos escolhido ε suficientemente pequeno podemos estar perto de uma raiz.

A Figura 3.11 mostra o resultado do algoritmo para alguns valores de ε. Note que conforme diminuímos o valor para ε melhores são os resultados em relação à imagem de referência.



(a) (b)

Figura 3.11: Superfície *Distel* usando o Algoritmo de Bisecção Intervalar. No item (a) com $\varepsilon = 2^{-9}$ e no item (b) com $\varepsilon = 2^{-11}$.

Capítulo 4

Implementação

Nos capítulos anteriores vimos como obter uma visualização de uma superfície implícita. Para isso utilizamos métodos por amostragem e métodos intervalares. Neste capítulo vamos abordar a implementação destes métodos em CPU e GPU, assim como comparar seus desempenhos.

Nosso objetivo, inicialmente, é obter uma imagem para uma dada cena. Nos testes que faremos a seguir essa imagem terá 512×512 pixels de resolução. Escolhemos algumas superfícies para comparar o desempenhos dos métodos; acreditamos que elas apresentam uma variação de complexidade visual e computacional que nos permite ilustrar o desempenho de cada método. Uma lista mais completa pode ser vista no Capítulo 8.

4.1 CPU

Implementamos os métodos em C++ e os executamos em um Pentium Xeon 2.0 GHz. Começaremos verificando o desempenho dos métodos por amostragem. Em cada teste, a etapa de busca pela raiz será realizada por bisecção usando 16 iterações.

4.1.1 Métodos por Amostragem

Sabemos que a qualidade do resultado para esses métodos depende diretamente da quantidade de amostras tomadas sobre o raio. Veremos que a

quantidade de amostras também influencia no desempenho. Na Tabela 4.1 vemos o tempo gasto para a visualização das superfícies de teste para uma quantidade variável de amostras. Vemos que existe um correlação clara entre o número de amostras e o desempenho.

Amostragem Uniforme						
Nº de Amostras:	16	32	64	128	256	512
Esfera[2]	143.97	178.21	234.09	345.36	567.07	1004.77
Dingdong[3]	119.51	179.42	303.17	527.83	960.10	1853.57
Tangle[4]	211.75	306.49	471.89	836.19	1459.61	2867.18
Heart[6]	170.49	244.99	400.81	720.89	1340.50	2622.67
Chmutov[8]	227.78	324.45	514.57	874.56	1604.50	3066.72

Amostragem Adaptativa						
Nº de Amostras:	16	32	64	128	256	512
Esfera[2]	459.86	812.98	1479.75	2830.18	5540.70	11066.60
Dingdong[3]	685.65	1333.25	2628.32	5214.35	10362.50	20685.90
Tangle[4]	212.44	303.25	507.69	890.78	1698.90	3338.19
Heart[6]	333.98	614.62	1183.90	2368.18	4708.58	9206.11
Chmutov[8]	238.82	353.98	576.54	1048.20	1970.21	3888.35

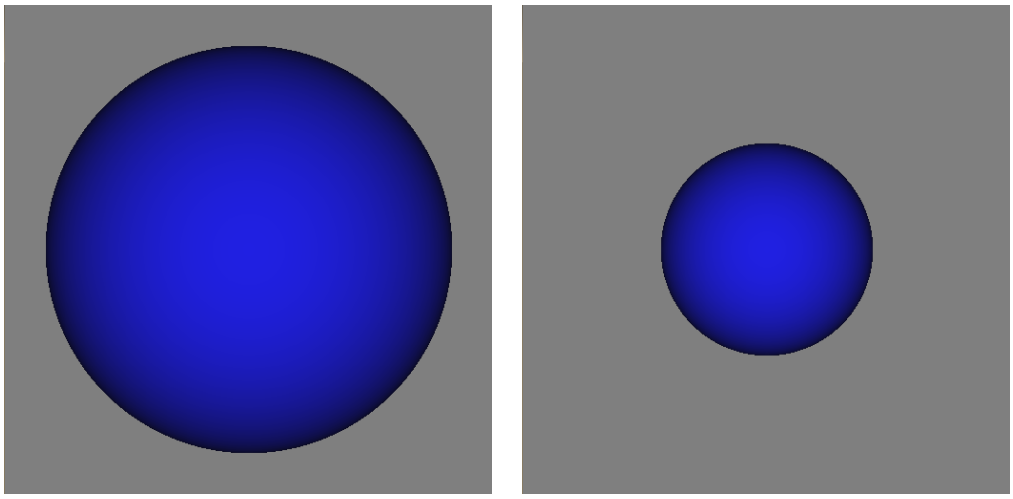
Tabela 4.1: Desempenho dos algoritmos de Amostragem (Uniforme e Adaptativa) executados em CPU com uma única *thread*. Na primeira coluna, ao lado no nome, temos o grau da superfície algébrica. Nas outras colunas temos o tempo gasto para realizar a visualização em milissegundos.

Na Tabela 4.2, vemos o comportamento do algoritmo quando variamos a área que a superfície ocupa na imagem (Figura 4.1). Note que ao diminuirmos o tamanho relativo da superfície, aumentamos o tempo para realizar a visualização. Isto é devido ao comportamento do algoritmo nos raios onde não existem raízes. Nestes casos o algoritmo toma e compara todas as amostras, o que é o pior caso possível.

Pixels(%)	Tempo(ms)
78.54	354.28
54.54	449.97
40.08	518.07
30.69	542.45

Pixels(%)	Tempo(ms)
24.24	581.07
19.63	579.49
16.22	589.80
13.63	597.37

Tabela 4.2: Amostragem Uniforme. Conforme diminuímos a área relativa da superfície pior é o desempenho.



(a) (b)

Figura 4.1: *Esfera*. Do item (a) para o (b) mudamos a áreas que a superfície ocupa relativa à imagem.

4.1.2 Métodos Intervalares

Vamos verificar o desempenho para os dois métodos intervalares vistos anteriormente: Mitchell e Bisseção Intervalar. Só podemos comparar o desempenho de dois métodos quando estes produzem resultados semelhantes. Como vimos no Capítulo 3, podemos arbitrar um valor $\varepsilon > 0$ tal que, para um intervalo com largura menor que ε, os métodos intervalares usam o critério da troca de sinais para testar a existência de raízes. Este critério garante que, para um método intervalar que subdivide recursivamente o intervalo de entrada N vezes, tenhamos um resultado equivalente a um métodos de amostragem com 2^N pontos. As Tabelas 4.3 e 4.4 mostram o desempenho dos métodos intervalares, notemos que a profundidade máxima indicada é referente ao número máximo de divisões recursivas do métodos.

Um dos motivos pelos quais os métodos intervalares superam os métodos por amostragem é que os intervalos referentes às partes do raio que estão longe da superfície são descartados logo nas primeiras etapas do processo. Podemos visualizar como esses algoritmos descartam intervalos nas Figuras 4.2 e 4.3. Nelas são mostradas, ao lado da imagem de referência, a profundidade máxima que o algoritmo de isolar raízes atingiu antes de descartar o

Desempenho do Algoritmo de Mitchell						
Profundidade Máxima:	4	5	6	7	8	9
Esfera[2]	169.07	172.03	172.80	169.56	173.50	172.42
Dingdong[3]	195.78	196.60	202.59	198.06	198.17	198.24
Tangle[4]	1389.09	1630.50	1779.65	1860.18	1895.88	1914.68
Heart[6]	754.46	809.82	856.63	861.44	867.36	873.45
Chmutov[8]	1569.28	1960.12	2398.54	2765.44	3168.45	3598.22

Tabela 4.3: Desempenho (ms) do algoritmo de Mitchell executado em CPU com uma única *thread*.

Desempenho da Bisseção Intervalar						
Profundidade Máxima:	4	5	6	7	8	9
Esfera[2]	222.33	258.25	279.84	304.71	331.21	356.58
Dingdong[3]	181.75	203.49	216.81	233.48	256.06	264.95
Tangle[4]	1201.17	1547.12	1946.86	2250.91	2574.32	2906.19
Heart[6]	340.23	367.33	409.53	428.44	456.86	482.08
Chmutov[8]	1004.97	1307.11	1688.58	2177.20	2821.55	3617.57

Tabela 4.4: Desempenho (ms) do algoritmo da Bisecção Intervalar executado em CPU com uma única *thread*.

raio ou retornar um intervalo para a bisecção. É possível notar que, para as áreas afastadas da superfície, os raios são descartados rapidamente.

Podemos agora verificar como estes algoritmos se comportam quando a área que superfície ocupa da imagem muda. Nas Tabelas 4.5 e 4.6, temos o tempo gasto para visualizar uma esfera em relação a área que ela ocupa na imagem. Vemos que, ao contrário dos métodos de amostragem, quanto menor é a área ocupada melhor é o desempenho.

Uma diferença importante entre as Figuras 4.2 e 4.3 aparece nos pixels sobre a superfície. Nestes casos a Bisecção Intervalar executa todos os passos e sempre atinge a profundidade máxima. Já o Algoritmo de Mitchell pode encontrar intervalos onde a função é monótona em etapas anteriores. Vemos que isso ocorre nas áreas afastadas da silhueta da superfície.

Esta diferença entre os algoritmos tem impacto no desempenho. Em algumas superfícies o Algoritmo de Mitchell consegue isolar, logo nas primeiras iterações, intervalos onde a função g_c é monótona. Nestes casos seu desempenho é superior à Bisecção Intervalar. Um exemplo é a superfície *Dingdong* na Figura 4.4 (a). Já nos casos em que Mitchell tem de executar muitas

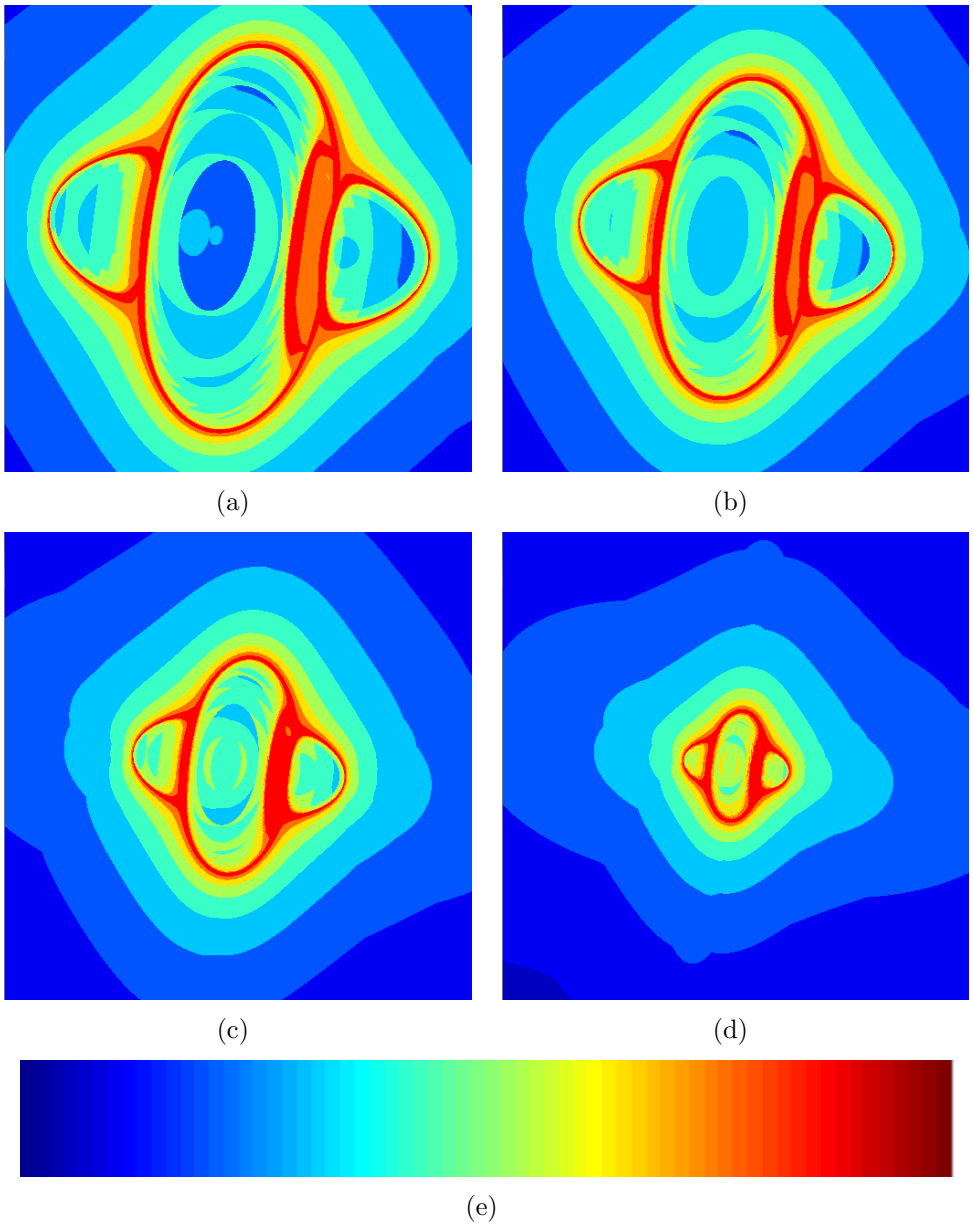
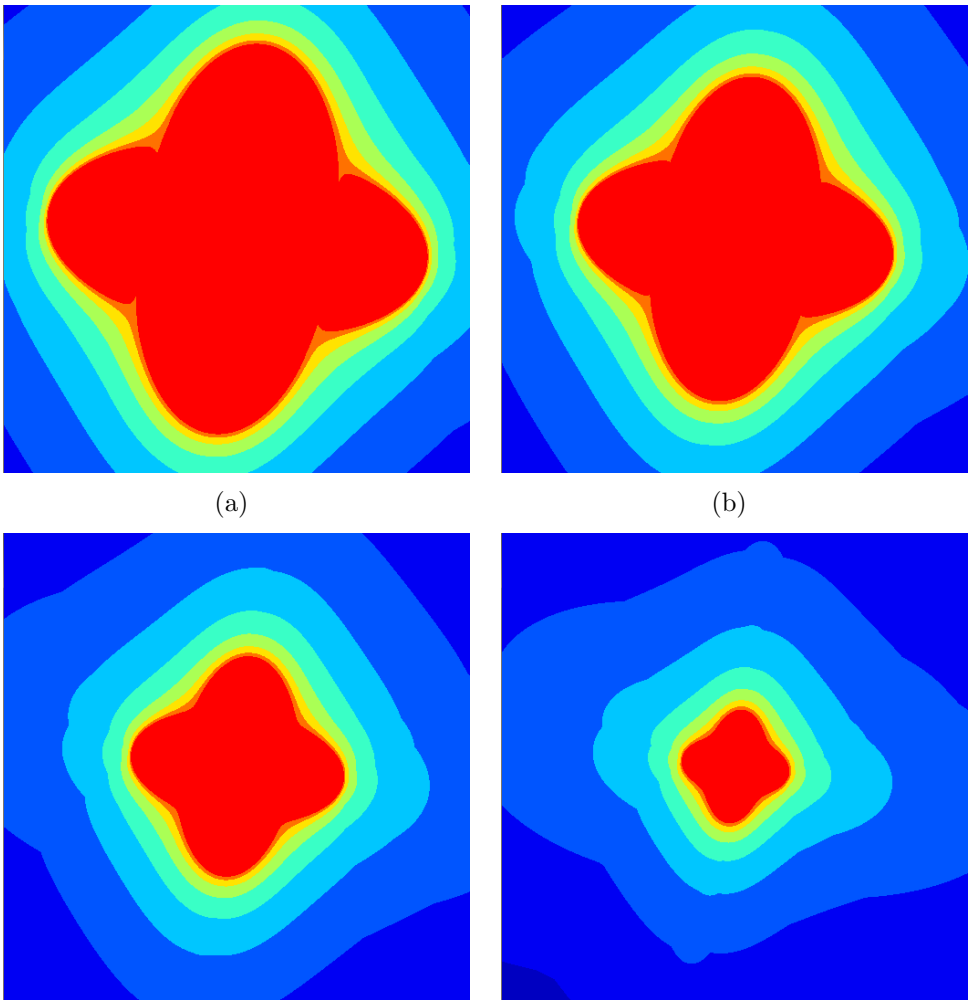


Figura 4.2: No item (a) a superfície *Mitchell* de referência. Nos items (b) a (d), o mapas de profundidade máxima atigida pelo algoritmo de Mitchell. No item (e) a referência das cores, o azul à esquerda vale zero, e o vermelho a direita tem a profundidade máxima.



(a) (b)

(c) (d)

Figura 4.3: No item (a) a superfície *Mitchell* de referência. Nos items (b) a (d), o mapas de profundidade máxima atingida pelo algoritmo de Bisecção Intervalar. O valor da cor é o mesmo que da Figura 4.2 item (e). Note que sobre a superfície o algoritmo sempre atinge profundidade máxima.

Pixels(%)	Tempo(ms)	Pixels(%)	Tempo(ms)
78.54	250.13	24.24	177.46
54.54	228.49	19.63	168.13
40.08	207.16	16.22	159.22
30.69	190.25	13.63	151.94

Tabela 4.5: Mitchell. Conforme diminuímos a área relativa da superfície melhor fica o desempenho.

Pixels(%)	Tempo(ms)	Pixels(%)	Tempo(ms)
78.54	399.93	24.24	204.70
54.54	334.28	19.63	182.17
40.08	276.31	16.22	167.04
30.69	234.61	13.63	157.17

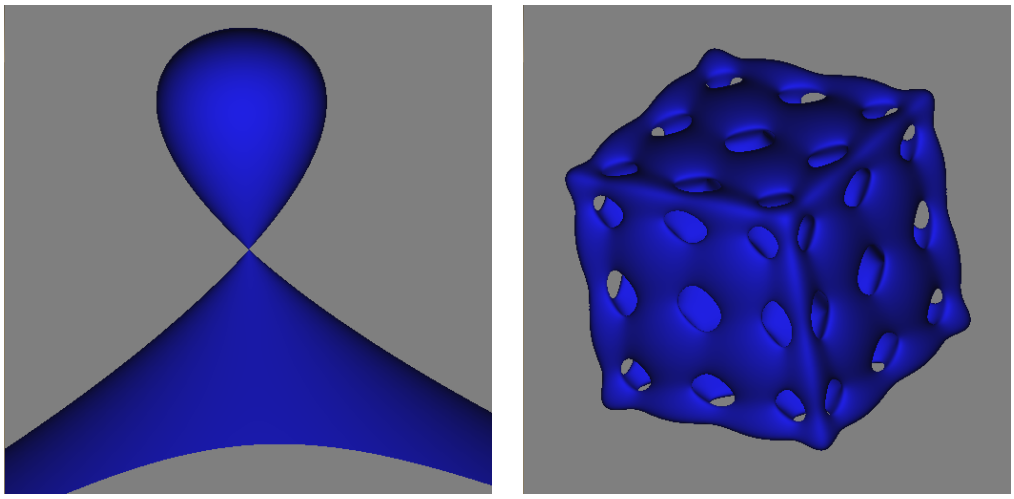
Tabela 4.6: Bisecção Intervalar. Conforme diminuímos a área da superfície melhor fica o desempenho.

iterações, a Bisecção Intervalar é superior. Isto se dá porque o cálculo da derivada intervalar é computacionalmente caro. Um exemplo é a superficie *Chmutov* de grau 8.

4.1.3 Paralelismo na CPU

Cada um dos algoritmos vistos anteriormente executa os cálculos para cada raio sequencialmente. Porém, segundo nossa formulação do problema, cada pixel é independente dos demais, pois sua cor é definida por um único raio. Podemos tirar partido desta independência e melhorar o desempenho. Atualmente, os computadores pessoais são equipados com vários processadores e estes, por sua vez, são constituídos por vários *núcleos*. Isto é, cada processador em um computador pode executar, *paralelamente*, vários programas distintos. Podemos usar esse paralelismo para calcular grupos de pixel em paralelo.

Outra fonte de paralelismo na CPU vem do uso das instruções *SSE* ([21]). Instruções SSE permitem que até quatro operações de ponto flutuante sejam executadas pelo custo de uma operação. Em nossa implementação usamos



(a) (b)

Figura 4.4: (a) *Dingdong* visualizada em 240 ms por Mitchell e 333 ms com Bisseção Intervalar. (b) *Chumtov* visualizada em 3,777 s por Mitchell e em 3,008 s peloa Bisseção Intervalar.

SSE para acelerar, principalmente, as operações vetoriais como multiplicação matriz–vetor e operações intervalares ([23]).

Na Tabela 4.7, vemos os resultados de desempenho para o Algoritmo de Mitchell quando executa paralelamente em um computador com um processador de quatro núcleos, com e sem o uso de instruções SSE.

Desempenho Algoritmo CPU			
	1 *thread*	4 *threads*	4 *threads* + SSE
Esfera[2]	173.16	47.46	39.47
Dingdong[3]	196.86	59.86	47.30
Tangle[4]	1968.53	562.20	468.37
Heart[6]	897.01	264.19	231.99
Chmutov[8]	3639.02	1096.56	912.70

Tabela 4.7: Desempenho (ms) do Algoritmo de Mitchell rodando em CPU, com $\varepsilon = 2^{-9}$.

4.2 GPU

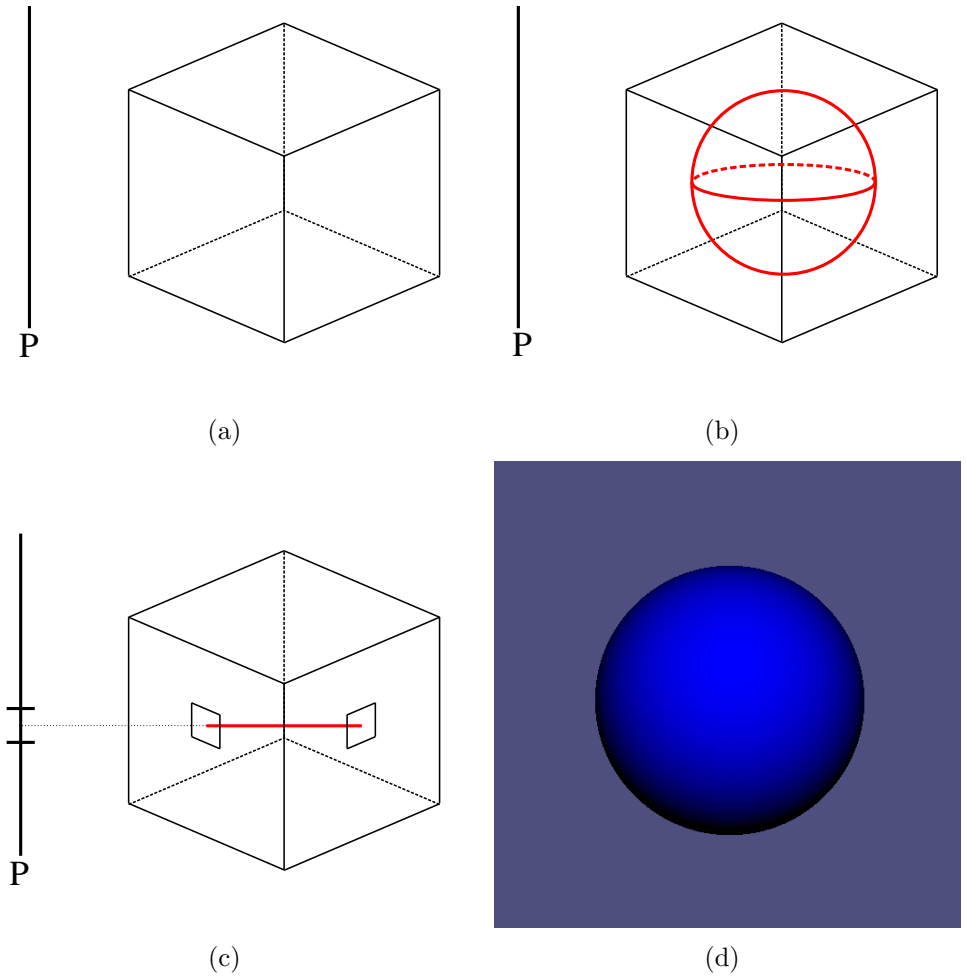
Atualmente, a maioria dos computadores vêm equipados com uma placa gráfica programável (GPU). Estas placas apareceram com o propósito de visualização de gráficos tridimensional em tempo real e no inicio operavam segundo um paradigma fixo de rasterização de polígonos. Com a evolução do *hardware*, alguns estágios do *pipeline* passaram a ser programáveis via *shaders*. Hoje, estas placas podem ser usadas para computação paralela mais geral, inclusive em problemas não relacionados à computação gráfica [20]. Em nossos testes usaremos uma placa nVidia GTX470.

4.2.1 Trabalhos Anteriores

Trabalhos anteriores na área de visualização de superfícies implícitas usando GPU foram realizados, majoritariamente, com o uso do *pipeline* gráfico tradicional: OpenGL e GLSL. Podemos citar duas abordagens comumente utilizadas.

A primeira, vista em trabalhos como [43], cria com OpenGL um *quad* que ocupa toda a imagem de saída. Este *quad* é processado pelo *pipeline* gráfico e é *rasterizado*, gerando *fragmentos*. Em seguida, para cada fragmento, é executado em paralelo um *shader* GLSL que contém o algoritmo de visualização. As coordenadas do centro de cada fragmento são transformadas em coordenadas relativas aos pontos da imagem virtual R através de uma transformação afim. E, para cada ponto, é executado o algoritmo.

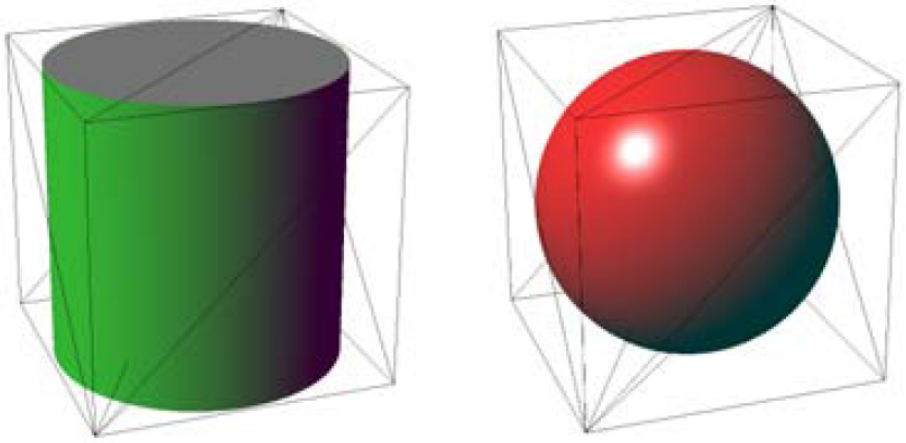
Outra abordagem é a usada por [25]. Nesses trabalhos o processo começa com a definição de um objeto gráfico, geralmente um tetraedro ou um paralelepípedo, que delimita a superfície e provê um sistema de referência. Esse objeto é processado pelo *pipeline* gráfico e suas faces são rasterizadas. Os fragmentos gerados pelo processo de rasterização são processados por um *shader* de fragmentos. Para cada fragmento, é gerado um raio que entra por uma face e, necessariamente, sai por outra. O algoritmo que busca raízes as procura entre o ponto de entrada e o ponto de saída do raio. Na Figura 4.5 temos a representação desse processo.



(a) (b)

(c) (d)

Figura 4.5: Visualização com OpenGL e GLSL usando um cubo, que delimita uma esfera. No item (a) temos um cubo que circunscreve a esfera. No item (b) esse cubo é rasterizado em fragmentos. Em (c) estes fragmentos dão origem ao raios. E em (d) o resultado final.



(a) (b)

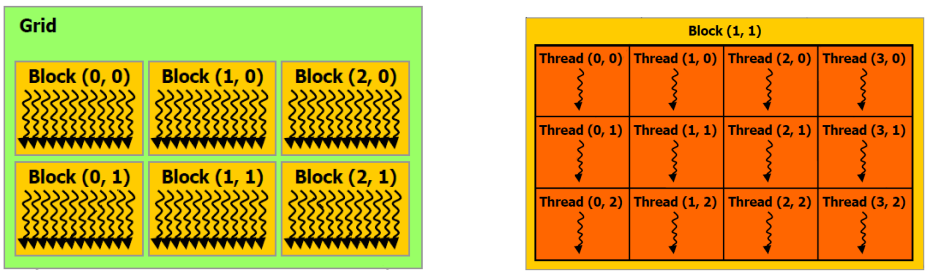
Figura 4.6: Exemplos de visualizações utilizando a técnica de Loop e Blinn [25].

4.2.2 CUDA

Em nosso trabalho utilizamos *CUDA* para implementar na GPU os métodos estudados. O modelo de programação paralelo CUDA foi proposto pela NVIDIA em 2006 [32] e, desde então, tem sido implementado em suas placas gráficas.

Cada programa em CUDA é chamado de *kernel* e é escrito na linguagem *C* ou *C++*. Cada *kernel* é executado em paralelo por diversas *threads*. O número de *threads* simultâneas pode chegar à ordem de milhares. Estas *threads* são agrupadas em *blocos*. Cada bloco tem uma quantidade de *threads* especificadas pelo programa, isto é, quando programamos em CUDA temos a liberdade de especificar como as *threads* são agrupadas. Isto é importante pois, como veremos mais adiante, as *threads* em um mesmo bloco não executam de forma totalmente independente. Por fim, os blocos são organizados em um *grid*. O esquema de organização é representado na Figura 4.7.

Ao executar um programa CUDA, a GPU recebe a configuração do *grid* e divide os blocos entre os vários *Streaming Multiprocessors* (SM's). Cada SM executa as *threads* de um bloco de forma concorrente. Um bloco termina quando todas as suas *threads* foram executadas. À medida que os blocos são



(a) (b)

Figura 4.7: Organização das *threads* em um programa CUDA. Em (a) vemos a organização dos blocos em um *grid*. Em (b) a organização de um bloco.

executados, os blocos restantes são enviados aos SM's ociosos.

Em cada SM que executa um bloco, as *threads* são agrupadas em conjuntos de 32 elementos chamados *warps*. Todas as *threads* em um *warp* executam a mesma instrução do programa em um determinado instante de tempo. Se as *threads* em um *warp* divergem quanto ao caminho de execução, então todos os caminhos são executados serialmente pelo *warp*, desabilitando as *threads* que não tomaram aquele caminho. Um dos fatores a serem considerados quando escrevemos programas eficientes em CUDA é a redução da divergência dentro dos *warps*.

É possível alocar para os blocos uma quantia de memória compartilhada (*shared memory*) de alto desempenho. Essa memória pode ser acessada pelas *threads* do bloco para compartilhar informação durante a execução do *kernel* e é independente para cada bloco. Cada SM tem uma quantidade limitada de memória compartilhada que é dividida entre os blocos ativos em um SM. Portanto, um dos fatores que determinam a quantidade de blocos ativos em um SM é a quantidade de memória compartilhada usada pelos blocos.

Outro fato importante é que cada *thread* de um bloco tem o seu próprio conjunto de registradores. Esses registradores são alocados quando um bloco é enviado ao SM. A exemplo de memória compartilhada, a quantidade de registradores por SM é limitada. Portanto, a quantidade de registradores usado por um bloco limita o número de blocos ativos em um SM.

Vários outros fatores limitam o desempenho de um *kernel* em CUDA.

Podemos citar, por exemplo, o uso mal coordenado da memória global da GPU e a transferência de dados entre CPU e GPU. Algumas referências são [20] e [32].

4.2.3 Implementação

Em nossa implementação usamos OpenGL, GLSL e CUDA. Em OpenGL criamos um *quad* que cobre a tela, esse *quad* é posteriormente texturizado por um *shader* simples, escrito em GLSL. O processo de visualização da superfície se dá, de fato, quando geramos a textura. Essa textura é gerada a partir de um *buffer* OpenGL alocado, uma única vez, na memória global da GPU. Depois, essa memória é mapeada para o espaço de endereços de CUDA. Em CUDA essa área de memória é preenchida com o valor das cores dos pixels da imagem de saída. Na Figura 4.8 (a) vemos a representação desse processo.

Em CUDA a visualização se dá em duas etapas. Na primeira, o kernel que implementa o algoritmo de visualização calcula os valores de $t_0 \in [0, 1]$ para cada pixel. Esses valores são armazenados em uma área de memória intermediária. Nos casos em que não existe raiz para um pixel, um valor constante 2 é armazenado. Chamaremos essa área de memória de *buffer de profundidade*. O *buffer* de profundidade é processado por um segundo *kernel* CUDA que calcula a posição de cada ponto e sua normal, com esses dados ele executa o processo de *shading* (Figura 4.8 b).

Decidimos executar a visualização em duas partes porque dessa forma cada *kernel* utiliza menos registradores. Essa redução do número de registradores permite que mais *threads* executem em paralelo na GPU, o que no final, torna o processo mais eficiente. Veremos no Capítulo 5 que o *buffer* de profundidade será útil no calculo do *anti–aliasing*. Podemos ver o desempenho dos métodos na Tabela 4.8.

4.2.4 Detalhes de Implementação

Uma das principais dificuldades em implementar os métodos intervalares na GPU é que, devido a limitações do *hardware*, não podemos usar funções re-

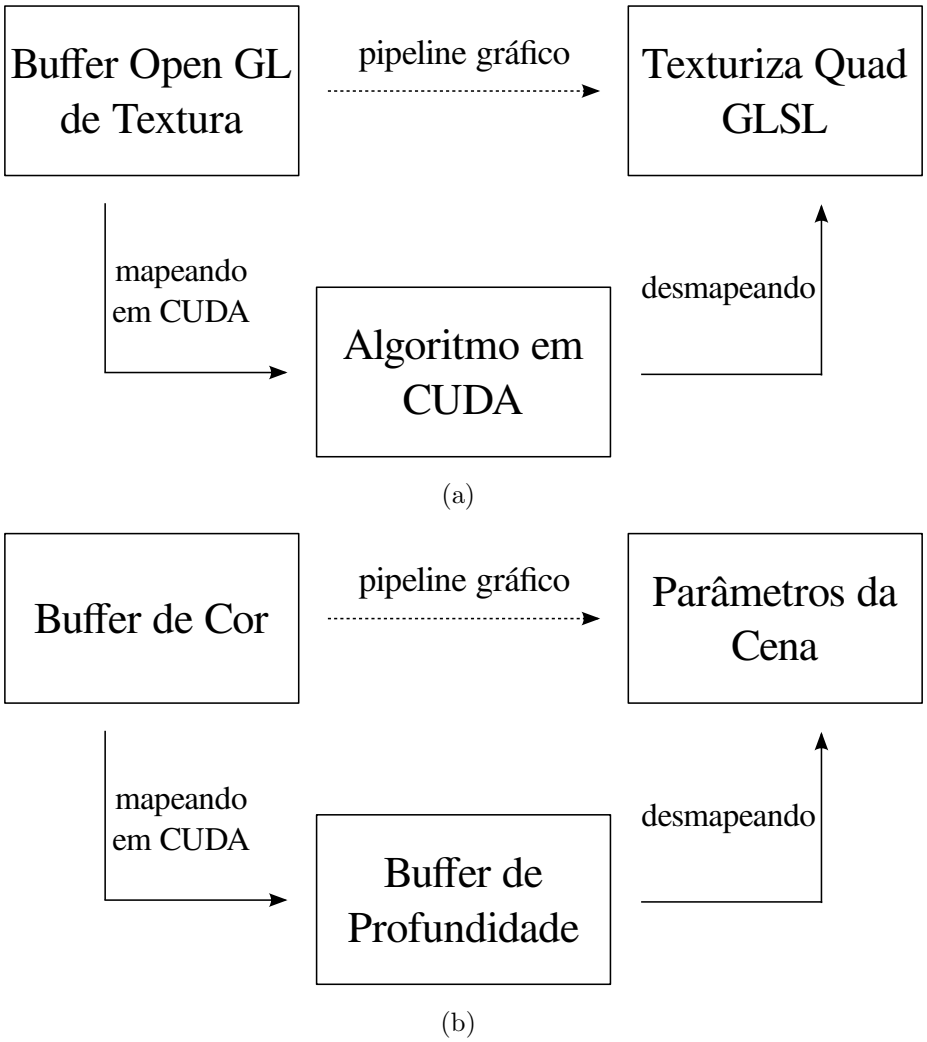


Figura 4.8: (a) Estrutura do programa de visualização usando CUDA. Em (b), a organização do algoritmo de visualização em duas partes. A primeira calcula o *buffer* de profundidade e a segunda executa o *shading*.

Desempenho dos Algoritmos em CUDA				
	Uniforme	Adaptativa	Mitchell	B. Intervalar
Esfera[2]	139.66	39.38	1032.42	587.40
Dingdong[3]	112.08	40.32	920.48	683.85
Tangle[4]	82.54	64.78	202.35	85.52
Heart[6]	97.21	44.02	431.72	436.26
Chmutov[8]	82.22	72.96	101.00	70.76

Tabela 4.8: Desempenho (em frames por segundo – *fps*) dos algoritmos em CUDA.

cursivas. Para contornar esse problema podemos implementar o algoritmo usando uma pilha [37] e nesse caso a pilha serve para guardarmos o caminho tomado pelo algoritmo enquanto procura a raiz. Porém, a implementação direta dessa estratégia se mostra muito ruim. A velocidade com que uma *thread* acessa a memória global é limitada e, em geral, gostaríamos de evitar leituras e escritas desnecessárias. Podemos simular uma pilha de forma eficiente levando em conta o fato de que operações de subdivisão binárias podem ser mapeadas nos bit de um número inteiro. Usando operações de *shift* e incrementos é possível manter o estado do processo de subdivisão dos intervalos.

Capítulo 5

Anti–Aliasing

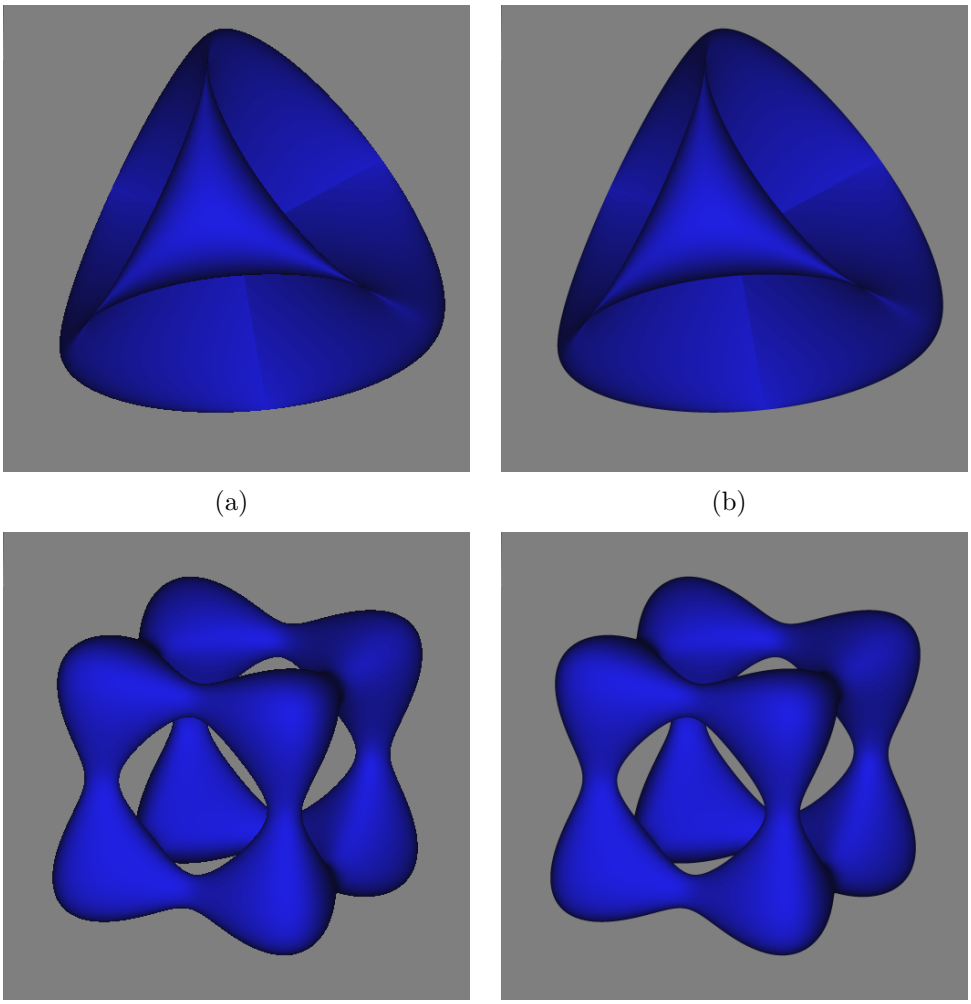
No Capítulo 4 vimos como implementar os métodos vistos no Capítulo 3. Neste capítulo, voltamos ao problema do *aliasing*. Veremos como implementar de forma eficiente algumas estratégias para reduzir esse problema.

5.1 Super Amostragem

No Capítulo 3 vimos que, ao escolher apenas um ponto como amostra para calcular o valor do pixel, introduzimos artefatos na imagem final. Esses artefatos são resultado do *aliasing* causado pela baixa taxa de amostragem da imagem. Apontamos que uma possível solução para o problema é a super–amostragem, ou seja, tomamos mais amostras e usamos seus valores para compor a imagem final. Na Figura 5.1 vemos o exemplo de duas superfícies, ambas mostradas com e sem a super–amostragem.

Apesar de simples a super–amostragem tem um problema grave. Estamos desperdiçando computação em áreas com pouca variação de cor. Na Tabela 5.1 vemos o desempenho do Algoritmo de Mitchell executando super–amostragem. É possível ver que o desempenho cai proporcionalmente ao número de amostras. Esse comportamento se repete para os outros métodos.

Para acelerar o processo, gostaríamos de poder executar a super–amostragem somente nos pixels onde a variação de cor é grande. Antes disto, porém, precisamos entender em que casos isto acontece. Na Figura 5.2 mos-



(a) (b)

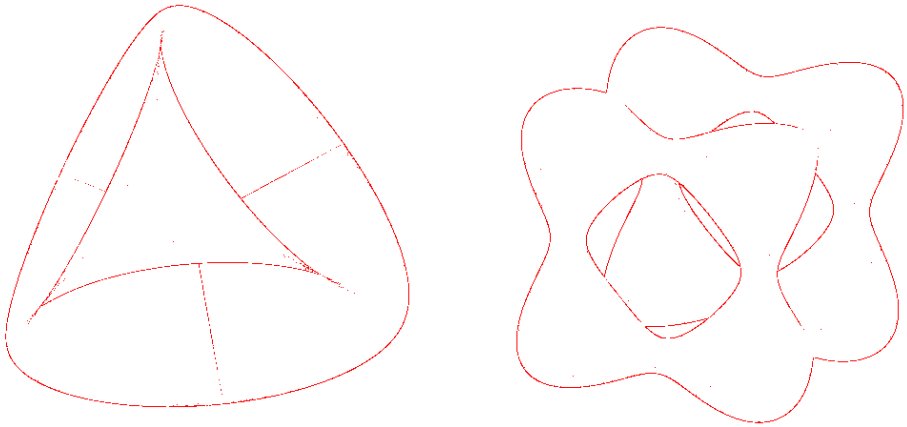
(c) (d)

Figura 5.1: No item (a) temos a superfície *Steiner* com uma amostra por pixel, no item (b) com 16 amostras. Em (c) temos a superfície *Tangle* com uma amostra e, em (d) com 16.

Desempenho do Algoritmo de Mitchell			
Nº de Amostras:	1	4	16
Esfera[2]	1032.42	132.75	39.19
Dingdong[3]	920.48	125.36	37.83
Tangle[4]	202.35	48.47	13.94
Heart[6]	431.72	79.83	23.33
Chmutov[8]	101.00	25.75	8.02

Tabela 5.1: Desempenho (fps) do Algoritmo de Mitchell para a super–amostragem.

tramos a diferença entre as imagens da Figura 5.1. Podemos notar que a maiores diferenças entre as imagens se dão nas regiões próximas à silhueta da superfície.



(a) (b)

Figura 5.2: Em vermelho, os pixels que mudaram quando aumentamos a amostragem de 1 para 16 amostras. Podemos notar que a variação de cor acontece perto das silhuetas.

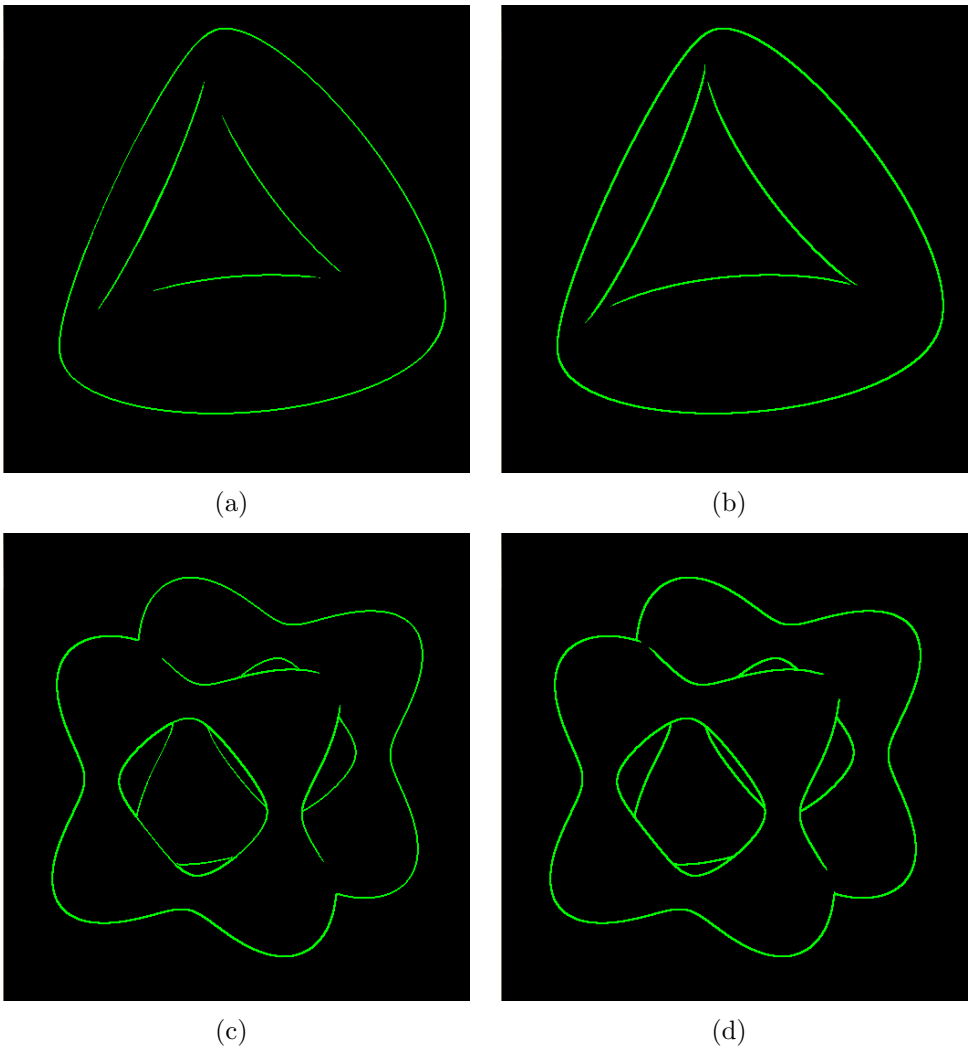
5.2 Super Amostragem Adaptativa

Implementamos um esquema de super–amostragem somente nos pixels onde há grande variação de cor. Para isso, processamos o *buffer* de profundidade de forma a encontrar as bordas da imagem. Um *kernel* CUDA, que implementa o operador de *Sobel*, toma o *buffer* de profundidade e aplica as seguintes convoluções:

$$\mathbf{Z}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} \quad \text{e} \quad \mathbf{Z}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \tag{5.1}$$

Definimos um valor $\lambda > 0$ e arbitramos que os pixels da borda são os pixels onde $\sqrt{\mathbf{Z}_x{}^2 + \mathbf{Z}_y{}^2} > \lambda$. A saída desse *kernel* é um *buffer* contendo

uma máscara, que indica quais pixels foram considerados parte da borda da imagem. A Figura 5.3 mostra a detecção das bordas da imagem e, por consequência, das silhuetas das superfícies mostradas na Figura 5.1 itens (a) e (b). Uma abordagem como essa pode ser vista em [31, 35]. Notemos que a quantidade de pixels detectados varia com o valor de λ. Em nossa implementação, λ foi determinado empiricamente para cada superfície.

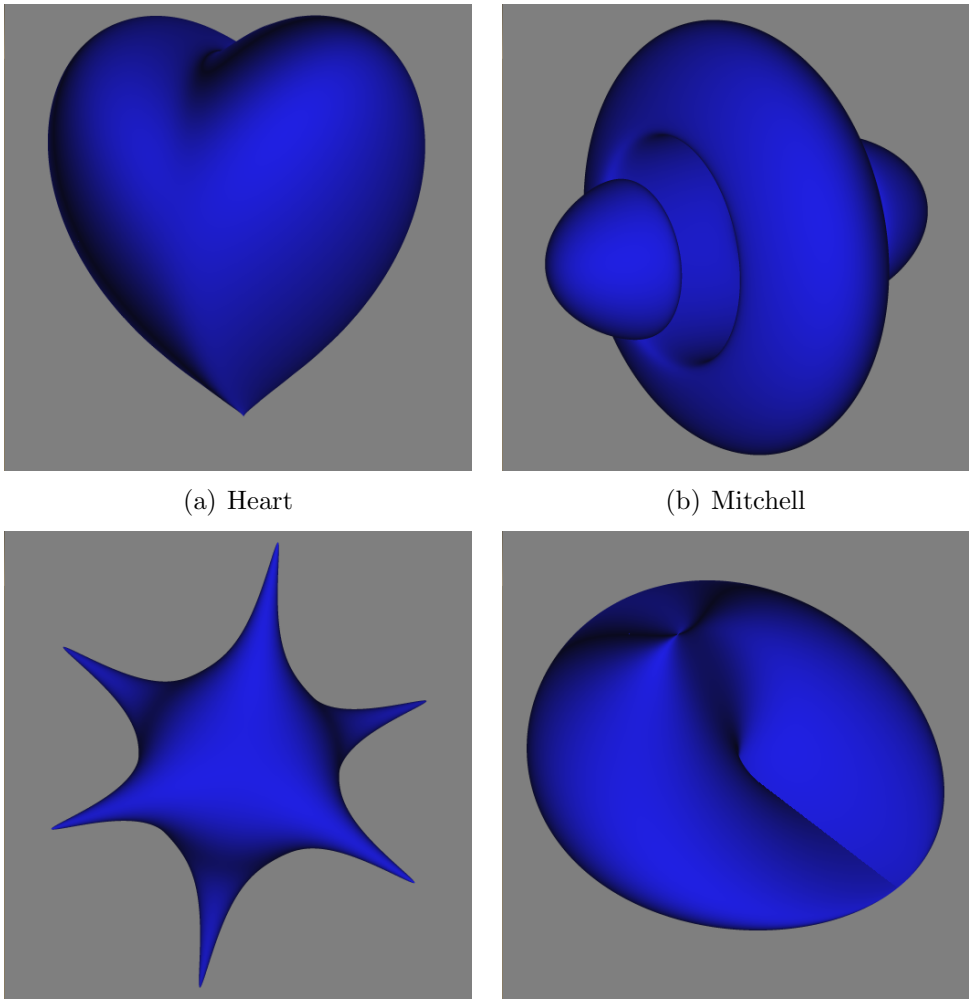


(a) (b)

(c) (d)

Figura 5.3: Detecção de bordas para as superfícies da Figura 5.1, itens (a) e (b) com $\lambda = 0{,}25$. Nos itens (c) e (d) temos as mesmas superfícies com um valor de $\lambda = 0{,}75$.

Em seguida, utilizamos a máscara como referência para realizar a super-amostragem somente nos pixels da borda. A cor desses pixels á calculada seguindo a Equação (2.3). No final, a imagem de saída é gerada quando sobrepomos a cor original dos pixels onde fizemos a super-amostragem pelas novas cores calculadas. Na Figura 5.4 temos exemplos de superfícies onde aplicamos esse processo.



(a) Heart (b) Mitchell

(c) Distel (d) Crosscap

Figura 5.4: Exemplos de Super-Amostragem Adaptativa.

Esse método, quando comparado à super-amostragem simples, produz resultados similares em qualidade porém com uma eficiência maior. Na Tabela 5.2 temos a comparação de desempenho entre esses métodos.

Desempenho Super-Amostragem Adaptativa				
	Simples	S. A. Adaptativa	Novas Amostras (%)	Perda *fps*(%)
Esfera[2]	1032.42	352.32	23	65
Dingdong[3]	920.48	303.41	29	67
Tangle[4]	202.35	86.53	48	57
Heart[6]	431.72	194.55	19	54
Chmutov[8]	101.00	40.07	53	60

Tabela 5.2: Desempenho (fps) da super-amostragem adaptativa comparado a amostragem simples. Na quarta coluna temos o percentual de novas amostras na borda em relação à quantidade de pixels na imagem. Na quinta coluna, temos a perda de desempenho. Todos os testes usaram o Algoritmo de Mitchell.

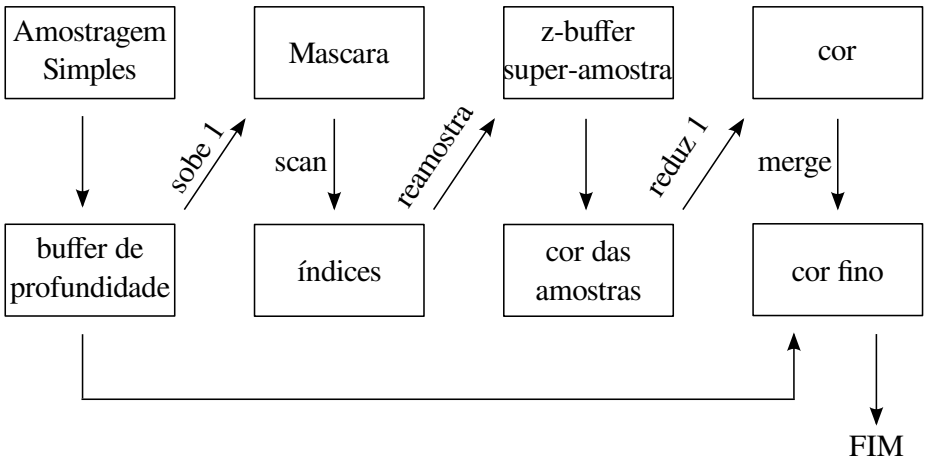
Poderíamos esperar que a perda de desempenho da super–amostragem adaptativa fosse próxima da quantidade de novas amostras a serem tomadas. Como vemos na quinta coluna da tabela, não é o que acontece. Podemos apontar alguns motivos para isto. Primeiro, a região da borda é onde o algoritmo tem o maior custo de execução (Figura 4.2), pois sempre atinge a maior profundidade ao tentar isolar a raiz. Essa região é onde as *threads* tem maior divergência dentro de um *warp*, o que causa perda de desempenho. Outro motivo é que esse método tem um custo extra devido à forma como é implementado, como veremos a seguir.

Na Figura 5.5 vemos uma representação dos passos usados na implementação.

Foge ao escopo deste trabalho apresentar uma explicação detalhada dos algoritmos de *scan* e *reduce* utilizados na implementação. Uma descrição detalhada pode ser encontrada em [13]. Porém, podemos resumir seu funcionamento da seguinte forma: *scan* toma um vetor de números e retorna outro vetor, que contém a soma acumulada das entradas do vetor inicial. O algoritmo *segmented reduce* executa a soma das entradas de um vetor, segundo um critério de agrupamento pré-definido, retornando um vetor com as somas de cada grupo.

Como havíamos visto, a amostragem simples gera o *buffer* de profundidade que por sua vez é processado para calcularmos a cor dos pixels. Com a super-amostragem adaptativa, adicionamos a etapa onde o *buffer* de pro-

fundidade é convertido em uma máscara que indica onde devemos realizar a super-amostragem. Essa máscara é processada pelo algoritmo de *scan* de forma a criarmos um vetor com índices. Esses índices indicam a posição onde serão guardados os resultados da super-amostragem, em um novo *buffer* de profundidade. Esse novo *buffer* de profundidade é convertido em um *buffer* de cor. Finalmente, o *buffer* de cor é reduzido pelo algoritmo *segmented reduce* em um *buffer* que contém a nova cor dos pixels da borda e, em seguida, esses valores são copiados para a imagem final.



(a)

Figura 5.5: Passos usados pela implementação em CUDA da super–amostragem adaptativa.

Capítulo 6

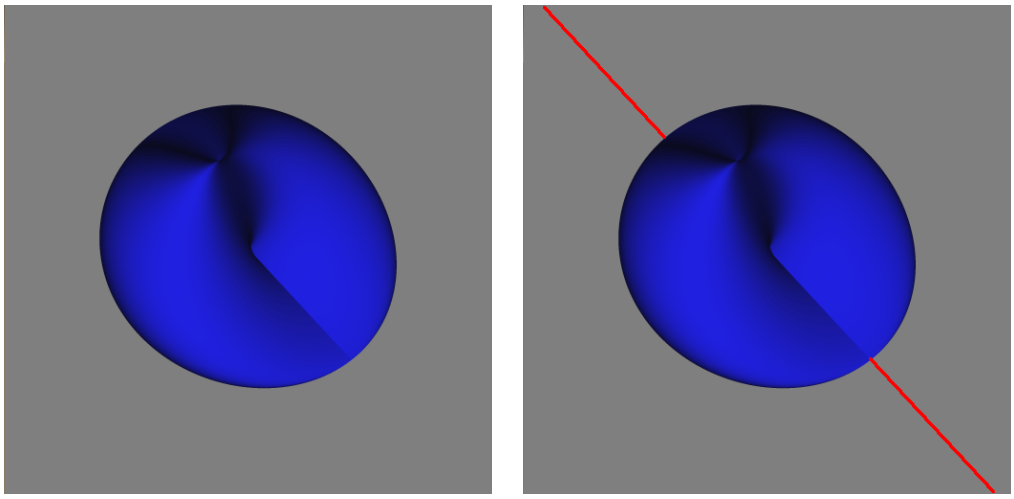
Beam–Casting e Subdivisão Espacial

No Capítulo 5 vimos como selecionar pixels na borda da superfície, e neles aplicar super-amostragem, de forma a conseguir uma visualização de maior qualidade. Neste capítulo apresentaremos outras estratégias de visualização.

6.1 Beam–Casting Intervalar

Apesar de produzir boas imagens a super-amostragem adaptativa não garante que o resultado é robusto. Consideremos a Figura 6.1 item (a), nela, vemos a superfície *Crosscap* com super-amostragem adaptativa. Apesar de ter os contornos suaves, a imagem não mostra um conjunto de pontos da superfície. Estes pontos estão sobre uma reta representada no item (b) por uma linha vermelha. De fato, a maioria das amostras pontuais falham ao tentar detectar esses pontos. Isto se dá porque os raios usados na amostragem dificilmente intersectam a reta contendo os pontos da superfície.

Uma solução para esse problema seria lançarmos muitos raios para cada pixel, afim de que algum detecte esses pontos. Ao invés disto, vamos usar uma técnica conhecida como *beam-tracing* [16]. Vejamos a Figura 6.2 (a). Podemos considerar que um feixe de raios parte da região r e viaja em direção à superfície. Caso possamos detectar se o feixe de raios atinge a superfície,



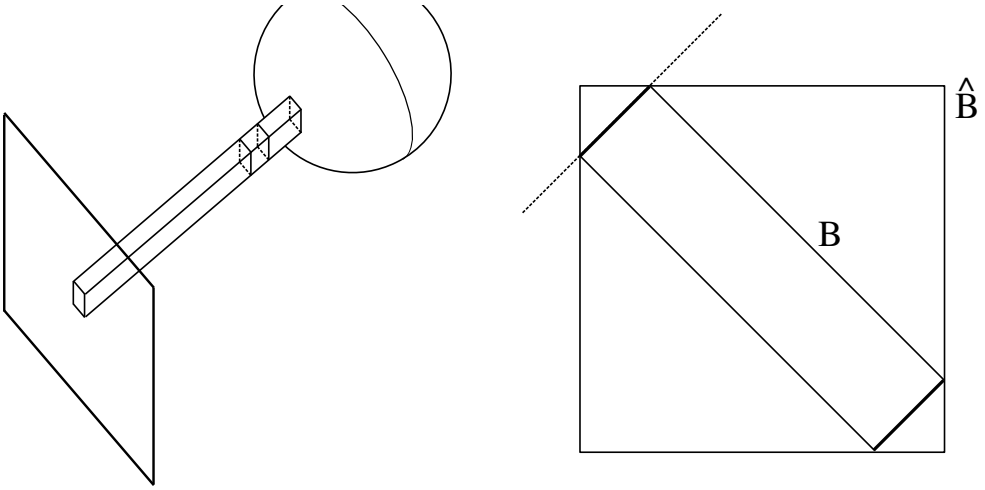
(a) (b)

Figura 6.1: No item (a) vemos *Crosscap* com super-amostragem adaptativa. No item (b), em vermelho, representamos os pontos que não foram detectados.

mesmo uma reta como a da Figura 6.1 será detectada. Usaremos aritmética intervalar para simular esse feixe.

Consideremos novamente a representação intervalar F da função f que define a superfície implícita. Sabemos que podemos avaliar F em blocos da forma $V := X \times Y \times Z \subset \mathbb{R}^3$. Nesse caso o intervalo $F(V)$ contém toda imagem de V por f. Agora, se tomarmos um feixe B, como na Figura 6.2 (a), podemos calcular o menor bloco $\hat{B}$ que o envolve (Figura 6.2 b). Precisamos disto porque F está representada em termos dos bloco no sistema de coordenadas do mundo e, portanto, só pode ser avaliada nesses blocos. Como $f(B) \subset F(B) \subset F(\hat{B})$, podemos procurar a intersecção do feixe B com a superfície usando um algoritmo similar a Bisecção Intervalar. Para isso usamos $F(\hat{B})$ como teste de inclusão.

O Algoritmo 6.1 mostra o processo. Notemos que, além da função intervalar F e do limite de precisão ε, o algoritmo toma como entrada três intervalos: U, V e $[t_i, t_f]$. Estes intervalos descrevem um bloco que representa o feixe em termos das coordenadas da câmera. Porém, é preciso que o feixe seja representado por um bloco em coordenadas do mundo, o que é feito



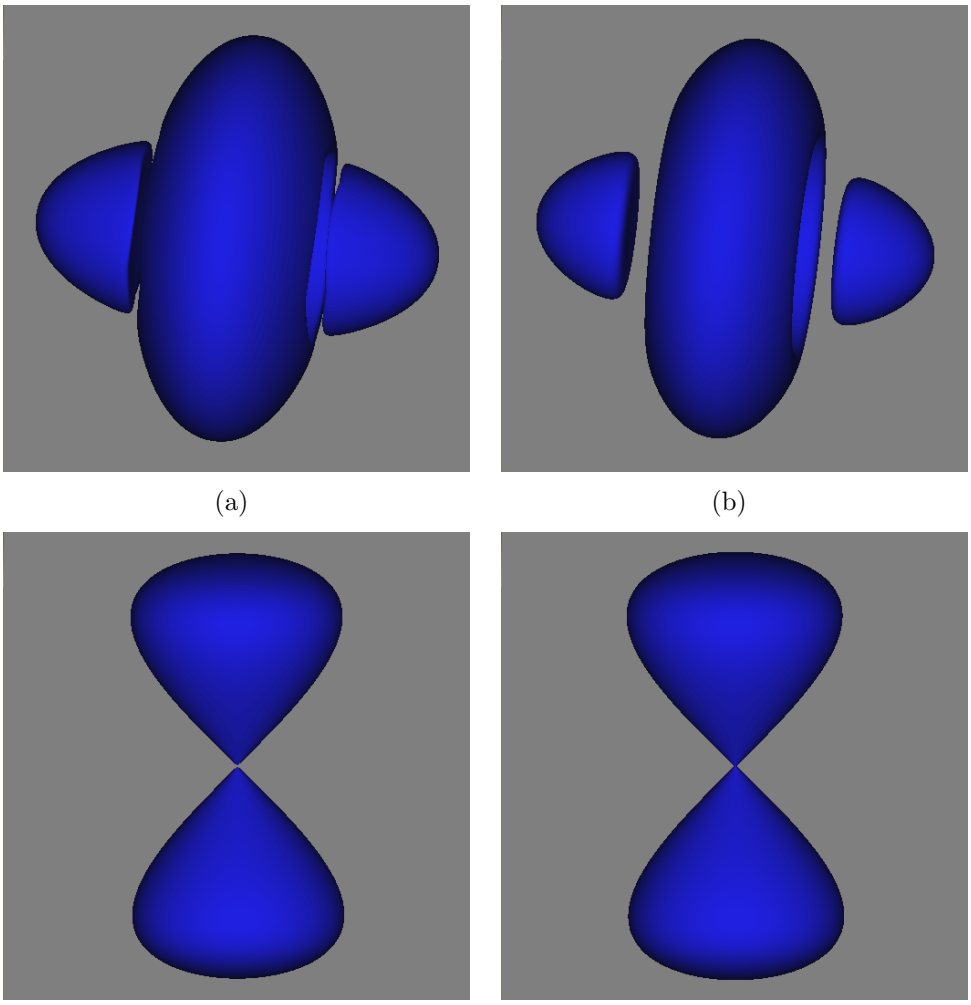
(a) (b)

Figura 6.2: Item (a) mostra um feixe de raios. O item (b) mostra o bloco que contém o feixe em coordenadas do mundo.

por $Bloco()$. Caso tenhamos alguma raiz em $\hat{B}$, testamos o comprimento do bloco B. Se B tem comprimento menor que ε então retornamos $[t_i, t_f]$. Caso contrário executamos o $BeamCasting$ recursivamente nas duas metades do feixe. O algoritmo retorna um intervalo $W \subset [t_i, t_f]$, de forma que o bloco $U \times V \times W$ é o bloco de menor profundidade que pode conter uma raiz. Ou retorna $\varnothing$ que indica que o feixe não atinge a superfície.

Podemos considerar que, nos casos onde o algoritmo acusa uma intersecção entre o feixe e a superfície, o centro do bloco é um ponto na superfície. Esta suposição simples produz resultados interessantes (Figura 6.3). A diferença entre os resultados produzidos por essa forma de visualização e os resultados produzidos pelos métodos convencionais de amostragem pontual se dá por dois motivos. Primeiro, para feixes perto da silhueta da superfície, a sobre–estimativa típica da AI nos induz a considerar como válidos blocos que não intersectam a superfície (Figura 6.4 a). Segundo, pela perda de precisão que o *beam casting* intervalar apresenta nos blocos pequenos.

Consideremos a Figura 6.4 item (b). Nela mostramos um feixe após algumas etapas de subdivisão do Algoritmo 6.1 e o menor bloco em coordenadas do mundo que o envolve. Quando uma nova subdivisão é realizada, o feixe é



(a) (b)

(c) (d)

Figura 6.3: Nos itens (a) e (c) temos a superfícies *Mitchell* e *Octdong* respectivamente, visualizadas com *Beam Casting*. Nos itens (b) e (d), temos como referência as mesmas superfícies visualizadas pela Bisecção Intervalar.

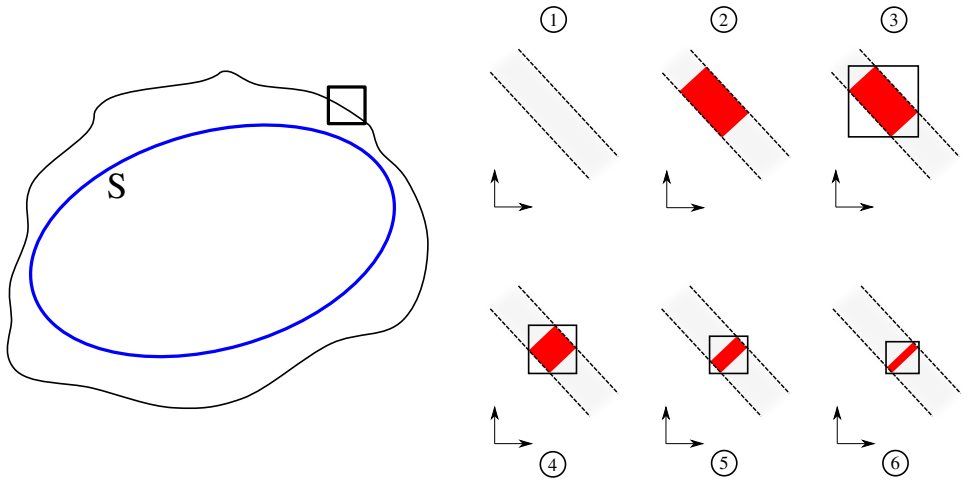
Algoritmo 6.1 Beam Casting

procedimento $BeamCasting(F,\, U,\, V,\, [t_i, t_f],\, \varepsilon)$
  $\hat{B} \leftarrow Bloco(U, V, [t_i, t_f])$
  se $0 \in F(\hat{B})$ **então**
    se $t_f - t_i < \varepsilon$ **então**
      retorne $[t_i, t_f]$
    se não
      $J \leftarrow BeamCasting(F, U, V, [t_i, \frac{t_i+t_f}{2}])$
      se $J \neq \varnothing$ **então**
        retorne J
      se não
        retorne $BeamCasting(F, U, V, [\frac{t_i+t_f}{2}, t_f])$
      fim se
    fim se
  se não
    retorne $\varnothing$
  fim se
fim procedimento

dividido apenas na direção de propagação (item c). Com exceção dos casos onde o feixe tem lados paralelos ao sistema de coordenadas do mundo, o bloco que o contém converge para um bloco de dimensões não desprezíveis (itens d e e). O que nos permite concluir que, nesses casos, $F(\hat{B})$ não nos fornece estimativas precisas. Ou seja, a partir de um certo valor para ε, $BeamCasting$ não fornece uma visualização mais acurada.

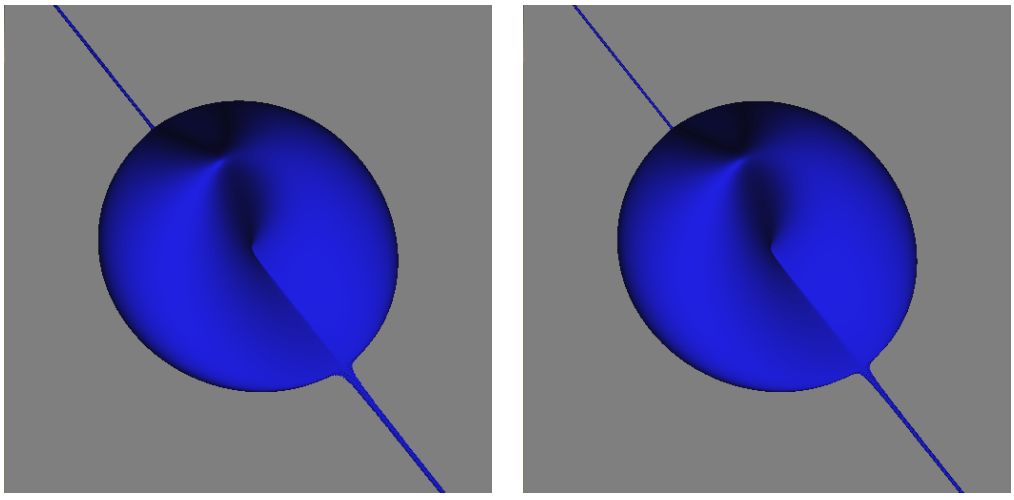
Por outro lado, *BeamCasting* nos permite visualizar os pontos de *Crosscap* que até então não havíamos podido detectar. Vejamos a Figura 6.5.

Uma forma de melhorarmos esse resultado é aplicarmos a mesma ideia que usamos na super–amostragem ao método de *beam casting*. Dividimos a área do pixel em partes iguais e usamos feixes mais estreitos. Dessa forma diminuímos o problema da precisão discutido acima pois podemos avaliar F em blocos de tamanho menor. Para cada feixe aplicamos o algoritmo $BeamCasting$ e calculamos a cor final do pixel somando a contribuição individual de cada feixe. Na Figura 6.6 vemos a superfície *Crosscap* visualizada dividindo o pixel e 4 partes (item a) e 16 partes (item b).



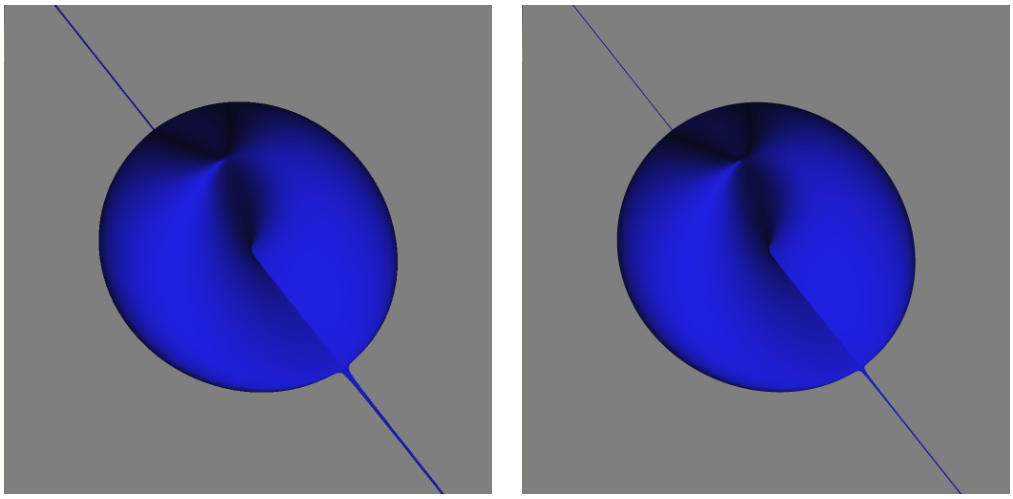
(a) (b)

Figura 6.4: No item (a) mostramos como a sobre–estimativa da AI nos faz considerar feixes perto da silhueta como feixes que realmente intersectam a superfície. No item (b) mostramos que a subdivisão do feixe não implica em um aumento na precisão das estimativas de F.



(a) (b)

Figura 6.5: *Crosscap* visualizada com *beam casting*. No item (a) podemos ver que os pontos que anteriormente não haviam sido detectados estão presente. A visualização foi feita com $\varepsilon = 2^{-9}$. No item (b) a visualização foi feita com $\varepsilon = 2^{-11}$, note que a diferença entre (a) e (b) é imperceptível.



(a) (b)

Figura 6.6: *Crosscap* visualizada com *beam casting*. No item (a) o pixel é dividido em 4 partes e foi usado $\varepsilon = 2^{-10}$. No item (b) o pixel foi dividido em 16 partes e foi usado $\varepsilon = 2^{-11}$.

O desempenho de *BeamCasting* pode ser visto na Tabela 6.1. Fica claro que o desempenho para o caso em que temos que tomar 16 feixes por pixels é ruim. Vamos tentar melhorá–lo.

Desempenho de *BeamCasting*			
Feixes por Pixel:	1	4	16
Esfera[2]	652.39	94.98	25.89
Dingdong[3]	714.00	106.08	30.09
Tangle[4]	132.87	29.10	7.69
Heart[6]	508.06	84.14	23.34
Chmutov[8]	167.13	31.06	6.50

Tabela 6.1: Desempenho de *BeamCasting* (fps) para ε valendo 2^{-9}, 2^{-10} e 2^{-11} respectivamente.

Podemos tentar aplicar o método de amostragem adaptativa ao *beam casting* e utilizar mais feixes apenas nos pixels da borda. Essa abordagem, porém, não surte o efeito desejado. Vejamos a Figura 6.7. No item (a) vemos a superfície visualizada com 1 feixe por pixel, ao calcularmos a borda (item b) detectamos uma faixa estreita em torno da superfície que julgamos conter a silhueta da superfície. Porém, como vimos acima, nossas estimativas

sofrem de excessos devido a limitações da AI e da geometria dos feixes. Ao re–amostrarmos a borda da superfície com 16 feixes por pixel, melhoramos nossas estimativas e, em geral, descobrimos que os pixels detectados na borda não contém pontos da superfície (item d). O resultado final, visto no item (c), mostra que o processo de super-amostragem simplesmente encolhe a borda anterior e cria uma nova borda, que ainda parece serrilhada.

Vamos, novamente, alterar o método de super-amostragem. Ao invés de detectarmos os pixels da borda, faremos a reamostragem em todos os pixels, com exceção daqueles que foram descartados pelo *beam tracing* na primeira amostragem. Desta vez, os resultados são satisfatórios. Podemos ver exemplos na Figura 6.8 e o desempenho na Tabela 6.2.

Desempenho de *BeamCasting*		
Feixes por Pixel:	4	16
Esfera[2]	106.60	29.76
Dingdong[3]	126.83	35.57
Tangle[4]	32.83	8.78
Heart[6]	92.49	26.03
Chmutov[8]	28.56	6.42

Tabela 6.2: Desempenho de *BeamCasting* (fps), reamostrando todos os pixels detectados.

Podemos fazer uma ultima análise desse caso. Vejamos a Figura 6.9, no item (a) representamos um feixe que viaja da direção w e o bloco B, de profundidade ε, detectado após as etapas de bisecção. Sabemos que o centro desse bloco pode ser uma aproximação grosseira da superfície e por isso realizamos a divisão do feixe em feixes mais estreitos. Porém, temos a garantia que, no volume do feixe que está antes do bloco detectado, não existem pontos das superfície. Isto se dá porque a AI nos garante que não existem raízes de f naquela região. Portanto, podemos otimizar a super-amostragem dos feixes mais estreitos fazendo com que estes partam, não do plano *near*, mas sim do início do bloco B (item b). Podemos ver o ganho em eficiência na Tabela 6.3.

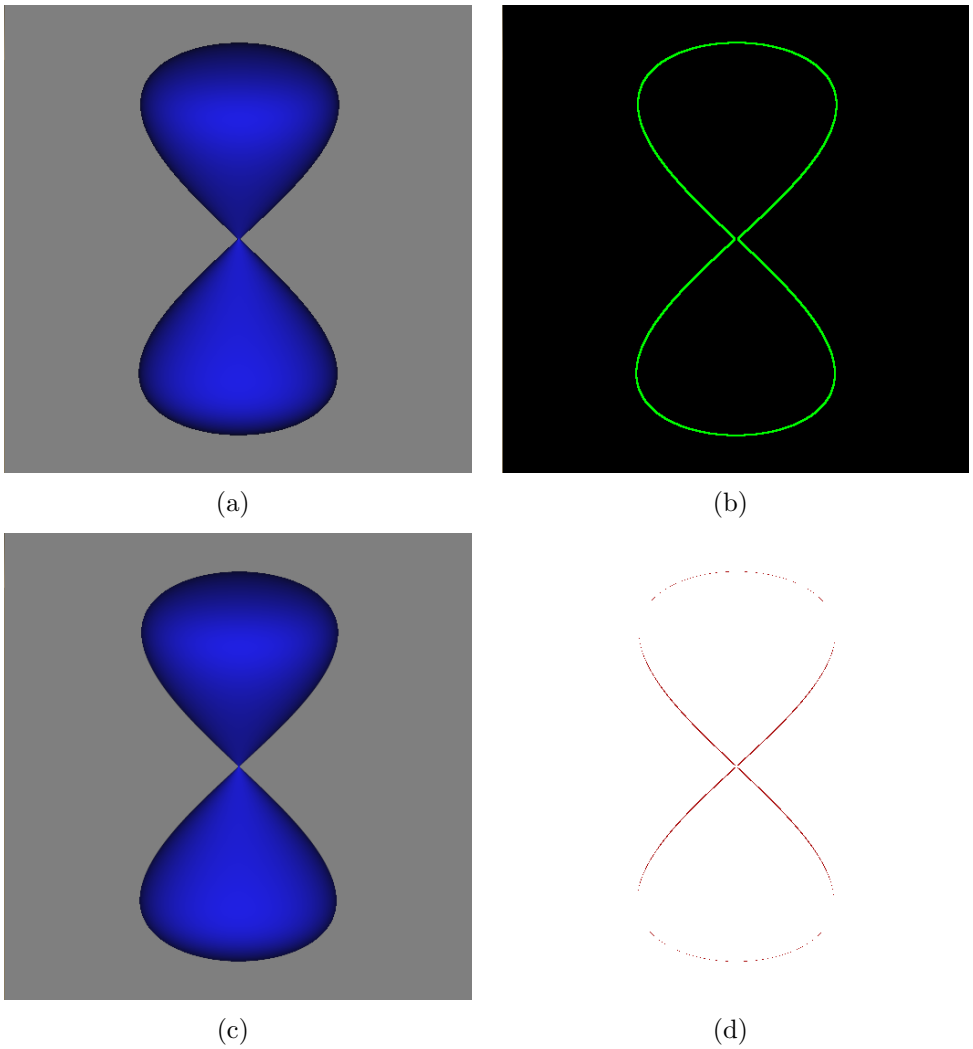
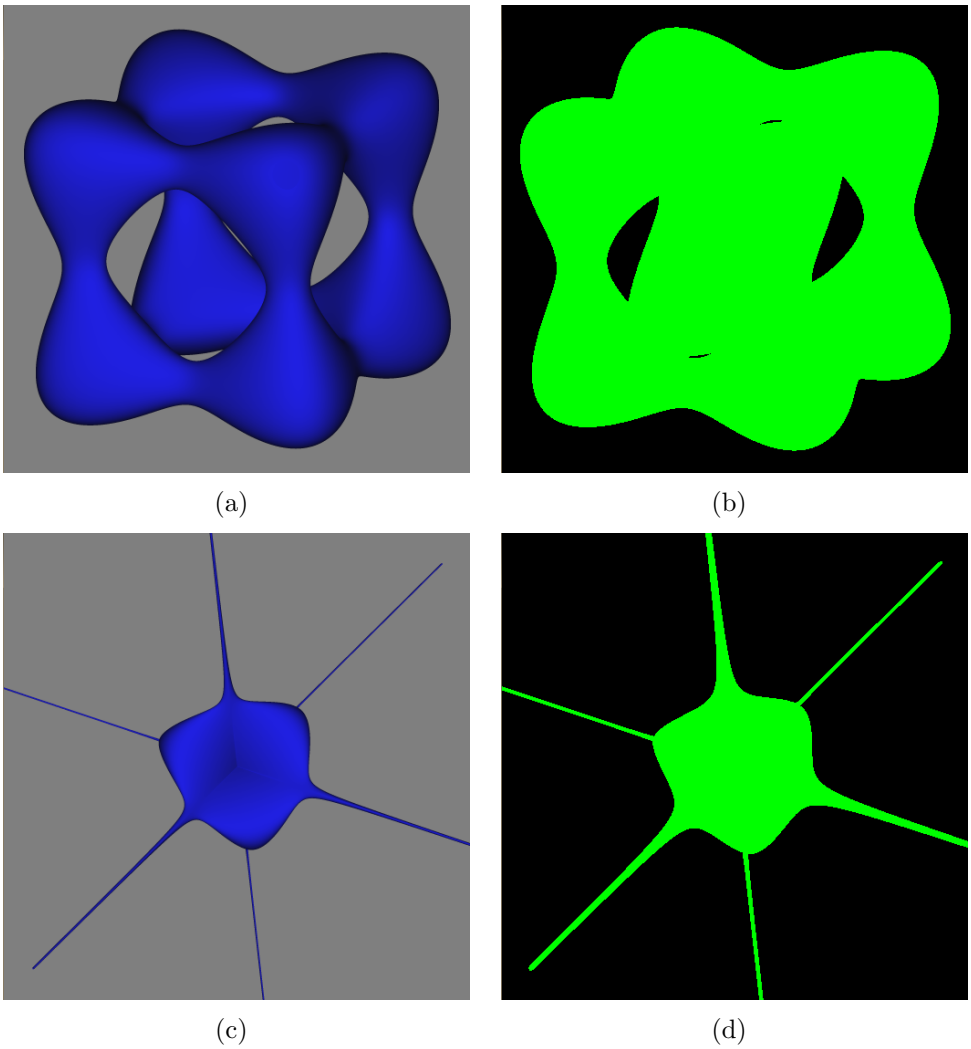


Figura 6.7: Superfície *Octdong*. Em (a) visualizada com 1 feixe por pixel. No item (b), o resultado da detecção de borda para o item (a). No item (c) podemos ver o item (a) após a reamostragem da borda com 16 feixes por pixel. Em (d) vemos, em vermelho, a interseção do item (c) com a borda detectada para o item (a).



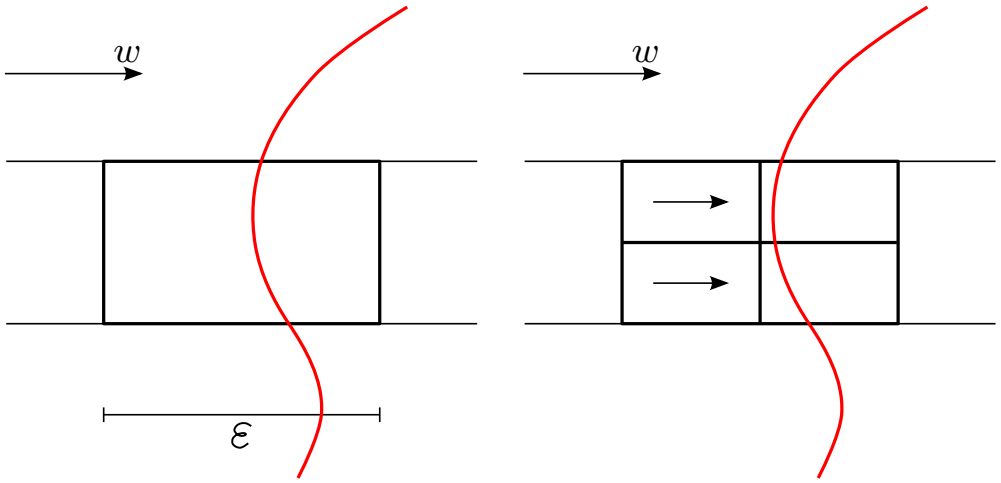
(a) (b)

(c) (d)

Figura 6.8: No item (a) temos a superfície *Tangle* e em (c) temos *Steiner* re–amostradas segundo as máscaras mostradas em (b) e (d) respectivamente.

Desempenho de *BeamCasting*		
Feixes por Pixel:	4	16
Esfera[2]	107.13	30.52
Dingdong[3]	129.17	36.31
Tangle[4]	33.47	8.93
Heart[6]	95.21	27.08
Chmutov[8]	29.19	6.54

Tabela 6.3: Desempenho de *BeamCasting* (fps), reamostrando todos os pixels detectados, com aceleração dos feixes.



(a) (b)

Figura 6.9: No item (a) vemos o bloco B, que resulta da bisecção do feixe. Sabemos que f não tem raízes na região antes de B, o que permite que os feixes mais estreitos partam do início de B. Que é mostrado em (b).

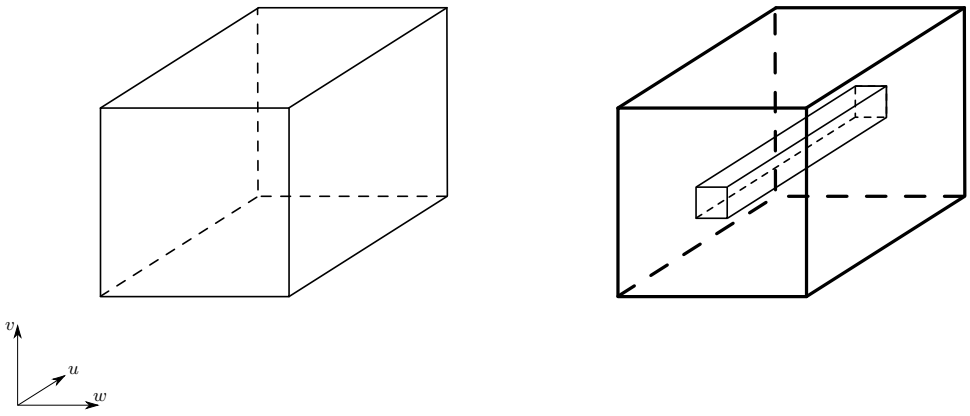
6.2 Subdivisão no Espaço da Imagem

Como vimos na seção anterior, podemos usar feixes de raios para detectar pontos não visíveis quando usamos amostragem pontual. Vimos também como subdividir um feixe de forma a melhorar a detecção da superfície. Essa subdivisão nos permitiu implementar uma forma de *anti-aliasing*. Por fim, apontamos para a possibilidade de usar o resultado de uma amostragem grosseira como ponto de partida para uma mais fina. Essa ideia vai nos permitir construir um algoritmo mais eficiente que os anteriores e, ainda sim, manter a qualidade final da visualização.

6.2.1 Espaço da Imagem

No Capítulo 2, vimos como definir o volume de visão (chamado de V) que é a região do espaço que desejamos visualizar. Estamos assumindo que todos os feixes usados para a visualização, são blocos contidos em V e que suas arestas são paralelas as arestas de V. Através de uma transformação afim, V pode ser identificado com um *volume normalizado* (Figura 6.10 a), cujos

lados são paralelos ao referencial $\{u, v, w\}$. Dessa forma, podemos escrever $V = [u_i, u_f] \times [v_i, v_f] \times [w_i, w_f]$ e cada feixe como um bloco $B := [a_i, a_f] \times [b_i, b_f] \times [w_i, w_f] \subset V$ (Figura 6.10 b). A face $[u_i, u_f] \times [v_i, v_f] \times \{w_i\}$ de V é identificada com o retângulo de imagem R, dizemos que esta é a face da imagem de V.

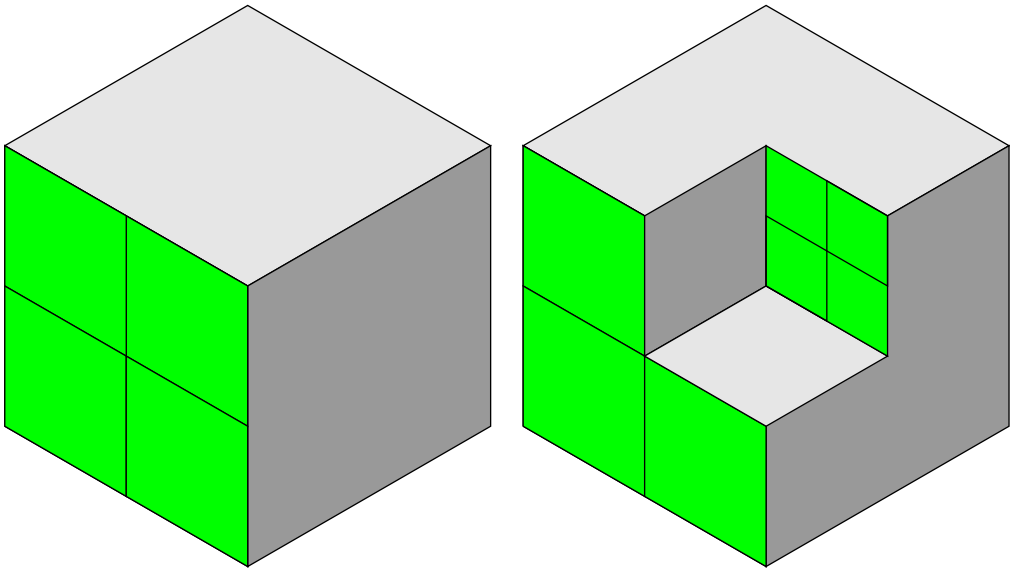


(a) (b)

Figura 6.10: No item (a) vemos o volume de visão normalizado. No item (b) a representação de um feixe no volume normalizado.

Nosso objetivo é escrever um algoritmo que divide a face da imagem de V em quatro partes iguais, e para cada parte tomar o feixe correspondente (Figura 6.11 a). Em cada feixe, procuramos o primeiro bloco que contém pontos da superfície usando *BeamCasting*. O início desse bloco serve como ponto de partida para uma nova execução do algoritmo, de forma recursiva (Figura 6.11 b). Dessa forma, a cada passo, melhoramos nossas estimativas e aceleramos o passo seguinte. O Algoritmo 6.2 detalha esse processo.

Além da função F o Algoritmo 6.2 recebe como entrada vários outros parâmetros. Os intervalos $[u_i, u_f]$ e $[v_i, v_f]$ definem o feixe em termos da área que ele ocupa da imagem. O valor w indica a partir de que profundidade começa o feixe, que sempre termina em w_f, que é o fundo do volume normalizado. O número inteiro d, indica a profundidade em que o algoritmo se encontra, junto do parâmetro $pmax$, ele serve para limitar a quantidade



(a) (b)

Figura 6.11: No item (a) a face da imagem é dividida em quatro partes iguais. No item (b) vemos o algoritmo aplicado recursivamente sobre a parte do feixe inicial que não foi eliminada.

Algoritmo 6.2 Subdivisão no Espaço da Imagem

procedimento *Subdivisão*(F, $[u_i, u_f]$, $[v_i, v_f]$, w, d, ε, $pmax$)
  $[a, b] = BeamCasting(F, [u_i, u_f], [v_i, v_f], [w, w_f], \varepsilon,)$
  se $[a, b] = \varnothing$ **então**
    $I([u + i, u_f], [v_i, v_f]) \leftarrow CordeFundo$
  se não
    se $d = pmax$ **então**
      $c \leftarrow Centro([u_i, u_f], [v_i, v_f], [a, b])$
      $I([u_i, u_f] \times [v_i, v_f]) \leftarrow Cor(c)$
    se não
      $u_m \leftarrow \frac{u_i + u_f}{2}$
      $v_m \leftarrow \frac{v_i + v_f}{2}$
      Subdivisão(F, $[u_i, u_m]$, $[v_i, v_m]$, a, $d + 1$, $\frac{\varepsilon}{2}$, $pmax$)
      Subdivisão(F, $[u_m, u_f]$, $[v_i, v_m]$, a, $d + 1$, $\frac{\varepsilon}{2}$, $pmax$)
      Subdivisão(F, $[u_i, u_m]$, $[v_m, v_f]$, a, $d + 1$, $\frac{\varepsilon}{2}$, $pmax$)
      Subdivisão(F, $[u_m, u_f]$, $[v_m, v_f]$, a, $d + 1$, $\frac{\varepsilon}{2}$, $pmax$)
    fim se
  fim se
fim procedimento

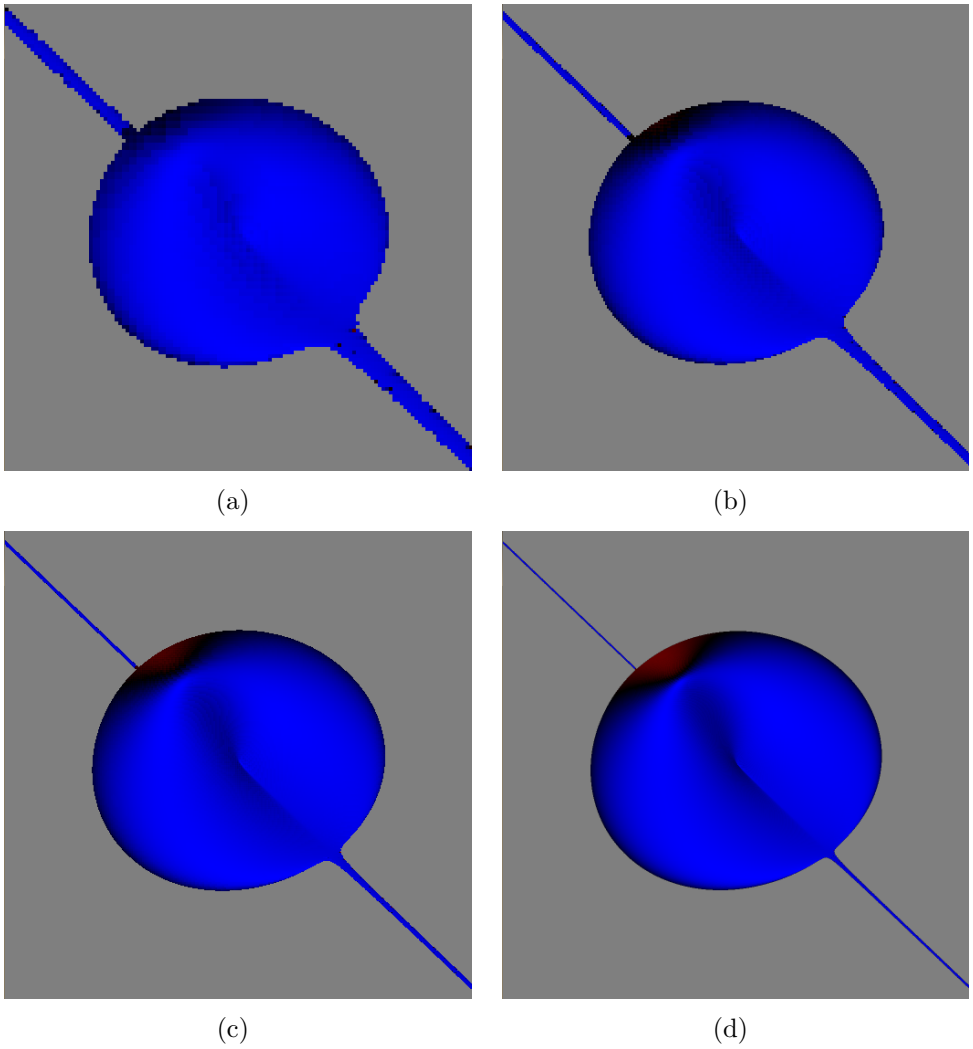
de recursões que o algoritmo executa. Sempre que invocamos *Subdivisão*, passamos $d = 0$. O valor ε é a precisão do algoritmo.

Em cada invocação, *Subdivisão* executa *BeamCast* de forma a encontrar o primeiro bloco $[u_i, u_f] \times [v_i, v_f] \times [a, b]$ com $b - a < \varepsilon$ que supomos conter pontos da superfície. Se nenhum bloco é encontrado ($[a, b] = \varnothing$), então toda área da imagem relativa ao feixe pode receber a cor de fundo. Caso contrário, verificamos se o algoritmo atingiu a profundidade máxima $pmax$, que indica o número máximo de divisões que faremos na face da imagem e, por consequência, a largura mínima dos feixes que iremos testar. No caso em que o algoritmo atingiu a profundidade máxima, tomamos uma amostra no centro do bloco como aproximação para a posição da superfície. A cor calculada nessa amostra define a cor da imagem na região do feixe. O último caso é quando ainda não atingimos a profundidade máxima. Procedemos dividindo a área da imagem relativa ao feixe em quatro partes, para cada parte invocamos o algoritmo recursivamente. Notemos que, nessas invocações, incrementamos o valor da profundidade para garantir que o algoritmo pare e dividimos ε pela metade para aumentar a precisão de *BeamCast*. O algoritmo termina quando toda imagem é definida.

A qualidade da imagem depende da profundidade máxima que o algoritmo atinge; podemos ver alguns exemplos na Figura 6.12. Ela mostra a mesma superfície com diferentes profundidades máximas.

Notemos que, da forma como colocamos o algoritmo, o processo de *anti-aliasing* ocorre naturalmente. Basta que a profundidade máxima seja suficiente para que a área de um pixel tenha várias amostras.

A melhora do desempenho em relação aos métodos anteriores se dá por duas razões. A primeira, como havíamos visto, é que a cada teste com o *BeamCast*, melhoramos a estimativa da profundidade da superfície. A segunda, é que feixes largos podem ser eliminados em etapas iniciais, evitando o desperdício de computação em áreas onde não existem pontos da superfície. A Figura 6.13 mostra como são eliminados os blocos durante o processo. A Tabela 6.4 mostra o desempenho do algoritmo.



(a) (b) (c) (d)

Figura 6.12: Superfície *Crosscap* visualizada com profundidade máxima: 7 (a), 8 (b) 9(c) e 10 (d).

Desempenho de *Subdivisão*			
N^o Amostras	1	4	16
Esfera[2]	692.26	118.17	40.00
Dingdong[3]	701.19	172.31	64.85
Tangle[4]	165.11	53.86	15.02
Heart[6]	603.71	152.64	54.07
Chmutov[8]	193.61	40.67	8.30

Tabela 6.4: Desempenho (fps) de *Subdivisão* com $\varepsilon = 2^{-9}$, 2^{-10} e 2^{-11} respectivamente.

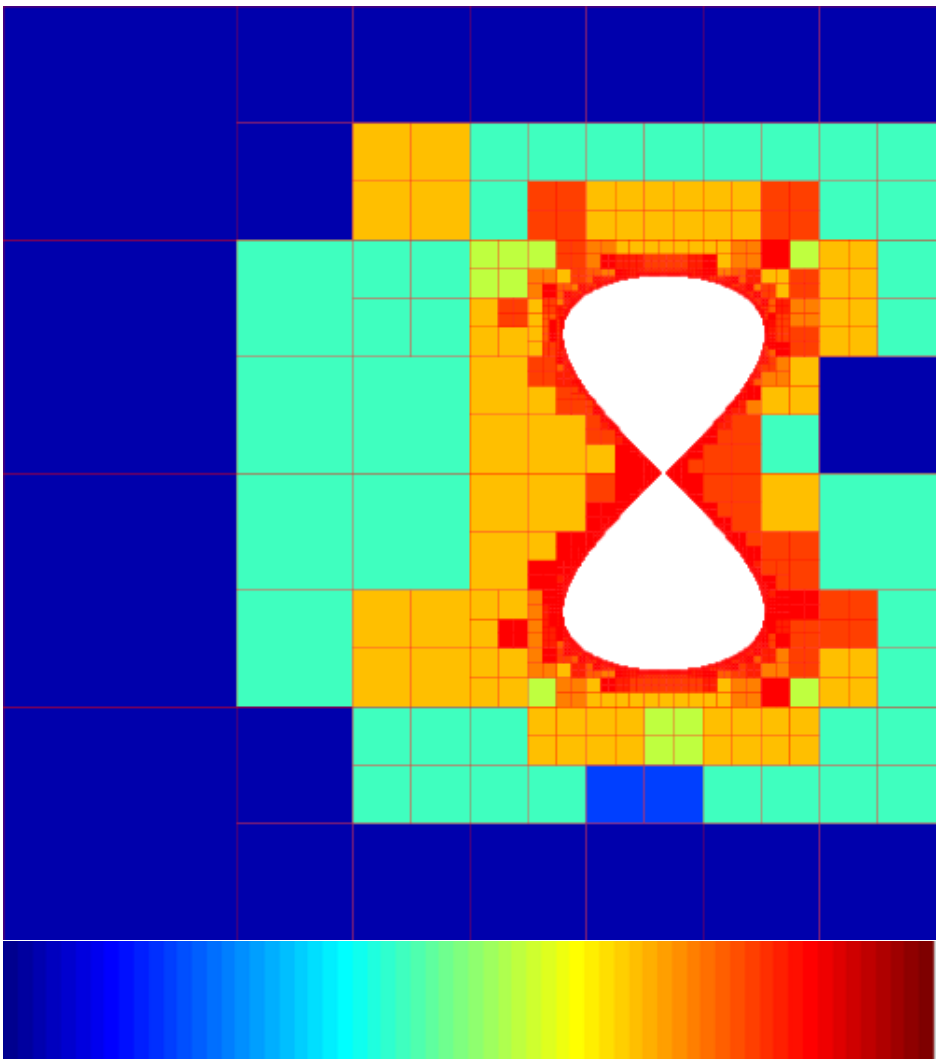


Figura 6.13: Superfície *Octdong*. Os quadrados na imagem mostram a largura dos blocos quando são eliminados. As cores indicam a profundidade na qual o algoritmo começou antes de eliminar o bloco ou encontrar uma raiz.

Capítulo 7

Conclusão

Ao longo deste trabalho abordamos vários aspectos relacionados ao problema de visualizar superfícies implícitas por traçado de raios. Começamos definindo um modelo matemático para o problema, vimos como este modelo pode ser discretizado e depois implementado.

Os algoritmos por amostragem, como havíamos visto, não são robustos. Ao utilizarem um teste de troca de sinais entre as amostras, estes métodos correm o risco de omitir raízes, o que compromete a qualidade da visualização, a geometria da superfície e a iluminação. Esses métodos, portanto, são extremamente sensíveis ao número de amostras tomadas. Vimos que, ao aumentarmos o número de amostras obtemos melhores resultados, porém isto acarreta em uma perda grande de desempenho. A vantagem desses métodos é sua simplicidade de implementação.

Ao usarmos a Aritmética Intervalar tornamos nossos algoritmos robustos, pois podemos eliminar intervalos com garantias de que neles não existem raízes. Em geral, estes métodos produzem resultados de melhor qualidade que os anteriores, e fiéis às características da superfície. Outra característica importante é que estes métodos nos permitem eliminar áreas vazias da imagem com muita rapidez. A dificuldade de implementação é compensada pela qualidade dos resultados. A respeito dos dois métodos apresentados nos Capítulo 3, podemos dizer que embora mais custoso computacionalmente, o Algoritmo de Mitchell pode ter melhor desempenho que Bisecção Intervalar

em alguns casos, mais especificamente quando a superfície tem pouca borda. Nestes casos é mais fácil encontrar intervalos onde a função sobre o raio é monótona. Para superfícies mais complexas, a Bisecção Intervalar tem melhor desempenho. A qualidade dos dois métodos é a mesma, já que ambos são robustos.

Podemos comparar os métodos de amostragem e intervalares da seguinte forma. Com baixa amostragem o primeiro é rápido mas pode produzir resultados muito ruins. Quando aumentamos o número de amostras para compensar a qualidade, percebemos que seu desempenho cai abaixo do desempenho dos algoritmos intervalares. Como vimos nos dados comparativos de desempenho, as placas gráficas atuais são capazes de visualizar superfícies com métodos intervalares a taxas muito acima das necessárias para termos interatividade. Isso nos permite concluir que os métodos de amostragem são necessários apenas em situações onde não dispomos de muito poder computacional, ou quando a função é do tipo caixa-preta.

No Capítulo 5 propusemos e implementamos uma estratégia de *anti–aliasing* adaptativa. Analisamos o *buffer* de profundidade produzido pelos algoritmos anteriores e encontramos os pixels onde são necessárias mais amostras. Desta forma economizamos computação executando mais trabalho somente onde é necessário. Esta estratégia se encaixa muito bem no paradigma de funcionamento da GPU, já que as etapas do processo são todas paralelizáveis. Apesar de ter um custo extra decorrente da implementação de certas etapas, como por exemplo a extração da borda da imagem, esta forma de abordar o problema produz resultados muito bons e de forma eficiente.

No Capítulo 6 vimos um algoritmo inspirado em *beam–tracing* que usa AI para detectar pontos especiais da superfície. Estes pontos, como vimos no caso da superfície *Crosscap*, não são detectados por amostragem pontual. Estendemos esse algoritmo e propusemos um novo método. Esse método usa uma divisão espacial baseada no espaço da imagem. Ao subdividirmos progressivamente a imagem, podemos tomar feixes de raios em diversas escalas e assim eliminar rapidamente grandes volumes vazios. Cada iteração do algoritmo acelera a execução da próxima, que nos dá um ganho de desempenho.

Em nosso trabalho implementamos todos os métodos descritos em CUDA.

Acreditamos que esta escolha foi acertada por vários fatores. Como os programas para CUDA são escritos em C ou C++, pudemos testar o código de nossos algoritmos na CPU e na GPU. Isso nos permitiu realizar diversas comparações importantes. Em CUDA, é possível verificar qual é o custo de uma implementação pelo número de registradores, pelo acesso à memória, número de *threads* concorrentes entre outros, além da existência de *profilers* com o qual coletamos estatísticas sobre o programa. Por exemplo, o Algoritmo de Mitchell utiliza 43 registradores, enquanto que o algoritmo que detecta a borda das imagens usa 8 registradores. Acreditamos que isto é uma vantagem sobre as implementações em GLSL. A possibilidade de escrever em áreas de memória global da GPU durante a execução de um *kernel* e a existência de algoritmos de *scan* e *reduce* nos possibilitaram implementar os algoritmos de *anti–aliasing* que mostramos no Capítulo 5, assim como todos os algoritmos do Capítulo 6.

7.1 Direções Futuras

1. Em nossa implementação, ao executarmos a super–amostragem, estamos supondo o uso de uma reconstrução com filtro do tipo caixa (ou nucleo de Haar). Talvez seja possível implementar de forma eficiente outros tipos de reconstrução aproveitando as possibilidades que a plataforma CUDA oferece. Como por exemplo a comunicação entre *threads* via memória compartilhada.

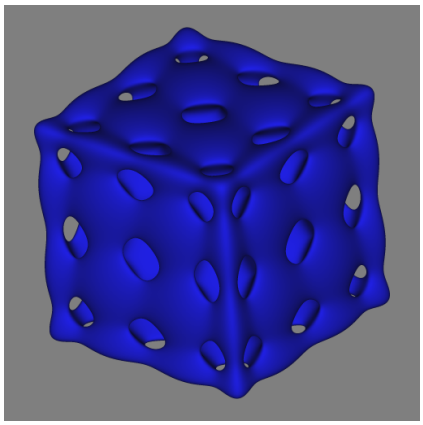
2. Podemos investigar melhor o impacto da divergência nos *warps*, no caso do Algoritmo de Divisão Espacial e nas amostragens feitas nas bordas das imagens.

3. Também podemos empregar a Aritmética Afim como forma de melhorar as estimativas dos algoritmos intervalares. Desta forma esperamos aumentar e precisão e acelerar a convergência dos métodos.

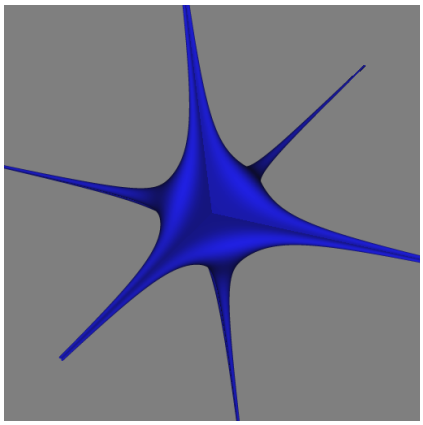
Capítulo 8

Lista de Superfícies e suas Equações

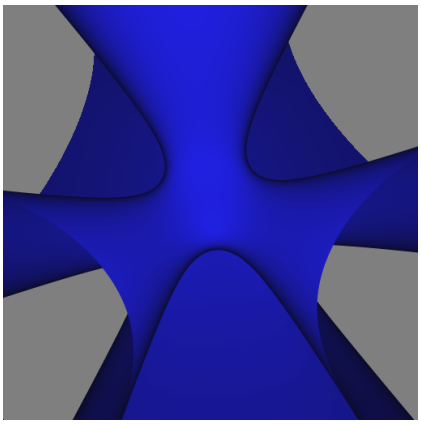
8.1 Superfícies



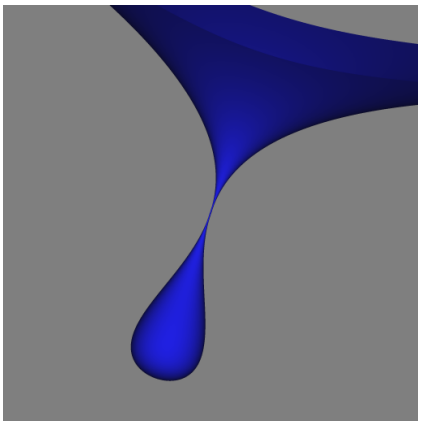
Steiner 1



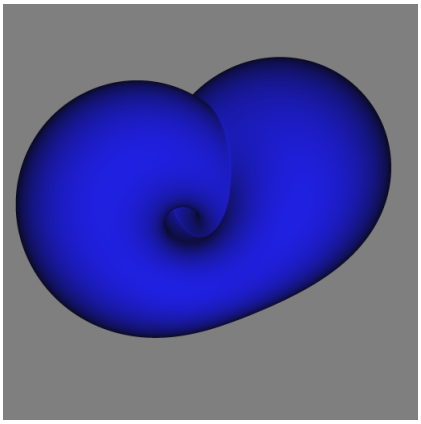
Steiner 2



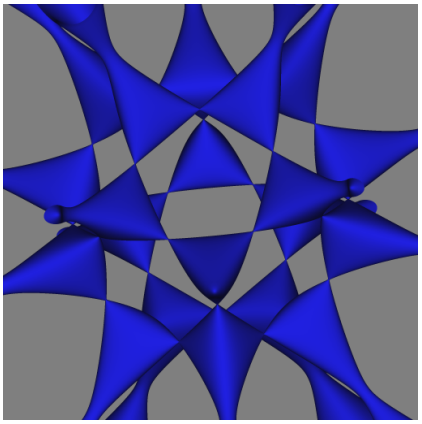
Clebsch



Teardrop

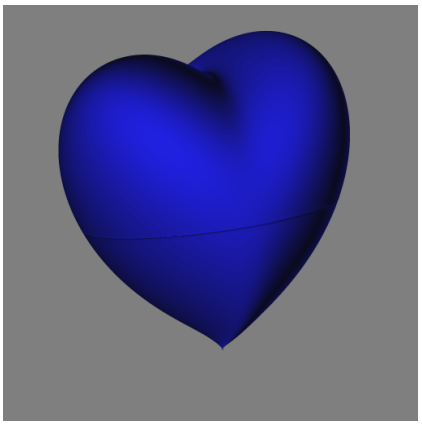


Klein Botle

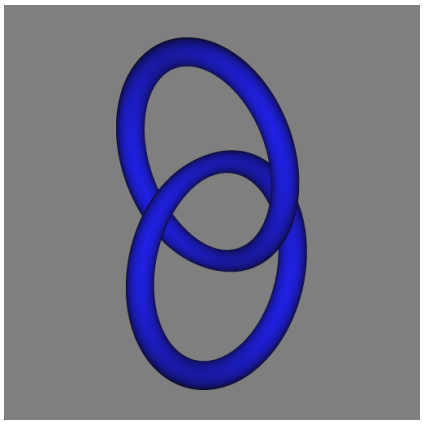


Barth-Sextic

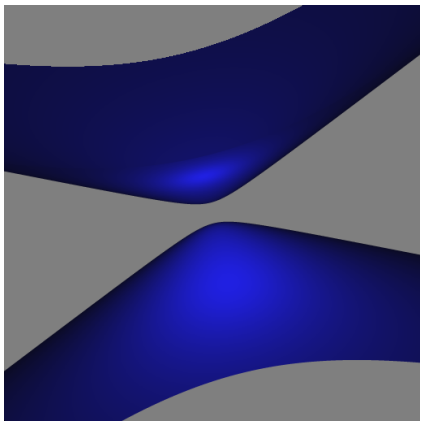
Vis a Vis



Heart



Linked Tori



Hyperboloid

Superfícies Algébricas de Grau 2	
Dattel	$3x^2 + 3y^2 + z^2 - 1$
Gupf	$x^2 + y^2 + z$
Hyperboloid	$-x^2/a^2 - y^2/a^2 + z^2/c^2 - 1$
Intersecting Planes	xy
Kegel	$x^2 + y^2 - z^2$
Parabolic Cylinder	$x^2 - y - r^2$
Sphere	$x^2 + y^2 + z^2 - r^2$
Spindel	$x^2 + y^2 - z^2$
Ufo	$z^2 - x^2 - y^2 - 1$
Zylinder	$y^2 + z^2 - 1$

Superfícies Algébricas de Grau 3	
Clebsch	$81(x^3 + y^3 + z^3) - 189\left(x^2(y+z) + y^2(x+z) + z^2(x+y)\right)$ $+54xyz + 126(xy + xz + zy) - 9(x^2 + y^2 + z^2) + 1$
Cayley	$-5\left(x^2(y+z) + y^2(x+z) + z^2(x+y)\right) + 2(xy + yz + xz)$
Calypso	$x^2 + y^2z = z^2$
Ding-Dong	$x^2 + y^2 - z(1 - z^2)$
Fanfare	$-x^3 + z^2 + y^2$
Harlekin	$x^3z + 10x^2y + xy^2 + yz^2 - z^3$
Kreuz	xyz
Pipe	$x^2 - z$
Sattel	$x^2 + y^2z + z^3$
Whitney	$x^2 - y^2z$

Superfícies Algébricas de Grau 4	
Berg	$x^2 + y^2z^2 + z^3$
Cross Cap	$-a^2y^2 + 4ax^2z + 4x^4 + 4x^2y^2 + 4x^2z^2 + y^4 + y^2z^2 = 0$
Cassini	$((x-a)^2 + y^2)((a+x)^2 + y^2) - z^4$
Columpius	$x^3y + xz^3 + y^3z + z^3 + 7z^2 + 5z$
Diabolo	$x^2 - (y^2 + z^2)^2$
Dullo	$(x^2 + y^2 + z^2)^2 - (x^2 + y^2)$
Dromedar	$x^4 - 3x^2 + y^2 + z^3$
Durchblick	$x^3y + xz^3 + y^3z + z^3 + 5z$
Flirt	$x^2 - x^3 + y^2 + y^4 + z^3 - 10z^4$
Geisha	$x^2yz + x^2z^2 - y^3z - y^3$
Helix	$6x^2 - 2x^4 - y^2z^2$
Herz	$y^2 + z^3 - z^4 - x^2z^2$
Himmel und Hölle	$x^2 - y^2z^2$
Kolibri	$x^3 + x^2z^2 - y^2$
Leopold	$100x^2y^2z^2 + 3x^2 + 3y^2 + z^2 - 1$
Linked Tori	$f(10x, 10y - 2, 10z)f(10z, 10y + 2, 10x) + 1000$ $f(X, Y, Z) = (X^2 + Y^2 + Z^2 + 13)^2 - 53(X^2 + Y^2)$
Miau	$x^2yz + x^2z^2 + 2y^3z + 3y^3$
Mitchell	$4(x^4 + (y^2 + z^2)^2 + 17x^2(y^2 + z^2)) - 20(x^2 + y^2 + z^2) + 17$
Miter	$4x^2(x^2 + y^2 + z^2) - y^2(-y^2 - z^2 + 1)$
Octdong	$x^2 + y^2 + z^4 - z^2$
Piriform	$a^2(y^2 + z^2) - ax^3 + x^4 = 0$
Schneeflocke	$x^3 + y^2z^3 + yz^4$
Steiner 1	$x^2y^2 + y^2z^2 + x^2z^2 + xyz$
Solitude	$x^2yz + xy^2 + y^3 + y^3z - x^2z^2$
Tangle	$x^4 - 5x^2 + y^4 - 5y^2 + z^4 - 5z^2 + 11,8$
Tanz	$x^4 - x^2 - y^2z^2$
Teardrop	$0,5x^5 + 0,5x^4 - y^2 - z^2$
Tobel	$x^3z + x^2 + yz^3 + z^4 - 3xyz$
Tooth	$x^4 + y^4 + z^4 = x^2 + y^2 + z^2$
Trichter	$x^2 + z^3 - y^2z^2$
Tuelle	$yz(x^2 + y - z)$
Vis a vis	$x^2 - x^3 + y^2 + y^4 + z^3 - z^4$
Wedeln	$x^3 - y(1 - z^2)^2$
Windkanal	$-x^2 + y^4 + z^4 - xyz - 100$
Xano	$x^4 + z^3 - yz^2$
Zeck	$x^2 + y^2 - z^3(1 - z)$
Citrus	$x^2 + z^2 + y^3(y - 1)^3$
Zweiloch	$x^3y + xz^3 + y^3z + z^3 + 7z^2 + 5z$

Superfícies Algébricas de Grau 5	
Calyx	$x^2 + y^2z^3 - z^4$
Daisy	$(x^2 - y^3)2 = (z^2 - y^2)^3$
Michelangelo	$x^2 + y^4 + y^3z^2$
Sofa	$x^2 + y^3 + z^5$
Taube	$256z^3 - 128x^2z^2 + 16x^4z + 144xy^2z - 4x^3y^2 - 27y^4$
Wigwam	$x^2 + y^2z^3$
Zeppelin	$xyz + yz + 2z^5$

Superfícies Algébricas de Grau 6	
Barth-Sextic	$4(c^2x^2 - y^2)(c^2y^2 - z^2)(c^2z^2 - x^2) - (1 + 2c)(x^2 + y^2 + z^2 - 1)^2$ $c = (1 + \sqrt{5})/2$
Buggle	$x^4y^2 + y^4x^2 - x^2y^2 + z^6$
Crixxi	$(y^2 + z^2 - 1)^2 + (x^2 + y^2 - 1)^3$
Cube	$x^6 + y^6 + z^6 - 1$
Distel	$x^2 + y^2 + z^2 + 1000(x^2 + y^2)(x^2 + z^2)(y^2 + z^2) - 1$
Eistüte	$(x^2 + y^2)^3 - 4x^2y^2(z^2 + 1)$
Eve	$0,5x^2 + 2xz^2 + 5y^6 + 15y^4 + 0,5z^2 - 15y^5 - 5y^3$
Klein Bottle	$(x^2 + y^2 + z^2 + 2y - 1)((x^2 + y^2 + z^2 - 2y - 1)^2 - 8z^2)$ $+16xz(x^2 + y^2 + z^2 - 2y - 1)$
Limao	$x^2 - y^3z^3$
Nepali	$(xy - z^3 - 1)^2 + (x^2 + y^2 - 1)^3$
Pilzchen	$(z^3 - 1)^2 + (x^2 + y^2 - 1)^3$
Plop	$x^2 + (z + y^2)^3$
Polsterzipf	$(x^3 - 1)^2 + (y^3 - 1)^2 + (z^2 - 1)^3$
Seepferdchen	$(x^2 - y^3)^2 - (x + y^2)z^3$
Heart	$(x^2 + 9/4y^2 + z^2 - 1)^3 - x^2z^3 - 9/80y^2z^3$
Stern	$400(x^2y^2 + y^2z^2 + x^2z^2) + (x^2 + y^2 + z^2 - 1)^3$
Subway	$x^2y^2 + (z^2 - 1)^3$
Twilight	$(z^3 - 2)^2 + (x^2 + y^2{-^3})^3$

Superfícies Algébricas de Grau 7	
Chmutov	$f(x) + f(y) + f(z) + 1$ $f(X) = 64x^7 - 112x^5 + 56x^3 - 7x$

Superfícies Algébricas de Grau 8	
Steiner	$(x^2y^2 + y^2z^2 + x^2z^2)^2 + xyz$
Chmutov	$f(x) + f(y) + f(z)$ $f(X) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1$

Referências Bibliográficas

[1] H. Blinn, *How to solve a quadratic equation?*, Computer Graphics and Applications, IEEE **25** (2005), no. 6, 76–79.

[2] J. Bloomenthal and C. Bajaj, *Introduction to implicit surfaces*, Morgan Kaufmann, 1997.

[3] O. Caprani, L. Hvidegaard, M. Mortensen, and T. Schneider, *Robust and efficient ray intersection of implicit surfaces*, Reliable Computing **6** (2000), no. 1, 9–21.

[4] J.L.D. Comba and J. Stolfi, *A afine arithmetic and its applications to computer graphics*, VI SIBGRAPI, 1990, pp. 9–18.

[5] A. de Cusatis Jr, L.H. De Figueiredo, and M. Gattass, *Interval methods for ray casting implicit surfaces with affine arithmetic*, XII SIBGRAPI, IEEE, 2002, pp. 65–71.

[6] J. Flórez, M. Sbert, M. Sainz, and J. Vehí, *Improving the interval ray tracing of implicit surfaces*, Advances in Computer Graphics (2006), 655–664.

[7] ———, *Guaranteed Adaptive Antialiasing Using Interval Arithmetic*, Computational Science–ICCS 2007 (2007), 166–169.

[8] ———, *Efficient ray tracing using interval analysis*, Parallel Processing and Applied Mathematics (2008), 1351–1360.

[9] A.S. Glassner, *Principles of digital image synthesis*, Morgan Kaufmann Publishers, 1995.

[10] A.J.P. Gomes, I. Voiculescu, J. Jorge, B. Wyvill, and C. Galbraith, *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*, Springer Verlag, 2009.

[11] Jonas Gomes and Luiz Velho, *Fundamentos da computação gráfica*, IMPA, 2003.

[12] P. Hanrahan, *Ray tracing algebraic surfaces*, ACM SIGGRAPH Computer Graphics **17** (1983), no. 3, 83–90.

[13] M. Harris, S. Sengupta, and J.D. Owens, *Parallel prefix sum (scan) with CUDA*, GPU Gems **3** (2007), no. 39, 851–876.

[14] J.C. Hart, *Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces*, The Visual Computer **12** (1996), no. 10, 527–545.

[15] J.C. Hart, E. Bachta, W. Jarosz, and T. Fleury, *Using particles to sample and control more complex implicit surfaces*, Shape Modeling International, 2002. Proceedings, IEEE, 2002, pp. 129–136.

[16] P.S. Heckbert and P. Hanrahan, *Beam tracing polygonal objects*, Proceedings of the 11th annual conference on Computer graphics and interactive techniques, ACM, 1984, pp. 119–127.

[17] D. Herbison-Evans, *Solving quartics and cubics for graphics*, Aademic Press, 1995.

[18] B. Jones, WG Waller, and A. Feldman, *Root isolation using function values*, BIT Numerical Mathematics **18** (1978), no. 3, 311–319.

[19] D. Kalra and A.H. Barr, *Guaranteed ray intersections with implicit surfaces*, SIGGRAPH'89, ACM, 1989, pp. 297–306.

[20] D.B. Kirk and W.H. Wen-mei, *Programming massively parallel processors: A Hands-on approach*, Morgan Kaufmann, 2010.

[21] A. Klimovitski, *Using SSE and SSE2: Misconceptions and reality*, Intel Developer UPDATE Magazine **6** (2001).

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

[23] B. Lambov, *Interval arithmetic using SSE-2*, Reliable Implementation of Real Number Algorithms: Theory and Practice (2008), 102–113.

[24] M.E. Lee, R.A. Redner, and S.P. Uselton, *Statistically optimized sampling for distributed ray tracing*, Proceedings of the 12th annual conference on Computer graphics and interactive techniques, ACM, 1985, pp. 61–68.

[25] C.T. Loop and J.F. Blinn, *Real-time GPU rendering of piecewise algebraic surfaces*, SIGGRAPH'06, ACM, 2006.

[26] ———, *Resolution-independent curve rendering using programmable graphics hardware*, July 21 2009, US Patent 7,564,459.

[27] W.E. Lorensen and H.E. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, SIGGRAPH'87, ACM, 1987, pp. 163–169.

[28] D.P. Mitchell, *Generating antialiased images at low sampling densities*, Proceedings of the 14th annual conference on Computer graphics and interactive techniques, ACM, 1987, p. 72.

[29] ———, *Robust ray intersection with interval arithmetic*, Proceedings of Graphics Interface, vol. 90, Halifax, 1990, pp. 68–74.

[30] R.E. Moore, *Interval analysis*, Englewood Cliffs, New Jersey (1966).

[31] D. Nehab and M. Gattass, *Ray path categorization*, sibgraphi (2000), 227.

[32] Nvidia, *Compute Unified Device Architecture Programming Guide*, NVIDIA: Santa Clara, CA (2007).

[33] A. Paiva, H. Lopes, T. Lewiner, and L.H. de Figueiredo, *Robust adaptive meshes for implicit surfaces*, SIBGRAPI'06, IEEE, 2006, pp. 205–212.

[34] M. Reimers and J. Seland, *Ray casting algebraic surfaces using the frustum form*, Computer Graphics Forum, vol. 27, John Wiley & Sons, 2008, pp. 361–370.

[35] P. Rokita, *Depth-based selective antialiasing*, Journal of Graphics, GPU, & Game Tools **10** (2005), no. 3, 19–26.

[36] F. Romeiro, L. Velho, and L.H. de Figueiredo, *Scalable GPU rendering of CSG models*, Computers & Graphics **32** (2008), no. 5, 526–539.

[37] S.J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, Prentice Hall, 2009.

[38] T.W. Sederberg, *Piecewise algebraic surface patches*, Computer Aided Geometric Design **2** (1985), no. 1-3, 53–59.

[39] J.S. Seland and T. Dokken, *Real time algebraic surface visualization*, Geometric modelling, numerical simulation, and optimization: applied mathematics at SINTEF (2007), 163.

[40] H. Shou, W. Song, J. Shen, R. Martin, and G. Wang, *A Recursive Taylor Method for Ray-Casting Algebraic Surfaces*, Proceedings of the 2006 International Conference on Computer Graphics & Virtual Reality, CGVR, Citeseer, 2006, pp. 26–29.

[41] HH Shou, R. Martin, I. Voiculescu, A. Bowyer, and GJ Wang, *Affine arithmetic in matrix form for polynomial evaluation and algebraic curve drawing*, Progress in Natural Science **12** (2002), no. 1, 77–81.

[42] H. Shoua, H. Linb, R. Martinc, and G. Wangb, *Modified Affine Arithmetic in Tensor Form*, The Proceedings of International Symposium on

Computing and Information (ISC&I 2004), Zhuhai, Guangdong, China, August, Citeseer, 2004, pp. 15–18.

[43] J.M. Singh and PJ Narayanan, *Real-Time Ray Tracing of Implicit Surfaces on the GPU*, IEEE Transactions on Visualization and Computer Graphics **16** (2010), no. 2, 261–272.

[44] A.R. Smith, *A Pixel Is Not A Little Square, A Pixel Is Not A Little Square, A Pixel Is Not A Little Square!(And a Voxel is Not a Little Cube*, Technical Memo 6, Microsoft Research, Citeseer, 1995.

[45] G. Varadhan, S. Krishnan, L. Zhang, and D. Manocha, *Reliable implicit surface polygonization using visibility mapping*, Proceedings of the fourth Eurographics symposium on Geometry processing, Eurographics Association, 2006, pp. 211–221.

[46] Luiz Velho and Jonas Gomes, *Sistemas gráficos 3d*, Série Computação e Matemática, SBM / IMPA, 2001.

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

[48] G. Xu, H. Huang, and C. Bajaj, *C1 modeling with A-patches from rational trivariate functions*, Computer Aided Geometric Design **18** (2001), no. 3, 221–244.