

TÉCNICAS PARA VISUALIZAÇÃO DIRETA DE MODELOS CSG

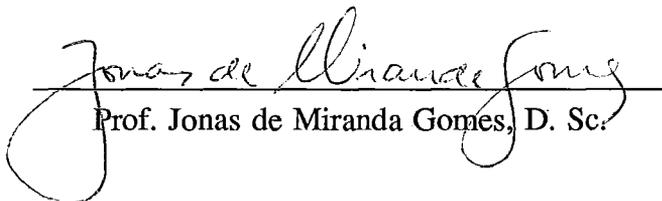
Claudio Esperança

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Ronaldo Cesar Marinho Persiano, D. Sc.
(Presidente)



Prof. Jonas de Miranda Gomes, D. Sc.



Prof. Marcelo Gattass, Ph. D.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 1990

ESPERANÇA, CLAUDIO

Técnicas para Visualização Direta de
Modelos CSG [Rio de Janeiro] 1990

VII, 152 p. 29,7 cm (COPPE/UFRJ, M. Sc.,
Engenharia de Sistemas e Computação, 1990)

Tese - Universidade Federal do Rio de
Janeiro, COPPE

1. Modelagem de Sólidos I. COPPE/UFRJ

II. Título (série).

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M. Sc.).

TÉCNICAS PARA VISUALIZAÇÃO DIRETA DE MODELOS CSG

Claudio Esperança

Junho de 1990

Orientador: Prof. Ronaldo Cesar Marinho Persiano

Programa: Engenharia de Sistemas e Computação

Este trabalho aborda diversos aspectos ligados à visualização de sólidos modelados em Geometria Sólida Construtiva. São apresentados algoritmos de visualização direta baseados nas técnicas de disparo de raios e decomposição espacial. Diversas idéias são introduzidas com a finalidade de melhorar o desempenho desses algoritmos, com ênfase na aplicação de processos de subdivisão recursiva do espaço.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Sciences (M. Sc.).

TECHNIQUES FOR DIRECT DISPLAY OF CSG MODELS

Claudio Esperança

June, 1990

Thesis Supervisor: Prof. Ronaldo Cesar Marinho Persiano

Department: Engenharia de Sistemas e Computação

This work covers many aspects related to CSG models display. Algorithms for direct solid display, based on ray-casting and space decomposition techniques, are presented. Some new ideas to enhance the performance of these algorithms are introduced, based mainly on the recursive spatial subdivision paradigm.

Dedico este trabalho aos meus pais, que me incentivaram, e a Astrid, que entendeu meus motivos.

Desejo agradecer a todo o pessoal do Laboratório de Computação Gráfica por seu apoio e dedicação, a meus colegas da COPPE, particularmente João Luiz pela ajuda inestimável, e ao Prof. Marinho por mostrar-me novos caminhos.

ÍNDICE

Capítulo I: Introdução	1
Capítulo II: Geometria Sólida Construtiva	3
Capítulo III: Visualização de modelos CSG	16
Capítulo IV: Subdivisão espacial	25
Capítulo V: Visualização através de disparo de raios	36
Capítulo VI: Visualização através do uso intensivo de subdivisão espacial	74
Capítulo VII: Considerações finais	117
Referências Bibliográficas	120
Apêndice A: Linguagem para Definição de Sólidos CSG	126
Apêndice B: Códigos LDS dos sólidos de teste	138
Apêndice C: Guia para executar os programas	146

Capítulo I

Introdução

O presente trabalho se insere no ramo da computação gráfica conhecido como "Modelagem de Sólidos". Este termo está ligado a um conjunto de técnicas e conhecimentos que nos permite tratar objetos sólidos com o auxílio de um computador. A principal motivação que deu origem ao surgimento dessa ciência pode ser atribuída à necessidade de obter uma metodologia automatizada para o desenho e manufatura de partes mecânicas. Essa demanda já era preenchida em parte pelos chamados sistemas de Projeto Assistido por Computador ou CAD (*Computer Aided Design*). Tais sistemas entretanto eram incapazes de representar e modelar objetos sólidos de maneira unificada - basicamente o que faziam era tratar entidades tridimensionais mais simples tais como linhas e polígonos, sendo que a maior parte da complexidade envolvida ficava a cargo do operador. À medida que os problemas envolvidos nesse tipo de modelagem iam ficando mais conhecidos, foram aparecendo uma série de técnicas e algoritmos que acabaram formando uma especialidade dentro da computação gráfica.

Hoje em dia, Modelagem de Sólidos é uma ciência em franco desenvolvimento mas suficientemente madura para ter aplicações em muitas outras áreas além do desenho mecânico, como por exemplo, medicina, artes plásticas, animação e cartografia. Apesar disso, há ainda um vasto potencial a ser explorado e colocado em prática.

Um sistema típico de Modelagem de Sólidos deve ser capaz de permitir ao usuário não só modelar a forma dos objetos, como também determinar muitas de suas propriedades, tais como massa, volume, área, etc. De preferência, essas facilidades devem estar disponíveis em um ambiente interativo onde o usuário possa ter uma rápida resposta aos seus comandos. Esses requisitos não são facilmente alcançados, principalmente no que diz respeito à visualização dos objetos. Por isso mesmo, considerável parcela de esforços tem sido aplicada no desenvolvimento de métodos cada vez mais eficientes para gerar imagens a partir de modelos de sólidos.

Existem várias maneiras para se representar computacionalmente um sólido - destacamos, por exemplo, Representação por Fronteiras (*Boundary Representation - B-Rep*), árvores octais ou *Octrees* e Geometria Sólida Construtiva (*Constructive Solid Geometry - CSG*). Neste trabalho pretendemos analisar algumas técnicas sendo

correntemente usadas para gerar imagens de sólidos representados neste último sistema, bem como alguns refinamentos dessas técnicas.

O problema de visualização de sólidos, apesar de ser um assunto bastante estudado, pela sua própria complexidade deu margem a diversas abordagens ainda não suficientemente exploradas. É o caso, por exemplo, das técnicas ligadas a subdivisão espacial às quais dispensaremos especial atenção.

É importante salientar que não há aqui o intuito de fazer uma abordagem geral sobre modelagem de sólidos ou mesmo de analisar profundamente todos os aspectos das representações CSG, mas apenas prover uma referência a alguns algoritmos de visualização de sólidos em CSG; muitos desses algoritmos já foram publicados com maior ou menor detalhe na literatura especializada e outros, tanto quanto me é dado saber, estão sendo divulgados pela primeira vez.

Na escolha dos algoritmos de visualização que apresentaremos, demos ênfase àqueles cuja implementação fosse exequível em máquinas sequenciais de uso geral. Essa preferência é explicada tão somente pelo fato de serem esses os equipamentos de que dispúnhamos para implementação. Temos plena consciência que o uso de arquiteturas especializadas e máquinas de processamento paralelo é hoje em dia alvo de intensa pesquisa e um trabalho nesse sentido está também no nosso horizonte.

Não abordaremos também métodos de visualização que impliquem em nos afastarmos muito da esfera das representações CSG. Estamos nos referindo mais especificamente às técnicas de visualização que impliquem numa mudança de representação de CSG para outro sistema. Assim, algoritmos de avaliação de fronteiras (*boundary evaluation*) ou conversão de CSG para Octrees estão fora do escopo deste trabalho, embora algumas técnicas associadas a estes assuntos sejam utilizadas.

Capítulo II

Geometria Sólida Construtiva

1. Modelos de representação de sólidos

Um ponto central no estudo da modelagem de sólidos é o estabelecimento de um conjunto mínimo de informações sobre o objeto que permita seu tratamento em um computador. Entre os diversos aspectos relacionados com um objeto sólido, sem dúvida o que merece maior atenção é sua geometria. Um conjunto de informações que descreva implícita ou explicitamente a geometria de objetos sólidos é chamado de *modelo de representação de sólidos*.

Não há hoje em dia um consenso sobre qual conjunto de informações é mais apropriado para descrever um sólido consistentemente, mesmo porque, segundo o tipo de aplicações que se tem em mente, um determinado conjunto pode apresentar mais vantagens do que outro. Em realidade, vários esquemas de representação foram propostos, sendo que os mais comuns podem ser classificados em uma das seguintes categorias:

1.1. Esquemas de decomposição do espaço

Nesses tipos de representação, o espaço é entendido não como um contínuo, mas sim como composto de um arranjo regular de unidades discretas de volume aos quais se dá o nome de *voxels*. Um sólido segundo essa abstração, pode ser descrito através de uma enumeração dos voxels nele contidos.

Tais esquemas são, por sua própria natureza, aproximativos. Para que se obtenha um grau razoável de precisão os voxels devem ser de pequenas dimensões. Isto tem como decorrência o fato que sólidos representados dessa forma tendem a gastar uma grande quantidade de memória. Na busca de uma maneira eficiente de armazenamento foi introduzido um tipo de estrutura de dados baseado em árvores de grau 8, isto é, onde cada nó interno possui oito filhos. Essa estrutura foi batizada portanto com o nome de *octree*.

Octrees se tornaram bastante populares nos últimos anos, principalmente com a queda nos preços dos "chips" de memória e dos dispositivos de armazenamento secundário.

1.2. Modelos de representação por fronteiras

Nesses modelos, os conjuntos de pontos que compõem um sólido são representados através de sua fronteira. A fronteira que delimita um sólido é normalmente apresentada como um conjunto de *faces*. Por sua vez, faces são descritas através de um conjunto de curvas unidimensionais que a limitam (*arestas*). Analogamente, também as arestas são limitadas por pontos (*vértices*). Pode-se entender então as representações por fronteiras como uma hierarquia de modelos.

Representações por Fronteiras (ou *Boundary Representations - B-Reps*) encontram-se entre os esquemas mais utilizados hoje em dia, o que se justifica pelo fato de ser conhecida uma grande quantidade de algoritmos com eles relacionados e estarem baseados em uma teoria bastante bem estudada.

1.3. Modelos construtivos

Aqui, a idéia é representar um sólido por um conjunto de pontos obtido através da combinação de conjuntos mais simples. Aqui também pode-se dizer que há uma hierarquia de modelos, sendo que no seu nível mais simples temos os chamados sólidos primitivos.

Os modelos que seguem essa filosofia são chamados pelo nome geral de Geometria Sólida Construtiva (ou *Constructive Solid Geometry - CSG*). São desse tipo os modelos que iremos estudar neste trabalho.

2. O esquema de representação CSG

Tentaremos nesta seção descrever o esquema de representação de sólidos conhecido como CSG. Para tanto, definiremos a classe de objetos que nos interessa modelar e a forma geral pela qual esses modelos são representados.

2.1. Sólidos representáveis em CSG

Quando dizemos que estamos interessados em montar um modelo, estamos na verdade falando de uma idealização de objetos do nosso mundo real. Como todas as idealizações, um modelo de representação de sólidos é naturalmente sujeito a uma série de restrições que irão limitar a classe de objetos que desejamos representar.

Como ponto de partida precisamos ter em mente um modelo matemático para o próprio espaço em que os sólidos se acham imersos. No nosso caso, tal modelo é dado pelo espaço euclidiano de dimensão 3, isto é, E^3 . Isto provê uma definição provisória:

Um sólido é um subconjunto *limitado e fechado* de E^3 .

Os termos usados na definição acima são emprestados do ramo da matemática conhecido como Topologia de Conjuntos de Pontos. O leitor interessado pode se dirigir às referências JANICH [1] e ARMSTRONG [2] para uma noção formal desses conceitos.

Informalmente falando, podemos entender por *limitado* aqueles conjuntos de pontos para os quais podemos definir uma "bola" suficientemente grande que o contenha. O conceito de *fechado* significa que a "fronteira", ou seja, os pontos-limites do conjunto pertencem ao próprio conjunto.

A definição acima já captura em essência o que normalmente entendemos por *sólido*, porém é ainda muito ampla. Tome por exemplo um conjunto de pontos em E^3 consistindo apenas de linhas e pontos isolados - tal conjunto não pode ser caracterizado como sólido. É necessário portanto acrescentar um requisito adicional que descarte subconjuntos de dimensão 0, 1 ou 2, tais como pontos, linhas ou superfícies. Este conceito pode ser expresso formalmente como:

A *regularização* de um conjunto de pontos A , chamada $r(A)$ é definida como $r(A) = \text{fecho}(\text{interior}(A))$. Conjuntos tais que $A \equiv r(A)$ são ditos *regulares*.

Podemos pensar informalmente na regularização de um conjunto como um processo onde se descarta todas as partes "isoladas" desse conjunto, cobre os pontos restantes com uma "casca" muito justa e preenche o resultado com material. REQUICHA [3] introduziu o termo *r-set* para denotar conjuntos regulares.

Um conjunto regular pode de maneira geral ser limitado por um número arbitrariamente grande de superfícies de fronteira. No caso concreto de um sistema de modelagem de sólidos CSG, é necessário ainda restringir os sólidos representáveis àqueles limitados por um número finito de superfícies exprimíveis por equações simples, geralmente polinômios em x , y e z . REQUICHA [3] formaliza esse ponto estabelecendo que as superfícies que delimitam os sólidos primitivos devem ser analíticas por partes.

Resumindo, definiremos *sólido* da seguinte forma:

Sólido é um subconjunto limitado por superfícies semi-analíticas, fechado e regular de E^3 .

2.2. Componentes de uma representação CSG

Podemos pensar em Geometria Construtiva Sólida como um sistema de representação onde a forma do sólido final é conseguida através operações que, partindo do espaço vazio, vão sucessivamente "acrescentando" ou "retirando" material. Esses incrementos e decrementos de material são conjuntos regulares simples, denominados *sólidos primitivos* que podem ser deformados e posicionados arbitrariamente no espaço de modelagem através de transformações lineares afins. A forma pela qual esses conjuntos são combinados entre si obedecem a operações bem definidas em Topologia. A representação CSG de um sólido é portanto composta de três tipos de elementos, a saber:

2.2.1. Sólidos primitivos

São conjuntos de pontos suficientemente simples para serem representados por um número pequeno de inequações algébricas. Por exemplo, uma esfera de raio 1 centrada na origem pode ser expressa pela inequação

$$x^2+y^2+z^2 \leq 1 \quad (\text{II.1})$$

Já um cilindro de seção circular com o eixo posicionado sobre o eixo coordenado z , raio unitário e limitado pelos planos $z=0$ e $z=1$ pode ser expresso pelo seguinte sistema de três inequações

$$\begin{cases} x^2+y^2 \leq 1 \\ z \leq 1 \\ z \geq 0 \end{cases} \quad (\text{II.2})$$

A maioria das implementações de sistemas CSG utilizam como sólidos primitivos objetos delimitados por planos ou superfícies quádricas.

2.2.2. Operações entre conjuntos

São operações derivadas da teoria de conjuntos de pontos que têm a função de combinar dois conjuntos de pontos já conhecidos. Os operadores mais comumente utilizados em sistemas CSG correspondem às operações de união (\cup), interseção (\cap) e diferença (\setminus), cujas definições são as seguintes:

$$\begin{aligned} A \cup B &= \{p \in E^3 \mid p \in A \text{ ou } p \in B\} \\ A \cap B &= \{p \in E^3 \mid p \in A \text{ e } p \in B\} \\ A \setminus B &= \{p \in E^3 \mid p \in A \text{ e } \neg p \in B\} \end{aligned} \quad (\text{II.3})$$

(Obs.: Na verdade, nos interessa usar operadores *regularizados* de conjuntos. O conceito de regularidade é abordado mais adiante). de conjuntos

2.2.3. Operações de instanciação

Esses componentes têm basicamente dois objetivos: (1) posicionar uma primitiva dentro da cena e (2) determinar precisamente as dimensões e a forma da primitiva.

O posicionamento do objeto dentro do espaço é normalmente especificado através de transformações rígidas, isto é, assume-se a primitiva segundo uma posição e orientação inicial dentro da cena, e através de rotações e translações esta é movida até o local desejado.

A segunda utilidade dos operadores de instanciação liga-se ao conceito de "família de objetos". Quando dizemos, por exemplo, "primitiva cilindro", temos apenas uma idéia geral da forma do objeto ao qual queremos nos referir, isto é, estamos falando de uma *família* de objetos com forma cilíndrica. Através de operadores de instanciação podemos especificar uma deformação para um cilindro básico - digamos, com raio e altura unitários - de forma a obter qualquer outro membro da *família cilindro*.

De maneira geral, podemos usar em modelos CSG quaisquer transformações de deformação, desde que essas sejam inversíveis. Tal restrição é feita de modo a evitar transformações que, por exemplo, destruam a "solidez" do objeto.

Os operadores de instanciação são normalmente expressos por transformações lineares - operadores de movimento rígido (translação e rotação) e de deformação linear (escala) - que podem ser aplicados em cascata. Por exemplo, para obter um elipsóide não centrado na origem a partir de uma esfera como a descrita acima, poderíamos aplicar um operador de escala seguido de um operador de translação.

2.3. Árvores CSG

Como podemos ver, a definição de um sólido pode ser expressa por uma expressão algébrica, e como tal admite uma produção recursiva da forma:

```
<sólido CSG> ::=  
    <sólido CSG> <operador> <sólido CSG> |  
    <instanciação> <sólido CSG> |  
    <sólido primitivo>
```

Este tipo de estrutura se presta sob medida para ser representada através de uma árvore binária, e é isto o que é feito normalmente. É comum encontrarmos na literatura o termo "árvore CSG" associado à representação de um sólido em CSG. Uma

implementação comum de árvore CSG possui em suas folhas as descrições dos sólidos primitivos aos quais já foram aplicados os operadores de instanciação, e em seus nós internos as descrições das operações de conjunto (veja a figura II.1).

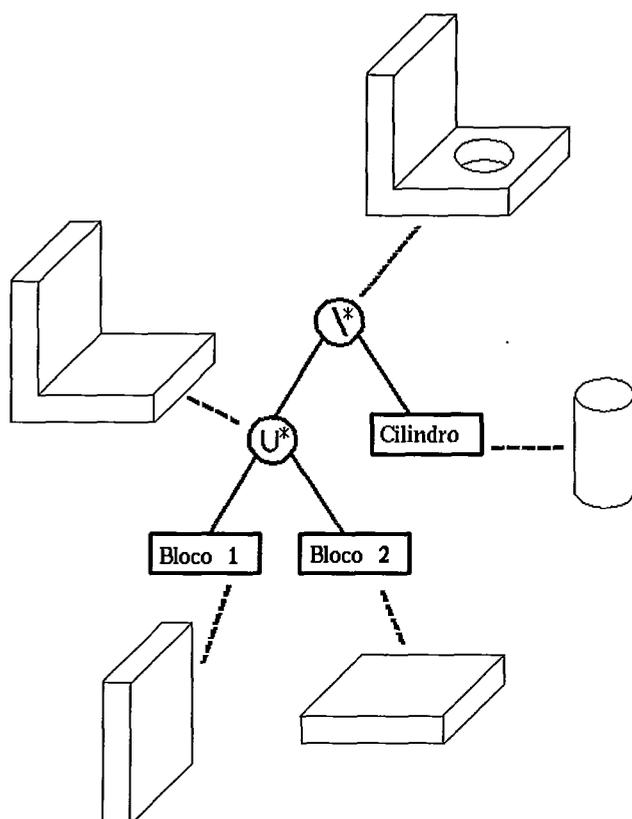


Figura II.1
Uma árvore CSG

Aqui abrimos um parêntese para discutir brevemente algumas variações desse esquema:

- Algumas implementações de CSG substituem o conceito de sólido primitivo pelo de semiespaço primitivo (veja SAMET [4]). Segundo essa noção, por exemplo, o sólido primitivo cilindro como descrevemos acima seria encarado como uma

árvore CSG em cujas folhas estariam as três inequações representando os semiespaços e em seus nós internos encontraríamos duas operações de interseção (veja a figura II.2)

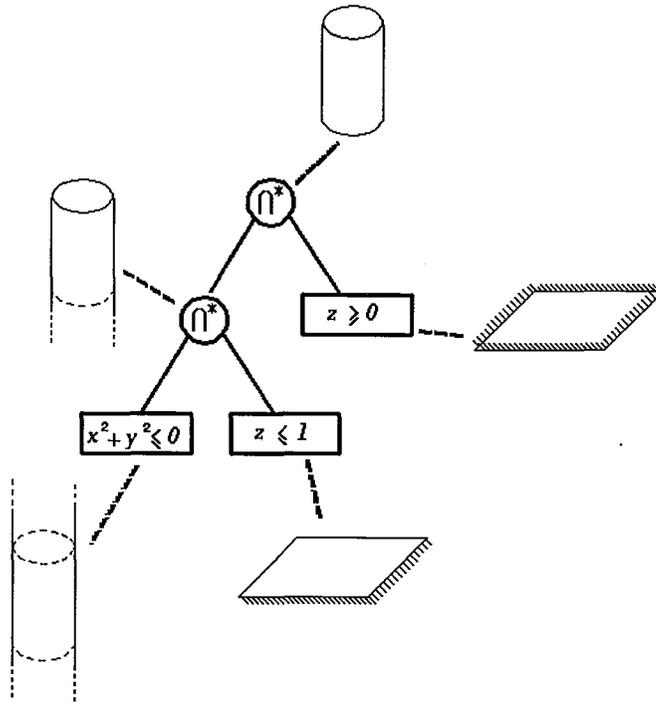


Figura II.2

Cilindro expresso como uma árvore CSG de semiespaços

- Outras implementações (veja MANTYLA [5]) preferem codificar explicitamente na estrutura de dados do sólido as informações de instanciação. A razão disso é permitir uma representação mais compacta de sólidos compostos de várias submontagens. Considere por exemplo, um sólido CSG que requer várias cópias de uma submontagem instanciadas de maneiras diferentes. Usando o esquema descrito anteriormente, seria necessário gerar várias cópias da sub-árvore, onde em cada uma as primitivas são instanciadas de maneira diferente. Um esquema alternativo seria então utilizar nós intermediários contendo apenas informações de instanciação, cada um deles por sua vez apontando para uma única cópia da sub-árvore. Tem-se então, não mais uma árvore binária, mas sim um grafo acíclico

direcionado (*DAG - Directed Acyclic Graph*). Um arranjo deste tipo é ilustrado na figura II.3.

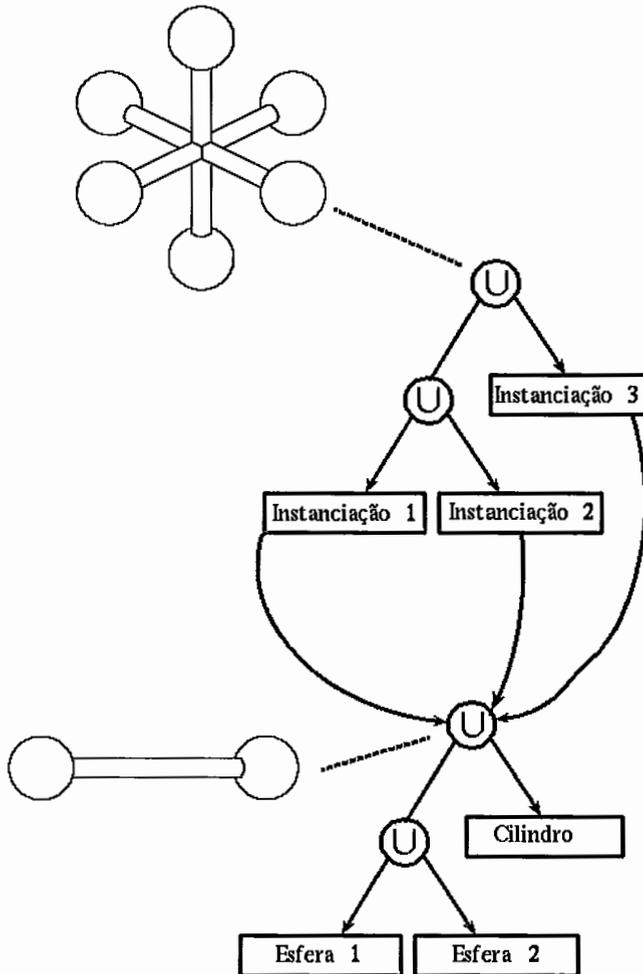


Figura II.3

Representação CSG através de um Grafo Acíclico Direcionado

2.4. Operações regularizadas de conjunto

Sólidos representados por uma árvore CSG são dados por um conjunto de pontos que atende a um sistema de n inequações relacionados por $n-1$ operadores de conjunto. Entretanto, o conjunto assim definido pode não atender ao pré-requisito que estabelecemos anteriormente, isto é, o conjunto pode não ser regular. Veja um exemplo desta situação na figura II.4, onde se faz a interseção de dois cilindros que se tocam.

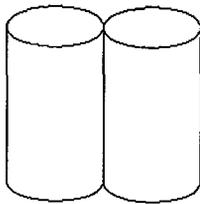


Figura II.4

Dois sólidos primitivos que se tocam

O efeito de termos conjuntos não regulares como resultado de uma representação CSG é indesejável não apenas por razões de elegância teórica. Considere uma aplicação onde operações de conjunto são usadas para testar a validade de uma montagem fazendo interseções entre todos os pares de componentes e verificando se o resultado é um conjunto vazio. Em casos assim é preferível considerar que a interseção de objetos que apenas se tocam é nula.

Para contornar essa dificuldade, a representação CSG se vale mais uma vez de conceitos derivados da Topologia para definir o que chamamos de operações regularizadas de conjunto. O conjunto de pontos resultante de uma operação regularizada é dado pelo fecho do interior do conjunto obtido pela operação correspondente não regularizada. Assim, os operadores regularizados \cap^* , \cup^* e \setminus^* são definidos por:

$$\begin{aligned} A \cap^* B &= \text{fecho}(\text{interior}(A \cap B)) \\ A \cup^* B &= \text{fecho}(\text{interior}(A \cup B)) \\ A \setminus^* B &= \text{fecho}(\text{interior}(A \setminus B)) \end{aligned} \tag{II.3}$$

3. Propriedades dos modelos CSG

Para melhor compreender a discussão que se segue, é conveniente distinguir alguns termos (uma abordagem mais rigorosa desses conceitos pode ser obtida em REQUICHA [6]). São eles:

- *Espaço de Modelagem*: é composto pelos sólidos (conjuntos de pontos) que nos interessa modelar. Em nossa discussão, o espaço de modelagem é dado por subconjuntos de E^3 que atendem à condição de regularidade, isto é, (*r-sets*).
- *Domínio do Espaço de Modelagem*: é o subconjunto do espaço de modelagem que contém os sólidos para os quais existe uma representação válida.
- *Esquema de Representação*: é uma maneira pela qual se constrói uma estrutura de símbolos (*representação*) para cada *r-set* pertencente ao domínio do espaço de modelagem. É também chamado de *modelo de representação*.
- *Representação Sintaticamente Válida*: É um conjunto de símbolos que foi construído segundo as normas preconizadas pelo esquema de representação.

Dentro desse contexto, podemos analisar algumas propriedades da classe de esquemas de representação que atendem pelo nome de CSG:

- *Validade*

CSG é um esquema válido, na medida que todas as representações sintaticamente corretas, isto é, todas as árvores CSG construídas conforme as regras que discutimos acima, estão efetivamente associadas a algum sólido. Esta característica é muito importante quando se considera o processo de modelagem; ela permite que o sistema modelador aceite ou rejeite uma representação apresentada pelo usuário baseado unicamente em sua validade sintática.

- *Unicidade*

Unicidade é a propriedade segundo a qual cada sólido admite uma única

representação. As árvores CSG, segundo essa noção, não são únicas, uma vez que uma infinidade de árvores CSG podem ser criadas para representar um mesmo sólido. Em particular, o objeto nulo, isto é, o espaço vazio possui uma série de árvores CSG que o representam. O problema de detectar quando uma árvore CSG representa o objeto nulo é, como veremos no capítulo IV, de fundamental importância em algoritmos de visualização baseados na técnica de subdivisão.

- *Não-ambiguidade*

Os modelos CSG não são ambíguos. Isto equivale a dizer que toda árvore CSG representa um único sólido. Esta propriedade é claramente um requisito praticamente indispensável em qualquer esquema para representação de sólidos.

- *Poder de expressão*

O poder de expressão de um esquema de representação está relacionado com a variedade de objetos que podem ser modelados. Em CSG, este fator está diretamente ligado à variedade de sólidos primitivos disponíveis. Desta forma, ao se avaliar uma determinada implementação de um modelador CSG é importante considerar não apenas de quais sólidos ela dispõe mas também a praticidade de se incluir novos sólidos primitivos. Um expediente que também pode ser usado para ampliar o poder de expressão de uma representação CSG é usar operadores de instanciação mais poderosos em adição aos tradicionais operadores de translação, rotação e escala, como por exemplo varredura ou *sweep* (veja MANTYLA [5]).

- *Concisão*

Representações segundo modelos CSG puros são de maneira geral concisas, isto é, requerem um dispêndio bastante modesto de memória. Entretanto, em muitos modeladores CSG esta concisão é sacrificada ao se agregar às árvores CSG uma série de informações que visam facilitar a execução de determinadas operações gráficas. Por exemplo, há modeladores híbridos (GMSOLID, PADL-1 - veja em REQUICHA [6]) onde as folhas das árvores CSG contêm informações sobre as fronteiras dos sólidos primitivos.

Os modelos CSG são baseados em um arcabouço matemático rigoroso e consistente e estão disponíveis hoje em dia toda uma família de algoritmos bem conhecidos e aplicáveis a uma série de problemas. Por causa disso, muitos modeladores CSG foram desenvolvidos e estão em uso.

Como nota bibliográfica final, é justo acrescentar que a teoria sobre modelagem CSG em termos de conceitos matemáticos rigorosos, foi desenvolvida pelos pesquisadores do projeto PADL da Universidade de Rochester (REQUICHA e VOELCKER [7], [8] e [9]). Esse projeto resultou na construção sucessiva de dois modeladores: o PADL-1 e o PADL-2, que ficaram muito conhecidos tanto na indústria quanto em meios acadêmicos. Eventualmente, eles serviram de base para outros modeladores comerciais, como por exemplo o GMSOLID, desenvolvido pela General Motors.

Capítulo III

Visualização de modelos CSG

1. Introdução

O sistema CSG de representação de sólidos é basicamente uma descrição de um processo de construção; ele não fornece informações explícitas sobre, por exemplo, a forma do objeto sendo representado. Por conseguinte, a geração de imagens correspondentes a um modelo sólido representado em CSG é não trivial e para resolvê-lo, uma série de técnicas foram desenvolvidas ao longo do tempo.

O propósito deste capítulo é dar um panorama geral sobre as técnicas mais importantes para visualização de modelos CSG. O material apresentado aqui foi em grande parte extraído da referência BRONSVOORT [10], e o leitor é encorajado a consultá-lo para maiores detalhes e referências adicionais.

Podemos dividir as diversas abordagens do problema em duas classes:

- (1) *Métodos baseados em avaliação de fronteiras*: são os que procuram se valer de algoritmos de visualização aplicáveis a modelos de fronteiras (*B-reps*). Para tanto é preciso aplicar um algoritmo de conversão de representações CSG para seus *B-reps* correspondentes - é o processo conhecido como "avaliação de fronteiras" ou *boundary evaluation*.
- (2) *Métodos de visualização direta*: são aqueles que buscam obter uma imagem de um sólido CSG utilizando tão somente informações contidas na própria representação.

2. Avaliação de fronteiras

O problema de conversão de um objeto representado em CSG para uma representação *B-Rep* correspondente, é conhecido como Avaliação de Fronteiras ou

Boundary Evaluation (alguns autores denominam *Boundary Merging* o problema de avaliação de operações booleanas entre sólidos dados por *B-Reps*, entretanto, usaremos o termo "Avaliação de Fronteiras" para designar ambos). Um trabalho onde este problema é abordado de maneira teoricamente consistente foi publicado por REQUICHA e VOELCKER [11].

Os algoritmos propostos por REQUICHA e VOELCKER se baseiam num procedimento geral de "gerar e testar". Primeiramente, todo o conjunto de possíveis faces e arestas do sólido são geradas. Depois, esses elementos são testados contra o modelo CSG para determinar quais delas pertencem efetivamente ao sólido em questão.

As faces e arestas candidatas são geradas a partir dos sólidos primitivos que compõem o modelo CSG. Assumindo que cada um dos sólidos primitivos já possui suas faces e arestas calculadas, o algoritmo passa então a calcular as faces e arestas resultantes da interseção desses elementos. Esta é uma das fases mais computacionalmente dispendiosas do algoritmo: ela requer que cada face de uma primitiva seja comparada com cada outra face das demais primitivas. Este processo pode incorrer em erros numéricos, por exemplo, quando duas faces são quase coplanares.

O segundo passo requer que as faces e arestas candidatas sejam classificadas em relação ao sólido composto. Farão parte do modelo de fronteiras apenas aquelas arestas e faces situadas sobre a casca do objeto; elementos que forem avaliados como "no interior" ou "no exterior" do objeto são descartadas. Este processo é denominado Classificação de Pertinência a Conjunto (*Set Membership Classification*). Por exemplo, se considerarmos um sólido composto de duas primitivas A e B associadas por um operador de união, cada um dos elementos candidatos é classificado com respeito a A e a B separadamente; apenas aqueles pertencentes à fronteira de A e no exterior de B ou então aqueles na fronteira de B e exteriores a A são considerados na fronteira do sólido composto. Regras similares são aplicadas para os operadores de interseção e diferença e o procedimento é repetido recursivamente em todos os demais nós da árvore CSG.

A Classificação de Pertinência é um procedimento relativamente simples quando não existem fronteiras que se tocam, entretanto, quando isso acontece podem surgir ambiguidades. Esse tipo de problema pode ser resolvido adicionando-se informações de vizinhança às classificações, isto é, informações sobre de que lado o material da objeto se encontra.

2.1. Avaliação de fronteiras para objetos polidrais

REQUICHA e VOELCKER assumem em sua discussão um modelo onde as primitivas são limitadas por faces planas, porém ressalvam que o algoritmo é aplicável a primitivas com faces curvas desde que procedimentos adequados para o cálculo das curvas de interseção entre essas faces sejam desenvolvidos. Tais procedimentos são abordados por LEVIN [12] e [13], MILLER [14] e SARRAGA [15] para o caso de superfícies quádricas.

Dois artigos relativamente recentes, LAIDLAW, TRUMBORE e HUGHES [16] e MANTYLA [17] elaboram bastante sobre técnicas para avaliação de operações booleanas em modelos *B-Rep*. As idéias básicas são as que esboçamos anteriormente, mas esses trabalhos entram com considerável profundidade nos detalhes envolvidos, e abordam casos especiais tais como faces coplanares, arestas colineares e vértices coincidentes. Ambos podem servir como ponto de partida para um trabalho de implementação.

Infelizmente, esses trabalhos relatam que problemas de precisão ainda são as maiores dificuldades. A origem desses problemas reside na limitação das representações de números em ponto flutuante, o que impede uma avaliação cem por cento segura de questões tais como "duas faces se interceptam?" ou "tal vértice está sobre uma face?". SEGAL e SEQUIN [18] propõem por exemplo um artifício baseado em tolerâncias e em última instância o uso de interação onde o usuário é consultado para resolver a questão.

Um outro aspecto importante é a ineficiência decorrente da fase de geração de possíveis faces e arestas do sólido onde cada par de objetos devem ser analisados. Requicha e Voelcker já se ocupam desse assunto em seu artigo prescrevendo o emprego de caixas envolventes e subdivisão espacial (ambos os conceitos serão vistos nos capítulos que se seguem).

NAVAZO, FONTDECABA e BRUNET [19] sugerem uma aproximação mais formal e elaborada do problema através uma representação híbrida. A idéia principal resume-se em montar uma estrutura hierárquica de subdivisão do objeto à semelhança de uma octree. Isto permite localizar as regiões onde pares de faces precisam ser testadas.

2.2. Avaliação de fronteiras de sólidos delimitados por superfícies quádricas

As idéias básicas utilizadas em métodos de avaliação de fronteiras para objetos delimitados por superfícies quádricas são as mesmas que para objetos poliedrais. Por exemplo, MILLER [20] também baseia seu algoritmo no mesmo paradigma de "gerar e testar". A maior diferença quando consideramos superfícies quádricas relaciona-se com a fase de geração de arestas candidatas. O cálculo de curvas de interseção entre superfícies quádricas são ainda mais difíceis de serem calculadas do que retas de interseção entre planos. MILLER [14] e [20] distinguem entre métodos algébricos e geométricos para determinar curvas de interseção. Os métodos algébricos baseiam-se em representações e algoritmos gerais para superfícies e curvas dadas por polinômios de grau 2 e 4 respectivamente. Isto enseja implementações relativamente simples mas requer o emprego de testes numericamente muito sensíveis entre grandezas sem uma importância geométrica direta. Em contraposição, os métodos geométricos abordam separadamente as curvas resultantes da interseção entre as diferentes combinações de pares de superfícies quádricas, entretanto empregam testes apenas entre grandezas geometricamente relevantes, implicando com isso numa estabilidade numérica muito mais fácil de controlar. Miller dá uma descrição bastante detalhada dos métodos geométricos aplicados às combinações de superfícies quádricas mais comuns.

Em termos de complexidade, os algoritmos de avaliação de fronteiras baseados na descrição que fizemos anteriormente são tidos como de $O(n^4)$, onde n é o número de semiespaços na árvore CSG (veja em TILOVE [21]). Na prática, entretanto, muitos pesquisadores desenvolveram técnicas que diminuem a necessidade de testar todos os pares de faces, o que constitui a fase mais computacionalmente dispendiosa do processo. A complexidade inerente aos algoritmos de avaliação de fronteira os torna pouco atraentes quando se deseja apenas obter uma imagem do sólido CSG. Entretanto, há ocasiões onde a avaliação de fronteiras é vantajosa, como por exemplo em modeladores híbridos. Para fins de exibição, uma conversão integral pode ser substituída com vantagens por um processo de conversão parcial. É sabido por exemplo que um processo de subdivisão relacionado com modelos Octree pode reduzir a complexidade do problema de avaliação de árvores CSG (WOODWARK [22], [23] e [24]).

3. Algoritmos de visualização direta

Uma abordagem alternativa é o uso dos chamados algoritmos de visualização direta. Nesses algoritmos, evita-se as dificuldades do processo de avaliação de fronteiras ao se empregar procedimentos de cálculo apenas em locais do objeto onde haverá alguma contribuição para a imagem sendo montada. Muitas técnicas derivadas dos algoritmos de visualização direta podem ser aplicadas inclusive para a avaliação de propriedades integrais, tais como volume, área, centro de massa, etc.

Como vantagens desse tipo de abordagem podemos enumerar:

- Os algoritmos de visualização direta normalmente são montados em cima de algoritmos de ocultamento de linhas e superfícies, evitando o custo computacional relacionado com avaliações de porções ocultas do sólido.
- É mais eficiente que utilizar um algoritmo de avaliação de fronteiras seguido de algoritmos de ocultamento de linhas e superfícies.
- Sendo geralmente algoritmos que trabalham no espaço da imagem, é possível utilizá-los com um maior ou menor grau de definição a um custo proporcional. Assim, quando soluções aproximadas são aceitáveis, pode-se dosar o esforço computacional necessário para obtê-las.
- Dificuldades de precisão numérica são muito mais facilmente contornáveis.

A principal desvantagem reside no fato que um modelo de fronteiras completo não é nunca computado. Quando se considera que a visualização de modelos de fronteiras conta com algoritmos muito bem conhecidos e de maneira geral muito rápidos, a adoção de algoritmos de visualização direta pode se mostrar uma escolha inadequada. Um exemplo dessa situação é quando, uma vez modelado o sólido, são desejadas diversas imagens tomadas segundo diferentes sistemas de projeção. Ora, se dispomos do modelo de fronteiras correspondente, a geração de imagens é trivial e muito eficiente; por outro lado, usando algoritmos de visualização direta seria necessário efetuarlos repetidas vezes - um processo muitas vezes de lentidão inaceitável. Em resumo, nos parece que durante a fase de modelagem de sólidos, os algoritmos de visualização direta costumam ser bastante vantajosos, porém essa vantagem diminui na proporção em que várias imagens de um mesmo sólido precisam ser produzidas; por exemplo, mesmo durante a fase de modelagem é comum se querer obter diversas imagens de um sólido segundo diferentes vistas.

Diferentemente do que acontece no caso de algoritmos de avaliação de fronteiras onde o paradigma de "gerar e testar" é universal, os algoritmos de visualização direta costumam empregar diversas abordagens.

3.1. Visualização direta de modelos poliedrais

Considerando modelos CSG compostos de primitivas de faces planas, podemos citar em primeiro lugar o algoritmo de visualização proposto por ATHERTON [25]. Atherton adaptou o conhecido algoritmo de ocultamento de linhas e superfícies baseado em linhas de rastreo (*scanlines*). O algoritmo requer que se disponha de *B-Rep's* de cada uma das primitivas empregadas na árvore CSG. Seguindo o mesmo procedimento geral dos algoritmos baseados em linhas de rastreo, o modelo é submetido a uma fase de ordenação onde se determina para cada face de cada primitiva quais as linhas de rastreo de interesse, isto é, os valores máximo e mínimo das coordenadas y ocupados pela projeção da face (assume-se que as linhas de rastreo correspondem a valores $y = constante$). Para cada uma das linhas de rastreo é gerada uma lista de segmentos de reta correspondente à interseção de cada uma das faces ativas naquela linha de rastreo com o plano $y = constante$ associado. Neste ponto, o algoritmo se desvia do processo tradicional na medida que, além da tarefa usual de determinar para uma determinada região da linha de rastreo qual o segmento mais próximo do observador, é necessário ainda assegurar a validade desse segmento com respeito ao modelo CSG, isto é, se ele faz parte da fronteira do sólido. Isto equivale a fazer uma avaliação de fronteiras restrita a uma pequena região do espaço. Se o segmento for inválido é testado o segundo segmento mais próximo do observador e assim por diante. O procedimento de teste empregado por Atherton envolve o paradigma de lançamento de raios (veja adiante), que por sua vez é calcado numa pesquisa feita de baixo para cima na árvore CSG. BRONSVOORT [26] demonstra uma técnica alternativa onde a pesquisa na árvore é feita de cima para baixo (isto é, das folhas para a raiz), conseguindo com isso uma grande redução no tempo de avaliação.

Ainda no âmbito de modelos CSG de objetos poliedrais, podemos citar o trabalho de JANSEN [27], onde o autor apresenta um algoritmo de visualização direta inspirado no paradigma de ocultamento de linhas e superfícies por listas de prioridade. BRONSVOORT [28] e VERROUST [29] apresentam uma abordagem do problema que consiste em dividir o espaço da imagem em áreas onde apenas uma face é visível.

De particular interesse para nós é o trabalho em que KOISTINEN, TAMMINEN e SAMET [30] sugerem o uso de estruturas de divisão espacial para resolver o problema. A idéia geral das técnicas de divisão espacial é o objeto do capítulo IV deste trabalho e uma explicação detalhada do algoritmo é dada no capítulo VI, assim como uma nova variante que melhora sensivelmente seu desempenho.

Apesar dessas diversas abordagens do problema, até o momento o algoritmo baseado em linhas de rastreo introduzido por Atherton bem como suas variantes permanece ainda como líder incontestado em termos de eficiência.

3.2. Visualização direta de modelos delimitados por quádricas

Surpreendentemente, um algoritmo de visualização direta de sólidos CSG delimitados por superfícies quádricas foi introduzido antes daqueles restritos a objetos poliedrais. Trata-se de uma adaptação da idéia do algoritmo de disparo de raios (*ray-casting*) introduzido por NAGEL e GOLDSTEIN [31]. Em sua concepção original, o algoritmo de lançamento de raios avalia cada ponto (pixel) do plano de projeção definindo uma reta (raio) entre esse ponto e a posição do observador. Para cada um dos objetos que compõem a cena são calculados os pontos de interseção com o raio, sendo que a cor do pixel é associada ao objeto que resultou no ponto de interseção mais próximo ao observador. ROTH [32] estendeu o conceito de lançamento de raios a fim de tratar objetos modelados em CSG. Nesse caso, a classificação raio \times sólido envolve o cálculo de operações de conjunto entre os intervalos onde o raio atravessa cada primitiva. Assim, o problema de computar união, interseção e diferença entre volumes é reduzido a fazer essas operações em apenas uma dimensão. Usando as técnicas de disparo de raios pode-se por exemplo exibir modelos CSG compostos de semiespaços delimitados por quádricas tanto em desenhos usando linhas (*wireframes*) como em desenhos sombreados (*shaded displays*). Tem ainda como vantagem adicional prestar-se à avaliação de outras características integrais do modelo tais como massa e volume. O método de lançamento de raios pode ser caracterizado como de "força bruta", uma vez que cada ponto da imagem é avaliado independentemente. Por isso mesmo, sua principal desvantagem reside no baixo desempenho. Técnicas auxiliares para minorar essa deficiência são abordadas por Roth em seu artigo. No capítulo V deste trabalho apresentamos uma descrição detalhada desses algoritmos bem como de algumas técnicas suplementares envolvendo basicamente o paradigma de

subdivisão espacial.

Outras abordagens ainda foram estudadas, sendo que todas elas conseguem um desempenho superior embora com uma qualidade de imagem aquém daquela conseguida através do uso puro e simples de disparo de raios.

PUEYO e MENDOZA [33], por exemplo, sugerem um algoritmo baseado em linhas de rastreio capaz de tratar objetos delimitados por superfícies quádricas. ROSSIGNAC e REQUICHA [34] introduziram uma abordagem derivada da técnica para ocultamento de faces não visíveis empregada em visualização de modelos de fronteiras - o *z-Buffer*. A idéia principal por trás desta abordagem é agregar algumas informações adicionais a cada elemento do *z-Buffer* com o objetivo de tornar possível o tratamento de operações de conjunto pixel a pixel. Trata-se de um tipo de abordagem altamente promissora uma vez que pode ser em grande parte implementada por hardware como o demonstra GOLDFEATHER [35] que, com o auxílio de um processador VLSI dedicado, conseguiu velocidades de visualização quase que em tempo real para um pequeno número de primitivas e inclusive com um dispêndio relativamente modesto de memória.

4. Conclusões

Avaliação de fronteiras e visualização direta podem ser vistas como técnicas complementares. Há ocasiões em que avaliação de fronteiras é indispensável - por exemplo em modeladores híbridos, onde *B-rep* é o principal esquema de representação, e formulações envolvendo operações de conjunto fazem parte do repertório de modelagem.

Um tópico interessante é delimitar quais os problemas que podem ser resolvidos sem recorrer a avaliação de fronteiras. Por exemplo, é possível realizar uma análise por elementos finitos?

Podemos esperar que grande parte do esforço de pesquisa em avaliação de fronteiras se concentre em resolver os problemas numéricos que ainda persistem e em buscar algoritmos mais eficientes. Outro item que deverá merecer atenção é o tratamento de modelos que admitem sólidos delimitados por quádricas ou outras classes de superfícies.

Durante a fase de modelagem, visualização direta apresenta uma boa relação custo \times desempenho, mesmo que seja inevitável o emprego a posteriori de um processo de avaliação de fronteiras. Visto isso, nos parece que o principal problema a ser atacado é a eficiência, sendo particularmente promissoras as abordagens que empregam arquiteturas especializadas.

Capítulo IV

Subdivisão Espacial

Neste capítulo faremos uma exposição de como paradigma de subdivisão espacial pode ser aproveitado no âmbito de representações CSG. O material exposto a seguir foi baseado nas referências pioneiras sobre subdivisão espacial apresentadas por TILOVE ([21] e [36]) e WOODWARK ([22] e [23]).

1. O conceito de subdivisão espacial

Em geral, os algoritmos usados em modelagem de sólidos lidam com relações entre entidades geométricas que representam algum aspecto da forma dos objetos. A complexidade inerente a esses algoritmos pode ser atribuída ao fato que, enquanto o número de entidades geométricas cresce linearmente com a complexidade do objeto, o número de relações que precisam ser computadas cresce em proporção quadrática, ou em alguns casos, segundo potências ainda maiores.

Via de regra, todos esses algoritmos atacam o problema tentando discernir rapidamente quais as relações que efetivamente precisam ser avaliadas e descartar aquelas que não contribuem para o objetivo final que se deseja alcançar.

O conceito de subdivisão espacial insere-se nesse contexto. A idéia, intuitivamente simples, é que entidades geométricas dispostas em posições próximas dentro do espaço têm uma maior probabilidade de gerar interações entre si do que outras posicionadas em pontos distantes.

1.1. O problema de detecção de objeto nulo

O esquema de representação CSG é não-único, isto vale dizer que um mesmo sólido pode ser representado por diferentes árvores CSG. Conseqüentemente, um problema que surge frequentemente é o de decidir se dois modelos CSG representam o

mesmo sólido. Esta pergunta pode ser respondida para dois modelos A e B se soubermos a resposta para a pergunta " $(A \setminus B) \cup (B \setminus A)$ é um objeto nulo?".

Aqui vemos uma circunstância onde uma conversão de CSG para outro modelo de representação poderia ser a chave para a solução do problema. Por exemplo, sendo Representação por Fronteiras um esquema de representação único, um objeto nulo em B-rep possui uma representação também nula.

Em TILOVE [36] o autor adota uma abordagem baseada em dois conceitos principais: o de redundância de primitivas e o de localização de árvores CSG.

1.2. Redundância de primitivas

Uma primitiva de uma árvore CSG (isto é, um nó folha) é dito redundante quando ele não contribui para a forma do sólido representado. Em outras palavras, uma primitiva redundante não agrega nem retira material ao conjunto de pontos que compõem o sólido.

Pode-se reconhecer uma primitiva redundante pelo fato que ela pode ser substituída na árvore CSG pelo conjunto vazio (\emptyset) ou pelo conjunto universo (W) sem alterar o sólido. No primeiro caso diz-se que a primitiva é \emptyset -redundante, e no segundo, W -redundante.

Tilove demonstra em seu artigo que numa representação CSG do objeto nulo todas as primitivas positivas são \emptyset -redundantes, e todas as primitivas negativas são W -redundantes. O "sinal" (positivo ou negativo) de uma primitiva, é dado pelo número de vezes que esta é subtraída na composição do sólido. Ele pode ser inferido contando-se o número de vezes que se passa por um nó interno do tipo diferença (\setminus) tomando-se o ramo da esquerda; se esse número é par, a primitiva é positiva, caso contrário, é negativa. Na figura IV.1 as primitivas "Bloco 1" e "Bloco 2" são positivas, enquanto que a primitiva "Cilindro 1" é negativa.

Na verdade, para detectar um objeto nulo, basta efetuar um teste de redundância sobre as primitivas positivas. Esse teste é simplificado pela observação que se a interseção entre o sólido S e uma de suas primitivas P_i é nula, então P_i é \emptyset -redundante. Observe que no âmbito deste teste, estamos preocupados apenas com a avaliação de S na região do espaço dada pelo *interior* de P_i , ou seja, estamos nos referindo à localização de S com relação a P_i .

1.3. Localização de uma árvore CSG

Seja um sólido S , representado por uma árvore CSG A , composta das primitivas $P_1, P_2 \dots P_n$. Define-se como localização de A com relação a um conjunto de pontos R , uma árvore A' , composta das primitivas $P'_1, P'_2 \dots P'_m$, tal que

$$\begin{aligned} S \cap^* R &\equiv S' \cap^* R \\ e \\ \{P_1, P_2, \dots, P_n\} &\supset \{P'_1, P'_2, P'_m\} \end{aligned} \tag{IV.1}$$

Por exemplo, na figura IV.1 a primitiva "Cilindro 1" pode ser substituída pelo conjunto vazio na região delimitada pelas linhas tracejadas.

A localização de uma árvore em relação a uma região R do espaço é potencialmente mais simples que a árvore original, uma vez que primitivas que não interceptam a região R podem ser substituídas por \emptyset , isto é, são \emptyset -redundantes na árvore localizada. Similarmente, primitivas que contêm a região R podem substituídas pelo conjunto universo W .

1.4. Simplificação de uma árvore CSG

Uma vez que detectemos uma ou mais primitivas redundantes em uma árvore CSG A , podemos aplicar um procedimento de simplificação para obter uma árvore A' com menos nós e que é equivalente à árvore original. Para isso, aplica-se recursivamente as propriedades das operações entre conjuntos conforme a tabela IV.1.

O algoritmo de detecção de objetos nulos proposto por Tilove pode ser então sumarizado da seguinte maneira:

- (1) Escolhe-se uma primitiva positiva P_i da árvore A correspondente ao sólido que se quer testar.
- (2) Monta-se uma localização A' da árvore A com respeito à região delimitada por P_i .
- (3) P_i é declarada \emptyset -redundante se $A' \cap^* P_i = \emptyset$. Neste caso, a árvore A é simplificada, substituindo-se P_i por \emptyset . Caso contrário, o algoritmo termina pois o

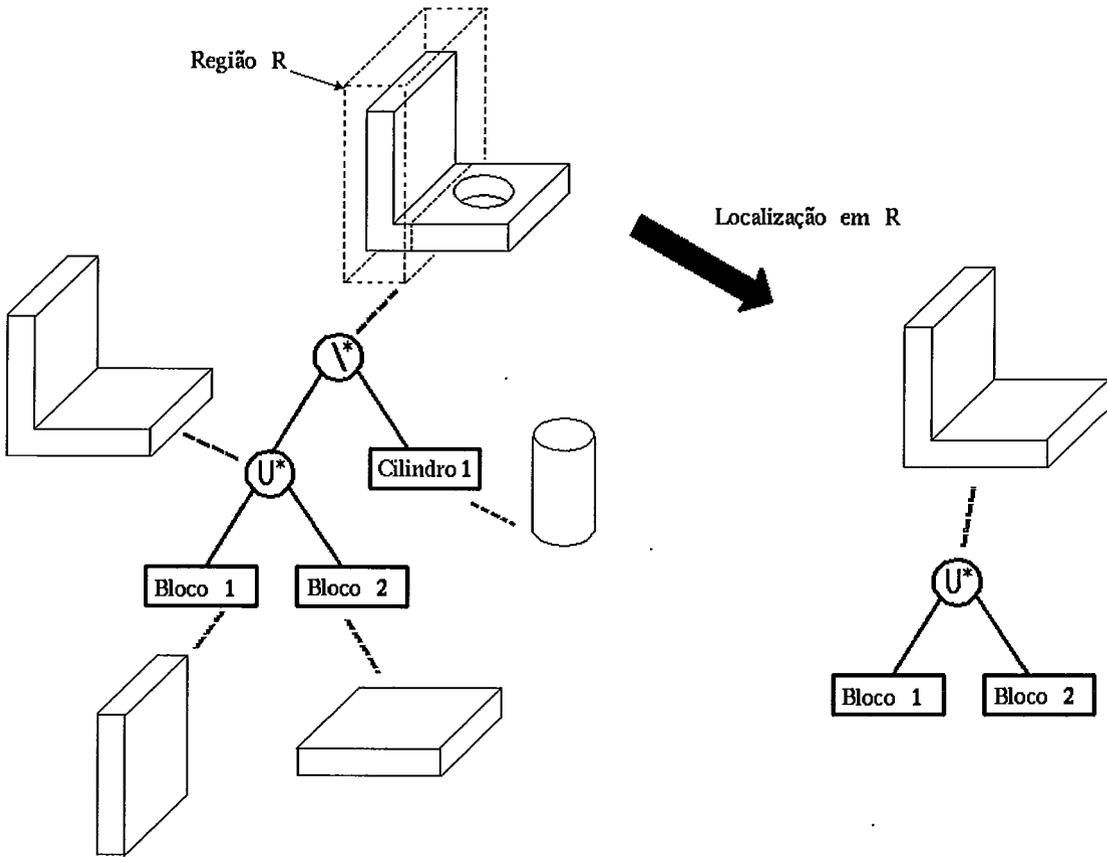


Figura IV.1
Localização de uma árvore CSG

objeto é não-nulo.

(4) Os passos (1), (2) e (3) são repetidos para todas as primitivas positivas da árvore.

O algoritmo de Tilove tem como principal mérito restringir o número de comparações geométricas entre pares de primitivas a um mínimo, uma vez que no passo (3) são consideradas apenas as primitivas da árvore localizada A' . Mesmo assim, a condição a ser avaliada no passo (3) requer testes que podem se mostrar muito custosos, uma vez que corresponde de maneira geral a um processo de avaliação de fronteiras.

A	A'
$P \cup^* \emptyset$	P
$P \cap^* \emptyset$	\emptyset
$P \setminus^* \emptyset$	P
$\emptyset^* \setminus P$	\emptyset
$P \cup^* W$	W
$P \cap^* W$	P
$P \setminus^* W$	\emptyset
$W \setminus^* P$	$\neg P$
$P \text{ op } Q$	$P \text{ op } Q$

Tabela IV.1

Regras para simplificação de árvores CSG

Em outras aplicações, entretanto, os conceitos de redundância de primitivas e de localização podem ser usados de maneira intensiva de forma a restringir a um mínimo a necessidade de efetuar cálculos no estilo "força bruta".

1.5. O processo de subdivisão recursiva

O teste de redundância não é de maneira geral simples de ser feito, com exceção dos casos onde a localização de uma árvore corresponde a \emptyset ou W . Podemos intuir que a probabilidade de isso ocorrer para uma dada região R do espaço é tanto maior quanto menor forem as dimensões de R . Com efeito, se analisarmos uma região correspondente a um único ponto x do universo onde o objeto está mergulhado veremos que a localização de uma primitiva P_i em relação a x é não trivial apenas na fronteira de P_i , sendo \emptyset ou W em todos os demais pontos. É desta observação que nasce o conceito de subdivisão espacial.

Um teste *suficiente* (embora não *necessário*) para garantir que uma primitiva P_i é redundante numa região R é dado por

$$P_i \cap R \equiv \begin{cases} \emptyset \\ \text{ou} \\ W \end{cases} \quad (\text{IV.2})$$

É importante notar que a inversa não é verdadeira em geral, isto é, se a eq. IV.2 não vale, não necessariamente a primitiva é não redundante em R . O que se pode garantir é que, se P_i é a única primitiva de A , então

Se $P_i \cap R \equiv \emptyset \Rightarrow P_i$ está "fora" de R

Se $P_i \cap R \equiv W \Rightarrow P_i$ contém R

De outra maneira, a fronteira de P_i passa por R

A tarefa de determinar se uma primitiva P_i intercepta ou não uma região R é tanto mais difícil e onerosa quanto maior é a "complexidade geométrica" de ambos. Por isto, é costume aplicar o processo de subdivisão de tal forma que as regiões R em que o espaço é subdividido possuam forma regular, tal como cubos ou paralelepípedos. Quanto às primitivas, é costume escolher um conjunto reduzido de sólidos simples delimitados por semiespacos planos como cubos e poliedros em geral, ou quádricas como esferas, cilindros, cones, etc.

Um processo de subdivisão recursiva consiste em dividir o universo em porções R cada vez menores e, a cada passo, eliminar da árvore localizada A' correspondente a R todas as primitivas P_i que foram reconhecidas como redundantes através do teste enunciado pela eq. IV.2 acima e usando o processo de simplificação conforme explicado anteriormente. O critério usado para cessar a subdivisão é dependente do objetivo final do algoritmo, como veremos a seguir.

2. Aplicações de subdivisão espacial

O algoritmo de subdivisão recursiva pode ser usado em muitas aplicações, dentre as quais destacamos:

- Detecção de objeto nulo;
- Avaliação de fronteiras;
- Conversão para modelos de representação baseados em subdivisão do espaço;

- Visualização direta de modelos CSG.

Os dois últimos itens são de maior importância para nós, e serão detalhados a seguir.

2.1. Transformação de CSG para um modelo de representação baseado em subdivisão do espaço.

É o caso onde se busca, por exemplo, um método para avaliação de octrees (veja em JACKINS e TANIMOTO [37] ou MEAGHER [38]).

A representação octree usa uma subdivisão recursiva do espaço de interesse em oito octantes que são arranjados em uma árvore de ordem 8. A cada octante é dado o nome de *voxel*. A figura IV.2 ilustra um sólido simples e sua representação em octree.

Cada nó de uma octree possui ordinariamente um código de classificação, cujos três valores possíveis, a saber, *cheio*, *vazio* e *misto*, indicam respectivamente se a porção correspondente do universo está preenchida pelo sólido, está vazia, ou está parcialmente cheia. Neste último caso, o nó contém 8 ponteiros para as descrições de seus nós-filho que são definidos semelhantemente.

Uma octree é uma representação aproximativa por excelência, uma vez que uma representação exata de um objeto geral requereria uma octree de altura infinita. O grau de aproximação é portanto dado pela altura máxima a que se permite a octree chegar e que é estabelecido a priori.

O processo de subdivisão espacial de CSG pode portanto ser aplicado diretamente para avaliação de octrees estabelecendo como limite para o processo justamente o conjunto de restrições que definem a representação octree desejada. Ou seja, o processo de subdivisão espacial da árvore CSG pára quando:

- A localização A' da árvore A num voxel resulta em \emptyset . Neste caso o nó da octree é marcado com o código *vazio*.
- A localização A' resulta em W . Neste caso o nó da octree é marcado com o código *cheio*.
- A localização A' é ainda não-trivial, mas a octree já chegou à altura máxima pré-estabelecida. Neste caso a estratégia é ditada pelo critério do implementador,

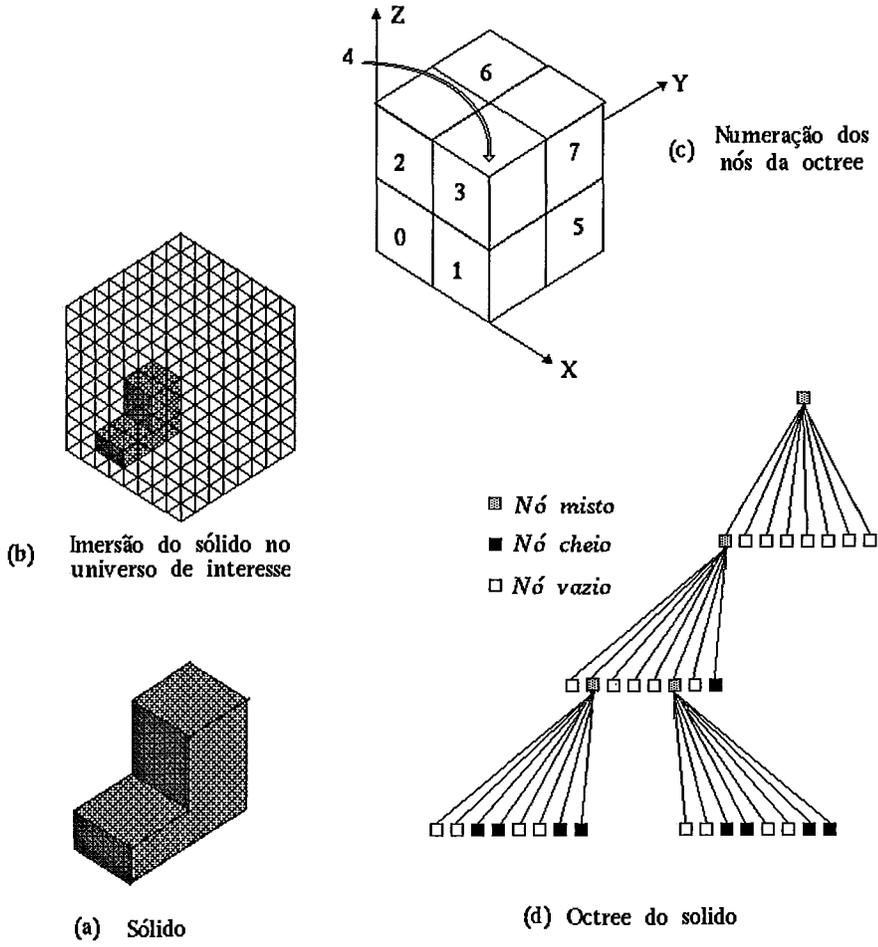


Figura IV.2
Representação octree de um sólido

sendo comum marcar o nó da octree como *cheio*, embora nesse caso não exista garantia que alguma porção do sólido esteja incluída no octante.

Embora tenhamos delineado a aplicação do processo de subdivisão espacial à conversão de CSG para octrees, ele se presta igualmente para outros tipos de representação de sólidos baseado em divisão hierárquica do espaço de interesse. SAMET e TAMMINEN [39], por exemplo aplicam esta técnica à geração de *bintrees*, um esquema aparentado ao de octrees, diferindo no fato que o espaço é sempre dividido em duas porções iguais, ao invés de oito.

2.2. Visualização direta de modelos CSG

Neste caso, o que se busca é uma maneira de reduzir a complexidade ligada ao problema de computar a imagem de uma projeção de um sólido. Em algoritmos deste tipo, o implementador primeiramente estipula um determinado número de casos simples para os quais ele dispõe de algoritmos de visualização direta. A idéia então é usar subdivisão espacial até que numa dada porção do espaço a localização do sólido se enquadre num desses casos especiais.

Normalmente, os algoritmos de visualização direta que se valem do paradigma de decomposição espacial costumam montar o espaço de interesse dentro de um paralelepípedo com faces perpendiculares aos eixos coordenados. Os eixos são orientados segundo a regra da mão esquerda, com o eixo Y apontando para cima, o eixo X para a esquerda e o eixo Z perpendicular ao plano de projeção afastando-se do observador. O plano de projeção é posicionado sobre o plano $Z = 0$ (veja a figura IV.3).

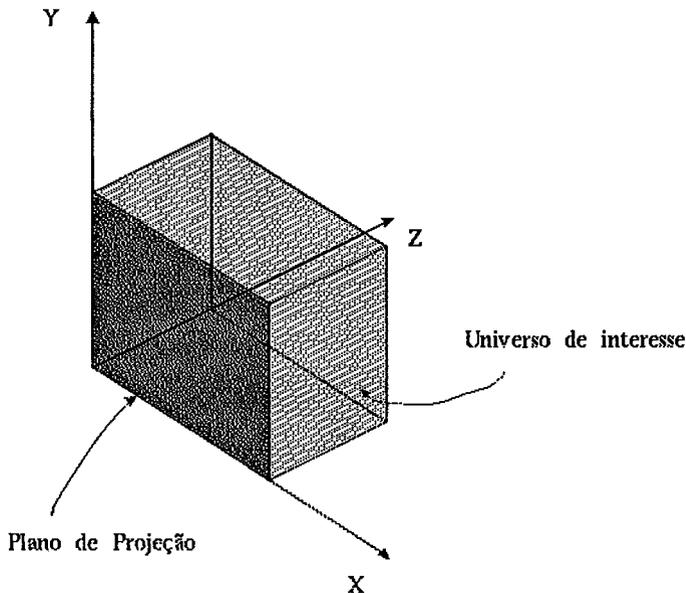


Figura IV.3

Sistema de coordenadas para visualização direta de modelos CSG

A subdivisão espacial do modelo CSG processa-se de tal forma que as regiões R correspondentes aos valores de Z mais próximos da tela sejam examinados antes daquelas mais afastadas. A complexidade do processo é reduzida pelo fato que, diferentemente do que vimos na conversão completa de CSG para octrees por exemplo, não é necessário examinar todos os voxels do universo. Uma vez que estamos preocupados em calcular uma *projeção* do sólido, após determinar que a projeção de um voxel traduz-se numa área a ser pintada na tela, nenhum voxel que se projete na mesma área X - Y mas com coordenada Z maior necessita ser avaliado. Assim, o problema de ocultamento de superfícies invisíveis é resolvido implicitamente pela própria ordem em que os voxels são visitados.

Para controlar quais porções da imagem do objeto já foram pintadas, normalmente usa-se uma estrutura de divisão espacial em duas dimensões análoga à estrutura hierárquica tridimensional sendo empregada para dividir o universo de interesse. Por exemplo, MORRIS e QUARENDON [40] que utilizam um processo de divisão do espaço em octantes (correspondente portanto à avaliação parcial de uma octree) empregam uma quadtree para manter a informação de projeção. Já KOISTINEN, TAMMINEN e SAMET [30], que dividem o espaço tridimensional usando uma bintree de ordem 3, empregam uma bintree de ordem 2 para controlar a projeção.

Merece ainda atenção no processo de subdivisão voltado para a visualização o caso em que o processo alcançou um voxel de tamanho mínimo (isto é, correspondente a um pixel da tela) mas a árvore localizada é ainda não trivial. Diferentemente do que ocorria no tipo de aplicação que vimos anteriormente, considerar simplesmente o voxel como cheio não resolve o problema, uma vez que a cor a ser empregada para a pintura do pixel depende de *qual* primitiva está localizada em primeiro plano. A solução para esta situação está em usar a estratégia de disparo de raios, isto é, é disparado um raio passando pelo centro do pixel do plano de projeção e pelo centro do voxel. Observe-se que neste caso, o custo computacional do disparo de raios é bastante reduzido uma vez que esta estratégia precisa ser empregada tão somente em áreas do plano de projeção correspondentes a arestas ou vértices. Releve-se também o fato que as árvores CSG sobre as quais os raios são disparados são localizações da árvore original e portanto significativamente menores do que esta.

3. Conclusões

Da discussão anterior podemos extrair algumas observações quanto à utilidade e à aplicabilidade do processo de subdivisão espacial de árvores CSG.

Em primeiro lugar, o processo explora a propriedade de similaridade espacial, isto é, em regiões próximas as árvores localizadas tendem a ser similares. É especialmente valiosa a capacidade que tem o processo de subdivisão de delimitar as regiões "vazias", "cheias" e de "fronteira" sem se valer de cálculos de interseção entre primitivas.

Em segundo lugar, o processo de subdivisão espacial é um método heurístico que pode ser enquadrado na classe de algoritmos que se valem da idéia de balanceamento. Isto vale dizer que a cada etapa de subdivisão é de se esperar que o problema se divida em várias partes cuja complexidade é menor que o problema original mas de ordem aproximadamente semelhante entre si.

Um outro ponto a ser considerado em aplicações de visualização é a conveniência ou não de usar subdivisão espacial de maneira exclusiva ou conjugada a outras técnicas. Nos parece que esta segunda alternativa possui um apelo mais forte. O uso exclusivo de subdivisão só se justifica até o momento em que seu custo computacional é inferior ao de outras técnicas. Como regra geral (empírica, é verdade), o processo de subdivisão pode ser empregado como ferramenta para reduzir o problema até o ponto em que este possa ser resolvido mais rapidamente por técnicas convencionais. Um dos objetivos deste trabalho é justamente explorar esse fato através do estudo de algumas variantes de métodos conhecidos de visualização onde subdivisão é empregada em maior ou menor grau; em outras palavras, busca-se aqui uma combinação equilibrada de subdivisão espacial com outras técnicas.

Capítulo V

Visualização através de disparo de raios

Neste capítulo abordaremos diversos aspectos da técnica de disparo de raios. Como orientação para o leitor, adiantamos abaixo a organização em seções deste texto:

- (1) *O método de disparo de raios*: estudo do modelo em que se baseia o método de disparo de raios bem como das estruturas de dados e algoritmos básicos relacionados.
- (2) *Aplicações*: como utilizar disparo de raios para resolver problemas ligados a representações CSG, com ênfase em aplicações de visualização.
- (3) *Otimização*: técnicas para melhorar o desempenho de algoritmos baseados em disparo de raios, com ênfase no uso do paradigma de subdivisão espacial.
- (4) *Resultados e conclusões*.

1. O método de disparo de raios

A técnica de disparo de raios constitui uma das principais ferramentas para a visualização de sólidos CSG. Ela pode ser usada por si só ou associada a outras técnicas. Neste último caso, o disparo de raios costuma ser empregado como critério de minerva quando não é mais desejável prosseguir com o método principal de visualização, ou porque ele é incapaz de avaliar corretamente uma porção da imagem, ou porque isto demandaria recursos maiores (de processador ou de memória) do que a utilização de raios.

1.1. Raízes históricas

A idéia de disparo de raios como uma técnica para ocultamento de linhas e superfícies foi introduzida por APPEL [41], e mais tarde usada por GOLDSTEIN e NAGEL [31] para a geração de imagens sombreadas. Entretanto, a principal referência sobre sua aplicação em modelagem de sólidos CSG é o trabalho de ROTH ([32]). Em seu artigo, Roth descreve uma adaptação do algoritmo básico de disparo de raios para utilização em modelos construtivos e demonstra sua aplicabilidade tanto na montagem de imagens sombreadas e gráficos de linhas, como no cálculo de algumas propriedades integrais, tais como volume e centro de massa. No artigo, os algoritmos são descritos em grande detalhe, abordando inclusive diversas técnicas de implementação e otimização.

1.2. Disparo de raios como técnica de ocultamento de linhas e superfícies

A maioria dos enfoques que se dá ao problema de ocultamento de linhas e superfícies, de uma forma ou de outra valem-se de características de similaridade da cena a ser visualizada. Isto é, o fato que regiões próximas na cena tendem a se apresentar com atributos visuais semelhantes na imagem sendo montada costuma ser aproveitado em muitos algoritmos, como por exemplo, os propostos por Warnock, Weiler-Atherton, os baseados em listas de prioridade e em linhas de rastreo.

Por outro lado, a técnica conhecida como "disparo de raios" ou *Ray-Casting* pode ser categorizada como de "força bruta", ou seja, cada ponto da imagem é avaliado de forma independente. A idéia básica é que a imagem de um objeto percebida por um observador é devida a raios de luz que são emitidos por alguma fonte luminosa, interfere de alguma forma com o objeto, e finalmente alcança o observador. Uma vez que muito raramente um raio emitido por uma fonte de luz efetivamente chega ao observador, acompanhar os raios neste sentido seria computacionalmente desvantajoso. Em vista disso, APPEL [41] sugeriu um acompanhamento de raios no sentido inverso. A idéia foi desenvolvida por NAGEL e GOLDSTEIN [31] da companhia *MAGI (Mathematics Applications Group, Inc.)*, e implementada num sistema de CAD/CAM chamado *Syntha Vision* (GOLDSTEIN e MALIN [42]).

A idéia consiste basicamente em definir uma reta entre a posição do observador e cada pixel da imagem a ser exibida - essa reta é acompanhada no sentido observador-

cena até que se encontre o objeto mais próximo. Caso o raio não intercepte nenhum objeto, o pixel é pintado com a cor do fundo; caso contrário, a cor do pixel é calculada considerando a iluminação recebida pelo objeto no ponto de interseção com o raio. O processo é ilustrado na figura V.1.

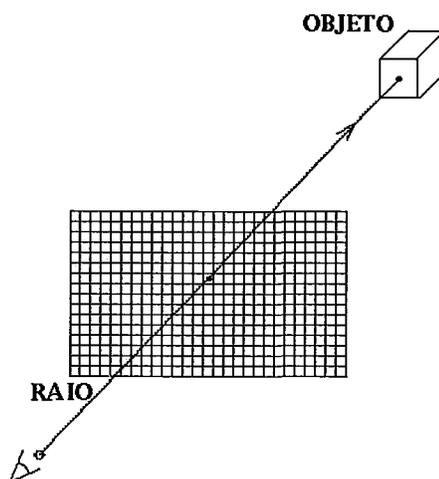


Figura V.1
Modelo para disparo de raios

Tal como foi concebido inicialmente, o cômputo da cor era feito de maneira bem simples, considerando apenas a direção do vetor normal ao objeto no ponto de interseção. Mais tarde, KAY ([43] e [44]) e WHITTED [45] implementaram modelos de iluminação mais sofisticados, levando em conta a reflexão de um objeto sobre outro, transparência e efeitos de sombra. Nesse contexto, disparo de raios foi rebatizado com o nome de "rastreamento de raios" ou *Ray-Tracing*. Agregando ainda outros elementos, como por exemplo técnicas de anti-serrilhado (*anti-aliasing*), foi possível produzir as primeiras imagens com alto grau de realismo.

1.3. Modelo projetivo da cena

Para nossos fins consideraremos uma disposição da cena como ilustrado na figura V.2. O plano de projeção (tela) é idealizado como um arranjo bidimensional de pontos (pixels) justaposto ao plano $Z=0$. A coordenada Z serve de base para a avaliação da

profundidade dos objetos, sendo que estes são posicionados em alguma parte do semiespaco $Z \geq 0$.

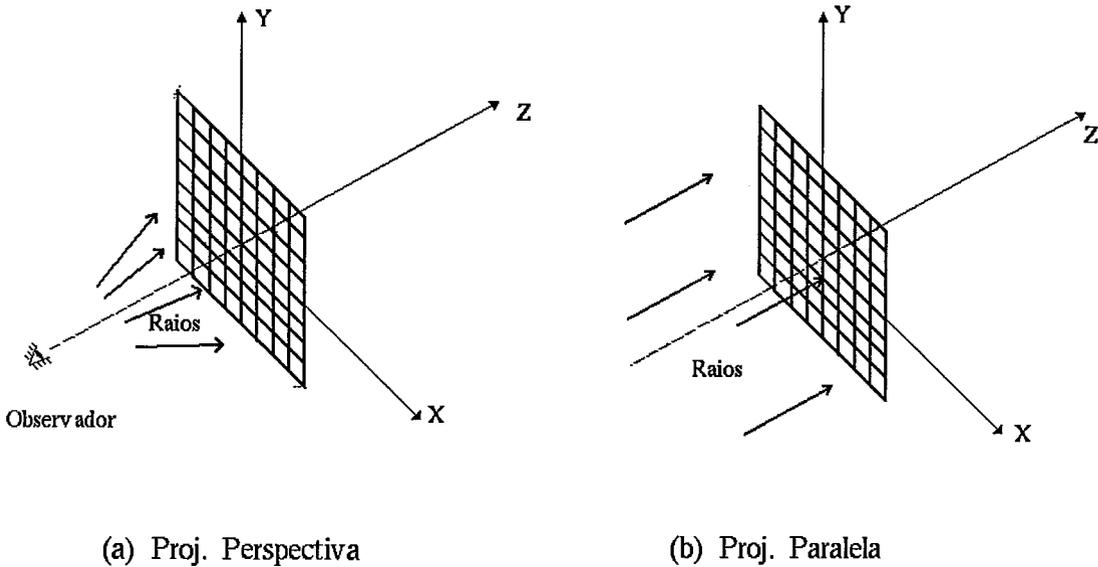


Figura V.2

Disposição da cena para o uso de disparo de raios

No caso de utilização de projeção paralela, os raios são montados com direção igual ao do eixo Z. No caso da projeção usada ser perspectiva, o observador é posicionado em algum ponto do semi-eixo $Z < 0$.

Um raio pode ser modelado simplesmente como uma semi-reta no espaço tridimensional. Como veremos adiante, uma representação conveniente para um raio é aquela dada por uma reta em forma paramétrica, com um ponto de origem (x_0, y_0, z_0) e um vetor direção (x_d, y_d, z_d) , $z_d > 0$. Nesta forma, os pontos (x, y, z) sobre a reta podem ser caracterizados de forma ordenada por um único parâmetro t , conforme a equação V.1 abaixo.

$$\begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix} + t \begin{bmatrix} x_d & y_d & z_d \end{bmatrix}, \quad t \geq 0 \quad (\text{V.1})$$

Assim, em projeção paralela, um raio que atravessasse o ponto (x_p, y_p) da tela é dado pelo ponto de origem $(x_p, y_p, 0)$ e pelo vetor direção $(0, 0, 1)$. Em projeção perspectiva,

teríamos para o ponto de origem e o vetor direção, respectivamente $(x_p, y_p, 0)$ e (x_p, y_p, z_{obs}) , dado que a origem do sistema de coordenadas da tela se encontra sobre $(0, 0, 0)$ e o observador em $(0, 0, -z_{obs})$.

1.4. Modelagem no espaço unidimensional

O método de disparo de raios se baseia no pressuposto que é possível obter uma aproximação de um modelo CSG amostrando-o através de raios.

Tomemos um raio correspondente a uma reta R , disparado sobre o espaço tridimensional onde o sólido S está imerso. O sólido S é definido por uma expressão composta dos operadores de união, interseção e diferença regularizados, e onde os operandos são conjuntos regulares de pontos $C_i, 1 \leq i \leq n$.

Desejamos saber se o conjunto de pontos dado por $R \cap S$ é igual àquele obtido substituindo-se cada operando C_i por $R \cap C_i$ na expressão do sólido, isto é,

$$R \cap (C_a \text{ op } C_b) = (R \cap C_a) \text{ op } (R \cap C_b) \quad (\text{V.2a})$$

Esta equação não é satisfeita se consideramos op como um operador regularizado na topologia de E^3 , pois o segundo termo da igualdade se reduziria \emptyset . Portanto, vamos reescrever a equação acima definindo op_1 e op_3 como operações regularizadas, respectivamente na topologia de E^1 e E^3 :

$$R \cap (C_a \text{ op}_3 C_b) = (R \cap C_a) \text{ op}_1 (R \cap C_b) \quad (\text{V.2b})$$

Uma situação onde a equação V.2b não se verifica é quando temos o raio tangenciando alguma primitiva. Nesse caso, a avaliação do primeiro termo da igualdade produziria pontos "isolados". Para eliminar essa dificuldade, imporemos que interseções $\text{reta} \times \text{sólido}$ sejam regularizadas em E^1 :

$$R \cap^*_1 (C_a \text{ op}_3 C_b) = (R \cap^*_1 C_a) \text{ op}_1 (R \cap^*_1 C_b) \quad (\text{V.2c})$$

Ainda resta-nos uma dificuldade adicional: considere por exemplo a situação ilustrada na figura II.4, admitindo um sólido composto pela interseção regularizada de dois cilindros que se tocam na região dada por um segmento de reta. Se R é disparado de forma a ser colinear com esse segmento de reta, a equação V.2 não é satisfeita para $op \equiv \cap^*$, embora seja válida para R sendo disparado segundo uma outra direção qualquer. Uma análise semelhante pode ser feita para casos onde a operação não

regularizada resulta em curvas ou superfícies.

Entretanto, se levamos em conta que, tipicamente, para a avaliação de modelos CSG uma grande quantidade de raios é disparada, podemos afirmar que situações como as que descrevemos são raras, e portanto, o modelo de disparo de raios pode ser considerado como consistente, isto é as propriedades do modelo podem ser calculadas usando essa técnica com uma margem de erro desprezível.

1.5. Representação para um conjunto regular de pontos da reta real

Definimos como "conjunto regular de pontos da reta real" um conjunto obtido interceptando-se de uma reta com um sólido CSG. Seja R uma reta, e C_i o conjunto de pontos correspondente a uma das primitivas de um sólido S qualquer. Se admitimos que C_i é um conjunto regular conforme descrito no capítulo II, então $R \cap^* C_i$ é um conjunto com um número finito de pontos de fronteira e, mais ainda, sendo C_i um conjunto limitado, esse número é par.

Um conjunto unidimensional de pontos pode portanto ser caracterizado por um número discreto de intervalos disjuntos sobre a reta R . Tais intervalos podem ser modelados determinando-se os pontos de fronteira do conjunto, isto é, os pontos onde a reta "entra" ou "sai" do sólido. A posição desses pontos da reta pode ser registrada através dos valores do parâmetro t correspondentes, que são então arranjados numa lista ordenada $t_i, 1 \leq i \leq n$, onde n é o número total de pontos de interseção entre o raio e a fronteira dos objetos da cena por ele interceptados. Assim, para valores de t menores que t_1 , os pontos correspondentes estão "fora" do sólido, em $t=t_1$ o raio "entra" no sólido, em $t=t_2$ ele "sai", e assim sucessivamente. Um exemplo desse processo pode ser apreciado na figura V.3.

Paralelamente à lista t_i , é necessário também armazenar informações sobre as superfícies interceptadas - essas informações são postas numa segunda lista, digamos, $S_i, 1 \leq i \leq n$. As informações sobre as superfícies serão usadas para a determinação da cor do pixel correspondente ao raio disparado. A composição de cada elemento da lista S_i depende do tipo de imagem que se quer gerar. Por exemplo, para a obtenção de imagens sombreadas segundo um modelo de iluminação simples, basta guardar o vetor normal à superfície e uma identificação do material de que é composto a primitiva atingida pelo raio.

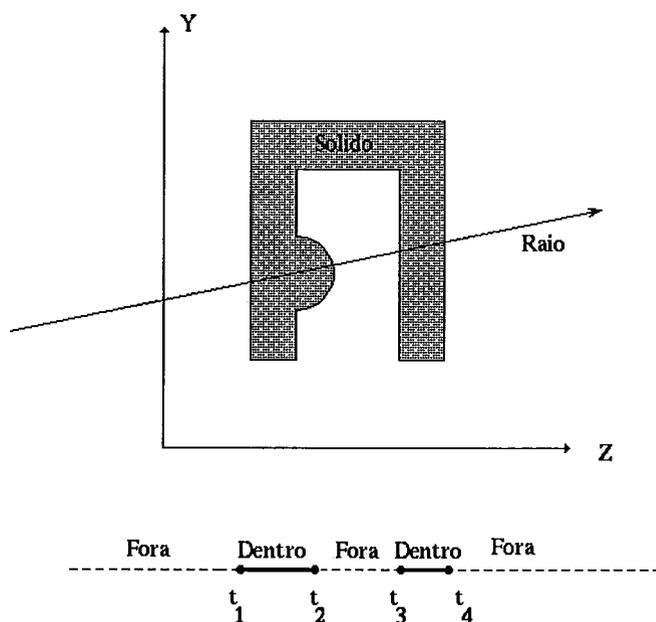


Figura V.3
Exemplo de um disparo de raio

1.6. Avaliação de uma árvore CSG na reta real

Fica claro então que as únicas operações geométricas necessárias aqui são as que correspondem ao cálculo dos pontos de interseção entre o raio e cada um dos sólidos primitivos que compõem o objeto.

As operações lógicas sobre os conjuntos de pontos sobre a reta real, representados por suas listas de interseção, são feitas recursivamente, percorrendo a árvore CSG em pós-ordem. Para cada nó interno da árvore, são computadas duas listas de interseção: uma associada à sub-árvore da esquerda e outra para a sub-árvore da direita. Essas listas são então analisadas e uma lista resultante da operação lógica representada pelo nó interno é calculada.

O processo para combinar duas listas é composto de três etapas:

- (1) *Composição*: É montada uma terceira lista intercalando-se ordenadamente os pontos de interseção das listas da esquerda e da direita;
- (2) *Classificação*: os intervalos da lista composta são classificados como "dentro" ou "fora", dependendo da operação lógica sendo realizada e das classificações dos segmentos das listas da esquerda e da direita.
- (3) *Simplificação*: São retirados da lista composta os pontos supérfluos, isto é, aqueles que têm seus dois segmentos vizinhos classificados de maneira igual (dentro/dentro ou fora/fora).

Este algoritmo é exemplificado na figura V.4.

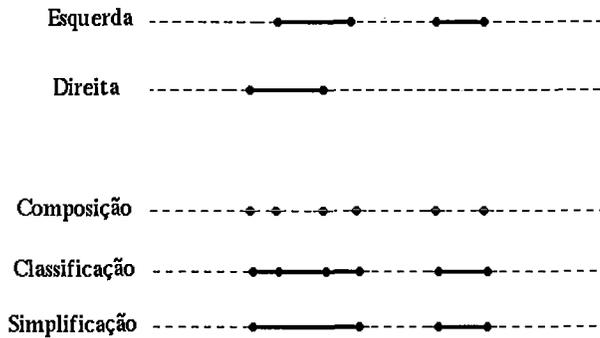


Figura V.4

Processo para combinação de listas de interseção.

1.7. Cálculo de interseções raio \times primitiva

Como discutimos no capítulo II, os sólidos primitivos, ao serem agregados à árvore CSG, podem sofrer um processo de instanciação, isto é, podem ser girados, transladados ou escalados. O cálculo de pontos de interseção de raios contra primitivas instanciadas arbitrariamente, se efetuado no espaço da cena, requer que se determine os coeficientes das equações das superfícies de fronteira dessas primitiva. Entretanto, por

exemplo, fazer o cálculo dos pontos de interseção de uma reta contra uma esfera de raio unitário centrada na origem mostra-se uma tarefa bem simples.

A solução então é definir cada tipo de primitiva segundo um sistema de coordenadas local. Assim, uma esfera primitiva é definida em seu sistema de coordenadas local da maneira como descrevemos há pouco; um cilindro é definido como sendo de seção circular, raio e altura unitários, posicionado com seu eixo sobre o semi-eixo $Z > 0$ e base sobre o plano $Z = 0$, e assim por diante, para todos os sólidos primitivos. A figura V.5 ilustra os sistemas de coordenadas locais para os sólidos primitivos presentes em nossa implementação: esferas, cilindros e paralelepípedos.

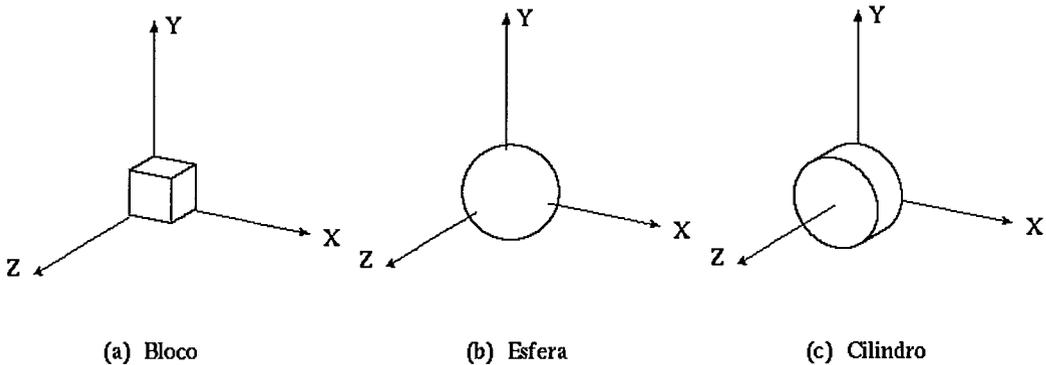


Figura V.5
Sistemas de coordenadas locais dos sólidos primitivos

A instanciação de um sólido primitivo é normalmente feita estabelecendo-se uma transformação linear afim, digamos T , correspondente à mudança do sistema de coordenadas local para o sistema de coordenadas da cena. Assim, para poder montar as listas de interseção raio \times primitiva, ao invés de efetuar o cálculo no sistema de coordenadas da cena, utiliza-se o sistema de coordenadas da primitiva. Isto é feito aplicando-se ao raio a inversa da transformação de instanciação, isto é, T^{-1} (observação: é garantida a existência de T^{-1} uma vez que estabelecemos no capítulo II que operações de instanciação são inversíveis).

Por conveniência em aplicações de mudança de sistemas de referência, utiliza-se coordenadas homogêneas, o que possibilita representar-se a transformação linear afim T por uma matriz 4×4 . Em coordenadas homogêneas um raio, tal como o definimos

na equação V.1 pode ser reescrito como

$$\begin{bmatrix} x & y & z & w \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & z_0 & 1 \end{bmatrix} + t \begin{bmatrix} x_d & y_d & z_d & 0 \end{bmatrix} \quad (\text{V.3})$$

Seja um raio definido no sistema de coordenadas da cena por $Origem = (x_0, y_0, z_0, 1)$ e $Direcao = (x_d, y_d, z_d, 0)$. Então, o raio transformado para o sistema de coordenadas local da primitiva, definido por $Origem' = (x_0', y_0', z_0', 1)$ e $Direcao' = (x_d', y_d', z_d', 0)$ pode ser calculado pelas seguintes equações:

$$\begin{aligned} \begin{bmatrix} x_0' & y_0' & z_0' & 1 \end{bmatrix} &= \begin{bmatrix} x_0 & y_0 & z_0 & 1 \end{bmatrix} \times T^{-1} \\ &e \\ \begin{bmatrix} x_d' & y_d' & z_d' & 0 \end{bmatrix} &= \begin{bmatrix} x_d & y_d & z_d & 0 \end{bmatrix} \times T^{-1} \end{aligned} \quad (\text{V.4})$$

Uma vez que estamos utilizando uma parametrização linear para o raio e que as transformações sobre ele aplicadas são lineares afim, podemos afirmar que a transformação do raio não altera sua parametrização. Isto é o mesmo que dizer que, sendo $P'(t)$ o ponto dado por $Origem' + t.Direcao'$, então vale a seguinte equação:

$$P'(t) = P(t) \times T^{-1} \quad (\text{V.5})$$

A significância prática da invariância da parametrização reside no fato que não é necessário calcular pontos de interseção, mas apenas os valores do parâmetro t correspondentes. Os sólidos primitivos ou as equações de suas superfícies de fronteira nunca são efetivamente transformados. O efeito obtido ao se girar, transladar ou escalar primitivas é reproduzido fazendo as operações inversas sobre os raios. Assim, elipsóides, cilindros de seção elíptica ou paralelepípedos genéricos podem ser modelados sem dificuldade.

1.8. Estruturas de dados para disparo de raios

A fim de montar uma implementação para a técnica descrita na seção anterior precisamos nos ocupar ainda de alguns detalhes.

Em primeiro lugar, é preciso montar uma estrutura para o armazenamento da árvore CSG correspondente ao sólido a ser visualizado. Convém aqui estabelecermos a convenção a ser usada na descrição das estruturas de dados e algoritmos. Para isso usaremos uma pseudo linguagem baseada na sintaxe da linguagem Modula-2. Há duas razões para esta escolha: (1) Modula-2 é uma linguagem aparentada com Pascal, oferecendo um bom repertório de estruturas primitivas, e (2) a grande maioria dos algoritmos que iremos descrever foi efetivamente implementada nesta linguagem.

A árvore é montada como uma estrutura encadeada através de ponteiros como é praxe na maioria das implementações de estruturas de dados recursivas. A montagem dos nós internos não oferece maiores dificuldades: um nó interno de uma árvore CSG consiste de um indicador da operação lógica a ser efetuada entre suas duas sub-árvores à esquerda e à direita.

Para a montagem dos nós externos, isto é, aqueles que descrevem os sólidos primitivos, Roth propõe que se armazene (no mínimo) o seguinte:

- Um código indicativo do tipo da primitiva (esfera, cilindro, etc);
- A matriz 4×4 que representa a operação de instanciação, isto é, a transformação que leva a primitiva de seu sistema de coordenadas local para o espaço de coordenadas da cena;
- A inversa da matriz de instanciação, isto é, a transformação a ser usada para levar um raio definido no sistema de coordenadas da cena para o sistema de coordenadas local da primitiva.

Estas estruturas são descritas em pseudo código da seguinte forma:

```
TYPE
  TipoOperadorCSG = (And, Or, Comp);
  Transformacao = ARRAY [1 .. 4],[1 .. 4] OF REAL;
  ArvoreCSG = POINTER TO NoArvoreCSG;
  NoArvoreCSG =
    RECORD
      CASE Simples : BOOLEAN OF
        | TRUE:
          Primitiva : TipoPrimitivaCSG;
          LocalParaCena,
          CenaParaLocal : Transformacao;
        | FALSE:
          Operador : TipoOperadorCSG;
          Esquerda,
          Direita: ArvoreCSG;
      END
    END;
END;
```

No código acima, *TipoPrimitivaCSG* é um tipo que identifica a natureza da primitiva presente num nó-folha da árvore.

A segunda estrutura que devemos considerar diz respeito à definição dos raios. Como vimos, um raio pode ser representado por ponto e um vetor em E^3 . Temos então a seguinte estrutura de dados:

```
TYPE
  PontoE3 = ARRAY [1 .. 3] OF REAL;
  Raio =
    RECORD
      Origem, Direcao : PontoE3
    END;
END;
```

Finalmente, definimos a estrutura para as listas de interseção. Estas são implementadas como listas encadeadas por ponteiros, sendo que cada elemento da lista contém o valor do parâmetro t correspondente, um ponteiro para o sólido primitivo interceptado e um código indicador da superfície.

Este último elemento tem a finalidade de permitir mais tarde o cálculo do vetor normal à primitiva. Assim, se um elemento da lista corresponde, por exemplo, à

interseção com um cilindro, o código indicador de superfície poderá valer 1, 2 ou 3, segundo a interseção tenha ocorrido com a "base", o "topo" ou a "parede" do cilindro. Similarmente, um código de superfície para interseções com blocos poderá valer de 1 a 6, conforme a face atingida pelo raio. No caso de uma esfera, o código de superfície é irrelevante.

Temos então o seguinte pseudo-código:

```
TYPE
  ListaIntersecao = POINTER TO NoIntersecao;
  NoIntersecao =
    RECORD
      T : REAL;
      Solido : ArvoreCSG;
      CodigoSuperficie : INTEGER;
      Prox : ListaIntersecao;
    END;
```

1.9. Algoritmo básico de disparo de raios

Dadas as estruturas de dados expostas anteriormente, podemos redigir um pseudo-código para a rotina principal do algoritmo de disparo de raios. A rotina *JogaRaio* recebe como parâmetros um sólido definido por uma árvore CSG e um raio, e retorna a lista de interseções correspondente:

```
PROCEDURE JogaRaio (R : Raio; A : ArvoreCSG) : ListaIntersecao;
```

```
  BEGIN
    IF A^.Simples THEN
      R := TransformaRaio (R, A^.CenaParaLocal);
      RETURN InterceptaPrimitiva (A^.Primitiva, R);
    ELSE
      RETURN
        CombinaListas (
          JogaRaio (R, A^.Esquerda),
          JogaRaio (R, A^.Direita),
          A^.Operador
        );
    END
  END JogaRaio;
```

O procedimento acima invoca algumas rotinas auxiliares, a saber:

- *TransformaRaio*: recebe um raio e uma matriz de transformação e retorna o raio no novo sistema de coordenadas.
- *InterceptaPrimitiva*: dado um raio e um identificador de primitiva, retorna a listas correspondente aos pontos de interseção raio \times primitiva.
- *CombinaListas*: retorna a lista de interseções resultante da combinação de duas listas segundo um operador lógico (interseção, união ou diferença) passados como parâmetros.

2. Aplicações

A rotina *JogaRaio* serve como base para muitas aplicações. Ela permite, por exemplo, avaliar propriedades tais como massa e volume de um dado sólido CSG.

Para calcular o volume de um sólido, basta que pensemos nos intervalos unidimensionais de pontos definidos pelas interseções dos raios contra o sólido como

paralelepípedos. Uma aproximação do volume pode então ser obtida disparando-se raios segundo uma projeção paralela e multiplicando-se a área de um pixel pelo somatório dos comprimentos dos intervalos dados pelas interseções raio \times sólido.

Para o cálculo da massa, o mesmo processo pode ser empregado, sendo que no cálculo devem ser levadas em conta as densidades das diversas primitivas.

Neste trabalho, no entanto, temos particular interesse nas aplicações relacionadas com a geração de imagens.

2.1. O processo de visualização

A confecção de imagens requer que se empregue um raio para a avaliação de cada pixel da tela. Isto é feito em linhas gerais segundo o seguinte pseudo-código:

```
PROCEDURE GeraImagem (  
    XMin, XMax : INTEGER;  
    YMin, YMax : INTEGER;  
    Arvore : ArvoreCSG  
);  
  
VAR  
    IX, IY : INTEGER;  
    R : Raio;  
    L : ListaIntersecao;  
    Cor : INTEGER;  
  
BEGIN  
    FOR IX := XMin TO XMax DO  
        FOR IY := YMin TO YMax DO  
            R := CriaRaio (IX, IY);  
            L := JogaRaio (R, Arvore);  
            Cor := ComputaCorPixel (R, L);  
            PintaPixel (IX, IY, Cor)  
        END  
    END  
END GeraImagem;
```

O procedimento *GeraImagem* recebe a árvore a ser visualizada (*Arvore*) e a definição do plano de projeção através de seus limites em X e Y (*XMin*, *XMax*, *YMin* e *YMax*). As seguintes rotinas auxiliares são usadas:

- *CriaRaio*: retorna a definição do raio que passa por um ponto (*X*, *Y*) da tela. São levados em consideração os dados relativos ao tipo de sistema projetivo sendo empregado.
- *ComputaCorPixel*: dado um raio e uma lista de interseções, esta rotina é responsável pela avaliação da cor do pixel correspondente. Os cálculos por ela realizados vão depender do tipo de imagem que se quer avaliar e de outros fatores relacionados com o modelo de iluminação.

- *PintaPixel*: rotina dependente do ambiente gráfico sendo utilizado. É responsável por exibir um ponto (X, Y) com uma cor dada.

2.2. Geração de imagens sombreadas

Para confeccionar uma imagem sombreada é preciso conhecer, para cada pixel da imagem, alguns parâmetros relativos ao modelo de iluminação adotado. Desses, a normal à superfície é a única característica geométrica que deve ser calculada dentro do sistema de disparo de raios. Este cálculo é simples de ser feito no sistema de coordenadas local da primitiva. Para uma esfera, por exemplo, dado que a interseção se deu para $t = t_0$, a normal pode ser calculada diretamente da equação da reta $(Orig', Dir')$:

$$Normal = Orig' + t_0 * Dir' \quad (V.6)$$

Naturalmente, o vetor normal calculado dessa maneira precisa ser então transportado para o sistema de referência da cena.

Em nossa implementação usamos o modelo de iluminação simplificado dado por WHITTED [45]:

$$I = k_a I_a + k_d \sum_{j=1}^m I_{l_j} \hat{n} \cdot \hat{L}_j + k_s \sum_{j=1}^m I_{l_j} (\hat{S} \cdot \hat{R}_j)^n \quad (V.7)$$

onde

- k_a coeficiente de iluminação ambiente
- k_d coeficiente de iluminação difusa
- k_s coeficiente de iluminação especular
- n constante de especularidade de Phong
- m número de fontes de iluminação
- I_{l_j} Intensidade da fonte de iluminação l_j

- \hat{n} vetor normal à superfície
- \hat{L}_j vetor desde o ponto da superfície até a fonte de iluminação j
- \hat{S} vetor desde o ponto da superfície até o observador
- \hat{R}_j vetor correspondente a um raio de luz da fonte de iluminação j refletido pela superfície

Os parâmetros especificados acima consistem de: (1) constantes arbitradas em função das características luminosas do objeto e das fontes luminosas e (2) vetores facilmente calculados por procedimentos usuais de álgebra linear.

Dependendo do modelo de iluminação a ser adotado, uma série de informações adicionais deve ser agregada à descrição de cada nó das listas de interseções.

Esses detalhes de implementação podem ser encontrados em ROGERS [46].

3. Geração de gráficos de linhas

O gráfico de linhas corresponde ao traçado tão somente das linhas correspondentes a arestas e ao perfil do sólido CSG. A principal utilidade deste tipo de representação é permitir uma visualização rápida embora grosseira do sólido sendo modelado. Idealmente, a geração de um gráfico de linhas deve ser o mais rápida possível a fim de permitir um bom nível de interatividade durante a fase de modelagem. A figura V.6 exemplifica um gráfico de linhas de um sólido CSG.

As arestas correspondem ao lugar geométrico dos pontos onde duas superfícies de fronteira se juntam. O que chamamos de "perfil" é dado pelos pontos onde os raios tangenciam a fronteira do sólido. Esses dois componentes podem ser facilmente detectados por disparo de raios pois seu lugar geométrico corresponde às áreas da imagem onde dois raios disparados próximos um do outro atingem duas primitivas diferentes ou duas superfícies de fronteira da mesma primitiva.

Apesar de ser a técnica de disparo de raios uma abordagem eminentemente "pesada" em termos de consumo de ciclos de processador, Roth demonstrou que no caso em que se deseja apenas um gráfico de linhas o processo pode ser substancialmente acelerado uma vez que não é necessário jogar raios sobre todos os pontos da imagem. Para tanto, se faz inicialmente uma varredura grosseira da imagem

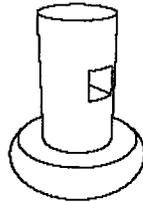


Figura V.6
Gráfico de linhas de um sólido CSG

disparando raios sobre os vértices de uma malha retangular sobreposta ao plano de projeção. Apenas nas células dessa malha onde os disparos correspondentes aos seus quatro vértices indicam uma mudança de estado (sólidos diferentes ou superfícies de um mesmo sólido) é necessário disparar raios adicionais a fim de localizar mais precisamente os pontos correspondentes à aresta ou à linha de perfil.

O processo de refinamento pode ser implementado de forma a aproximar-se de uma busca binária. Assim, por exemplo, se é detectada uma mudança de estado ao jogar-se dois raios em (X,Y) e $(X+1,Y)$, um raio adicional seria disparado em $(X+0.5,Y)$. O processo seria então repetido até chegar-se à resolução do dispositivo.

Obviamente, usando amostragem corre-se o risco de deixar algum detalhe mais fino desaparecido. Na dúvida, é sempre facultado ao usuário usar uma amostragem mais fina.

Em nossa implementação, o processo de amostragem foi montado de forma ligeiramente diferente da proposta por Roth. O algoritmo empregado baseia-se em uma subdivisão espacial de cada célula da malha à maneira de uma *Quadtree*.

Considere uma célula da malha de amostragem onde seus 4 vértices, aos quais nos referiremos pelos nomes *NW*, *NE*, *SW* e *SE*, foram avaliados usando disparo de raios. Se os raios disparados sobre esses quatro vértices atingiram o mesmo sólido sobre a mesma superfície de fronteira, então assume-se que a célula não contém nenhuma aresta ou linha de perfil e a avaliação da célula é encerrada.

Caso contrário, são disparados 5 raios adicionais: 4 sobre o ponto médio de cada aresta e um sobre o centro da célula. Esses pontos são chamados, respectivamente, de *N*, *S*, *W*, *E* e *C* (veja a figura V.7). Esses 5 pontos, somados aos 4 pontos originais

definem os vértices de 4 sub-regiões da célula, que são então analisadas recursivamente. O processo termina quando a região da célula sendo analisada corresponde a um pixel da imagem - sendo que o pixel correspondente é pintado se a avaliação dos 4 vértices é não homogênea.

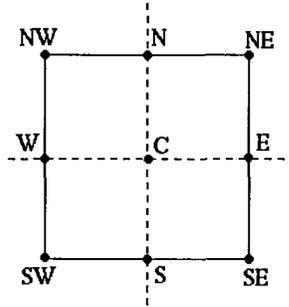


Figura V.7

Subdivisão de uma célula de amostragem

Um detalhe de implementação a ser observado é que o processo de subdivisão descrito acima pode visitar o mesmo ponto da célula duas vezes. Para evitar isso, define-se uma matriz de $n \times n$ elementos, onde n é o número de pixels correspondente a uma aresta da célula de amostragem. Cada elemento da matriz contém, além das informações necessárias para o teste de homogeneidade entre vértices, um *indicador* que é ligado sempre que aquele elemento é avaliado. A implementação dessa matriz em pseudo-código é dada por:

```
TYPE
  ElementoMatriz =
    RECORD
      Indicador : BOOLEAN;
      Info : ListaIntersecao;
    END;
VAR
  MatrizCelula :
    ARRAY [0 .. TamanhoCelula],[0 .. TamanhoCelula]
    OF ElementoMatriz;
```

A rotina de visita da imagem evita a avaliação dupla de um ponto consultando o valor de *Indicador* do elemento correspondente da matriz - um novo raio é disparado apenas se este estiver desligado. O pseudo-código para a rotina *VisitaPonto* é dado por:

```
VAR
  X0, Y0 : INTEGER;

PROCEDURE VisitaPonto (X, Y : INTEGER) : ListaIntersecao;

VAR
  Lista : ListaIntersecao;
  R : Raio;

BEGIN
  IF NOT MatrizCelula [X, Y].Indicador THEN
    R := CriaRaio (X0 + X, Y0 + Y);
    Lista := JogaRaio (R, Arvore);
    MatrizCelula [X, Y].Indicador := TRUE;
    MatrizCelula [X, Y].Info := Lista;
  END;
  RETURN MatrizCelula [X, Y].Info
END VisitaPonto;
```

onde *X0* e *Y0* são as coordenadas do vértice inferior esquerdo da célula, e *X* e *Y* são as coordenadas do pixel em relação a esse vértice.

A visita recursiva dos pontos da célula é feita pela rotina *AvaliaCelula* cujo pseudo-código apresentamos abaixo:

```
PROCEDURE AvaliaCelula (X, Y, Tamanho : INTEGER);
```

```
VAR
```

```
    NW, NE, SW, SE : ListaIntersecao;
```

```
BEGIN
```

```
    SW := VisitaPonto (X, Y);
```

```
    SE := VisitaPonto (X+Tamanho,Y);
```

```
    NW := VisitaPonto (X, Y+Tamanho);
```

```
    NE := VisitaPonto (X+Tamanho, Y+Tamanho);
```

```
    IF NOT Homogeneo (NW, NE, SW, SE) THEN
```

```
        IF Tamanho = 1 THEN
```

```
            PintaPixel (X0+X, Y0+Y, 1)
```

```
        ELSE
```

```
            Tamanho := Tamanho DIV 2;
```

```
            AvaliaCelula (X, Y, Tamanho);
```

```
            AvaliaCelula (X+Tamanho, Y, Tamanho);
```

```
            AvaliaCelula (X, Y+Tamanho, Tamanho);
```

```
            AvaliaCelula (X+Tamanho, Y+Tamanho, Tamanho);
```

```
        END
```

```
    END
```

```
END AvaliaCelula;
```

A figura V.8 ilustra o processo de divisão recursiva de uma célula que contém uma aresta.

4. Otimização

Roth demonstra que o principal fator na determinação da complexidade do processo de disparo de raios é o número de classificações raio primitiva, isto é, quanto maior o número médio de interseções entre cada raio e as primitivas da árvore CSG, mais demorado é o processo. Neste sentido, a melhoria mais evidente que podemos pensar para o algoritmo é evitar fazer testes de interseções raio \times primitiva em áreas onde garantidamente não há chances de interseção.

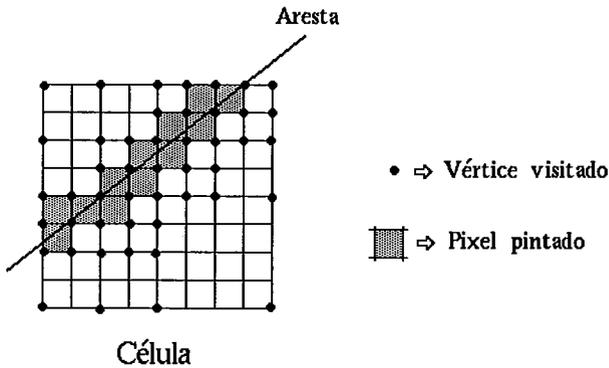


Figura V.8
Traçado de uma aresta

4.1. O uso de caixas envolventes

Uma caixa envolvente de um sólido é um poliedro convexo que o contém. Roth introduz um refinamento no algoritmo usando como caixas envolventes paralelepípedos com faces perpendiculares aos eixos coordenados do espaço da cena. Para representar uma caixa utiliza-se uma 6-upla definindo as cotas mínimas e máximas com relação a cada um dos 3 eixos coordenadas, ou seja:

$$\text{Caixa} = (X_{\min}, X_{\max}, Y_{\min}, Y_{\max}, Z_{\min}, Z_{\max}) \quad (\text{V.8})$$

Uma vez definidas as caixas envolventes, um raio sendo disparado sobre o ponto (X, Y) do plano de projeção pode ser desconsiderado para fins de cálculo de interseção se uma das seguintes condições for satisfeita:

$$\left\{ \begin{array}{l} X < X_{\min} \\ X > X_{\max} \\ Y < Y_{\min} \\ Y > Y_{\max} \end{array} \right. \quad (\text{V.9})$$

As caixas envolventes são calculadas numa fase anterior à visualização do sólido e dependem naturalmente do sistema de projeção sendo empregado. O cômputo da caixa envolvente de uma primitiva é composta das seguintes fases:

- Relacione os vértices de um poliedro que englobe a primitiva em seu sistema de coordenadas local. Por exemplo, para um bloco, utilize simplesmente seus vértices $(0,0,0)$, $(0,0,1)$, $(0,1,0)$, etc; para um cilindro, os vértices dos dois quadrados que contêm seção na base e no topo, e assim por diante para as demais primitivas.
- Transforme os pontos relacionados para o sistema de coordenadas normalizado da cena.
- Projete os pontos transformados segundo o esquema projetivo sendo usado. Se a projeção é paralela, um ponto (X,Y,Z) será projetado em $(X,Y,0)$. Se a projeção é perspectiva com o observador posicionado em $(0,0,Z_o)$, o ponto será projetado em $(X*Z_o/(Z_o-Z), Y*Z_o/(Z_o-Z), 0)$.
- Calcule os valores máximo e mínimo entre os valores das coordenadas X e Y dos pontos projetados e das coordenadas Z dos pontos não projetados. Esses limites definem a caixa envolvente da primitiva. Observe que a coordenada Z será usada apenas para distinguir relações entre profundidades, o que pode ser feito satisfatoriamente empregando os pontos não transformados pela projeção.

Para calcular a caixa envolvente correspondente a um nó interno da árvore CSG do sólido, *CaixaNo*, dadas as caixas correspondentes às sub-árvores à esquerda e à direita do nó interno, respectivamente *CaixaEsq* e *CaixaDir*, utiliza-se as seguintes regras:

- União:

$$\begin{aligned} \text{CaixaNo} = & \\ & (\min (\text{CaixaEsq}.X_{\min} , \text{CaixaDir}.X_{\min}), \\ & \max (\text{CaixaEsq}.X_{\max} , \text{CaixaDir}.X_{\max}), \\ & \min (\text{CaixaEsq}.Y_{\min} , \text{CaixaDir}.Y_{\min}), \\ & \max (\text{CaixaEsq}.Y_{\max} , \text{CaixaDir}.Y_{\max}), \\ & \min (\text{CaixaEsq}.Z_{\min} , \text{CaixaDir}.Z_{\min}), \\ & \max (\text{CaixaEsq}.Z_{\max} , \text{CaixaDir}.Z_{\max})) \end{aligned}$$

- Interseção:

```
CaixaNo =  
  ( max ( CaixaEsq.Xmin , CaixaDir.Xmin ),  
    min ( CaixaEsq.Xmax , CaixaDir.Xmax ),  
    max ( CaixaEsq.Ymin , CaixaDir.Ymin ),  
    min ( CaixaEsq.Ymax , CaixaDir.Ymax ),  
    max ( CaixaEsq.Zmin , CaixaDir.Zmin ),  
    min ( CaixaEsq.Zmax , CaixaDir.Zmax ) )
```

- Diferença:

```
CaixaNo = CaixaEsq
```

Adaptando a estrutura de dados descrita anteriormente para incluir caixas envolventes, temos a seguinte definição para um nó da árvore CSG:

```
TYPE  
  CaixaEnvolvente =  
    RECORD  
      Minimo, Maximo : PontoE3;  
    END;  
  NoArvoreCSG =  
    RECORD  
      Caixa : CaixaEnvolvente;  
      CASE Simples : BOOLEAN OF  
        | TRUE:  
          Primitiva : TipoPrimitivaCSG;  
          LocalParaCena,  
          CenaParaLocal : Matriz;  
        | FALSE:  
          Operador : TipoOperadorCSG;  
          Esquerda,  
          Direita: ArvoreCSG;  
      END  
    END;  
END;
```

O processo de determinação das caixas envolventes pode ser realizado pela rotina *AvaliaCaixas* cujo pseudo-código apresentamos abaixo:

```
PROCEDURE AvaliaCaixas (A : ArvoreCSG);

VAR
  I : INTEGER;

BEGIN
  IF A^.Simples THEN
    A^.Caixa := CaixaPrimitiva (A^.Primitiva, A^.CenaParaLocal);
  ELSE
    AvaliaCaixas (A^.Esquerda);
    AvaliaCaixas (A^.Direita);
    CASE A^.Operador OF
      | And:
        FOR I := 1 TO 3 DO
          A^.Caixa.Minimo [I] :=
            max (A^.Esquerda^.Minimo [I], A^.Direita^.Minimo [I]);
          A^.Caixa.Maximo [I] :=
            min (A^.Esquerda^.Maximo [I], A^.Direita^.Maximo [I]);
        END;
      | Or:
        FOR I := 1 TO 3 DO
          A^.Caixa.Minimo [I] :=
            min (A^.Esquerda^.Minimo [I], A^.Direita^.Minimo [I]);
          A^.Caixa.Maximo [I] :=
            max (A^.Esquerda^.Maximo [I], A^.Direita^.Maximo [I]);
        END;
      | Comp:
        A^.Caixa := A^.Esquerda.Caixa;
    END;
  END
END AvaliaCaixas;
```

A figura V.8 mostra como se processa o cômputo de caixas envolventes para um exemplo de sólido CSG.

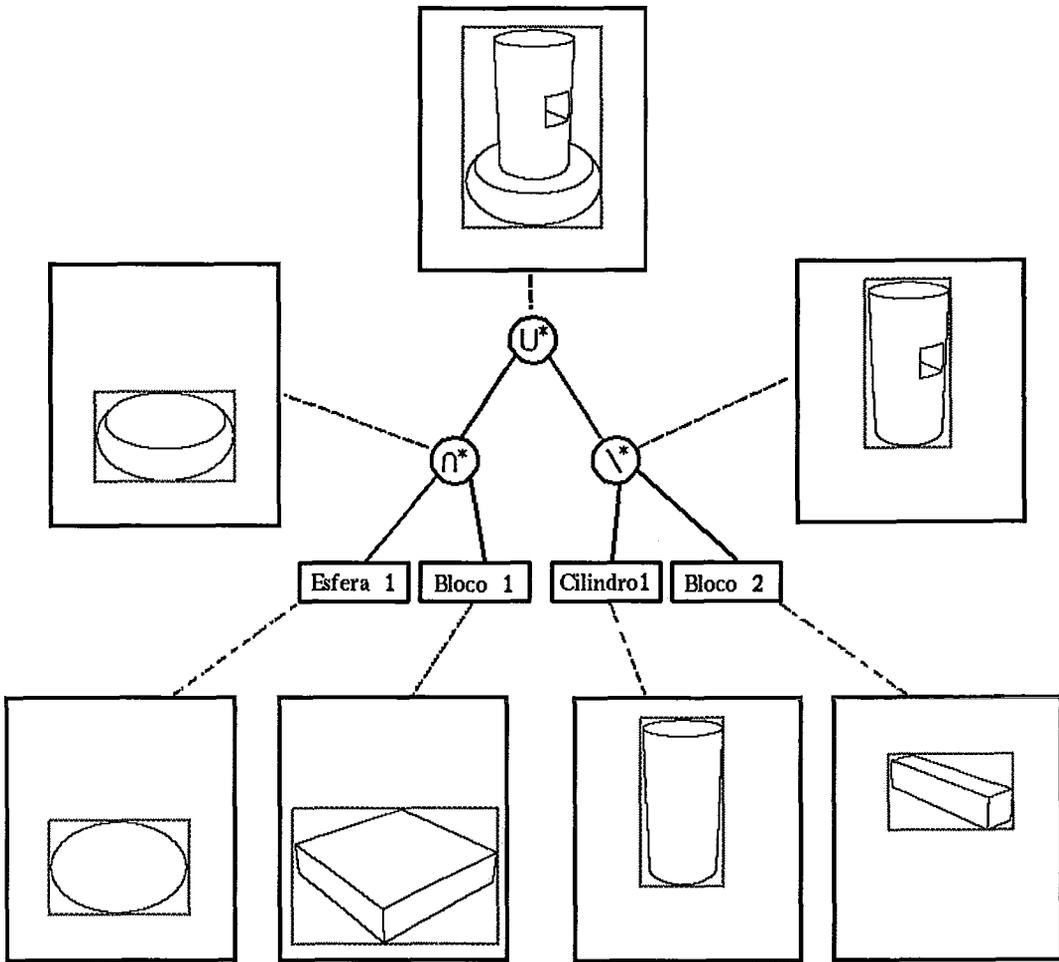


Figura V.9
Determinação de caixas envolventes de um sólido CSG

4.2. Adaptação do modelo com caixas envolventes

Uma análise mais detalhada do uso feito por Roth do modelo com caixas envolventes nos revela uma grande similaridade com o modelo de decomposição espacial exposto no capítulo IV. Com efeito, Roth aproxima a localização de árvores CSG através de caixas envolventes para eliminar a necessidade de caminhar por sub-árvores cuja contribuição para a determinação da cor de um pixel é garantidamente nula.

Podemos observar que o uso de caixas envoltivas tende a abreviar o percurso de visita dos nós da árvore. Entretanto, em muitos casos, o algoritmo só conclui que um raio pode ser descartado depois de trilhar um longo caminho sobre a árvore.

Em contraposição, se calcularmos uma localização da árvore para a região varrida pelo raio, esta possuirá apenas as primitivas potencialmente relevantes.

Um exemplo nos ajuda a visualizar esta diferença. Tomemos o sólido ilustrado na figura V.10 abaixo, composto da união de 2 cilindros.

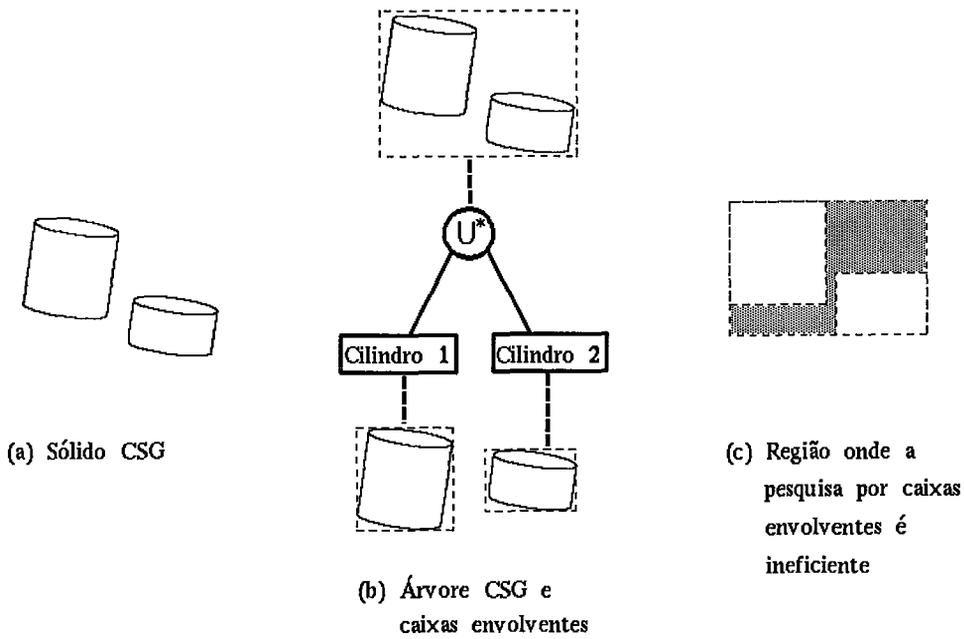


Figura V.10
Ineficiência na pesquisa por caixas envoltivas

Podemos ver que a caixa envoltiva associada à raiz da árvore compreende uma região onde o raio não atingirá nenhum sólido primitivo. Nessa região, apenas após analisar as caixas dos sólidos à esquerda e à direita da raiz, será possível descartar o raio.

Como decorrência desta análise, foi montado um algoritmo usando caixas envoltivas e subdivisão espacial. A idéia consiste em ter, para cada pixel da imagem, uma árvore localizada.

O processo de localização da árvore original para uma dada região da imagem é baseado em testes sobre as caixas envolventes das primitivas, e não sobre as próprias primitivas. A razão disso é que testes sobre caixas envolventes é bem menos dispendioso em termos de tempo de processamento. Como decorrência, podemos ver que computar uma localização para a área ocupada por cada pixel da imagem não deve resultar numa boa razão custo \times benefício. Uma solução de compromisso é dividir a imagem numa grade grosseira, tal que cada célula da grade contenha um arranjo de, digamos, $n \times n$ pixels. Um diagrama esquemático dessa situação é mostrado na figura V.11.

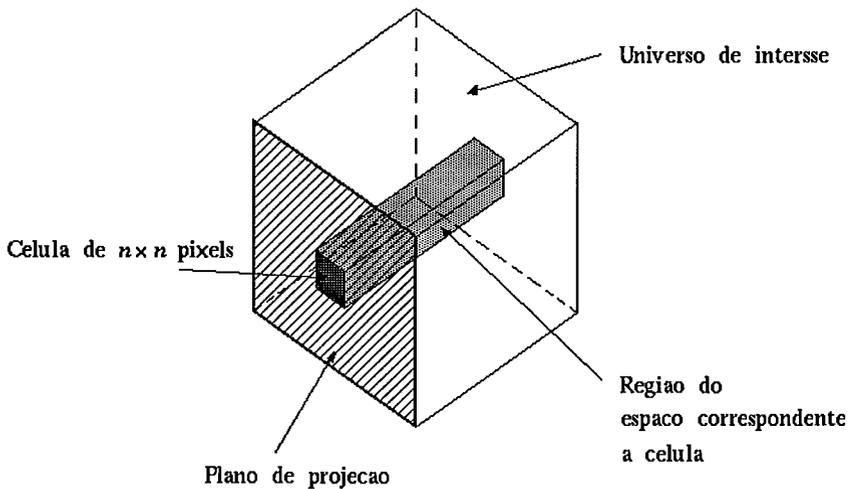


Figura V.11
Subdivisão por células

Resumindo, o processo é composto dos seguintes passos:

- A fase de pré-processamento da árvore CSG para a determinação das caixas envolventes é feita da maneira exposta por Roth e que descrevemos anteriormente.
- O plano de projeção é dividido em uma malha retangular grosseira onde cada célula corresponde a um arranjo de $n \times n$ pixels. O valor de n pode ser ajustado através da linha de comando que invoca o programa de visualização, valendo

tipicamente entre 4 e 32.

- Para cada raio disparado é determinada a célula da malha correspondente. O processo de determinação dos pontos de interseção e de combinação das listas de intervalos é feito sobre uma árvore CSG localizada para a região ocupada pela célula.

No algoritmo que descrevemos, é fundamental o procedimento responsável pela simplificação de uma árvore CSG para a região correspondente a uma célula. Observe que agora, estamos lidando com caixas em duas dimensões e não mais em três. Para exprimir essa situação usa-se uma estrutura de dados como a definida abaixo:

```
TYPE
  Caixa2D =
    RECORD
      XMin, XMax, YMin, YMax : INTEGER
    END;
```

É importante observar que uma caixa envolvente, digamos C , provê apenas uma aproximação para o conjunto de pontos um sólido S , isto é, $C \supset S$. Sendo assim, um teste de interseção de um sólido contra uma região dada por R , se conduzido com base em sua caixa envolvente, pode nos assegurar que S é \emptyset -redundante, mas não provê informações sobre sua W -redundância, isto é,

$$\begin{aligned} C \cap R = \emptyset &\Rightarrow S \text{ é } \emptyset\text{-redundante} \\ C \supset R &\not\Rightarrow S \text{ é } W\text{-redundante} \end{aligned} \tag{V.10}$$

O teste de redundância é efetuado pela rotina *TestaRedundancia*, que recebe como parâmetros uma caixa C correspondente ao sólido que se quer testar e uma caixa R associada a uma região R da imagem. Seu pseudo-código e apresentado a seguir:

```
PROCEDURE TestaRedundancia (C, R : Caixa2D) : BOOLEAN;

  BEGIN
    RETURN
      (C.XMin > R.XMax) OR (C.YMin > R.YMax) OR
      (C.XMax < R.XMin) OR (C.YMax < R.YMin);
  END TestaRedundancia;
```

O procedimento para simplificar uma árvore A para uma região dada por uma caixa bidimensional R pode ser descrito pelo seguinte pseudo-código:

```
PROCEDURE SimplificaArvore (  
    A : ArvoreCSG; R : Caixa2D;  
    VAR NovaA : ArvoreCSG;  
    VAR C : Caixa2D  
);  
  
VAR  
    NovaEsquerda,  
    NovaDireita : ArvoreCSG;  
    CaixaEsquerda,  
    CaixaDireita : Caixa2D;  
  
BEGIN  
IF A^.Simples THEN  
    C := CaixaEnvolventeParaCaixa2D (A^.Caixa.Minimo [1]);  
    IF TestaRedundancia (C, R) THEN  
        NovaA := NIL  
    ELSE  
        NovaA := CopiaArvoreSimples (A);  
    END  
ELSE  
SimplificaArvore (A^.Esquerda, R, NovaEsquerda, CaixaEsquerda);  
SimplificaArvore (A^.Direita, R, NovaDireita, CaixaDireita);  
CASE A.Operador OF  
| Or:  
    IF (NovaEsquerda = NIL) THEN  
        IF (NovaDireita = NIL) THEN  
            NovaA := NIL  
        ELSE  
            NovaA := NovaDireita; C := CaixaDireita;  
        END  
    ELSIF (NovaDireita = NIL) THEN  
        NovaA := NovaEsquerda; C := CaixaEsquerda;  
    ELSE  
        NovaA := CriaArvoreComposta (Or, NovaEsquerda, NovaDireita);  
        C := UniaoCaixa2D (CaixaEsquerda, CaixaDireita);  
    END  
END
```

```
END;
| And:
  IF (NovaEsquerda = NIL) OR (NovaDireita = NIL) THEN
    NovaA := NIL;
  ELSE
    NovaA := CriaArvoreComposta (Or, NovaEsquerda, NovaDireita);
    C := IntersecaoCaixa2D (CaixaEsquerda, CaixaDireita);
    IF (C.XMin > C.XMax) OR (C.YMin > C.YMax) THEN
      NovaA := NIL
    END
  END;
| Comp:
  IF (NovaEsquerda = NIL) THEN
    NovaA := NIL
  ELSIF (NovaDireita = NIL) THEN
    NovaA := NovaEsquerda;
    C := CaixaEsquerda;
  ELSIF TestaRedundancia (CaixaDireita, CaixaEsquerda) THEN
    NovaA := NIL;
  ELSE
    NovaA := CriaArvoreComposta (Comp, NovaEsquerda, NovaDireita);
    C := CaixaEsquerda;
  END
END;
END
END SimplificaArvore;
```

No código acima foram usadas as seguintes rotinas auxiliares:

- *CaixaEnvolventeParaCaixa2D*: Retorna uma estrutura do tipo *Caixa2D* correspondente a uma caixa envolvente. O processo consiste em descartar os valores mínimo e máximo correspondentes à coordenada Z.
- *CopiaArvoreSimples*: Retorna uma cópia de uma árvore CSG composta de apenas uma primitiva.

- *CriaArvoreComposta*: Retorna uma nova árvore composta da união, interseção ou diferença de duas sub-árvores. O operador e ponteiros para as duas sub-árvores são passados como parâmetros.
- *IntersecaoCaixa3D* e *UniaoCaixa2D*: Retornam estruturas do tipo *Caixa2D* correspondentes, respectivamente, à interseção e união de duas caixas bidimensionais passadas como parâmetros. O algoritmo empregado é em tudo análogo ao que é feito para caixas envolventes tridimensionais.

5. Resultados e conclusões

5.1. Resultados obtidos

A fim de avaliar a eficiência do algoritmo de disparo de raios modificado e compará-lo com o algoritmo original tal como proposto por Roth, ambas as versões foram implementadas, tanto para a geração de imagens sombreadas quanto de gráficos de linha.

Os sólidos de teste foram usados nesta avaliação foram modelados com o auxílio de uma linguagem formal especialmente desenvolvida para definição de sólidos CSG. A descrição da Linguagem para Definição de Sólidos (LDS) pode ser encontrada no apêndice A. O apêndice B contém os códigos LDS dos sólidos de teste. O apêndice C contém instruções sobre a utilização dos programas que compõem o sistema de visualização de sólidos CSG. Na figura V.12 (a), (b), (c), (d) e (e) são reproduzidas as imagens dos gráficos de linha dos sólidos de teste.

Na primeira bateria de testes, procurou-se avaliar a influência da complexidade da árvore CSG na melhoria do desempenho obtida com o algoritmo modificado. Para tanto, foram gerados gráficos de linhas de quatro sólidos com diferentes números de primitivas. As imagens geradas possuem aproximadamente 256×256 pixels, e o algoritmo modificado empregou uma grade com células de 8×8 pixels. O resultado encontra-se na tabela V.1 (as medidas de tempo estão expressas em segundos).

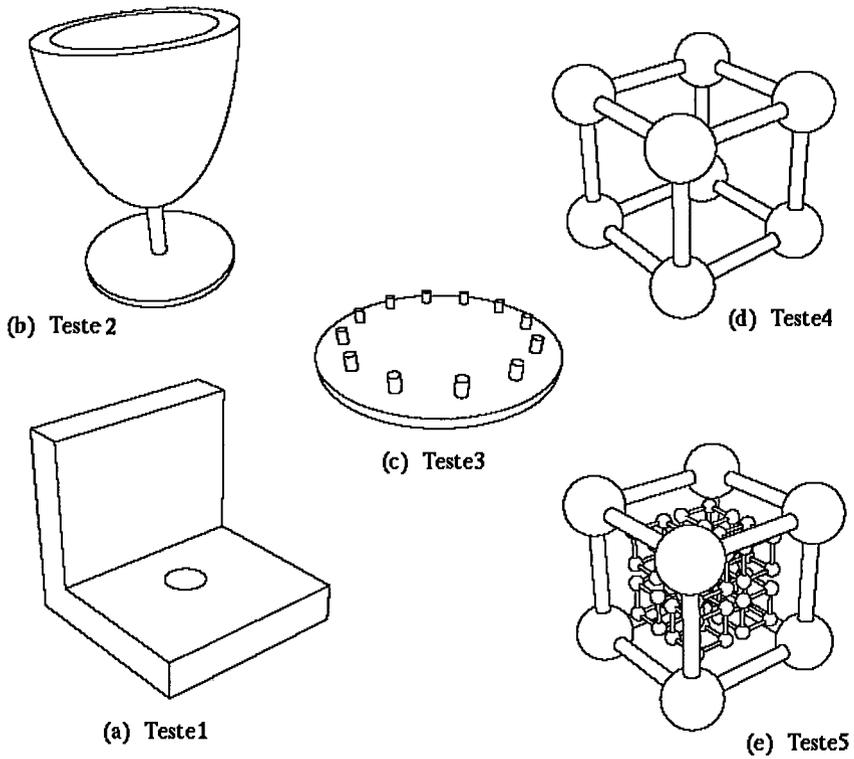


Figura V.12
Sólidos de teste

Sólido CSG	No de primitivas	Algoritmo Original	Algoritmo Modificado			Razão (Original = 1)
			Pré- proc.	Disparo de raios	Total	
Teste1	3	43.2	0.4	34.1	34.4	0.79
Teste2	5	50.1	0.9	43.4	44.3	0.88
Teste3	13	30.2	1.1	22.5	23.6	0.78
Teste4	20	70.9	2.6	41.8	44.4	0.62
Teste5	180	225.6	21.8	113.8	135.3	0.60

Tabela V.1

Pode-se observar que o algoritmo modificado foi mais eficiente que o algoritmo original, principalmente quando aplicado em sólidos complexos. Isto se justifica pelo próprio princípio que guiou a modificação, isto é, reduzir a complexidade média das árvores sobre as quais os raios são disparados.

Outro fato observado é que o incremento no desempenho é limitado pela grande parcela representada pelos testes de interseção na complexidade total do algoritmo de disparo de raios. Ou seja, apesar do algoritmo modificado eliminar muitos testes de interseção raio \times caixa envolvente, o número de raios disparados, assim como o número de testes raio \times primitiva permanecem os mesmos.

Uma segunda bateria de testes foi realizada comparando os dois algoritmos aplicados num mesmo sólido representado de maneiras diversas. O sólido *Teste3*, mostrado na figura V.12 (c), foi construído diferentemente através de três árvores CSG com altura progressivamente maior. Também nestes testes foram gerados gráficos de linha com aproximadamente 256×256 pixels e subdivisões de 8×8 pixels. O resultado é ilustrado na tabela V.2 abaixo:

Sólido CSG	Altura da árvore	Algoritmo original	Algoritmo modificado	Razão (Original = 1)
Teste3_A	4	30.4	23.5	0.77
Teste3_B	8	40.8	23.7	0.58
Teste3_C	14	57.3	23.6	0.41

Tabela V.2

Confirmando nossas previsões, percebe-se que o algoritmo original é bastante dependente da estrutura da árvore. Por outro lado, o processo de localização realizado no algoritmo modificado o torna menos sensível à maneira pela qual o sólido é construído.

A terceira bateria de testes foi realizada com o intuito de analisar a influência do tamanho da imagem no comportamento dos dois algoritmos. O mesmo sólido (*Teste5*) foi visualizado gerando três imagens com $k \times k$ pixels, para $k = 128, 256$ e 512 . Nas três imagens utilizou-se uma grade de subdivisão equivalente - desta maneira, quisemos isolar o efeito do processo de subdivisão nos tempos obtidos. Os resultados são apresentados na tabela V.3.

Tam. da Imagem	n	No de Raios	Alg. Orig.	Algoritmo Modificado			Razão (Orig.=1)
				Pré-proc.	Disparo de raios	Total	
128	4	7961	87.2	21.6	43.5	65.1	0.74
256	8	20895	224.2	21.6	111.2	132.8	0.59
512	16	53431	560.6	21.6	273.9	295.5	0.52

Tabela V.3

Como era de se esperar, a melhoria introduzida pelo algoritmo modificado independe do tamanho da imagem gerada. Com efeito, se desprezarmos o tempo gasto no pré-processamento da árvore, isto é, se calcularmos as razões *tempo gasto em disparo de raios pelo algoritmo modificado* \times *tempo gasto pelo algoritmo original*, obteremos sempre valores próximos de 0.5 (para este sólido de teste em particular). Este teste também confirma a estrita relação existente entre o tempo gasto e o número de raios disparados.

Um outro ponto analisado é como o comportamento do algoritmo modificado varia em função do tamanho da célula da grade de subdivisão. Com esse objetivo, foi realizada uma bateria de testes onde o mesmo sólido (*Teste4*) foi pré-processado segundo malhas progressivamente mais grosseiras. A imagem gerada em todos testes foi de 256×256 pixels e os resultados são listados na tabela V.4.

n	Pré- processamento	Disparo de raios	Total
4	10.4	40.8	51.2
8	2.6	41.3	43.9
12	1.2	41.8	43.0
16	0.7	42.3	42.9
20	0.5	42.9	43.4
24	0.3	44.2	44.5
28	0.3	45.6	45.9
32	0.2	46.3	46.5

Tabela V.4

Pelo comportamento geral desses números, podemos inferir que há certamente um valor "ótimo" para o tamanho da célula ao se aplicar o algoritmo de visualização em um determinado sólido. Entretanto, há uma faixa bem larga para o valor de n onde a variação do desempenho é pequena. Com efeito, ao aumentar muito o valor de n , o algoritmo modificado tende a se comportar como o algoritmo original, e, diminuindo-se demasiadamente o valor de n , o tempo dispendido no cômputo das localizações passa a pesar mais do que o ganho obtido no disparo de raios. Isto pode ser mais bem observado na figura V.13.

A localização da faixa de valores de n para a qual se obtém melhor desempenho depende da complexidade do objeto, do tamanho da imagem, do tamanho relativo das primitivas em relação a imagem e de uma série de outros fatores. Para os objetos CSG utilizados nos testes, observou-se que, usando uma grade de subdivisão de aproximadamente 32 x 32 células, obtém-se desempenhos próximos do ótimo.

5.2. Conclusões

O algoritmo modificado mostrou ser vantajoso em virtualmente todos os casos. A melhoria no desempenho mostrou de maneira geral ser maior para objetos mais complexos. A influência do tamanho da imagem no ganho obtido restringe-se à contribuição do tempo dispendido no cômputo das subdivisões no tempo total de

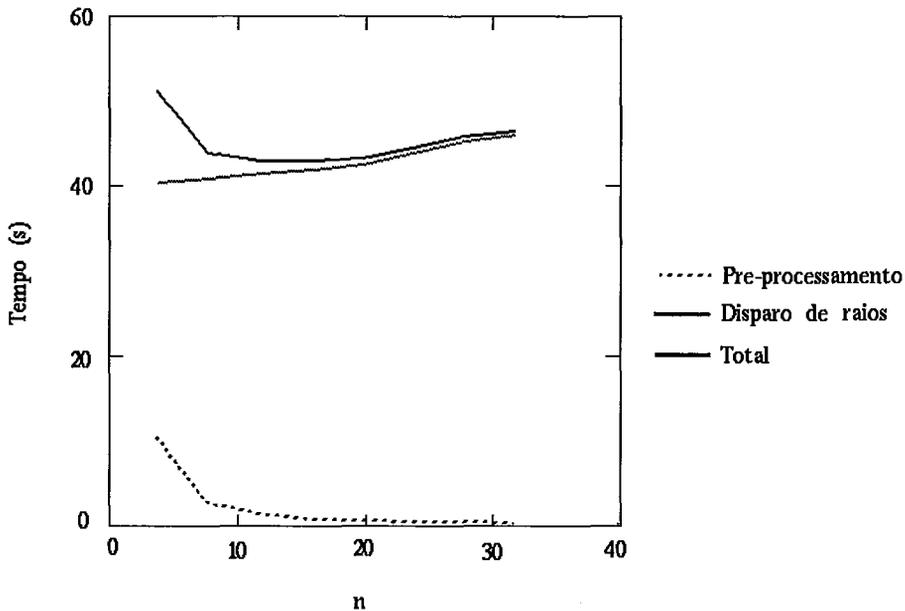


Figura V.13

Variação do desempenho em função de n

visualização. O algoritmo modificado demonstrou ser especialmente vantajoso na medida que é pouco influenciado pela forma com que os objetos foram modelados, ao contrário do que acontece com o algoritmo de disparo de raios tradicional.

Um refinamento que pode ser introduzido no algoritmo é um método pelo qual o processo de subdivisão não mais se dê para uma grade de tamanho fixo, mas que se adapte às condições de complexidade do objeto e de tamanho da imagem. A divisão do espaço da imagem à maneira de uma *quadtree* pode ser a solução, desde que critérios apropriados de subdivisão sejam usados, isto é, um fator fundamental para a implementação da idéia é saber quando vale a pena subdividir um quadrante da imagem.

Deve-se acrescentar que as idéias incorporadas no algoritmo modificado podem ser aplicadas em outras aplicações onde caixas envolventes são utilizadas no tratamento de sólidos CSG. Por exemplo, nos parece que essas idéias podem ser adaptadas em algoritmos de avaliação de fronteiras, ou mesmo em sistemas de geração de imagens realistas que usem Rastreamento de Raios (*Ray-Tracing*).

Capítulo VI

Visualização através do uso intensivo de subdivisão espacial

Como vimos no capítulo IV, o uso de subdivisão espacial provê uma abordagem para o problema de visualização de modelos construtivos. Veremos neste capítulo uma solução apresentada por KOISTINEN, TAMMINEN e SAMET [30] que emprega quase que exclusivamente o paradigma da subdivisão espacial. Ao final, introduziremos idéias para o aprimoramento desse algoritmo e apresentaremos resultados de experimentos onde algumas dessas conjecturas foram postas à prova.

1. Visualização de modelos CSG através de conversão para Bintrees

O algoritmo proposto por KOISTINEN et alii baseia-se na conversão parcial de modelos CSG compostos de semiespaços planos para um tipo de modelo de decomposição espacial denominado *bintree*.

O processo de conversão baseia-se no conceito de subdivisão recursiva, sendo que os autores desenvolvem um método bastante econômico para classificar semiespaços planos contra regiões dadas por paralelepípedos.

1.1. Modelo CSG composto de semiespaços planos

O algoritmo em questão é aplicável apenas a um tipo muito particular de representação CSG. Nesse modelo, a única primitiva empregada é o semiespaço plano, isto é, a primitiva que corresponde a todos os pontos (x_1, x_2, x_3) do universo E^3 que satisfazem uma inequação da forma

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4 \geq 0 \quad (\text{VI.1})$$

A inequação V.1 pode ser reescrita, se considerarmos os pontos x em E^3 expressos em coordenadas homogêneas, como:

$$a \cdot x \geq 0 \tag{VI.2}$$

onde

$$a = [a_1 \ a_2 \ a_3 \ a_4] \quad \text{e} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$$

Estaremos considerando, como de hábito, operações regularizadas de conjunto, e portanto o semiespaço denotado pela equação VI.2 acima é tida como equivalente a $a \cdot x > 0$.

Neste tipo particular de árvore CSG não há necessidade de nós internos associados a operações de diferença uma vez que, para dois conjuntos A e B vale a relação

$$A \setminus B = A \cap (\neg B) \tag{VI.3}$$

e o complemento de um semiespaço plano pode ser facilmente obtido trocando-se o sinal dos coeficientes a_1 , a_2 , a_3 e a_4 que o definem.

Bintrees

Vimos no capítulo IV que há algoritmos de visualização direta de sólidos CSG que empregam um modelo de subdivisão regular do espaço. Em se tratando do espaço euclidiano tridimensional, talvez o modelo mais popular seja o que conhecemos como *octree*, onde uma região do espaço dada por um cubo de lado l é dividida em oito subregiões, cada uma correspondendo a um cubo de lado $l/2$. Analogamente, o modelo de decomposição mais conhecido para espaços bidimensionais é conhecido como *quadtree*, onde quadrados de lado l são partidos em quatro quadrados de lado $l/2$.

Um modelo alternativo de divisão do espaço é o conhecido como *bintree*, idealizado por SAMET e TAMMINEN [39], e que possui a particularidade de ser aplicável a espaços com qualquer dimensão inteira n . Segundo esse modelo, a cada passo, uma região é partida em duas subregiões por um hiperplano perpendicular a um dos eixos coordenados.

Tomemos por exemplo, uma região de E^2 dada por $x_{\min} \leq x \leq x_{\max}$ e $y_{\min} \leq y \leq y_{\max}$. Em duas dimensões, um hiperplano corresponde a uma reta, portanto, podemos partir

essa região em duas subregiões de mesma área por uma reta dada por $x = (x_{\min} + x_{\max}) / 2$. Por sua vez, cada uma das duas subregiões formadas podem ser divididas pela reta $y = (y_{\min} + y_{\max}) / 2$. Após esses passos, obtemos quatro subregiões de mesma área, semelhantes às que obteríamos após uma subdivisão de uma quadtree.

Se considerarmos uma região tridimensional dada por $x_{\min} \leq x \leq x_{\max}$, $y_{\min} \leq y \leq y_{\max}$ e $z_{\min} \leq z \leq z_{\max}$, as subdivisões seriam obtidas por planos $x = (x_{\min} + x_{\max}) / 2$, $y = (y_{\min} + y_{\max}) / 2$ e $z = (z_{\min} + z_{\max}) / 2$. Teríamos com isso, oito subregiões à maneira de uma subdivisão de uma octree. O processo é ilustrado na figura VI.1.

Como se vê, podemos pensar no esquema de subdivisão espacial por bintrees como sendo uma generalização de quadtrees e octrees.

Uma implementação possível para bintrees é mostrada no pseudo-código abaixo:

```
TYPE
  BinTree =
    POINTER TO NoBinTree;
  TipoNoBinTree =
    (Interno, Cheio, Vazio);
  NoBinTree =
    RECORD
      CASE Tipo : TipoNoBinTree OF
        | Interno :
          Esquerda,
          Direita : BinTree
      END
    END;
```

1.2. Subdivisão controlada pela projeção

No algoritmo de visualização de sólidos CSG proposto por KOISTINEN et alii [30], o espaço de interesse é partido segundo uma bintree de dimensão 3.

A bintree de dimensão 3 não é calculada explicitamente, em vez disso, o algoritmo divide o espaço recursivamente em busca de regiões que contenham uma porção do sólido suficientemente simples para ser pintada diretamente sobre o plano de projeção.

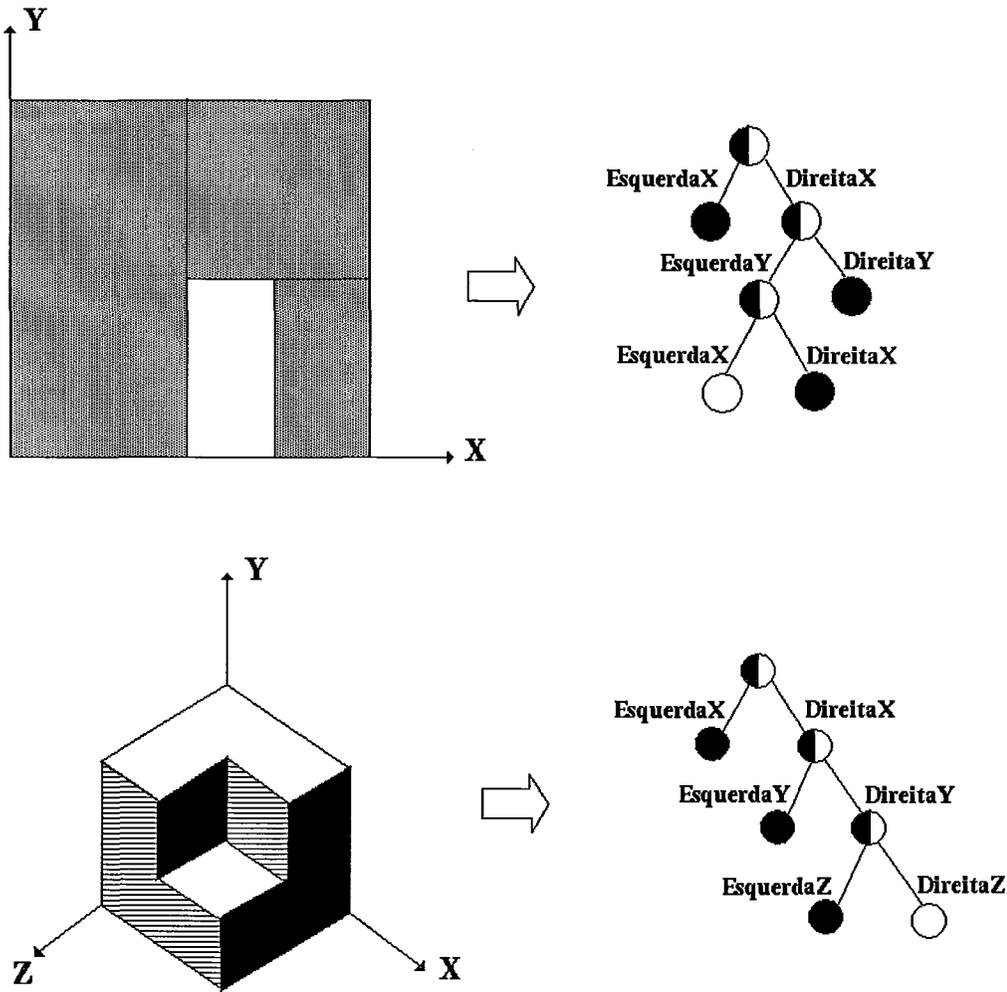


Figura VI.1
Subdivisão em bintrees

Uma segunda bintree, de dimensão 2, é empregada para manter controle sobre a porção da imagem projetada computada até o momento, isto é, as folhas "cheias" da bintree representam áreas do plano de projeção já pintadas.

O algoritmo emprega um sistema de coordenadas orientado pela regra da mão *esquerda*, com o plano de projeção situado sobre o plano $X-Y$ e o eixo Z "entrando" no plano. Uma vez que o procedimento de divisão recursiva é montado de forma a percorrer o espaço do universo de interesse sempre de frente para trás, isto é, segundo

valores crescentes de z , subdivisões equivalentes a folhas da bintree de dimensão 3 que se projetem sobre áreas "cheias" da bintree de dimensão 2 não precisam ser visitadas pelo procedimento de divisão recursiva. A figura VI.2 ilustra essa situação.

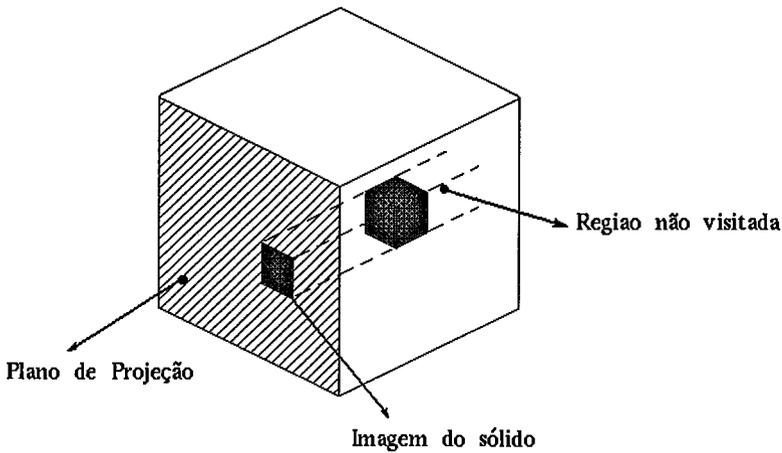


Figura VI.2
Subdivisão controlada pela projeção

1.3. Classificação dos semiespaços contra as subdivisões

Como vimos anteriormente, é fundamental ao processo de subdivisão o teste de uma primitiva P contra uma região R com vistas a determinar

$$P \cap^* R = \emptyset \text{ ou } W \tag{VI.4}$$

No nosso caso, as regiões R têm a forma de paralelepípedos cujas faces são perpendiculares aos eixos coordenados e as primitivas P são semiespaços planos. Nestas circunstâncias, o citado teste equivale a determinar se o paralelepípedo está todo contido na região do semiespaço ou está todo fora dessa região. Para tanto, basta classificar cada um dos 8 vértices do paralelepípedo com relação ao semiespaço. Se

todos eles estiverem do mesmo lado do plano que delimita o semiespaço, valerá a condição da equação VI.4. Caso contrário, o plano secciona o paralelepípedo. Essas três situações são ilustradas na figura VI.3.

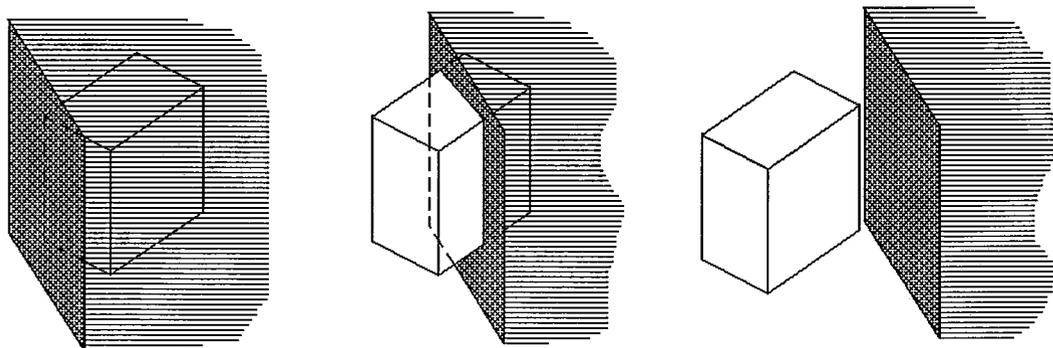


Figura VI.3

Teste de um semiespaço plano contra um paralelepípedo

A classificação de um ponto x contra um semiespaço pode ser determinada avaliando-se naquele ponto a forma linear que define o semiespaço, ou seja,

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4 = a \cdot x \begin{cases} \geq 0 \Rightarrow P \text{ dentro do semiespaço} \\ < 0 \Rightarrow P \text{ fora do semiespaço} \end{cases} \quad (\text{VI.5})$$

A fase preparatória do algoritmo consiste em efetuar a classificação de cada um dos semiespaços que compõem o sólido a ser visualizado contra o cubo que define o universo de interesse. Qualquer semiespaço que atenda à condição da equação VI.4 é substituído na árvore por \emptyset ou W conforme o caso e a árvore é então simplificada segundo a tabela IV.1. Neste ponto portanto, todos os semiespaços remanescentes na árvore interceptam o cubo do universo.

O processo de subdivisão recursiva é então aplicado, dividindo o cubo universo sucessivamente por planos perpendiculares aos eixos x , y e z . A cada passo é necessário voltar a determinar a classificação de cada semiespaço com relação aos dois paralelepípedos resultantes da subdivisão.

A tarefa é facilitada se observarmos que só é preciso avaliar a classificação de dois dos vértices diagonalmente opostos de cada paralelepípedo, a saber, aqueles para os quais $a \cdot x$ possui os valores máximo e mínimo. Para cada plano, a posição desses vértices em relação a qualquer paralelepípedo cujas faces estejam alinhadas com os eixos coordenados é sempre a mesma. No algoritmo, os valores $vmax$ e $vmin$ correspondentes às classificações desses dois vértices são calculados para cada um dos semiespaços que compõem o sólido e armazenados na estrutura de dados das folhas correspondentes na árvore CSG.

Sempre que um nó da bintree de dimensão 3 é subdividido, o valor $vmax$ ou $vmin$ (nunca ambos simultaneamente) de cada filho muda com respeito ao do nó pai. Isto se explica pelo fato de $vmin$ e $vmax$ corresponderem via-de-regra a vértices do paralelepípedo em extremidades opostas de uma de suas diagonais. (Tal não se dá apenas quando um dos valores a_1 , a_2 ou a_3 é igual a 0; nesse caso a $vmin$ e $vmax$ podem corresponder 2 ou mais vértices, e portanto haverá sempre dois vértices - um associado a $vmin$ e outro a $vmax$ opostos por uma das diagonais do paralelepípedo).

Consideremos uma subdivisão correspondente a um paralelepípedo cujas dimensões em cada um dos eixos coordenados são respectivamente w_1 , w_2 e w_3 . Então, se dividirmos esse paralelepípedo ao meio por um plano perpendicular ao eixo coordenado i , a variação no valor de $vmax$ ou $vmin$ será de

$$\delta_i = \frac{a_i w_i}{2} \quad (\text{VI.6})$$

Por conveniência, estruturamos a bintree de forma que os filhos à esquerda de um nó dividido por um plano perpendicular à coordenada i correspondam aos menores valores de x_i e os à esquerda, aos maiores. Portanto, para o filho à esquerda, se $\delta_i > 0$, então δ_i é subtraído de $vmax$, senão δ_i é subtraído de $vmin$. De maneira similar, para o filho à direita, se $\delta_i > 0$, $vmax$ é incrementado de δ_i , caso contrário δ_i é adicionado a $vmin$.

Com o propósito de acelerar o processo de subdivisão, durante a fase de pré-processamento, os incrementos δ_i associados a cada semiespaço são calculados e armazenados num vetor dentro da estrutura de dados de cada folha da árvore CSG.

O nível máximo a que se permite a subdivisão chegar, e conseqüentemente o comprimento do vetor de incrementos δ_i , é determinado pela resolução da imagem que se quer gerar. Para uma imagem de $n \times n$ pixels, a bintree de dimensão 2 possuirá uma altura máxima igual a $2 \log_2 n$ e a subdivisão poderá chegar, no máximo, a uma profundidade de $3 \log_2 n$.

Abaixo apresentamos um pseudo-código para descrever as estruturas de dados apresentadas até o momento.

CONST

MaxSubdivisao = $3 * \log_2 n$;

TYPE

IndiceSubdivisao =

[1 .. MaxSubdivisao];

Primitiva =

RECORD

SemiEspaco : ARRAY [1 .. 4] OF REAL;

Incremento : ARRAY [IndiceSubdivisao] OF REAL;

END;

ArvoreCSG =

POINTER TO NoCSG;

TipoArvoreCSG =

(NoInterno, Folha, Nula, Universo);

NoCSG =

RECORD

CASE Tipo : TipoArvoreCSG OF

| Folha :

Prim : POINTER TO Primitiva;

VMin,

VMax : REAL;

| NoInterno :

Operador : (Uniao, Intersecao);

Esquerda,

Direita : Arvore;

END;

Com as estruturas de dados assim definidas, podemos montar uma rotina para computar a localização de um sólido após a subdivisão de uma região do espaço. O pseudo-código a seguir implementa este cômputo, onde

- *Nivel* corresponde ao nível da subdivisão sendo computada.
- *ArvoreFolha* aponta para um nó folha, isto é, um nó contendo uma primitiva.

- *DirecaoEsquerda* é um valor lógico que diz se a subdivisão que estamos computando corresponde a um filho à esquerda na bintree tridimensional.
- *ArvoreNula* e *ArvoreUniverso* são árvores CSG especiais criadas no começo do programa e que apontam para um nó onde o campo *Tipo* vale respectivamente, *Nula* e *Universo*.
- *CriaFolha* é uma rotina auxiliar que retorna um ponteiro para um nó do tipo *Folha* com seus campos *Prim*, *VMin* e *VMax* inicializados com os valores que recebe como parâmetros.

```
PROCEDURE AvaliaSemiespaco (  
    Nivel : IndiceSubdivisao;  
    ArvoreFolha : ArvoreCSG;  
    DirecaoEsquerda : BOOLEAN  
) : ArvoreCSG;  
  
VAR  
    NovoVMin, NovoVMax : REAL;  
    Delta : REAL;  
  
BEGIN  
    Delta := ArvoreFolha^.Prim^.Incremento [Nivel];  
    NovoVMin := ArvoreFolha^.VMin;  
    NovoVMax := ArvoreFolha^.VMax;  
    IF DirecaoEsquerda THEN  
        IF Delta > 0.0 THEN  
            NovoVMax := NovoVMax - Delta  
        ELSE  
            NovoVMin := NovoVMin - Delta  
        END  
    ELSE  
        IF Delta > 0.0 THEN  
            NovoVMin := NovoVMin + Delta  
        ELSE  
            NovoVMax := NovoVMax + Delta;  
        END  
    END;  
    IF NovoVMax < 0.0 THEN  
        RETURN ArvoreNula  
    ELSIF NovoVMin > 0.0 THEN  
        RETURN ArvoreUniverso  
    ELSE  
        RETURN CriaFolha (ArvoreFolha^.Prim, NovoVMin, NovoVMax)  
    END  
END AvaliaSemiespaco;
```

1.4. Subdivisão e simplificação de uma árvore CSG

O procedimento *AvaliaSemiespaco* representa o passo fundamental para obter a localização de uma árvore CSG de semiespaços planos em uma das subregiões obtidas durante uma etapa de subdivisão. A árvore localizada é construída aplicando esse procedimento em cada um dos nós folha da árvore original.

Para os casos onde *AvaliaSemiespaco* retorna *ArvoreNula* ou *ArvoreUniverso* é efetuado um processo de simplificação que consiste em percorrer a árvore original em pós-ordem e aplicar as regras definidas na tabela IV.1. Esse procedimento pode ser expresso pelo seguinte pseudo-código:

```
PROCEDURE SimplificaArvore (  
    Nivel : IndiceSubdivisao;  
    Arvore : ArvoreCSG;  
    DirecaoEsquerda : BOOLEAN  
): ArvoreCSG;  
  
VAR  
    Esq, Dir : ArvoreCSG;  
BEGIN  
    CASE Arvore^.Tipo OF  
    | Nula: RETURN ArvoreNula;  
    | Universo: RETURN ArvoreUniverso;  
    | Folha : RETURN AvaliaSemiEspaco (Nivel, Arvore, DirecaoEsquerda);  
    | NoInterno:  
        Esq := SimplificaArvore (Nivel, Arvore^.Esquerda, DirecaoEsquerda);  
        Dir := SimplificaArvore (Nivel, Arvore^.Direita, DirecaoEsquerda);  
        CASE Arvore^.Operador OF  
        | Uniao:  
            IF (Esq = ArvoreUniverso) OR (Dir = ArvoreUniverso) THEN  
                RETURN ArvoreUniverso;  
            ELSIF Esq = ArvoreNula THEN  
                RETURN Dir;  
            ELSIF Dir = ArvoreNula THEN  
                RETURN Esq;  
            ELSE  
                RETURN CriaArvoreComposta (Uniao, Esq, Dir);  
            END;  
        | Intersecao:  
            IF (Esq = ArvoreNula) OR (Dir = ArvoreNula) THEN  
                RETURN ArvoreNula;  
            ELSIF Esq = ArvoreUniverso THEN  
                RETURN Dir;  
            ELSIF Dir = ArvoreUniverso THEN  
                RETURN Esq;  
            ELSE  
                RETURN CriaArvoreComposta (Intersecao, Esq, Dir);
```

```
END;  
END;  
END;  
END SimplificaArvore;
```

Os parâmetros de *SimplificaArvore* são análogos aos de *AvaliaSemiEspaco*. A rotina *CriaArvoreComposta* retorna um ponteiro para um novo nó interno cujos campos são inicializados com valores recebidos como parâmetros.

Uma observação importante: Os pseudo-códigos apresentados têm o objetivo de descrever estruturas de dados e algoritmos de forma esquemática. Por isso mesmo, muitos detalhes de implementação são omitidos. Assim, no pseudo-código da rotina *SimplificaArvore*, as regras de simplificação que discutimos no capítulo IV foram transpostas quase que literalmente. Numa implementação cuidadosa, o código seria bem mais elaborado - veja, por exemplo, que se consideramos uma operação de interseção e o resultado da avaliação da sub-árvore da esquerda é *ArvoreNula*, então não é necessário avaliar a sub-árvore da direita.

1.5. Pré-processamento do sólido

Vamos agora descrever mais detalhadamente os procedimentos a serem efetuados antes de se iniciar o processo de subdivisão recursiva. Em primeiro lugar, estabelecemos os limites do nosso universo de interesse. Estes correspondem ao volume delimitado pelos planos

$$0 \leq x, y, z \leq 1 \quad (\text{VI.7})$$

Esta imposição implica em que o sólido será visualizado recortado por esses planos. Consequentemente, caso estes limites não sejam adequados para o propósito do usuário, este deverá então determinar um outro volume de recorte - o sistema aplicará então uma transformação linear afim (obtida pela composição de operações de rotação/translação/escala apropriadas) sobre o sólido de maneira a posicioná-lo dentro do universo de interesse. Este procedimento é aplicado também quando o usuário deseja um sistema de projeção diferente daquele usado implicitamente pelo algoritmo.

No caso particular no qual o usuário deseje uma vista em perspectiva, uma única transformação linear não pode ser aplicada a todos os semiespaços planos que compõem o sólido. Entretanto, é possível calcular uma transformação linear que leva cada um dos semiespaços originais para o espaço deformado pela projeção perspectiva.

Para tanto, escolhe-se três pontos no plano original e calcula-se os pontos correspondentes no plano transformado. Se considerarmos um sistema projetivo onde o plano de projeção é dado por $z = 0$ e o centro de projeção encontra-se sobre o eixo z em $z = z_0, z_0 < 0$, então um ponto P no plano original é levado em sua contrapartida P' no plano transformado pela equação

$$\begin{bmatrix} x'_P & y'_P & z'_P \end{bmatrix} = \begin{bmatrix} \frac{x_P z_0}{z_0 - z_P} & \frac{y_P z_0}{z_0 - z_P} & z_P \end{bmatrix} \quad (VI.8)$$

basta então calcular a equação do plano que passa pelos três pontos transformados.

Uma vez posicionados adequadamente os semiespaços dentro do universo de interesse são acrescentados à árvore CSG dois semiespaços adicionais correspondentes aos dois planos de recorte do volume de vista, isto é, $z \geq 0$ e $z \leq 1$.

Cada um dos semiespaços é então classificado em relação ao universo de interesse. Como vimos anteriormente, isto é feito determinando-se o valor mínimo e máximo de $a \cdot x$ quando se substitui o vetor x pelas coordenadas de cada um dos vértices do cubo unitário $0 \leq x, y, z \leq 1$. É imediato ver que neste caso $vmin$ é obtido somando-se a_4 aos coeficientes $a_i, i \leq 3$ tais que $a_i \leq 0$. Similarmente, $vmax$ é obtido somando-se a a_4 os demais coeficientes positivos. Resumindo, temos:

$$\begin{aligned} vmin &= a_4 + \sum_{i=1}^3 a_i \cdot sign(a_i) \\ vmax &= a_4 + \sum_{i=1}^3 a_i \cdot sign(-a_i) \end{aligned} \quad (VI.9)$$

onde

$$sign(x) = \begin{cases} 0 & \text{se } x \geq 0 \\ 1 & \text{se } x < 0 \end{cases}$$

Em seguida, com base nos valores de $vmin$ e $vmax$, a árvore é simplificada substituindo-se por \emptyset aqueles semiespaços para os quais $vmax < 0$ e por W aqueles para os quais $vmin \geq 0$ e aplicando-se as regras constantes da tabela IV.1. Para cada um dos semiespaços remanescentes é então calculada uma tabela de incrementos δ_i correspondentes a cada nível de subdivisão, isto é, para uma imagem de $n \times n$ pixels (onde n é uma potência de 2) temos,

$$\delta_i = a_j \cdot \left(\frac{1}{2}\right)^{(i-1) \div 3}, \quad 1 \leq i \leq 3 \cdot \log_2 n \quad (VI.10)$$

onde

$$j = \begin{cases} i, & \text{se } i \bmod 3 \neq 0 \\ 0, & \text{se } i \bmod 3 = 0 \end{cases}$$

São calculadas ainda duas tabelas de incrementos - $incx$ e $incy$. Para cada passo i no processo de subdivisão recursiva $incx_i$ e $incy_i$ correspondem ao deslocamento no espaço da imagem do vértice inferior esquerdo do filho à direita em relação àquele do pai. Assumindo para a imagem um sistema de coordenadas no qual $(0,0)$ corresponde ao vértice inferior esquerdo da imagem e $(n-1,n-1)$ ao vértice superior direito, então, para i variando como na equação VI.10 temos

$$incx_i = \begin{cases} \frac{n}{2^{(i-1) \div 3}}, & \text{se } i \bmod 3 = 0 \\ 0, & \text{se } i \bmod 3 \neq 0 \end{cases} \quad \text{e} \quad incy_i = \begin{cases} \frac{n}{2^{(i-1) \div 3}}, & \text{se } i \bmod 3 = 1 \\ 0, & \text{se } i \bmod 3 \neq 1 \end{cases} \quad (\text{VI.11})$$

O significado das equações VI.11 é que as subdivisões de nível 0, 3, 6, etc, são feitas por planos perpendiculares ao eixo x ; conseqüentemente, se a área correspondente à projeção de uma dada região do espaço tem seu vértice inferior direito em (x,y) , então a subregião à direita do plano de corte está associada a uma área do plano de projeção com vértice inferior esquerdo em $(x+n/2,y)$, $(x+n/4,y)$, $(x+n/8,y)$, etc. Para subdivisões de nível k , tal que $k \bmod 3 = 1$, os deslocamentos se dão na direção y . Se $k \bmod 3 = 2$, as subdivisões são feitas por planos perpendiculares a z , e portanto ambas as subregiões se projetam sobre a mesma área que a região do nó pai.

Podemos sumarizar os procedimentos preparatórios através do seguinte pseudo-código:

```
VAR
  Raiz : Arvore;
  IncX, IncY : ARRAY [1 .. MaxNivelSubdivisao] OF INTEGER;

PROCEDURE PreProcessamento;
BEGIN
  (* Ler a árvore CSG Raiz *)
  (* Transformar os nós Folha da árvore segundo a projeção *)
  (* Calcular os vetores IncX e IncY *)
  (* Classificar os nós Folha com respeito ao cubo do universo *)
  (* Simplificar a árvore *)
END PreProcessamento;
```

1.6. O algoritmo de visualização

O algoritmo baseia-se em um único procedimento recursivo que é encarregado de computar uma parte da imagem de um sólido. Esse trecho da imagem é correspondente à porção do sólido que faz interseção com a região do espaço sendo avaliada. Ao procedimento são passados os seguintes parâmetros:

- *X e Y*: coordenadas do vértice inferior esquerdo da área onde a região do espaço é projetada.
- *Nivel*: nível de subdivisão.
- *Arvore*: árvore CSG correspondente a uma localização do sólido na região.
- *Projecao*: bintree que descreve a imagem já computada do sólido na área de projeção da região.

De posse desses dados, o procedimento precisa discernir uma das seguintes situações:

- *Arvore* é nula, ou *Projecao* corresponde a um nó do tipo *Cheio*, isto é, toda a área da imagem ocupada pela projeção da região já foi computada anteriormente. Neste caso, nada mais precisa ser feito.
- A subdivisão chegou ao nível máximo, e conseqüentemente a área da imagem sendo computada corresponde a um pixel. Temos então duas hipóteses: (1) a árvore contém um único semiespaço; neste caso é computada a cor correspondente e o pixel é pintado, ou (2) a árvore é composta; neste caso é necessário utilizar um mecanismo de disparo de raios para avaliar a cor do pixel. No caso (2), um raio paralelo ao eixo z é disparado na direção do centro do pixel buscando com isso detectar o primeiro dos semiespaços atingidos. Observe que, eventualmente, esse raio pode não tocar nenhum dos semiespaços da árvore, por exemplo, quando há dois semiespaços que não se tocam e com coeficientes a_3 nulos.
- Nenhuma das condições acima foi atendida. Neste caso, a região é dividida novamente em duas, as localizações de *Arvore* são calculadas para as duas subregiões formadas e o procedimento é invocado recursivamente.

A rotina *Visualiza* provê uma descrição em pseudo-código para esse procedimento:

```
PROCEDURE Visualiza (  
    X, Y : INTEGER;  
    Nivel : IndiceSubdivisao;  
    Arvore : ArvoreCSG;  
    VAR Projecao : BinTree;  
);  
  
VAR  
    PrimitivaInterceptada : POINTER TO Primitiva;  
    Esq, Dir : ArvoreCSG;  
  
BEGIN  
    IF (Projecao^.Tipo = Cheio) OR (Arvore = ArvoreNula) THEN  
        RETURN;  
    ELSIF Nivel = MaxSubdivisao THEN  
        IF Arvore^.Tipo = Folha THEN  
            PintaPixel (X, Y, Cor (Arvore^.Prim));  
            Projecao^.Tipo := Cheio;  
        ELSE  
            PrimitivaInterceptada := DisparaRaio (X, Y, Arvore);  
            IF PrimitivaInterceptada <> NIL THEN  
                PintaPixel (X, Y, Cor (PrimitivaInterceptada));  
                Projecao^.Tipo := Cheio;  
            END  
        END  
    ELSE  
        Esq := SimplificaArvore (Nivel, Arvore, TRUE);  
        Dir := SimplificaArvore (Nivel, Arvore, FALSE);  
        IF Nivel MOD 3 = 0 THEN (* Subdivisao em Z *)  
            Visualiza (X, Y, Nivel+1, Esq, Projecao);  
            Visualiza (X, Y, Nivel+1, Dir, Projecao);  
        ELSE (* Subdivisao em X ou Y *)  
            IF Projecao^.Tipo = Vazio THEN Divide (Projecao) END;  
            Visualiza (X, Y, Nivel, Nivel+1, Esq, Projecao^.Esquerda);
```

```
Visualiza (X+IncX[Nivel], Y+IncY[Nivel],
          Nivel+1, Dir, Projecao^.Direita);
IF (Projecao^.Esquerda^.Tipo = Cheio) AND
   (Projecao^.Direita^.Tipo = Cheio) THEN
  Junta (Projecao);
END;
END
END
END Visualiza;
```

As seguintes rotinas auxiliares foram usadas:

- *Cor*: Aplica o modelo de iluminação à primitiva passada como parâmetro. Retorna um inteiro correspondente à cor apropriada.
- *PintaPixel*: Rotina para pintar o pixel na posição (X,Y) da tela com uma cor dada.
- *DisparaRaio*: Dados a posição (X,Y) do plano de projeção e uma árvore CSG, retorna a primeira primitiva interceptada por um raio paralelo ao eixo z disparado na direção do ponto (X,Y). Caso nenhuma primitiva seja interceptada, retorna NIL.
- *Divide*: Subdivide um nó do tipo *Vazio* da bintree, colocando em seu lugar um nó interno com dois filhos vazios.
- *Junta*: Substitui um nó interno da bintree que aponta para dois filhos cheios por um único nó do tipo *Cheio*.

1.7. Análise do algoritmo

A análise do algoritmo de visualização de sólidos CSG pode ser simplificada se observarmos que, em termos de complexidade, o algoritmo resume-se na conversão de uma representação CSG para uma representação de decomposição do espaço. Em verdade, o algoritmo apresentado pode, com pequenas modificações, ser adaptado para avaliar, digamos, uma representação octree do sólido; para tanto, ignora-se os testes que impedem a visitação de subregiões "escondidas" atrás de áreas cheias da imagem.

Ao avaliar apenas as regiões que contribuem para a imagem, o algoritmo de visualização realiza, de modo geral, "metade" do trabalho do algoritmo de conversão -

a outra "metade" do trabalho seria realizada, por exemplo, se quiséssemos obter uma imagem do sólido visto pelo observador posicionado sobre o semi-eixo $z < 0$ (estamos falando, é claro, do caso médio).

SAMET [4] (pp. 352-355) aponta em sua análise do algoritmo de conversão de sólidos CSG para bintrees, que a complexidade é proporcional à soma dos tamanhos das árvores CSG localizadas para cada nó visitado da bintree tridimensional. Embora o algoritmo, a cada subdivisão, procure reduzir o tamanho da árvore através do processo de simplificação, esse número pode ser bastante grande. O fato que a árvore simplificada precisa ser copiada a cada nível de subdivisão não afeta a complexidade do algoritmo, embora seja possível evitar o trabalho de cópia através de uma programação mais cuidadosa. Num caso típico, entretanto, à medida que refinamos a subdivisão, muitos dos nós da árvore CSG são descartados, e podemos nesse caso concentrar nossa análise sobre as subregiões de tamanho mínimo.

Na discussão que se segue estaremos considerando subregiões (voxels) com volume próximo de zero, isto é, tendendo a um ponto. Num objeto poliedral, cada um de seus vértices é formado pelo encontro de pelo menos três semiespaços; cada aresta é formada pelo encontro de exatamente dois semiespaços, e em cada face, apenas um semiespaço plano se encontra ativo. Podemos então estimar o número total de semiespaços ativos enumerando os voxels que interceptam vértices, arestas e faces do objeto poliedral.

O número de voxels que interceptam faces é proporcional à área do objeto (MEAGHER [47]), enquanto que o número total de voxels que interceptam arestas é proporcional aos comprimentos das arestas na resolução dada (HUNTER [48] e [49]). O número de voxels que contêm vértices é limitado pelo número total de vértices, independentemente da resolução, isto é, do nível de subdivisão. Assumindo uma resolução de n , o número de voxels contendo mais que um semiespaço cresce em $O(2^n)$, enquanto que o número total de voxels cresce em $O(2^{2n})$. Portanto, o número médio de semiespaços ativos em uma região do tamanho de um voxel aproxima-se assintoticamente de 1.

Se excluirmos sólidos CSG degenerados como, por exemplo, objetos com faces 'tangentes', podemos concluir que, para todos os fins práticos, a complexidade do algoritmo de visualização exposto é de $O(2^{2n})$, ou seja, é proporcional à soma das áreas dos objetos que compõem a cena.

2. Otimização

2.1. Melhorando a avaliação de faces

A análise que fizemos na seção anterior nos mostra que um aspecto crítico em termos de complexidade do algoritmo é o fato que as faces de cada objeto são avaliadas voxel a voxel. Tomemos por exemplo uma face F cuja projeção ocupe na imagem uma área de k pixels - o algoritmo precisará avaliar exatamente k voxels a fim de pintar F . Isto pode ser mais bem observado ao analisar a rotina *Visualiza*: de maneira geral, a menos que a árvore localizada seja nula ou que a subdivisão tenha chegado à região ocupada por um voxel, *Visualiza* é sempre chamada recursivamente.

Este procedimento se justificaria se estivéssemos tratando de um algoritmo de conversão de árvores CSG para bintrees. Entretanto, como estamos preocupados apenas em montar uma imagem do sólido, é razoável pensar em uma maneira pela qual se possa encurtar a profundidade da subdivisão em alguns casos particulares.

Claramente, é desejável encerrar o processo de subdivisão quando chegamos a uma região contendo apenas um semiespaço. Ora, para tal região, a pintura da imagem pode restringir-se à avaliação de uma bintree de dimensão 2 e não de dimensão 3 conforme é feito no algoritmo original.

Tomemos uma região delimitada por um paralelepípedo na qual apenas um semiespaço permanece ativo - nesta situação a imagem pode ser construída pintando a projeção do plano que define o semiespaço na área compreendida entre a projeção das duas retas de interseção desse plano com as faces do paralelepípedo perpendiculares ao eixo Z . A figura VI.4 (a) ilustra o problema.

Uma maneira de descrever a área a ser pintada é associar semiespaços bidimensionais às projeções das retas de interseção. Um semiespaço bidimensional corresponde ao conjunto de pontos $x \in E^2$ que satisfazem uma inequação da forma

$$a_1x_1 + a_2x_2 + a_3 \geq 0 \quad (\text{VI.12})$$

que pode ser reescrita, considerando os pontos x dados em coordenadas homogêneas, como

$$a \cdot x \geq 0 \quad (\text{VI.13})$$

onde

$$a = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \quad \text{e} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

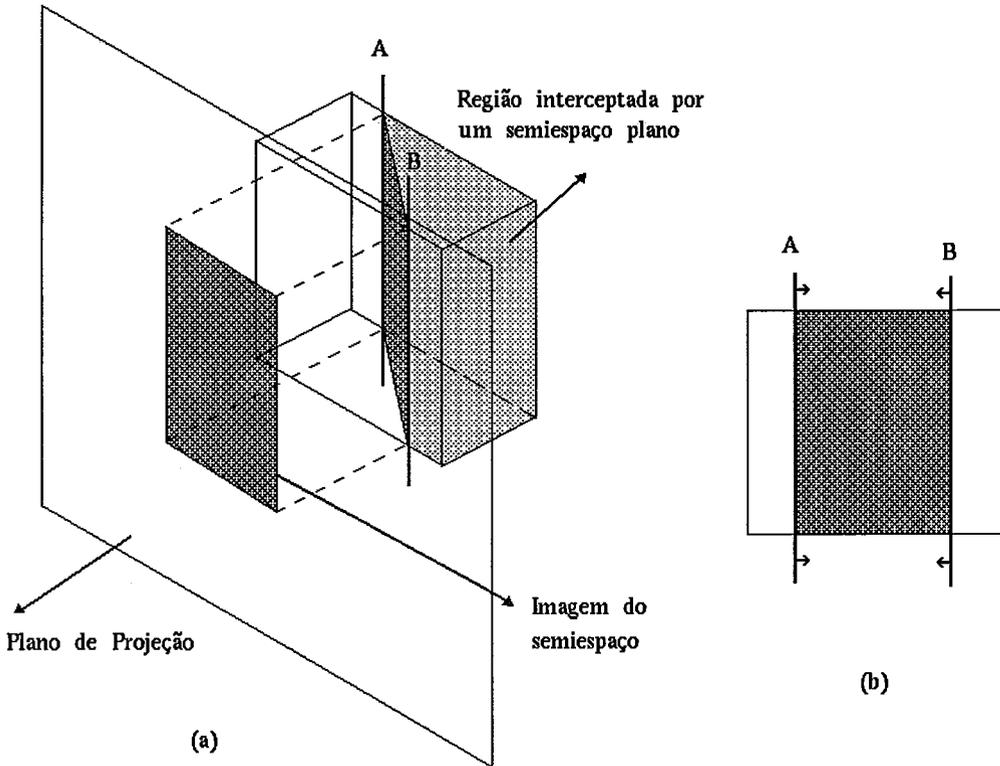


Figura VI.4

Pintura de uma região contendo um semiespaço ativo

Chamemos de A e B os semiespaços associados às interseções do plano com as faces $z = z_{max}$ e $z = z_{min}$ do paralelepípedo, então a área é dada por $A \cap^* B$ (veja na figura VI.4 (b)). A pintura dessa área pode então ser efetuada de maneira análoga à usada para converter uma árvore CSG para uma bintree tridimensional.

O cálculo dos coeficientes de A e B é feito substituindo-se $z=z_{max}$ e $z=z_{min}$ na equação do plano e escolhendo os sinais convenientemente. Seja A dado por (a_1, a_2, a_3) , B dado por (b_1, b_2, b_3) e o plano P dado por (p_1, p_2, p_3, p_4) , então,

$$a_1 = p_1, \quad a_2 = p_2, \quad a_3 = p_3 \cdot z_{max} + p_4$$

e

$$b_1 = -p_1, \quad b_2 = -p_2, \quad b_3 = -p_3 \cdot z_{min} - p_4$$

(VI.13)

Para construir um procedimento para pintar uma área dada pela interseção de semiespaços bidimensionais usa-se a mesma estratégia empregada para a avaliação de uma árvore CSG em três dimensões, a saber:

- Os semiespaços podem ser arranjados segundo uma lista - não é mais necessário empregar uma árvore, uma vez que o único operador aplicado sobre os semiespaços é o de interseção.
 - Os quatro vértices do retângulo correspondente à projeção do paralelepípedo são classificados com relação a cada semiespaço a fim de avaliar os valores v_{min} e v_{max} .
 - Para cada semiespaço é calculada uma tabela de incrementos δ_i correspondente à variação de v_{min} ou v_{max} após uma subdivisão.
 - O retângulo original é recursivamente dividido por retas paralelas ao eixo x e y em vezes alternadas. Após cada subdivisão a lista é simplificada segundo as regras características da operação de interseção.
 - Se o valor v_{min} de um determinado semiespaço é maior que zero, então ele é W -redundante, e pode portanto ser substituído por um nó especial representando o conjunto universo. Caso este seja o único semiespaço remanescente na lista, o retângulo correspondente à subdivisão pode ser pintado com a cor do plano.
- Se o valor v_{max} de um determinado semiespaço é menor que zero, então ele é \emptyset -redundante, e portanto o retângulo correspondente à subdivisão não faz parte da área a ser pintada.

A seguir descrevemos através de um pseudo-código as estruturas de dados a serem empregadas na rotina para pintura de áreas:

TYPE

```
Primitiva2D =
  RECORD
    SemiEspaco : ARRAY [1 .. 3] OF REAL;
    Incremento : ARRAY [IndiceSubdivisao] OF REAL;
  END;
Lista2D =
  POINTER TO No2D;
No2D =
  RECORD
    Prim : POINTER TO Primitiva2D;
    VMin, VMax : REAL;
    Prox : Lista2D;
  END;
VAR
  ListaNula, ListaUniverso : Lista2D;
```

As variáveis *ListaNula* e *ListaUniverso* são inicializadas no começo do programa e servem para indicar as ocasiões em que o processo de simplificação resulta em uma primitiva \emptyset - ou *W*-redundante.

O procedimento empregado para subdividir uma estrutura do tipo *Lista2D* segue em linhas gerais o seguinte pseudo-código:

```
PROCEDURE SimplificaLista2D (
  Nivel : IndiceSubdivisao;
  Lista : Lista2D;
  DirecaoEsquerda : BOOLEAN;
) : Lista2D;
BEGIN
  IF Lista^.Prox = NIL THEN
    RETURN AvaliaSemiespaco2D (Nivel, Lista, DirecaoEsquerda);
  ELSE
    Esq := AvaliaSemiespaco2D (Nivel, Lista, DirecaoEsquerda);
    Dir := SimplificaLista2D (Nivel, Lista^.Prox);
    IF (Esq = ListaNula) OR (Dir = ListaNula) THEN
      RETURN ListaNula
    ELSIF Esq = ListaUniverso THEN
```

```
    RETURN Dir
  ELSIF Dir = ListaUniverso THEN
    RETURN Esq
  ELSE
    Esq^.Prox := Dir;
    RETURN Esq;
  END;
END
END SimplificaLista2D;
```

Neste procedimento usamos a rotina auxiliar *AvaliaSemiespaco2D*, que é a contrapartida bidimensional da rotina *AvaliaSemiespaco* que descrevemos anteriormente, isto é, retorna uma *Lista2D* simples (com o campo *Prox* nulo) correspondente à localização de um semiespaço bidimensional após uma subdivisão.

2.2. Melhorando a avaliação de arestas

O mesmo raciocínio desenvolvido na seção anterior pode ser estendido para o caso de regiões que contenham dois semiespaços. Ou seja, desejamos encerrar o processo de subdivisão tridimensional quando a árvore localizada para uma dada região é composta da união ou interseção de dois semiespaços planos.

É razoável pensar que tal situação ocorra frequentemente na avaliação de regiões próximas a arestas do sólido, embora seja possível chegar-se, através do processo de subdivisão espacial, a regiões com dois semiespaços ativos sem que esses definam alguma aresta do sólido. A figura VI.5 mostra alguns exemplos de regiões contendo dois semiespaços planos.

Através desta modificação podemos esperar uma melhoria na complexidade de caso médio do algoritmo uma vez que:

- O processo de subdivisão tridimensional é abreviado também para regiões onde apenas dois semiespaços planos permanecem ativos, sendo substituído por um processo de subdivisão bidimensional.
- Em regiões onde os dois semiespaços formam uma aresta, esta pode ser avaliada diretamente calculando-se a reta de interseção dos dois planos, eliminando portanto a necessidade de disparo de raios nessas regiões.

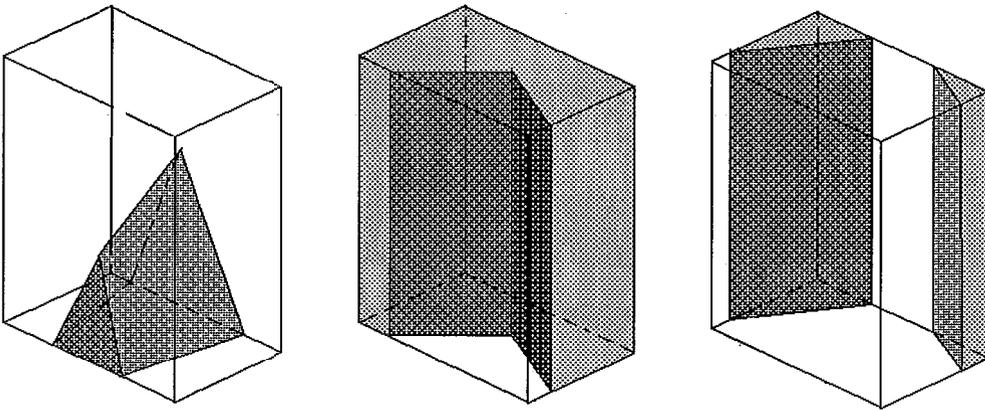


Figura VI.5

Regiões com dois semiespaços ativos

Podemos ver que, tipicamente, a projeção de regiões desse tipo não pode ser descrita simplesmente através da interseção de semiespaços bidimensionais. Isto pode ser entendido se observarmos que a interseção de n semiespaços bidimensionais se traduz em um polígono convexo, enquanto que a imagem decorrente da projeção de uma região contendo dois semiespaços planos pode conter diversos polígonos convexos.

Uma solução para esse problema é adotar uma representação baseada em uma árvore CSG bidimensional, isto é, os polígonos seriam representados por sub-árvores montadas a partir da interseção dos diversos semiespaços correspondentes a cada uma de suas arestas, e a imagem como um todo seria representada aplicando o operador "união" ligando essas sub-árvores. A figura VI.6 ilustra o processo.

O problema fundamental dessa abordagem é que arestas comuns a dois polígonos precisam ser representadas duas vezes na árvore. Para eliminar esse tipo de redundância foi elaborada uma representação alternativa, onde uma imagem composta pela justaposição de polígonos pode ser vista como uma sobreposição de polígonos mais simples. A figura VI.7 exemplifica a idéia.

A representação de uma sobreposição pode ser feita através de uma lista de polígonos arranjados segundo sua prioridade de visualização. Assim, o primeiro polígono da lista tem prioridade sobre o segundo, o segundo sobre o terceiro, e assim

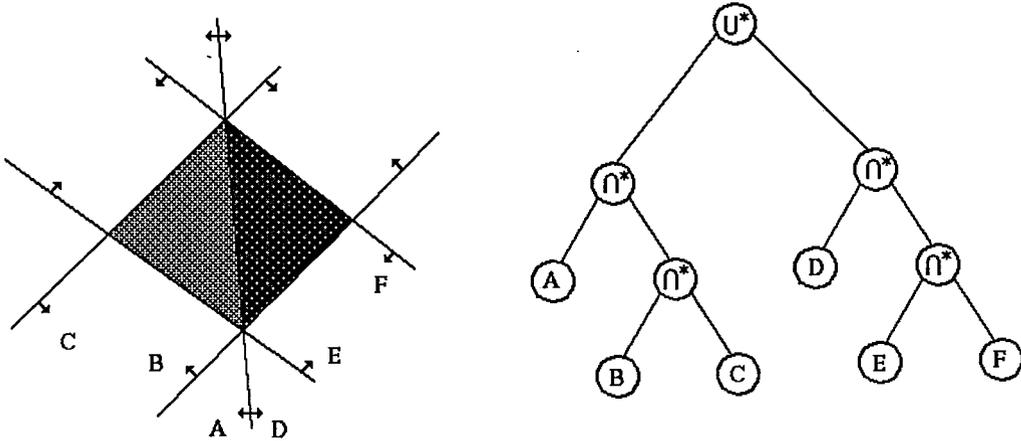


Figura VI.6

Representação de imagens por uma árvore CSG bidimensional

por diante. O conceito de prioridade se traduz no fato que, se detectamos uma área para a qual um determinado polígono de prioridade P é W -redundante (isto é, a área é totalmente coberta pelo polígono), então todos os polígonos de prioridade menor que P não contribuem para a imagem e podem ser descartados.

Podemos estender as estruturas de dados apresentadas na seção anterior para incluir arranjos de polígonos ordenados por prioridade de visualização. O pseudo-código abaixo capta a idéia:

TYPE

Arranjo2D =

 POINTER TO NoArranjo2D;

NoArranjo2D =

 RECORD

 Poligono : Lista2D;

 CorPoligono : INTEGER;

 Prox : Arranjo2D

 END;

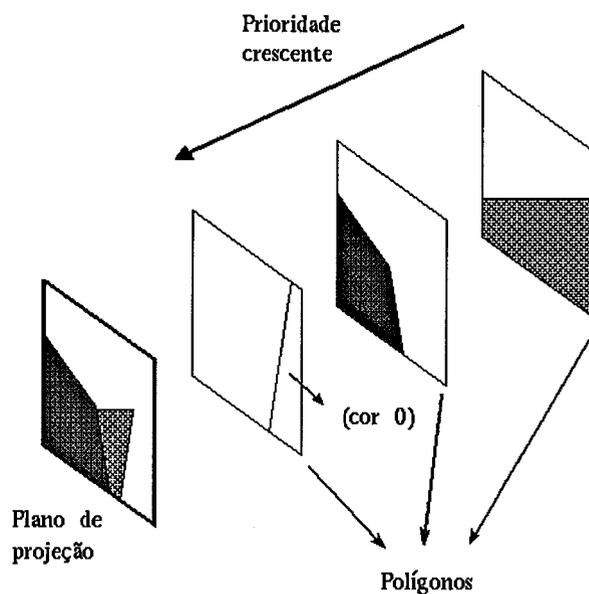


Figura VI.7

Representação de imagens por sobreposição de polígonos

O campo *CorPoligono* informa a cor a ser usada durante a pintura das áreas cobertas por *Poligono*.

Precisamos também montar um procedimento de divisão e simplificação de uma lista de polígonos. Este se resume em implementar a noção de prioridade de visualização dos polígonos, bem como em descartar polígonos que se tornem \emptyset -redundantes durante o processo de subdivisão:

```
PROCEDURE SimplificaArranjo2D (  
    Nivel : IndiceSubdivisao;  
    Arranjo : Arranjo2D;  
    DirecaoEsquerda : BOOLEAN  
    ) : Arranjo2D;  
  
VAR  
    Lista : Lista2D;  
    NovoArranjo : Arranjo2D;  
  
BEGIN  
    IF Arranjo = NIL THEN  
        RETURN NIL  
    END;  
    Lista := SimplificaLista2D (Nivel, Arranjo^.Poligono, DirecaoEsquerda);  
    IF Lista = ListaNula THEN  
        RETURN SimplificaArranjo (Nivel, Arranjo^.Prox, DirecaoEsquerda);  
    ELSIF Lista = ListaUniverso THEN  
        RETURN CriaArranjoSimples (Lista, Arranjo^.CorPoligono);  
    ELSE  
        NovoArranjo := CriaArranjoSimples (Lista, Arranjo^.CorPoligono);  
        NovoArranjo^.Prox :=  
            SimplificaArranjo (Nivel, Arranjo^.CorPoligono);  
        RETURN NovoArranjo  
    END  
END SimplificaArranjo2D;
```

A rotina *CriaArranjoSimples* retorna um *Arranjo2D* composto de um único polígono cuja cor e lista de semiespaços são passadas como parâmetros.

Definidas as estruturas de dados e de posse da rotina *SimplificaArranjo2D*, podemos montar um procedimento para fazer a pintura de uma imagem descrita por um arranjo de polígonos. A rotina *VisualizaArranjo2D* é análoga à rotina *Visualiza* apresentada anteriormente, com as seguintes adaptações:

- *VisualizaArranjo2D* atua sobre um arranjo de polígonos e não sobre uma árvore CSG.

- A visualização de um arranjo nulo não altera a imagem.
- Caso o arranjo passado como parâmetro corresponda a uma área retangular totalmente coberta por um polígono, esta pode ser pintada de uma só vez. Uma exceção é dada pelo caso em que o polígono tem cor 0 (convenciona-se que um polígono de cor 0 corresponde a uma área não coberta pelo arranjo), nessa situação o arranjo é tratado como se fosse nulo.
- Para o caso onde a subdivisão atingiu o nível máximo, correspondendo portanto à área de um pixel, é possível escolher uma cor dentre as cores dos polígonos que compõem o arranjo para pintar esse pixel. Essa escolha deverá ocasionar um pequeno erro que é perfeitamente tolerável, uma vez que este ocorrerá na vizinhança imediata de uma aresta.

Eis o pseudo-código da rotina *VisualizaArranjo2D*:

```
PROCEDURE VisualizaArranjo2D (  
    X, Y : INTEGER;  
    Nivel : IndiceSubdivisao;  
    Arranjo : Arranjo2D;  
    VAR Projecao : BinTree  
);  
  
VAR  
    Esq, Dir : Arranjo2D;  
  
BEGIN  
    IF (Projecao^.Tipo = Cheio) OR (Arranjo = NIL) OR  
       (Arranjo^.Poligono = ListaUniverso) AND (Arranjo^.CorPoligono = 0) THEN  
        RETURN;  
    ELSIF Nivel = MaxSubdivisao THEN  
        PintaPixel (X, Y, Arranjo^.CorPoligono);  
    ELSIF (Arranjo^.Poligono = ListaUniverso) AND (Projecao^.Tipo = Vazio) THEN  
        PintaRetangulo (X, Y, Nivel, Arranjo^.CorPoligono);  
        Projecao^.Tipo := Cheio  
    ELSE  
        IF Projecao^.Tipo = Vazio THEN  
            Divide (Projecao);  
        END;  
        Esq := SimplificaArranjo2D (Nivel, Arranjo, TRUE);
```

```
Dir := SimplificaArranjo2D (Nivel, Arranjo, FALSE);
VisualizaArranjo2D (X, Y, Nivel+1, Esq, Projecao^.Esquerda);
VisualizaArranjo2D (X + IncX [Nivel], Y + IncY [Nivel],
    Nivel+1, Dir, Projecao^.Direita);
IF (Projecao^.Esquerda^.Tipo = Cheio) AND
    (Projecao^.Direita^.Tipo = Cheio) THEN
    Junta (Projecao);
END
END
END VisualizaArranjo2D;
```

A rotina *PintaRetangulo* recebe como parâmetros as coordenadas do vértice inferior esquerdo de um retângulo, um inteiro correspondente ao nível da subdivisão e um índice de cor. As dimensões do retângulo podem ser inferidas pelo valor do nível da subdivisão em questão. A função dessa rotina é simplesmente pintar um retângulo com uma cor.

2.3. Montando arranjos bidimensionais

Vimos nas seções anteriores que é possível montar uma estrutura, à qual chamamos de *arranjo*, para representar a imagem da projeção de uma região correspondente a um paralelepípedo onde a localização da árvore CSG do sólido resume-se a: (1) um único semiespaço plano, (2) uma união de dois semiespaços e (3) uma interseção de dois semiespaços.

Existem muitas maneiras pelas quais podemos montar esses arranjos, porém é importante fazê-lo o mais economicamente possível, isto é, utilizando um número mínimo de semiespaços bidimensionais.

Os semiespaços bidimensionais são montados a partir da equação de uma reta e uma especificação de sinal. A reta é dada pela interseção de dois planos em E^3 , enquanto que a especificação de sinal distingue um dos dois semiespaços bidimensionais formados ao partirmos E^2 pela reta.

Na discussão que se segue usaremos as seguintes definições:

- Sejam A e B planos em E^3 , definidos respectivamente pelos coeficientes (a_1, a_2, a_3, a_4) e (b_1, b_2, b_3, b_4) , sendo que A e B não são paralelos e A é não

perpendicular ao eixo Z . Então A e B se interceptam segundo uma reta R , a qual divide A em dois semiplanos que podem ser projetados no plano $z = 0$ resultando em dois semiespaços bidimensionais em E^2 .

- Chamaremos de *ProjecaoPositiva*(A,B) ao semiespaco bidimensional com coeficientes (p_1, p_2, p_3) dados por:

$$\left[p_1 \ p_2 \ p_3 \right] = \left[a_1 - \frac{a_3 b_1}{b_3} \quad a_2 - \frac{a_3 b_2}{b_3} \quad a_4 - \frac{a_3 b_4}{b_3} \right] \quad (\text{VI.13})$$

- Analogamente, chamaremos de *ProjecaoNegativa*(A,B) ao semiespaco bidimensional com coeficientes (p_1, p_2, p_3) dados por:

$$\left[p_1 \ p_2 \ p_3 \right] = \left[\frac{a_3 b_1}{b_3} - a_1 \quad \frac{a_3 b_2}{b_3} - a_2 \quad \frac{a_3 b_4}{b_3} - a_4 \right] \quad (\text{VI.14})$$

- $P_z(\bar{z})$ é definido como o plano $z = \bar{z}$.
- $P_z(zmin)$ e $P_z(zmax)$ são os planos perpendiculares ao eixo Z que delimitam o paralelepípedo da região.
- $Cor(s)$ é a cor associada à projeção do semiespaço s .
- *Poligono*($c, s_1 \cdots s_n$) é o polígono formado pela interseção dos semiespaços bidimensionais $s_1 \cdots s_n$ e que será pintado com a cor c .
- *Arranjo*($p_1 \cdots p_m$) é o arranjo formado pela sobreposição dos polígonos $p_1 \cdots p_m$, sendo que p_1 tem a maior, e p_m a menor prioridade.

Dadas as definições acima, podemos construir três rotinas que têm a função de montar arranjos bidimensionais para os três tipos de regiões enumeradas no início desta seção.

Uma observação importante é que em muitos casos não é necessário incluir em um arranjo semiespaços bidimensionais dados pela interseção de $P_z(zmin)$ com o plano correspondente a um semiespaço tridimensional. Isto acontece quando se pode garantir que o parâmetro *Projecao* da rotina *VisualizaArranjo2D* corresponde a uma bintree que possui nós cheios cobrindo a mesma área que seria descrita por um semiespaço bidimensional.

Por exemplo, a rotina *ArranjoFace*, que trata uma região contendo um único semiespaço tridimensional S , pode ser traduzida no seguinte pseudo-código:

PROCEDURE ArranjoFace (S : Primitiva) : Arranjo2D;

```
BEGIN
  IF S.SemiEspaco [3] = 0.0 THEN
    (* S é paralelo ao eixo Z *)
    RETURN NIL
  ELSE
    RETURN
      Arranjo (
        Poligono (
          Cor (S),
          ProjecaoPositiva (S, Pz(zmax))
        )
      )
  END
END ArranjoFace;
```

Alguns detalhes podem ser observados na rotina acima:

- Não é necessário usar o semiespaço bidimensional correspondente à interseção de S com $P_z(z_{min})$.
- Quando S é perpendicular ao eixo Z , o arranjo é nulo, uma vez que S não contribui para a imagem.
- Regiões do tipo face que contenham semiespaços tridimensionais voltados na direção oposta à do observador (com coeficiente $a_3 < 0$) nunca são visitadas pelo processo de subdivisão recursiva do espaço controlado pela projeção.

A rotina *ArranjoUniao*, cujo pseudo-código apresentamos a seguir, monta um arranjo bidimensional equivalente à projeção de uma região interceptada por semiespaços ligados pelo operador união.

PROCEDURE ArranjoUniao (S1, S2 : Primitiva) : Arranjo2D;

```
BEGIN
  IF S1.SemiEspaco [3] <= 0.0 THEN
    RETURN ArranjoFace (S2)
  ELSIF S2.SemiEspaco [3] <= 0.0 THEN
    RETURN ArranjoFace (S1)
  ELSE
    RETURN
      Arranjo (
        Poligono (
          Cor (S2),
          ProjecaoPositiva (S2, Pz(zmax)),
          ProjecaoPositiva (S2, S1))
        ),
        Poligono (
          Cor (S1),
          ProjecaoPositiva (S1, Pz(zmax)))
      )
  )
END
END ArranjoUniao;
```

As seguintes observações são relevantes para a análise da rotina *ArranjoUniao*:

- Não é necessário usar os semiespaços bidimensionais correspondentes à interseção de $S1$ ou $S2$ com $P_z(z_{min})$.
- $S1$ ou $S2$ podem estar voltados na direção oposta à do observador (mas não ambos ao mesmo tempo).
- Se $S1$ tem seu coeficiente $a_3 \leq 0$, $S1$ não contribui para a imagem da região, e portanto o arranjo corresponde a uma região do tipo face formada apenas por $S2$. Raciocínio análogo pode ser aplicado a $S2$.
- Não é necessário efetuar qualquer tipo de comparação geométrica entre as retas de interseção que irão formar os polígonos do arranjo; o algoritmo trata implicitamente qualquer caso. A figura VI.8 ilustra a montagem de arranjos do tipo união para dois casos típicos.

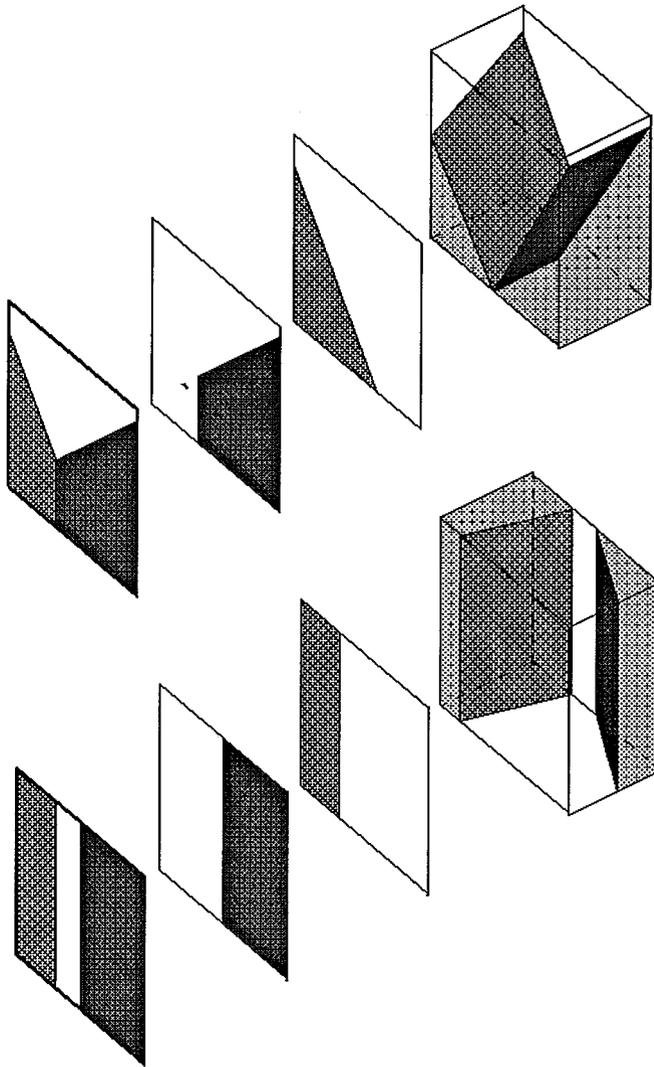


Figura VI.8

Montagem de arranjos bidimensionais do tipo união

Apresentamos a seguir a terceira rotina - *ArranjoIntersecao* - que trata regiões contendo uma interseção de dois semiespaços:

```
PROCEDURE ArranjoIntersecao (S1, S2 : Primitiva) : Arranjo2D;
```

```
BEGIN
```

```
IF (S1.SemiEspaco [3] <= 0.0) AND
  (S2.SemiEspaco [3] <= 0.0) THEN
  RETURN NIL
ELSIF S2.SemiEspaco [3] <= 0.0 THEN
  RETURN ArranjoIntersecao (S2, S1)
ELSIF S1.SemiEspaco [3] <= 0.0 THEN
  RETURN
  Arranjo (
    Poligono (0,ProjecaoPositiva (S2, Pz(zmin))),
    Poligono (0,ProjecaoNegativa (S2, Pz(zmax))),
    Poligono (
      Cor (S2),
      ProjecaoPositiva (S2, S1)
    )
  )
ELSE
  RETURN
  Arranjo (
    Poligono (0,ProjecaoNegativa (S2, Pz(zmax))),
    Poligono (
      Cor (S2),
      ProjecaoPositiva (S2, S1)
    ),
    Poligono (
      Cor (S1),
      ProjecaoPositiva (S1, Pz(zmax))
    )
  )
END
END ArranjoIntersecao;
```

Pode-se notar na rotina *ArranjoIntersecao* que:

- Se nenhum dos semiespaços $S1$ e $S2$ está voltado para o observador, não há contribuição para a imagem.
- No caso de $S1$ ou $S2$ estarem voltados para a direção oposta à do observador, é necessário retirar da imagem o semiespaço bidimensional dado por *ProjecaoPositiva (S1 ou S2, P_z(zmin))*.

Duas situações ilustrando a montagem de um arranjo do tipo interseção são mostradas na figura VI.9.

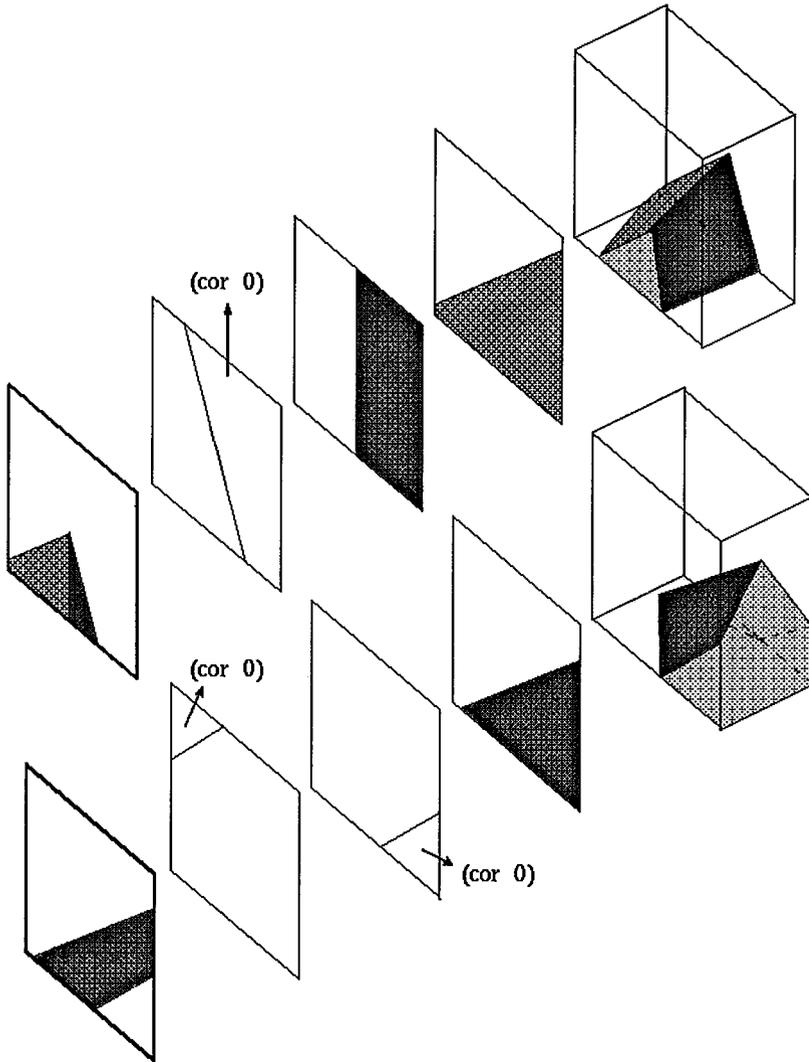


Figura VI.9

Montagem de arranjos bidimensionais do tipo interseção

2.4. Algoritmo de visualização modificado

Dispomos agora de todos os elementos para montar um algoritmo de visualização baseado naquele apresentado por TAMMINEN et alii, diferenciando-se desse pelo fato do processo de subdivisão tridimensional ser abreviado quando as regiões visitadas são interceptadas por apenas um ou dois semiespaços. Seu pseudo-código é apresentado abaixo.

```
PROCEDURE Visualiza (  
    X, Y : INTEGER;  
    Nivel : IndiceSubdivisao;  
    Arvore : ArvoreCSG;  
    VAR Projecao : BinTree;  
);  
  
VAR  
    PrimitivaInterceptada : POINTER TO Primitiva;  
    Esq, Dir : ArvoreCSG;  
    Arranjo : Arranjo2D;  
  
BEGIN  
    IF (Projecao^.Tipo = Cheio) OR (Arvore = ArvoreNula) THEN  
        RETURN;  
    ELSIF Nivel = MaxSubdivisao THEN  
        IF Arvore^.Tipo = Folha THEN  
            PintaPixel (X, Y, Cor (Arvore^.Prim));  
            Projecao^.Tipo := Cheio;  
        ELSE  
            PrimitivaInterceptada := DisparaRaio (X, Y, Arvore);  
            IF PrimitivaInterceptada <> NIL THEN  
                PintaPixel (X, Y, Cor (PrimitivaInterceptada));  
                Projecao^.Tipo := Cheio;  
            END  
        END  
    END  
    ELSIF Arvore^.Tipo = Folha THEN  
        VisualizaArranjo2D (  
            X, Y, Nivel,
```

```
        ArranjoFace (Arvore^.Primitiva),
        Projecao);
ELSIF (Arvore^.Esquerda^.Tipo = Folha) AND
      (Arvore^.Direita^.Tipo = Folha) THEN
  IF (Arvore^.Operador = Uniao) THEN
    Arranjo :=
      ArranjoUniao (
        Arvore^.Esquerda^.Primitiva,
        Arvore^.Direita^.Primitiva))
  ELSE
    Arranjo :=
      ArranjoIntersecao (
        Arvore^.Esquerda^.Primitiva,
        Arvore^.Direita^.Primitiva))
  END;
  VisualizaArranjo2D (X, Y, Nivel, Arranjo, Projecao)
ELSE
  Esq := SimplificaArvore (Nivel, Arvore, TRUE);
  Dir := SimplificaArvore (Nivel, Arvore, FALSE);
  IF Nivel MOD 3 = 0 THEN (* Subdivisao em Z *)
    Visualiza (X, Y, Nivel+1, Esq, Projecao);
    Visualiza (X, Y, Nivel+1, Dir, Projecao);
  ELSE (* Subdivisao em X ou Y *)
    IF Projecao^.Tipo = Vazio THEN Divide (Projecao) END;
    Visualiza (X, Y, Nivel, Nivel+1, Esq, Projecao^.Esquerda);
    Visualiza (X+IncX[Nivel], Y+IncY[Nivel],
      Nivel+1, Dir, Projecao^.Direita);
    IF (Projecao^.Esquerda^.Tipo = Cheio) AND
      (Projecao^.Direita^.Tipo = Cheio) THEN
      Junta (Projecao);
    END;
  END
END
END Visualiza;
```

3. Resultados e conclusões

3.1. Resultados obtidos

Os algoritmos descritos neste capítulo foram implementados de forma a serem aplicáveis a objetos descritos em LDS. Os programas foram construídos de forma a substituir as primitivas delimitadas por superfícies quádricas por poliedros que as aproximam - esferas são substituídas por icosaedros e cilindros por prismas de seção octogonal.

Foram utilizados basicamente os mesmos objetos empregados no capítulo V. O objeto *Teste5* não foi usado nos testes, uma vez que este pode ser considerado como uma variante do sólido *Teste4*. Em contrapartida, achamos interessante utilizar um sólido de teste (*Teste6*) semelhante àquele empregado por KOISTINEN et alii [30]. As imagens geradas desses sólidos são ilustradas na figura VI.10.

Conduzimos uma primeira bateria de testes aplicando o algoritmo original e o algoritmo modificado para gerar imagens dos sólidos *Teste1* a *Teste4*. As imagens foram geradas numa estação de trabalho *Sun-3/60* com 128×128 pixels, e os resultados são apresentados na tabela VI.1 (tempos são dados em segundos).

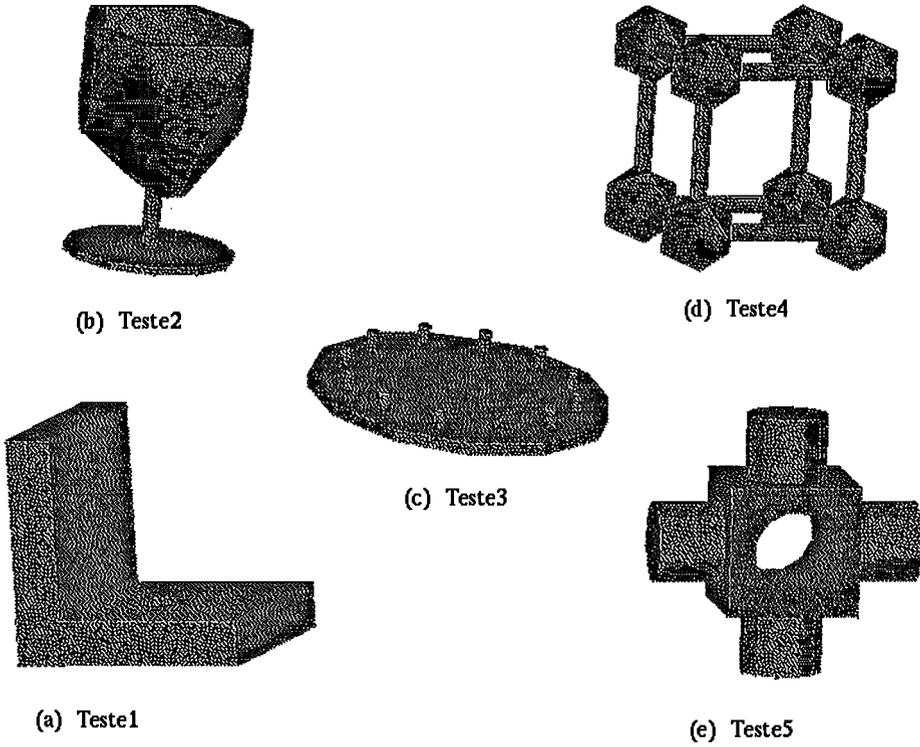


Figura VI.10
Sólidos de teste

Sólido CSG	No de semiespaços	Algoritmo Original	Algoritmo Modificado	Razão (Original = 1)
Teste1	22	24.9	13.4	0.54
Teste2	66	39.0	32.1	0.82
Teste3	130	39.0	35.0	0.89
Teste4	280	92.4	93.5	1.01

Tabela VI.1

Analisando os resultados obtidos, podemos observar um ganho em termos de tempo de processamento em quase todos os casos, à exceção do sólido *Teste4*. Repetimos estes mesmos testes, agora na geração de imagens de 256 × 256 pixels, obtendo os seguintes resultados:

Sólido CSG	No de semiespaços	Algoritmo Original	Algoritmo Modificado	Razão (Original = 1)
Teste1	22	87.7	22.6	0.25
Teste2	66	97.7	51.6	0.52
Teste3	130	85.9	60.3	0.70
Teste4	280	184.9	176.3	0.95

Tabela VI.2

Comparando os resultados mostrados nas tabelas VI.1 e VI.2, observamos um aumento do ganho do algoritmo modificado sobre o algoritmo original, em função do aumento na resolução.

A melhoria obtida é menos perceptível para o caso dos sólidos *Teste3* e *Teste4*. Com efeito, o ganho que se espera para o algoritmo modificado é devido à melhoria na avaliação de faces e arestas "isoladas". Nesses dois sólidos essas características não são muito notadas, isto é, a subdivisão tridimensional tende a se propagar até regiões muito pequenas.

Para podermos observar melhor o comportamento de ambos os algoritmos, foram efetuados testes adicionais contabilizando uma série de fatores importantes na análise de complexidade. Usou-se um único sólido (*Teste6*) para gerar imagens sucessivamente maiores, obtendo-se os seguintes resultados:

	Resolução da imagem		
	128	256	512
Tempo pré-processamento	3.6	3.6	3.6
ALG. ORIGINAL			
Raios	2189	4629	10662
Subdivisões	22717	84814	332290
Aval. de semiespaços	76441	216646	727855
Tempo total	32.1	81.6	257.3
ALG. MODIFICADO			
Raios	269	289	266
Subdivisões 3D	5246	6470	7674
Subdivisões 2D	3200	9665	22954
Aval. de semiespaços 3D	36762	43527	50136
Aval. de semiespaços 2D	8264	23852	54791
Regiões do tipo face	526	737	949
Regiões do tipo união	256	349	440
Regiões do tipo interseção	1171	1557	1948
Tempo total	25.0	38.4	62.7

Tabela VI.3

As estatísticas mostradas na tabela VI.3 sugerem os seguintes fatos:

- O número total de subdivisões (3D+2D) exigidas pelo algoritmo modificado varia de forma aproximadamente linear com a resolução da imagem, enquanto que no algoritmo original, esta variação é quadrática.
- O número de raios disparados permanece aproximadamente constante no algoritmo modificado. Isto pode ser atribuído à necessidade de disparar raios apenas em regiões próximas a vértices. No algoritmo original, raios são disparados em regiões próximas a vértices e arestas, explicando-se portanto a variação linear do número de raios em função da resolução da imagem.
- O número de regiões tratadas através da visualização de arranjos (regiões do tipo face/união/interseção) varia muito pouco ao se aumentar a resolução.

3.2. Conclusões

No desenvolvimento deste trabalho, tivemos oportunidade de tomar contato com muitas das questões que regem a utilização de técnicas de subdivisão espacial. Muito particularmente, pudemos comprovar que um fator preponderante na aplicação dessas técnicas é avaliar precisamente até que ponto sua utilização pura e simples pode nos conduzir a um algoritmo ótimo.

As modificações que propusemos ao algoritmo de KOISTINEN et alii estão longe de serem suficientes para a montagem de um algoritmo realmente eficiente para visualização direta de modelos CSG, mas de toda forma, provêm uma abordagem encorajadora.

Uma série de outras idéias podem ainda ser incorporadas a esse algoritmo, dentre as quais destacamos:

- A utilização de caixas envolventes pode se mostrar vantajosa quando os limites dos poliedros que formam as primitivas são conhecidas.
- Uma modelagem através de arranjos bidimensionais pode ser feita para regiões próximas de vértices de primitivas.
- No processo de classificação de semiespaços tridimensionais assume-se que os planos que os delimitam se prolongam indefinidamente. Quando os sólidos são montados através de primitivas poliedrais, a extensão desses planos é conhecida - isto pode ser usado para acelerar o processo de classificação dos semiespaços.
- Uma estrutura híbrida composta de uma bintree bidimensional tendo arranjos como folhas pode ser usada no controle da projeção. Neste caso, é preciso desenvolver uma técnica através da qual dois arranjos correspondentes à mesma área da imagem possam ser combinados e simplificados. Isto pode eliminar uma boa parte do trabalho efetuado na avaliação de pseudo-arestas, isto é, arestas formadas ao se interceptar semiespaços com os planos perpendiculares ao eixo Z.

Capítulo VII

Considerações finais

1. Avaliação do uso de subdivisão espacial

Neste ponto podemos fazer um balanço do que foi obtido com a introdução das diversas técnicas de otimização apresentadas.

Nos capítulos V e VI lidamos com subdivisão espacial segundo diferentes ângulos. No algoritmo de visualização baseado em disparo de raios, utilizamos subdivisão espacial com vistas a obter ganhos em termos de eficiência, pela redução da complexidade em cada disparo. Já no caso do algoritmo de visualização por conversão para bintrees, esses ganhos foram obtidos *reduzindo-se* a frequência com que a subdivisão espacial é aplicada.

É bastante evidente que as idéias relacionadas com subdivisão espacial, isto é, os conceitos de redundância de primitivas, localização e simplificação de árvores CSG, constituem elementos valiosos quando se trata de reduzir a complexidade de diversos problemas relacionados com modelos CSG. São conceitos elegantes, teoricamente bem fundamentados e de fácil aplicabilidade. Por isso mesmo, é tentador procurar utilizá-los de maneira indiscriminada, e devemos confessar que incorremos neste erro frequentemente. Aprendemos por experiência que existe uma justa medida além da qual o uso de subdivisão se mostra contraproducente - a questão resume-se então em identificar esses limites. Apresento aqui algumas considerações sobre este problema:

- Subdividir o espaço e obter localizações de árvores são operações que tendem a ser repetidas um grande número de vezes. Tais operações devem merecer particular cuidado durante a implementação. O algoritmo de KOISTINEN, TAMMINEN e SAMET [30] nos mostra um excelente exemplo de como isso pode ser feito.
- A identificação de situações onde a complexidade do problema foi reduzida a ponto de poder ser tratada por outros meios é um fator crítico. Essa identificação deve ser feita economicamente e, por outro lado, as situações devem ser tão genéricas quanto possível. O primeiro requisito costuma se contrapor ao segundo.

- Testes de redundância de primitivas constituem outro ponto-chave. Por um lado, é importante que esses testes sejam exatos, por outro, soluções aproximadas costumam ser mais simples. Nesse sentido, o modelo de caixas envolventes pode ser considerada uma boa solução de compromisso em muitos casos.
- Se o modelo do sólido nos fornece informações geométricas de algum tipo, essas devem ser usadas. No algoritmo baseado em conversão para bintrees por exemplo, apesar de serem conhecidos limites geométricos mais precisos sobre os semiespaços, estes são ignorados.
- Deve-se procurar evitar a repetição de cálculos geométricos (interseções de planos, testes de raios contra primitivas etc). Já a complexidade desses cálculos deve representar um fator de menor importância, uma vez que dispositivos dedicados a esse fim se tornam cada vez mais comuns.

2. Tópicos para pesquisa futura

Atualmente, métodos de visualização direta são empregados primordialmente durante a fase de modelagem em CSG, uma vez que oferecem um desempenho superior a avaliação de fronteiras seguida de visualização *B-Rep*. Apesar disso, os algoritmos de visualização para modelos de fronteiras são pelo menos uma ordem de magnitude mais eficientes, o que torna *B-Rep* a representação de escolha em aplicações onde rapidez de visualização é um ponto fundamental.

Tal situação deve persistir, a menos que surja uma abordagem que permita a obtenção mais rápida de imagens de modelos CSG do que os métodos conhecidos até o momento. Muitos grupos de pesquisa têm se dedicado a essa tarefa, sendo que os caminhos adotados podem ser divididos em duas categorias: (1) a pesquisa no sentido de desenvolver arquiteturas dedicadas e (2) adaptação e refinamento de técnicas já conhecidas. Esses caminhos não são mutuamente exclusivos, pelo contrário, nos parece que uma solução adequada do problema será obtida conjugando-se essas duas abordagens.

De qualquer forma, é muito provável que o uso de subdivisão espacial deverá representar um papel importante nesses esforços. Uma sugestão que nos permitimos oferecer é estudar métodos de visualização onde subdivisão espacial não foi ainda aplicada com sucesso palpável. Dentre esses, o algoritmo baseado em linhas de rastreio proposto por ATHERTON [25] constitui um bom candidato.

Uma das razões pelas quais o algoritmo de Atherton é tão eficiente é o fato que ele parte de primitivas cuja representação por fronteiras é conhecida. No curso de alguns experimentos que fizemos com esse algoritmo, precisamente por não tirarmos partido dessa particularidade, não foi possível obter resultados significativos.

Um outro tópico que vem merecendo muita atenção é o que diz respeito ao aumento do poder de expressão dos modelos CSG. Há muitos ângulos pelos quais o problema pode ser encarado. Uma primeira idéia é aumentar a família de sólidos primitivos para incluir, por exemplo, sólidos delimitados por superfícies paramétricas (polinomiais e racionais). Outra abordagem é utilizar novos operadores de instanciação - sob este aspecto, transformações projetivas constituem uma idéia de aplicação imediata.

Nas conclusões dos capítulos III, V e VI, apresentamos também algumas sugestões que podem orientar pesquisadores que desejem dar continuidade a este trabalho.

Referências Bibliográficas

- [1] JANICH, K., "Topology", *Springer-Verlag*, 1984.
- [2] ARMSTRONG, M. A., "Basic Topology", *Springer-Verlag* 1983.
- [3] REQUICHA, A. A. G., "Representations for rigid solids: theory, methods and systems", *Computing Surveys*, vol 12, no 4, dezembro de 1980.
- [4] SAMET, H., "The Design and Analysis of Spatial Data Structures" *Addison Wesley*, 1989.
- [5] MANTYLA, M., "An Introduction do Solid Modeling" *Computer Science Press*. 1988.
- [6] REQUICHA, A.A.G. e VOELKER, H.B., "Solid Modeling: Current Status and Research Directions" *IEEE Computer Graphics & Applications*, October 83, pp. 25-37, 1983.
- [7] REQUICHA, A. A. G., "Mathematical models of rigid solids" *Technical Memorandum No. 28, Production Automation Project, University of Rochester*, dezembro de 1980.
- [8] REQUICHA, A. A. G. e VOELCKER, H. B., "Constructive Solid Geometry" *Technical Memorandum No. 25, Production Automation Project, University of Rochester*, 1977.
- [9] REQUICHA, A. A. G. e VOELCKER, H. B., "Geometric modeling of physical parts and processes" *IEEE Computer*, Vol 10, no 2, pp. 48-57, 1977.
- [10] BRONSVOORT, W. F. "Boundary evaluation and direct display of CSG models" *Computer-Aided Design*, vol 20, no 7, pp. 416-419, 1988.

- [11] REQUICHA, A. A. G. e VOELCKER, H. B., "Boolean operations in solid modeling: boundary evaluation and merging algorithms", *Proceedings of the IEEE*, Vol 73, no 1, pp. 30-44, janeiro de 1985.
- [12] LEVIN, J. L., "Mathematical models for determining the intersections of quadric surfaces", *Computer Graphics and Image Processing*, no 11, 1979.
- [13] LEVIN, J. L., "Parametric algorithm for drawing pictures of solid objects composed of quadric surfaces", *Communications of the ACM*, outubro de 1976.
- [14] MILLER, J. R., "Analysis of quadric surface based solid models", *IEEE Computer Graphics & Applications*, janeiro de 1988.
- [15] SARRAGA, R. F. "Algebraic models for intersections of quadric surfaces in GMSOLID", *Computer Vision, Graphics and Image Processing*, no 22, 1983.
- [16] LAIDLAW, D. H., TRUMBORE, W. B., e HUGHES, J. F., "Constructive Solid Geometry for polyhedral objects", *Computer Graphics* vol 20, no 4, pp. 161-170, agosto de 1986.
- [17] MANTYLA, M., "Boolean operations of 2-manifolds through vertex neighborhood classification", *ACM Transactions on Graphics*, vol 5, no 1, pp. 1-29, janeiro de 1986.
- [18] SEGAL, M., e SEQUIN, C. H., "Partitioning polyhedral objects into non-intersecting parts", *IEEE Computer Graphics & Applications* vol 8, no 1, pp. 53-67, janeiro de 1988.
- [19] NAVAZO, I., FONTDECABA, J., e BRUNET, P., "Extended octrees, between CSG trees and boundary representations", *North-Holland em "Proceedings of Eurographics '87"*, pp. 239-247, 1987.
- [20] MILLER, J. R., "Geometric approaches to nonplanar quadric surface intersection curves" *ACM Transactions on Graphics* vol 6, no 4, pp. 274-307, outubro de 1987.

- [21] TILOVE, R. B. , REQUICHA, A. A. G. e HOPKINS, M. R., "Efficient editing of solid models by exploiting structural and spatial locality" *Computer Aided Geometric Design* no 1, pp. 227-239, 1984.
- [22] WOODWARK, J. R., "Generating wireframes from set-theoretic solid models by spatial subdivision" *Computer Aided Design*, vol 18, no 6, pp. 307-315, 1986.
- [23] WOODWARK, J. R. e QUINLAN K. M., "The derivation of graphics from volume models by recursive subdivision of the object space" *Proceedings Computer Graphics Conference 80* Online Publishers, London, pp 335-343, 1980.
- [24] WOODWARK, J. R. e QUINLAN K. M., "Reducing the effect of complexity on volume model evaluation" *Computer-Aided Design* vol 14, no 2, pp. 89-95, 1982.
- [25] ATHERTON, P. R., "A scan-line hidden surface removal procedure for constructive solid geometry" *ACM Computer Graphics* vol 17, no 3, pp. 73-82, 1983.
- [26] BRONSVOORT, W. F., "Techniques for reducing boolean evaluation time in CSG scan-line algorithms", *Computer-Aided Design*, vol 18, no 10, pp. 533-538, dezembro de 1986.
- [27] JANSEN, F. W., "A CSG list priority hidden surface algorithm", *North-Holland*, em "Proceedings of Eurographics '85", pp. 51-62, 1985.
- [28] BRONSVOORT, W. F., "An algorithm for visible-line and visible-surface display of CSG models" *Visual Computing*, vol 3, no 4, pp. 176-185, dezembro de 1987.
- [29] VERROUST, A., "Visualization algorithm for CSG polyhedral solids", *Computer-Aided Design*, vol 19, no 10, pp. 527-533, dezembro de 1987.
- [30] KOISTINEN, P., TAMMINEN, M. e SAMET, H., "Viewing solid models by bin-tree conversion" *North-Holland* em "Proceedings of the EUROGRAPHICS'86", pp. 147-157, 1986.

- [31] GOLDSTEIN, R. A., e NAGEL, R., "3-D visual simulation", *Simulation*, vol 16, no 1, pp. 25-31, janeiro de 1971.
- [32] ROTH, S. D, "Ray casting for modeling solids" *Computer Graphics and Image Processing* vol 18, no 9, pp. 109-144, 1982.
- [33] PUEYO, X., e MENDOZA, J. C., "A new scan line algorithm for rendering of CSG trees" *North-Holland* em "Proceedings of Eurographics '87", pp. 347-361, 1987.
- [34] ROSSIGNAC, J. R. e REQUICHA, A. A. G., "Depth-buffering display techniques for Constructive Solid Geometry" *IEEE Computer Graphics & Applications* September 1986, pp. 29-39, 1986.
- [35] GOLDFEATHER, J., MOLNAR, S., TURK G. e FUCHS H., "Near real-time CSG rendering using tree normalization and geometric pruning" *IEEE Computer Graphics & Applications* May 1989, pp. 20-28, 1989.
- [36] TILOVE, R. B., "A null-object detection algorithm for constructive solid geometry" *Communications of the ACM* July 1984, vol 27, no 7, pp. 684-694, 1984.
- [37] JACKINS, C. L. e TANIMOTO S. L., "Octrees and their use in representating three-dimensional objects", *Computer Graphics and Image Processing*, vol 14, pp 249-270, 1980.
- [38] MEAGHER, D, "Geometric modeling using octree encoding", *Computer Graphics and Image Processing*, vol 19, pp 129-147, 1982.
- [39] SAMET, H e TAMMINEN, M., "Bintrees, CSG trees and time", *Computer Graphics*, (SIGGRAPH'85), vol 19, no 3, pp. 121-130, 1985.
- [40] MORRIS, D. T. e QUARENDON, P., "An algorithm for direct display of CSG objects by spatial subdivision", *Springer-Verlag* em "Fundamental Algorithms for Computer Graphics", NATO ASI-Series, vol F17, pp. 725-736, 1985.

- [41] APPEL, A., "Some techniques for shading machine renderings of solids", *AFIPS 1968 Spring Joint Computing Conference*, pp. 37-45, 1968.
- [42] GOLDSTEIN, R. A. e MALIN, L., "3D Modeling with the Syntha Vision System", *First National Conference on Computer Graphics in CAD/CAM Systems, MIT*, pp. 244-247, abril de 1979.
- [43] KAY, D. S., "Transparency, refraction and ray tracing for computer synthesized images", *Master Thesis, Program of Computer Graphics, Cornell University*, janeiro de 1979.
- [44] KAY, D. S. e GREENBERG, D., "Transparency for computer synthesized images", *Computer Graphics*, em "Proceedings of SIGGRAPH '79", vol 13, pp. 158-164, 1979.
- [45] WHITTED, T., "An improved illumination model for Shaded Display" *Communications of the ACM* Vol 23, pp. 343-349, 1980
- [46] ROGERS, D. F., "Procedural elements for computer graphics" *McGraw-Hill Book Company* 3a edição, 1988.
- [47] MEAGHER, D., "Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3-d objects by computer", *Electrical and Systems Engineering Technical Report IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY, outubro de 1980*.
- [48] HUNTER, G. M., "Efficient computation and data structures for graphics", *Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ*, 1978.
- [49] HUNTER, G. M., STEIGLITZ, K., "Operations on images using quad trees", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol 1, no 2, pp. 145-153, abril de 1979.

- [50] COMBA, J. L., ESPERANÇA, C., PERSIANO, R. C. M., "Algoritmos para exibição de imagens coloridas", *Anais do III Simpósio de Computação Gráfica e Processamento de Imagens*, pp. 184-191, junho de 1990.

Apêndice A

Linguagem para Definição de Sólidos CSG

1. Introdução

Uma facilidade essencial para um sistema de modelagem de sólidos é a maneira pela qual os sólidos são descritos. No caso de modelos construtivos como é o caso de CSG, essas descrições são naturalmente simples, consistindo basicamente de uma relação de sólidos primitivos componentes (esferas, cilindros, blocos, etc) aos quais são associadas informações relativas às suas dimensões e ao seu posicionamento dentro universo de modelagem; uma vez que esses componentes estão definidos é preciso então caracterizar as operações de conjunto que os relacionam.

Idealmente, um sistema para a entrada desses dados consistiria de uma interface gráfica onde os objetos sendo modelados poderiam ser visualizados interativamente e em tempo real. No sentido deste ideal muitos grupos de pesquisa têm se dedicado nos últimos anos, sendo que os resultados práticos de que temos conhecimento ainda são limitados e de qualquer maneira requerem equipamentos de arquitetura dedicada e de grande poder de processamento.

Como solução de compromisso, resolvemos criar uma linguagem formal que permitisse uma expressão o mais natural possível dessas informações. A esta linguagem demos o nome de Linguagem de Definição de Sólidos CSG ou, abreviadamente, LDS. O texto que se segue é uma explanação concisa dessa linguagem. Instruções sobre a utilização do programa compilador bem como dos outros programas componentes do sistema de modelagem de sólidos CSG podem ser encontradas no apêndice B.

2. Formato geral de um programa LDS

Um "programa" LDS consiste de uma única expressão que corresponde à árvore CSG de um sólido. A cada sólido assim definido é associado um nome. Desta maneira,

em linhas gerais, uma especificação LDS tem o seguinte aspecto:

Nome { *Definição* }

onde:

Nome é o nome associado ao sólido e

Definição é a expressão que define o sólido.

Observe que a linguagem é recursiva na medida que dentro da definição de um sólido pode haver a definição de um outro sólido. O objetivo dessa facilidade é permitir um recurso análogo ao da subprogramação em linguagens de alto nível. Assim, é possível empregar na expressão que define um sólido nomes de sólidos definidos anteriormente. Por exemplo:

```
Solido1 {  
  
    Solido2 { definição de Solido2 }  
  
    Expressão  
  
}
```

onde *Expressao* emprega **Solido2** como um de seus elementos.

Repare no exemplo acima que **Solido1** pode ser visto como o "programa principal", enquanto que **Solido2** é uma "subrotina" que é usada dentro de **Solido1**.

A linguagem de definição de sólidos adota um formato livre, isto é, não há linhas ou colunas determinadas onde escrever o texto. Aconselha-se entretanto utilizar indentação para tornar o código mais legível.

Os identificadores podem possuir até 60 caracteres de comprimento e devem ser constituídos de caracteres alfanuméricos ou do caractere "_" (sublinhado). O primeiro caractere deve ser uma letra. Não há distinção entre caracteres minúsculos e maiúsculos, portanto:

identificador
Identificador
IDENTIFICADOR

são considerados iguais.

É possível também inserir comentários no programa. Para tal utiliza-se o caractere % (porcento). Todo o texto à direita do sinal de porcento é ignorado pelo compilador.

Para que o leitor possa desde já ter uma idéia mais definida de uma especificação em LDS incluímos aqui um exemplo:

```
Solido__Exemplo {
%
% Especificação de um sólido composto por um
% paralelepípedo atravessado por três hastes
%
Haste {
%
% Haste é um cilindro de raio 0.2, comprimento
% 2, com sua base sobre o plano XZ
%
ROTACAO [-90,0,0]
ESCALA [0.2,2,0.2]
CILINDRO
}

Placa {
%
% Placa é um paralelepípedo de dimensões
% XYZ = 5 x 0.2 x 3
% centrado sobre a origem dos eixos coordenados
%
TRANSL [-0.5,-0.5,-0.5]
ESCALA [5,0.2,3]
BLOCO
}
```

```
%  
% "Programa" principal  
%  
Placa  
+ Haste  
+ TRANSL [-1.5,0,0] Haste  
+ TRANSL [1.5,0,0] Haste  
}
```

O código acima corresponde ao sólido ilustrado na figura A.1 abaixo.

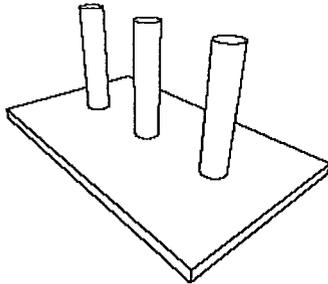


Figura A.1
Um sólido exemplo

3. Sólidos Primitivos

Nesta implementação estão disponíveis os seguintes sólidos primitivos:

- **BLOCO** - Bloco cúbico de lado unitário situado no primeiro octante com um vértice na origem.
- **ESFERA** - Esfera de raio unitário centrada na origem

- **CILINDRO** - Cilindro raio e altura unitários com a base sobre o plano $Z = 0$, topo sobre o plano $Z = 1$ e centrado sobre o eixo coordenado Z .
- **PLANO** - Semiespaço plano correspondente à inequação $Z > 0$

O "sólido" PLANO foi incluído no sistema para facilitar a criação de poliedros.

4. Instanciando Sólidos

Os sólidos primitivos em suas posições e tamanhos originais não são gerais o suficiente; por conseguinte é preciso que se disponha de operadores de instanciação capazes de deformá-los e posicioná-los dentro do universo de interesse. Com este objetivo, esta implementação dispõe dos operadores translação, rotação e escala.

Para aplicar esses operadores a um sólido primitivo ou a um sólido composto utiliza-se a seguinte sintaxe:

TRANSL <vetor> <sólido> ou
ROTACAO <vetor> <sólido> ou
ESCALA <vetor> <sólido>

<vetor> é um conjunto de três números reais entre colchetes e separados por vírgula, correspondentes às coordenadas X , Y e Z , isto é:

[X , Y , Z]

TRANSL significa translação, isto é, o sólido citado a seguir será movido no espaço segundo as coordenadas do vetor. Por exemplo,

TRANSL [10, 15, 30] BLOCO

denota um cubo de lado unitário deslocado 10 unidades no eixo X , 15 unidades no eixo Y e 30 unidades no eixo Z .

ROTACAO indica uma rotação sucessiva do sólido ao redor de cada um dos eixos coordenados. Os valores de *<vetor>* são os ângulos de rotação expressos em graus (o sentido de rotação é ilustrado pela figura A.2.)

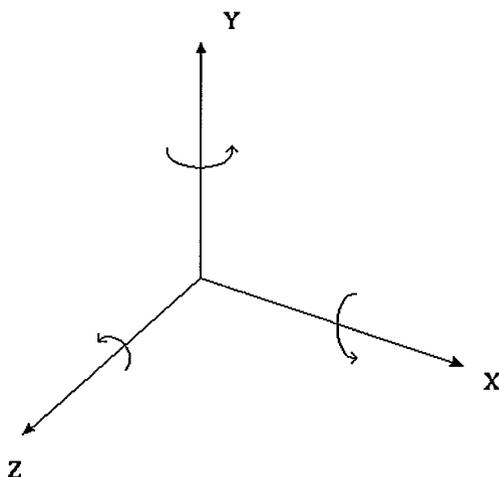


Figura A.2

Sentido de rotação em torno dos eixos coordenados

Por exemplo,

ROTACAO [10, 20, 0] CILINDRO

produziria um cilindro girado 10 graus ao redor do eixo X e, subsequentemente, girado 20 graus ao redor do eixo Y.

ESCALA permite alterar as proporções e o tamanho dos sólidos. É importante notar que não é possível especificar escala 0 em nenhuma das coordenadas de *<vetor>*. Por exemplo, a expressão

ESCALA [5, 4, 3] BLOCO

proporcionaria um paralelepípedo 5 vezes mais largo, 4 vezes mais alto e 3 vezes mais

profundo que um cubo.

A ordem de precedência desses operadores é sempre da esquerda para a direita.
Por exemplo:

```
ESCALA [2,1,1]
  TRANSL [2,0,0]
    ROTACAO [10,20,30]
      BLOCO
```

corresponde a:

1. Dobrar a dimensão X do bloco;
2. Transladá-lo 2 unidades no eixo X;
3. Rodá-lo 10 graus sobre o eixo X;
4. Rodá-lo 20 graus sobre o eixo Y, e
5. Rodá-lo 30 graus sobre o eixo Z

Para realizar essas operações na ordem inversa, poderíamos usar o seguinte trecho:

```
ROTACAO [0,0,30]
  ROTACAO [0,20,0]
    ROTACAO [10,0,0]
      TRANSL [2,0,0]
        ESCALA [2,1,1]
          BLOCO
```

Os operadores de instanciação também podem ser aplicados a expressões, de maneira que é possível aplicar um conjunto de operadores a um sólido ou a uma parte de um sólido.

Exemplos:

TRANSL [1, 1, 0] ROTACAO [30, 0, 60] BLOCO

ESCALA [2, 2, 0] (
ROTACAO [10,0,0] CILINDRO - ESFERA
)

5. Especificando o material

Além dos operadores de instanciação geométricos, existe ainda um quarto operador de instanciação cuja finalidade é definir um material para o sólido. Isto vai ocasionar, quando da visualização, na escolha de uma cor para a imagem do sólido. A sintaxe deste operador é a seguinte:

MATERIAL [*<mat>*] *<sólido>*

onde *<mat>* é um número indicativo do material. Caso o operador de material não seja aplicado ao sólido, é assumido o código de material 1.

Por exemplo:

MATERIAL [2] BLOCO

faz com que um cubo composto do material de número 2 seja gerado.

A relação entre o código do material e sua aparência visual é estabelecida durante o processo de visualização. O programa **Shaded**, responsável pela geração de imagens sombreadas, possui uma tabela interna que associa uma série de constantes relativas ao modelo de iluminação para cada código de material de 1 a 64. Essa tabela padrão pode ser modificada pelo usuário para associar parâmetros diferentes a estes ou outros materiais na faixa de 1 a 255, permitindo inclusive o mapeamento de texturas (xadrez, listado, etc). Veja o apêndice C para uma descrição dos materiais padrões bem como do procedimento a ser efetuado para a modificação da tabela.

6. Operadores de conjuntos

A montagem de um sólido em CSG se faz basicamente através de operações entre conjuntos de pontos - são as conhecidas operações de união, interseção e complemento (ou diferença). Na linguagem de definição de sólidos esses operadores são caracterizados por símbolos, a saber:

- + União
- * Interseção
- Diferença

Por exemplo:

BLOCO + TETRAEDRO

simboliza um sólido composto da união de um BLOCO com uma ESFERA.

Todos os operadores booleanos possuem a mesma precedência, de maneira que as expressões são avaliadas da esquerda para a direita. Assim, o sólido

ESFERA - CILINDRO * TETRAEDRO

indica que o cilindro será subtraído da esfera e então interceptado com o tetraedro. Note que as operações booleanas não são comutativas.

Os operadores de conjunto têm menor precedência que os de instanciação. Então, no sólido

TRANSL [0.3,0.9,0.3] BLOCO + ESFERA

o operador TRANSL está sendo aplicado apenas ao BLOCO e não ao conjunto BLOCO + ESFERA.

É possível entretanto alterar a ordem de avaliação das expressões usando parênteses, por exemplo:

```
TRANSL [0.3,0.9,0.3] (BLOCO + ESFERA)
```

7. Usando sólidos definidos externamente

Depois de definido, é comum querermos aproveitar um sólido para definir outro mais complexo. A maneira mais imediata de fazer isso é repetir a definição do mais simples como "subrotina" do mais complexo. Por exemplo:

```
TESTE {  
  CRUZ {  
    ESCALA [5, 1, 1] BLOCO +  
    ESCALA [1, 5, 1] BLOCO  
  }  
  CRUZ - ESCALA [0.8,0.8,1.2] CRUZ  
}
```

Existe entretanto uma maneira mais prática de fazer o mesmo, desde que o sólido mais simples já tenha sido compilado: em vez de repetir sua definição, escrevemos apenas seu nome e acrescentamos o símbolo @. Quando isso é feito, o compilador carregará a definição do sólido existente no arquivo .COD correspondente. (o arquivo .COD deverá estar presente no diretório corrente). O exemplo acima poderia então ser escrito:

```
TESTE {  
  CRUZ @  
  CRUZ - ESCALA [0.8,0.8,0.8] CRUZ  
}
```

onde CRUZ.COD é um arquivo presente no diretório corrente.

8. Gramática BNF

<sólido> ::=

<nome sólido> { *<bloco>* } |

<nome sólido> @

<bloco> ::=

<sólido> *<bloco>* |

<expressão>

<expressão> ::=

<primitiva> |

<nome sólido> |

<transformação> *<vetor>* *<expressão>* |

<expressão> *<operador>* *<expressão>* |

<tipo material> *<expressão>* |

(*<expressão>*)

<primitiva> ::=

BLOCO |

ESFERA |

CILINDRO |

PLANO

<transformação> ::=

TRANSL |

ROTACAO |

ESCALA

<vetor> ::=

[*<número>*,*<número>*,*<número>*]

<operador> ::=

+

-

*

<tipo material> ::=
MATERIAL [*<inteiro>*]

<número> ::=
<inteiro> |
<inteiro>.*<inteiro>*

<inteiro> ::=
<dígito> |
*<dígito>**<inteiro>*

<dígito> ::=
0|1|2|3|4|5|6|7|8|9

<nome> ::=
<letra> |
*<letra>**<sequência>*

<sequência> ::=
<caractere> |
*<caractere>**<sequência>*

<letra> ::=
A|B|C|D|E|F|G|H|I|
J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z|a|
b|c|d|e|f|g|h|i|j|
k|l|m|n|o|p|q|r|s|
t|u|v|w|x|y|z

<caractere> ::=
_ |
<letra> |
<dígito>

Apêndice B

Códigos LDS dos sólidos de teste

1. Teste1

```
Teste1 {  
  Cantoneira {  
    escala [0.2,1,1] bloco +  
    escala [1,0.2,1] bloco  
  }  
  Furo {  
    rotacao [-90,0,0]  
    escala [0.1, 1, 0.1]  
    transl [0.5, -0.5, 0.5] cilindro  
  }  
  Cantoneira - Furo  
}
```

2. Teste2

```
Teste2 {  
  Haste {  
    rotacao [-90, 0, 0] escala [0.1, 1, 0.1] cilindro  
  }  
  Copo {  
    (transl [0.0, 2.0, 0.0]  
    (  
      escala [1, 2, 1] esfera -  
      escala [0.8, 1.9, 0.8] esfera  
    )  
  )  
}
```

```
    )
  ) *
  escala [2, 2, 2] transl [-1, 0, -1] bloco
}
Base {
  rotacao [-90, 0, 0] escala [1, 0.1, 1] cilindro
}
rotacao [10, 20, 0] (
  base +
  haste +
  transl [0, 0.9, 0] copo
)
}
```

3. Teste3

```
Teste3 {
  ParDeDentes {
    Dente {
      rotacao [-90,0,0] escala [0.1,0.5,0.1] cilindro
    }
    transl [2,0,0] dente +
    transl [2,0,0] rotacao [0,30,0] dente
  }
  Placa {
    rotacao [-90,0,0] escala [2.5, 0.2, 2.5] cilindro
  }
  Placa +
  (
    (
      ParDeDentes +
      rotacao [0,60,0] ParDeDentes
    ) +
  (
```

```
        rotacao [0,120,0] ParDeDentes +
        rotacao [0,180,0] ParDeDentes
    )
) +
(
    rotacao [0,240,0] ParDeDentes +
    rotacao [0,300,0] ParDeDentes
)
}
```

4. Teste4

```
Teste4 {
    Haste {
        rotacao [-90,0,0] escala [0.05, 1, 0.05] cilindro
    }
    Bola {
        escala [0.2, 0.2, 0.2] esfera
    }
    ParDeHastes {
        Haste + transl [0, 0, 1] Haste
    }
    Cubo {
        ParDeHastes +
        transl [1, 0, 0] ParDeHastes +
        rotacao [90, 0, 0] transl [0,1,0] (
            ParDeHastes +
            transl [1, 0, 0] ParDeHastes
        ) +
        rotacao [0,0,-90] transl [0,1,0] (
            ParDeHastes +
            transl [1, 0, 0] ParDeHastes
        )+
    }
}
```

```
        Bola +
        transl [1,0,0] Bola +
        transl [0,1,0] Bola +
        transl [1,1,0] Bola +
        transl [0,0,1] Bola +
        transl [1,0,1] Bola +
        transl [0,1,1] Bola +
        transl [1,1,1] Bola
    )
}
Cubo
}
```

5. Teste5

```
Teste5 {
    Haste {
        rotacao [-90,0,0] escala [0.05, 1, 0.05] cilindro
    }
    Bola {
        escala [0.2, 0.2, 0.2] esfera
    }
    ParDeHastes {
        Haste + transl [0, 0, 1] Haste
    }
    Cubo {
        ParDeHastes +
        transl [1, 0, 0] ParDeHastes +
        rotacao [90, 0, 0] transl [0,1,0] (
            ParDeHastes +
            transl [1, 0, 0] ParDeHastes
        ) +
        rotacao [0,0,-90] transl [0,1,0] (
            ParDeHastes +
```

```
transl [1, 0, 0] ParDeHastes
)+
(
  Bola +
  transl [1,0,0] Bola +
  transl [0,1,0] Bola +
  transl [1,1,0] Bola +
  transl [0,0,1] Bola +
  transl [1,0,1] Bola +
  transl [0,1,1] Bola +
  transl [1,1,1] Bola
)
}
Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.30,0.30,0.30] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.30,0.70,0.30] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.70,0.30,0.30] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.70,0.70,0.30] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.30,0.30,0.70] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.30,0.70,0.70] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.70,0.30,0.70] Cubo +
transl [-0.5,-0.5,-0.5] escala [0.25,0.25,0.25]
transl [0.70,0.70,0.70] Cubo
}
```

6. Teste6

```
Teste6 {
  cil {
    transl [0,0,-0.5] escala [0.3,0.3,2] cilindro
  }
  material [2] transl [-0.5, -0.5, -0.5] bloco +
  rotacao [90,0,0] cil +
  material [3] rotacao [0, 90, 0] cil -
  cil
}
```

7. Teste3_A

```
Teste3_A {
  ParDeDentes {
    Dente {
      rotacao [-90,0,0] escala [0.1,0.5,0.1] cilindro
    }
    transl [2,0,0] dente +
    transl [-2,0,0] dente
  }
  Placa {
    rotacao [-90,0,0] escala [2.5, 0.2, 2.5] cilindro
  }
  Placa +
  (
    (
      ParDeDentes +
      rotacao [0,30,0] ParDeDentes
    ) +
    (
      rotacao [0,60,0] ParDeDentes +

```

```
        rotacao [0,90,0] ParDeDentes
    )
) +
(
    rotacao [0,120,0] ParDeDentes +
    rotacao [0,150,0] ParDeDentes
)
}
```

8. Teste3_B

```
Teste3_B {
    Dente {
        rotacao [-90,0,0] escala [0.1,0.5,0.1] cilindro
    }
    Dente1 {
        transl [2,0,0] dente
    }
    Dente2 {
        transl [-2,0,0] dente
    }
    ParDeDentes {
        Dente1 + Dente2
    }
    Placa {
        rotacao [-90,0,0] escala [2.5, 0.2, 2.5] cilindro
    }
    Placa +
    ParDeDentes +
    rotacao [0,30,0] ParDeDentes +
    rotacao [0,60,0] ParDeDentes +
    rotacao [0,90,0] ParDeDentes +
    rotacao [0,120,0] ParDeDentes +
    rotacao [0,150,0] ParDeDentes
}
```

}

9. Teste3_C

```
Teste3_C {
  Dente {
    rotacao [-90,0,0] escala [0.1,0.5,0.1] cilindro
  }
  Dente1 {
    transl [2,0,0] dente
  }
  Dente2 {
    transl [-2,0,0] dente
  }
  Placa {
    rotacao [-90,0,0] escala [2.5, 0.2, 2.5] cilindro
  }
  Placa +
  Dente1 +
  Dente2 +
  rotacao [0,30,0] Dente1 +
  rotacao [0,30,0] Dente2 +
  rotacao [0,60,0] Dente1 +
  rotacao [0,60,0] Dente2 +
  rotacao [0,90,0] Dente1 +
  rotacao [0,90,0] Dente2 +
  rotacao [0,120,0] Dente1 +
  rotacao [0,120,0] Dente2 +
  rotacao [0,150,0] Dente1 +
  rotacao [0,150,0] Dente2
}
```

Apêndice C

Guia para executar os programas

1. Introdução

Neste apêndice apresentaremos de forma sumária as informações necessárias para a execução dos programas que compõem o sistema de visualização de sólidos CSG. Um manual mais completo está em fase de conclusão e estará disponível oportunamente.

Atualmente, tanto os fontes quanto os códigos executáveis desses programas podem ser encontrados no Laboratório de Computação Gráfica da COPPE-UFRJ. Os arquivos estão residentes no diretório "claudio" e podem ser acessados por qualquer usuário habilitado a fazer uso das estações de trabalho *Sun*.

Fazem parte do sistema os seguintes programas:

- *CompilaCsg*: Compilador da Linguagem de Definição de Sólidos.
- *WireFrame*: Gerador de gráficos de linhas.
- *Shaded*: Gerador de imagens sombreadas.
- *FigToImg*: Programa para transformar arquivos "raster" com 24 "bit planes" em imagens com no máximo 256 cores distintas (8 "bit planes").

2. CompilaCsg

Este programa processa um arquivo de especificação LDS gerando um arquivo binário contendo uma árvore CSG correspondente ao sólido.

2.1. Linha de comando

```
CompilaCsg {-L} {-A} <nome>
```

onde

-L Lista arquivo fonte

-A Lista árvore CSG Gerada

<nome>

nome do arquivo .CSG a ser compilado

2.2. Semântica

Os arquivos processados por este programa devem se encontrar no diretório corrente. Não há necessidade de citar a extensão ".CSG" do arquivo fonte, o programa a provê automaticamente.

O arquivo binário é gerado com o mesmo <nome> do arquivo fonte ao qual é acrescentado a extensão ".COD".

3. Wireframe

Este programa exhibe um gráfico de linhas correspondente a um sólido CSG dado por um arquivo gerado pelo programa *CompilaCsg*

3.1. Linha de Comando

```
WireFrame {-X<xmin>,<xmax>}
           {-Y<ymin>,<ymin>} {-P<nomeproj>}
           {-N<numpixel> {-S<tamsub>}}
           {-A{<tamamostra>}} {-O} <nome>
```

onde

-X Coordenadas X da janela de projeção.

<xmin>

Valor mínimo da coordenada X da janela. Por default é estimado pelo programa.

<xmax>

Valor máximo da coordenada X da janela. Por default é estimado programa.

-Y Coordenadas Y da janela de projeção.

<ymin>

Valor mínimo da coordenada Y da janela. Por default é estimado programa.

<ymin>

Valor máximo da coordenada Y da janela. Por default é estimado programa.

-P Especifica um arquivo (".PRJ") que define o sistema projetivo a ser empregado durante a visualização.

<nomeproj>

Nome do arquivo ".PRJ".

-N Especifica número de pixels da imagem.

<numpixel>

Numero de pixels da maior coordenada da janela. Default:128.

-A Altera a amostragem na geração do gráfico de linhas.

<tamamostra>

Tamanho da amostra. Default:4.

-O Oculta arestas com mesma normal.

-Q Termina o programa imediatamente após a exibição da imagem. Por default, o programa aguarda que o usuário aperte um botão do ratinho.

-S Especifica tamanho da grade de subdivisão.

<tamsub>

Tamanho da subdivisao. Default:8.

<nome>

Nome do arquivo ".COD" a ser visualizado.

3.2. Semântica

O programa *WireFrame* lê um arquivo contendo o código de um sólido CSG, gerado pelo programa *CompilaCsg* e gera uma imagem do tipo "gráfico de linhas". A imagem, além de ser exibida na tela, é armazenada num arquivo cujo nome é igual ao do arquivo que define o sólido, acrescentado com a extensão ".WFR".

O programa amostra a imagem disparando raios sobre pontos selecionados da tela. Este processo pode impedir a reprodução de detalhes mais finos. Isto pode ser corrigido alterando-se o parâmetro "-A" para um valor menor. Similarmente, especificando um valor maior para "-A" obtém-se uma visualização mais rápida.

4. Shaded

Gera imagens sombreadas de um sólido CSG.

4.1. Linha de Comando

```
Shaded {-X<xmin>,<xmax>}  
        {-Y<ymin>,<ymin>}{-P<nomeproj>}  
        {-N<numpixel> {-S<tamsub>}}  
        {-A} {-F<nomefonte>} <nome>
```

onde

-X Coordenadas X da janela de projeção.

<xmin>

Valor mínimo da coordenada X da janela. Por default é estimado pelo programa.

<xmax>

Valor máximo da coordenada X da janela. Por default é estimado programa.

-Y Coordenadas Y da janela de projeção.

<ymin>

Valor mínimo da coordenada Y da janela. Por default é estimado programa.

<ymax>

Valor máximo da coordenada Y da janela. Por default é estimado programa.

-P Especifica um arquivo (".PRJ") que define o sistema projetivo a ser empregado durante a visualização.

<nomeproj>

Nome do arquivo ".PRJ".

-N Especifica número de pixels da imagem.

<numpixel>

Numero de pixels da maior coordenada da janela. Default:128.

-S Especifica tamanho da grade de subdivisão.

-A Usa técnica de antiserrilhado para melhorar a imagem gerada.

<tamsub>

Tamanho da subdivisão. Default:8.

<nome>

Nome do arquivo ".COD" a ser visualizado.

-F Especifica um arquivo (".FNT") que define a localização e a composição das fontes de iluminação pontuais utilizadas na cena.

-T Especifica um arquivo de texturas (".TTX").

<textura>

Nome do arquivo ".TTX".

4.2. Semântica

A imagem gerada por "Shaded" é obtida através de um processo de disparo de raios em toda a área definida para o plano de projeção. É utilizado um modelo de iluminação simples que trata componentes devidas a iluminação ambiente, difusa e especular.

O programa exibe na tela uma aproximação da imagem calculada. Isto deve-se ao fato de termos disponíveis dispositivos capazes de exibir toda a gama de cores que são computadas. Entretanto, a imagem é armazenada em um arquivo (com extensão ".FIG") onde cada pixel é representado no sistema de cores RGB no qual cada componente possui 8 bits. Este arquivo pode servir de base a um processo de discretização onde é calculada uma tabela de cores apropriada, permitindo uma boa aproximação da imagem com até 256 cores distintas. O programa *FigToImg* realiza esse processo.

5. FigToImg

Realiza um processo através do qual é possível reproduzir uma imagem onde cada pixel é representado por 24 bits em um dispositivo de exibição matricial de 8 bits por pixel.

5.1. Linha de Comando

```
FigToImg {-M<funcao>} {-D} <nome>
```

onde

-M Altera a função para o cálculo da distância entre cores no espaço RGB. Por *default*, a distância entre duas cores é dada pela maior diferença entre suas coordenadas.

<funcao>

E: distância euclideana; *M* -> distância *manhattan*.

-D Faz com que o programa, além de escolher uma tabela de cores apropriada, utilize um algoritmo de difusão de erros para aproximar melhor áreas com uma variação suave de tons de cor.

<nome>

Nome do arquivo que contém a imagem em 24 bits por pixel.

5.2. Semântica

O programa implementa o algoritmo para exibição de imagens coloridas descrito em COMBA [50]. A imagem resultante é armazenada num arquivo cujo nome é composto de <nome> com a extensão ".IMG".