Capítulo 1 Introdução

1.1 Motivação

Superfícies implícitas são ferramentas poderosas na área de Modelagem Geométrica, em Computação Gráfica, principalmente quando combinadas entre si, através de técnicas como CSG (*Construtive Solid Geometry*) [Requicha 80] e operações de *blending* [Bloomenthal et al. 97], permitindo a construção e a manipulação de objetos complexos. Entretanto, a visualização de instâncias não triviais destas superfícies ainda não é uma tarefa das mais simples em Computação Gráfica.

Dentre as técnicas disponíveis para a visualização de objetos implícitos, a mais natural parece ser o ***raycasting*** (lançamento de raios) [Appel 68], que apresenta algumas importantes vantagens sobres outras alternativas, como a "poligonalização" para posterior aplicação de técnicas de visualização tradicionais para polígonos. Entre estas vantagens, estão a simplicidade conceitual, a economia de memória pela visualização direta (sem a necessidade de construção de grandes estruturas de dados) e a geração de imagens exatas e realísticas.

Atualmente, o *raycasting* constitui uma técnica já bastante explorada na literatura, e implementada em inúmeros sistemas. Porém, muitas destas implementações prevêem apenas a visualização de um pequeno conjunto de superfícies simples (esferas, planos, quádricas genéricas, etc.), geralmente utilizadas como primitivas na construção de objetos mais complexos, ou de umas poucas classes de superfícies, consideradas "bem comportadas", para as quais são bem conhecidas fórmulas para a solução do problema central do *raycasting*, que é o de determinar as interseções das superfícies com os raios lançados na visualização da cena.

Não há, na literatura, muitos exemplos de técnicas robustas e eficientes para a solução deste problema no caso genérico, isto é, para superfícies arbitrárias. Uma técnica robusta – embora computacionalmente cara – é a aplicação de **métodos intervalares** à resolução das equações de interseção. Tais métodos, cuja utilização em Computação Gráfica é razoavelmente recente [Mitchell 90], [Snyder 92], têm encontrado na **aritmética intervalar** tradicional, IA [Moore 66], a ferramenta natural para sua implementação, considerando-se que até alguns anos atrás, tratava-se do principal modelo para computação numérica com intervalos, se não o único.

A **aritmética afim** (AA) [Comba & Stolfi 93] proposta como uma alternativa a IA, foi projetada com o objetivo de acelerar métodos intervalares em diversas aplicações. Embora os cálculos individuais com AA tendam a ser mais caros do que com IA, a sua maior precisão em sequências de cálculos pode levar diversos algoritmos a uma convergência mais rápida, com a necessidade de menos passos, compensando o custo extra e resultando num melhor desempenho global. No entanto, cada classe de aplicações exige investigação em separado.

1.2 Objetivos

O principal objetivo deste trabalho é aplicar a aritmética afim ao *raycasting* de superfícies implícitas genéricas, utilizando alguns aspectos numéricos específicos de AA para tentar acelerar o cálculo intervalar das interseções entre raio e superfície. São implementadas diferentes versões (intervalares e híbridas) do algoritmo de *raycasting*, cada uma delas utilizando IA e AA nas contas com intervalos, e é feita a comparação entre seus desempenhos relativos. Deseja-se, particularmente, avaliar se a utilização de AA fornece algum ganho significativo de eficiência em relação à utilização de IA.

A organização do restante deste texto é apresentada na próxima seção, através de uma descrição resumida do conteúdo dos demais capítulos.

1.3 Organização do texto

O capítulo 2 introduz as superfícies implícitas, começando com a definição da representação matemática na forma implícita, e com alguns exemplos. É apresentada também, de forma resumida, uma comparação com outra forma de descrição matemática útil, a paramétrica. O texto tenta também ilustrar a importância das superfícies implícitas na área de modelagem, através das duas principais técnicas para a sua combinação em modelos mais complexos: operações booleanas e operações de *blending*. Por fim, são citadas outras técnicas disponíveis para a visualização de superfícies e modelos implícitos.

O capítulo 3 discute em detalhes a técnica de *raycasting* de superfícies implícitas. O algoritmo é formalizado, com destaque para a importância do cálculo das interseções e é apresentada uma proposta de utilização de duas etapas na abordagem genérica deste cálculo. Ao final do capítulo, é apresentado um breve resumo dos principais trabalhos relacionados a esta linha de pesquisa encontrados na literatura.

No capítulo 4, são comentados os métodos numéricos para a resolução de equações, ressaltando-se a diferença entre métodos numéricos convencionais e intervalares e a utilidade de cada tipo em diferentes etapas do cálculo das interseções. São apresentados três métodos convencionais – bisseção, *regula-falsi,* e Brent – utilizados na implementação deste trabalho. Como representantes dos métodos intervalares, são apresentados o algoritmo de Moore – talvez o principal exemplo encontrado na literatura – e uma versão simplificada (sem o uso de derivada) deste algoritmo, à qual iremos nos referir, por vezes, como algoritmo de bisseção intervalar.

No capítulo 5, são apresentados os dois principais modelos de aritmética com intervalos – aritmética intervalar tradicional e aritmética afim – que é uma ferramenta necessária e natural para a implementação dos métodos intervalares. Para cada um dos modelos, são definidas a forma de representação dos intervalos e as principais operações aritméticas, além de algumas propriedades destas operações. É feita também uma comparação entre os resultados gerados por cada um deles, ressaltando que AA, por armazenar informação extra, relacionada às correlações entre variáveis e sub-fórmulas de uma mesma expressão, implica em cálculos mais caros mas fornece intervalos mais precisos. É apresentada também uma

idéia de otimização, baseada em aspectos específicos de AA, que pode ser aplicada a cada passo de algoritmos intervalares, a custo quase zero.

O capítulo 6 aborda questões e detalhes relativos à implementação, como as bibliotecas que implementam IA e AA e sua utilização no cálculo intervalar de um grande número de funções. É apresentado também o *SuperRay*, programa desenvolvido para a visualização de superfícies implícitas genéricas, em duas versões (IA e AA).

O capítulo 7 descreve os testes de desempenho realizados, os exemplos de superfícies utilizados, e apresenta os resultados dos testes.

No capítulo 8, são apresentadas algumas conclusões, baseadas na análise dos resultados e no trabalho como um todo.

Finalmente, o capítulo 9 − Trabalhos Futuros − aponta algumas direções interessantes para a possível continuidade deste trabalho.

Capítulo 2 Superfícies implícitas

2.1 Introdução

A Computação Gráfica é o ramo da Ciência da Computação que trata da *síntese de imagens digitais*, ou seja, a construção de imagens no computador, ou através dele a partir de dados numéricos e modelos matemáticos, os quais representam objetos ou fenômenos que deseja-se simular ou representar graficamente. De acordo com este objetivo, costuma-se dividir a Computação Gráfica em duas grandes áreas: **Modelagem** e **Visualização**. Enquanto a primeira preocupa-se com a obtenção ou geração dos dados e modelos e sua representação adequada no computador, a segunda trata da tarefa de exibir imagens corretas e convincentes de cenas contendo os objetos e fenômenos representados, com um maior ou menor grau de realismo, dependendo da aplicação esperada.

Um dos problemas essenciais tratados na Modelagem é a forma de representação dos elementos que compõem uma cena. Há muitos esquemas de representação disponíveis, adequados a diferentes objetivos; uma discussão mais completa pode ser encontrada em [Foley et al. 90] e [Watt 93]. Em algumas situações, porém, a escolha mais adequada é utilizar equações matemáticas para representar os elementos (ou partes destes), que costumam ser objetos geométricos como curvas, superfícies ou sólidos. Entre as principais vantagens desta forma de representação, estão o poder de concisão (que geralmente implica em economia de memória), e a capacidade de descrever muitos objetos de maneira exata, não aproximada. Interessa-nos particularmente o caso das **superfícies** – entidades geométricas bi-dimensionais imersas no espaço tridimensional [Bloomenthal et al. 97] – que historicamente têm se mostrado úteis e flexíveis na construção de objetos tridimensionais complexos. Colocando-se de modo simplificado, há duas formas básicas de se descrever superfícies matematicamente: **implícita** e **paramétrica**. As duas formas são apresentadas na próxima seção.

2.1.1 Definição

Na representação implícita, as coordenadas do espaço tridimensional, *x*, *y* e *z*, são tomadas como argumentos de uma função *f*, que na maioria das aplicações é escalar. Uma superfície implícita ***S*** é definida como o conjunto dos pontos do espaço que satisfazem à equação $f(x,y,z) = 0$. Formalmente:

$$S = \{ \ (x,y,z) \in \Re^3 \ \big| \ f(x,y,z) = 0 \ \}$$

É comum referir-se abreviadamente à função $f(x,y,z)$ como sendo a própria superfície. A figura 2.1 (gerada com o programa MapleV) ilustra o exemplo clássico da esfera de raio unitário, com centro na origem do sistema de coordenadas, que pode ser expressa por:

$$x^2 + y^2 + z^2 - 1 = 0$$

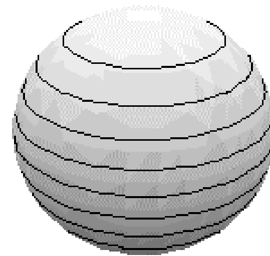


Figura 2.1 - Esfera definida na forma implícita.

Um plano genérico tem sua representação implícita na forma:

$$ax + by + bz - d = 0$$

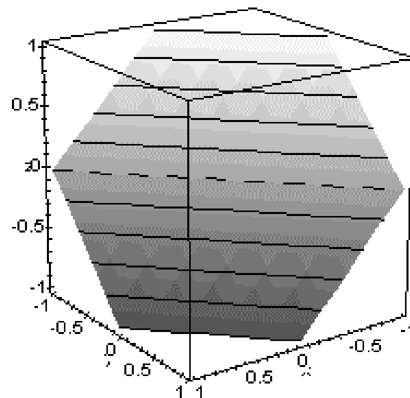


Figura 2.2 - Plano definido na forma implícita.

Na outra forma de representação, a paramétrica, cada uma das coordenadas é expressa como função de uma ou mais variáveis independentes, chamadas parâmetros. O número de parâmetros utilizado na representação de um objeto é igual ao número de dimensões geométricas deste objeto (dois, no caso de superfícies):

$$x = f_1(u,v); \qquad y = f_2(u,v); \qquad z = f_3(u,v);$$

onde u e v são as variáveis de parâmetro, uma para cada dimensão. Para a esfera citada anteriormente, por exemplo, uma possível parametrização é dada por:

$$x = f_1(\varphi,\phi) = sen\varphi\, cos\phi$$

$$y = f_2(\varphi,\phi) = sen\varphi\, sen\phi \qquad \varphi \in [0, \pi]$$

$$z = f_3(\varphi,\phi) = cos\varphi \qquad \phi \in [0, 2\pi),$$

um caso particular do sistema de coordenadas esféricas (quando o raio é fixado em 1), onde φ e ϕ são chamados, respectivamente, co-latitude e longitude de um ponto (x,y,z) na esfera.

2.1.2 Representação implícita × paramétrica

Não há necessariamente uma forma de representação melhor, dentre as duas. Cada uma delas oferece vantagens e desvantagens em relação à outra, mostrando-se mais ou menos adequada em determinada situação.

A forma implícita, por exemplo, oferece maior facilidade na localização de pontos arbitrários em relação a uma superfície. Por exemplo, se temos um ponto $P=(x_p, y_p, z_p)$ e uma superfície fechada, dada por: $f(x,y,z) = 0$**,** basta substituirmos as coordenadas de P em f e teremos três possibilidades:

$$f(x_p,y_p,z_p) \Rightarrow \begin{cases} < 0 \rightarrow \text{ponto } P \text{ é interior à superfície;} \\ = 0 \rightarrow \text{ponto } P \text{ está sobre a superfície;} \\ > 0 \rightarrow \text{ponto } P \text{ é exterior à superfície.} \end{cases}$$

Já a forma paramétrica oferece grande facilidade em operações que, de alguma forma, necessitem "percorrer" uma superfície – ou, a partir de um ponto, determinar um outro ponto

vizinho – muito úteis em aplicações como mapeamento de texturas e exibição por rasterização, comuns em visualização de polígonos.

O cálculo do vetor normal a um ponto de uma superfície também é bastante simples na representação implícita (apesar de não ser muito mais complexo na forma paramétrica). As coordenadas da normal num ponto p são obtidas como os valores das derivadas parciais de f no ponto p:

$$(\partial f/\partial x , \partial f/\partial y , \partial f/\partial z) \text{ em } p.$$

Entretanto, independentemente das vantagens e desvantagens de cada representação, nos últimos anos tem crescido bastante o interesse em torno de superfícies implícitas, estimulando e sendo estimulado por uma série de trabalhos que introduziram técnicas mais sofisticadas e eficientes para sua manipulação e visualização, e levando a um incremento em sua utilização, tanto na comunidade de Computação Gráfica como em segmentos relacionados, como a Engenharia e a indústria do entretenimento. Um excelente texto a respeito é encontrado em [Bloomenthal et al. 97].

2.2 Aplicações de superfícies implícitas em Modelagem

Instâncias isoladas de superfícies implícitas são interessantes do ponto de vista da investigação matemática, mas possuem uma aplicação reduzida na construção de objetos e cenas. É através da possibilidade de combinação em modelos mais complexos que tais superfícies tornam-se uma ferramenta de modelagem bem mais poderosa, com aplicação em áreas como Animação, CAD e Visualização Científica. Segundo [Gomes & Velho 92], as duas principais técnicas de combinação de elementos implícitos são as operações booleanas, essenciais na modelagem CSG, e as operações de *blending*. Uma breve introdução a estas duas técnicas é apresentada a seguir.

2.2.1 Operações booleanas e modelagem CSG

Historicamente, a forma mais comum de utilização de objetos implícitos em Computação Gráfica é nos sistemas de modelagem **CSG** (***Construtive Solid Geometry***), bastante populares desde que a técnica foi apresentada, no início da década de oitenta, em

[Requicha 80], e que permitem modelar sólidos matematicamente complexos por um processo construtivo.

A modelagem CSG constitui, ao mesmo tempo, uma técnica de construção e um esquema de representação para os sólidos geométricos presentes numa cena. No processo de construção, o usuário dispõe de um conjunto de primitivas – objetos ou superfícies com uma geometria simples – quase sempre representadas na forma implícita. Sólidos mais complexos podem ser construídos gradualmente, a partir destas primitivas. As mais comuns, que estão presentes na maioria das implementações, são esferas, caixas, pirâmides, faces planas, cones, cilindros, e superfícies quádricas genéricas.

Estes elementos são combinados através das operações binárias clássicas sobre conjuntos: união, interseção e diferença, também conhecidas como operações booleanas, e que são sempre aplicadas a pares de elementos. Na verdade, estas operações devem ser aplicadas segundo certas restrições (**operações regularizadas**), de forma a garantir certas características desejáveis aos sólidos resultantes. Em poucas palavras, deve-se garantir que o resultado de cada operação seja um sólido cuja fronteira seja considerada uma **variedade (*manifold*)**, um conceito fundamental em Modelagem Geométrica [Gomes & Velho 92].

Antes de uma operação ser aplicada, os elementos podem ser posicionados e dimensionados através de transformações afins, como translação, rotação, reflexões e mudanças de escala. Ao longo da construção de um objeto complexo, informações semelhantes a um "histórico" vão sendo armazenadas em uma estrutura de dados conhecida como Árvore CSG, que é semelhante a uma árvore binária comum, mas que deve ser heterogênea, pois distingue-se ao menos dois tipos de informação a serem armazenadas nos nós: folhas representam primitivas, enquanto nó intermediários representam uma operação booleana ou uma transformação afim.

O que torna a técnica ainda mais poderosa e interessante é a possibilidade de objetos complexos, obtidos por este processo, terem sua definição armazenada, e passarem a ser considerados como novas primitivas.

2.2.2 Operações de *blending*

Um fator que tem contribuído bastante para o aumento da popularidade de superfícies implícitas nos últimos anos é a combinação de tais superfícies em modelos mais complexos através de operações de *blending*. Esta técnica tem sido utilizada, sobretudo, nas áreas de modelagem, animação e CAD [Bloomenthal et al. 97]. Um ***blend*** de duas ou mais superfícies – as quais podem ou não interceptar-se – é uma forma geométrica que, de maneira informal, equivale a uma transição suave entre as superfícies originais [Gomes & Velho 92].

Objetos implícitos são particularmente adequados para operações de *blending*. Dado um conjunto de objetos ou superfícies definidos implicitamente, $f_1(x), f_2(x), \ldots f_n(x)$, obtém-se um *blend* basicamente através da composição das funções: $B(\ f_1(x),\ \ldots\ f_n(x)\ )$, segundo alguma regra de composição, de forma que se obtenha também um objeto implicitamente definido $B{:}R^n \rightarrow R$.

Uma operação de *blending* pode ser aplicada a um conjunto de objetos implícitos de duas formas básicas: globalmente ou localmente. Num *blending* global, todos os objetos são considerados e exercem influência na forma final de cada região do *blend* resultante, mesmo que de modo não homogêneo (pode-se atribuir, por exemplo, um peso a cada objeto, ou a influência de um objeto em cada ponto ser função da distância). Já *blendings* locais podem ser definidos restringindo-se a aplicação da operação de *blending* de duas formas básicas: ela pode atuar apenas sobre certos subconjuntos de objetos (por exemplo, atuando a cada vez em um par de objetos, que podem ser armazenados em uma estrutura semelhante a uma árvore CSG), ou pode-se limitar espacialmente o domínio no qual a função de *blending* é computada, de forma que a operação influa somente em algumas regiões de alguns objetos. Esta segunda opção é bastante utilizada em CAD, para atenuar "sulcos" e saliênicias, que normalmente resultam de operações de união ou interseção entre duas peças.

A forma mais simples de *blending* é o *blending* linear, dado por

$$B(\ f_1(x),\ \ldots\ f_n(x)\ ) = \sum_{i=1,n} [\ f_i(x) - 1\ ],$$

que oferece como vantagens a simplicidade conceitual e facilidade de implementação e a eficiência no cálculo. Entretanto, as superfícies resultantes tendem a ser pouco fiéis às originais (não apenas suavizam as interseções mas acrescentam volume aos objetos, mesmo em regiões que estão distantes de qualquer interseção).

Dois outros tipos comuns de *blend*, baseados em diferentes opções de combinação das funções, são o *blending* hiperbólico [Kleck 89] e o super-elíptico [Rockwood & Owen 87]. O *blending* hiperbólico, dado por

$$B(\ f_1(x), \dots f_n(x)\ ) = \pi_{i=1,n} [\ f_i(x) - 1\ ],$$

fornece resultados melhores do que o *blending* linear, mas também aproximados, isto é, não totalmente fiéis às primitivas originais. Já o *blending* super-elíptico fornece resultados exatos.

2.3 Visualização de superfícies implícitas

Ao longo das últimas décadas, a representação implícita não tem tido uma utilização comercial tão grande quanto outras, como malhas de polígonos, malhas de *patches* paramétricos, ou mesmo algumas estruturas espaciais, úteis em visualização volumétrica.

Mesmo assim, há um razoável conjunto de técnicas disponíveis para a visualização de objetos implícitos, que permitem a sua exibição baseada nas três formas possíveis em Visualização: por pontos, por linhas ou pelo preenchimento contínuo de regiões (*shading*).

Na exibição por pontos, a principal abordagem é usar sistemas de partículas para gerar pontos distribuídos no interior da superfície e fazê-los migrar até a superfície e projetá-los na tela [Bloomenthal et al. 97]. No caso das linhas, há técnicas não muito complexas para se calcular curvas de contorno ou curvas de nível sobre as superfícies e exibi-las. Já a abordagem de "percorrer" as superfícies, gerando malhas de meridianos e paralelos, comum na exibição de *patches* paramétricos, é bem mais difícil na representação implícita.

Para o preenchimento contínuo, que gera imagens mais interessntes, sendo mais adequado em diversas aplicações, há duas abordagens: "renderizar" a superfície diretamente, a partir de sua equação matemática – e, neste caso, a técnica natural é o *raycasting*, discutido em detalhes no próximo capítulo – ou aproximá-la por um outro esquema de representação, o que permite a utilização de técnicas de visualização específicas (algumas vezes baratas e de fácil acesso, como as técnicas para polígonos) mas, por outro lado, tende a gerar estruturas de dados que podem vir a consumir grandes quantidades de memória. É o caso da

“poligonalização”, uma aproximação linear por partes da superfície, gerando uma malha de polígonos. Um algoritmo interessante é comentado e implementado em [Bloomenthal 94]. A maioria das outras técnicas de aproximação das superfícies é baseada em subdivisão espacial, com a geração de estruturas de dados espaciais, como *octrees* [Foley et al. 90]. Um exemplo é a enumeração adaptativa descrita em [Figueiredo & Stolfi 96]. Estas estruturas podem ser processadas por algoritmos de visualização volumétrica, mas também podem ser usadas na aceleração do *raycasting* [Glassner 84].

Uma discussão mais completa sobre “poligonalização” e outras técnicas é encontrada em [Bloomenthal et al. 97] e em [Gomes & Velho 92]. A técnica de *raycasting* direto (a “renderização” direta, a partir das equações, sem nenhuma aproximação ou técnica intermediária), que interessa efetivamente a este trabalho, é discutida em detalhes no próximo capítulo.

Capítulo 3 Raycasting de superfícies implícitas

3.1 O algoritmo de *raycasting*

Um dos primeiros trabalhos a utilizar a idéia do lançamento de raios para visualizar cenas foi [Appel 68]. Desde então, este tornou-se um dos paradigmas mais difundidos em Visualização. A idéia é simples: lançar "raios de visão" a partir do ponto de vista do observador da cena, em direção ao espaço dos objetos, determinar a primeira interseção entre cada raio e um destes objetos, qual a cor correta do objeto naquele ponto, e utilizar esta cor para a exibição da imagem na tela, no ponto em que esta também é interceptada pelo raio. A idéia é ilustrada na figura 3.1.

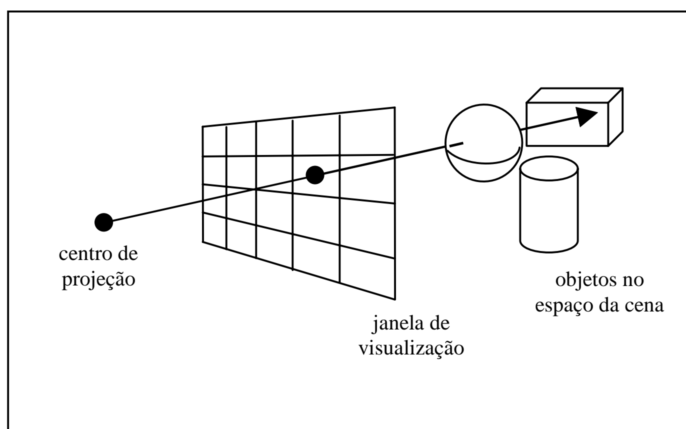


Figura 3.1 - Idéia básica do algoritmo de *raycasting*.

Na maior parte das vezes, utiliza-se um raio para cada pixel e cada raio passa pelo centro do pixel (um caso típico de um conceito denominado, em Computação Gráfica, amostragem pontual [Foley et al. 90]). Entretanto, há variações que utilizam mais de um raio por pixel (super-amostragem), para melhorar a qualidade da imagem, evitando efeito de

serrilhado nas bordas ou simulando características físicas de lentes. E outras, ainda, tentam simular raios que não sejam exatamente lineares, isto é, sem espessura, mas sim entidades geométricas com volume (cones, prismas, etc.) e que, como tal, interceptam uma certa área de cada pixel, podendo melhorar a amostragem (amostragem por área) e a qualidade da cena.

O pseudo-código para uma versão básica do algoritmo de *raycasting* é apresentado a seguir:

```
início
    determine o centro de projeção e a janela de visualização da cena
    para  cada linha da imagem  faça
        para  cada  pixel  na  linha  faça
            determine o raio que vai do centro de projeção ao pixel
            para  cada objeto na cena  faça
                determine a interseção mais próxima do raio com o objeto, se houver
                se  objeto é interceptado e  interseção é mais próxima que a mais próxima até agora  então
                    guarde a interseção como a mais próxima
                    guarde o nome do objeto como o primeiro interceptado
                fim se
            fim para
            determine a normal ao objeto no ponto de interseção mais próximo
            use a normal em um modelo de iluminação para determinar a cor do pixel
            pinte o pixel
        fim para
    fim para
fim
```

Neste ponto, é interessante ressaltar que este texto utiliza o termo *raycasting* para se referir ao lançamento de raios, ou traçado de raios não recursivo, em que cada raio é considerado apenas até a primeira interseção com um objeto de cena, e que se presta bem à resolução do problema de determinação da visibilidade (um problema fundamental de visualização). Nesta forma, o algoritmo faz uso de algum modelo de iluminação local e necessita de técnicas adicionais para a geração de fenômenos luminosos como sombras e refração. Por outro lado, há o traçado de raios recursivo – *ray tracing* ou *recursive ray tracing* – uma extensão do *raycasting* em uma técnica de visualização mais poderosa, introduzida em [Whitted 80], e que revolucionou o conceito de imagens realísticas em Computação Gráfica

no início da década de oitenta. Nesta técnica, após a determinação da primeira interseção de cada raio com um objeto, também é calculada localmente a equação de iluminação no ponto, mas além disso, são calculados dois outros raios, um refletido e um refratado. Estes raios são processados recursivamente, interceptando outros objetos da cena e retornando com uma contribuição para a cor final no ponto citado. Trata-se, portanto, de um algoritmo de iluminação global.

Embora qualquer implementação de *raycasting* possa ser adaptada sem muita dificuldade para fazer *ray tracing* recursivo, este não é um dos objetivos do presente trabalho.

3.2 O cálculo das interseções

A operação central em qualquer versão do algoritmo de *raycasting* é o cálculo das interseções, sendo responsável por grande parte da complexidade teórica na implementação e do custo computacional no processamento das cenas. Em algumas aplicações – como no caso da visualização de um único objeto ou superfície, sem a construção de estruturas de dados – é suficiente encontrar a primeira interseção (a mais próxima da origem do raio). Já em outras, como em sistemas CSG completos, é necessário calcular e, por vezes, armazenar todas as interseções de cada raio com cada objeto componente da cena.

Durante o cálculo, é comum expressar-se cada raio na forma paramétrica, em uma equação na forma:

$$r(t) = O + t\,D,$$

onde $O = (O_x, O_y, O_z)$ é o ponto de origem do raio no espaço da cena, $D = (D_x, D_y, D_z)$ é um vetor unitário que representa a sua direção, e t é o parâmetro que, ao variar de 0 (valor na origem) a infinito, determina todos os pontos sobre este raio.

A equação acima pode ser desmembrada em:

$$x(t) = O_x + t\,D_x$$

$$y(t) = O_y + t\,D_y$$

$$z(t) = O_z + t\,D_z$$

Para encontrar os pontos de interseção do raio com uma superfície implícita, representada por uma equação $g(x,y,z) = 0$, deve-se determinar os pontos do espaço que pertencem ao raio e, ao mesmo tempo, satisfazem à equação. De um modo geral, o que se faz é substituir os argumentos *x, y,* e *z*, na função *g*, pelas respectivas expressões em função de *t*, obtendo uma equação do tipo $g(\ x(t), y(t), z(t)\ ) = 0$ ou, resumidamente:

$$f(t) = g(\ r(t)\ ) = 0$$

Logo, o problema consiste em: para cada raio *r(t)* lançado na cena e para cada objeto ou superfície (que este raio possa vir a interceptar), encontrar as raízes da equação de uma variável $f(t) = 0$. Versões mais simples de *f(t)*, como equações lineares e quadráticas, possuem soluções bem definidas, mas a resolução de equações não-lineares genéricas é bem mais complexa, e tradicionalmente é resolvida com o auxílio de métodos numéricos. A abordagem mais utilizada, neste caso, é dividir o problema em duas etapas: **isolamento** de raízes e **refinamento**.

A etapa de isolamento consiste em identificar intervalos $[t_i\ ,\ t_{i+1}]$ que contenham, com certeza, ao menos uma raiz da equação $f(t) = 0$. Idealmente, uma única raiz. A etapa de refinamento da raiz consiste em tomar um ou mais destes intervalos e ir estreitando seus limites, convergindo para o valor exato da raiz, ou de uma delas. Geralmente é suficiente aproximar-se por um certa tolerância, que varia de acordo com o objetivo da aplicação, podendo ser arbitrária, determinada pelo usário, ou extraída a partir de alguma informação da cena. Os objetivos de cada etapa são ilustrados nas figuras a seguir:

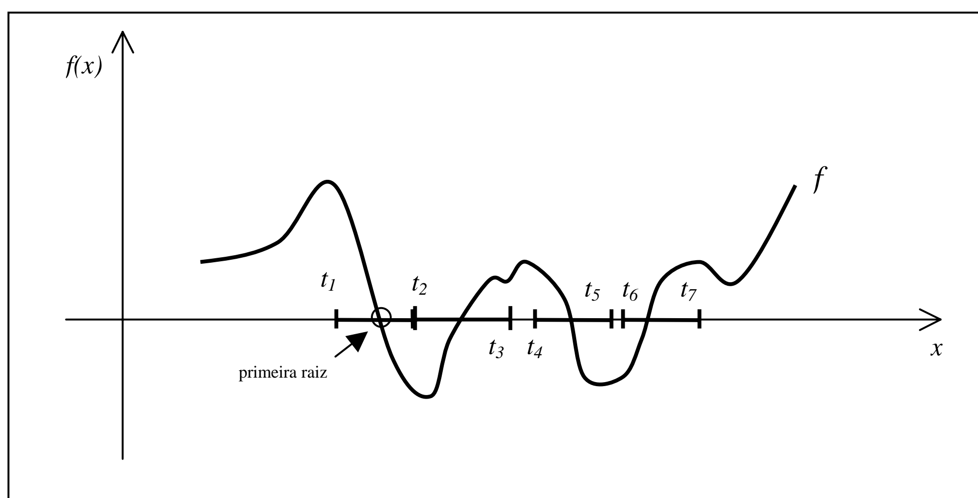


Figura 3.2 - Etapa de isolamento de intervalos com raízes.

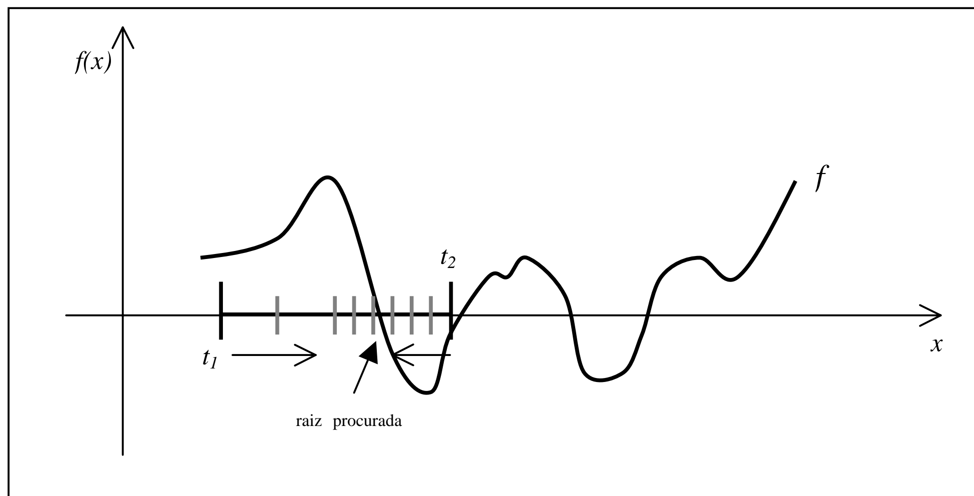


Figura 3.3 - Etapa de refinamento de raízes.

O problema de refinamento pode ser considerado como bem resolvido em cálculo numérico, de modo que há uma série de algoritmos disponíveis para sua solução, desde o clássico método da bisseção, o mais simples de todos, robusto, porém lento, até métodos bem mais eficientes – e consideravelmente mais complexos – como o de Brent [Brent 73].

A maior dificuldade, portanto, reside na etapa de isolamento das raízes. Os métodos numéricos convencionais – aqueles encontrados em textos introdutórios sobre o assunto – não podem ser aplicados de forma consistente a este problema, que historicamente tem sido tratado com sucesso com um conjunto de técnicas conhecido como **análise intervalar** (*range or interval analysis*) [Moore 79], que geralmente fornece soluções robustas, embora fazer contas com intervalos seja obviamente menos eficiente que computar com valores simples.

Um dos principais algoritmos neste sentido foi proposto por Moore [Moore 66], é razoávelmente simples e pode ser aplicado a um grande número de funções. Em poucas palavras, a idéia do algoritmo é realizar uma bisseção recursiva no domínio da função $f(t)$, calculando, por métodos intervalares, o resultado da função e de sua derivada, $f'(t)$, aplicadas a intervalos cada vez menores deste domínio. A cada passo, o algoritmo pode desprezar o intervalo, ao indentificar que este não pode conter raízes, ou subdividi-lo, caso contrário. O processo continua até isolar todos os intervalos que contenham uma única raiz – ou simplesmente aquele que contenha a primeira, dependendo da aplicação. Sobre estes intervalos é realizada a etapa de refinamento, por meio de métodos numéricos convencionais.

O algoritmo é apresentado em maiores detalhes no capítulo 4, onde discute-se também algumas variações para a versão básica proposta por Moore.

3.3 Trabalhos anteriores

Conforme vimos anteriormente, a parte mais difícil no cálculo de interseções raio-superfície é a etapa de isolamento das raízes de equações do tipo $g(\ x(t),\ y(t),\ z(t)\ )=0$. Quando a função $g(x,y,z)$, que define a superfície, é polinomial, a superfície é dita algébrica, e o problema de determinar os zeros da função resultante $f(t)$ é consideravelmente mais simples. Métodos consistentes − um deles baseado na regra de sinais de Descartes − já foram desenvolvidos para a determinação das raízes reais de polinômios. Estes métodos foram aplicados por Hanrahan no *raycasting* de objetos implícitos algébricos [Hanrahan 83].

Entretanto, quando a função $g(x,y,z)$ não é polinomial, a superfície é dita não-algébrica, e o problema de encontrar as raízes é mais complexo. Até o início da década de noventa, segundo [Mitchell 90], não havia muitos trabalhos neste sentido disponíveis na literatura. A seguir, é apresentado um breve resumo de alguns destes trabalhos, incluindo os mais recentes.

Em 1982, Blinn "renderizou" *Blobs*, superfícies definidas como somas de Gaussianas tridimensionais, utilizando alguns métodos heurísticos para determinar as raízes da função $f(t)$ [Blinn 82].

Em 1985, Toth foi um dos primeiros a utilizar métodos intervalares para fazer *raycasting*. No entanto, seu trabalho tratou da visualização de superfícies definidas parametricamente [Toth 85].

Em 1989, Kalra e Barr propuseram um algoritmo bastante robusto, baseado no conceito de superfícies LG (*LG-surfaces*), para fazer *raycasting* de uma classe razoavelmente grande de superfícies não-algébricas. Mais especificamente, aquelas para as quais vale a chamada condição de Lipschitz, baseada no conceito de constante de Lipschitz [Kalra & Barr 89].

Em 1990, Mitchell utilizou métodos intervalares − o algoritmo de Moore − e aritmética intervalar tradicional (IA) nas etapas de isolamento e refinamento das raízes, para fazer

raycasting de superfícies não-algébricas definidas de forma implícita, incluindo exemplos de *blends* [Mitchell 90].

Em 1993, Comba e Stolfi propuseram a aritmética afim, AA, como uma alternativa à IA, e sugeriram a sua utilização em algumas aplicações de algoritmos intervalares em Computação Gráfica, normalmente implementadas com IA, como *raycasting* e enumeração adaptativa de superfícies implícitas [Comba & Stolfi 93].

Em 1996, Figueiredo e Stolfi compararam a utilização de IA e de AA na enumeração adaptativa de superfícies implícitas por subdivisão espacial e com a construção de estruturas de dados espaciais chamadas *octrees*, e obtiveram resultados bastante favoráveis à utilização de AA, que mostrou uma maior precisão na determinação das células do espaço candidatas a conter a superfície, levando a um menor número de células e à construção de *octrees* menores [Figueiredo & Stolfi 96].

Por fim, [Hart 93] apresenta um interessante *survey* sobre diferentes formas de se fazer *raycasting* em superfícies e objetos implícitos, que enumera técnicas específicas para categorias particulares de funções ou primitivas, como *Blobs*, *Metaballs*, e outras.

Capítulo 4 Métodos numéricos para resolução de equações

O problema de determinar um ou mais **zeros** de uma função – valores do domínio que anulam esta função – encontra inúmeras aplicações em Computação Gráfica, assim como em outras áreas. Interessa-nos aqui, especificamente, seu uso na determinação de interseções entre raios e superfícies definidas implicitamente, mas podemos citar outras aplicações: cálculo de interseção entre superfícies, localização (temporal e espacial) de colisões entre objetos, deformações, etc.

Como vimos no capítulo anterior, a função resultante envolvida no cálculo das interseções no algoritmo de *raycasting*, $f(t)$, é uma função de uma variável, de modo que nosso problema não é dos mais difíceis, e reduz-se a determinar as raízes de uma equação do tipo $f(t) = 0$.

Infelizmente, para a maior parte destas equações não são conhecidos métodos algébricos de resolução (fórmulas "prontas"), como há para equações de segundo grau, por exemplo. Felizmente, este é um dos problemas clássicos (*root finding*) estudados no Cálculo Numérico, e há um bom número de algoritmos disponíveis, que mostram-se satisfatórios em muitas situações. A maioria deles – que chamaremos aqui de métodos numéricos convencionais – não utilizam nenhum tipo de cálculo intervalar (não fazem contas com intervalos), e não oferecem garantia de sucesso na etapa de isolamento de intervalos que contenham uma ou mais raízes, mas vários podem ser usados com segurança na etapa de refinamento. Já os métodos intervalares, cujo principal representante na literatura é o algoritmo de Moore – citado anteriormente e que será exposto em mais detalhes nas próximas seções – utilizam o conceito de aritmética com intervalos e prestam-se, de forma robusta e segura, tanto ao isolamento quanto ao refinamento das raízes. No entanto, métodos intervalares são quase que obrigatoriamente mais lentos, de forma que o ideal, para muitas aplicações, é a combinação

dos dois tipos de métodos, dando origem ao que chamamos aqui de métodos híbridos: intervalares na etapa de isolamento, convencionais na etapa de refinamento.

4.1 Métodos numéricos convencionais

Entre os métodos convencionais mais conhecidos, podemos citar os métodos da bisseção, da secante, da falsa posição (*regula-falsi*), método de Newton-Raphson [Santos 76], além de alguns menos conhecidos, dificilmente encontrados em textos introdutórios, como o método de Brent e o de Ridder, dois dos mais eficientes conhecidos. Interessam-nos particularmente três destes algoritmos, utilizados neste trabalho, como veremos posteriormente, na implementação dos algoritmos de visualização. São eles:

- Método da bisseção;
- Método da falsa posição, ou *regula-falsi*;
- Método de Brent;

A seguir é apresentada uma breve introdução aos métodos de bisseção e *regula-falsi*. Será comentado também o método de Brent, consideravelmente mais complexo. Um texto bastante completo sobre o assunto encontra-se em [Press et al. 92].

4.1.1 Método da bisseção

Este é o exemplo clássico entre os métodos convencionais para resolução de equações e, provavelmente, o mais simples [Press et al. 92]. Como a maioria dos métodos numéricos convencionais, o da bisseção exige que a raiz da função f esteja **confinada** (*bracketed* é o termo original, em inglês) ao intervalo inicial $[a,b]$, recebido. Ou seja, f deve ser definida em todo o intervalo $[a,b]$ e $f(a)$ e $f(b)$ devem ter sinais opostos. Em outras palavras, diz-se que o intervalo é **intervalo suporte** para a função f.

Se esta condição é satisfeita, então, baseando-se num caso particular do Teorema do Valor Intermediário [Munen & Foulis 82] − um teorema clássico do Cálculo − pode-se afirmar que: se f é contínua em $[a,b]$, então é garantido que o gráfico de f em $[a,b]$ corta ao

menos uma vez (na verdade, um número ímpar de vezes) o eixo das abcissas. Consequentemente, f possui ao menos uma raiz no intervalo.

Verificando-se a condição anterior, a idéia da bisseção é simples: o algoritmo recebe, a cada passo, um intervalo suporte [*a,b*], calcula *m*, o valor médio do intervalo, e *f(m)*, o valor da função *f* em *m*. Se *f(m)* tem sinal oposto a *f(a)*, utiliza *m* para substituir o outro extremo do intervalo, *b*, e o algoritmo prossegue, sendo aplicado ao novo intervalo [*a,m*]. Caso, contrário, o novo intervalo é [*m,b*]. O algoritmo, seja implementado de forma recursiva ou iterativa, pára quando a diferença $b - a$ for menor que alguma tolerância desejada, ε, ou se o valor *m* calculado for uma raiz ($f(m) = 0$). A cada iteração, o tamanho do intervalo que contém uma ou mais raízes é reduzido pela metade. As figuras a seguir ilustram o funcionamento do algoritmo de bisseção, mostrando dois passos consecutivos:

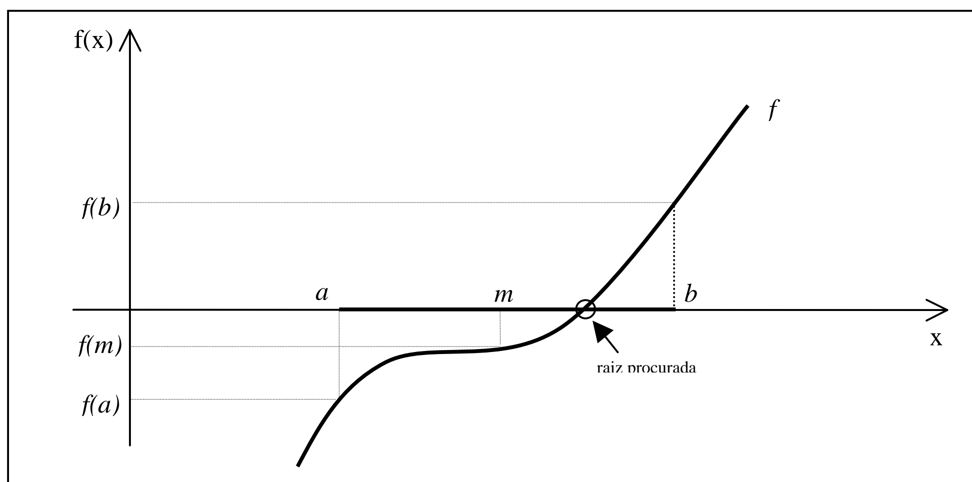


Figura 4.1 - Método da bisseção - passo *n*.

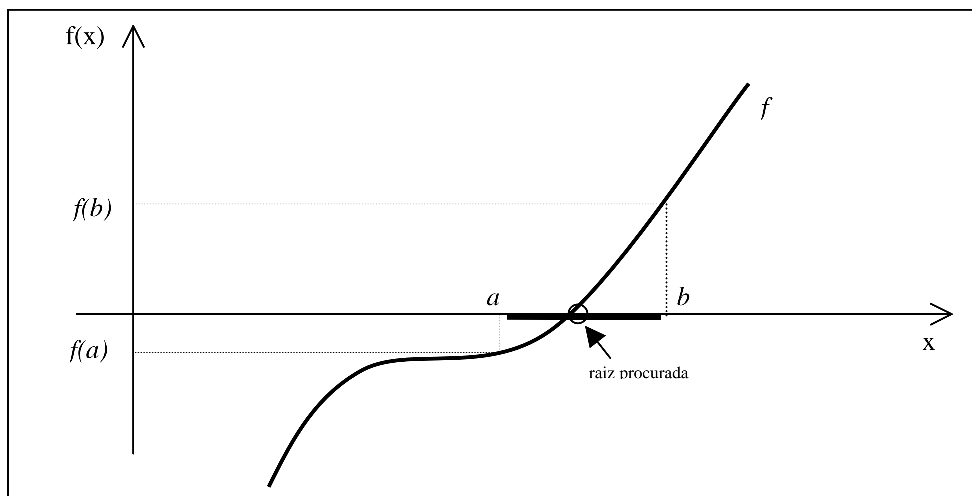


Figura 4.2 - Método da bisseção - passo *n+1*.

O pseudo-código para uma versão iterativa do algoritmo é apresentado a seguir:

```
início
t1 = a.      t2 =  b.
f1 = f(t1).
f2 = f(t2).
se  ( f1 < 0  e  f2 < 0 ) ou ( f1 > 0  e  f2 > 0 )  então
    retorne   erro!                                   // função não está confinada!
enquanto  (  t2 − t1  ≥  ε )  faça
    m  =  (t1 + t2) / 2.
    fm  = f( m ).
    se  ( fm = 0 )  então       retorne  m.           // raiz exata foi encontrada!
    se  ( f1* fm < 0 )  então
        t2 = m.        f2 = fm.
    senão
        t1 = m.        f1 = fm.
    fim se
fim enquanto
retorne   t1.
fim
```

A grande vantagem da bisseção, além de tratar-se de um algoritmo simples, é que apresenta convergência garantida: se houver uma única raiz de *f* no intervalo [*a,b*], o algoritmo sempre convergirá para ela. Se houver duas, ou mais raízes, o método encontrará uma delas, sendo difícil prever qual (mas não será necessariamente a primeira, o que seria particularmente útil para algoritmos de *raycasting*). Uma discussão a respeito é encontrada em [Corliss 77].

A desvantagem do algoritmo é ser considerado lento, quando comparado a outros disponíveis na literatura. O método da bisseção apresenta convergência linear: se após n passos (iterações ou recursões) a raiz procurada está contida em um intervalo de tamanho ε_n, após o próximo passo estará contida em um intervalo de tamanho $\varepsilon_{n+1} = \varepsilon_n / 2$. Em outras palavras, isto significa que o número de bits corretos da solução, numa representação de máquina, aumenta linearmente com *n,* o número de passos executados.

4.1.2 Método da falsa posição (*regula-falsi*)

O método da falsa posição, ou *regula-falsi*, como também é conhecido, é geralmente mais rápido que a bisseção, porém aplica-se bem apenas a funções com comportamento suave nas regiões próximas às raízes procuradas. Pode ser considerado uma variação de outro método semelhante, o da Secante. Ambos os métodos assumem que o gráfico da função seja aproximadamente linear no intervalo de interesse, de forma que, a cada passo, um segmento de reta que aproxima a função no intervalo é usado para obter a próxima estimativa para a raiz. Ambos exigem que a função esteja inicialmente confinada (*bracketed*) no intervalo inicial [*a,b*]. Falsa posição, porém, apesar de convergir um pouco mais lentamente, oferece a vantagem de garantir que, após cada passo, a função permanece confinada, enquanto no método da secante isto não ocorre sempre, de forma que a convergência não é garantida.

A cada passo, o algoritmo recebe um intervalo suporte [*a,b*], traça uma reta imaginária entre os pontos extremos do gráfico de *f*, (*a, f(a)*) e (*b, f(b)*), e calcula o valor *c*, abcissa do ponto onde esta reta corta o eixo das abcissas. Depois, *f(c)* é avaliada e seu sinal comparado com o de *f(a)* e de *f(b)*, de forma que *c* substitui o extremo de [*a,b*] cujo valor em *f* tenha o mesmo sinal que *f(c)*. Isto é, se *f(a)* e *f(c)* possuem o mesmo sinal, prossegue-se processando [*c,b*]. Caso contrário, o próximo passo processa [*a,c*].

O funcionamento do método da falsa posição é ilustrado na figura a seguir:

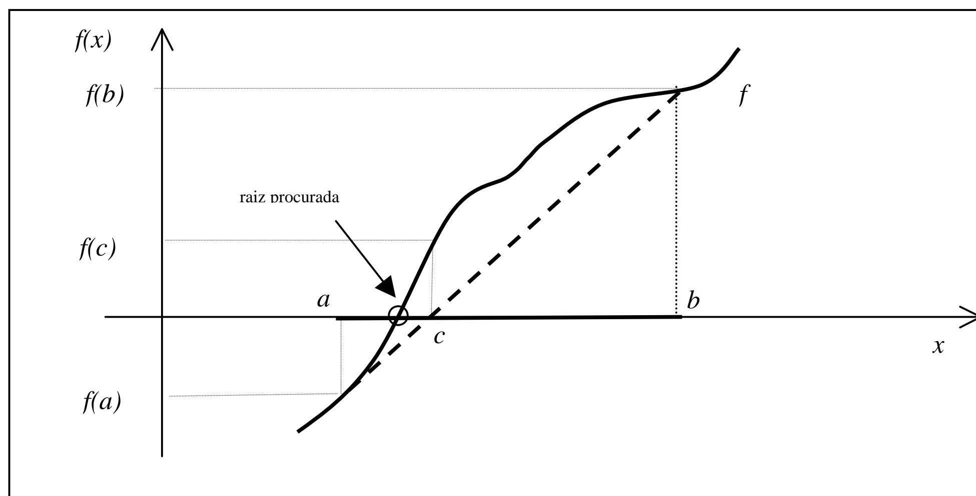


Figura 4.3 - Método da falsa posição (*regula falsi*) - passo *n*.

A determinação da exata ordem de convergência do algoritmo envolve cálculos bastante complexos, mas, segundo [Press et al. 92], este apresenta convergência super-linear no caso médio. Entretanto, falsa posição pode, para alguns casos patológicos de funções (e naquelas que não sejam suficientemente suaves na região de interesse), apresentar convergência mais lenta do que a bisseção. O mesmo ocorre para Secante e alguns outros algoritmos considerados rápidos. A figura a seguir ilustra um destes casos:

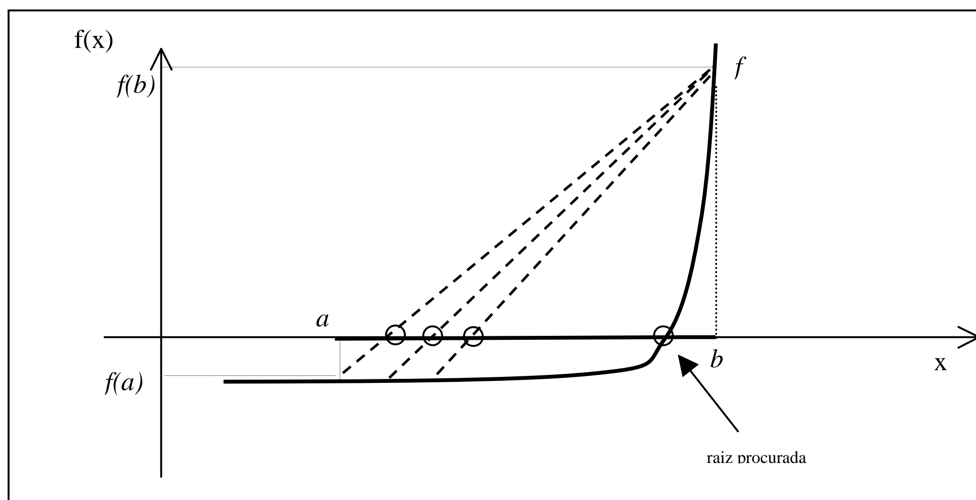


Figura 4.4 - Exemplo de caso ruim para *regula falsi* e outros métodos.

O pseudo-código para o algoritmo, numa versão iterativa, é apresentado a seguir:

```
início
fl = f( a ).    fh = f( b ).
se ( fl < 0  e  fh < 0 ) ou ( fl > 0  e  fh > 0 ) então
    retorne  erro!                                   // função não está confinada!
se ( fl < 0 ) então
    xl = a.    xh = b.
senão
    xl = b.    xh = a.
    troque( fl, fh ).
fim se
enquanto ( xh − xl ≥ ε ) faça
    dx = xh − xl.
    c = xl + dx * fl / (fl − fh).
    fc = f( c ).
    se ( fc = 0 ) então      retorne c.              // raiz exata foi encontrada!
    se ( fc < 0 ) então
        xl = c.    fl = fc.
    senão
        xh = c.    fh = fc.
    se ( fl < 0  e  fh < 0 ) ou ( fl > 0  e  fh > 0 ) então
        retorne  erro!                               // função não está mais confinada!
fim enquanto
retorne  xl.
fim
```

4.1.3 Método de Brent

Um algoritmo bem mais complexo e bastante eficiente, no caso médio, que combina a segurança (quanto à convergência) da bisseção com a velocidade de métodos super-lineares, foi desenvolvido na década de sessenta por Wijngaarden, Dekker e outros, no Mathematical Center, em Amsterdã, tendo sido melhorado posteriormente por Brent [Brent 73]. Por concisão, é referido normalmente na literatura como o método de Brent, ou de Brent-Dekker.

O algoritmo apresenta convergência garantida, desde que a função possa ser avaliada em todo o intervalo inicial. O presente trabalho faz uso de uma implementação do método de Brent na linguagem C, disponível em [Press et al 92], que alterna etapas dedicadas ao confinamento da raiz (*root bracketing*), etapas de bisseção e etapas que aplicam uma técnica conhecida como interpolação quadrática inversa, que apresenta convergência quadrática nos trechos em que a função é "bem comportada", apesar de instável nos demais trechos. Resumidamente, o que a implementação faz é monitorar se o método supostamente super-linear (no caso, quadrático) está realmente convergindo do modo esperado e, caso contrário, intercala o método da bisseção, voltando a garantir uma convergência que, no pior caso, é linear.

4.2 Métodos numéricos intervalares

Como descrito anteriormente, os métodos intervalares para resolução de equações (ou determinação de zeros de funções) são aqueles que prevêem, de algum modo, o cálculo exato ou estimado da(s) faixa(s) de valores assumidos pela função quando aplicada a intervalos de valores − e não apenas a valores individuais − de seu domínio. Tais métodos exigem a utilização de algum tipo de aritmética intervalar − um modelo de computação numérica que defina contas sobre intervalos − para sua realização prática. Aritméticas com intervalos são apresentadas no capítulo seguinte.

Historicamente, algoritmos intervalares têm sido aplicados com segurança e robustez à etapa de isolamento de raízes de equações, mas prestam-se, de forma igualmente robusta, à etapa de refinamento das soluções. No entanto, contas com intervalos são inevitavelmente mais caras do que contas com valores simples, de modo que, para diversas aplicações, algoritmos puramente intervalares podem mostrar-se lentos, de forma que muitos trabalhos têm dado preferência à combinação dos dois tipos de métodos, intervalares e convencionais, dando origem ao que chamamos aqui de **métodos híbridos**, ou seja: recorrem à robustez e à garantia de técnicas intervalares na etapa de isolamento e, quando possível, alternam para algum método convencional, mais rápido, para o refinamento da solução.

4.2.1 O algoritmo de Moore

O principal exemplo encontrado na literatura é um algoritmo genérico, e relativamente simples, para a determinação de zeros de funções de uma variável, proposto por Moore [Moore 66]. Dentro da classificação apresentada, é considerado um algoritmo híbrido, e tem sido aplicado com sucesso tanto a funções racionais como a qualquer expressão envolvendo funções transcendentais familiares.

O algoritmo começa com um intervalo inicial $[a,b]$ e baseia-se no cálculo de $F([a,b])$, a extensão intervalar de uma função f sobre um intervalo $[a,b]$ (melhor definida no próximo capítulo). No caso do *raycasting*, $[a,b]$ será um intervalo sobre o domínio de t − variável sobre a qual estão definidos, parametricamente, os raios lançados − e pode ser obtido, por exemplo,

de uma caixa de visualização (*bounding box*) definida para a cena. O algoritmo é recursivo e pára quando o tamanho do intervalo atinge um determinado valor de tolerância, ε, que pode ser ajustado de acordo com a aplicação ou determinado pela precisão aritmética da máquina. O algoritmo é descrito a seguir:

```
início
se ( b – a ) < ε então
    retorne.                                   // Solução já atingiu precisão desejada
calcule F( [a,b] ).                            // Calcula extensão intervalar de f
se 0 ∉ F( [a,b] ) então
    retorne.                                   // não há raízes no intervalo [a,b]
senão
    calcule F'( [a,b] ).
    se 0 ∉ F'( [a,b] ) então                   // f é monótona no intervalo [a,b]
        se f(a) f(b) > 0 então
            retorne.                           // não há raízes no intervalo [a,b]
        senão                                  // Há uma única raiz no intervalo
            Refine a solução com um método numérico convencional
        fim se
    senão                                      // F( [a,b] ) e F'( [a,b] ) contém 0
        calcule o ponto médio de [a,b], (a + b)/2.
        processe recursivamente [ a, (a+b)/2 ] e [ (a+b)/2, b ].
    fim se
fim se
fim
```

Este algoritmo pode ser adaptado de diferentes maneiras, de acordo com sua aplicação. É interessante notar que, na etapa de processamento recursivo, é realmente útil processar primeiro o intervalo mais próximo, [*a,(a+b)/2*]. Nos casos em que deseja-se encontrar apenas a interseção mais próxima (a menor raiz), o algoritmo pode parar ao encontrar a primeira raiz, isto é, no caso do intervalo [*a,(a+b)/2*] conter uma raiz. Não há necessidade de processar a segunda metade.

Já no caso da visualização de modelos CSG, é necessário determinar todas as interseções com o raio (encontrar todas as raízes). Neste caso, o algoritmo deve sempre executar todas as recursões. Por outro lado, se o objetivo for simplesmente determinar a

existência de sombras na cena, o algoritmo pode ser adaptado para simplesmente desprezar a etapa de refinamento. A simples prova da existência de uma ou mais raízes pode ser suficiente para o cálculo de regiões com sombra.

Por outro lado, pode-se observar, pelo algoritmo acima, que grande parte do custo computacional em cada recursão reside no cálculo de F([*a*,*b*]) e F'([*a*,*b*]), qualquer que seja a aritmética com intervalos utilizada. Sendo assim, é razoável perguntar se versões com uma menor carga de cálculos intervalares não poderiam rivalizar em eficiência com a versão anterior. Uma opção é abrir mão do refinamento por métodos convencionais, utilizando apenas F([*a*,*b*]) e deixando de calcular F'([*a*,*b*]). Na verdade, deixa de haver distinção clara entre as etapas de isolamento e refinamento e temos um algoritmo puramente intervalar, uma versão simplificada do algoritmo anterior:

início

se *(b – a) <* ε ***então***

    retorne.   *// Solução já atingiu precisão desejada*

calcule F([a,b]).   *// Calcula extensão intervalar de f*

se $0 \notin F([a,b])$ ***então***

    retorne.   *// não há raízes no intervalo [a,b]*

senão   *// F([a,b]) contém 0*

    calcule o ponto médio de [a,b], (a + b)/2.

    processe recursivamente [a, (a+b)/2] e [(a+b)/2, b].

fim se

fim

Estes algoritmos, como a maioria das técnicas intervalares, têm como premissa básica a avaliação de funções sobre intervalos de valores. Ou seja, a determinação de um intervalo final que contenha todos os valores assumidos por uma função *f(x)*, quando aplicada a todos os valores de um intervalo [*a*,*b*]. Para alguns poucos casos, pode ser fácil determinar este resultado, mas na maioria das vezes, não é uma tarefa trivial, mesmo quando realizada manualmente. Há necessidade de um modelo numérico computacional que defina um conjunto de operações – uma aritmética – sobre intervalos e como estas podem ser usadas para calcular, ou estimar, este intervalo final. No próximo capítulo são apresentadas dois dos principais modelos disponíveis na literatura para a computação numérica com intervalos.

Capítulo 5 Aritméticas com intervalos

5.1 Aritmética Intervalar Tradicional (IA)

Um dos primeiros modelos de computação numérica baseados no conceito de intervalo foi proposto por Moore [Moore 66]. Ele formalizou conceitos e definiu um conjunto de operações, a que chamou **aritmética intervalar** (IA), e mais tarde aplicou esta ferramenta a diversos problemas comuns na área de Matemática Aplicada [Moore 79].

Um intervalo $[a, b]$, onde a é chamado limite inferior e b limite superior, corresponde à faixa de valores reais entre a e b.

Um número real simples, a, pode sempre ser representado por um intervalo $[a, a]$, chamado de intervalo degenerado.

5.1.1 Operações básicas em IA

Abaixo, são apresentadas as operações aritméticas básicas com intervalos, como definidas originalmente:

$$[a, b] + [c, d] = [a + c, b + d]$$

$$[a, b] - [c, d] = [a - c, b - d]$$

$$[a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

$$[a, b] / [c, d] = [a, b] * [1/d, 1/c], \text{ se } 0 \notin [c, d]$$

É interessante notar, particularmente para fins de implementação, que em vários casos a multiplicação intervalar pode ser feita com menos de quatro multiplicações reais (o que é dispendioso). Por exemplo, se a, b, c são todos positivos, então simplesmente [a,b]*[c,d] = [ac, bd].

Da forma como foi proposta por Moore, não está definida a operação de divisão intervalar quando $0 \in [c,d]$. No entanto, para fins de implementação, pode ser interessante retornar alguma representação interna do intervalo $[-\infty, \infty]$, ao invés de simplesmente gerar uma exceção ou condição de erro. Mais tarde, Hansen desenvolveu uma definição mais genérica desta operação [Hansen 78], a fim de poder utilizá-la numa versão intervalar do Método de Newton, em casos em que a derivada era igual a 0.

$$
1/[a,b] = \begin{cases} [1/b,\ 1/a] & \text{, se } 0 \notin [c, d] \\ [1/b,\ \infty] & \text{, se } a = 0 \\ [-\infty,\ 1/a] & \text{, se } b = 0 \\ [-\infty,\ 1/a] \cup [1/b,\ \infty] & \text{, caso contrário.} \end{cases}
$$

Diversas outras operações ou funções matemáticas, além das operações aritméticas básicas, podem ter definidas suas equivalentes intervalares. A função valor absoluto (módulo), por exemplo, pode ser definida como:

$$
\text{abs}(\,[a,b]\,) = \begin{cases} [a,b] & \text{se } a \geq 0, \\ [-b, -a] & \text{se } b \leq 0, \\ [\,0\,,\ \max(\,|a|\,,\,|b|\,)\,] & \text{caso contrário.} \end{cases}
$$

De um modo geral, dada uma função racional qualquer, *r(x)*, define-se sua extensão intervalar natural, R(X), ou simplesmente **extensão intervalar**, como a avaliação da expressão *r* sobre um argumento $X=[x_0,x_1]$, usando-se as operações aritméticas descritas acima, ou alguma variação delas. Em outras palavras, calcula-se a expressão *r* por métodos intervalares, para um argumento intervalar. Qualquer que seja a definição das operações, e sua implementação, é fundamental que o resultado seja um intervalo que contém necessariamente

r(X), a chamada **faixa exata** (*exact range*) da função real *r(x)* sobre o intervalo X. Isto é, o conjunto de todos os valores que *r(x)* assume quando *x* varia entre x_0 e x_1. Formalmente, temos:

$$R(X) \supseteq r(X) = \{ r(x) \mid x \in X \}$$

Na prática, o intervalo R(X) pode vir a ser muito maior que *r*(X), a faixa exata da função *r* sobre o intervalo X. Após grandes cadeias de operações, este exagero pode levar a resultados inúteis, dependendo da aplicação. Infelizmente, tais cadeias não são raras em cálculos científicos, como aqueles presentes em diversos algoritmos de Computação Gráfica.

Além disso, o tamanho do resultado intervalar calculado depende da expressão de *r*. A forma como a função *r* é expressa influi diretamente no tipo, na quantidade das operações intervalares e na ordem em que estas são avaliadas, de modo que expressões equivalentes na aritmética ordinária (geram valores idênticos) podem gerar resultados diferentes quando estendidas para a aritmética intervalar. Num caso clássico, temos que X(Y+Z) não é igual a XY + XZ, ou seja, não vale a distributividade da multiplicação em relação à adição. Na verdade, $X(Y+Z) \subseteq XY + XZ$.

Note-se que este aspecto deve ser fortemente considerado na implementação de algoritmos intervalares. Em geral, deve-se dar preferência (quando possível) a versões condensadas de equações, em detrimento de versões expandidas. A opção pelas primeiras tende a gerar intervalos menores e permitir um menor número de passos em algoritmos intervalares. Este aspecto é exemplificado no capítulo de testes.

Uma das mais importantes propriedades da extensão intervalar de funções é a chamada **inclusão por monotonicidade:** para dois intervalos *X* e *X'*, se $X' \subset X$, então $r(X') \subseteq r(X)$. A propriedade é ilustrada na figura a seguir:

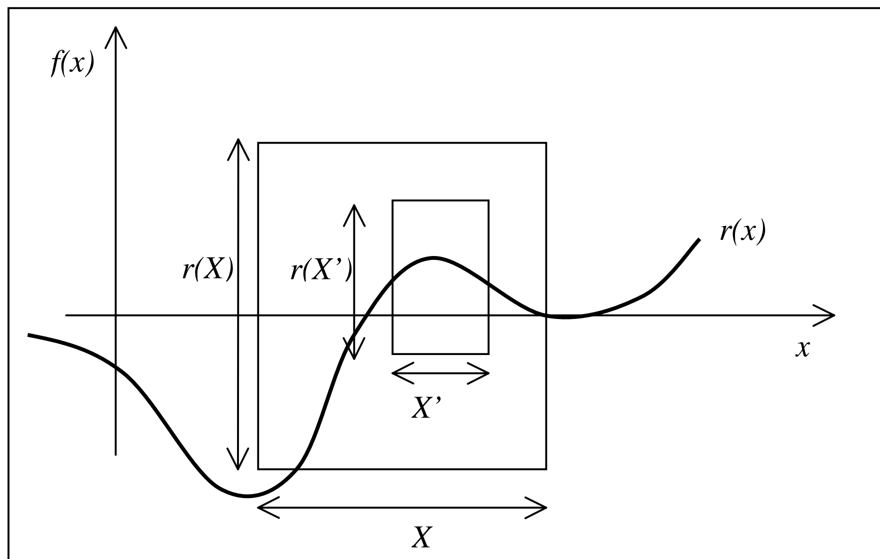


Figura 5.1 - Propriedade da inclusão por monotonicidade em IA.

Entre outras coisas, isto significa que, à medida que o intervalo *X* torne-se mais estreito, o intervalo r(*X*) também irá estreitar-se, obrigatoriamente.

O conceito de extensão intervalar natural foi definido originalmente para funções racionais, mas pode ser estendido também para funções transcendentais.

No caso de *r(x)* ser uma função monótona qualquer, é fácil determinar sua extensão intervalar e, além disso, esta sempre coincide com a faixa exata *r(X)*. Dois exemplos:

$$[a,b]^3 = [a^3,b^3] \qquad (r(x)=x^3 \text{ aplicada ao intervalo } [a,b])$$

$$e^{[a,b]} = [e^a, e^b].$$

É interessante notar que, enquanto $[a,b]^3$, quando implementada, pode fornecer diretamente o intervalo exato $[a^3,b^3]$, calcular $[a,b]\ [a,b]\ [a,b]$ geralmente fornece intervalos bem maiores (que contêm o intervalo exato).

Além disso, sabe-se que qualquer função *f(x)* contínua em um intervalo $[a,b]$, seja monótona ou não, pode ser decomposta em seções monótonas. Na verdade, o intervalo $[a,b]$ pode ser decomposto em intervalos nos quais *f* é monótona, separados por pontos críticos, ou seja, mínimos e máximos locais que ocorrem nos valores de *x* para os quais $f'(x) = 0$. Se, dada

uma função f, contínua em um intervalo $[a,b]$, é possível determinar tais pontos críticos, c_1, c_2, ... , então pode-se calcular o resultado intervalar exato de f no intervalo, que é dado por:

$$F([a,b]) = [\ \min(f(a), f(b), f(c_1), f(c_2), \ldots)\ ,\ \max(f(a), f(b), f(c_1), f(c_2), \ldots)\ ].$$

Definições e um tratamento completo sobre operações intervalares e suas propriedades são vistos em [Moore 66] e em [Moore 79], e uma discussão detalhada sobre a implementação da aritmética, incluindo algoritmos, pode ser encontrada em [Figueiredo & Stolfi 97].

5.1.2 O problema da explosão de erro nos cálculos com IA

Se, por um lado, teoricamente IA fornece uma ferramenta simples, robusta e elegante para a implementação de análise intervalar, com algumas importantes aplicações em Computação Gráfica [Mitchell 91], [Duff 92], [Snyder 92], por outro lado, IA apresenta, na prática, uma considerável fraqueza, que acaba inviabilizando algumas destas aplicações: em geral, o intervalo computado para uma expressão pode vir a ser muito maior que a faixa exata de valores que aquela expressão pode assumir. Talvez tão maior ao ponto de tornar-se inútil, dependendo da sequência de cálculos envolvida e das exigências da aplicação em questão.

Em grande parte, esta imprecisão deve-se ao fato de as operações de IA serem bastante conservadoras, assumindo que os valores desconhecidos envolvidos em uma expressão são sempre independentes entre si, podendo, cada um, variar sobre toda a extensão do intervalo que o representa. Entretanto, geralmente isto não é verdade: podemos ter por exemplo, numa mesma expressão, termos como x e $(100 - x)$, ou y e yz. Sabemos que, quando há qualquer restrição matemática entre dois ou mais argumentos de uma expressão, nem todas a combinações de valores são válidas para estes argumentos, mesmo dentro dos intervalos correspondentes. Por exemplo, se temos x e $(100 - x)$ em uma expressão, com x desconhecido, porém dentro do intervalo [0,100], sabemos que, se x valer 30, então $(100 - x)$ só pode valer 70. Contudo, IA pela natureza da representação usada para os intervalos, e pela forma como estão definidas suas operações [Moore 66], não leva em consideração este tipo de informação: x e $(100 - x)$ são encarados como termos independentes, praticamente como variáveis diferentes. Mesmo diferentes ocorrências de x na expressão são tratadas como

valores/intervalos independentes. O efeito disso, é que os intervalos computados tendem a ser bem maiores que o necessário.

Consideremos, por exemplo, o cálculo intervalar da expressão $x(10 - x)$ (como apresentado em [Figueiredo & Stolfi 96]), onde é sabido que x encontra-se no intervalo $\underline{x} = [4, 6]$. Pelas definições das operações de adição e multiplicação, temos:

$$x \rightarrow \underline{x} = [4, 6] ; \qquad 10 \rightarrow [10, 10] ;$$

$$(10 - \underline{x}) = [10, 10] - [4, 6] = [10 - 4, 10 - 6] = [6, 4] = [4, 6],$$

$$\underline{x}(10 - \underline{x}) = [4, 6] * [4, 6] = [min(16, 24, 24, 36), max(16, 24, 24, 36)] = [16, 36].$$

Com um pouco de Cálculo, pode-se determinar a exata faixa da expressão $x(10 - x)$ sobre o intervalo [4 , 6]: o valor da expressão nos dois extremos é o mesmo, 24; como trata-se de uma expressão de 2º grau, ($10x - x^2$), temos um máximo no ponto onde a derivada, $10-2x$, é 0. Este ponto tem abcissa 5 e o valor da expressão é 25. Logo, a faixa exata no intervalo é [24 , 25]. Repare que o intervalo encontrado pelos cálculos com IA, [16 , 36], está correto, sem dúvida (contém o intervalo exato), mas é 20 vezes maior que o necessário.

Se calcularmos com IA outra expressão, $10x - x^2$, que é equivalente a $x(10 - x)$, para o mesmo intervalo em x, [4 , 6], obteremos o intervalo [4 , 44], que constitui uma estimativa pior ainda que a anterior, e confirma o fato de que expressões equivalentes, descritas de formas diferentes, podem levar a resultados bem diferentes. Por outro lado, se calcularmos a expressão $25 - (x - 5)^2$, também equivalente a $x(10 - x)$, obteremos [24 , 25]. Repare que aqui IA fornece o resultado exato − o menor intervalo possível. Isto ocorre porque há apenas uma ocorrência de x na expressão (não há a possibilidade de termos correlacionados serem tratados como variáveis independentes).

Apesar de terem sido apresentadas algumas definições alternativas de IA, como em [Hansen 78], esta tem sido, ao longo das últimas décadas e até alguns anos atrás, a ferramenta padrão para a implementação de algoritmos intervalares, como o de Moore e semelhantes. Atualmente, a aritmética afim, proposta no início da década de noventa, constitui provavelmente a principal alternativa à IA tradicional. A técnica é apresentada na próxima seção.

5.2 Aritmética Afim (AA)

Em 1993, Comba e Stolfi [Comba & Stolfi 93] propuseram um novo modelo de computação numérica para a análise intervalar (*range analysis*), o qual denominaram **aritmética afim**, ou AA, como alternativa à aritmética intervalar tradicional. Trata-se de uma extensão de IA proposta basicamente com o objetivo de tratar o problema da explosão de erro, que ocorre principalmente em longas cadeias de cálculos com intervalos. Da mesma forma que IA, AA trata automaticamente dos possíveis erros de arredondamento e truncamento. A diferença é que, além disso, AA armazena mais informação na representação de cada quantidade parcialmente desconhecida (intervalo). O propósito desta informação extra é não deixar que se perca de vista as correlações entre duas ou mais quantidades que não são totalmente independentes.

Na aritmética afim, um valor (ou quantidade) parcialmente desconhecido, *x*, é representado não apenas como um intervalo, mas como uma forma afim $\underline{x}$, que nada mais é do que um polinômio de 1º grau:

$$\underline{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \ldots + x_n\varepsilon_n ,$$

onde os x_i são coeficientes reais conhecidos e os ε_i são variáveis simbólicas, chamadas *ruídos*, cujos valores exatos são parcialmente desconhecidos, podendo assumir qualquer valor no intervalo U = [−1 , +1]. Cada um dos ruídos está associado a uma possível fonte de erro, ou incerteza, no valor exato de *x*, gerada em algum momento numa seqüência de cálculos. Como veremos adiante, algumas operações com formas afins (adição, subtração, multiplicação por um valor escalar, ...) não geram tais fontes de erro, enquanto outras operações geram (multiplicação, cálculo da raiz quadrada, log, etc.).

Justamente pelo fato de AA armazenar mais informação para representar cada intervalo, duas consequências surgem: as operações em AA tendem a ser mais caras do que em IA, mas tendem a gerar intervalos mais "justos", mais precisos, e esta maior precisão é capaz de compensar o custo dos cálculos em diversas aplicações, como em [Figueiredo & Solfi 97].

As principais operações aritméticas com valores afins, assim como sua implementação, serão discutidas na próxima seção. Antes disso, examinaremos a correlação entre os conceitos de intervalo e de valor afim, bem como a correlação entre as duas aritméticas :

Primeiramente, se uma quantidade x está armazenada, representada como uma forma afim $\underline{x} = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n$, então o exato valor de x está garantidamente no intervalo $[\underline{x}]$, denominado faixa (*range*) de x, definido por:

$$[\underline{x}] = [\, x_0 - \xi \,..\, x_0 + \xi \,], \text{ onde } \xi = \Sigma_{i=1,n} |x_i|$$

Neste caso, $[\underline{x}]$ é realmente o menor intervalo que, garantidamente, contêm o valor x, assumindo-se que cada ε_i pode variar livremente no intervalo $[-1 , +1]$. Desta forma, pode ser feita a conversão de qualquer valor afim em um valor intervalar.

Também é simples a conversão de valores intervalares em valores afins. Dado um intervalo x = [a , b], representando uma quantidade parcialmente conhecida, x, em IA, a forma afim equivalente é dada por:

$$\underline{x} = x_0 + x_1\varepsilon_1 , \quad \text{onde} \quad x_0 = (\, b + a \,)/2 \quad \text{e}$$

$$x_1 = (\, b - a \,)/2$$

ou seja, x_0 é o ponto (valor) médio do intervalo x, e x_1 é a distância deste valor até as extremidades.

A principal característica deste modelo numérico é o fato de o mesmo símbolo de ruído ε_i poder contribuir para a "incerteza" de duas ou mais quantidades envolvidas em uma expressão (e que podem tratar-se de valores de entrada, de saída ou resultados intermediários). O compartilhamento de um mesmo símbolo ε_i por duas formas afins $\underline{x}$ e $\underline{y}$ indica uma dependência parcial entre as quantidades subjacentes, x e y, que estão sendo representadas por $\underline{x}$ e $\underline{y}$. A magnitude e o sinal desta interdependência serão determinadas pelos coeficientes correspondentes, x_i e y_i (é interessante citar que os sinais de x_i e y_i não são necessariamente significativos, mas sim os sinais relativos entre eles [Comba & Stolfi 93]).

Vamos supor, por exemplo, duas quantidades x e y, representadas pelas formas afins

$$\underline{x} = 10 + 2\varepsilon_1 + 1\varepsilon_2 + \quad - 1\varepsilon_4$$

$$\underline{y} = 20 - 3\varepsilon_1 \quad + 1\varepsilon_3 + 4\varepsilon_4$$

Pelas conversões mostradas anteriormente, podemos afirmar que *x* está contido no intervalo [6 , 14], e *y* no intervalo [12 , 28]. Entretanto, como os dois compartilham os mesmos símbolos ε_1 e ε_4, com coeficientes não-nulos, sabemos que não são totalmente independentes, um do outro. Na verdade, se traçar-mos o gráfico no plano cartesiano e dermos uma interpretação geométrica para o problema [Figueiredo & Stolfi 97], veremos que o conjunto de possíveis valores para o par (*x,y*) está restrito à região do $\Re^2$ equivalente ao polígono (cinza escuro) mostrado na figura 5.2, que é substancialmente menor que o retângulo [6 , 14] × [12 , 28], e que é a única informação geométrica que os intervalos simples nos fornecem. Obviamente, esta informação de interdependência seria perdida se as formas afins $\underline{x}$ e $\underline{y}$ fossem simplesmente substituídas pelos intervalos equivalentes, [$\underline{x}$] e [$\underline{y}$], apesar de estes últimos, individualmente, representarem exatamente as mesmas faixas de valores que os primeiros.

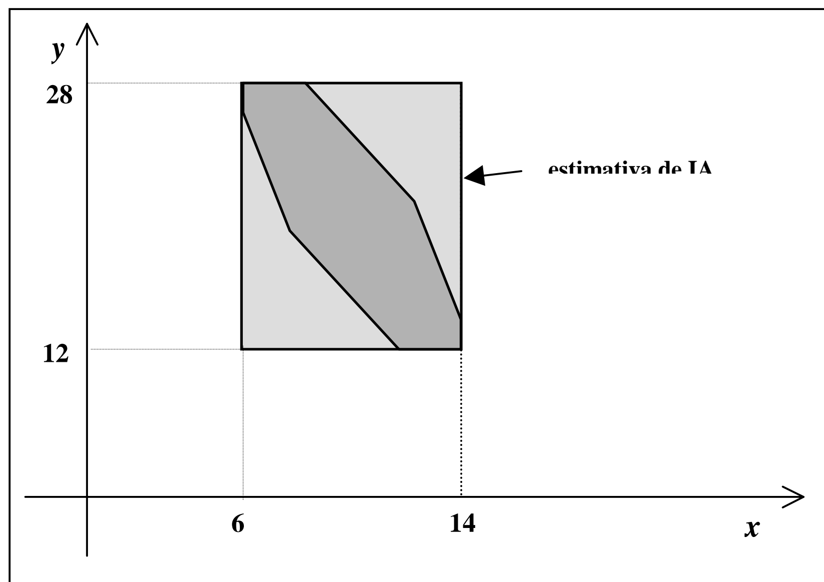


Figura 5.2 – Região no R^2 para os possíveis valores de (x,y) com IA e AA.

5.2.1 Operações básicas em AA

Podemos compreender melhor as operações com intervalos distinguindo dois tipos básicos de operações: afins e não-afins.

Por definição, podemos enumerar as operações afins básicas:

$$\underline{x} + \underline{y} = (x_0 + y_0) + (x_1 + y_1)\,\varepsilon_1 + \ldots + (x_n + y_n)\,\varepsilon_n$$

$$\underline{x} - \underline{y} = (x_0 - y_0) + (x_1 - y_1)\,\varepsilon_1 + \ldots + (x_n - y_n)\,\varepsilon_n$$

$$\alpha\,\underline{x} = (\alpha\, x_0) + (\alpha\, x_1)\,\varepsilon_1 + \ldots + (\alpha\, x_n)\,\varepsilon_n$$

Estas operações não acrescentam novos termos ao polinômio resultante. Já as operações não-afins, como, por exemplo, o produto de duas formas afins, o cálculo de seno, coseno, logaritmos, etc., acrescentam novos termos, aumentando o tamanho dos polinômios.

Diversas outras operações em AA são definidas em [Figueiredo & Stolfi 97], e muitas delas têm seus algoritmos apresentados, em pseudo-código.

Revendo os exemplos de expressões computadas com IA na seção anterior, temos que AA fornece, para $x(10 - x)$, o intervalo [24 , 26], que não é o resultado exato mas é 10 vezes menor que o fornecido por IA. Obviamente, as contas com AA para chegar a este resultado são mais caras, devido à sua própria definição. Tanto para $10x - x^2$ como para $25 - (5 - x)^2$, o intervalo fornecido é o mesmo: [24 , 25], o resultado exato.

Sob uma interpretação geométrica (confirmada com alguns cálculos), pode-se perceber que AA aproxima a raiz desconhecida de uma função *f*, em um intervalo $[t_{min}, t_{max}]$, através de um paralelogramo como o mostrado na figura abaixo, enquanto IA aproxima através do retângulo $\{(t_{min}, f(t_{min})), t_{max}, f(t_{max})\}$, certamente uma aproximação pior.

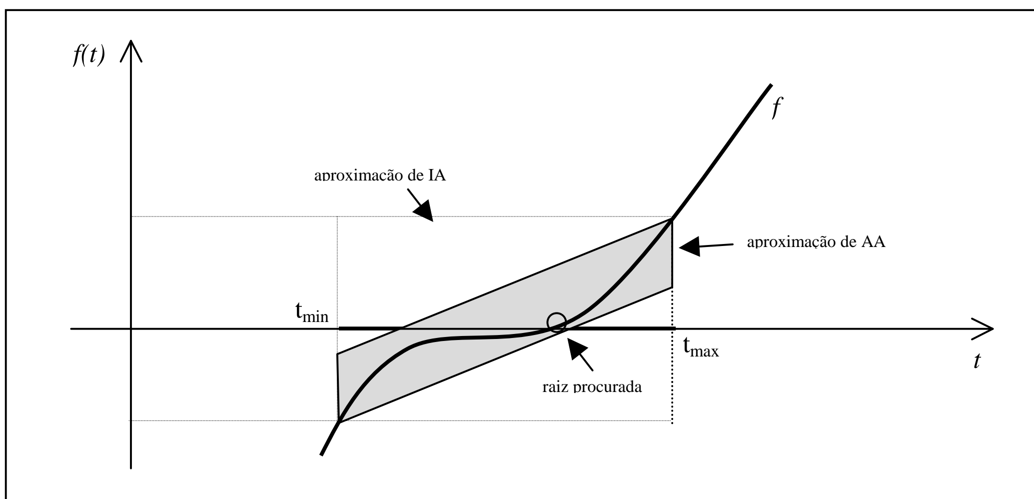


Figura 5.3 - Interpretação geométrica das aproximações com IA e com AA.

5.2.2 Otimização em algoritmos intervalares com AA

Além do fato de geralmente produzir resultados (intervalos) mais precisos, possibilitando, em alguns algoritmos, um menor número de cálculos ou recursões, AA oferece uma vantagem adicional, mais uma vez pelo fato de armazenar mais informação na representação de cada intervalo: permite que seja realizada, quase sem custo, uma etapa de otimização em algoritmos como o de bisseção intervalar, que computam intervalos cada vez menores a cada passo, buscando convergir para um intervalo limite.

A otimização é tipicamente realizada em cada passo (recursão ou iteração) do algoritmo, quase a custo zero, e no entanto pode vir a reduzir bastante o intervalo computado normalmente, a cada passo – podendo levar, inclusive, a descartar intervalos que ainda não seriam descartados – levando a uma convergência mais rápida do algoritmo.

A idéia da operação de otimização, ilustrada na figura 5.3, aplica-se perfeitamente ao algoritmo de bisseção intervalar e é implementada com alguns poucos cálculos. Em cada recursão, inicia-se com um intervalo $t = [t_{min}, t_{max}]$, é computada a extensão intervalar F(t), e determina-se f_{min} e f_{max}, extremos de F(t). A otimização consiste em computar um novo intervalo $[i_{min}, i_{max}]$, que será a interseção do paralelogramo de aproximação de AA (ou das projeções de seus lados sobre o eixo *t*, no caso de tal paralelogramo não interceptar este eixo, como no caso da figura) com o intervalo original $[t_{min}, t_{max}]$. A razão para isto é simples: se AA garante que o gráfico da função está contido no paralelogramo, então a raiz procurada só pode estar nesta interseção, e não em todo o intervalo original. A idéia pode ser melhor compreendida observando-se a figura:

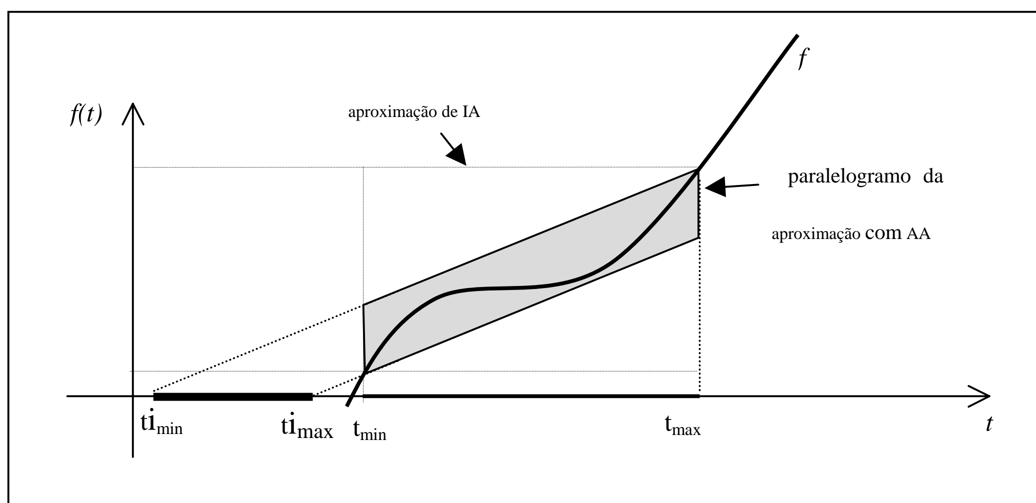


Figura 5.4 - Otimização em algoritmos intervalares com AA.

Uma forma simples de computar $[i_{min}, i_{max}]$ é fazê-lo em duas etapas: primeiramente, calcula-se o intervalo $[ti_{min}, ti_{max}]$, interseção do paralelogramo com o eixo *t*, das abcissas (onde *f(t)=0*). Depois, calcula-se a interseção de $[ti_{min},ti_{max}]$ com o intervalo original $[t_{min},t_{max}]$, resultando em $[i_{min}, i_{max}]$. Este novo intervalo muitas vezes é bem menor que o original, podendo até ser vazio. No primeiro caso, o algoritmo prossegue (quebrando o intervalo em dois e investigando cada um recursivamente) sobre um intervalo menor, possibilitando uma convergência mais rápida. No segundo caso, o intervalo original está descartado (não pode haver raízes de f em $[t_{min},t_{max}]$), e não são feitas novas recursões. Descrevemos, a seguir, os cálculos envolvidos na otimização.

Em cada passo do algoritmo, após o cálculo intervalar de *f(t)*, temos $t \in [t_{min}, t_{max}]$ representado em AA, na forma:

$$\underline{t} = t_0 + t_1\varepsilon_1 , \qquad \text{onde} \quad t_0 = (t_{min} + t_{max})/2 \quad \text{e} \quad t_1 = (t_{min} - t_{max})/2$$

e F, que, tendo sido obtida após uma cadeia de cálculos em AA, está representada na forma:

$$f = f_0 + f_1\,\varepsilon_1 + f_2\,\varepsilon_2 + \ldots + f_n\,\varepsilon_n .$$

É interessante notar que a variável ε_1 é comum às duas formas afins, t e f, e trata-se realmente da mesma variável de ruído, mostrando que as duas grandezas não são totalmente independentes (se fossem, a única aproximação possível para o gráfico de f(t) dentro do intervalo $[t_{min}, t_{max}]$ seria simplesmente o retângulo que IA fornece, e não o paralelogramo de AA). Logo, podemos aproveitar este termo em comum para relacionar as duas formas afins, t e F (que, por ora, chamaremos de f). Podemos, primeiramente, colocar f na forma

$$f = f_0 + f_1\,\varepsilon_1 + h\,\varepsilon ,$$

onde $h = |f_2| + \ldots + |f_n|$, e pode ser facilmente obtido fazendo-se: *norma*(f) $- |f_1|$ (a **norma** de um valor afim, definida como $|f_1| + |f_2| + \ldots + |f_n|$, é uma operação útil, envolvida em diversas outras operações, e é comentada no capítulo sobre implementação).

Como queremos calcular a interseção de f com o eixo das abcissas (f(t) = 0), devemos tomar f com valor 0. Se $\underline{t} = t_0 + t_1\varepsilon_1$, então $\varepsilon_1 = (t - t_0) / t_1$. Substituindo ε_1 na equação de f, temos:

$$0 = f_0 + f_1\,(\,(t - t_0) / t_1\,) + h\,\varepsilon$$

Desenvolvendo a equação, temos:

$$0 = t_1 f_0 + f_1 t - f_1 t_0 + t_1 h \varepsilon$$

$$. \quad . \quad .$$

$$t = (t_0 - (t_1/f_1) f_0) + (t_1/f_1) \; h \; \varepsilon \quad ,$$

onde t é, na verdade, uma faixa de valores (ε pode variar entre –1 e +1), que representa o intervalo intermediário [ti_{min}, ti_{max}], de interseção do paralelepípedo com o eixo das abcissas. Eliminando-se a variável de ruído ε e fazendo $q = (t_1/f_1)$ temos:

$$ti_{min} = t_0 - q f_0 - | q \; h |$$

e

$$ti_{max} = t_0 - q f_0 + | q \; h |$$

Depois, calcula-se [ti_{min}, ti_{max}] ∩ [t_{min}, t_{max}], obtendo o intervalo [i_{min}, i_{max}], que será, no máximo, igual ao menor dos dois intervalos anteriores, podendo ser menor que os dois, ou mesmo nulo, dependendo da ocorrência de uma das situações mostradas na figura seguinte.

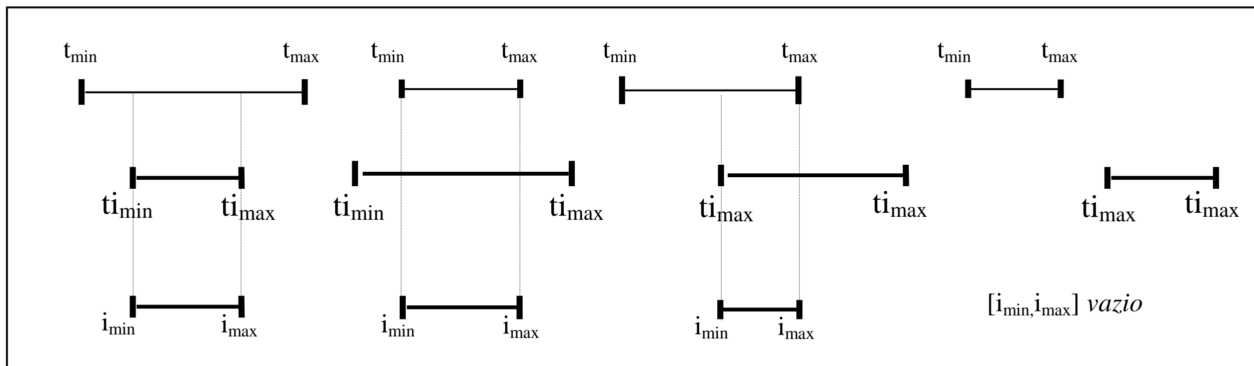


Figura 5.5 - Casos de interseção possíveis na otimização com AA.

A seguir é apresentado um trecho de código, em linguagem C, que implementa esta otimização. Na aplicação desenvolvida, este trecho é executado apenas quando a opção **otimização AA** está selecionada pelo usuário no momento da visualização:

```
        double   q, h, pri, sec, timin, timax ;
        // f e t sao formas afins, do tipo AAform
        // (documentado no capítulo Implementação)
        // tmin e tmax sao double

  q = - t[1] / f[1] ;
  h = aa_norm( f ) - fabs( f[1] ) ;
  pri =  t[0] + q * f[0] ;
  sec = fabs( q * h );
  timin = pri - sec ;
  timax = pri + sec ;
  // Calcula a intersecao dos intervalos ti e t, joga em t
  if( timin > tmin )  tmin = timin ;
  if( timax < tmax )  tmax = timax ;

  if( tmin >= tmax )  {          // Se a intersecao e vazia
     aa_close();
     return( 0 );                // Não ha zeros no intervalo
  }
  aa_range( f , &fmin, &fmax );

  . . .
```

Alguns outros trechos de código, relacionados a aspectos não triviais da implementação, principalmente à utilização das operações intervalares no cálculo das extensões intervalares de funções e derivadas, são descritos no próximo capítulo, que detalha algumas questões relativas à implementação.

Capítulo 6 Implementação

A parte de implementação envolvida no presente trabalho incluiu, em sua última etapa, o desenvolvimento de um programa para a visualização de superfícies implícitas, o qual denominamos *SuperRay*. O programa possui interface gráfica com o usuário, e foi desenvolvido para execução em micro-computadores da linha PC, sobre o sistema operacional Windows 95. Foi desenvolvido em linguagem C (padrão ANSI), utilizando técnicas comuns de programação estruturada e orientada a eventos, com o ambiente integrado de desenvolvimento Borland C++ Builder, versão 1.0, da Borland International. A interface gráfica foi construída baseada na VCL (Visual Component Library), biblioteca de componentes gráficos da Borland. Foram também utilizados a linguagem de programação Lua [Ierusalimschy et al. 96], desenvolvida na Pontifícia Universidade Católica do Rio de Janeiro, e o software MapleV, versão 4.0, para a automatização de algumas tarefas, como a geração automática de derivadas parciais e a visualização de exemplos.

6.1 As bibliotecas de operações IA e AA

A implementação desenvolvida faz uso de duas bibliotecas (*ia.c* e *aa.c*), com rotinas escritas em linguagem C, que implementam um bom número de operações definidas em IA e AA, baseando-se nos algoritmos encontrados em [Figueiredo & Stolfi 97]. A grande maioria das operações possui uma equivalente em IA e outra em AA, implementadas na forma de funções C com o mesmo nome e protótipo, definido em um único arquivo de cabeçalho, *aa.h*, comum às duas bibliotecas. Desta forma um programa que faça uso das funções pode, na maior parte do tempo, utilizar o mesmo código, de forma transparente (exceto para algumas poucas operações específicas). Há necessidade apenas de "re-ligar" o código a uma ou outra biblioteca, para gerar versões IA ou AA do mesmo programa. Isto facilita uma comparação direta dos desempenhos relativos entre as versões.

As duas bibliotecas trabalham com o mesmo tipo básico, `AAform`, para a representação e manipulação dos intervalos. A seguir, é apresentada a definição deste tipo, assim como alguns detalhes de implementação e os protótipos das rotinas para as principais operações:

```
#define    AA_N    32               // Limite ajustável com a necessidade
#define    real    double
typedef    real    AAform[AA_N + 1];     //Representação dos intervalos

void  aa_open(void);                //Devem ser usados para abrir e fechar
void  aa_close(void);               //uma cadeia de cálculos com intervalos

void  aa_interval(AAform a, real amin, real amax);    //Cria um intervalo
//Retorna, em amin e amax, os extremos do intervalo a:
void  aa_range(AAform a, real* amin, real* amax);

void  aa_set(AAform a, AAform b);                     //Atribuição:  a = b
void  aa_clear(AAform a);                             //"Limpa" a

//Operações afins:
void  aa_add(AAform a, AAform b, AAform c);     //Adição:      a = b + c
void  aa_sub(AAform a, AAform b, AAform c);     //Subtração:   a = b - c
void  aa_trans(AAform a, AAform b, real t);     //Translação   a = b + t
void  aa_neg(AAform a, AAform b);               //Negação:     a = - b
void  aa_scale(AAform a, AAform b, real t);     //Prod. por escalar: a = tb

//Operações não-afins:
void  aa_mul(AAform a, AAform b, AAform c);     //  a = b * c
void  aa_div(AAform a, AAform b, AAform c);     //  a = b / c
void  aa_sqr(AAform a, AAform b);               //  a = b^2
void  aa_exp(AAform a, AAform b);               //  a = exp( b )
void  aa_log(AAform a, AAform b);               //  a = log( b )
void  aa_inv(AAform a, AAform b);               //  a = 1/b
void  aa_sin(AAform a, AAform b);               //  a = sin( b )
void  aa_cos(AAform a, AAform b);               //  a = cos( b )
void  aa_atan(AAform a, AAform b);              //  a = atan( b )

//Operação típica de AA (embora a rotina também funcione em IA):
real aa_norm(AAform a);       //Norma de a = somatorio, de 1 a n, dos xi
```

6.2 Calculando expressões com as bibliotecas

A seguir, são apresentados três exemplos (esfera, gota e toro) que ilustram como as rotinas das duas bibliotecas podem ser utilizadas, de forma transparente, no cálculo intervalar de um grande número de expressões. Os exemplos consistem em rotinas na linguagem C, extraídas da implementação, e utilizam um *array*, t, de variáveis temporárias do tipo `AAform`, para armazenar resultados intermediários. A partir de determinado ponto no desenvolvimento do trabalho, a obtenção deste tipo de código, que antes era feita manualmente, foi automatizada com um programa escrito em Lua e com o programa MapleV.

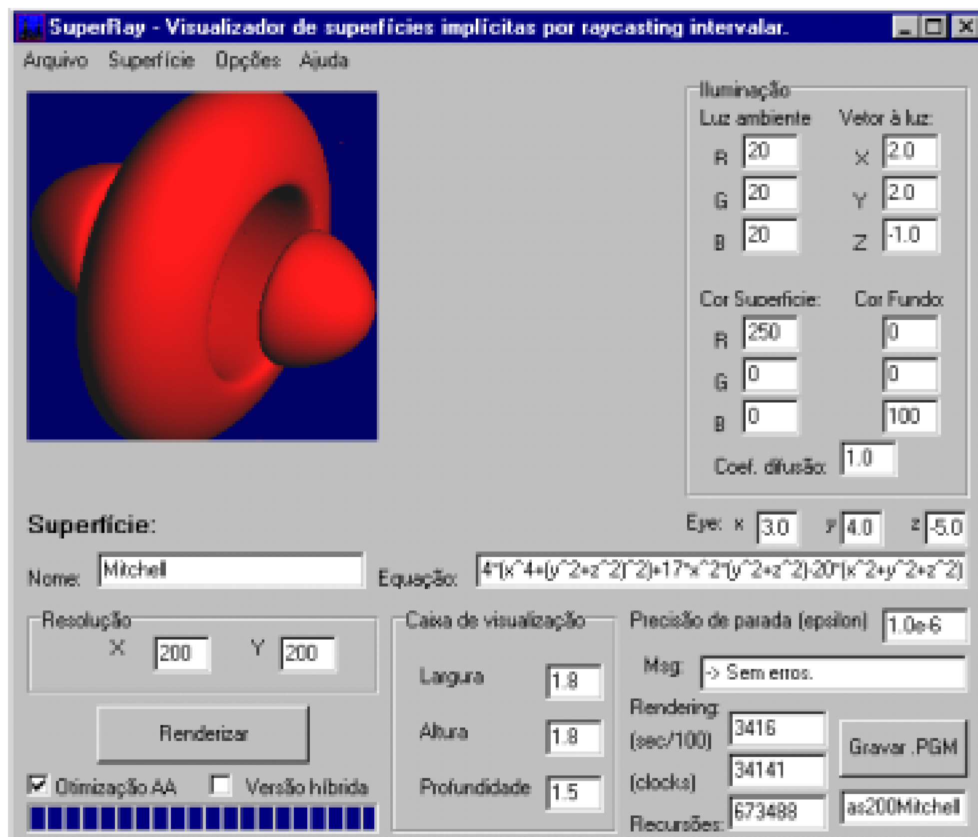
```
//-----------------------------------------------------------------------
void i_esfera(  AAform  x,  AAform  y,  AAform  z, AAform  f  )
{     // esfera:   f(x,y,z) = x^2 + y^2 + z^2 - 1
      static  AAform  t[5];
  aa_sqr(t[0],x);
  aa_sqr(t[1],y);
  aa_sqr(t[2],z);
  aa_add(t[3],t[0],t[1]);
  aa_add(t[4],t[3],t[2]);
  aa_trans(f,t[4],-1.0);
}

//-----------------------------------------------------------------------
void i_gota(  AAform  x,  AAform  y,  AAform  z, AAform  f  )
{           // gota:  f(x,y,z) = 4*(x^2+y^2) - (1+z)*(1-z)^3
            static AAform t[10];
   aa_sqr(t[0],x);
   aa_sqr(t[1],y);
   aa_add(t[2],t[0],t[1]);
   aa_scale(t[3],t[2],4);
   aa_trans(t[4],z,1);
   aa_neg(t[5],z);
   aa_trans(t[6],t[5],1);
   aa_sqr(t[7],t[6]);
   aa_mul(t[8],t[7],t[6]);
   aa_mul(t[9],t[4],t[8]);
   aa_sub(f, t[3],t[9]);
}
//----------------------------------------------------------------------
```

```
void i_toro(  AAform  x,  AAform  y,  AAform  z, AAform  f  )
{            // toro:  f(x,y,z) = (x^2+y^2+z^2+1-0.25)^2 - 4*(x^2+y^2)
      static AAform t[10];
   aa_sqr( t[0],x );
   aa_sqr( t[1],y );
   aa_add( t[2],t[0],t[1]);
   aa_sqr( t[3],z );
   aa_add( t[4],t[2],t[3]);
   aa_trans( t[5],t[4],1);
   aa_trans( t[6],t[5],-0.25);
   aa_sqr( t[7],t[6] );
   aa_scale( t[8],t[2],4 );
   aa_sub( f, t[7], t[8] );
}
```

6.3 O programa SuperRay

A tela principal do aplicativo SuperRay (na sua versão AA, que difere apenas em pequenos detalhes da versão IA) é exibida na figura abaixo:



Capítulo 7 Testes

7.1 Descrição dos testes

Utilizando-se o programa desenvolvido, em suas duas versões básicas (IA e AA), foram realizados testes comparativos entre o desempenho da técnica de visualização de superfícies implícitas por raycasting, utilizando o algoritmo de bisseção intervalar para a determinação das interseções, em diversas versões de implementação e de configuração do algoritmo, resultantes das seguintes variações:

- Utilização de IA ou AA;
- Utilização ou não de otimização específica em AA;
- Versão puramente intervalar (resolução totalmente intervalar das equações) ou versão híbrida (cálculo intervalar da função e de sua derivada, na etapa de isolamento da raiz, e utilização de métodos numéricos na etapa de refinamento).
- Nas versões híbridas, variação do algoritmo convencional utilizado na etapa de refinamento, entre as opções bisseção, *regula-falsi* e Brent.

Foi escolhido, para a medição e comparação dos tempos de visualização, um pequeno elenco de superfícies consideradas interessantes, ou por tratarem-se de exemplos clássicos (como esfera e toro), ou por serem representantes de alguma característica peculiar, como a função de Mitchell, por possuir várias componentes conexas, ou a de Steiner, considerada na literatura como um teste difícil para qualquer *raytracer* de superfícies implícitas. Foram consideradas diferentes formas de apresentação da mesma equação, o que explica o fato de alguns exemplos terem uma versão "expandida", em contraste à versão mais condensada de suas equações.

7.2 Exemplos

A seguir, são listadas as equações dos principais exemplos, juntamente com o resultado de sua visualização pelo programa:

Esfera: $x^2 + y^2 + z^2 - 1 = 0$

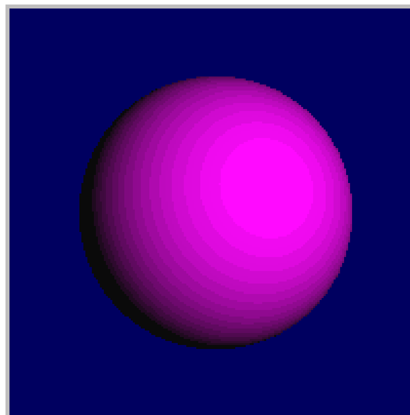


Figura 7.1 – Esfera.

Gota:

$4(x^2 + y^2) - (1 + z)(1 - z)^3 = 0$

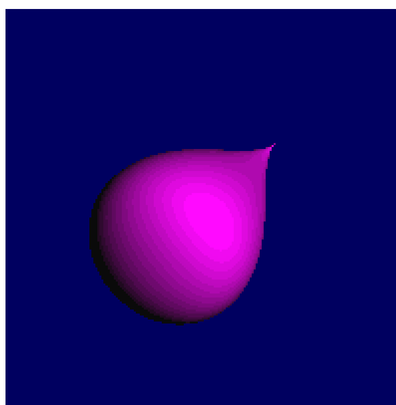


Figura 7.2 – Gota.

Gota (expandida):

$4x^2 + 4y^2 - 1 + 2z - 2z^3 + z^4 = 0$

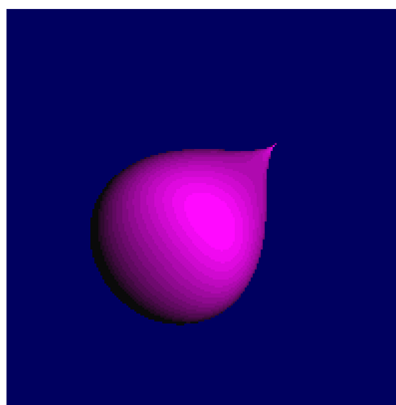


Figura 7.3 - Gota (equação expandida).

Toro: $(x^2 + y^2 + z^2 + 1.0 - 0.25)^2 - 4(x^2 + y^2) = 0$

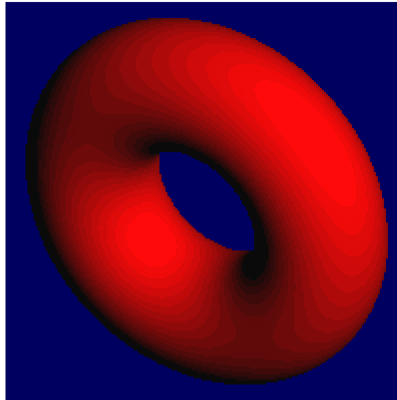


Figura 7.4 - Toro.

Bitoro:

$(4x^2(1 - x^2) - y^2)^2 + z^2 - 0.25 = 0$

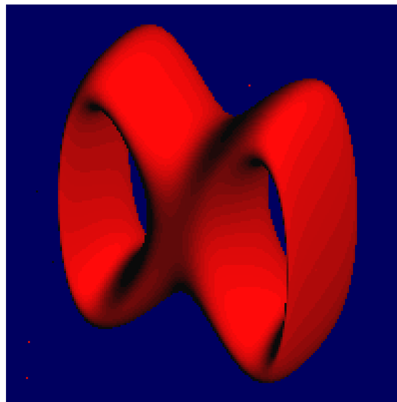


Figura 7.5 - Bitoro.

Bitoro (versão expandida):

$$16x^4 - 32x^6 - 8x^2y^2 + 16x^8 + 8x^4y^2 + y^4 + z^2 - 0.25 = 0$$

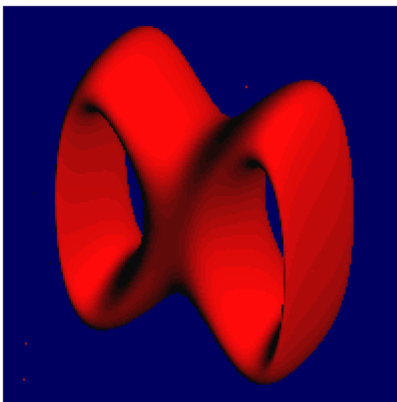


Figura 7.6 - Bitoro (equação expandida).

Função de Mitchell:

$$4(x^4 + (y^2 + z^2)^2) + 17x^2(y^2 + z^2) - 20(x^2 + y^2 + z^2) + 17 = 0$$

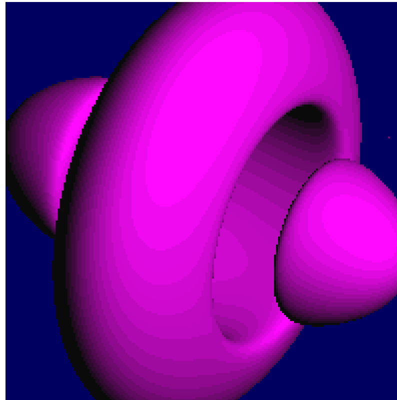


Figura 7.7 - Superfície de Mitchell.

"Six-peak": $(3x^2 - y^2)^2y^2 - (x^2 + y^2)^4 - z = 0$

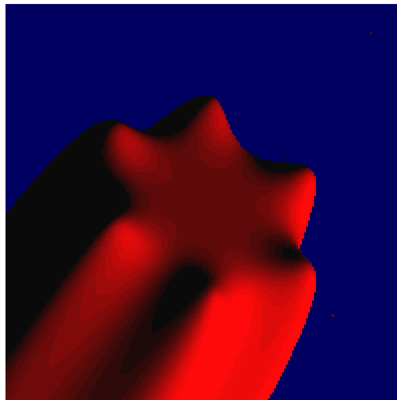


Figura 7.8 - "Six-peak".

Função de Steiner:

$$x^2y^2 + y^2z^2 + z^2x^2 + xyz = 0$$

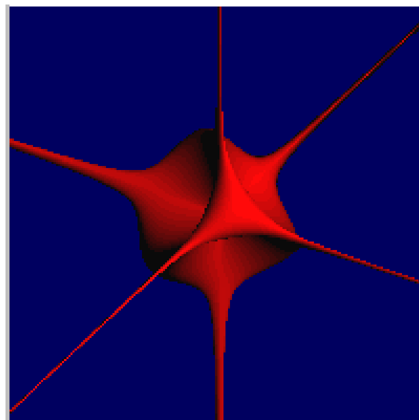


Figura 7.9 - Superfície de Steiner.

Durante os testes, foi feita a visualização de cada exemplo com variações em diversos parâmetros que, além de influenciarem a imagem resultante, mostraram fatores de influência sobre o tempo total de execução do algoritmo, bem como no desempenho relativo entre as versões implementadas. Os principais parâmetros testados são:

- Variação na posição (coordenadas cartesianas, *x*, *y* e *z*) do ponto de vista (***eye***) para a cena. Foram feitas variações na direção (vetor que contem o *eye* e a origem, (0,0,0) do sistema de coordenadas da cena) e na distância deste ponto à origem, particularmente explorando algumas direções peculiares como, por exemplo, o *eye* localizado em um dos eixos (*x*,*y* ou *z*).

Obs: apesar de o programa utilizar projeção ortogonal (os raios lançados no algoritmo de raycasting são paralelos), este ponto é utilizado como referência (centro) para o cálculo do plano de projeção, do qual são lançados os raios.

- Resolução das imagens geradas (linhas de pixels × pixels por linha).

- Variação da tolerância (*epsilon*), utilizada como critério de parada no refinamento das raízes das equações (no cálculo das interseções raio-superfície), tanto nas versões puramente intervalares como nas híbridas, na etapa de refinamento por algoritmos numéricos.

- Variação das dimensões da caixa de visualização (***box***) da cena. A box consiste em um paralelepípedo em cujo interior são lançados os raios, no raycasting.

Obs.: A face frontal da *box* coincide com o plano de visualização, de onde os raios são lançados em onde, teoricamente, forma-se a imagem. A profundidade da *box* é usada como intervalo inicial de valores para o parâmetro *t* (sobre o qual está parametrizado cada raio), no algoritmo de bisseção intervalar, que computa a primeira interseção de cada raio com a superfície visualizada.

7.3 Resultados dos testes

A seguir são apresentados os resultados do desempenho das diferentes versões do programa. A máquina utilizada para a maioria dos testes foi um PC equipado com processador Pentium de 166MHz, com 32Mb de memória RAM, executando Windows 95, com esquema de memória virtual e sistema de arquivos a 32bits, informando 80% de seus recursos livres.

A grande maioria das tabelas exibe os tempos totais de visualização da cena, para cada exemplo, em cada versão do algoritmo, expressos em centésimos de segundo. A exceção é a tabela 7.2, que expressa, em cada caso, o número total de recursões do algoritmo na cena (o somatório das recursões necessárias a cada pixel). No caso das versões híbridas, são computadas apenas as recursões intervalares, ou seja, aquelas em que o algoritmo faz contas com intervalos, antes de entrar no refinamento por algum método numérico convencional. Além disso, são exibidos apenas os resultados das versões híbridas utilizando bisseção e Brent.

Obs: As células em branco indicam combinações de exemplos e versões que não foram processadas, devido a algumas decisões de implementação. Em cada linha está destacado, em negrito, o melhor resultado para a visualização do exemplo em questão.

- **Tabela 7.1 - Conjunto padrão de parâmetros de cena:**

Resolução: 64 x 64 Epsilon: 1.0^{-6} Box: 1.5,1.5,1.5 Eye: (3.0, 4.0, -5.0)

Superfície	IApuro	IAbis	IAbrent	AAOpuro	AAObis	AAObrent	AApuro	AAbis	AAbrent
esfera	115	98	98	**77**	83	82	154	116	116
gota	**143**	150	148	226	304	305	297	340	340
gota(exp)	269			**165**			275		
toro	390	247	246	**115**	203	203	302	387	385
bitoro	**192**	193	193	300	574	572	478	1060	1050
bitoro(exp)	2885			**730**			1130		
Mitchell	1164	550	542	**285**	519	520	642	1134	1130
six-peak	340	395	396	**260**	705	703	560	1270	1260
Steiner	**330**	422	422	460	1052	1040	554	1376	1380

- **Tabela 7.2 - Número de recursões intervalares, para o mesmo conjunto de parâmetros:**

Superfície	IApuro	IAbis	IAbrent	AAOpuro	AAObis	AAObrent	AApuro	AAbis	AAbrent
Esfera	94.711	41.588	41.588	27.291	16.195	**16.054**	82.470	33.228	33.228
Gota	88.691	**54.222**	**54.222**	74.252	67.794	66.985	110.061	77.000	77.000
gota(exp)	129.222			**47.132**			96.891		
toro	316.567	89.197	89.197	43.027	41.338	**37.951**	147.938	75.289	75.289
bitoro	122.639	**61.402**	**61.402**	109.006	104.320	108.872	210.041	165.212	165.212
bitoro(exp)	568.901			**153.140**			243.679		
Mitchell	469.729	156.945	156.945	71.254	**57.274**	69.332	187.066	147.760	147.760
six-peak	241.952	131.331	131.331	86.924	**81.550**	81.556	202.650	157.905	157.905
Steiner	175.529	130.974	130.974	123.927	117.913	**117.251**	167.370	159.964	159.964

- **Tabela 7.3 - Aumentando-se o intervalo inicial no domínio de *t* (profundidade da box):**

Resolução: 64 x 64 Epsilon: 1.0^{-6} Box: 1.5,1.5,15.0 Eye: (3.0, 4.0, -5.0)

Superfície	IApuro	IAbrent	AAOpuro	AAObrent	AApuro	AAbrent
esfera	137	116	**82**	99	169	124
gota	**190**	210	505	660	633	556
gota(exp)	**335**		340		387	
toro	390	307	**220**	360	317	440
bitoro	**230**	264	604	1140	734	1183
bitoro(exp)	2970		**1220**		1322	
Mitchell	1280	626	**516**	870	725	1281
six-peak	403	465	**280**	765	602	1455
Steiner	**370**	560	910	2000	789	1236

- **Tabela 7.4 - Aumentando-se a resolução da cena (de 64 x 64 para 128 x 128) :**

Superfície	IApuro	IAbrent	AAOpuro	AAObrent	AApuro	AAbrent
esfera	472	404	**316**	332	600	464
gota	**580**	596	868	1240	1140	1324
gota(exp)	1092		**676**		1132	
toro	1548	992	**448**	784	1160	1596
bitoro	**772**	800	1156	2316	1892	4224
bitoro(exp)	11568		**2972**		4420	
Mitchell	4664	2092	**1120**	2016	2516	4644
six-peak	1308	1532	**1032**	2800	2172	5192
Steiner	**1264**	1688	1820	4120	2136	5552

• **Tabela 7.5 - Alterando-se a direção da reta suporte do ponto de visualização (*eye*), para coincidir com o eixo de coordenadas *z* :**

Resolução: 64 x 64 Epsilon: 1.0^{-6} Box: 1.5,1.5,1.5 Eye: (0.0,0.0,-5.0)

Superfície	IApuro	IAbrent	AAOpuro	AAObrent	AApuro	AAbrent
esfera	73	**60**	71	77	132	150
gota	**127**	143	242	315	276	290
gota(exp)	**190**		192		260	
Toro	138	**105**	115	192	270	298
bitoro	94	**74**	88	127	446	950
bitoro(exp)	**125**		148		1060	
Mitchell	830	500	**190**	274	590	1045
six-peak	310	278	**240**	295	510	1130
Steiner	**149**	187	165	280	534	1270

• **Tabela 7.6 - Utilizando-se um valor de *epsilon* 10^3 vezes maior:**

Resolução: 64 x 64 Epsilon: 1.0^{-3} Box: 1.5,1.5,1.5 Eye: 3.0,4.0,-5.0)

Superfície	IApuro	IAbrent	AAOpuro	AAObrent	AApuro	AAbrent
Esfera	88	93	**61**	76	83	108
Gota	**121**	148	183	242	300	313
gota(exp)	198		**126**	186		
Toro	205	236	**92**	115	215	341
Bitoro	**127**	188	250	305	550	988
bitoro(exp)	970		**564**	760		
Mitchell	465	495	**237**	285	480	1020
six-peak	236	360	**207**	269	680	1168
Steiner	**245**	400	358	455	1040	1245

- **Tabela 7.7 - Utilizando-se um valor de *epsilon* 10^3 vezes menor:**

Resolução: 64 x 64 Epsilon: 1.0^{-9} Box: 1.5,1.5,1.5 Eye: (3.0,4.0,-5.0)

Superfície	IApuro	IAbrent	AAOpuro	AAObrent	AApuro	AAbrent
Esfera	132	112	**85**	88	189	135
Gota	**163**	171	252	249	337	396
Gota(exp)	321		**196**		335	
Toro	446	295	**139**	142	358	433
Bitoro	**223**	237	364	335	536	1220
Bitoro(exp)	3429		**857**		1285	
Mitchell	1398	659	**331**	340	763	1318
Six-peak	420	477	312	**310**	641	1415
Steiner	**407**	511	521	580	668	1624

- **Tabela 7.8 - Rodando em uma máquina com maior capacidade de processamento: *Pentium* 300 MHz, 32 Mb RAM, e todo o restante da configuração igual à máquina anterior.**

Superfície	IApuro	IAbrent	AAOpuro	AAObrent	AApuro	AAbrent
Esfera	96	75	**57**	59	118	87
Gota	**115**	122	170	229	225	259
Toro	330	194	**83**	146	221	281
Bitoro	**153**	157	218	426	365	828
Mitchell	917	426	**211**	375	476	816
Six-peak	268	315	**196**	522	432	992
Steiner	**269**	335	329	764	422	1031
Total	2148	1624	1264	2521	2259	4294
Total anterior	2674	2045	1723	3425	2987	5661
Ganho (%)	19,7	20,6	***26,6***	26,4	24,4	24,1

Capítulo 8 Análise dos resultados e conclusões

Baseado na análise dos resultados apresentados na seção anterior, e na utilização do programa de um modo geral, foi possível extrair algumas conclusões a respeito do trabalho desenvolvido. Um resumo de tais conclusões é apresentado a seguir:

- Observando-se qualquer uma das tabelas, pode-se perceber que nenhuma das diferentes versões implementadas apresentou uma vantagem generalizada, conseguindo superar todas as outras em eficiência para todos os exemplos. O desempenho relativo entre as versões mostrou-se bastante dependente do exemplo utilizado. Enquanto a maioria das superfícies foi visualizada com mais eficiência com a versão **AAO pura** (Mitchell, toro, esfera, "six-peak", versões expandidas), outras favoreceram a utilização de IA (gota, bitoro, Steiner).

- Observando-se a tabela 7.2, fica claro que AA realmente leva a um menor número de recursões nos algoritmos. Em vários casos, esta vantagem é suficiente para compensar o fato de as contas serem mais caras com AA; em outros, não chega a compensar. Não foi possível detectar um padrão (um critério que permita predizer qual exemplo é mais favorável a cada técnica) a partir da observação da geometria ou da topologia das superfícies. É possível, sim, analisar-se as equações das mesmas. Particularmente, equações com grande incidência de diferentes produtos de termos intervalares (variáveis ou sub-fórmulas) mostraram-se piores para o desempenho de AA. Como exemplo, podemos citar o caso da equação da função de Steiner: $x^2y^2 + y^2z^2 + z^2x^2 + xyz$. É interessante notar que quadrados não estão no mesmo caso. São produtos dos mesmos termos e, nesse caso, AA leva vantagem por conhecer a correlação entre estes termos (que neste caso é total). Com potências genéricas ocorre algo um

pouco diferente, ao menos na implementação de AA utilizada neste trabalho: um termo como x^5 é visto como x^4x, que é calculado como $(x^2)^2x$. Mesmo assim, AA conhece as relações.

- A utilização de AA sem otimização mostrou-se não compensadora, sendo sempre mais lenta que AA otimizada e, na grande maioria das vezes, mais lenta que as versões IA. Entretanto, não há motivo aparente para que não se utilize a otimização em AA, ao menos em algoritmos semelhantes ao de bisseção intervalar. Fazer isto significaria desperdiçar características peculiares à forma como AA representa os valores, utilizando-a simplesmente como uma "imitação" de IA.

- A expansão das equações leva a uma eficiência menor. No entanto, este efeito mostrou-se bem mais intenso em IA do que em AA. Exemplos que na versão condensada foram mais rápidos com IA favoreceram AA em suas versões expandidas. Além disso, as versões híbridas do algoritmo em geral são menos afetadas na passagem de uma versão condensada para uma expandida da mesma equação.

- Enquanto IA é beneficiada na passagem do algoritmo puramente intervalar para o algoritmo híbrido, AA não sofre o mesmo benefício, principalmente no caso de AA otimizada. Mesmo assim, para fins de implementação, é importante deixar claro que, para maior eficiência das versões híbridas de AA, é fundamental que o código com as rotinas para o cálculo intervalar da função e de sua derivada seja gerado − provavelmente, de forma automatizada − de uma só vez, para que seja identificado o maior número possível de termos em comum.

- A diferença entre a utilização de um ou outro método numérico para o refinamento das raízes (bisseção, *regula-falsi*, Brent), nas versões híbridas, mostrou não ser significativa em termos de eficiência. Particularmente, a diferença entre o uso de bisseção, considerada um algoritmo "lento", e de Brent, considerado como um dos mais "rápidos" disponíveis na literatura, foi muito pequena. Sem dúvida, o que contribui para esta pequena diferença é o fato de o tempo gasto nesta etapa ser pequeno quando comparado à etapa de isolamento de raízes,

onde são feitas contas com intervalos, mais caras. Além disso, é provável que seja pequeno o número de iterações destes algoritmos necessárias na etapa de refinamento desta aplicação e, não chagando a configurar uma grande vantagem de um método sobre outro. Entretanto, este número não foi medido neste trabalho. Trata-se apenas de uma suposição.

- Baseado nas observações anteriores, e nos resultados apresentados em [Figueiredo & Stolfi 96] – onde a utilização de AA na enumeração adaptativa de superfícies implícitas, por subdivisão espacial, mostrou-se indiscutivelmente vantajosa, levando a uma maior eficiência e à construção de estruturas de dados (*octrees*) menores, com economia de memória – é razoável apresentar a seguinte conclusão: para aplicações semelhantes à desenvolvida neste trabalho, onde o *raycasting* é feito diretamente a partir da descrição matemática das superfícies, sem construção de estruturas de dados auxiliares, não é necessariamente recomendável a utilização de AA em substituição a IA. Já em aplicações onde as superfícies são aproximadas por técnicas de subdivisão espacial semelhantes às descritas em [Figueiredo & Stolfi 96], associado ou não à utilização de *raycasting*, é recomendável a utilização de AA. Inclusive, a junção das duas técnicas, como forma de acelerar a visualização, constitui um interessante desdobramento deste trabalho, citado como trabalho futuro, na seção seguinte.

Capítulo 9 Trabalhos futuros

Ao longo do desenvolvimento deste trabalho surgiram algumas idéias sobre trabalhos a serem desenvolvidos futuramente, dos quais apresentamos um resumo a seguir. Alguns consistem em novos elementos de pesquisa, como variações nos algoritmos ou inclusão de novas técnicas na visualização das superfícies. Outros referem-se simplesmente a melhorias na implementação e inclusão de novos recursos aos aplicativos desenvolvidos, que não foram ainda realizadas devido a limitações inerentes a prazos.

- Alteração simples do aplicativo desenvolvido para permitir a visualização de um número ilimitado de exemplos de superfícies, obtidos, por exemplo, com a entrada das equações pelo usuário.

- Utilização de técnicas de minimização do efeito de **serrilhado** (algumas vezes referido, de forma equivocada, como *aliasing*) às bordas das superficies visualizadas, a fim de melhorar qualidade da imagem final. Uma possível abordagem para isto seria utilizar um algoritmo adequado para identificar quais as regiões da imagem que apresentam serrilhado e fazer mais de uma amostragem de cor (lançar mais de um raio por pixel) nestas regiões.

- Ampliação do aplicativo para um sistema de modelagem CSG completo, permitindo a combinação de várias superfícies implícitas através de operações booleanas. Isto implica basicamente em duas alterações simples: a construção de uma estrutura de dados árvore binária CSG e a adaptação das versões do algoritmo de bisseção intervalar para determinar e armazenar todas as interseções de cada raio.

* Tentativa de explorar aspectos de coerência geométrica das superfícies para acelerar a visualização, principalmente a partir da utilização, como entrada inicial para o algoritmo de bisseção intervalar, de intervalos menores no domínio do parâmetro *t.* Isto significa menos recursões e uma convergência mais rápida. Por exemplo: se o algoritmo está no meio da visualização de uma região contínua de uma superfície, provavelmente o ponto de interseção com o próximo raio estará próximo do ponto encontrado para o raio anterior. Se isto for verdade, o algoritmo não precisa iniciar a busca das interseções para o próximo raio com o intervalo inicial padrão em *t* (aquele obtido, por exemplo, a partir da profundidade de uma *bounding box* da cena), que chamaremos de T. Pode tentar partir de um intervalo em *t* menor, vizinho ao valor de *t* da interseção com o último raio. Se esta aposta falhar, o algoritmo pode voltar atrás e partir do intervalo padrão T. Esta idéia é mostrada na figura a seguir:

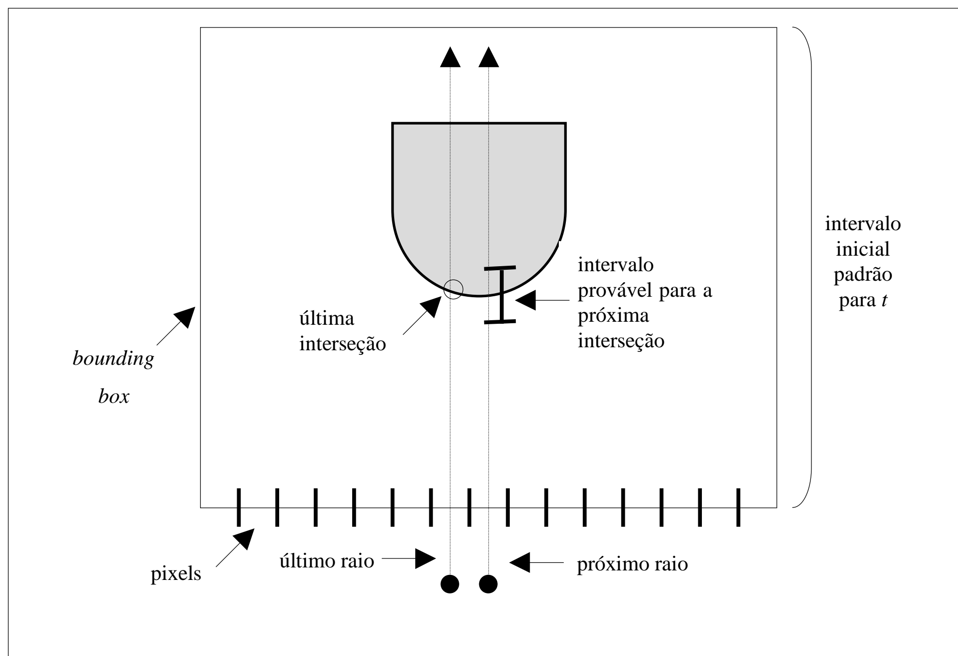


Figura 9.1 - Idéia de exploração da coerência nas superfícies para acelerar o *raycasting*.

Entretanto, esta idéia ainda não é robusta, e carece de evolução. É preciso criar maneiras de prever e tratar casos em que a idéia falha grosseiramente, como o caso mostrado na figura a seguir:

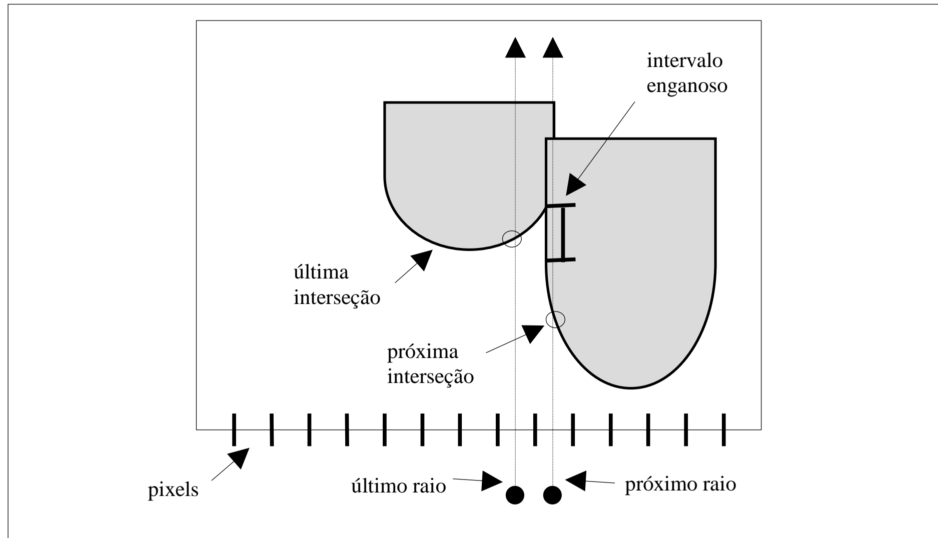


Figura 9.2 - Um dos casos em que a idéia da consistência pode falhar.

- Utilização de estruturas de dados espaciais, como as *octrees* descritas em [Figueiredo & Stolfi 96], como forma de aceleração do *raycasting*, na tentativa de minimizar o número de cálculos de interseções. Levando-se em conta que, neste trabalho, a utilização de AA na enumeração adaptativa das superfícies por subdivisão espacial levou à construção de *octrees* menores (com menos células candidatas a conter a superfície), é razoável supor que a posterior utilização destas estruturas no *raycasting* leve a uma vantagem de desempenho de AA sobre IA maior do que a obtida no presente trabalho. Com menos células candidatas, menos testes de interseção deverão ser feitos. Algumas idéias neste sentido podem ser encontradas em [Glassner 84].

Referências Bibliográficas

[Appel 68] Appel, A. *"Some techniques for shading machine renderings of solids"*, Spring Joint Computer Conference 68, 1968, 37-45.

[Blinn 82] Blinn, J. F. *"A generalization of algebraic surface drawing"* ACM Transactions on Graphics 1 (3), 1982, 235-256.

[Bloomenthal et al. 97] Bloomenthal, J. *et al.*, *Introduction to Implicit Surfaces*, Morgan Kaufman, 1997.

[Bloomenthal 94] Bloomenthal, J. *"An implicit surface polygonizer"*, *Graphics Gems IV*, Academic Press, 1994, 324-349.

[Brent 73] Brent, R. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, 1973, capítulos 3 e 4.

[Comba & Stolfi 93] Comba, J. L., D Stolfi, J., *"Affine arithmetic and its Applications to Computer Graphics"*, Anais do VI SIBGRAPI, 1993, 9-18.

[Corliss 77] Corliss, G. *"Which root does the bisection algorithm find?"*, SIAM Review 19 (2), 1977, 325-327.

[Duff 92] Duff, T. *"Interval arithmetic and recursive subdivision for implicit functions and construtive solid geometry"*, Computer Graphics 26 (2), 1992, 131-138, SIGGRAPH 92 Proceedings.

[Figueiredo & Stolfi 96] Figueiredo, L. H. de, Stolfi, J., *"Adaptative Enumeration of Implicit Surfaces with Affine Arithmetic"*, Computer Graphics Forum 15, 1996, 287-296.

[Figueiredo & Stolfi 97] Figueiredo, L. H. de, Stolfi, J., *Métodos Numéricos Auto-Validados e Aplicações*, 21º Colóquio Brasileiro de Matemática, IMPA, 1997.

[Foley et al. 90] Foley, J. D. *et al.* *Computer Graphics – Principles and Practice*, Addison-Wesley, 1990.

[Glassner 84] Glassner, A. *"Space subdivision for fast ray tracing"*, IEEE Computer Graphics & Applications, 1984.

[Gomes & Velho 92] Gomes, J. M., Velho, L., *Implicit Objects in Computer Graphics,* Monografias de Matemática Nº 53, IMPA, 1992.

[Hanrahan 83] Hanrahan, P. *"Ray tracing algebraic surfaces",* Computer Graphics, 17 (3), 1983, 83-90.

[Hansen 78] Hansen, E. *"A globally convergent interval method for computing and bounding real roots"*, BIT 18(4), 1978, 415-424.

[Hart 93] Hart, J. C. *"Ray tracing implicit surfaces"*, SIGGRAPH 93 Course Notes, 1993.

[Ierusalimschy et al 96] Ierusalimschy, R., Figueiredo, L. H. de, Celes, W. *"Lua - an extensible extension language"*, Software: Practice & Experience, 26 (6), 1996, 635-652.

[Kalra & Barr 89] Kalra, D., Barr, A. H., *"Guaranteed ray intersection with implicit surfaces"*, Computer Graphics 23 (3), 1989, 297-306.

[Kernighan & Ritchie 90] Kernighan, B., Ritchie, D. *A Linguagem de Programação C− Padrão ANSI*, Editora Campus, 1990.

[Kleck 89] Kleck, J. *"Modeling using implicit surfaces"*, Master's thesis, University of California at Santa Cruz, 1989.

[Mitchell 90] Mitchell, D. P., *"Robust ray intersection with Interval Arithmetic"*, Graphics Interface 90, 1990, 68-74.

[Mitchell 91] Mitchell, D. P., *"Three applications of Interval Analysis in Computer Graphics"*, SIGGRAPH' 91 Course Notes: Frontiers in Rendering, 1991.

[Moore 66] Moore, R. E., *Interval Analysis*, Prentice-Hall, 1966.

[Moore 79] Moore, R. E., *Methods and Applications of Interval Analysis*, SIAM, 1979.

[Munem & Foulis 82] Munem, M., Foulis, D. *Cálculo − Volume 1*, Editora Guanabara, 1982.

[Phong 75] Phong, B., *"Illumination for computer–generated pictures"*, Comm. ACM, 18 (3), 1975, 311-317.

[Press et al. 92] Press *et al.*, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

[Requicha 80] Requicha, A., *"Representation of Rigid Solids: Theory, Methods and Systems"* ACM Computing Surveys, 12 (4), 1980.

[Rockwood & Owen 87] Rockwood, A. P., Owen, J. *"Using implicit surfaces to blend arbitrary solid models"*, in *Geometric Modeling: Algorithms and Trends*, SIAM, 1987.

[Santos 76] Santos, V. R. de B., *Curso de Cálculo Numérico, 3ª edição*, LTC, 1976.

[Schildt 96] Schildt, H. *C Completo e Total – 3ª edição*, MAKRON Books, 1996.

[Snyder 92] Snyder, J. M. *"Interval analysis for Computer Graphics",* Computer Graphics 26 (2), 1992, 121-130, SIGGRAPH 92 Proceedings.

[Toth 85] Toth, D. L. *"On ray tracing parametric surfaces"*, Computer Graphics 19 (3), 1985, 171-179.

[Watt 93] Watt, A. *3D Computer Graphics – 2nd edition*, Addison-Wesley, 1993.

[Watt & Watt92] Watt, A., Watt, M. *Advanced Animation and Rendering Techniques – Theory and Practice*, Addison-Wesley, 1992.

[Whitted 80] Whitted, T. *"An Improved Illumination Model for Shaded Display"*, Communications of the ACM, 23 (6), 1980.