Síntese de Imagem e Rastreamento de Raio

Hae Yong Kim

Dissertação apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
para a obtenção do grau de
Mestre em Matemática Aplicada.

Área de Concentração: Ciência de Computação
Orientador: Prof. Dr. Routo Terada

São Paulo, julho de 1992

Contents

[1] Boundary Representation.
[2] Non-manifold.
[3] Constructive Solid Geometry.
[4] Parabolic blending.
[5] Lofted or ruled surface.

[6] Tradução livre de linear coons surface. Coon ou racoon é animal carnívoro noturno americano, de família do urso, semelhante ao guaximim, com cauda longa e espessa. Também pode significar a pele desse animal.

[7] Blobby model.

[8] Digital Differential Analyzer.

[9] Edge flag.
[10] Ordered edge list.
[11] Ray-casting.
[12] Buffer significa pára-choque. Portanto, a tradução literal seria pára-choque z.

[13] Painter's algorithm.
[14] Half-tone. Às vezes traduzido para português como meia tonalidade.
[15] Ordered dither.

Chapter 1

Introdução e Agradecimentos

Pode-se afirmar que os algoritmos de síntese de imagem[1] estão entre os algoritmos mais utilizados na Computação Gráfica. Podemos dizer que eles são, atualmente, tão importantes quanto os algoritmos de ordenação ou indexação de arquivo. Qualquer programa que trabalhe com objetos tridimensionais seria inútil se esses objetos não pudessem ser mostrados na tela ou impressos. Qualquer sistema CAD[2] precisa implementar um ou mais algoritmos de síntese de imagem para permitir que o usuário veja o objeto tridimensional que está sendo construído. Os algoritmos de síntese de imagem também são utilizados visualizar as funções matemáticas $\mathbf{R}^2 \rightarrow \mathrm{R}$, para criar filmes ou comerciais de televisão, assim como nos simuladores de vôo. Observe que muitas das aplicações citadas acima movimentam grandes somas monetárias, o que torna estes algoritmos economicamente importantes. Outro interesse para o estudo desses algoritmos é o fato deles necessitarem um tempo de processamento muito grande. Assim, há um interesse enorme em descobrir algoritmos mais rápidos. Existem computadores paralelos construídos especialmente para acelerar a síntese de imagem, utilizados, por exemplo, nos simuladores de vôo. Atualmente, gastam-se horas, dias e até meses de processamento nos computadores de grande porte para gerar as imagens para poucos minutos de filme ou de comercial de televisão.

Do ponto de vista técnico, os algoritmos de síntese de imagem são fascinantes pela diversidade de soluções que apresentam. Neles utiliza-se uma grande variedade de técnicas computacionais e teorias matemáticas. Existem algoritmos tipo "dividir e conquistar" enquanto que um outro utiliza a programação linear. Existem algoritmos baseados na ordenação assim como existe um outro baseado na na teoria de transferência de calor por irradiação. Por fim, as pessoas que estudam este assunto e que têm acesso a um computador razoavelmente rápido com uma boa resolução gráfica podem receber a grata recompensa de ver, literalmente, os frutos do seu trabalho.

Esta dissertação apresenta, de um modo sistemático, as técnicas e teorias associadas com a síntese de imagem, dispersas por vários livros e artigos. Foram selecionados os algoritmos e as técnicas que o autor julgou serem de maior interesse prático. Os algoritmos sem interesses práticos são apresentados na

[1] Também chamados de algoritmos de visualização.
[2] Computer Aided Design.

medida em que a compreensão deles é necessária para entender o funcionamento de outros algoritmos. O autor procurou, na medida do possível, não omitir os detalhes dos algoritmos.

Neste trabalho foram desenvolvidos dois algoritmos inéditos. São eles:

1. Baseado no método de perturbação e na invisibilidade quantitativa, é desenvolvido um novo algoritmo, de complexidade $O(n\ logn)$ no caso médio, para eliminar as arestas escondidas, denominado pelo autor de algoritmo de engorda.

2. O rastreamento de raio bidirecional, que consegue calcular corretamente as sombras de objetos transparentes.

Infelizmente, estes dois algoritmos ainda não foram implementados. O algoritmo de engorda já está sendo implementado. O autor pretende começar tão logo possível a implementação do rastreamento bidirecional. Além destes algoritmos, é desenvolvida pela primeira vez uma técnica para calcular a probabilidade de falha do método de perturbação. Muitos dos algoritmos aqui apresentados foram implementados pelo autor ou pelos seus alunos e as imagens geradas podem ser vistas ao longo deste trabalho. Entre estas implementações, destaca-se o programa RAIOS, um rastreador de raio que foi matéria das seguintes reportagens:

- *Tese na USP cria um programa* publicada no Caderno de Informática do jornal *O Estado de S. Paulo* (pg 16) no dia 2 de dezembro de 1991.

- *Professor desenvolve sistema para CAD* publicada no Caderno de Informática do jornal *Folha de S. Paulo* (caderno 5, pg 11) no dia 15 de abril de 1992.

Alguns trechos desta dissertação já foram publicados em anais de congressos e em revistas científicas:

- O artigo *Como Calcular a Probabilidade de Falha do Método de Perturbação* foi publicado nos anais da Quarta Semana de Informática da UFBA (pp 13–20). O evento ocorreu entre os dias 6 e 10 de abril de 1992.

- O artigo *RAIOS – uma Implementação de Ray Tracing* foi aceito para a publicação pelo Corpo Editorial da Revista Brasileira de Computação.

Acreditamos que muitas outras partes deste trabalho possam ser publicadas em revistas especializadas. Optamos por não utilizar trema, pois a tendência das boas editoras é deixar de usar trema nos seus livros. Os termos técnicos foram traduzidos de inglês para português exceto as siglas consagradas pelo uso, como CSG, B-rep, etc. Também não foram traduzidos os termos cujas traduções tornam-se estranhas, como z-buffer cuja tradução literal seria pára-choque z.

O autor gostaria de agradecer em primeiro lugar ao professor Routo Terada que orientou este trabalho com muita paciência. Também ao professor Antônio Elias que me incentivou o estudo dos modeladores de sólidos. Ao professor Marcos Gubitoso que me ensinou a utilizar as estações de trabalho Sparc e o

modelador de sólidos Noodles. Ao Eduardo Wronowski que está implementando o algoritmo de engorda. Aos meus antigos alunos de iniciação científica e do curso de computação gráfica que implementaram vários algoritmos descritos neste trabalho.

Part I

Técnicas Básicas

Chapter 2

Síntese de Imagem

2.1 Áreas de Computação Gráfica

A Computação Gráfica trabalha basicamente com dois tipos de estruturas de dados que podem ser chamadas de modelo e imagem. Um modelo é uma representação abstrata de um objeto que pode ser, por exemplo, as equações das faces do objeto. Uma imagem é uma matriz onde cada elemento representa a intensidade luminosa de um ponto. Uma foto digitalizada é, por exemplo, uma imagem. A Computação Gráfica pode ser subdividida em quatro grandes áreas de acordo com a estrutura de dados com a qual trabalha: processamento de imagem, análise de imagem, modelagem geométrica e síntese de imagem[1].

Processamento de imagem: Esta área trabalha sobre uma imagem que já existe, alterando-a. Não trabalha sobre modelos. Alguns exemplos desta área de Computação Gráfica são:

1. Algumas fotografias das revistas coloridas foram processadas por computador para enfatizar as bordas. O computador detecta as regiões onde há uma mudança brusca de cor, por exemplo, de marrom escuro para azul claro. Depois, torna a borda marrom mais escura ainda e a borda azul mais clara do que antes. Isto ressalta os objetos, facilitando distingui-los.

2. Para imprimir uma imagem digitalizada numa impressora a laser, a imagem deve ser antes processada pelo algoritmo de meio tom, pois a impressora a laser consegue imprimir somente pontos pretos ou brancos e numa imagem queremos ter as tonalidades de cinza. O mesmo problema aparece quando se precisa imprimir fotos num jornal. Este algoritmo procura colocar mais pontos pretos (ou então colocar pontos maiores) nas regiões mais escuras e coloca menos pontos (ou pontos de tamanhos menores) nas regiões mais claras (veja a figura 2.2).

[1]Uma observação a fazer é que a tradução literal do termo "computer graphics" para português não é "computação gráfica" mas "gráficos do computador" ou "gráficos gerados por computador" e corresponde mais propriamente à síntese de imagem.

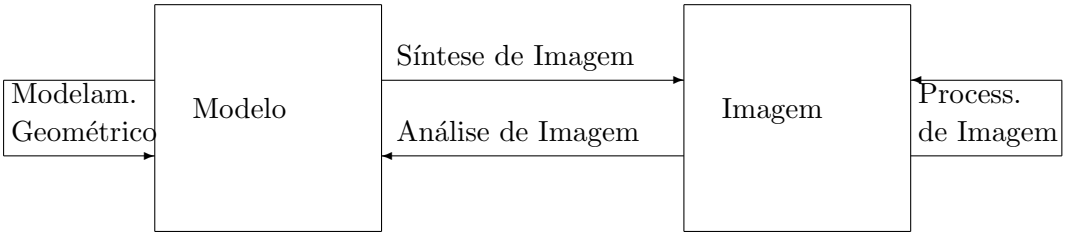


Figure 2.1: Áreas de Computação Gráfica

3. A compactação de imagens também faz parte do processamento de imagem. Existem métodos de compactação de imagens que, utilizando fractais, chega à taxa de compressão de 10000 para 1. Veja [Barnsley, 88].

4. Um outro exemplo bem conhecido de processamento de imagem é retocar fotografias com o auxílio do computador. Existem programas comerciais que permitem eliminar, por exemplo, as rugas de uma fotografia.

Análise de imagem: Os algoritmos de análise de imagem procuram reconhecer o que uma imagem representa. O ponto de partida é uma imagem e se deseja obter o modelo correspondente. Alguns exemplos:

1. A partir da imagem digitalizada de uma página do livro é possível reconhecer quais são as letras que compõe aquela página. O resultado final deste reconhecimento é um arquivo texto que pode ser editado em qualquer editor de texto.

2. Outro exemplo é a visão robótica. O programa deve analisar a imagem capturada por uma câmara de vídeo reconhecendo quais são e a que distância estão os objetos presentes nessa imagem.

Modelagem geométrica: Esta área trabalha somente sobre os modelos. O ponto de partida são os modelos simples, chamados modelos primitivos. Efetuando certas operações sobre eles, constroem-se os modelos mais complexos. Ou seja, a modelagem geométrica estuda as diferentes maneiras de se descrever um objeto. Exemplos:

1. Num modelador de sólidos, partindo de sólidos simples cria um sólido complexo através das operações booleanas (união, intersecção e diferença) e transformações geométricas (rotação, translação e mudança de escala).

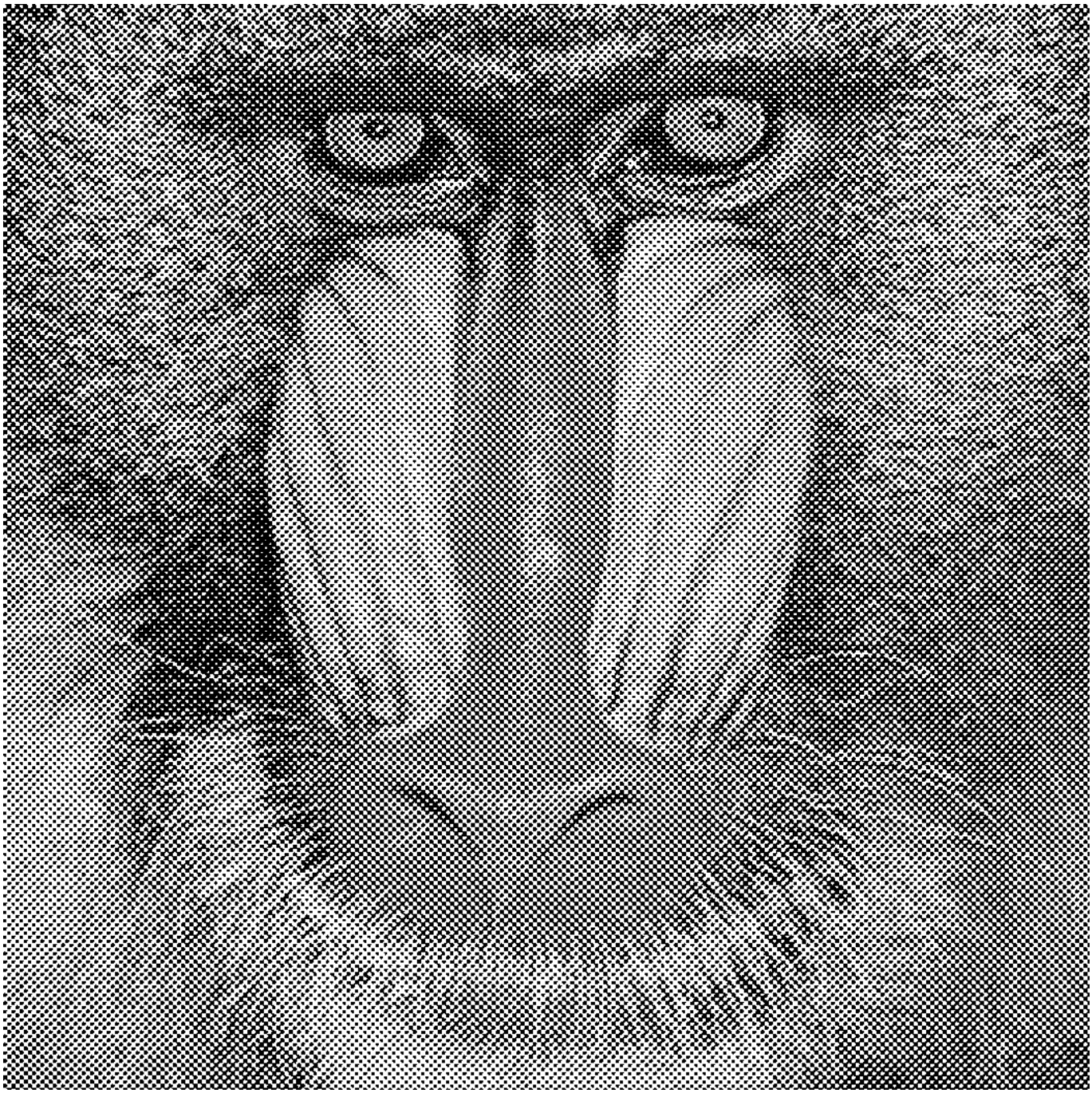


Figure 2.2: Uma imagem processada por algoritmo de meio tom e para ser impressa por uma impressora a laser.

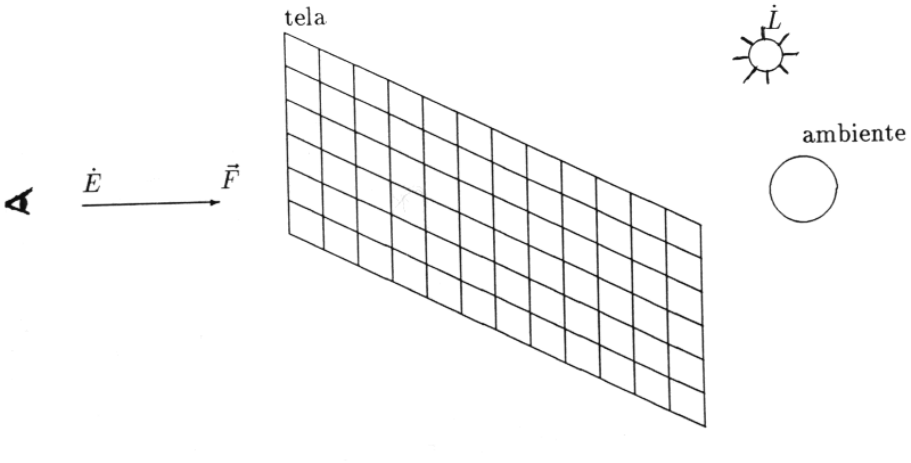


Figure 2.3: Problema de síntese de imagem

2. No CAD[2], utilizando segmentos de retas e outras figuras simples cria desenhos complexos.

3. Existem técnicas especiais para descrever objetos extremamente complexos como uma nuvem, uma montanha ou uma explosão de fogos de artifício.

Síntese de imagem Por fim, chegamos ao assunto deste trabalho. A síntese de imagem consiste em criar uma imagem a partir de um modelo. Exemplos desta área da Computação Gráfica, como já foi citado no Capítulo 1, são:

1. Filmes e comerciais de televisão gerados por computador.

2. Simuladores de vôo.

3. Visualização de sólidos criados por modeladores de sólidos.

4. Visualização de gráficos matemáticos tridimensionais.

2.2 Formulação de Problema

O problema de síntese de imagem pode ser formulada da seguinte maneira:

Definição 2.1 *Dadas a descrição de um ambiente no espaço, a posição do observador $\dot{E}$ e a direção de visão $\vec{F}$, calcule a imagem vista pelo observador.*

[2]Computer Aided Design.

Se queremos obter imagens mais realistas, é necessário fornecermos também as posições ($\dot{L}$) e as intensidades ou os espectros[3] ($\bar{L}$) das fontes de luz. Utilizaremos as siglas $\dot{E}, \vec{F}, \dot{L}$ e $\bar{L}$ ao longo deste trabalho. Explicando o vocabulário usado na definição 2.1, temos:

Definição 2.2 *Um* **ambiente** *é a totalidade dos objetos a serem mostrados.*

Definição 2.3 *Uma* **imagem** *em preto e branco é uma matriz de números inteiros onde cada elemento da matriz representa a intensidade luminosa de um pixel. Uma imagem colorida é composta de três matrizes inteiras, uma matriz para cada cor primária.*

Definição 2.4 *Um* **pixel**[4] *é a menor porção de uma imagem. É um ponto da tela de computador. Um pixel não é necessariamente quadrado, pois pode ser retangular.*

Definição 2.5 *A* **tela** *é o espaço de memória usado para armazenar uma imagem. Toda informação colocada na tela pode ser vista pelo usuário.*

Definição 2.6 *Quando queremos obter uma imagem em preto e branco, a* **intensidade da fonte de luz** *deve ser um número real. Se desejamos obter uma imagem colorida, o espectro da fonte de luz deve ter três componentes reais: vermelho, verde e azul. Num ambiente pode haver mais de uma fonte de luz.*

Observe que o problema de síntese de imagem é mais amplo e engloba o problema de eliminação de superfícies escondidas. No problema de eliminação de superfícies escondidas, só estamos interessados em calcular quais partes das superfícies são visíveis ao observador. Não estamos preocupados se a imagem criada é realista ou não. O problema de síntese de imagem, por sua vez, está interessado em todas as técnicas para criar uma imagem bidimensional a partir da descrição de um ambiente tridimensional. Algumas dessas técnicas gerarão imagens muito realistas porém consumirão muito tempo de processamento, como é o caso de rastreamento de raio e radiosidade. Outras técnicas gerarão imagens esquemáticas mas serão muito rápidas computacionalmente, como o algoritmo do horizonte flutuante e outros algoritmos que geram imagens "armação de arame[5]".

[3] Intensidade dos diferentes comprimentos de onda. Veja a seção 7.1.
[4] Picture element.
[5] Wireframe.

Chapter 3

Modelagem Geométrica

Os algoritmos de síntese de imagem dependem estritamente do modelo utilizado para descrever o ambiente. Não é possível sintetizar uma imagem se não se conhece como descrever um objeto tridimensional. Deste modo, as diferentes maneiras de se descrever um objeto tridimensional, ou seja a modelagem geométrica, serão expostas antes de se explicar a síntese de imagem propriamente dita.

Definição 3.1 *Um* **modelo** *é um objeto construído artificialmente que facilita a observação de um outro objeto.*

Por exemplo, um modelo físico é um objeto concreto, tridimensional, de coisas tais como prédios, navios e carros. Um modelo molecular reproduz o arranjo de átomos numa molécula. Um modelo matemático representa algum aspecto de um fenômeno modelado em termos de dados numéricos e equações. Um modelo computacional consiste de dados armazenados num arquivo de computador que pode ser usado para efetuar as tarefas similares aos outros tipos de modelos. Do mesmo modo,

Definição 3.2 *A totalidade de dados necessários para representar a forma geométrica de um objeto é chamado de* **modelo geométrico**.

Convém salientar que a área de modelagem geométrica é muito ampla. Não é objetivo deste trabalho aprofundar no assunto. Desta maneira, muitos temas de modelagem geométrica não foram abordados aqui por apresentarem pouco interesse para o problema de síntese de imagem. Outros temas foram apresentados breve e superficialmente, somente o que é útil para o problema de síntese de imagem. Boas referências para a modelagem geométrica são [Requicha, 80], [SolidMod] e [MathElem].

3.1 Primitivos Básicos

Os modelos primitivos mais básicos são armazenados no computador da seguinte maneira:

Ponto:

É representado como uma sequência de três números reais, $\dot{P} = (P_x, P_y, P_z)$. Como veremos no capítulo 4, às vezes é mais conveniente representar um ponto do espaço como uma sequência de quatro números reais, $\dot{P} = (P_x, P_y, P_z, P_w)$. Esta representação é chamada de coordenadas homogêneas. Na linguagem C, temos a seguinte declaração do ponto: `typedef struct {double x,y,z,w;} PONTO.`

Linha:

Um segmento de reta (também chamado de aresta ou linha) é representado como dois pontos. Na linguagem C, temos a seguinte declaração: `typedef struct {PONTO p1,p2;} LINHA.`

Polígono:

É representado como uma sequência de pontos. Pode ser armazenado num vetor ou numa lista ligada circular. Vamos escrever um polígono G como $(\dot{G}_0, \dot{G}_1, ..., \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$ ou como $(\dot{G}_0, \dot{G}_1, ..., \dot{G}_{n-1})$. Um polígono no plano pode estar orientado em sentido horário ou anti-horário. Por exemplo, $G = ((0,0),(1,0),(0,1))$ está em sentido anti-horário enquanto que $H = ((0,0),(0,1),(1,0))$ está em sentido horário.

Um polígono com furos será representado como uma série de sequências de pontos. A primeira sequência representará o contorno externo do polígono e cada uma das demais sequências representará um furo. Assim, por exemplo, o polígono da figura 3.1 será representado por

$$G = ((G_0, G_1, G_2, G_3); (G_4, G_5, G_6); (G_7, G_8, G_9))$$

Se a primeira sequência estiver em sentido anti-horário, as demais sequências estarão em sentido horário e vice-versa. Uma outra maneira de representar um polígono com furos é criar as "arestas fantasmas". Assim, o polígono da figura 3.1 também pode ser representado por:

$$G = (\dot{G}_0, \dot{G}_4, \dot{G}_5, \dot{G}_6, \dot{G}_4, \dot{G}_0, \dot{G}_1, \dot{G}_2, \dot{G}_3, \dot{G}_7, \dot{G}_8, \dot{G}_9, \dot{G}_7, \dot{G}_3, \dot{G}_0)$$

Onde as arestas $(\dot{G}_0, \dot{G}_4)$, $(\dot{G}_4, \dot{G}_0)$, $(\dot{G}_3, \dot{G}_7)$ e $(\dot{G}_7, \dot{G}_3)$ são fantasmas.

3.2 B-Rep — Representação através de contornos[1]

Definição 3.3 *Um modelo* **B-rep** *modela um sólido através da representação de suas superfícies.*

Suponha que queremos modelar o paralelepípedo da figura 3.2. Precisamos armazenar os vértices, as arestas, as faces e algumas outras informações geométricas. Estes dados constituem o modelo B-rep. Existem muitos modelos B-rep diferentes. Podemos subdividi-los em dois grandes grupos: modelos que só trabalham com poliedros e modelos que também trabalham com arestas e faces curvas.

[1] Boundary Representation.

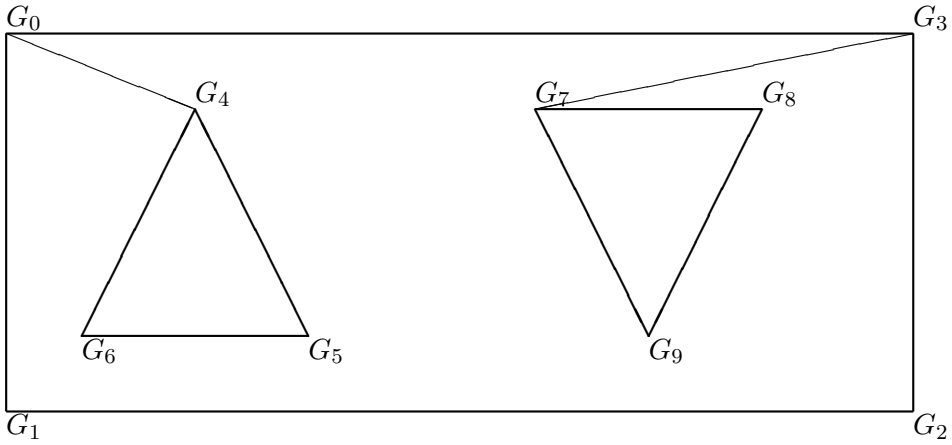


Figure 3.1: Polígono com furos e arestas fantasmas (linhas finas)

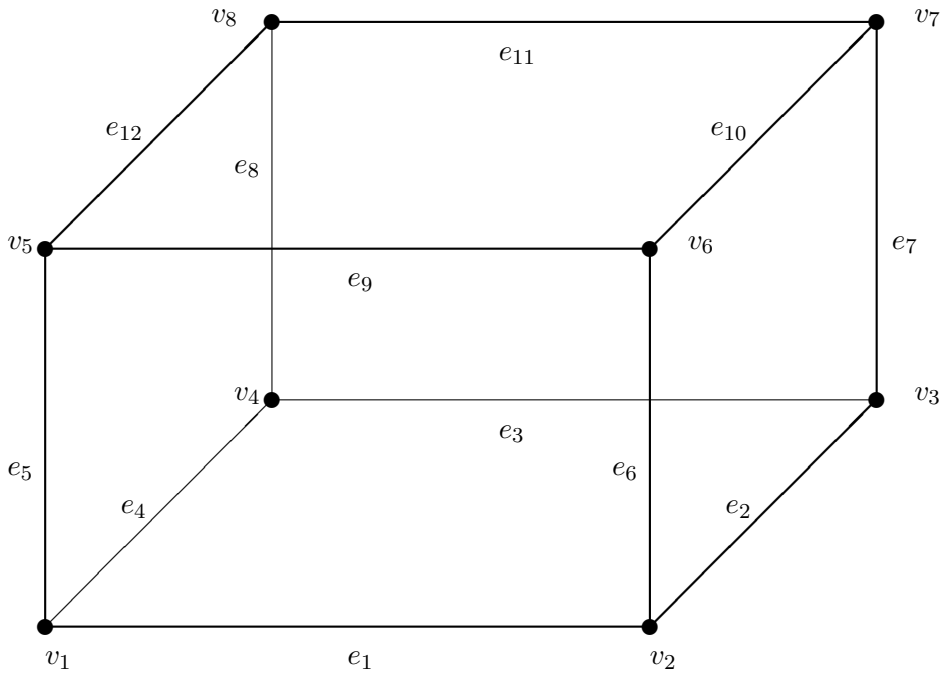


Figure 3.2: Sólido modelado em B-rep

vértice	coordenadas	face	vértices
v_1	x_1, y_1, z_1	f_1	v_1, v_2, v_3, v_4
v_2	x_2, y_2, z_2	f_2	v_6, v_2, v_1, v_5
v_3	x_3, y_3, z_3	f_3	v_7, v_3, v_2, v_6
v_4	x_4, y_4, z_4	f_4	v_8, v_4, v_3, v_7
v_5	x_5, y_5, z_5	f_5	v_5, v_1, v_4, v_8
v_6	x_6, y_6, z_6	f_6	v_8, v_7, v_6, v_5
v_7	x_7, y_7, z_7		
v_8	x_8, y_8, z_8		

Figure 3.3: Modelo B-rep baseado em vértice

3.2.1 Modelo B-rep para poliedros

Num modelo B-rep poliedral temos, na estrutura de dados do modelo, vértices, arestas e faces. Uma aresta é um segmento de reta que tem como extremidades dois vértices. Uma face é um polígono delimitado por um conjunto de arestas. Normalmente, uma face é orientada de modo que se possa distinguir o lado interno do externo. Serão apresentados aqui, como exemplos, dois modelos B-rep poliedrais simples. Os exemplos foram tirados de [SolidMod, pp 101–106].

Modelo B-rep baseado no vértice:

O objeto da figura 3.2 pode ser representado pelo modelo da figura 3.3. Nele, armazenamos as coordenadas dos vértices num vetor e cada face é representada como uma sequência de vértices. Note que as sequências dos vértices devem estar numa ordem consistente: ou sempre no sentido horário ou sempre no sentido anti-horário, quando visto de fora do sólido. No exemplo da figura 3.3, utilizou-se a ordem horária. A orientação consistente é útil em muitos algoritmos. Por exemplo, na eliminação de superfícies escondidas, essa orientação é utilizada para detectar as faces irrelevantes, como veremos na seção 6.1.

Modelo B-rep baseado na aresta:

O objeto da figura 3.2 também pode ser representado pelo modelo da figura 3.4. Armazenamos as coordenadas dos vértices num vetor. Uma aresta é representada como um par de vértices. Uma face é, por sua vez, representada como uma sequência de arestas. Como no modelo anterior, as faces estão orientadas consistentemente. A orientação de uma face é dada pelo sentido da sua primeira aresta. Por exemplo, a face f_2 tem como primeira aresta e_9. Logo, a orientação da face é de v_5 para v_6, de v_6 para v_2, etc. Há outros modelos B-rep mais elaborados que não foram descritos aqui.

aresta	vértices	vértice	coordenadas	face	arestas
e_1	v_1, v_2	v_1	x_1, y_1, z_1	f_1	e_1, e_2, e_3, e_4
e_2	v_2, v_3	v_2	x_2, y_2, z_2	f_2	e_9, e_6, e_1, e_5
e_3	v_3, v_4	v_3	x_3, y_3, z_3	f_3	e_{10}, e_7, e_2, e_6
e_4	v_4, v_1	v_4	x_4, y_4, z_4	f_4	e_{11}, e_8, e_3, e_7
e_5	v_1, v_5	v_5	x_5, y_5, z_5	f_5	e_{12}, e_5, e_4, e_8
e_6	v_2, v_6	v_6	x_6, y_6, z_6	f_6	$e_{12}, e_{11}, e_{10}, e_9$
e_7	v_3, v_7	v_7	x_7, y_7, z_7		
e_8	v_4, v_8	v_8	x_8, y_8, z_8		
e_9	v_5, v_6				
e_{10}	v_6, v_7				
e_{11}	v_7, v_8				
e_{12}	v_8, v_5				

Figure 3.4: Modelo B-rep baseado em aresta

face	vértices
f_1	(x_4, y_4, z_4) (x_3, y_3, z_3) (x_2, y_2, z_2) (x_1, y_1, z_1)
f_2	(x_6, y_6, z_6) (x_2, y_2, z_2) (x_1, y_1, z_1) (x_5, y_5, z_5)
f_3	(x_6, y_6, z_6) (x_2, y_2, z_2) (x_3, y_3, z_3) (x_7, y_7, z_7)
f_4	(x_4, y_4, z_4) (x_8, y_8, z_8) (x_7, y_7, z_7) (x_3, y_3, z_3)
f_5	(x_5, y_5, z_5) (x_1, y_1, z_1) (x_4, y_4, z_4) (x_8, y_8, z_8)
f_6	(x_8, y_8, z_8) (x_7, y_7, z_7) (x_6, y_6, z_6) (x_5, y_5, z_5)

Figure 3.5: Modelo B-rep simplificado

Modelo B-rep simplificado:

Para muitos algoritmos de síntese de imagem que trabalham apenas com poliedros, não há necessidade de dispor-se de todas as informações que um modelo B-rep fornece. Para esses algoritmos, todo o ambiente pode ser descrito simplesmente como um conjunto de polígonos sem relações entre si. Assim, o poliedro da figura 3.2 pode ser representado pelos dados da figura 3.5. A diferença principal entre este modelo e os anteriores é que aqui não há uma orientação coerente das faces. Isto implica que é praticamente impossível determinar se um ponto pertence ou não ao interior do sólido. Porém, muitos algoritmos de síntese de imagem não necessitam desta informação. Podemos dizer que este modelo representa as superfícies e os anteriores representam os sólidos.

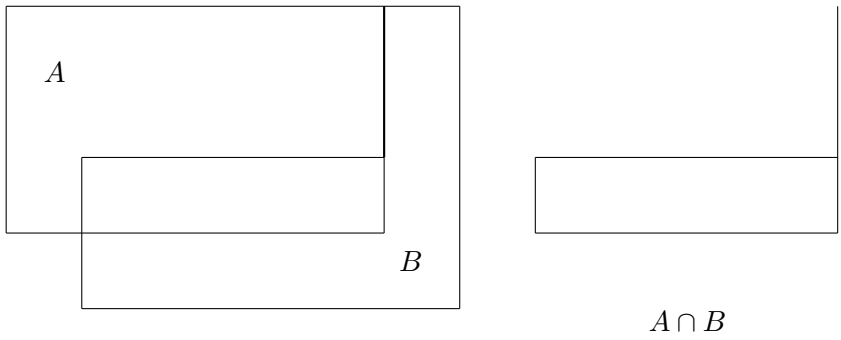


Figure 3.6: Geração de superfície não-variedade através de uma operação booleana.

3.2.2 Modelo B-rep para objetos com faces curvas

Um modelo B-rep cujas faces podem ser curvas pode ser criado utilizando splines ou técnicas assemelhadas. Dados os pontos de controle podemos criar uma superfície curva que passa por eles. Veja a seção 3.4 e [MathElem, pp 165-186].

Aplicando certas transformações geométricas sobre os pontos de controle (veja o capítulo 4), todo o objeto deve sofrer a mesma transformação. Por exemplo, se rotacionarmos os pontos de controle 30^o em torno de uma reta R, todo o objeto deve sofrer essa mesma rotação. O mesmo vale para a translação e a mudança de escala.

3.2.3 Modelo não-variedade[4]

Tradicionalmente, um modelador de sólidos procurava modelar um objeto físico real completa e eficientemente. No processo de criação de modelo e na análise, porém, as abstrações são frequentemente usadas. Portanto, é desejável permitir que as abstrações sejam modeladas, assim como os objetos físicos, no mesmo ambiente de modelagem. Isto levou a criar um novo conceito na modelagem geométrica: modelar tanto os objetos não-físicos como os objetos físicos. Esta nova técnica é chamada modelagem geométrica não-variedade.

O modelador de sólidos Noodles é capaz de representar objetos compostos de faces planas, arestas e vértices permitindo que o usuário escolha o nível de representação entre os modelos armação de arame,

[4]Non-manifold.

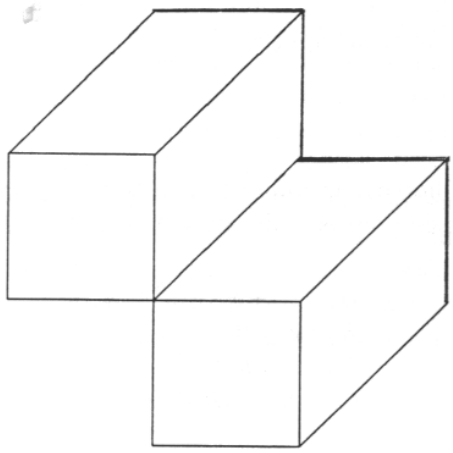


Figure 3.7: Sólido não-variedade

superfície e sólido. É capaz de representar até os pontos isolados no espaço. Outra vantagem de modelo não-variedade é ser fechado sob as operações booleanas. Na figura 3.6, a intersecção de dois polígonos gerou uma figura impossível de ser representada em modelo tradicional. Porém, ela pode ser representada num modelo não-variedade. A figura 3.7 mostra um sólido não-variedade.

3.3 CSG — Geometria Construtiva de Sólidos[5]

3.3.1 Modelo semi-espaço

Todo conjunto de pontos $A \subseteq \mathbf{R}^3$ pode ser pensado como tendo uma função característica g_A associada.

$$g_A : \mathbf{R}^3 \longrightarrow \{0, 1\}$$

$$g_A(\dot{X}) = \begin{cases} 1 & \text{se } \dot{X} \in A \\ 0 & \text{se } \dot{X} \notin A \end{cases}$$

Qualquer conjunto de pontos de $\mathbf{R}^3$ pode ser representado através da sua função característica. Porém, representar um conjunto qualquer de $\mathbf{R}^3$ através da sua função característica pode não ser útil, pois a função característica pode ser tão complexo quanto o próprio conjunto. Porém, existe uma classe de conjuntos de pontos que pode ser representada facilmente através de funções analíticas. Consideraremos que

[5]Constructive Solid Geometry.

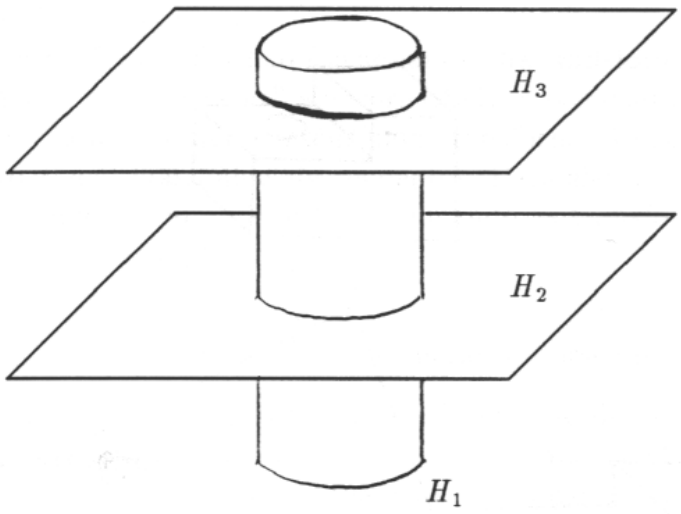


Figure 3.8: Operações booleanas sobre semi-espaços

$$\begin{cases} \dot{X} \in A \text{ se } f_A(\dot{X}) \leq 0 \\ \dot{X} \notin A \text{ se } f_A(\dot{X}) > 0 \end{cases}$$

Exemplos:

Semi-espaço: $f(\dot{X}) = ax + by + cz + d$
Cilindro infinito: $f(\dot{X}) = x^2 + y^2 - r^2$
Esfera: $f(\dot{X}) = x^2 + y^2 + z^2 - r^2$
Sendo x, y e z as coordenadas de $\dot{X}$.

O cone infinito e o toróide também podem ser representados da mesma maneira. A superfície $f(\dot{X}) = 0$ divide o espaço em dois subconjuntos. Por isso, cada um dos dois subconjuntos do espaço determinados por estas superfícies são chamados de semi-espaços.

3.3.2 Operações booleanas

Podemos aplicar as operações booleanas união ($\cup$), intersecção ($\cap$) e diferença ($\setminus$) sobre os semi-espaços para modelar os conjuntos de pontos mais complexos. Exemplo (veja a figura 3.8):

$$H_1 : x^2 + y^2 - r^2 \leq 0$$
$$H_2 : z \geq 0$$
$$H_3 : z - h \leq 0$$
$$C = H_1 \cap H_2 \cap H_3$$

Definição 3.4 *Vamos chamar de* **primitivos** *de um modelo os conjuntos de pontos básicos a partir dos quais, efetuando certas operações, modelamos os conjuntos mais complexos.*

Os semi-espaços determinados pelas funções analíticas são, portanto, os primitivos do modelo semi-espaço.

3.3.3 Modelo CSG

Modelo semi-espaço puro é matematicamente rigoroso. Para o usuário, porém, é mais fácil trabalhar com os modelos primitivos limitados, como cubos, cilindros limitados, esferas, cones limitados, etc. Utilizando os primitivos limitados, o usuário não pode gerar, por engano, sólidos ilimitados como o resultado das operações booleanas. O modelo que trabalha com primitivos limitados chama-se modelo CSG (Geometria Construtiva de Sólidos). Observe que, na teoria, cada primitivo de CSG foi descrito como uma combinação booleana de semi-espaços. Na prática, porém, isto pode não ser interessante. Assim, um cilindro limitado é mais facilmente representável armazenando, em vez das equações de um cilindro e de dois planos, simplesmente as duas extremidades do eixo do cilindro e o seu raio. A esfera, por sua vez, pode ser representada armazenando o centro e o raio. Além das operações booleanas, vamos acrescentar no modelo as transformações geométricas tais como a rotação, a translação e a mudança de escala. Então, um objeto modelado em CSG pode ser representado através da árvore CSG, definido como (veja a figura 3.9):

$$\text{<árvore CSG>::=} \begin{cases} \text{<primitivo> |} \\ \text{<árvore CSG><operação booleana><árvore CSG> |} \\ \text{<árvore CSG><transformação geométrica>} \end{cases}$$

Uma árvore CSG é armazenada na memória do computador utilizando a estrutura de dados árvore. Consequentemente, fazer uma operação booleana ou uma transformação geométrica sobre um modelo CSG é praticamente instantânea, pois basta acrescentar um nó na árvore CSG. Porém, qualquer algoritmo de síntese de imagem que rodar sobre o modelo CSG será lento, pois todas as operações da árvore precisam ser avaliadas quando se aplica o algoritmo.

Muitos algoritmos para modelos CSG não funcionam se há transformações geométricas na árvore. A solução para este problema é "pré-processar" a árvore CSG, convertendo-a numa árvore equivalente sem as transformações. Um nó da árvore que contém uma transformação geométrica T pode ser eliminado aplicando a transformação T nas suas folhas descendentes. Para efetuar esta operação, sugerimos que se percorra a árvore em pós-ordem ou em pré-ordem, aplicando as transformações. Uma árvore CSG sem as transformações geométricas é definida como:

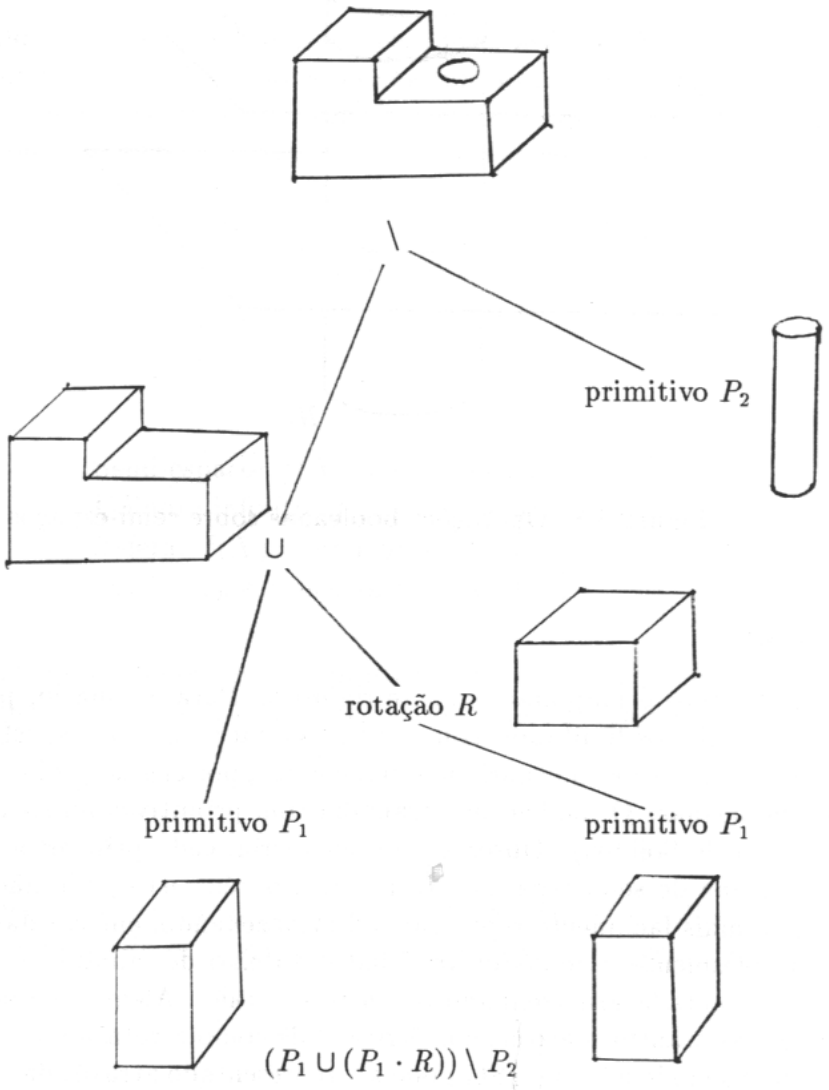


Figure 3.9: Árvore CSG

$$\text{<árvore CSG>::=} \begin{cases} \text{<primitivo>} \mid \\ \text{<árvore CSG><operação booleana><árvore CSG>} \end{cases}$$

Existem modeladores B-rep que dão aos usuários a impressão de estarem trabalhando com um modelo CSG, pois oferecem as mesmas operações de um modelador CSG (exemplo: noodles). Porém, internamente, a cada operação booleana que efetua, está sendo calculado o modelo B-rep resultante. Os algoritmos para efetuar as operações booleanas num modelo B-rep são bastante complexos e lentos (eles têm complexidades quadráticas). Em compensação, quase todos os algoritmos de síntese de imagem rodam mais rápidos num modelo B-rep do que num modelo CSG. Por fim, existem modeladores híbridos que trabalham temporariamente com o modelo CSG e, somente quando o usuário tiver conseguido modelar o sólido definitivo, o modelo B-rep correspondente é gerado.

3.3.4 Modelo CSG simplificado

Chamaremos de modelo CSG simplificado aquele modelo onde só se pode efetuar a operação de união e as transformações geométricas. O algoritmo de rastreamento de raio trabalha normalmente com este tipo de modelo.

3.4 Arestas e superfícies curvas

Para modelar arestas e superfícies curvas são necessárias técnicas diferentes daquelas expostas nas seções anteriores. Nesta seção, descreveremos algumas delas, mas muitas outras foram sugeridas e estão em uso com graus de sucesso variados. Na literatura, têm aparecido constantemente novas idéias e técnicas nesta área.

Antes de descrever os métodos em detalhe, convém ter em mente algumas idéias fundamentais a respeito das equações que descrevem curvas ou superfícies. Existem dois tipos básicos de equações: implícita e explícita (também chamada de paramétrica). Uma superfície implícita é descrita por uma função $\dot{F}(x, y, z) = 0$ e uma superfície explícita é descrita por $\dot{P}(u, w) = [\, x(u, w),\ y(u, w),\ z(u, w)\,]$. Por exemplo, uma circunferência no plano pode ser descrita implicitamente pela equação $x^2 + y^2 = r^2$ ou explicitamente $\dot{P}(t) = [\, r\ cos(t),\ r\ sen(t)\,],\ 0 \leq t < 2\pi$. Observe que a função característica vista na seção 3.3 é uma equação implícita. Uma superfície implícita é chamada algébrica se somente polinômios são usados na sua equação e, caso contrário, é chamada analítica.

As vantagens das superfícies implícitas são:

1. Facilidade de calcular a classificação dentro/fora utilizada nos algoritmos para CSG.

2. Facilidade de calcular a intersecção raio/superfície utilizada no rastreamento de raio.

As vantagens das superfícies explícitas são:

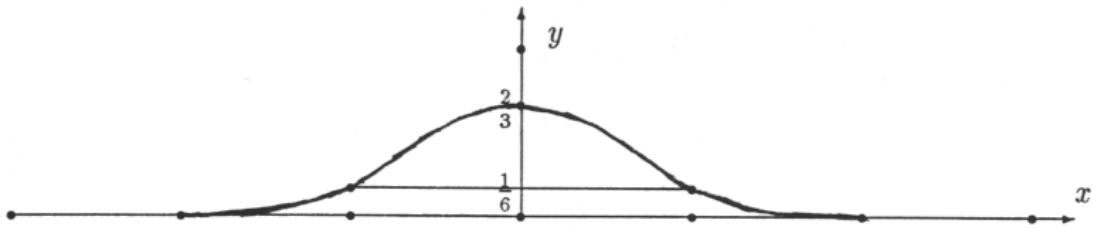


Figure 3.10: Função $L(x)$

1. Facilidade para gerar polígonos que aproximam a superfície.
2. A parametrização da superfície é necessária para alguns tipos de mapeamento de textura.

Para incluir neste trabalho, escolhemos duas técnicas para descrever as curvas paramétricas que julgamos serem práticas, simples de serem implementadas e eficientes. Outras técnicas podem ser encontradas no livro [MathElem].

3.4.1 B-spline com base não-recursiva

[MathElem] apresenta uma técnica para gerar curva B-spline cuja base é uma função recursiva. Computacionalmente, ela não é prática pois exige um grande tempo de processamento para gerar a base. Explicaremos nesta subseção uma técnica apresentada em [ConcBas] cuja base não é recursiva. Acreditamos que as curvas geradas pelos dois métodos sejam muito semelhantes. Convém notar que existem ainda outras técnicas para gerar curvas usando a base B-spline. [CalcNum, 91] é um exemplo disso.

A base de B-spline cúbico será para nós funções da forma $L(x-i)$, onde i é um número inteiro (positivo ou negativo) e $L(x)$ vale:

$$L(x) = \begin{cases} 0 & 2 \leq x \\ (2-x)^3/6 & 1 \leq x < 2 \\ (4-6x^2+3x^3)/6 & 0 \leq x < 1 \\ L(-x) & x < 0 \end{cases}$$

Observe que a função $L(x)$ é recursiva na definição somente por uma questão de facilidade de notação. Essa recursão é facilmente eliminável na implementação e, mesmo que não a elimine, a função pode ser avaliada rapidamente pelo computador.

Sejam dados n pontos de controle $\dot{P}_1, \dot{P}_2, \ldots, \dot{P}_n$. Então a curva paramétrica:

$$\dot{P}(t) = \sum_{i=1}^{n} \dot{P}_i L(t-i), \quad 2 \leq t \leq n-1$$

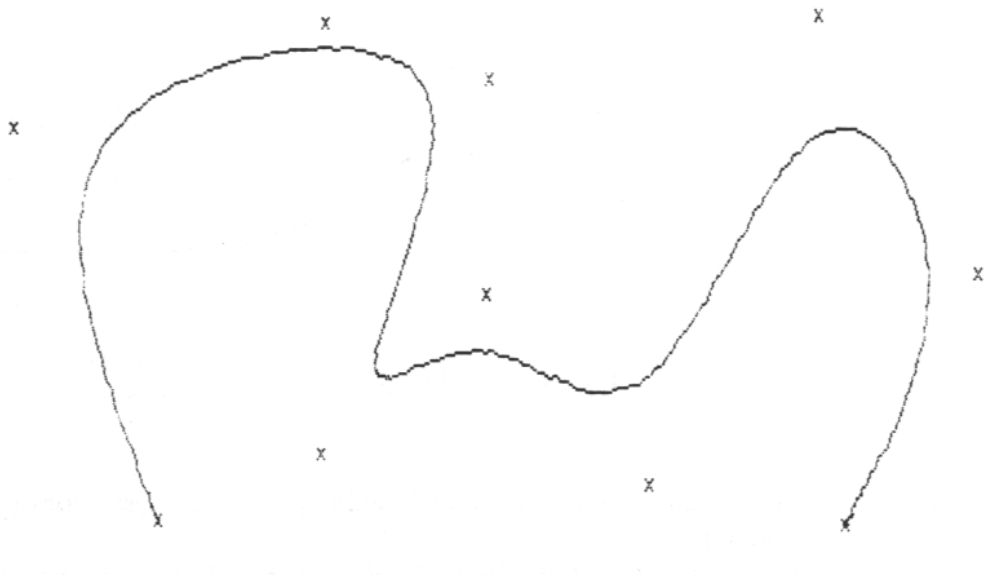


Figure 3.11: Curva B-spline não recursiva.

é uma B-spline e essa curva passa próximo aos pontos $\dot{P}_2, \dot{P}_3, \ldots, \dot{P_{n-1}}$. Porém, nas extremidades ($t < 2$ ou $t > n-1$), a curva se afasta dos pontos $\dot{P}_1$ e do $\dot{P}_n$. Para evitar este inconveniente, repita três vezes o mesmo ponto inicial e final, isto é, faça $\dot{P}_1 = \dot{P}_2 = \dot{P}_3$ e $\dot{P}_{n-2} = \dot{P}_{n-1} = \dot{P}_n$ e trace a curva acima. Ela parte de $\dot{P}_3$, passa próximo aos pontos de controle $\dot{P}_4, \ldots, \dot{P}_{n-3}$ e chega ao $\dot{P}_{n-2}$. De fato, se algum ponto de controle repete-se três vezes seguidas então a curva B-spline passará por esse ponto (mas as derivadas nesse ponto não serão contínuas).

Pode-se imaginar que as funções $L(t-i)$ são "pesos" e que a curva B-spline é uma "média ponderada" dos pontos de controle. O peso de um ponto de controle $\dot{P}_i$ é máximo quando $t = i$ e é zero quando $t < i-2$ ou $i+2 < t$. Uma consequência interessante deste fato é que alterando um ponto de controle $\dot{P}_i$ a curva irá ser alterada somente no trecho $i-2 \leq t \leq i+2$. Esta característica chama-se controle local e tem importância prática pois quando o usuário alterar um dos pontos de controle, o programa precisa alterar somente um pequeno trecho da curva. A curva B-spline é utilizada pela maioria dos programas gráficos interativos.

Se é necessário obter uma curva B-spline fechada com pontos de controle $\dot{P}_3, \ldots, \dot{P}_{n-2}$ faça $\dot{P}_2 = \dot{P}_{n-2}$, $\dot{P}_1 = \dot{P}_{n-3}$, $\dot{P}_{n-1} = \dot{P}_3$ e $\dot{P}_n = \dot{P}_4$ e trace a curva acima para $2 \leq t \leq n-2$. Curva B-spline não recursiva foi implementada pelo autor e o resultado pode ser visto na figura 3.11.

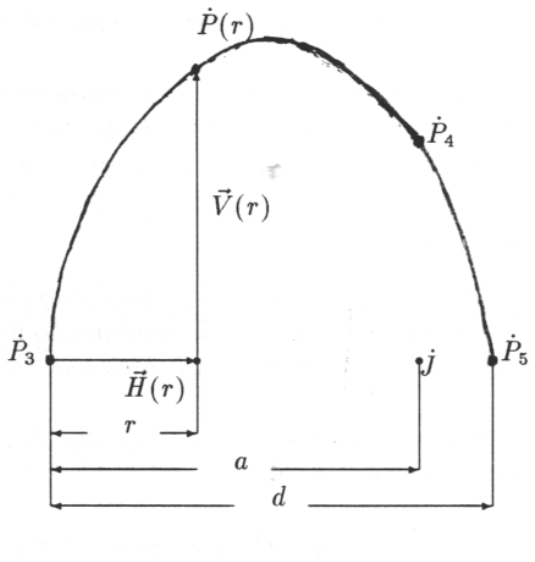


Figure 3.12: Parábola $\dot{P}(r)$

3.4.2 Combinação parabólica[8]

A técnica de combinação parabólica foi sugerida pela primeira vez por [Overhauser, 68]. Uma curva suave entre dois pontos de controle interiores é gerada pela combinação linear de dois segmentos de parábolas. A primeira parábola é definida pelos três primeiros pontos e os três últimos pontos definem a segunda parábola.

Considere dados quatro pontos de controle $\dot{P}_3, \dot{P}_4, \dot{P}_5$ e $\dot{P}_6$. A parábola espacial $\dot{P}(r)$ que atravessa $\dot{P}_3, \dot{P}_4$ e $\dot{P}_5$ é dada pela seguinte soma vetorial:

$$\dot{P}(r) = \dot{P}_3 + \vec{H}(r) + \vec{V}(r), \quad 0 \leq r \leq d$$

Onde:

$$\begin{cases} \vec{H}(r) = r \; versor(\dot{P}_5 - \dot{P}_3) \\ \vec{V}(r) = \alpha \; r \; (d - r) \; versor(\dot{P}_4 - \dot{J}) \end{cases}$$

Ainda falta determinar d, α e $\dot{J}$. A expressão para d é imediata:

$$d = distancia(\dot{P}_5, \dot{P}_3)$$

[8] Parabolic blending.

Vamos definir $a = (\dot{P}_4 - \dot{P}_3) \circ (\dot{P}_5 - \dot{P}_3)/d$. Ou seja, a é a distância entre $\dot{P}_3$ e $\dot{J}$. Então:

$$\dot{J} = \dot{P}_3 + a\ versor(\dot{P}_5 - \dot{P}_3)$$

α pode ser determinada considerando que $\vec{V}(r)$ vale a seguinte expressão quando $r = a$:

$$\vec{V}(a) = \alpha\ a\ (d - a)\ versor(\dot{P}_4 - \dot{J}) = distancia(\dot{P}_4, \dot{J})\ versor(\dot{P}_4 - \dot{J})$$

Isolando, α, obtemos:

$$\alpha = distancia(\dot{P}_4, \dot{J})\ /\ (a\ (d - a))$$

Observe que na realidade existem infinitas parábolas que passam por três pontos não colineares. Nós escolhemos aquela com a diretriz[9] paralela à reta $(\dot{P}_3, \dot{P}_5)$.

De modo análogo, a parábola espacial $\dot{Q}(s)$ que passa pelos pontos $\dot{P}_4, \dot{P}_5$ e $\dot{P}_6$ é:

$$\dot{Q}(s) = \dot{P}_4 + \vec{H}(s) + \vec{V}(s) \quad 0 \leq s \leq e$$

Onde:

$$\begin{cases} \vec{H}(s) = s\ versor(\dot{P}_6 - \dot{P}_4) \\ \vec{V}(s) = \beta\ s\ (e - s)\ versor(\dot{P}_5 - \dot{K}) \\ e = distancia(\dot{P}_6, \dot{P}_4) \\ b = ((\dot{P}_5 - \dot{P}_4)\ (\dot{P}_6 - \dot{P}_4))\ /\ e \\ \dot{K} = \dot{P}_4 + b\ versor(\dot{P}_6 - \dot{P}_4) \\ \beta = distancia(\dot{P}_5, \dot{K})\ /\ (b\ (e - b)) \end{cases}$$

Com isso, estamos preparados para escrever a nossa função interpoladora que passa pelos pontos $\dot{P}_3, \dot{P}_4, \dot{P}_5$ e $\dot{P}_6$:

- Entre $\dot{P}_3$ e $\dot{P}_4$, desenhe a curva $\dot{P}(a\ t)$ com $0 \leq t < 1$.
- Entre $\dot{P}_4$ e $\dot{P}_5$, desenhe a curva $\dot{C}(t) = (1 - t)\ \dot{P}(a + t\ (d - a)) + t\ \dot{Q}(b\ t)$ para $0 \leq t < 1$.
- Entre $\dot{P}_5$ e $\dot{P}_6$ desenhe a curva $\dot{Q}(b + (e - b)\ t)$ para $0 \leq t \leq 1$.

Quando houver mais de quatro pontos de controle, para cada intervalo deve-se criar uma combinação de parábolas diferentes. Esta técnica também tem controle local, isto é, a alteração de um ponto de controle $\dot{P}_i$ só irá refletir no trecho $\dot{P}_{i-1}$ a $\dot{P}_{i+1}$ da curva. A combinação parabólica foi implementada pelo autor e o resultado pode ser visto na figura 3.14. Esta técnica falha se os pontos de controle mudam de direção bruscamente, como pode ser observado no trecho final da curva.

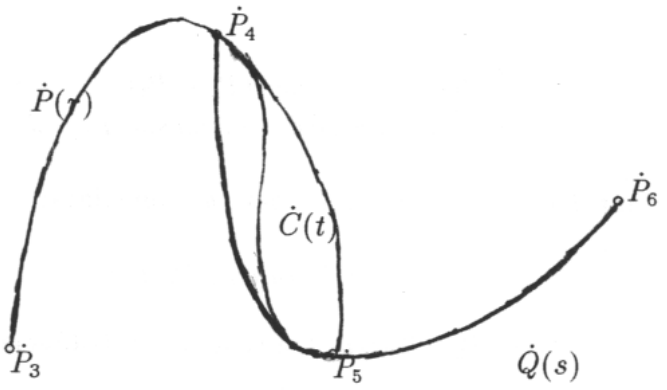


Figure 3.13: Combinação parabólica

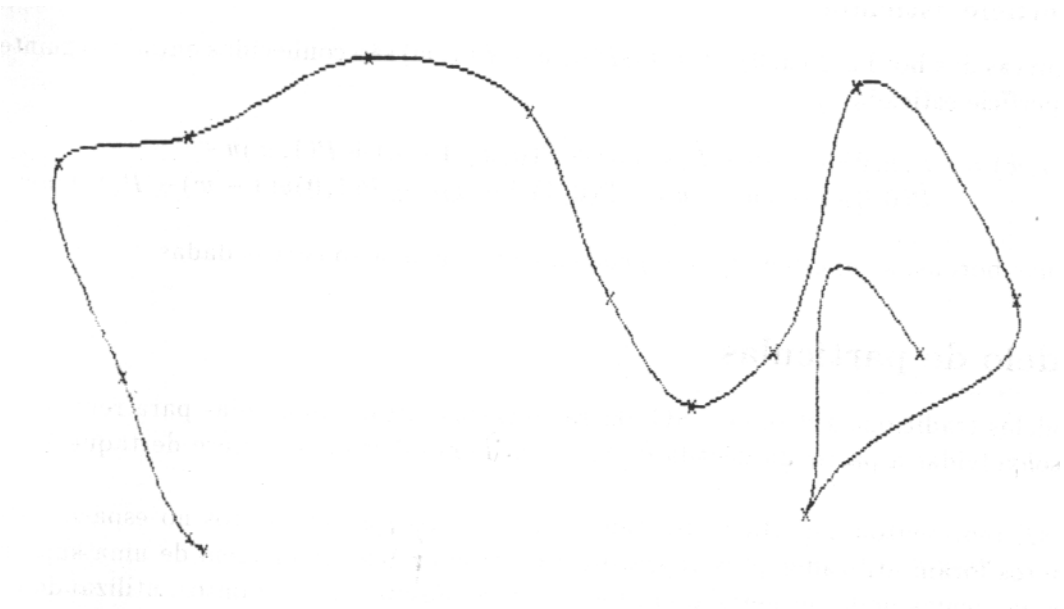


Figure 3.14: Curva gerada pela combinação parabólica.

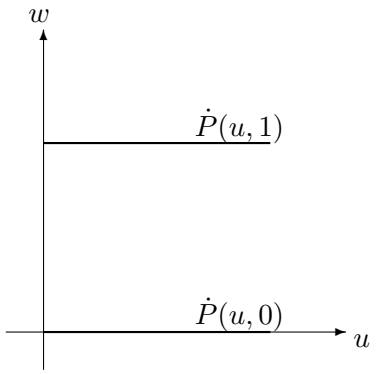


Figure 3.15: Superfície dirigida

3.4.3 Superfície dirigida[10]

Vimos, nas duas subseções anteriores, algumas técnicas para criar curvas paramétricas no espaço. Descreveremos agora duas técnicas que permitem obter superfícies paramétricas a partir das equações das suas bordas. A primeira delas é a superfície dirigida.

Assuma que duas curvas das bordas em lados opostos do quadrado unitário do plano (u, v) é conhecido, digamos $\dot{P}(u, 0)$ e $\dot{P}(u, 1)$ (veja a figura 3.15). Estas curvas podem ser descritas pelos métodos de descrição de curvas já vistos (B-spline ou combinação parabólica). Uma superfície dirigida pelas curvas $\dot{P}(u, 0)$ e $\dot{P}(u, 1)$ é obtida pela interpolação linear entre essas duas curvas:

$$\dot{Q}(u, w) = \dot{P}(u, 0)\ (1 - w) + \dot{P}(u, 1)\ w$$

3.4.4 Superfície esticada[11]

Se as quatro curvas das bordas $\dot{P}(u, 0), \dot{P}(u, 1), \dot{P}(0, w)$ e $\dot{P}(1, w)$ são conhecidas então a seguinte equação descreve a superfície esticada:

$$\begin{aligned}\dot{Q}(u, w) = \quad & \dot{P}(u, 0)(1 - w) + \dot{P}(u, 1)w + \dot{P}(0, w)(1 - u) + \dot{P}(1, w)u - \\ & \dot{P}(0, 0)(1 - u)(1 - w) - \dot{P}(0, 1)(1 - u)w - \dot{P}(1, 0)u(1 - w) - \dot{P}(1, 1)uw\end{aligned}$$

[9] Uma parábola é definida da seguinte maneira: Dados um ponto $\dot{F}$ e uma reta d, pertencentes a um plano α, $\dot{F} \notin d$, a parábola é o conjunto dos pontos de α que estão a mesma distância de $\dot{F}$ e de d. Chamamos a reta d de diretriz da parábola.

[10] Lofted or ruled surface.

[11] Tradução livre de linear coons surface. Coon ou racoon é animal carnívoro noturno americano, de família do urso, semelhante ao guaximim, com cauda longa e espessa. Também pode significar a pele desse animal.

Note que nas bordas, a superfície $\dot{Q}(u, w)$ coincide com as quatro curvas dadas.

3.5 Modelo de partículas

Além dos modelos tradicionais B-rep e CSG, outras técnicas menos conhecidas para representar objetos 3D foram desenvolvidas a partir da década de 80. A primeira delas que merece destaque é o modelo de partículas.

[Norton, 82] representou superfícies fractais como uma coleção de pontos no espaço. Mais de um milhão de pontos foram utilizados para representar cada superfície. A imagem de uma superfície representada pelas partículas pode ser gerada, em tempo linear no número de pontos, utilizando o algoritmo z-buffer. Este é um modelo interessante para representar superfícies muito complexas que não podem ser aproximadas por polígonos.

[Reeves, 83] utilizou o modelo de partículas para representar fogos de artifício e a explosão do planeta do filme *Star Trek II: The Wrath of Khan*. As partículas são criadas estocasticamente, movem no espaço de acordo com as leis físicas e mudam de cor, de transparência e de tamanho no decorrer do tempo.

[Blinn, 82] usou partículas para simular as nuvens e discutiu a função de reflexão para luz que incide nas partículas. Esta técnica foi utilizada para produzir as imagens dos anéis do Saturno. O modelo de partículas também pode ser utilizado para representar fumaças.

3.6 Fractais

Uma outra técnica para modelar objetos 3D são os fractais. Existem muitas técnicas diferentes para gerar diferentes fractais. Os fractais são utilizados para descrever nuvens, montanhas, árvores, etc. Citaremos aqui um método simples para criar curvas e superfícies fractais descrito em [Fournier, 82]. Esta técnica é utilizada para criar montanhas artificiais.

Queremos criar uma função fractal entre os pontos t_1 e t_2. Nestes pontos, a função vale f_1 e f_2, respectivamente. Veja a figura 3.16a. Ache o ponto $t_m = (t_1 + t_2)/2$ e gere uma variável randômica l de média zero e variância unitária. A altura da função fractal no ponto t_m deverá valer $f_m = (f_1 + f_2)/2 + l\,s$, onde s é uma constante. Gere recursivamente a função fractal entre os pontos t_1 e t_m e os pontos t_m e t_2, utilizando uma constante s menor. Esta técnica, escrita em Pascal, fica:

```
procedure fractal(t1,t2,epsilon,h,escala:real; semente:integer);
var f1,f2,razao,std:real;

  procedure subdivide(f1,t1,f2,t2,std:real);
  var tm,fm:real;
  begin
    if t2-t1>epsilon then begin
      tm:=(t1+t2)/2;
```

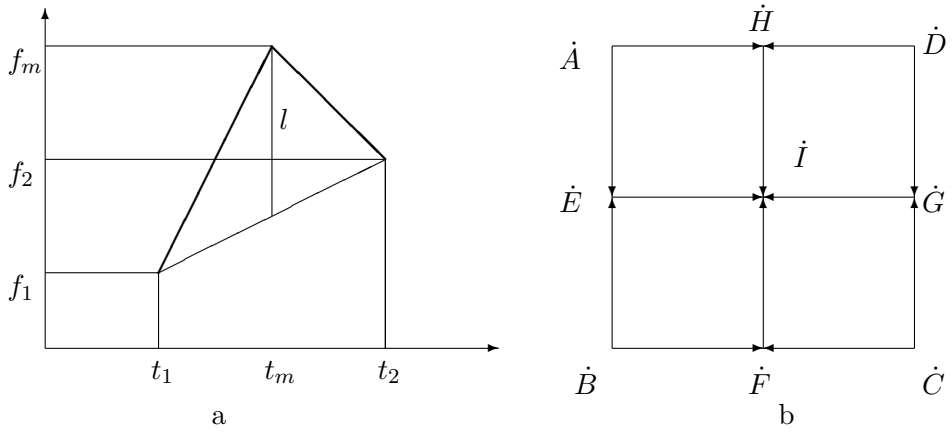


Figure 3.16: Método para gerar função e superfície fractal.

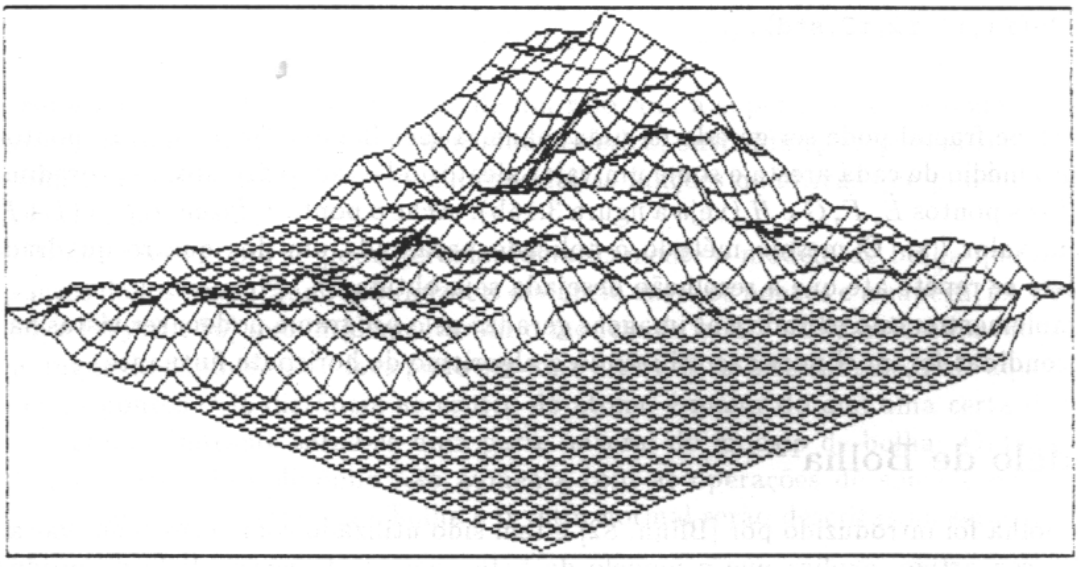


Figure 3.17: Uma montanha fractal.

```
      fm:=(f1+f2)/2+std*gauss(semente,tm);
      std:=std*razao;
      subdivide(f1,t1,fm,tm,std);
      subdivide(fm,tm,f2,t2,std);
    end else output(f1,t1,f2,t2);
  end;

begin
  f1:=gauss(semente,t1)*escala;
  f2:=gauss(semente,t2)*escala;
  razao:=2 ^ (-h); { ^ indica a exponenciacao }
  std:=escala*razao;
  subdivide(f1,t1,f2,t2,std);
end;
```

Uma superfície fractal pode ser gerada de uma maneira semelhante. Dados quatro pontos $\dot{A}$, $\dot{B}$, $\dot{C}$ e $\dot{D}$, gere o ponto médio de cada aresta e some um valor aleatório $l\,s$, como fizemos no procedimento *fractal* acima, gerando os pontos $\dot{E}$, $\dot{F}$, $\dot{G}$ e $\dot{H}$ (veja a figura 3.16b). Ache o ponto $\dot{I}$, fazendo $\dot{I} = (\dot{E}+\dot{F}+\dot{G}+\dot{H})/4$ e somando um valor $l\ s$. O mesmo método é aplicado para cada um dos quatro quadrados menores resultantes. Isto se repete até que a resolução desejada seja obtida.

O autor implementou este método e as imagens geradas pelo programa podem ser vistas na figura 3.17. As arestas escondidas foram eliminadas utilizando o algoritmo de horizonte flutuante.

3.7 Modelo de Bolha[12]

O modelo de bolha foi introduzido por [Blinn, 82] e tem sido utilizado com sucesso em várias aplicações diferentes. No seu artigo, explica que o modelo de bolha nasceu da necessidade de produzir imagens de modelos moleculares que imitassem melhor a função densidade de elétrons do que a representação tradicional através de esferas e cilindros. No modelo de bolha, um átomo se comporta como uma bolha de líquido, que se pode aglutinar com outras bolhas, formando modelos mais complexos. Este método consegue gerar de um modo trivial superfícies implícitas de qualquer formato cuja imagem pode ser gerada através dos algoritmos de lançamento ou rastreamento de raio. Além disso, podemos aplicar facilmente as operações união e diferença de conjuntos entre dois ou mais modelos de bolhas. Isto é muito semelhante às facilidades oferecidas pelo modelo CSG. O modelo de bolha nada mais é do que a soma e/ou subtração de funções implícitas simples para formar funções mais complexas. A superfície de um modelo de bolha consiste de pontos no espaço onde essa função assume um valor T.

[Blinn, 82] trabalhou com funções implícitas que representavam a densidade de elétrons numa molécula. A mecânica quântica representa os elétrons de um átomo como uma função densidade $\mathrm{R}^3 \rightarrow \mathrm{R}$.

[12] Blobby model.

Esta função, para um átomo de hidrogênio, é:

$D(x,y,z) = e^{-ar}$, onde $r = \sqrt{(x-x_1)^2 + (y-y_1)^2 + (z-z_1)^2}$ e (x_1, y_1, z_1) é o centro do átomo.

Esta função será o modelo primitivo do modelo de bolha. Podemos representar uma coleção de átomos somando-se os campos escalares gerados pelos átomos individuais:

$$D(x,y,z) = \sum_i b_i \, e^{-a_i \, r_i}$$

onde r_i é a distância de (x,y,z) ao centro do i-ésimo átomo. Uma superfície pode ser definida como aqueles pontos onde esta função densidade é igual a um valor de disparo T:

$$F(x,y,z) = D(x,y,z) - T = 0$$

Como veremos na seção 10.5, a intersecção de um raio com a superfície acima pode ser determinada por um método numérico. Além da função densidade esférica, outras funções podem ser utilizadas como os modelos primitivos. [Blinn, 82] descreve em especial as funções quádricas que podem gerar cilindros, elipsóides, hiperbolóides, etc.

[Muraki, 91] descreveu como se pode gerar automaticamente o modelo de bolha que se ajusta a um conjunto de pontos dados procurando minimizar a distância entre os pontos e a superfície. Esse artigo cita [Nishimura, 85] e [Wyvill, 86] que implementaram o modelo de bolha utilizando uma função polinomial por trechos no lugar da exponencial. Isto pode ser uma alternativa interessante, pois aquela função vale zero para todos os pontos cuja distância ao centro do átomo é maior do que uma certa distância. Com isso, o teste de volume limitante torna-se uma parte natural do modelo de bolha. Outra característica interessante é que o grau de polinômio não aumenta com as operações de soma e de subtração. As técnicas para calcular a intersecção raio/bolha e o vetor normal serão descritas na seção 10.5.

Chapter 4

Transformações Geométricas

Uma ferramenta muito importante que auxilia a resolução do problema de síntese de imagem são as transformações geométricas. Elas permitem efetuar rotação, translação, reflexão, mudança de escala e transformação em perspectiva. Como veremos, essas transformações nada mais são do que multiplicações de matrizes. A teoria sobre as transformações geométricas encontra-se dispersa por vários livros e artigos. São eles [MathElem], [IntrRay], [ProgPrinc] e [Sutherland, 74]. Procuramos reunir essas informações num todo organizado.

4.1 Coordenadas Homogêneas

Na computação gráfica, costuma-se representar um ponto de $\mathbf{R}^3$ como uma sequência de quatro números em ponto flutuante:

$$\dot{P} = (P_x, P_y, P_z, P_w)$$

Esta maneira de representar os pontos do espaço chama-se representação em coordenadas homogêneas. Como veremos, ela tem uma série de vantagens em relação à representação usual de um ponto como uma tripla ordenada. Generalizando o conceito,

Definição 4.1 *A representação de um vetor como n componentes num vetor com $n+1$ componentes é chamada de representação na* **coordenada homogênea.**

O ponto $\dot{P} = (P_x, P_y, P_z, P_w)$ representa o ponto do espaço com as coordenadas:

$$(\frac{P_x}{P_w}, \frac{P_y}{P_w}, \frac{P_z}{P_w})$$

Se $P_w = 1$, dizemos que a representação está normalizada.

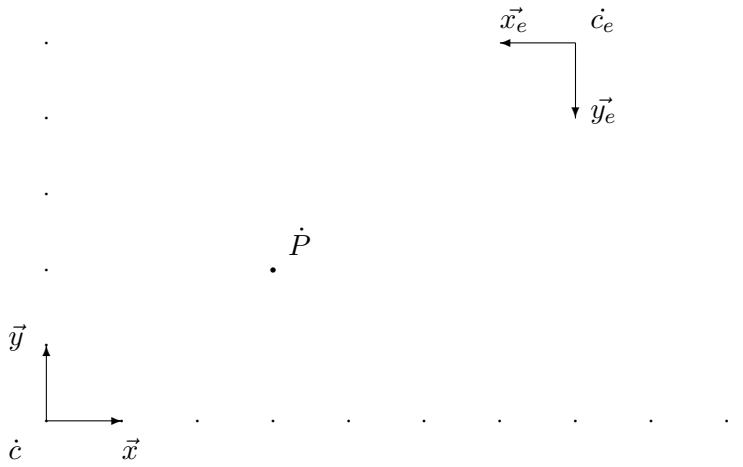


Figure 4.1: Dois sistemas de coordenadas

4.2 Sistema de Coordenadas, vetor-posição e vetor-direção

Para resolver o problema de síntese de imagem, é conveniente descrever os objetos num sistema diferente daquele utilizado pelo usuário. Para isso, primeiro iremos formalizar o conceito de sistema de coordenadas que é bastante semelhante ao da base ortonormal. Uma base ortonormal de $\mathbf{R}^n$ consiste de n vetores unitários ortogonais dois a dois. Para que esta base torne-se um sistema de coordenadas, é necessário acrescentar-lhe um ponto de $\mathbf{R}^n$ que indique o centro do sistema. Assim, podemos definir um sistema de coordenadas como:

Definição 4.2 *Um* **sistema de coordenadas** *do espaço* $\mathbf{R}^n$ *consiste de uma sequência de n vetores de comprimento unitário, ortogonais dois a dois, e de um ponto de* $\mathbf{R}^n$ *que chamaremos de centro do sistema.*

Exemplo 4.1 $E = [\vec{x_e} = (-1, 0);\ \vec{y_e} = (0, -1);\ \dot{c_e} = (7, 5)]$ *define o sistema de coordenadas de* $\mathbf{R}^2$ *da figura 4.1.*

Observação 4.1 *O sistema de coordenadas canônico de* $\mathbf{R}^2$ *é* $C = [\vec{x} = (1, 0),\ \vec{y} = (0, 1),\ \dot{c} = (0, 0)]$. *Do mesmo modo, o sistema de coordenadas canônico de* $\mathbf{R}^3$ *é* $C = [\vec{x} = (1, 0, 0),\ \vec{y} = (0, 1, 0),\ \vec{z} = (0, 0, 1),\ \dot{c} = (0, 0, 0)]$.

Observação 4.2 *Utilizaremos a notação* $\dot{P} = (P_x, P_y, P_z)_E$ *para indicar que as coordenadas do ponto* $\dot{P}$ *estão no sistema* E. *Omitiremos o índice quando pelo contexto ficar claro o sistema utilizado ou quando se utilizar o sistema canônico.*

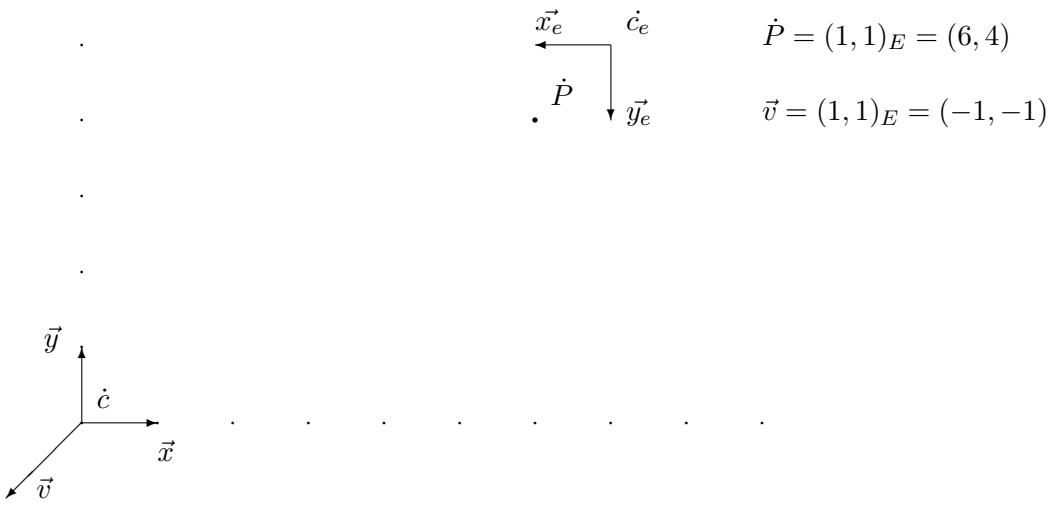


Figure 4.2: Ponto e vetor

Assim, o ponto $\dot{P} = (3, 2)$ no sistema canônico pode ser escrito como $\dot{P} = (4, 3)_E$ no sistema E do exemplo 4.1 (figura 4.1).

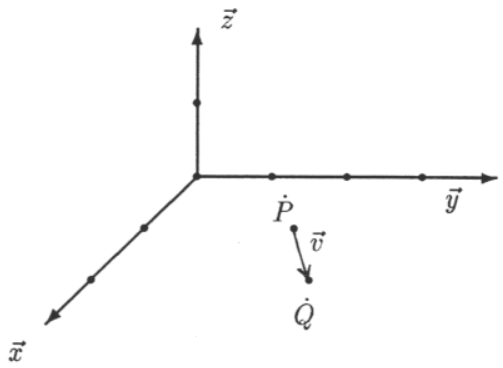
Os leitores devem ter reparado que ao longo deste trabalho temos utilizado notações diferentes para indicar um ponto (sinônimo de vetor-posição) e um vetor (vetor-direção). Para denotar um vetor-posição, temos colocado um ponto acima da letra enquanto que para denotar um vetor-direção, temos utilizado uma flecha, pois quando trabalhamos num sistema de coordenadas, há necessidade de distinguir esses dois conceitos. Suponha dados:

- Um sistema de coordenadas $E = [\vec{e_1}, \vec{e_2}, \ldots, \vec{e_n}, \dot{e_c}]$ de $\mathbf{R}^n$
- Um ponto $\dot{P} = (P_1, P_2, \ldots, P_n)_E$
- Um vetor $\vec{v} = (v_1, v_2, \ldots, v_n)_E$

Com a notação $\dot{P} = (P_1, P_2, \ldots, P_n)_E$ queremos indicar a posição $P_1\vec{e_1} + P_2\vec{e_2} + \ldots + P_n\vec{e_n} + \dot{e}_c$ de $\mathbf{R}^n$. Porém, com a notação $\vec{v} = (v_1, v_2, \ldots, v_n)_E$ queremos indicar o vetor $v_1\vec{e_1} + v_2\vec{e_2} + \ldots + v_n\vec{e_n}$ de $\mathbf{R}^n$.

Exemplo 4.2 *Seja* $E = [\vec{x_e} = (-1, 0);\ \vec{y_e} = (0, -1);\ \dot{c_e} = (7, 5)]$. *A notação* $\dot{P} = (1, 1)_E$ *indica a posição* $1\vec{x_e} + 1\vec{y_e} + \dot{c_e}$ *de* $\mathbf{R}^2$*, ou seja, a posição* $(6, 4)$. *A notação* $\vec{v} = (1, 1)_E$*, por sua vez, indica o vetor* $1\vec{x_e} + 1\vec{y_e}$ *de* $\mathbf{R}^2$*, ou seja, o vetor* $(-1, -1)$ *(veja a figura 4.2).*

Em coordenadas homogêneas normalizadas, escrevemos um vetor-posição $\dot{P} = (P_x, P_y, P_z)$ como $(P_x, P_y, P_z, 1)$ e um vetor-direção $\vec{v} = (v_x, v_y, v_z)$ como $(v_x, v_y, v_z, 0)$. Como se pode observar, uma das vantagens de se trabalhar com as coordenadas homogêneas é que tanto o vetor-posição quanto o vetor-direção podem ser representados utilizando uma única notação.



$$\dot{P} = [1, 2, 0, 1]$$

$$\vec{v} = [1, 1, 0]$$

$$T_{\vec{v}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$\dot{Q} = \dot{P}\, T_{\vec{v}} = [2, 3, 0, 1]$$

Figure 4.3: Translação

4.3 Translação

Definição 4.3 *Vamos chamar de* **translação** *na direção $\vec{v}$ a operação que soma, para cada ponto do objeto, o vetor $\vec{v}$.*

Para fazer uma translação do ponto $\dot{P} = (P_x, P_y, P_z, P_w)$ na direção $\vec{v} = (v_x, v_y, v_z)$, faça a multiplicação matricial $\dot{Q} = \dot{P}\, T_{\vec{v}}$, onde

$$T_{\vec{v}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ v_x & v_y & v_z & 1 \end{bmatrix}$$

E $\dot{Q}$ será o ponto $\dot{P}$ transladado na direção do vetor $\vec{v}$. Veja a figura 4.3.

Observação 4.3 *Não utilizando a representação homogênea, é impossível achar uma matriz $T_{3\times3}$ que translade qualquer ponto de R^3 numa direção $\vec{v}$.*

Observação 4.4 *Se o ponto $\dot{P}$ estava normalizado antes de efetuar a translação, $\dot{Q} = \dot{P}\, T_{\vec{v}}$ também estará normalizado. Como veremos, esta propriedade é válida para todas as transformações geométricas que apresentamos neste capítulo. Uma consequência disto é que, na prática, não é necessário armazenar a coordenada w no computador, uma vez que ela será sempre 1.*

Observação 4.5 *Se queremos aplicar a translação sobre um objeto modelado por B-rep, como já vimos na seção 3.2, basta aplicar a translação sobre todos os vértices desse objeto. Para isso, escreva as coordenada de todos os n vértices desse objeto numa matriz $M_{n\times 4}$ e efetue a multiplicação matricial com a matriz de translação $T_{\vec{v}}$. Veja a figura 4.4. O mesmo vale para as outras transformações geométricas que iremos descrevendo daqui para a frente.*

Observação 4.6 *Se nós multiplicarmos um vetor-direção pela matriz de translação, o vetor permanece inalterado. Isto é coerente com o conceito de vetor-direção, pois não faz sentido transladar um vetor-direção.*

4.4 Rotação

Seja $\dot{P}$ um ponto representado em coordenadas homogêneas. Para rotacioná-lo em torno do eixo de coordenadas $\vec{x}$ de α radianos, multiplique-o pela matriz $R_{\vec{x},\alpha}$:

$$R_{\vec{x},\alpha} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & s & 0 \\ 0 & -s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Onde, } \; c = \cos(\alpha) \;\text{ e }\; s = \sin(\alpha).$$

Para rotacionar em torno dos eixos $\vec{y}$ e $\vec{z}$ de α radianos, utilize as matrizes $R_{\vec{y},\alpha}$ e $R_{\vec{z},\alpha}$, respectivamente.

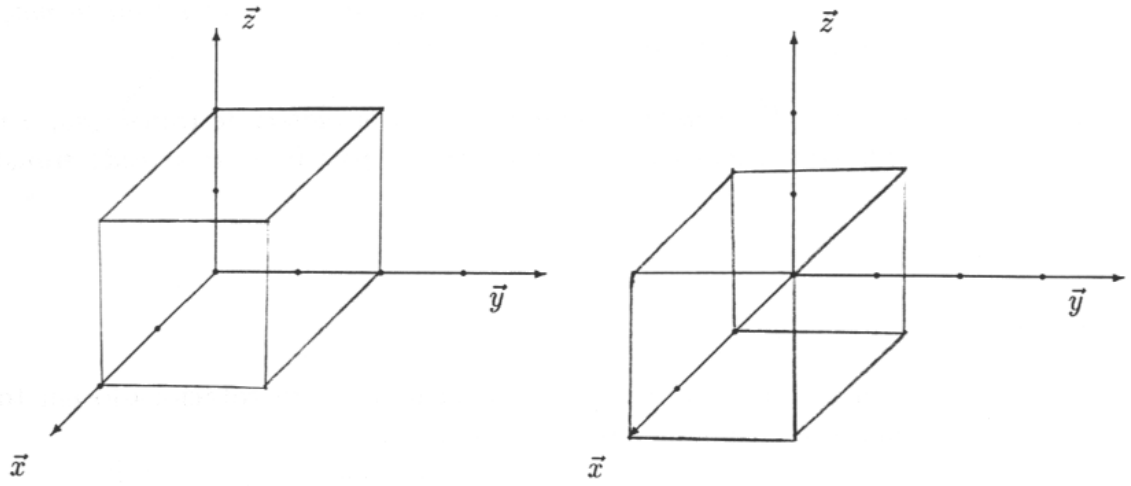
$$R_{\vec{y},\alpha} = \begin{bmatrix} c & 0 & -s & 0 \\ 0 & 1 & 0 & 0 \\ s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_{\vec{z},\alpha} = \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O sentido da rotação obedece à regra da mão direita (veja as figuras 4.5 e 4.6). Também aqui valem as observações 4.4 e 4.5. Tanto o vetor-posição como o vetor-direção podem sofrer uma rotação.

4.5 Mudança de Escala

Os termos do diagonal principal da matriz de transformação produzem a mudança de escala. Se multiplicarmos um ponto $\dot{P} = (P_x, P_y, P_z, P_w)$ pela matriz $E_{a,b,c}$ abaixo, a coordenada P_x será multiplicada por a, P_y por b e P_z por c.

$$E_{a,b,c} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 2 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 0 & 2 & 2 & 1 \\ 0 & 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 1 \\ 3 & 2 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 3 & 0 & 2 & 1 \\ 3 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 0 & 2 & 1 \end{bmatrix}$$

Figure 4.4: Translação de um objeto

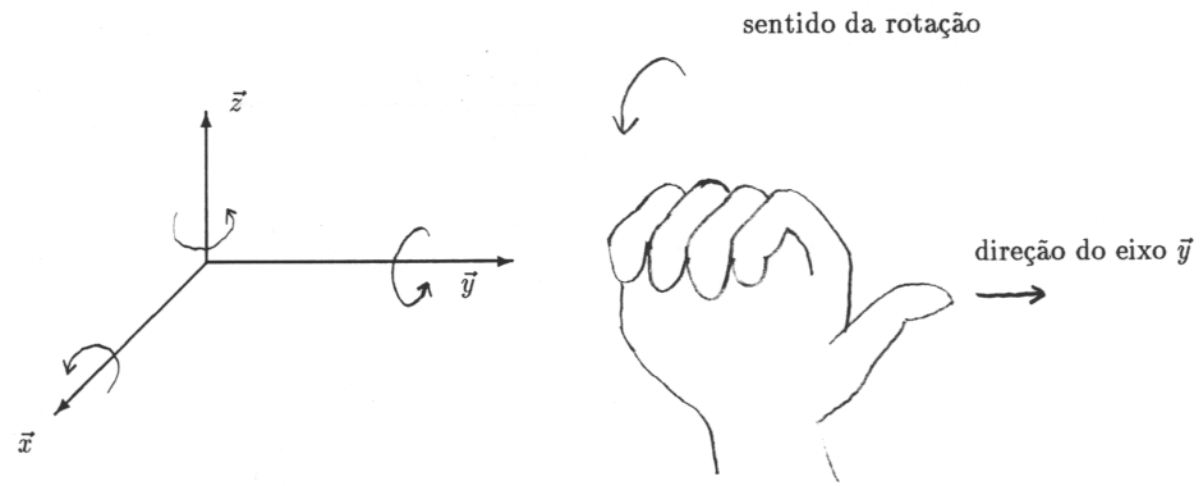
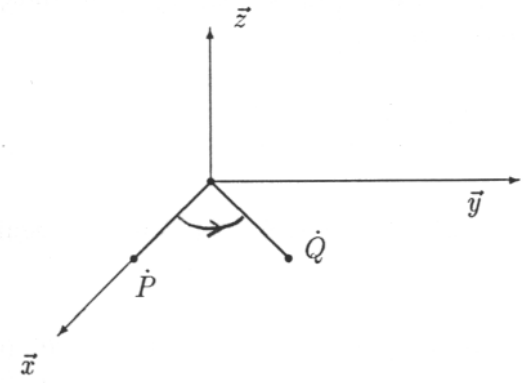


Figure 4.5: Regra da mão direita



$$\dot{P} = [1, 0, 0, 1]$$

$$\alpha = 60^o$$

$$c = \cos(\alpha) = 1/2 \quad s = \sin(\alpha)$$

$$R_{\vec{z},\alpha} = \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\dot{Q} = \dot{P}\, R_{\vec{z},\alpha} = [1/2, \sqrt{3}/2, 0, 1]$$

Figure 4.6: Rotação

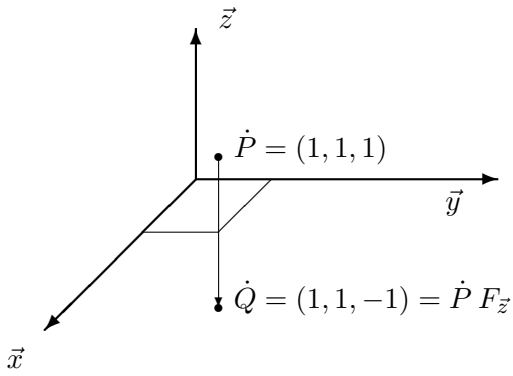


Figure 4.7: Reflexão

Por exemplo, se multiplicarmos todos os pontos de um objeto por $E_{2,2,2}$, esse objeto será ampliado duas vezes. Na mudança de escala também valem as observações 4.4 e 4.5. A mudança de escala é aplicável tanto para o vetor-posição como para o vetor-direção.

4.6 Reflexão

Se queremos refletir um objeto sobre os planos $x = 0, y = 0$ ou $z = 0$, utilizamos as seguintes matrizes de transformação:

$$F_x = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad F_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad F_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Veja a figura 4.7. A reflexão é aplicável tanto ao vetor-direção como ao vetor-posição.

4.7 Composição de Transformações

Será possível armazenar uma sequência de transformações numa única matriz 4×4? A resposta é sim e vejamos por quê. Sejam $T_1, T_2, \ldots, T_n$ matrizes 4×4 que representam as transformações geométricas. Se queremos aplicar as transformações $T_1, T_2, \ldots, T_n$ sobre um ponto $\dot{P}$, na ordem dada, devemos calcular o seguinte produto matricial, da esquerda para a direita:

$$\dot{P}\, T_1\, T_2\, T_3\, \cdots\, T_n.$$

Porém, como a multiplicação matricial possui a propriedade associativa, a expressão acima pode ser reescrita como:

$$\dot{P}\,(T_1\,T_2\,T_3\,\cdots\,T_n)$$

Ora, $T_1\,T_2\,T_3\,\cdots\,T_n$ é uma matriz 4×4 e pode ser armazenada, no computador, numa única matriz de números em ponto flutuante 4×4.

Exemplo 4.3 *Se queremos rotacionar um objeto de* 60^o *em torno da reta que passa pelos pontos* $(1,1,0)$ *e* $(1,1,1)$*, a matriz de transformação é dada pela expressão* $T_{-1,-1,0}\,R_{\vec{z},60^o}\,T_{1,1,0}$.

Observação 4.7 *Seja* T *uma transformação geométrica composta de apenas transformações vistas neste capítulo, ou seja, translação, rotação, mudança de escala e reflexão. Seja* $\dot{P}$ *um ponto representado em coordenadas homogêneas normalizadas. Então* $\dot{P}\,T$ *também estará normalizada. Se aplicamos a transformação* T *a um vetor-direção, o resultado será um outro vetor-direção.*

4.8 Transformações Geométricas em 2D

As transformações geométricas no plano são muito semelhantes àquelas no espaço. Primeiro, nós devemos escrever um ponto em R^2 em coordenadas homogêneas normalizadas, ou seja, acrescentando-lhe o número 1 na terceira coordenada.

Exemplo 4.4 *Escrevendo o ponto* $(2,3)$ *no sistema de coordenadas homogêneas normalizadas, obtemos* $(2,3,1)$.

Translação

A matriz de translação na direção $\vec{v}$ é:

$$T_{\vec{v}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ v_x & v_y & 1 \end{bmatrix}$$

Rotação

A matriz de rotação no sentido anti-horário de α radianos é:

$$R_\alpha = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{Onde, } \; c=\cos(\alpha) \;\text{ e }\; s=\sin(\alpha).$$

Mudança de escala

A matriz de mudança de escala é:

$$E_{a,b} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

reflexão

A matriz de reflexão sobre a reta x=0 (eixo de coordenadas $\vec{y}$) é:

$$R_x = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A matriz de reflexão sobre a reta y=0 (eixo de coordenadas $\vec{x}$) é:

$$R_y = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Observação 4.8 *Assim como foi visto na seção 4.7, no plano também é possível compor as transformações e armazenar o resultado numa única matriz* 3×3.

Observação 4.9 *A observação 4.4 também vale no plano, ou seja, as transformações acima levam um ponto normalizado a um outro ponto normalizado.*

4.9 Transformação Geométrica da Equação do Plano

Como se sabe, um plano L pode ser representado pela equação:

$$ax + by + cz + d = 0$$

Onde (a, b, c) é um vetor normal ao plano L unitário e d é a distância do plano L à origem do sistema de coordenadas $(0, 0, 0)$. Logo, um plano L pode ser representado como uma quádrupla, assim como um vetor-posição ou um vetor-direção. O que devemos fazer se queremos aplicar uma transformação geométrica T na equação do plano? Primeiro, calcule $(T^{-1})^t$, a transposta da matriz inversa de T. Depois calcule:

$$L' = L\ (T^{-1})^t$$

Isto fará com que obtenhamos o plano L' que é o plano L após sofrer a transformação geométrica T. [IntrRay] afirma que a maneira correta de se aplicar uma transformação geométrica num vetor-normal a uma superfície é tratá-lo como um plano tangente. Não é correto tratá-lo como um vetor-direção.

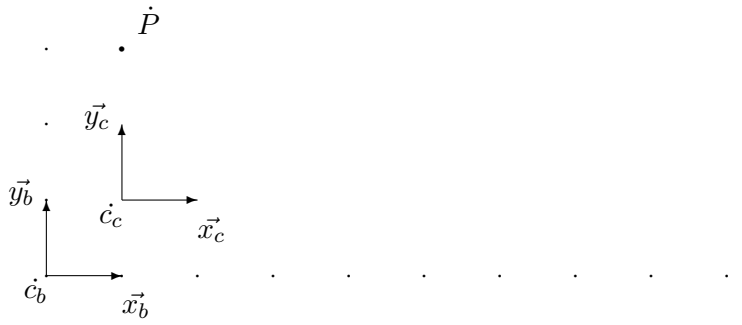


Figure 4.8: Translação de sistema de coordenadas

4.10 Transformações Geométricas de Sistemas de Coordenadas

Translação de sistema de coordenadas

Transladar um sistema de coordenadas na direção do vetor $\vec{v}$ equivale a transladar todos os pontos do ambiente na direção do vetor $-\vec{v}$. Isto vale tanto em 2D como em 3D.

Exemplo 4.5 *Veja a figura 4.8. No sistema B, o ponto $\dot{P}$ tinha coordenadas $(1,3)_B$. Vamos transladar o sistema B na direção do vetor $\vec{v} = (1,1)$, obtendo o sistema C. No sistema C, o ponto $\dot{P}$ tem coordenadas $(0,2)_C$. Ora, $(1,3) - \vec{v} = (0,2)$, confirmando o que foi afirmado acima.*

Rotação de sistema de coordenadas

Fazer uma rotação de sistema de coordenadas de α radianos equivale a rotacionar todos os pontos do ambiente de $-\alpha$ radianos. Isto vale tanto em 2D como em 3D.

Reflexão de sistema de coordenadas

Se queremos efetuar uma reflexão de sistema de coordenadas, efetue essa mesma reflexão em todos os pontos do ambiente.

Mudança de escala de sistema de coordenadas

Multiplicar o sistema por $E_{a,b,c}$ equivale a multiplicar todos os pontos do ambiente por $E_{\frac{1}{a},\frac{1}{b},\frac{1}{c}}$.

Resumo

Resumindo, aplicar uma transformação geométrica T no sistema de coordenadas equivale a aplicar a transformação geométrica T^{-1} nos objetos.

4.11 Sistemas de Coordenadas O, E, U e S

Existem quatro sistemas de coordenadas especiais que auxiliam a resolução do problema de síntese de imagem. São eles:

- Do objeto (O): é um sistema de coordenadas de $\mathbf{R}^3$.
- Do observador (E): é um sistema de coordenadas de $\mathbf{R}^3$.
- Da tela em ponto flutuante (U): é um sistema de coordenadas de $\mathbf{R}^2$ ou de $\mathbf{R}^3$ e daremos respectivamente os nomes U^2 ou U^3.
- Da tela inteira (S): é um sistema de coordenadas de $\mathbf{N}^2$.

Um algoritmo de síntese de imagem tem como entrada os objetos descritos no sistema O. Este sistema de coordenadas é tridimensional e as coordenadas são reais, ou seja, é um sistema de coordenadas de $\mathbf{R}^3$. A imagem gerada por um algoritmo de síntese de imagem, por sua vez, é uma matriz de números inteiros, é um sistema de coordenadas bidimensional onde as coordenadas são números inteiros, pois os índices de uma matriz são sempre inteiros. Nesta seção, é exposta a sequência de transformações necessárias para converter um objeto do sistema do objeto para o sistema da tela. Iremos converter primeiro do sistema O para o sistema E, depois de E para U e, por fim, de U para S.

Definição 4.4 *O sistema de coordenadas onde o ambiente foi descrito é chamado* **sistema de coordenadas do objeto** ($O = [\vec{x_o}, \vec{y_o}, \vec{z_o}, \dot{c_o}]$).

Definição 4.5 *O sistema de coordenadas onde a imagem gerada é descrita é chamado* **sistema de coordenadas da tela** ($S = [\vec{x_s}, \vec{y_s}, \dot{c_s}]$).

4.11.1 Do sistema O para E (Transformação de Visão)

Quando os objetos estão descritos no sistema do objeto, é necessário informar a posição do observador ($\dot{E}$) e a direção de visão ($\vec{F}$) ao algoritmo de síntese de imagem para que se possa calcular a imagem visível ao observador. Como veremos, as informações $\dot{E}$ e $\vec{F}$ não serão mais necessárias no sistema E.

Definição 4.6 *O* **sistema de coordenadas do observador** ($E = [\vec{x_e}, \vec{y_e}, \vec{z_e}, \dot{c_e}]$) *é o sistema onde o centro $\dot{c_e}$ coincide com o olho do observador ($\dot{E}$) e a direção de visão $\vec{F}$ coincide com o eixo $\vec{z_e}$. Além disso, o eixo $\vec{y_e}$ fica "para cima" do observador e o eixo $\vec{x_e}$ fica "à direita" do observador (veja a figura 4.9).*

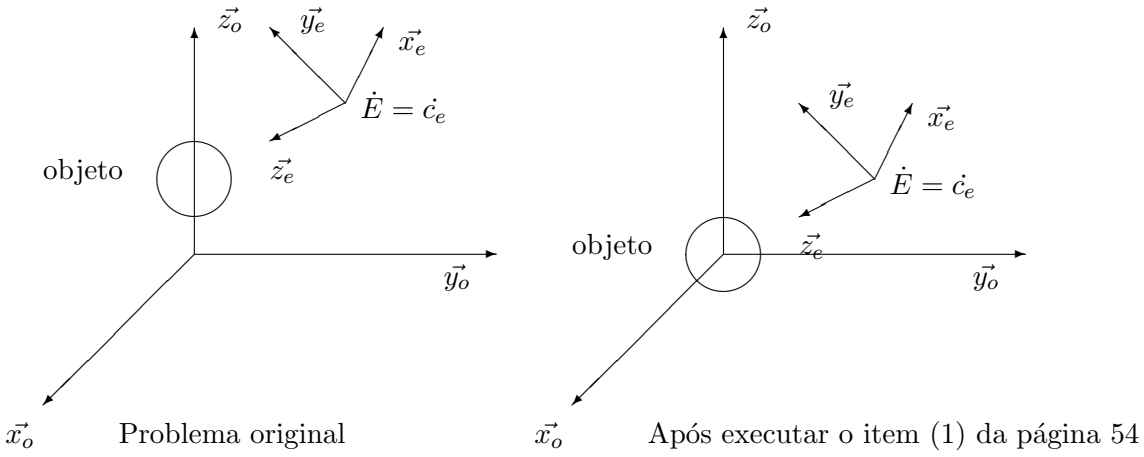


Figure 4.9: Sistema de coordenadas do objeto e do observador

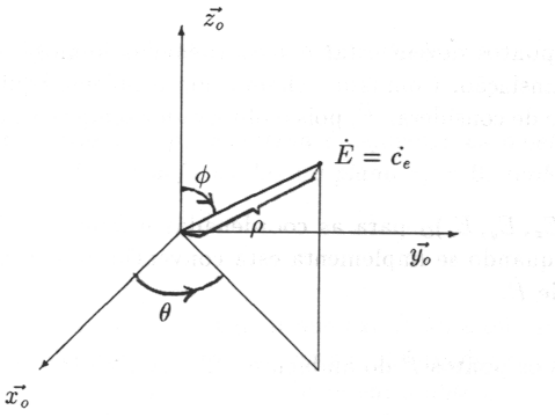


Figure 4.10: Coordenada esféricas

Observação 4.10 *O sistema E tem orientação contrária ao sistema O, isto é, no sistema E não vale a regra da mão direita mas a regra da mão esquerda para efetuar as rotações e o produto vetorial.*

Para converter um ponto $\dot{P}$ representado no sistema O para o sistema E, aplique as seguintes operações:

(1) O problema torna-se mais fácil de se resolver se o observador estiver olhando para o ponto $(0,0,0)_O$. Para isso, pegue um ponto qualquer para onde observador está olhando, por exemplo, $\dot{A} = \dot{E} + \vec{F}$. Agora, vamos transladar o ponto $\dot{E}$ e todos os pontos do ambiente na direção $\vec{v} = \dot{c_o} - \dot{A}$. Ou seja, faça:

(1a) $\dot{E} := \dot{E}\, T_{\vec{v}}$;
(1b) Para todo ponto $\dot{P} \in$ ambiente faça $\dot{P} := \dot{P}\, T_{\vec{v}}$;

Evidentemente, os pontos devem estar em coordenadas homogêneas (de preferência normalizadas) para poder aplicar a translação. Com isso, achamos um problema equivalente ao problema original onde não há mais necessidade de considerar $\vec{F}$, pois o observador sempre estará olhando para o ponto $(0,0,0)_O$. Veja a figura 4.9.

(2) Converta $\dot{E} = (E_x, E_y, E_z)_O$ para as coordenadas esféricas $(\rho, \theta, \phi)_O$ (figura 4.10). É necessário tomar certos cuidados quando se implementa esta conversão, pois é fácil resultar uma divisão por zero para valores especiais de $\dot{E}$.

(3) Multiplique todos os pontos $\dot{P}$ do ambiente pela matriz M:

$$M = \begin{bmatrix} -s_t & -c_p c_t & -s_p c_t & 0 \\ c_t & -c_p s_t & -s_p s_t & 0 \\ 0 & s_p & -c_p & 0 \\ 0 & 0 & \rho & 1 \end{bmatrix} \qquad \text{Onde, } s_t = \sin(\theta),\ c_t = \cos(\theta),\ s_p = \sin(\phi) \text{ e } c_p = \cos(\phi)$$

Depois desta multiplicação, todos os pontos do ambiente estarão no sistema E.

Observação 4.11 *A matriz M é resultado da composição das seguintes transformações geométricas elementares: $M = R_{\vec{z},\frac{\pi}{2}-\theta}\, R_{\vec{x},\phi-\pi}\, F_x\, T_{\dot{c_o}-\dot{E}}$. Fica mais fácil de imaginarmos estas transformações se pensamos que quem está sofrendo as transformações é o sistema de coordenadas (em vez de imaginar que são os objetos que sofrem as transformações). Assim, primeiro rotacionamos o sistema de coordenadas em torno do eixo $\vec{z}$ de $-(\frac{\pi}{2} - \theta)$ radianos. Depois rotacionamos o sistema de coordenadas em torno do eixo $\vec{x}$ de $-(\phi - \pi)$ radianos e assim por diante.*

Observação 4.12 *Podemos economizar o número de multiplicações matriciais efetuando primeiro os itens (1a) e (2). Depois, multiplique todos os pontos $\dot{P}$ do ambiente pela matriz $T_{\vec{v}}\, M$.*

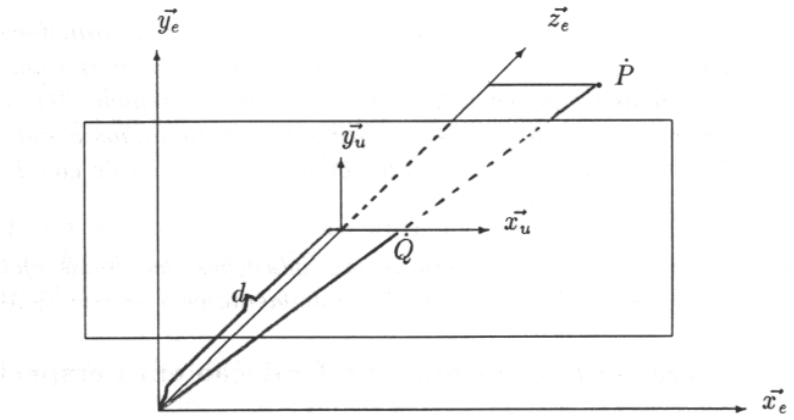


Figure 4.11: Projeção em perspectiva

4.11.2 Do sistema E para S (Transformação e Projeção em Perspectiva)

Após executar as operações indicadas na subseção 4.11.1, os objetos estão no sistema E.

Definição 4.7 *Se simplesmente ignorarmos as coordenadas z_e de todos os pontos (ficando, portanto, só com as coordenadas x_e e y_e) teremos a* **projeção ortogonal** *dos objetos no plano $z_e = 0$.*

Dois objetos de mesmo tamanho terão tamanhos iguais na projeção ortogonal, quer o objeto esteja próximo do observador quer o objeto esteja longe do observador. Isto não ocorre na realidade, pois sabemos que, dados dois objetos A e B de mesmo tamanho, sendo que o objeto A está mais próximo ao observador que o objeto B, para o observador parecerá que o objeto A é maior do que o objeto B. Para simular este efeito, devemos fazer ou uma transformação ou uma projeção em perspectiva. Os dois termos não são sinônimos, pois:

Definição 4.8 *Fazer uma* **projeção em perspectiva** *é projetar os objetos do ambiente no plano $z_e = 0$, levando em consideração o efeito perspectiva.*

Definição 4.9 *Fazer uma* **transformação em perspectiva** *é deformar os objetos de tal forma que, se fizermos uma projeção ortogonal dos objetos deformados no plano $z_e = 0$, tenhamos a projeção dos objetos em perspectiva.*

Projeção em Perspectiva

Para efetuar a projeção em perspectiva, vamos imaginar que existe uma tela imaginária à frente do observador (veja a figura 4.11). Vamos chamar a distância entre o observador e a tela imaginária de d.

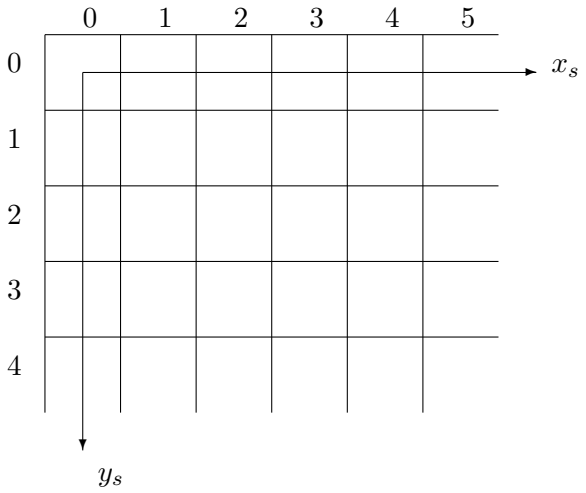


Figure 4.12: Sistema S

Na figura 4.11, projetando o ponto $\dot{P} = (P_x, P_y, P_z)_e$ na tela imaginária obtemos o ponto $\dot{Q} = (Q_x, Q_y)_u$ com as coordenadas $Q_x = dP_x/P_z$ e $Q_y = dP_y/P_z$. Com isso, obtivemos a transformação do sistema E para o sistema U^2.

Definição 4.10 *O sistema de coordenadas onde se descreve os objetos após efetuar a projeção em perspectiva (ou a transformação em perspectiva) chama-se* **sistema de coordenadas da tela em ponto flutuante** *(U).*

Existem, na verdade, dois sistemas de coordenadas da tela em ponto flutuante diferentes entre si. Um é bidimensional e resulta da aplicação de projeção em perspectiva (U^2). Outro é tridimensional e resulta da aplicação de transformação em perspectiva (U^3).

Transformação em Perspectiva

Também é possível efetuar uma transformação em perspectiva. As características essenciais desta transformação são:

1. As coordenadas Q_x e Q_y são obtidas como na projeção em perspectiva, ou seja, $Q_x = dP_x/P_z$ e $Q_y = dP_y/P_z$.
2. Devemos calcular a "profundidade em perspectiva" para cada ponto de modo que uma reta continue sendo uma reta, um plano continue sendo um plano após aplicar a transformação em perspectiva.

Também devemos preservar a ordenação em profundidade, isto é, a ordem das coordenadas z no sistema E deve ser mantida no sistema U. Esta ordem será usada pelos algoritmos de síntese de imagem para decidir quais são os objetos escondidos. Tudo isso pode ser satisfeito fazendo $Q_z = -d \ / \ P_z$.

A matriz que efetua a transformação em perspectiva é:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Esta matriz não leva um ponto normalizado em outro normalizado, de modo que é necessário aplicar explicitamente a operação de normalização após efetuar a multiplicação.

4.11.3 Do sistema U para S

Após efetuar a projeção ou a transformação em perspectiva, os pontos estão no sistema U, mas ainda não estão prontos para serem colocados na tela do computador, por vários motivos. Primeiro porque na tela do computador não existem pontos com coordenadas fracionárias. Segundo porque um pixel da tela pode ser retangular, em vez de ser quadrado. Terceiro porque normalmente o ponto $(0, 0)$ da tela está no canto superior esquerdo, em vez de estar localizado no centro da tela. Quarto porque a tela do computador pode ter mais ou menos pixels, tanto no sentido horizontal como no sentido vertical. Portanto, é necessário corrigir estas distorções para chegar ao sistema de coordenadas da tela inteira.

Vamos chamar de $MaxU.x$ a maior coordenada x no sistema U de toda a imagem a ser mostrada. De modo análogo, definimos $MaxU.y$, $MinU.x$ e $MinU.y$. Chamemos de $Asp.x$ e $Asp.y$ a largura e a altura de um pixel, respectivamente. $Asp.x$ e $Asp.y$ podem ser medidos em qualquer unidade de comprimento, pois na realidade, só nos interesssa a razão entre elas: $Asp.x/Asp.y$. Sejam $MaxD.x$ e $MaxD.y$ respectivamente largura da tela menos um e altura da tela menos um. Então, o seguinte trecho de programa em Pascal converte um ponto $\dot{P}_u$ para o sistema de coordenadas inteiras da tela $\dot{P}_s$. $Ps.x$ e $Ps.y$ devem pertencer ao intervalo $[0 \ldots MaxD.x]$ e $[0 \ldots MaxD.y]$, respectivamente.

Normalmente queremos que a imagem ocupe toda a tela (se queremos que a imagem ocupe somente uma parte da tela basta redimensionarmos $MaxD$). Mas na maior parte das vezes não será possível fazer com que a imagem ocupe toda a tela porque o formato da imagem será diferente do formato da tela. Por exemplo, se a imagem é quadrada e a tela retangular, será impossível fazer com que a imagem ocupe toda a tela sem distorcê-lo. Neste caso, faremos com que a imagem seja ampliada o máximo até que, ampliando mais, partes da imagem fiquem fora da tela. Se a parte superior (ou inferior) da imagem começa a sair fora da tela antes das partes laterais então indicaremos este fato atribuindo $TRUE$ à variável $GrudaY$ e a imagem final terá bordas superior e inferior grudadas na tela. Caso contrário, atribuiremos $FALSE$ ao $GrudaY$ e as bordas laterais da imagem estarão grudadas na tela. O seguinte trecho de programa Pascal calcula $GrudaY$:

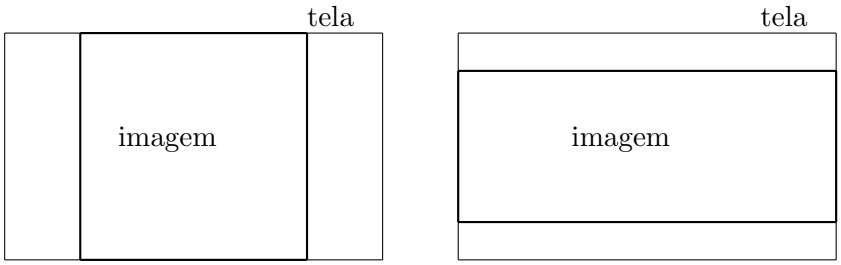


Figure 4.13: Variável GrudaY

```
if (MaxU.y-MinU.y)/(MaxU.x-MinU.x) > (Asp.y*MaxD.y)/(Asp.x*MaxD.x)
 then GrudaY:=true else GrudaY:=false;
```

Agora, vamos calcular o fator de multiplicação fat:

```
if GrudaY then fat:=MaxD.y/(Asp.x*(MaxU.y-MinU.y))
          else fat:=MaxD.x/(Asp.y*(MaxU.x-MinU.x));
```

E o cálculo de *Desloc* que indica a translação necessária:

```
if GrudaY then begin Desloc.y:=0;
 Desloc.x:=(MaxD.x-(fat*Asp.y*(MaxU.x-MinU.x)))/2;
end else begin Desloc.x:=0;
 Desloc.y:=(MaxD.y-(fat*Asp.x*(MaxU.y-MinU.y)))/2;
end;
```

Tudo o que fizemos acima pode ser calculado uma única vez no começo do programa. Temos agora todas as informações necessárias para converter cada ponto $\dot{P}_u$, para o sistema de coordenadas S. Basta fazermos:

```
Ps.x:=Desloc.x+fat*Asp.y*(Pu.x-MinU.x);
Ps.y:=MaxD.y-(Desloc.y+fat*Asp.x*(Pu.y-MinU.y));
```

Chapter 5

Algoritmos Básicos

5.1 Algoritmos para desenhar reta

Provavelmente, o algoritmo mais básico da Computação Gráfica é aquele que traça um segmento de reta entre dois pontos dados. Praticamente todos os pacotes gráficos e compiladores incluem uma rotina para esta finalidade na sua biblioteca. Assim sendo, poderíamos pensar que não há necessidade de conhecer estes algoritmos, pois bastaria usar aqueles fornecidos pelos compiladores. Isto é falso, pois existem muitos algoritmos cujo funcionamento está baseado naqueles algoritmos. Por exemplo, é necessário entender os algoritmos para desenhar reta para poder compreender os algoritmos de preenchimento de polígono. E a compreensão destes é, por sua vez, necessária na implementação de muitos algoritmos de eliminação de superfícies escondidas.

Definição 5.1 *O problema de traçar reta consiste em, dados dois pontos com as coordenadas inteiras (sistema de coordenadas da tela), determinar os pontos da tela que devem ser acesas para que pareça, visualmente, ter um segmento de reta entre os dois pontos.*

Observe que "parecer visivelmente ter um segmento de reta" é algo muito vago. Veja a figura 5.1. Lá vemos dois desenhos diferentes para a mesma reta entre os pontos $\dot{P}_1 = (0, 0)$ e $\dot{P}_2 = (5, 3)$. A maioria dos pacotes gráficos traçam as retas do tipo A. Turbo Graphix Toolbox (versão 4.0), porém, traça retas do tipo B. Um algoritmo muito semelhante ao tipo A é utilizado no algoritmo de z-buffer e um algoritmo que traça retas tipo B é utilizado para acelerar a velocidade dos rastreadores de raio. Veremos primeiro uma série de algoritmos tipo A e depois descreveremos um algoritmo tipo B.

5.1.1 DDA[1]

O algoritmo DDA, que traça retas tipo A, pode ser escrita como:

[1] Digital Differential Analyzer.

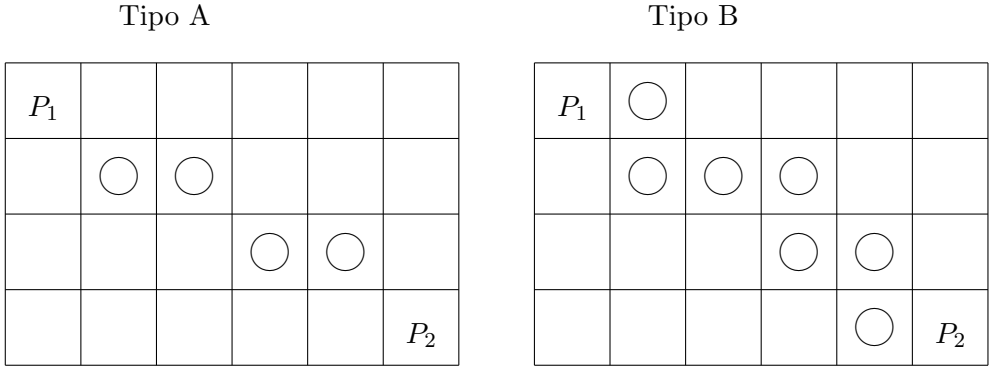


Figure 5.1: Traçamento de reta

```
Procedure retaA1(x1,y1,x2,y2,c:integer);
var len,i:integer; dx,dy,x,y:real;
begin
  len:=maximo( abs(x2-x1), abs(y2-y1) );
  dx:=(x2-x1)/len; dy:=(y2-y1)/len; x:=x1; y:=y1;
  for i:=0 to len do begin {*}
    ponto( round(x), round(y), c); x:=x+dx; y:=y+dy;
  end;
end;
```

Observe que este algoritmo é extremamente simples. Mesmo assim, ele consegue traçar segmentos de reta em qualquer direção. Nesse algoritmo, *len* indica o número de pontos da tela que devem ser acesas, dx indica quanto se deve andar na direção x a cada iteração e dy indica quanto se deve andar na direção y. Repare que dx e dy são números reais no intervalo $[-1 \ldots 1]$ e sempre pelo menos uma das duas tem o valor um. A cada iteração, somamos dx na variável x e dy na variável y e o ponto mais próximo ao (x, y) é aceso. O defeito deste algoritmo é recorrer ao uso de variáveis reais. As operações em ponto flutuante são normalmente muito mais lentas que as operações inteiras. Nas próximas subseções, veremos como eliminar as operações em ponto flutuante.

5.1.2 Eliminando o arredondamento de DDA

O primeiro passo para eliminar as operações em ponto flutuante é eliminar a operação de arredondamento. E isto pode ser conseguido introduzindo o conceito de erro:

```
Procedure retaA2(x1,y1,x2,y2,c:integer);
var i,x,y,len:integer; dx,dy,errx,erry:real;
begin
  len:=maximo( abs(x2-x1), abs(y2-y1) );
  dx:=(x2-x1)/len; dy:=(y2-y1)/len; x:=x1; y:=y1;
  for i:=0 to len do begin {*}
    ponto(x, y, c); errx:=errx+dx; erry:=erry+dy;
    if abs(errx)>=0.5 then begin
      x:=x+sinal(dx); errx:=errx-sinal(dx);
    end;
    if abs(erry)>=0.5 then begin
      y:=y+sinal(dy); erry:=erry-sinal(dy);
    end;
  end;
end;
```

A função $sinal(x)$ devolve -1 se x é negativo, 0 se x é nulo e $+1$ se x é positivo. No ponto marcado com o asterisco, $Errx$ equivale ao $x - round(x)$ do procedimento $reta1$. Do mesmo modo, $Erry$ e $y - round(y)$ são equivalentes nos dois procedimentos. Em outras palavras, $errx$ ($erry$) do procedimento $reta2$ indica o erro que se comete arredondando x (y) para $round(x)$ ($round(y)$) no procedimento $reta1$. Portanto, a variável $errx$ ($erry$) precisa ser ajustada toda vez que ela ultrapassar 0.5 em valor absoluto, pois o maior erro que se pode cometer no arredondamento é ± 0.5.

5.1.3 Eliminando as variáveis reais de DDA

As variáveis reais podem ser eliminadas multiplicando $errx$, $erry$ e todas as demais constantes e variáveis associadas com o erro por $2\,len$. Com isso, o algoritmo DDA torna-se:

```
procedure retaA3(x1,y1,x2,y2,c:integer);
var i,x,y,dx,dy,len,errx,erry:integer;
begin
  dx:=x2-x1; dy:=y2-y1; len:=maximo(abs(dx),abs(dy));
  errx:=0; erry:=0; x:=x1; y:=y1;
  for i:=0 to len do begin
    ponto(x,y,c); errx:=errx+2*dx; erry:=erry+2*dy;
    if abs(errx)>=len then begin
      x:=x+sinal(dx); errx:=errx-sinal(dx)*2*len;
    end;
    if abs(erry)>=len then begin
```

```
      y:=y+sinal(dy); erry:=erry-sinal(dy)*2*len;
    end;
  end;
end;
```

Observe que, para acelerar o processamento, as expressões $2 \times dx$, $2 \times dy$, $sinal(dx)$, $sinal(dy)$, $sinal(dx) \times 2 \times len$ e $sinal(dy) \times 2 \times len$ podem ser avaliadas uma única vez fora do laço e armazenadas em variáveis. Com isto, dentro do laço efetuamos somente 6 somas, 2 comparações e 2 módulos em cada iteração. Na realidade, é possível escrever este procedimento de modo que em cada iteração efetue somente 3 somas e 1 comparação. Como veremos na próxima subseção, isto é possível subdividindo o problema em 2 casos.

5.1.4 Algoritmo de Bresenham

Melhorando ainda mais a velocidade de processamento, obtemos:

```
procedure retaA4(x1,y1,x2,y2,c:integer);
var x,y,dx,dy,a,err,dx2,dy2,sobe:integer;
begin
  dx:=x2-x1; dy:=y2-y1;
  if abs(dx)>=abs(dy) then begin {octante 1, 4, 5 ou 8}
    if dx<0 then begin troca(x1,x2); troca(y1,y2); dx:=-dx; dy:=-dy; end;
    a:=abs(2*dy); err:=0; dx2:=dx*2;
    sobe:=sinal(dy); y:=y1;
    for x:=x1 to x2 do begin
      ponto(x,y,c); err:=err+a;
      if err>=dx then begin y:=y+sobe; err:=err-dx2; end;
    end;
  end else begin {octante 2, 3, 6 ou 7}
    if dy<0 then begin troca(x1,x2); troca(y1,y2); dx:=-dx; dy:=-dy; end;
    a:=abs(2*dx); err:=0; dy2:=dy*2;
    sobe:=sinal(dx); x:=x1;
    for y:=y1 to y2 do begin
      ponto(x,y,c); err:=err+a;
      if err>=dy then begin x:=x+sobe; err:=err-dy2; end;
    end;
  end;
end;
```

Este procedimento efetua somente 3 somas e 1 comparação em cada iteração.

5.1.5 Um algoritmo que traça retas tipo B

O seguinte algoritmo traça retas do tipo B, utilizando somente variáveis inteiras:

```
procedure retaB(x1,y1,x2,y2,c:integer);
var x,y,dx,dy,passox,passoy,direcao:integer;
begin
  x:=x1; y:=y1; passox:=1; passoy:=1;
  if x1>x2 then passox:=-1; if y1>y2 then passoy:=-1;
  dx:=abs(x2-x1); dy:=abs(y2-y1);
  if dx=0 then direcao:=-1 else direcao:=0;
  while not ((x=x2) and (y=y2)) do begin
    ponto(x,y,c);
    if direcao<0 then begin y:=y+passoy; direcao:=direcao+dx; end
                 else begin x:=x+passox; direcao:=direcao-dy; end;
  end;
end;
```

Existem algoritmos semelhantes aos que acabamos de ver para traçar círculos, elipses e parábolas. Eles também não utilizam variáveis em ponto flutuante e por isso são extremamente eficientes. Estes algoritmos podem ser encontrados em [ProcElem].

5.2 Teste do sentido horário e aplicações

Na computação gráfica muitas vezes queremos saber, dados três pontos $\dot{A}$, $\dot{B}$ e $\dot{C}$ no plano, se estes três pontos, na ordem dada, estão em sentido horário, anti-horário ou alinhados. Este teste pode ser feita calculando o sinal do seguinte determinante:

$$\begin{vmatrix} 1 & A_x & A_y \\ 1 & B_x & B_y \\ 1 & C_x & C_y \end{vmatrix}$$

Para facilitar a notação, escreveremos $H(\dot{A},\dot{B},\dot{C})$ para indicar o determinante acima. $\dot{A}$, $\dot{B}$ e $\dot{C}$ estão em sentido anti-horário se, e somente se, $H(\dot{A},\dot{B},\dot{C}) > 0$. Se $H(\dot{A},\dot{B},\dot{C}) < 0$ então os três pontos estão em sentido horário. Se a função resultar nulo, os três pontos estão alinhados. Além do sinal da função H indicar o sentido dos três pontos, o valor absoluto da função H é sempre duas vezes maior que a área do triângulo ABC. Estes fatos tornam a função H muito útil em Computação Gráfica. Aplicando a regra de Chió, conseguimos simplificar o determinante acima para:

$$H(\dot{A},\dot{B},\dot{C}) = \begin{vmatrix} B_x - A_x & B_y - A_y \\ C_x - A_x & C_y - A_y \end{vmatrix}$$

Esta expressão necessita de apenas duas multiplicações e cinco subtrações para ser calculada.

5.2.1 Verificar se dois segmentos de reta se cruzam

A primeira aplicação da função H é determinar se dois segmentos de reta se cruzam. Sejam dados dois segmentos de reta AB e CD. Estes dois segmentos se cruzam quando:

$$H(A,B,C)\,H(A,B,D) \leq 0 \quad \text{e} \quad H(C,D,A)\,H(C,D,B) \leq 0$$

Este teste falha se os quatro pontos $\dot{A}, \dot{B}, \dot{C}$ e $\dot{D}$ estão alinhados.

5.2.2 Verificar se um polígono é côncavo ou convexo

Uma outra aplicação da função H é dado um polígono G:

$$G = (\dot{G}_{-1} = \dot{G}_{n-1}, \dot{G}_0, \dot{G}_1, \ldots, \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$$

determinar se G é côncavo ou convexo. Isto pode ser feito verificando se os sinais de

$$H(\dot{G}_{i-1}, \dot{G}_i, \dot{G}_{i+1}) \ , \quad 0 \leq i \leq n-1$$

são todos iguais ou não. Se todas elas tiverem o mesmo sinal então o polígono G é convexo, senão é côncavo.

5.3 Pseudo-ângulo

Um vetor-direção $\vec{v}$ pode ser representado de um modo alternativo fornecendo, em vez de v_x e v_y, o ângulo formado pelo vetor $\vec{v}$ com o eixo $\vec{x}$. Se o vetor $\vec{v}$ pertence ao primeiro quadrante, a fórmula: ângulo$(\vec{v}) = arctg(v_y/v_x)$ calcula esse ângulo. Se o vetor pertence a um outro quadrante, o ângulo pode ser calculado de um modo semelhante, sempre utilizando a função $arctg$. Porém, calcular a função $arctg$ no computador é uma tarefa demorada. Por isso, em Computação Gráfica costuma-se utilizar o pseudo-ângulo o que evita o cálculo da função $arctg$. O pseudo-ângulo apresenta a mesma relação de ordem que o ângulo verdadeiro, isto é:

$$\text{ângulo}(\vec{v}) < \text{ângulo}(\vec{w}) \quad \text{se, e somente se,} \quad \text{pseudo-ângulo}(\vec{v}) < \text{pseudo-ângulo}(\vec{w})$$

Existem muitas maneiras de construir o pseudo-ângulo e o livro [Algorithms] apresenta um deles. Neste trabalho, citaremos um outro método que tem uma relação estreita com o conceito de cubo direcional que utilizaremos na seção 11.3. Por analogia, chamaremos o método de quadrado direcional. O quadrado direcional tem os lados de comprimento dois e está localizado no centro de sistema de coordenadas (veja a figura 5.2a).

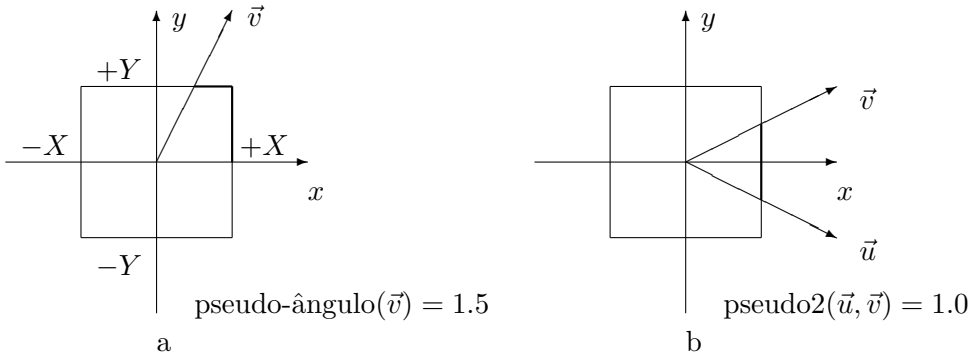


Figure 5.2: Quadrado direcional e o pseudo-ângulo.

O pseudo-ângulo de um vetor-direção $\vec{v}$ é a distância percorrida em sentido anti-horário em cima das arestas do quadrado direcional, a partir do ponto $(1, 0)$, até chegar ao ponto de intersecção do quadrado direcional com o vetor $\vec{v}$. Logo, o pseudo-ângulo pode variar de 0 a 8. Para calcular o pseudo-ângulo, primeiro determine o eixo dominante do vetor $\vec{v}$. O eixo dominante é $+X$, $-X$, $+Y$ ou $-Y$ e indica qual das quatro arestas do cubo direcional o vetor irá intersectar. Ele pode ser calculado comparando os valores absolutos $\mid v_x \mid$ e $\mid v_y \mid$. Conhecendo o eixo dominante, fica fácil calcular o pseudo-ângulo:

```
function pseudo(v:vetor):real;
var ax,ay,p:real;
begin
  ax:=abs(v.x); ay:=abs(v.y);
  if (ax>ay) then begin
    if (v.x>0) then begin p:=v.y/v.x; if p<0 then p:=p+8 end
               else p:=v.y/v.x+4
  end else begin
    if (v.y>0) then p:=2-v.x/v.y
               else p:=6-v.x/v.y;
  pseudo:=p;
end;
```

5.3.1 Pseudo-ângulo entre dois vetores-direções

O pseudo-ângulo formado por dois vetores-direções $\vec{u}$ e $\vec{v}$ pode ser determinado, na maioria das vezes, calculando:

$$\text{pseudo}(\vec{v}) - \text{pseudo}(\vec{u})$$

Esta expressão falha no caso como o da figura 5.2b. O pseudo-ângulo entre os vetores $\vec{u}$ e $\vec{v}$ vale um. Porém, calculando a expressão acima, obtemos:

$$\text{pseudo}(\vec{v}) - \text{pseudo}(\vec{u}) = 0.5 - 7.5 = -7.0$$

Este erro deve ser corrigido somando-se 8 (isto é, 360 graus) ao valor obtido. Portanto, a função que calcula o pseudo-ângulo entre dois vetores fica:

```
function pseudo2(u,v:vetor):real;
var r:real;
begin
  r := pseudo(v) - pseudo(u);
  if r<=-4 then r:=r+8;
  if 4<r then r:=r-8;
  pseudo2:=r;
end;
```

Esta função devolve um número real entre -4 e 4 que corresponde à distância que devemos andar sobre as arestas do quadrado direcional para chegar à intersecção de $\vec{v}$ com o quadrado partindo da intersecção de $\vec{u}$ com o quadrado.

5.4 Teste de pertinência de um ponto ao polígono

Um teste que aparece constantemente em todos os campos da Computação Gráfica e especialmente na área de síntese de imagem é o teste para verificar se um ponto está dentro ou fora de um polígono. Pela importância que tem este teste, que deve ser computacionalmente o mais rápido possível, dedicamos-lhe uma seção do nosso. Nesta seção, descreveremos este teste somente no caso 2D. Quando for necessário (e será necessário no rastreamento de raio) estenderemos este teste para o caso tridimensional. A versão tridimensional deste teste consiste em descobrir se uma semi-reta passa ou não passa através de um polígono dado.

5.4.1 Quando o polígono é uma janela

Vamos chamar de janela um retângulo perpendicular aos eixos das coordenadas (veja a figura 5.3). Uma janela fica definida especificando o seu diagonal, ou seja, fornecendo as coordenadas de dois vértices opostos. Em Pascal, uma janela pode ser definida como:

```
type janela=record Gmin, Gmax: ponto end;
```

Para verificar se um ponto $\dot{P}$ pertence ou não a uma janela G faça:

janela não-janela

Figure 5.3: Janela e não-janela

Se $(G_{min}.x \leq P.x \leq G_{max}.x)$ e $(G_{min}.y \leq P.y \leq G_{max}.y)$
então pertence senão não pertence;

5.4.2 Quando o polígono é convexo

Seja $G = (\dot{G}_0, \dot{G}_1, \ldots, \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$ um polígono convexo orientado em sentido anti-horário e $\dot{P}$ um ponto. O seguinte trecho de programa testa se $\dot{P} \in G$:

pertence:=true;
Para i:=0 até n-1 faça
Se $\dot{G}_i, \dot{G}_{i+1}$ e $\dot{P}$ estão, nesta ordem, em sentido horário
então pertence:=false;

Isto é, se todas as triplas $\dot{G}_i, \dot{G}_{i+1}, \dot{P}$ para $i \in 0 \ldots n-1$ estiverem em sentido anti-horário então o ponto $\dot{P}$ pertence ao polígono G.

5.4.3 Quando o polígono pode ser côncavo (caso geral)

Quando o polígono pode não ser convexo, o teste torna-se mais complexo. Existem dois métodos principais para resolver este problema:

Soma dos pseudo-ângulos

Neste teste, devemos somar pseudo2($\dot{G}_i - \dot{P}, \dot{G}_{i+1} - \dot{P}$) para $i \in 0 \ldots n-1$. Se esta soma for 8 pseudo-ângulos então $\dot{P} \in G$ e se for zero então $\dot{P} \notin G$ (veja a figura 5.4). Este processo pode ser esquematizado como segue:

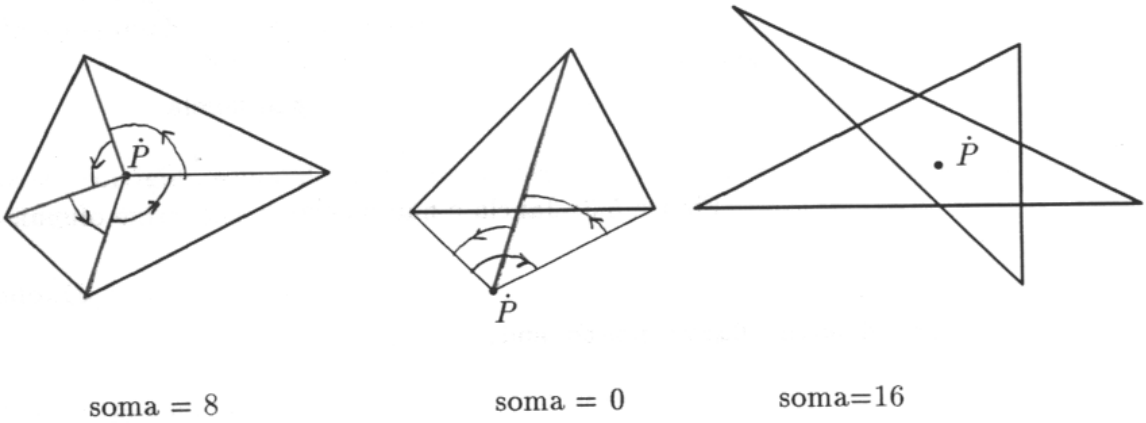


Figure 5.4: Soma dos pseudo-ângulos

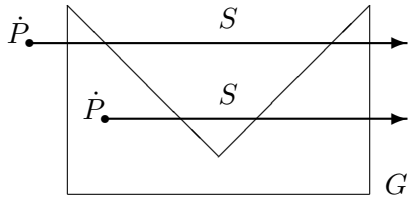


Figure 5.5: Número de intersecções de uma semi-reta com polígono

soma :=0;
Para i:=0 até $n-1$
  soma := *soma* + pseudo2($\dot{G}_i - \dot{P}$, $\dot{G}_{i+1} - \dot{P}$);
Se *soma*=8 então $\dot{P} \in G$;
Se *soma*=0 então $\dot{P} \notin G$;

Podem aparecer outros casos onde a soma resulta um múltiplo de 8, como vemos no último desenho da figura 5.4.

Número de intersecções de uma semi-reta com as arestas

Neste teste, contamos o número de arestas de G interceptadas por S, onde S é uma semi-reta qualquer com origem em $\dot{P}$. Se esse número for ímpar, $P \in G$. Caso contrário, $P \notin G$ (veja a figura 5.5). O algoritmo é:

cont:=0;
Seja S uma semi-reta qualquer com origem em $\dot{P}$;
Para i:=0 até $n-1$ faça
  Se S intercepta a aresta $(\dot{G}_i, \dot{G}_{i+1})$
    então *cont*:=*cont*+1;
Se *cont* mod $2 = 1$
  então $\dot{P} \in G$
  senão $\dot{P} \notin G$;

Poderíamos pensar que se a semi-reta S cruza um ou mais vértices de G então estes vértices deveriam receber um tratamento especial. Mas isto não é necessário, pois basta convencionar que, sempre que a semi-reta S cruzar um vértice $\dot{v}$, $\dot{v}$ está do lado esquerdo de S (ou convencionar que todo $\dot{v}$ interceptado por S fica do lado direito de S). Este teste poderia ser implementado utilizando a função H. Mas há uma maneira mais eficiente, descrita pelo seguinte trecho de programa em C:

```
1 for (i=0; i<=n; i++)
2  { H[i]=diferenca(G[i],P); }
3
4 cont=0;
5 for (i=0; i<n; i++) { j=i+1;
6  if ((H[i].y>=0.0) != (H[j].y>=0.0)) {
7   if ((H[i].x>=0.0) != (H[j].x>=0.0)) {
8    dx=H[j].x-H[i].x; dy=H[j].y-H[i].y;
9    x=-H[i].y*dx/dy+H[i].x; if (x>=0.0) cont++; }
```

```
10   else if (H[i].x>=0.0) cont++;
11  }
12 }
13 if (cont%2==0) return 0; else return 1;
```

Nas linhas 1 e 2, transladamos o polígono G e o ponto $\dot{P}$ de modo que a coordenada de $\dot{P}$ seja (0,0). Denominamos o polígono translado de H. Neste processo, efetuam-se duas subtrações para cada vértice do polígono. Como pode-se traçar a semi-reta S en qualquer direção, escolhemos a direção que facilite os cálculos: a direção do eixo x (a semi-reta S satisfaz $y = 0$ e $x \geq 0$).

Entre as linhas 4 e 10, conta-se o número de arestas que a semi-reta S cruza. Esta tarefa é simplificada tendo em conta os seguintes fatos (considere $j = i + 1$):

1. Se H_{iy} e H_{jy} têm o mesmo sinal, é impossível que a aresta H_iH_j cruze a semi-reta S.
2. Se H_{ix} e H_{jx} são negativos, com certeza H_iH_j não cruza a semi-reta S.
3. Se H_{ix} e H_{jx} são positivos e os sinais de H_{iy} e de H_{jy} são diferentes então com certeza H_iH_j cruza S.
4. Se nenhuma das três alternativas anteriores é verdadeira, restou apenas o caso onde os sinais de H_{iy} e de H_{jy} são diferentes e os sinais de H_{ix} e de H_{jx} também são diferentes. Neste caso, é preciso efetuar alguns cálculos aritméticos para verificar se há intersecção entre H_iH_j e S. Neste cálculo, efetuam-se três operações de soma/subtração, duas multiplicações/divisões e uma comparação. Repare que d_y é sempre diferente de zero, o que garante que não ocorrerá nenhuma divisão por zero.

Somando-se o número de operações em ponto flutuante, para cada vértice do polígono G, temos:

- Nas linhas 1 e 2, efetuam-se duas subtrações.
- Nas linhas 4 a 13:
  - Em 50% dos casos ($sinal(H_{iy}) = sinal(H_{jy})$) efetua-se apenas uma comparação.
  - Em 25% dos casos ($sinal(H_{iy}) \neq sinal(H_{jy}$) e ($sinal(H_{ix}) = sinal(H_{jx})$) efetuam-se 3 comparações.
  - Em 25% dos casos ($sinal(H_{iy}) \neq sinal(H_{jy}$) e ($sinal(H_{ix}) \neq sinal(H_{jx})$) efetuam-se 3 comparações, 4 somas ou subtrações e 2 multiplicações ou divisões.

Ao todo, temos, em média, 2 comparações, 3 somas/subtrações e 0.5 multiplicações/divisões (operações em ponto flutuante) para cada vértice do polígono G. Este teste pode ser acelerado pré-processando

o polígono para achar a "janela limitante[2]". Antes de efetuar o teste de pertinência para o caso geral, podemos testar se o ponto $\dot{P}$ pertence à janela limitante. Isto pode ser feito conforme explicamos na subseção 5.4.1. Se $\dot{P}$ não pertence à janela limitante de G, evidentemente $\dot{P}$ não pertence à G. Assim, aplicamos o teste geral somente se $\dot{P}$ pertence à janela limitante de G.

5.5 Preenchimento de polígono

Muitos algoritmos de síntese de imagem necessitam dos algoritmos de preenchimento de polígono para poderem funcionar (por exemplo, z-buffer e algoritmo de subdivisão da tela). Nesta seção, explicaremos dois algoritmos de preenchimento de polígono: chaveamento pela aresta e lista de arestas ordenadas. Também nesta seção, introduziremos o conceito de vértice par-ímpar, útil para o preenchimento de polígono. Outros algoritmos de preenchimento de polígono podem ser encontrados no livro [ProcElem]. Lá, como não se utiliza o conceito par-ímpar, os algoritmos tornam-se diferentes da maneira como são apresentados aqui. O problema de preenchimento de polígono pode ser definido:

Definição 5.2 *Dado um polígono* $G = (\dot{G}_0, \dot{G}_1, \ldots, \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$ *com as coordenadas inteiras (o polígono está no sistema S), determinar todos os pixels da tela que pertencem a* G.

O método mais simples é examinar cada pixel da tela para verificar se ele pertence ou não ao polígono. Esta técnica é ineficiente, pois provavelmente a maioria dos pixels não pertence ao polígono. Além disso, para verificar se um pixel pertence ao polígono, é necessário gastar um tempo da ordem do número de vértices do polígono.

5.5.1 Algoritmo de chaveamento pela aresta[3]

Um dos algoritmos de preenchimento de polígono é o chaveamento pela aresta que pode ser esquematizado como segue:

Passo 1: Dado um polígono $G = (\dot{G}_0, \dot{G}_1, \ldots, \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$ a ser preenchido, descubra a janela limitante J de G. Pinte J com a cor de fundo f.

Passo 2: Desenhe as arestas do polígono com a cor c tomando certos cuidados que serão descritos mais para frente (veja a figura 5.6a).

Passo 3: Depois, execute o seguinte algoritmo:

```
Dentro:=false;
Para y:=0 até ymax faça
   Para x:=0 até xmax faça
```

[2]Rounding box. É a menor janela que contém o polígono.
[3]Edge flag.

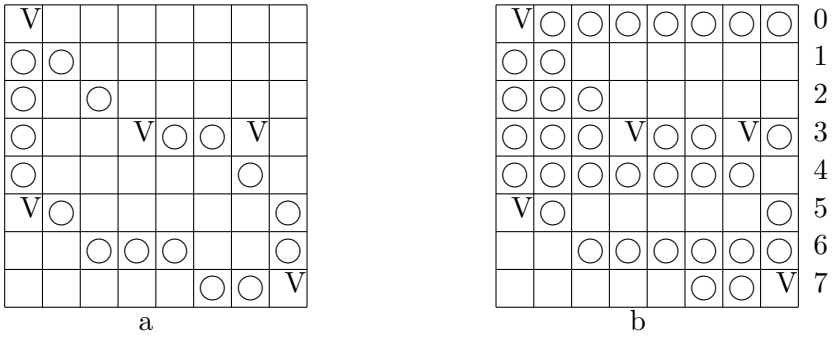


Figure 5.6: Solução errada para o problema de preenchimento de polígono.

```
      Se cor(x,y)=c então dentro:=not dentro;
      Se dentro então ponto(x,y,c);
   FimPara;
FimPara;
```

Passo 4: Desenhe novamente as arestas do polígono, pois haverá pontos da borda que não foram acesas.

A função $cor(x, y)$ devolve a cor do pixel (x, y). A variável *dentro* indica se o ponto (x, y) está dentro do polígono ou não. Se está dentro então pinta (x, y) com a cor c. A variável *dentro* muda toda vez que encontrar uma aresta do polígono, isto é, um pixel com a cor c. Observe a figura 5.6a. Ela mostra o estado da janela J após executar os dois primeiros passos. Ao executar o terceiro passo, as linhas 0, 3 e 5 serão pintadas incorretamente (figura 5.6b). Para corrigir este erro, é necessário tomar dois cuidados:

Cuidado 1: Quando traçar uma aresta no passo 2, em cada linha horizontal Y deve acender no máximo um ponto.

Cuidado 2: Existem alguns vértices (que chamaremos de vértices pares) que devem ser contados como se não existissem. Os vértices que devem ser contados como se houvesse um ponto aceso serão chamados, por sua vez, de ímpares.

Levando em conta estes dois cuidados, após executar o passo 2, obtemos a figura 5.7a em vez da figura 5.6a. E, executando o passo 3, obtemos a solução correta mostrada na figura 5.7b. Os pontos com círculos pretos não são acesos no passo 3 mas no passo 4, quando traçar novamente as arestas do polígono. Traçar as arestas de modo que em cada linha horizontal Y acenda no máximo um ponto pode ser satisfeita alterando o algoritmo DDA (procedimento *retaA3*) ou o de Bresenham (procedimento *retaA4*). Alterando o algoritmo de Bresenham, obtemos:

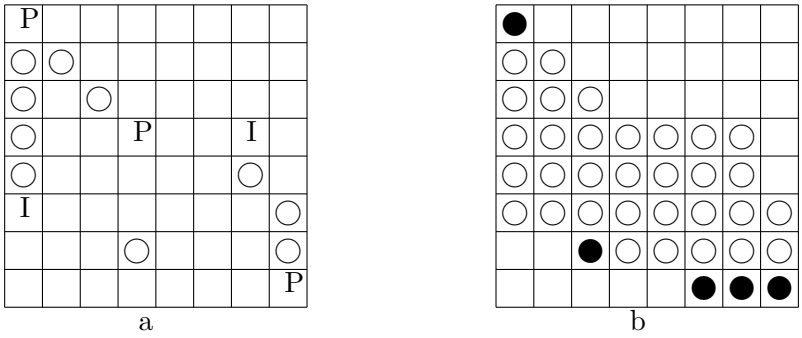


Figure 5.7: Solução correta para o problema de preenchimento de polígono.

```
procedure retaF(x1,y1,x2,y2,c:integer);
var x,y,dx,dy,a,err,dx2,dy2,sobe:integer;
begin
  dx:=x2-x1; dy:=y2-y1;
  if dy<0 then begin troca(x1,x2); troca(y1,y2); dy:=-dy; dx:=-dx; end;
  a:=abs(2*dx); err:=0; dy2:=dy*2; sobe:=sinal(dx); x:=x1;
  for y:=y1 to y2-1 do begin
    ponto(x,y,c); err:=err+a;
    while err>=dy do begin
      x:=x+sobe; err:=err-dy2;
    end;
  end;
  ponto(x2,y2,c);
end;
```

O último ponto é aceso fora do loop para assegurar que irá acender exatamente o vértice $(x2, y2)$. Isto é necessário para que o vértice $(x2, y2)$ possa ser apagado se ele for par. Só resta-nos saber como determinar quando um vértice é par e quando é ímpar. De um modo informal, um vértice é par se os seus dois vértices vizinhos estão num mesmo lado da reta $y = y_i$, onde y_i é a coordenada y do vértice $\dot{G}_i$, e $\dot{G}_i$ será ímpar se os seus dois vértices vizinhos estão em lados opostos da reta $y = y_i$. Isto pode ser testado calculando:

$$S = sinal(y_i - y_{i-1})\ sinal(y_{i+1} - y_i)$$

Se $S = 1$ então G_i é ímpar. Se $S = -1$ então G_i é par. Mas se $S = 0$, não se pode afirmar nada sobre G_i. Veja a figura 5.8 onde todos os vértices têm $S = 0$. Alguns desses vértices são pares e outros

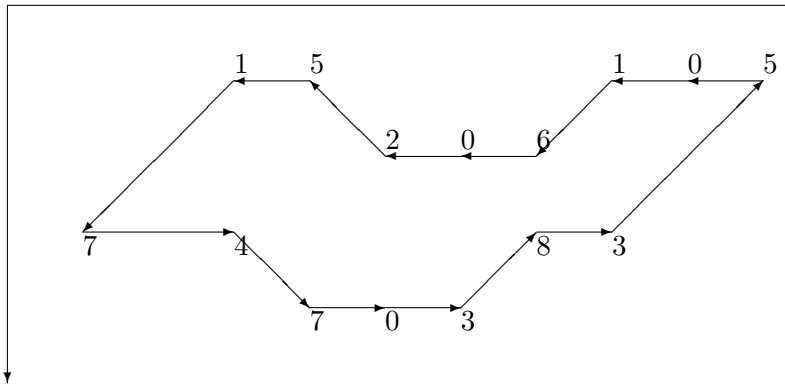


Figure 5.8: Vértices com $S = 0$.

são ímpares. Um vértice com $S = 0$ pertence a um dos nove casos possíveis, numerados de 0 a 8, como se observa na figura 5.9. Um vértice com $S = 0$ que pertence a um caso par será um vértice par e aquele que pertence a um caso ímpar será um vértice ímpar.

O conceito de vértice par-ímpar pode ser aplicado para outros problemas além do preenchimento de polígono, como:

1. Teste de inclusão de um ponto num polígono.
2. Hachuramento de polígono.
3. Corte de um polígono por uma reta.
4. Cálculo de "invisibilidade quantitativa", utilizada nos algoritmos de eliminação de arestas escondidas.

Como veremos na seção 5.7, estes problemas também podem ser resolvidos utilizando o método de perturbação. Porém, o método de perturbação não funciona de modo apropriado se os vértices têm coordenadas inteiras. Neste caso, deve-se utilizar o conceito de vértice par-ímpar.

Considere, por exemplo, o problema de corte de polígono por reta. O conceito de vértice par-ímpar é facilmente aplicável se a reta é horizontal ou vertical. Mas é preciso aprofundar os estudos para se certificar de que este método funciona mesmo que a reta esteja inclinada. A mesma observação vale para os três outros problemas.

O algoritmo de chaveamento pela aresta somente foi apresentado aqui porque facilita a explicação do próximo algoritmo. Pois o algoritmo de chaveamento (e também praticamente todos os outros algoritmos de preenchimento de polígono) necessitam da garantia de que dentro da janela limitante não existem pontos de cor c. Nós garantimos isto preenchendo toda a janela limitante com a cor de fundo. Evidentemente, há muitos casos em que não se pode apagar toda a janela limitante. Basta pensar no

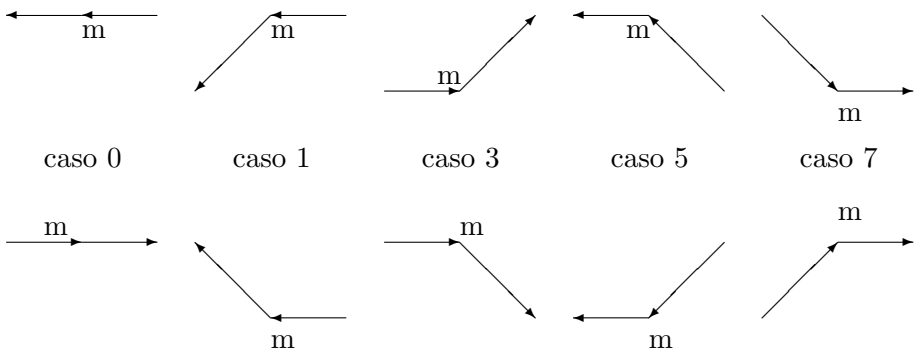


Figure 5.9: Todos os possíveis casos de vértices com $S = 0$.

caso em que o fundo já está desenhado e queremos acrescentar um polígono por cima do desenho de fundo. Neste caso, seria desastroso se tivéssemos que apagar toda a janela limitante. Outras vezes, o polígono não será pintado com uma única cor, o que também impede que possamos utilizar o algoritmo de chaveamento pela aresta. Precisamos de um algoritmo que descubra exatamente quais são os pixels que pertencem ao polígono sem recorrer às informações armazenadas na tela. O algoritmo de lista de arestas ordenadas satisfaz isso.

5.5.2 Lista de arestas ordenadas[4]

O algoritmo lista de arestas ordenadas é como segue:

Passo 1: Tomando os dois cuidados vistos na subseção anterior, trace as arestas do polígono. Mas, em vez de acender os pixels da tela, armazene as coordenadas dos pixels que seriam acesos num vetor V.

Passo 2: Ordene o vetor V tendo como chave primária y e como chave secundária x.

Passo 3: Extraia os elementos do vetor V aos pares. Sejam (x_1, y_1) e (x_2, y_2) um par extraído. Como o vetor está ordenado, devemos ter $y_1 = y_2$ e $x_1 \leq x_2$. Acenda todos os pixels da linha y_1 que tenham coordenada x entre x_1 e x_2, inclusive. Repita este passo até que todos os elementos tenham sido extraídos do vetor V.

[4]Ordered edge list.

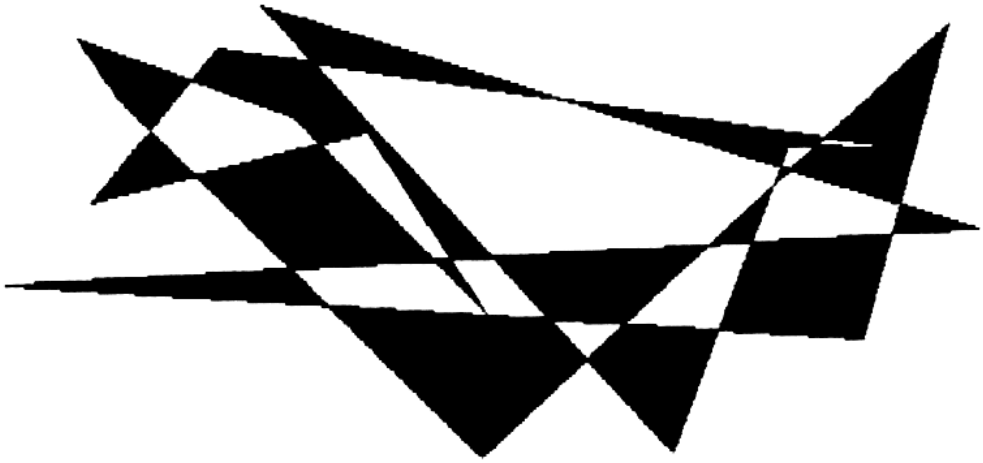


Figure 5.10: Preenchimento de polígono usando a lista de arestas ordenadas.

Passo 4: Trace novamente as arestas do polígono, pois isto irá acender alguns pontos que não foram acesos no passo 3.

Este algoritmo pode ser acelerado construindo uma lista ligada para cada linha de rastreio y. Nelas, serão armazenadas as coordenadas x das arestas que seriam armazenadas no vetor V pelo algoritmo original. Com isto, não há mais necessidade de ordenar todo o vetor V mas somente cada uma das listas ligadas. Este algoritmo foi implementado pelo autor e a sua saída pode ser vista na figura 5.10.

5.6 Corte de polígono

Existem dois problemas básicos, úteis em todas as áreas de Computação Gráfica. São eles o problema de corte de um polígono por uma reta e o problema de recorte de um polígono contra uma janela. Os dois estão muito relacionados entre si. É necessário conhecê-los para minimizar o gasto de memória e de tempo no z-buffer.

5.6.1 Corte de polígono por reta

O problema de corte de polígono por reta pode ser enunciado da seguinte maneira:

Definição 5.3 *Dado um polígono G e uma reta R, calcule os polígonos que resultam ao dividir G por R.*

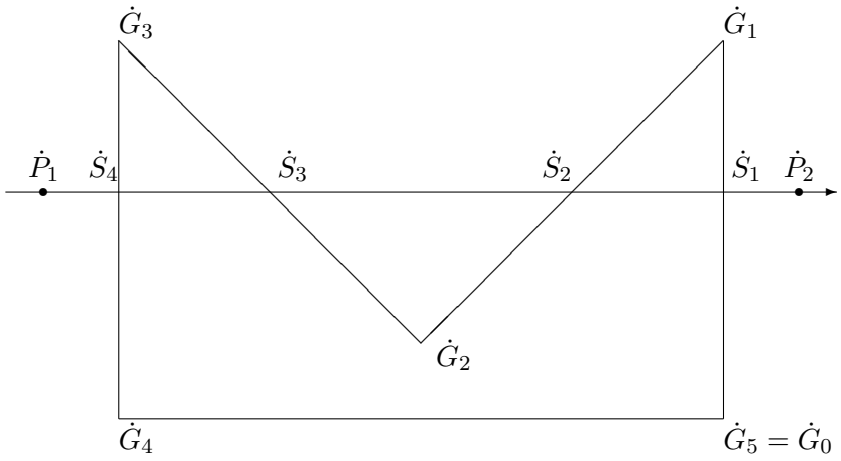


Figure 5.11: Corte de polígono por reta.

Vamos expor o algoritmo supondo que não existem vértices de G interceptados pela reta R. Depois, explicaremos o que fazer se aparecerem tais vértices. Sejam dados o polígono $G = (\dot{G}_0, \dot{G}_1, \ldots, \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$ e a reta $\dot{R}(t) = (\dot{P}_2 - \dot{P}_1)t + \dot{P}_1$. Acompanharemos o algoritmo com o exemplo da figura 5.11.

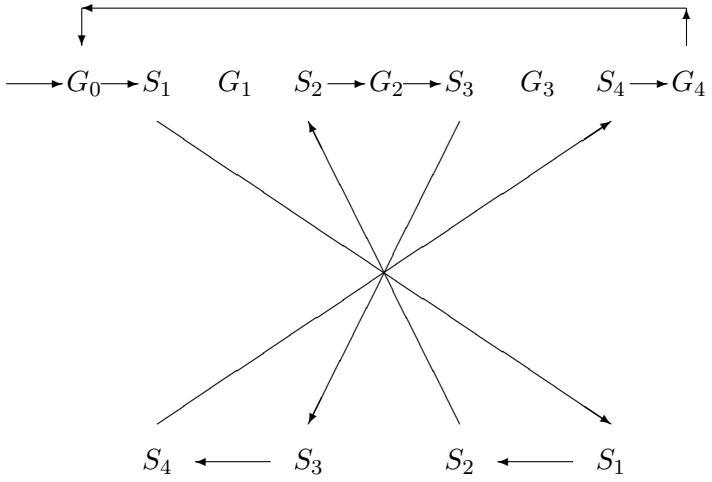
a) Ache a intersecção de cada uma das arestas de G com R e insira as coordenadas dessas intersecções na lista de vértices do polígono G e numa lista à parte que chamaremos de S. No nosso exemplo, construimos:

$$G = (G_0, S_1, G_1, S_2, G_2, S_3, G_3, S_4, G_4)$$

$$S = (S_1, S_2, S_3, S_4)$$

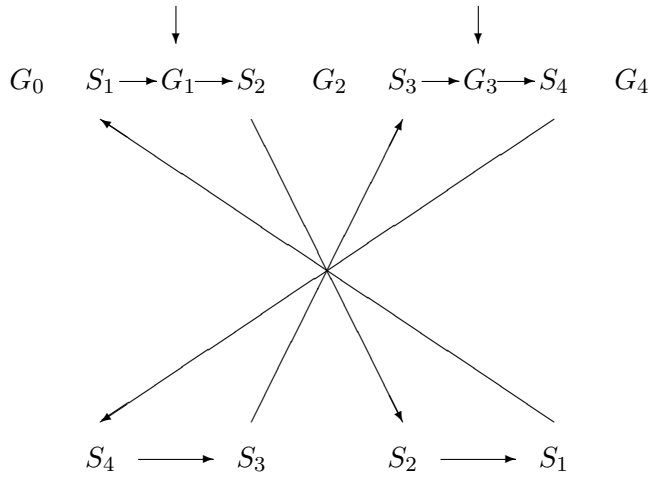
b) Como os pontos S_i pertencem à reta R, $\dot{S}_i = (\dot{P}_2 - \dot{P}_1)t_i + \dot{P}_1$ para algum valor t_i. Ordene a lista S pelo valor t_i. No exemplo, $S = (S_4, S_3, S_2, S_1)$.

c) Os pontos S_i, reunidos dois a dois, formam novas arestas dos polígonos resultantes da corte. Assim, S_4S_3 e S_2S_1 são as novas arestas dos polígonos após a corte. Percorra as listas G e S, criando os polígonos resultantes da corte. Comece a percorrer, por exemplo, a partir de G_0. Quando se encontra um ponto S_i, pule para a lista S, percorra uma aresta e volte para a lista G. Este processo se repete até voltar ao ponto de partida. Então, achamos um polígono resultante (veja a figura 5.12). Para achar os outros polígonos, ache um vértice G_i que ainda não foi percorrido e repita o processo (veja a figura 5.13).



Polígono resultante $H_1 = (G_0, S_1, S_2, G_2, S_3, S_4, G_4)$

Figure 5.12: Corte de polígono: montagem de polígono resultante



Polígonos resultantes $H_2 = (G_1, S_2, S_1)$ e $H_3 = (G_3, S_4, S_3)$

Figure 5.13: Corte de polígono: outros polígonos resultantes

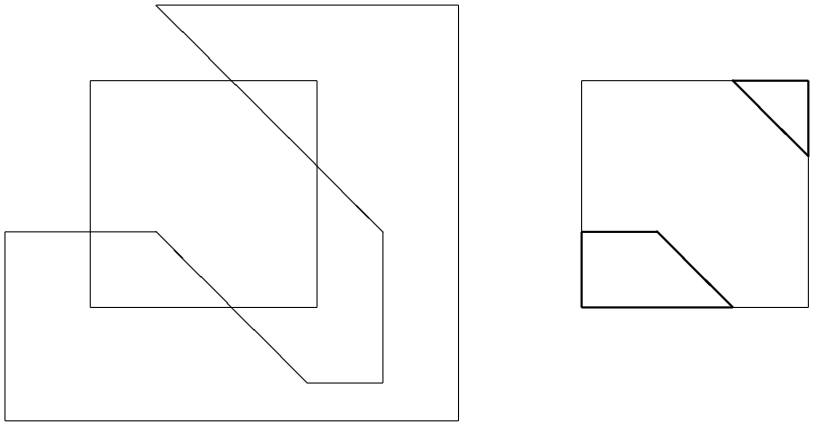


Figure 5.14: Recorte de polígono contra janela

d) Entre os polígonos resultantes, aqueles que percorreram as arestas de S da esquerda para direita ficam à esquerda de R. No nosso exemplo, são os polígonos H_2 e H_3. Aqueles que percorreram as arestas de S da direita para esquerda ficam à direita de R.

O autor implementou este algoritmo mas a sua saída não foi colocada aqui pois é numérica, isto é, o programa devolve as coordenadas dos vértices dos polígonos resultantes.

Se algum vértice de G é cruzado por R então o algoritmo acima deixa de funcionar. A solução mais simples para este problema é utilizar o método de perturbação perpendicular, conforme a subseção 5.7.3. A probabilidade deste método não funcionar é da ordem de $1/10^5$ quando se trabalha com variáveis precisão simples e é da ordem de $1/10^{15}$ quando se trabalha com variáveis precisão dupla.

5.6.2 Recorte de polígono contra janela

Para determinar as partes de um polígono que ficam dentro de uma janela basta aplicar o algoritmo de corte de polígono por reta quatro vezes. Observe que nós queremos obter a descrição das superfícies dos polígonos que ficam dentro da janela e não a descrição das partes das arestas que pertencem à janela. Veja a figura 5.14.

5.6.3 Corte de polígono com furos por reta

Um polígono com furos, descrita conforme a seção 3.1, pode ser cortada por uma reta alterando ligeiramente o algoritmo já visto. Primeiro aplicamos o algoritmo de corte normal. Se apareceram um ou mais furos, deve-se decidir a qual dos polígonos pertence cada um dos furos. No exemplo da figura 5.15, após

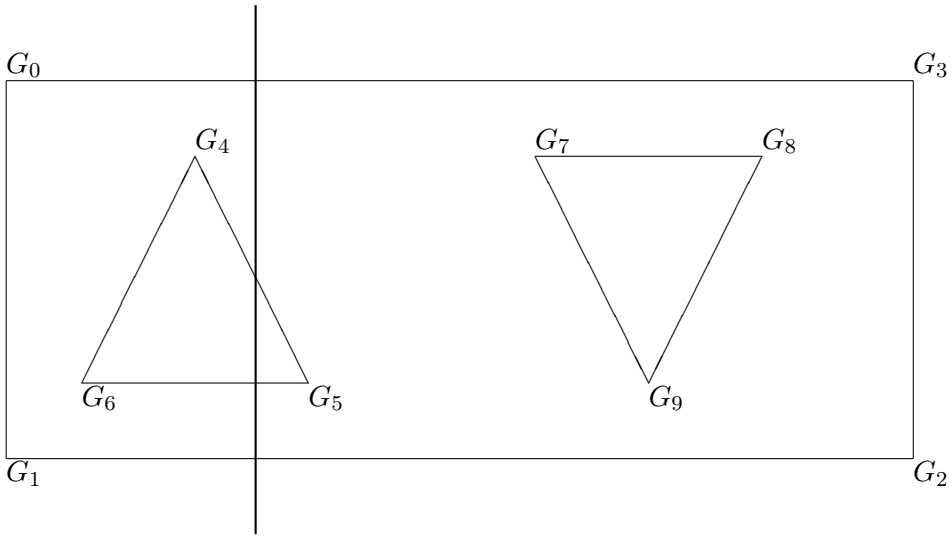


Figure 5.15: Corte de polígono com furo

ter aplicado o algoritmo de corte de polígono já visto, deve-se decidir a qual dos dois polígonos resultantes pertence o furo (G_7, G_8, G_9).

5.7 Método de Perturbação

5.7.1 Introdução

Algumas singularidades difíceis de serem resolvidas podem aparecer em muitos algoritmos de Computação Gráfica. Por exemplo, no algoritmo de corte de polígono, o que se deve fazer quando a reta de corte cruza um dos vértices do polígono ou quando ela coincide com uma das arestas (figura 5.18a)? Veremos nesta seção uma técnica para resolver estes problemas. As singularidades como essas podem ser classificadas em:

1. Singularidade VxV: O par de vértices (v, u) é uma singularidade VxV se eles têm as mesmas coordenadas.

2. Singularidade VxA: O par (v, e) é uma singularidade VxA se a aresta e passa justamente em cima do vértice $\dot{v}$.

3. Singularidade AxA: O par (e_1, e_2) é uma singularidade AxA se as arestas e_1 e e_2 coincidem em mais de um ponto.

O vértice (ou a aresta) será chamado singular se pertence a alguma singularidade. Às vezes, é conveniente considerar dois objetos muito próximos um do outro como singulares pois as variáveis em ponto flutuante têm precisão finita. A frequência com que aparecem estas singularidades seria muito baixa se as coordenadas dos vértices, fossem aleatórias. Porém, como o usuário, ao fornecer as coordenadas dos vértices fornece de preferência números inteiros ou com poucas casas decimais, as singularidades aparecem muito mais frequentemente do que poderia parecer à primeira vista. Os seguintes algoritmos têm problemas com as singularidades. Mas existem muitos outros com os mesmos problemas.

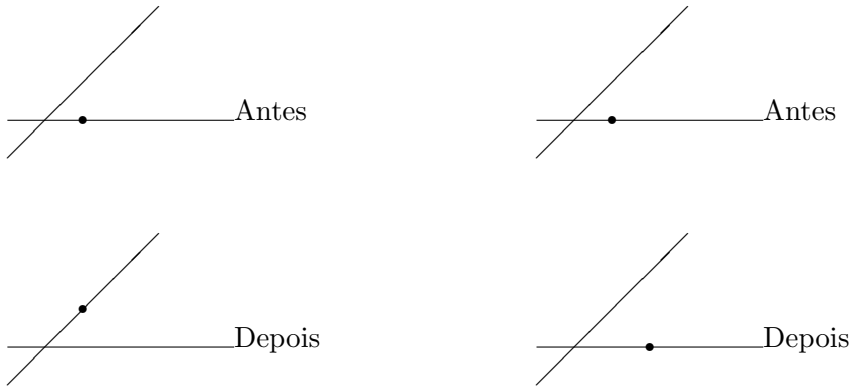
- Operações booleanas (união, intersecção e diferença) entre dois polígonos ou poliedros (figura 5.20).
- Preenchimento e hachuramento de polígono (figura 5.17).
- Corte de um polígono por uma reta ou de um poliedro por um plano (figura 5.18a).
- Cálculo da "invisibilidade quantitativa" utilizada nos algoritmos de eliminação de arestas escondidas.
- Cálculo do volume de um sólido modelado por CSG usando o algoritmo de lançamento de raio[5].

Um tipo de singularidade ligeiramente diferente também aparece em programação linear. Nesta área, a singularidade é chamada de degeneração. O "problema de precisão" (veja a seção 10.6) também pode ser pensado como sendo um problema de singularidade. Diante destas singularidades, três atitudes são possíveis nos cientistas de computação e nos programadores:

1. Ignorar as singularidades. Em [Algorithms, p 313] o teste de intersecção entre dois segmentos de reta não funciona se as quatro extremidades dos segmentos são colineares.
2. Perturbar o vértice ou a aresta singular e reaplicar o algoritmo. [ProgLin] adota esta posição. Sugere o método de perturbação sem explicar como se pode escolher na prática o valor de ε, que deve ser "suficientemente pequeno" para não gerar outras singularidades.
3. Prever e resolver sistematicamente todas as singularidades. [SolidMod] prefere esta filosofia. Com isso, evidentemente, os algoritmos tornam-se muito mais complicados.

Provavelmente o método de perturbação é a técnica mais simples para eliminar as singularidades. Porém os autores relutam em usar este método por não dispor de garantias da sua confiabilidade. Para elucidar este quadro confuso, iremos analisar detalhadamente o método de perturbação e calcular a sua probabilidade de falha p. Também iremos descrever os cuidados que se deve tomar para que p seja pequena. Um algoritmo clássico, conhecido e utilizado por praticamente todos os cientistas de computação é o quicksort. Existe uma pequena probabilidade de quicksort funcionar mal e gastar um tempo muito grande para ordenar um vetor. Mas esta probabilidade é tão remota que, na prática, todos utilizam quicksort como o algoritmo de ordenação preferido. O mesmo pode acontecer com o método de perturbação, se tomarmos os devidos cuidados.

[5] Ray casting



a: reincidência α b: reincidência β

Figure 5.16: Tipos de reincidência

5.7.2 Reincidência na singularidade tipo α e tipo β

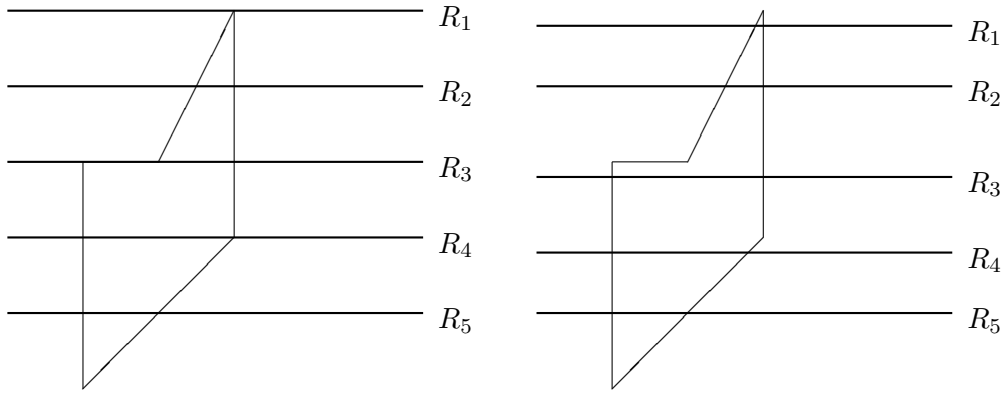
A singularidade que aparece mais frequentemente é VxA. Normalmente, os outros dois casos (VxV e AxA) podem ser considerados como casos particulares de VxA. O método de perturbação falha quando, ao efetuarmos uma perturbação, acabamos gerando uma nova singularidade. Esta reincidência na singularidade pode ser classificada em dois:

- Tipo α: Um par (v, e) que não era uma singularidade antes da perturbação torna-se singular após a perturbação (figura 5.16a). Este tipo de reincidência tem baixa probabilidade de ocorrência.
- Tipo β: Um par (v, e), que constituía uma singularidade antes da perturbação, continua sendo singular após a perturbação (figuras 5.16b, 5.20a e 5.20b). Este tipo de reincidência tem probabilidade de ocorrência muito maior que o tipo α. Portanto, iremos desenvolver métodos especiais para evitá-lo. Como veremos, existem algoritmos onde é impossível que ocorra a reincidência β.

5.7.3 Algoritmos sem singularidade VxV

Considere o algoritmo de hachuramento de um polígono mostrado na figura 5.17a. As retas de hachuramento R_2 e R_5 não são singulares. Assim sendo, basta traçar um segmento de reta de uma intersecção até a outra para processá-los[6]. As retas de hachuramento R_1, R_3 e R_4 porém, fazem parte de singularidades

[6]Se a reta de hachuramento possui mais de duas intersecções, devemos ordenar essas intersecções pelas suas coordenadas x e traçar segmentos de reta de duas em duas intersecções.



a: hachuramento com singularidades b: perturbação perpendicular

Figure 5.17: Hachuramento de polígono

VxA, AxA e VxA, respectivamente. Nestes casos, a técnica habitual falha. Porém, perturbando R_1, R_3 e R_4 no sentido perpendicular desaparecem as singularidades, como se observa na figura 5.17b. Observe que aqui não pode existir o caso VxV. Logo, perturbando as retas de hachuramento singulares no sentido perpendicular, é impossível que ocorra a reincidência tipo β.

A probabilidade de reincidência p na singularidade α pode ser calculada fazendo as seguintes simplificações:

- Suposição 1: Como foi observado na subseção 5.7.2, AxA é um caso particular de VxA. Basta, pois, calcular a probabilidade de reincidir no caso VxA.

- Suposição 2: Considere que as coordenadas dos vértices estão armazenados em variáveis ponto flutuante com b bits de mantissa. No exemplo 5.2 será analisado o que acontece se utilizar variáveis inteiras. Sejam $l = 2^b - 1$ e δ o maior número entre os valores absolutos de todas as coordenadas de vértices. Um número em ponto flutuante x_f será considerado, para efeito de cálculo de probabilidade, como sendo $x_i = trunca \mid l\, x_f/\delta \mid$, ou em termos de linguagem C: `xi=(long int) fabs(l*xf/delta)`. A função que converte x_f em x_i será chamada *inteira*. $Inteira(x_f)$ é sempre um número inteiro entre 0 e l, ou seja, que pode ser armazenado em b bits. Esta simplificação é razoável, uma vez que normalmente todas as coordenadas têm ordens de grandeza semelhantes. Além disso, se fizesse os cálculos sem esta simplificação, a probabilidade de reincidência p seria menor. Daqui para frente, aplicar a função *inteira* a um ponto $\dot{P}$ irá significar aplicar a função *inteira* a cada uma das coordenadas de $\dot{P}$, para simplificar a notação.

- Suposição 3: Supõe-se que as coordenadas dos vértices, após aplicar a função *inteira*, estão distribuídas uniformemente em todo intervalo de 0 a l.
- Suposição 4: Vamos perturbar uma aresta e. Iremos considerar que a probabilidade de reincidência após esta perturbação e a probabilidade de aresta e causar uma singularidade quando as suas coordenadas são sorteadas aleatoriamente (porém mantendo o comprimento inalterado) são iguais.

Sejam e uma aresta perturbada e d_i o comprimento inteiro de e, ou seja, se e tem extremidades $\dot{P}$ e $\dot{Q}$, $d_i = inteira(\sqrt{(P_x - Q_x)^2 - (P_y - Q_y)^2})$. Pegue um vértice $\dot{v}$ e seja $v_i = inteira(\dot{v})$. Cada coordenada de v_i é um número entre 0 e l, pela suposição 2. A probabilidade de que este vértice esteja justamente em cima da aresta e é $p < d_i/l^2$, pelas suposições 3 e 4. A exponenciação por dois é devido ao fato de estarmos trabalhando no plano. E a probabilidade p de que entre m vértices exista pelo menos um vértice em cima da aresta perturbada satisfaz a seguinte inequação:

$$p < \frac{md_i}{l^2} \tag{5.1}$$

desprezando os termos com graus maiores. Sugerimos que a distância a ser perturbada seja aproximadamente $\varepsilon \approx 20\delta/l$.

Exemplo 5.1 *Suponha que se está hachurando um polígono com 10 vértices. Suponha também que estão sendo utilizadas variáveis em ponto flutuante com mantissa de 24 bits. Então, pela inequação 5.1, a probabilidade de falha do método de perturbação é:*

$$p < \frac{10 \times 2^{24}}{(2^{24})^2} \approx \frac{1}{167000}$$

Se utilizasse variável dupla precisão com 53 bits de mantissa teria:

$$p < \frac{10 \times 2^{53}}{(2^{53})^2} \approx \frac{1}{10^{15}}$$

Ou seja, utilizando variável dupla precisão, pode-se considerar que é praticamente impossível que ocorra uma reincidência na singularidade.

Se necessitar de probabilidade de falha ainda menor, pode-se verificar explicitamente se ocorreu uma reincidência α e, caso tenha ocorrido, tentar uma outra perturbação. Mas este teste leva tempo $O(n_v)$ para cada aresta perturbada, onde n_v é o número de vértices. Como a probabilidade de ocorrência é baixa, julgamos que na maioria das aplicações não vale a pena perder este tempo computacional.

Exemplo 5.2 *Suponha que se está preenchendo um polígono com 10 vértices. As coordenadas dos vértices são números inteiros que indicam as coordenadas da tela do computador. Suponha que a tela possui 500 × 500 pixels. Então:*

$$p < \frac{10 \times 500}{500^2} \approx \frac{1}{50}$$

Com o que se pode concluir que neste problema não se deve utilizar o método de perturbação.

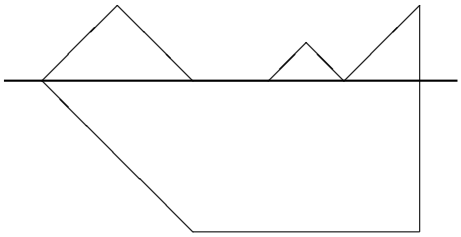


Figure 5.18: Corte de polígono

No algoritmo de corte de um polígono por uma reta (figura 5.18a) pode-se utilizar a mesma perturbação vista nesta seção e aplicar a mesma análise feita nos exemplos 5.1 e 5.2. Nos problemas estudados, pode-se também optar por perturbar os vértices singulares em vez de perturbar as arestas. Com isso, nenhuma reincidência (tanto α como β) poderá ocorrer.

5.7.4 Algoritmos com singularidade VxV

Na subseção anterior, vimos que nos algoritmos sem singularidade VxV não há possibilidade de ocorrer reincidência tipo β se efetuarmos uma perturbação perpendicular na aresta. Isto, porém, não é verdade se pode ocorrer caso VxV. Observe a figura 5.20a. Se a perturbação for perpendicular à aresta, acaba recaindo na singularidade inexoravelmente. É a reincidência tipo β. Esta reincidência tem uma probabilidade de ocorrência muito maior que a reincidência α. Para eliminar as singularidades nos algoritmos onde podem existir casos VxV, propomos duas soluções:

1. Efetue repetidamente as perturbações aleatórias, até que não ocorra reincidência tipo β.
2. Efetue repetidamente as perturbações no sentido perpendicular até que não ocorra reincidência β.

Observe que o teste para verificar a ocorrência de reincidência β é rápido. Leva tempo $O(k)$ onde k é o número de arestas que incidem nos (ou passam muito próximos a) vértices singulares.

Perturbações aleatórias repetidas

A primeira solução proposta é efetuar repetidamente perturbações aleatórias até que não ocorra nenhuma reincidência β. A probabilidade de ocorrer uma reincidência β quando efetuamos uma perturbação aleatória pode ser calculada da seguinte maneira. Seja ε a distância perturbada, ou seja,

$$\varepsilon = \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2} \quad \text{e} \quad \varepsilon_i = inteira(\varepsilon)$$

Onde $\dot{v}$ é o vértice original e $\dot{u}$ é o vértice perturbado. Sejam $v_i = inteira(\dot{v})$, $u_i = inteira(u)$ e $k =$ número de arestas que incidem em $\dot{v}$ ou em $\dot{u}$. Existem $2\pi\varepsilon_i$ pontos que distam ε_i de v_i, das quais k

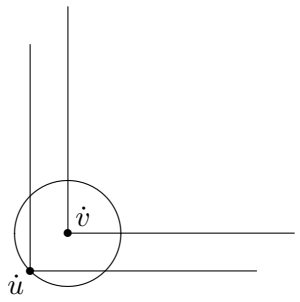


Figure 5.19: Perturbação aleatória

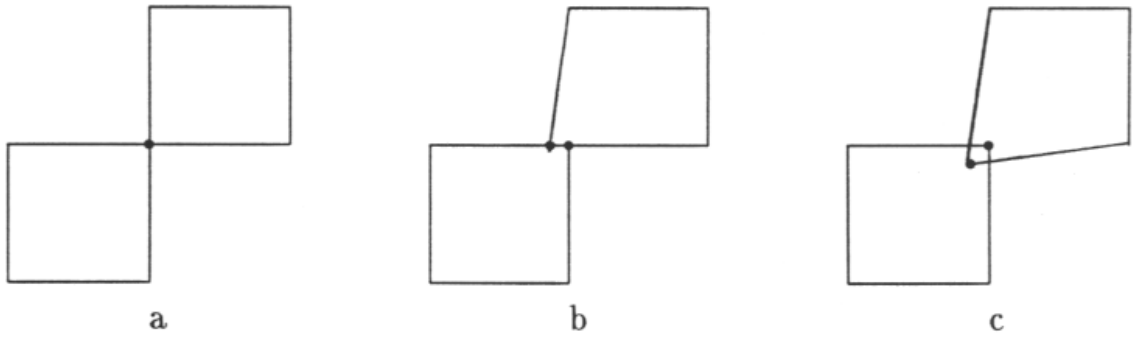


Figure 5.20: Operações booleanas com perturbações perpendiculares

causam singularidades. Veja a figura 5.19. Logo, a probabilidade de reincidência β, após perturbar um vértice que pertence ao caso VxV é:

$$p = \frac{k}{2\pi\varepsilon_i} < \frac{k}{6\varepsilon_i} \tag{5.2}$$

Do qual podemos concluir que convém escolher ε grande suficiente para diminuir reincidência tipo β mas que ao mesmo tempo deve ser pequeno bastante para não alterar de modo significativo o desenho final.

Exemplo 5.3 *Considere o cálculo de união entre dois polígonos. Neste algoritmo pode aparecer caso* VxV. *Sugerimos que utilize:*

$$\varepsilon \approx 2^{-0.6b}\delta. \tag{5.3}$$

Utilizando variáveis de dupla precisão, teremos com ε acima

$$p < \frac{k}{6\varepsilon_i} = \frac{k}{6\ inteira(2^{-0.6b}\delta)} = \frac{k}{14000000}.$$

Como k normalmente é menor que 10, a probabilidade acima é bastante pequena. Se utilizar variável precisão simples obteremos $p < k/4600$.

Perturbações perpendiculares repetidas

Observe novamente a figura 5.20. Após a primeira perturbação perpendicular caimos no caso VxA. E, após a segunda perturbação perpendicular, o caso singular desaparece. Qual é a probabilidade de após a segunda perturbação persistir a singularidade? Será igual à probabilidade de ocorrer a reincidência tipo α perturbando o vértice que forma caso VxA? Infelizmente, isto não é verdade. Pois, após a segunda perturbação, os vértices $\dot{v}$ e $\dot{u}$ estão separados por uma distância ε. Portanto, existe a probabilidade $p < k\,/\,6\varepsilon_i$ de ocorrer reincidência β, como vimos na subsubseção anterior. Sugerimos que se utilize o mesmo ε visto na equação 5.3.

Part II

Eliminação de Arestas e Superfícies Escondidas

Chapter 6

Eliminação de Arestas Escondidas

A imagem mais simples de se obter é a "armação de arame[1]". Ela consiste em mostrar apenas as arestas dos objetos. Uma armação de arame pode ter as arestas escondidas eliminadas ou não. Para se criar uma armação de arame sem eliminar as arestas escondidas, basta aplicar as transformações geométricas (vistas na seção 4.11) nos objetos. Exemplificando, se o ambiente consiste de um conjunto de poliedros, primeiro aplique as transformações geométricas nos vértices dos objetos, escrevendo todos os vértices na coordenada da tela. Depois, trace uma linha entre cada par de vértices que constituem as extremidades de uma aresta. O problema torna-se mais complexo se queremos eliminar as arestas invisíveis. Vamos definir:

Definição 6.1 *Uma aresta é* **totalmente invisível** *se entre ela e o observador existe uma ou mais faces que a tornam invisível. Uma aresta é* **parcialmente visível** *se entre ela e o observador existe uma ou mais faces que ocultam-na parcialmente. Uma aresta é* **totalmente visível** *se não existe nenhuma face entre ela e o observador.*

Eliminando as arestas escondidas, a imagem torna-se mais clara. Compare as figuras 6.1 e 6.2.

6.1 Faces e arestas irrelevantes

Definição 6.2 *Seja F uma face do poliedro H. Vamos chamar de* **normal externo**[2] *à F ao vetor unitário, ortogonal à face F, que aponta para fora de H.*

Definição 6.3 *Uma face, descrito em sistema U[3] (isto é, aplicou-se a ela a transformação em perspectiva) é* **irrelevante**[3] *se o produto escalar entre o seu normal externo e a direção de visão $\vec{F}$ for positivo. Caso contrário, ela será* **relevante**.

[1] Wire-frame.
[2] Outward-pointing normal.
[3] Back-face.

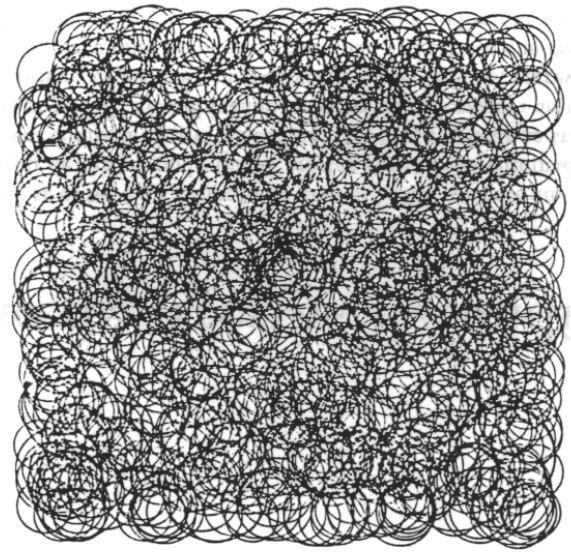


Figure 6.1: Armação de arame sem eliminar as arestas escondidas

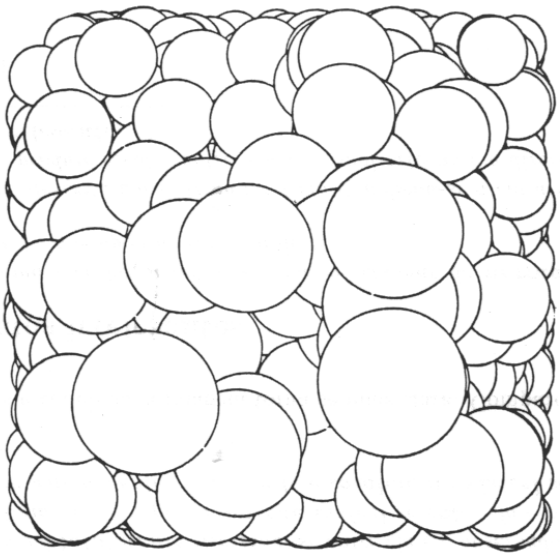


Figure 6.2: Armação de arame eliminando as arestas escondidas

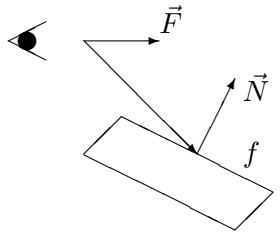


Figure 6.3: O conceito de face irrelevante não vale no sistema E

Uma face irrelevante é sempre totalmente invisível, pois ela é escondida pelo próprio sólido ao qual faz parte. Porém, o inverso nem sempre é verdadeiro, isto é, existem faces relevantes que também são invisíveis (total ou parcialmente). Eliminando todas as faces irrelevantes, podemos acelerar o algoritmo de eliminação de superfícies escondidas de modo significativo, pois aproximadamente a metade de todas as faces são irrelevantes. Observe que o conceito de face irrelevante só é válido dentro do sistema U^3. No sistema E, pode acontecer casos como o da figura 6.3. A face f é visível ao observador apesar de ser irrelevante, pois $\vec{F} \circ \vec{N} > 0$. No sistema U^3 não acontecem casos como este, pois em U^3 é como se o observador estivesse situado em $(0, 0, -\infty)$ e, portanto, todos os raios que chegam ao observador são paralelos. Vamos definir agora a aresta irrelevante:

Definição 6.4 *Uma aresta é* **irrelevante** *se ela é intersecção de duas faces irrelevantes. Caso contrário, ela será relevante.*

Uma aresta irrelevante sempre é totalmente invisível ao observador. Ela é sempre escondida pelo sólido ao qual pertence. Porém, existem arestas relevantes que também são invisíveis (total ou parcialmente). Nos algoritmos de eliminação de arestas escondidas podemos, portanto, eliminar todas as arestas irrelevantes. Isto não irá alterar a imagem final criada. Nesses algoritmos, além das arestas irrelevantes, também podemos eliminar as faces irrelevantes. Pois, como uma face irrelevante f nunca é visível, podemos concluir que sempre existe uma face relevante g entre f e o observador. Isto implica que se uma face irrelevante f esconde uma aresta e então sempre existe uma face relevante g que também esconde e. Deste modo, não perdemos nenhuma informação ao eliminarmos todas as faces irrelevantes. O conceito de aresta irrelevante também só é válido dentro do sistema U^3.

Se no ambiente há apenas poliedros convexos onde cada poliedro não esconde nenhum outro poliedro então uma aresta é invisível se, e somente se, é irrelevante. Observe que neste caso, não existem arestas parcialmente invisíveis. Usando estes fatos, é possível determinar rapidamente se uma aresta é visível ou invisível em tal ambiente. O processo resume-se em aplicar a transformação em perspectiva e depois calcular um produto escalar para cada face do sólido para determinar quais são as faces irrelevantes e, com essa informação, determinar as arestas irrelevantes.

Se não estamos interessados em mostrar os objetos em perspectiva, o problema torna-se mais simples ainda, pois não há mais necessidade de aplicar a transformação em perspectiva. Então, basta calcular

um produto escalar para cada face do sólido. Isto é tão rápido que mesmo num computador como PC ou Apple é possível mostrar uma armação de arame de um poliedro convexo rotacionando, com as arestas invisíveis eliminadas em tempo real.

6.2 Força Bruta

O primeiro algoritmo de eliminação de arestas escondidas pode ser esquematizado como segue:

```
Algoritmo ForçaBruta;
  Leia Ė, F⃗ e a descrição do ambiente;
  Coloque os objetos do ambiente na coordenada do observador;
  Efetue a transformação em perspectiva, colocando os objetos na coordenada U³;
  Sejam F o conjunto de todas as faces e E o conjunto de todas as arestas;
  Para cada face f ∈ F faça
    Para cada aresta e ∈ E faça
      Se f esconde totalmente e então E := E − {e};
      Se f não esconde e então não faça nada;
      Se f esconde parcialmente e então
        E := (E − {e}) ∪ {pedaços de e não escondidos por f};
    FimPara;
  FimPara;
FimAlgoritmo;
```

O conjunto E contém, no início do programa, todas as arestas do ambiente. O algoritmo consiste em eliminar do conjunto E todas as arestas invisíveis. Para isso, testa-se cada uma das faces contra todas as arestas. Se uma face f esconde totalmente uma aresta e, a aresta e pode ser eliminada do conjunto E. Se a face f esconde parcialmente e então a aresta e é substituída pelas suas partes não escondidas por f. No final do processo, teremos dentro do conjunto E somente as partes visíveis das arestas. Observe que se trabalhássemos na coordenada do observador, o cálculo de intersecção entre uma face e uma aresta tornar-se-ia muito mais complexo. Por isso, trabalhamos na coordenada da tela em ponto flutuante U^3. [ProgPrinc] descreve um programa de eliminação de arestas escondidas tipo "força bruta" que funciona na coordenada do observador. Este programa tornar-se-ia muito mais simples se efetuasse uma transformação em perspectiva e calculasse as intersecções no sistema U^3.

Vamos analisar a complexidade do algoritmo de força bruta. Para cada face f_i, gasta-se um tempo $O(m_i n_a)$, onde m_i é o número de arestas da face f_i e n_a é o número total de arestas do ambiente. Somando-se os tempos gastos por cada uma das faces, temos $\sum_{i=1}^{n_f} m_i\, n_a \; = n_a \sum_{i=1}^{n_f} m_i$, onde n_f é o número total de faces do ambiente. Lembrando que numa aresta incidem sempre duas faces, $\sum_{i=1}^{n_f} m_i \; = \; 2\; n_a$ e a complexidade total do algoritmo é $O(n_a^2)$.

Este algoritmo pode ser melhorado fazendo uma subdivisão da tela em janelas menores e criando as listas de candidatos. Para cada janela, cria-se uma lista de arestas e de faces que lhe pertencem

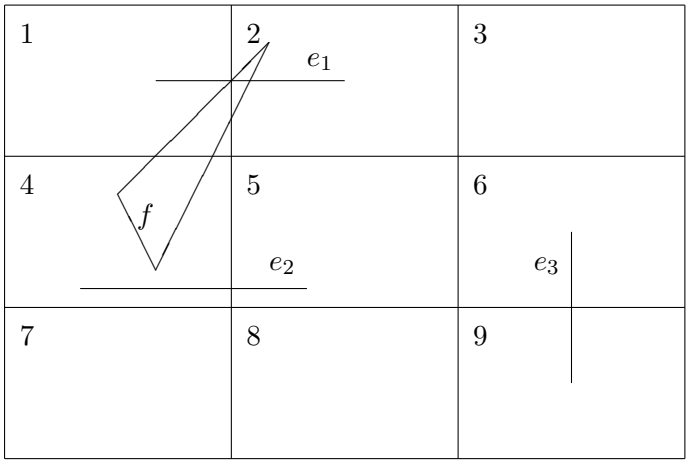


Figure 6.4: Aceleração do algoritmo força bruta através da subdivisão de tela

(completa ou parcialmente). Uma face f só precisa ser testada contra aquelas arestas que pertencem a uma das janelas que também contém f. Veja a figura 6.4. A face f pertence às listas das janelas 1, 2 e 4. Logo, ela não precisa ser testada contra e_3 que pertence às listas 6 e 9 mas precisa ser testada contra as arestas e_1 e e_2. O melhoramento de tempo é muito grande se os comprimentos das arestas são pequenos. Também pode-se criar a subdivisão da tela não-uniforme utilizando quadtree ou a árvore BSP (veja a subseção 11.2.2). Com isso, as janelas serão menores naquelas regiões onde há muitos objetos e serão maiores nas regiões pouco povoados.

Um programa de eliminação de arestas escondidas baseado na técnica de força bruta, escrito em linguagem C, pode ser encontrado em [ProgPrinc]. Este programa foi traduzido para Pascal pelo aluno Fábio F. Domingues como parte do programa de iniciação científica e o resultado pode ser visto na figura 6.5. Para imprimir a saída, foram utilizadas as rotinas gráficas para impressoras matriciais desenvolvidas pelo aluno Fábio A. R. Corrêa que também participou da iniciação científica.

6.3 Invisibilidade Quantitativa

Um algoritmo mais rápido que o algoritmo de força bruta pode ser obtido utilizando o conceito de invisibilidade quantitativa. A invisibilidade quantitativa pode ser definida como segue:

Definição 6.5 *Um ponto $\dot{v}$ do espaço tem invisibilidade quantitativa n se existem n polígonos entre o observador e $\dot{v}$.*

Como consequência imediata da definição acima, um ponto $\dot{v}$ é visível se, e somente se, a sua invisibilidade quantitativa é zero. A invisibilidade quantitativa de uma aresta e pode mudar somente onde a

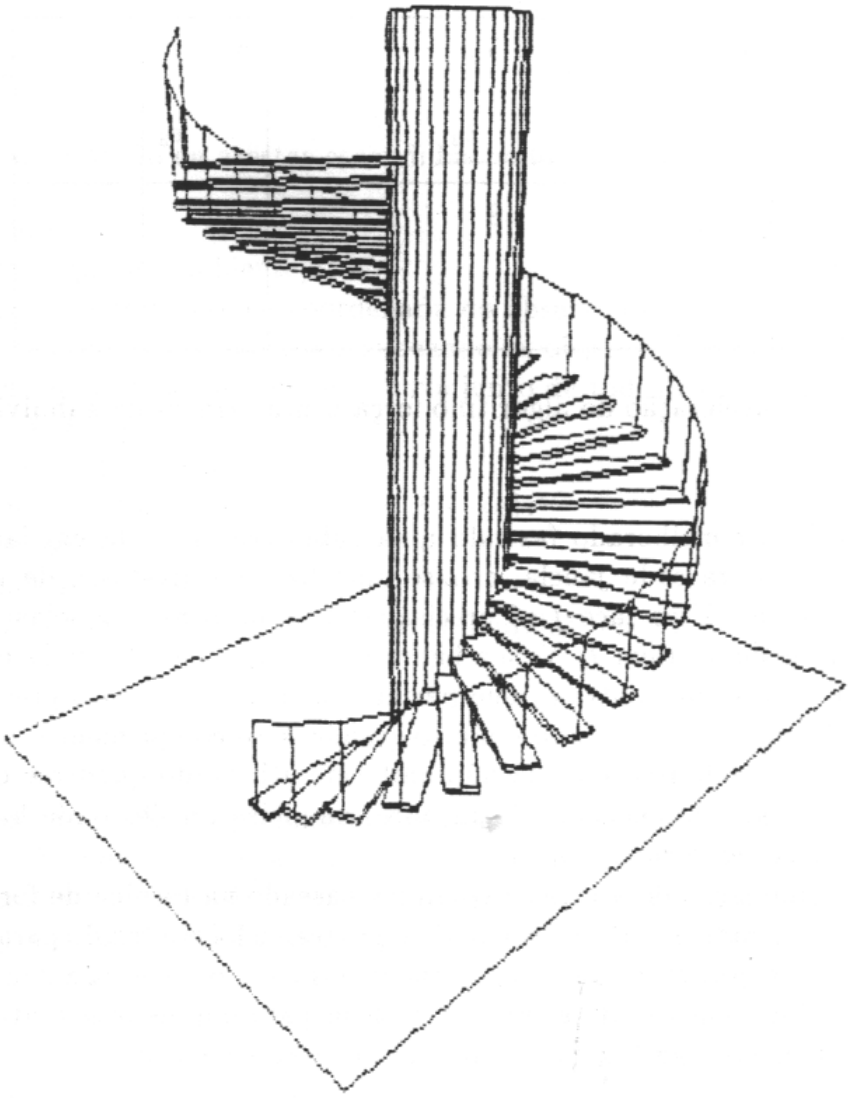


Figure 6.5: Imagem gerada pelo algoritmo “Força Bruta”

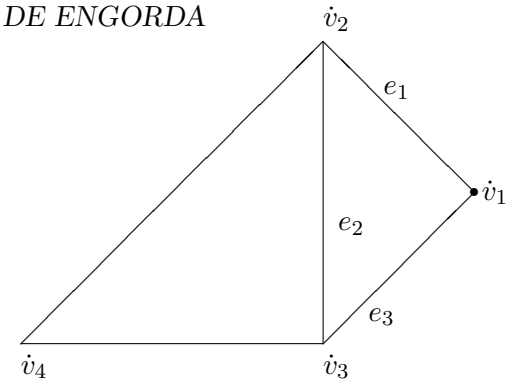


Figure 6.6: Correção de invisibilidade do vértice inicial.

projeção de e cruza uma outra aresta. Neste cruzamento, a invisibilidade quantitativa pode aumentar ou diminuir de um.

Calculando a intersecção de e com todas as outras arestas, a aresta e fica dividida, pelos pontos de intersecção, em vários segmentos. Se a invisibilidade quantitativa do vértice inicial é conhecida, a invisibilidade de cada um dos segmentos pode ser calculada somando as mudanças de invisibilidade que somente acontecem nas extremidades dos segmentos. [Sutherland, 74] sugere que se calcule a invisibilidade do vértice inicial pela busca exaustiva de todas as faces relevantes para contar quantas faces escondem o vértice inicial. O cálculo da invisibilidade inicial e o cálculo dos incrementos determinam a invisibilidade quantitativa do vértice final que pode ser usada, por sua vez, para determinar a invisibilidade inicial de outras arestas que emanam do vértice final. Uma busca exaustiva deve ser feita para cada conjunto de arestas de um mesmo sólido. Mais tarde, veremos como evitar esta busca.

Às vezes, uma correção deve ser aplicada à invisibilidade quantitativa do vértice final para determinar a invisibilidade das arestas que emanam daquele vértice. Veja a figura 6.6. A invisibilidade quantitativa do vértice $\dot{v}_1$ deve ser zero quando percorre e_1 e e_3 e deve ser um quando percorre e_2. Estas correções causam um grande aborrecimento para o programador. [Sutherland, 74] escreve: *Uma série de singularidades desagradáveis podem ocorrer que requerem uma atenção cuidadosa para calcular a invisibilidade corretamente. Loutrel* (que implementou este algoritmo) *comentou-nos: "Todos eles tiveram de ser resolvidos no programa e isso não foi nenhum pic-nic".* Vejamos agora como resolver estas singularidades de modo simples e elegante, utilizando o algoritmo de engorda.

6.4 Algoritmo de engorda

[Sutherland, 74] dá uma idéia sobre os algoritmos de eliminação de arestas escondidas existentes naquele tempo. De lá para cá, estes algoritmos não tiveram um avanço significativo, como podemos supor pela bibliografia do artigo [Kamada, 87], que não cita nenhum algoritmo de eliminação de arestas escondidas desenvolvida após o artigo de Sutherland. Utilizando o método de perturbação e a invisibilidade quan-

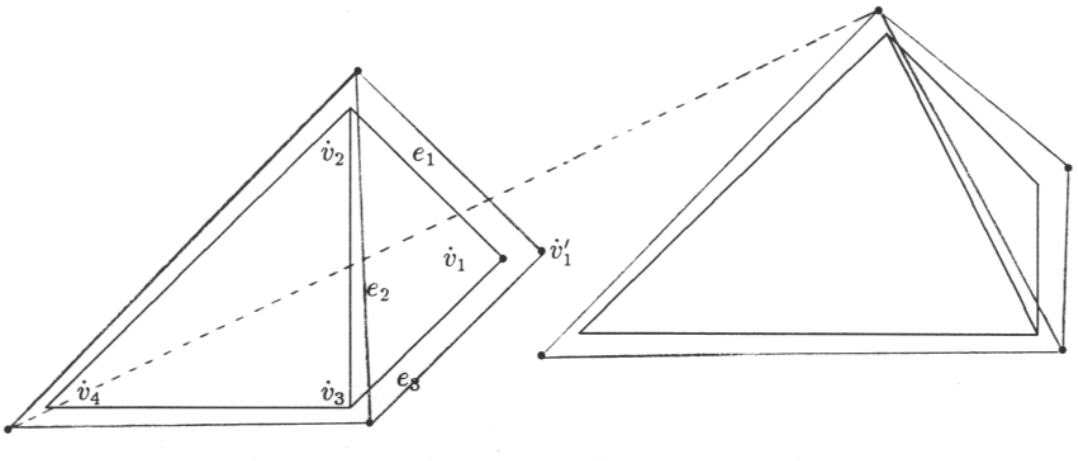


Figure 6.7: Tetraedros "engordados" e aresta fantasma (tracejada).

titativa, conseguimos criar um movo algoritmo de eliminação de arestas escondidas. Neste algoritmo, primeiro devemos "engordar" os sólidos, criando os objetos "engordados". Engordar é uma maneira particular de perturbar os vértices. Imagine que as faces dos objetos são as originais mas as arestas e os vértices dos objetos são os engordados. Então, todos os vértices têm uma invisibilidade quantitativa definida. Por exemplo, a invisibilidade de vértice $\dot{v}_1$ do tetraedro da figura 6.7 era indefinida. Engordando-o obtemos o vértice $\dot{v}_1'$ com a invisibilidade definida e igual a zero. O método para engordar os objetos não é simples, pois não basta efetuar uma mudança de escala. Veja a figura 6.8. Se efetuarmos uma mudança de escala, o espaço entre as duas pernas do objeto irá aumentar e não gerará a perturbação desejada. Quando se engorda um objeto, o espaço vazio entre as duas pernas deve diminuir. Antes de descrevermos o algoritmo, vamos definir o vetor cume de um vértice, que será utilizado no processo de engorda.

Definição 6.6 *Seja $\dot{v}$ um vértice onde incidem as faces $f_1, f_2, \ldots, f_k$ que possuem respectivamente os normais externos $\vec{N}_1, \vec{N}_2, \ldots, \vec{N}_k$. Um vetor unitário $\vec{C}$ é um* **vetor cume** *do vértice $\dot{v}$ se $\vec{C} \circ \vec{N}_i > 0$, $\forall\, i \in 1 \ldots k$.*

De um modo informal, para calcular o vetor cume $\vec{C}$ de um vértice $\dot{v}$ calcule primeiro um plano L que toca o vértice $\dot{v}$ mas que não penetra em nenhuma das faces que incidem em $\dot{v}$. O vetor cume $\vec{C}$ é o vetor normal a L que aponta para longe do sólido. Veja a figura 6.11. Uma característica do vetor cume $\vec{C}$ é o seguinte: Seja $\dot{v}_1' = \dot{v} + \varepsilon\vec{C}$ o vértice $\dot{v}$ perturbado. Então, um observador que está na posição $\dot{v}_1'$ "enxerga" todas as faces que incidem em $\dot{v}$. Também definamos:

Definição 6.7 *Um vetor cume $\vec{B}$ de um vértice $\dot{v}$ é* **o melhor** *se para qualquer vetor cume $\vec{C}$ de $\dot{v}$ diferente de $\vec{B}$, temos: $MIN_{i\in 1\ldots k}\{\vec{C} \circ \vec{N}_i\} \leq MIN_{i\in 1\ldots k}\{\vec{B} \circ \vec{N}_i\}$.*

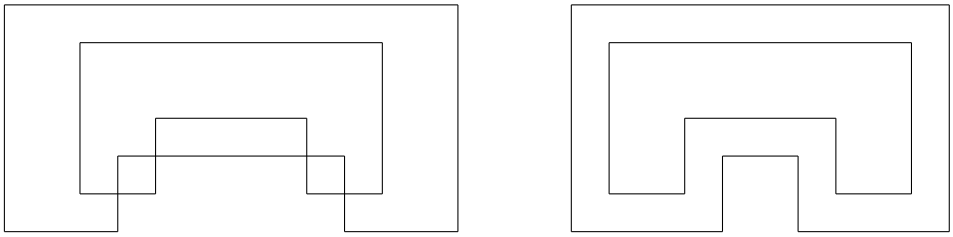


Figure 6.8: Diferença entre ampliar e engordar

Infelizmente, não conseguimos achar um método eficiente de calcular o melhor vetor cume. Deixamos este problema em aberto. Mas para o nosso algoritmo funcionar, basta um vetor cume $\vec{C}$ qualquer, desde que

$$\text{MIN}_{i \in 1 \ldots k}\{\vec{B} \circ \vec{N}_i\} > 0$$

E um vetor cume pode ser calculado da seguinte maneira:

$\vec{C} := versor(\sum_{i=1}^{k} \vec{N}_i)$
Ache i tal que $\vec{C} \circ \vec{N}_i$ seja mínimo.
Enquanto $\vec{C} \circ \vec{N}_i \leq 0$ faça:
    $\vec{C} := versor(\vec{C} + \vec{N}_i)$
    Ache i tal que $\vec{C} \circ \vec{N}_i$ seja mínimo.
FimEnquanto

Vejamos agora o algoritmo em si. O usuário deve fornecer ao algoritmo a posição do observador $\dot{E}$, a direção de visão $\vec{F}$, as coordenadas (x_i, y_i, z_i) para cada vértice $\dot{v}_i$ e, para cada face, a sequência de vértices na ordem anti-horária quando vista do lado de fora do sólido (veja as figuras 6.9 e 6.6). Todas as faces devem pertencer a algum sólido e não pode haver interpenetração entre dois sólidos.

1. Leia a lista de vértices, a lista de faces, a posição do observador $\dot{E}$ e a direção de visão do observador $\vec{F}$.

2. Efetue uma transformação geométrica, colocando todos os vértices na coordenada do observador. Depois, efetue uma transformação em perspectiva, não esquecendo de calcular a profundidade em perspectiva.

vértice	coordenadas	face	vértices
$\dot{v}_1$	x_1, y_1, z_1	f_1	$\dot{v}_2, \dot{v}_4, \dot{v}_3$
$\dot{v}_2$	x_2, y_2, z_2	f_2	$\dot{v}_1, \dot{v}_3, \dot{v}_4$
$\dot{v}_3$	x_3, y_3, z_3	f_3	$\dot{v}_1, \dot{v}_4, \dot{v}_2$
$\dot{v}_4$	x_4, y_4, z_4	f_4	$\dot{v}_1, \dot{v}_2, \dot{v}_3$

Figure 6.9: Coordenadas dos vértices e vértices que incidem numa face (sólido da figura 6.6).

vértice	coordenadas	vetor cume	face	normal
$\dot{v}_1$	f_2, f_3, f_4	$\vec{C}_1$	f_1	$\vec{N}_1$
$\dot{v}_2$	f_1, f_3, f_4	$\vec{C}_2$	f_2	$\vec{N}_2$
$\dot{v}_3$	f_1, f_2, f_4	$\vec{C}_3$	f_3	$\vec{N}_3$
$\dot{v}_4$	f_1, f_2, f_3	$\vec{C}_4$	f_4	$\vec{N}_4$

Figure 6.10: Faces que incidem num vértice, vetores cume e normais das faces.

3. Gere a lista de faces que incidem num vértice a partir da lista de vértices que incidem numa face (figura 6.10).

4. Para cada face, calcule o vetor normal unitário. Esta operação está descrita no artigo [Sutherland, 74] e na seção 10.3.

5. Calcule o vetor cume para cada vértice (figura 6.10).

6. Gere os vértices perturbados $\dot{v}'_i = \varepsilon \vec{C}_i + \dot{v}_i$, *para* $i = 1 \ldots n$. Os vértices perturbados pertencem aos poliedros "engordados", que cobrem os sólidos originais.

7. O algoritmo não funciona se houver uma reincidência na singularidade. Pode-se imaginar que no problema original todos os vértices feram singulares. Após a engorda, os vértices originais não devem passar por cima das arestas perturbadas e vértices perturbados não devem passar por cima das arestas originais, pois isto seria uma reincidência na singularidade. Não precisamos nos preocupar com as reincidências α, pois é quase impossível de acontecer, principalmente se são utilizadas as variáveis dupla precisão. Quanto à reincidência β, precisamos tomar mais cuidado, pois há maior probabilidade de ocorrer. Se escolhermos ε cuidadosamente, como sugere a equação 5.3, podemos deixar a probabilidade de reincidência β em níveis aceitáveis. Se o programa necessita de uma segurança maior, pode-se testar explicitamente a reincidência β e escolher um outro vetor cume no caso da reincidência β.

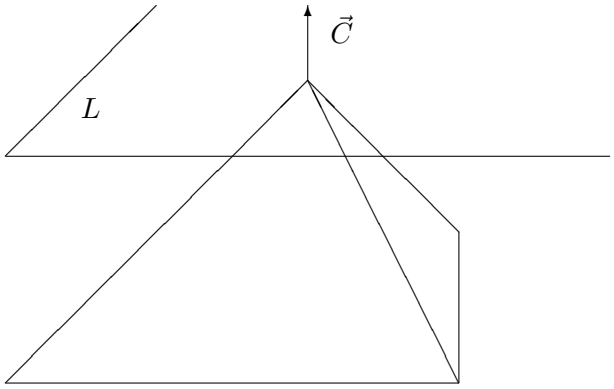


Figure 6.11: Vetor cume

8. Elimine todas as arestas e faces irrelevantes, tanto originais como perturbados. Estamos desconfiados de que esta eliminação poderia ter sido feita logo após o parágrafo 2 e ter calculado o vetor cume utilizando somente as faces relevantes, mas não temos certeza de que isto funciona. Depois de eliminar as faces irrelevantes, todas as faces restantes são relevantes e estão em sentido anti-horário se projetados no plano do observador.

9. Seja $\dot{v}$ o vértice original com a menor coordenada x. O vértice $\dot{v}'$, ou seja, $\dot{v}$ perturbado, tem a coordenada x menor que $\dot{v}$. Logo, a invisibilidade quantitativa de $\dot{v}'$ é zero. Acrescente uma "aresta fantasma" de $\dot{v}'$ a cada um dos sólidos perturbados (veja a figura 6.7). Com este truque, eliminamos a necessidade de fazer uma busca exaustiva para cada sólido, citada na seção 6.3.

10. Ache todas as intersecções entre as arestas perturbadas (inclusive as arestas fantasmas, que devem ser consideradas como arestas perturbadas) e as arestas originais. Este algoritmo está descrito em [Algorithms, pp 345-359]. Para cada aresta perturbada e', ordene as arestas originais que ela intercepta na ordem em que e' será percorrida no próximo parágrafo.

11. Calcule a invisibilidade quantitativa para cada intervalo de arestas perturbadas (inclusive as arestas fantasmas). A invisibilidade quantitativa é zero em $\dot{v}'$ porque nós escolhemos $\dot{v}$ com a menor coordenada x. Some 1 (subtraia 1) toda vez que uma aresta perturbada cruzar uma aresta original entrando num (saindo de um) polígono. Podemos distinguir se a aresta perturbada está entrando ou saindo do polígono calculando um produto vetorial (lembre-se: todos os polígonos estão em ordem anti-horária). Soma (subtração) não é aplicada se a aresta perturbada que estamos percorrendo estiver mais próxima

ao observador do que a aresta original interceptada por aquela. Podemos testar isto calculando duas interpolações lineares de profundidades em perspectiva, mencionadas no parágrafo 2 deste algoritmo.

12. Os intervalos de arestas perturbadas (excluindo as arestas fantasmas) com a invisibilidade quantitativa zero são visíveis. Trace-os na tela do computador.

Análise de complexidade: Sejam n_a o número de arestas e i o número de intersecções entre as arestas perturbadas e as arestas originais.

- Os parágrafos 1, 2, 3, 4, 6, 8 e 9 têm complexidade $O(n_a)$.
- Os parágrafos 5 e 7 normalmente têm complexidade $O(n_a)$, salvo em raras exceções.
- O parágrafo 10 tem complexidade $O(n_a \log n_a + i \log i)$.
- Os parágrafos 11 e 12 têm complexidade $O(n_a + i)$.

Portanto, o algoritmo todo tem complexidade $O(n_a \log n_a + i \log i)$. Se supomos que i é da ordem de n_a, como costuma acontecer na prática, o algoritmo é $O(n_a \log n_a)$. Mas no pior caso, i pode ser da ordem de n_a^2.

6.5 Horizonte Flutuante

Quando queremos achar uma solução mais restrita para o problema de eliminação de arestas escondidas podem aparecer algoritmos inesperadamente eficientes. Vimos um exemplo disso na seção 6.1 quando no ambiente havia apenas poliedros convexos que não escondiam um a outro. Quando o objeto a ser mostrado é uma função de duas variáveis $z = f(x, y)$ então temos um algoritmo especial, eficiente e largamente utilizado: o algoritmo de horizonte flutuante, publicado em [Wright, 73].

Se quiséssemos mostrar uma função $z = f(x, y)$, para $x_i \leq x \leq x_f$ e $y_i \leq y \leq y_f$, traçando n_x curvas com os valores x constantes, sem eliminar as arestas escondidas, faríamos:

```
x:=xi; y:=yi; dx:=(xf-xi)/(nx-1); dy:=(yf-yi)/(ny-1);
for i:=0 to nx-1 do begin
 z:=f(x,y); move(x,y,z);
 for j:=1 to ny-1 do begin
  y:=y+dy; z:=f(x,y); traca(x,y,z);
 end;
 x:=x+dx; y:=yi;
end;
```

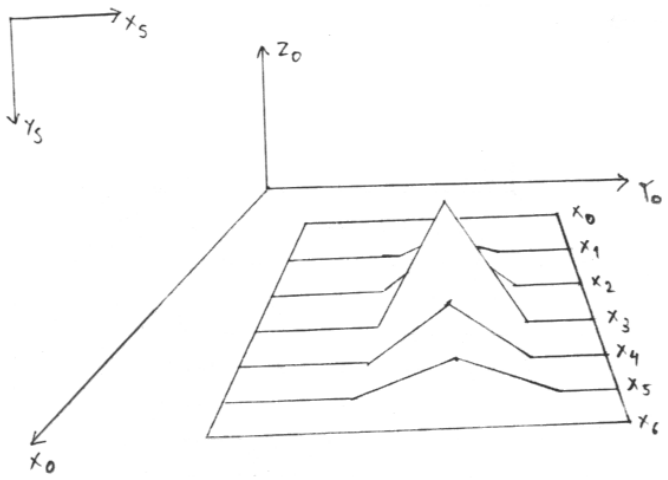


Figure 6.12: Horizonte flutuante: necessidade de vetor "alto"

Vamos chamar as curvas assim traçadas de $X_0, X_1, \ldots, X_{n-1}$, onde a curva X_i tem a coordenada x constante que vale x_i. O procedimento $move(x, y, z)$ converte o ponto (x, y, z) para sistema S e desloca uma caneta imaginária para este ponto. O procedimento $traça(x, y, z)$ converte o ponto (x, y, z) para o sistema S e traça um segmento de reta da última posição de caneta até a nova.

Resta explicar como podemos eliminar as arestas escondidas. Veja a figura 6.12. Para eliminar as arestas escondidas devemos desenhar as curvas na ordem $X_{n-1}, X_{n-2}, ..., X_0$. Antes de acender cada pixel (x_s, y_s), um teste é feito para verificar se o pixel é invisível. Se o valor y_s é menor que qualquer pixel anterior aceso na coluna x_s, então (x_s, y_s) é visível ao observador. Utilize um vetor "alto" para fazer este teste, que em Pascal seria declarado:

```
var alto: array [0..ResX] of integer;
```

Onde a constante $ResX$ é a resolução horizontal da tela. A variável $alto[x_s]$ armazena o menor valor de y_s aceso na coluna x_s. Não se esqueça de que no sistema S, a coordenada y cresce de cima para baixo. Do mesmo modo, é necessário ter um vetor "baixo" que indique o maior valor de y_s aceso na coluna x_s (veja a figura 6.13). Concluindo, para verificar se um ponto (x_s, y_s) é visível, teste:

```
if (ys<alto[xs]) or (baixo[xs]<ys) then visivel else invisivel;
```

Mas, e se o observador estivesse numa outra posição qualquer, de modo que o gráfico final aparecesse rotacionado? A ordem $X_{n-1}, X_{n-2}, ..., X_0$ não valeria mais. Num caso geral, deve-se traçar primeiro as

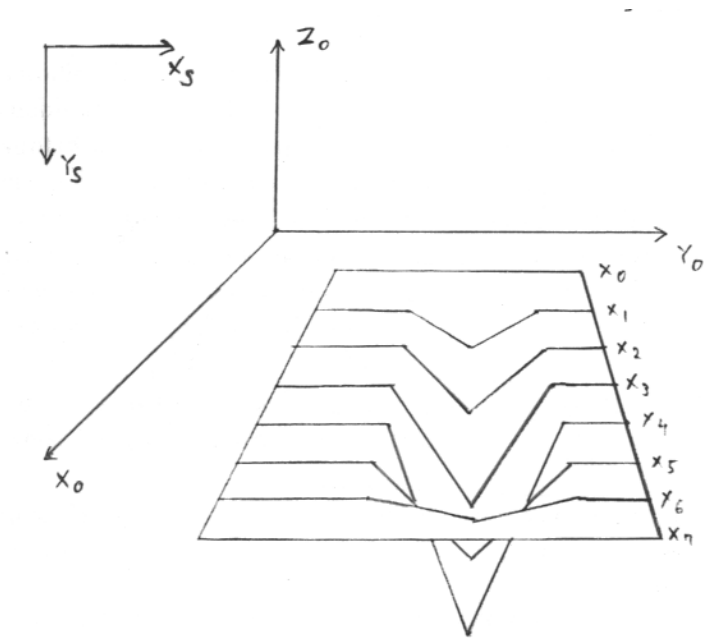


Figure 6.13: Horizonte flutuante: necessidade de vetor "baixo"

curvas mais próximas ao observador (veja a figura 6.14). Mas como podemos determinar qual é a curva mais próxima ao observador? Faça o seguinte cálculo:

```
dx:=(xf-xi)/(nx-1);
inicx:=round( (E.x-xi)/dx ); {E.x e' a coordenada x do observador}
if inicx<0 then inicx:=0;
if inicx>nx-1 then inicx:=nx-1;
```

Inicx assim calculado fornece o índice da curva que deve ser desenhada em primeiro lugar. Deve-se desenhar as curvas mais próximas de *inicx* antes de desenhar as mais afastadas. Uma possível ordem é desenhar as curvas de *inicx* até $nx-1$ em primeiro lugar e depois de $inicx-1$ até 0.

Para traçar as curvas X e as curvas Y não basta repetir o processo visto acima duas vezes (veja figura 6.15). Existe uma ordem especial em que as curvas devem ser desenhadas. Para determinar esta ordem, calcule primeiro quais curvas são mais horizontais: as curvas X ou Y. Digamos que as curvas X sejam mais horizontais. As curvas X devem ser processadas na ordem vista anteriormente. Mas, antes de traçar cada curva X_i, as partes das curvas Y entre a curva X traçada por último e X_i devem ser traçadas. Estas partes das curvas Y também devem obedecer à mesma ordem vista anteriormente, isto é, trace primeiro as curvas Y de *inicy* até $ny-1$ e depois trace as de $inicy-1$ até 0. Veja a figura 6.15.

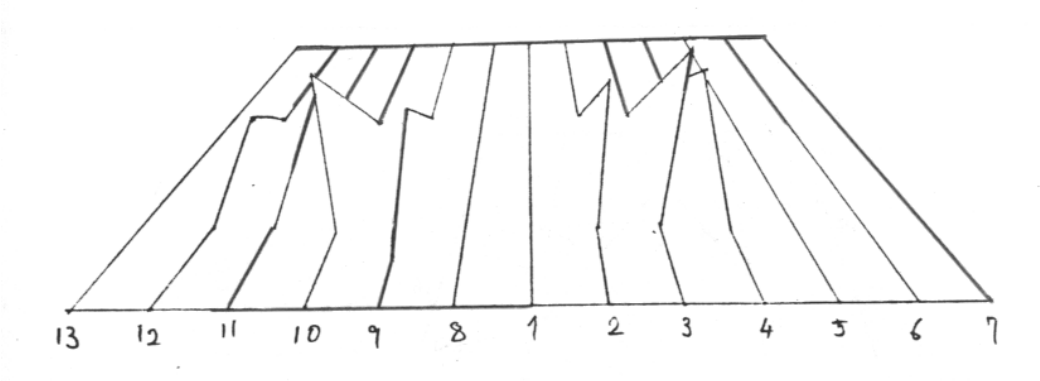


Figure 6.14: Horizonte flutuante: ordem das curvas X

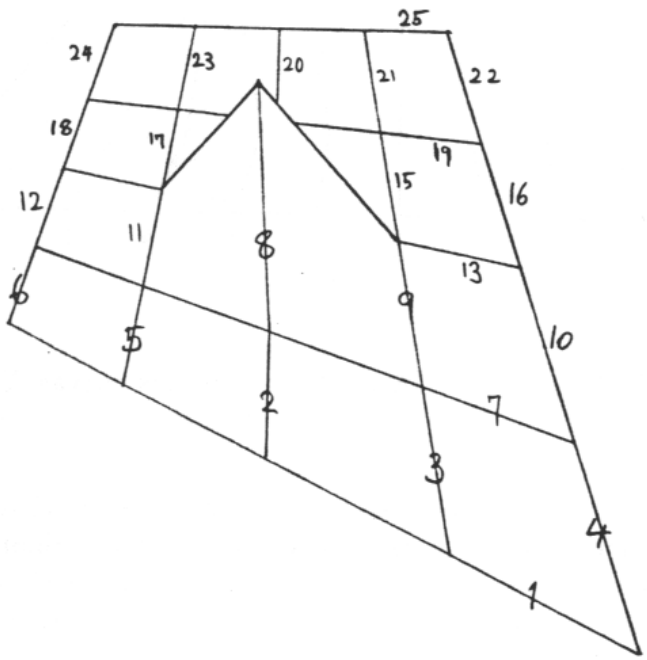


Figure 6.15: Horizonte flutuante: curvas X e Y

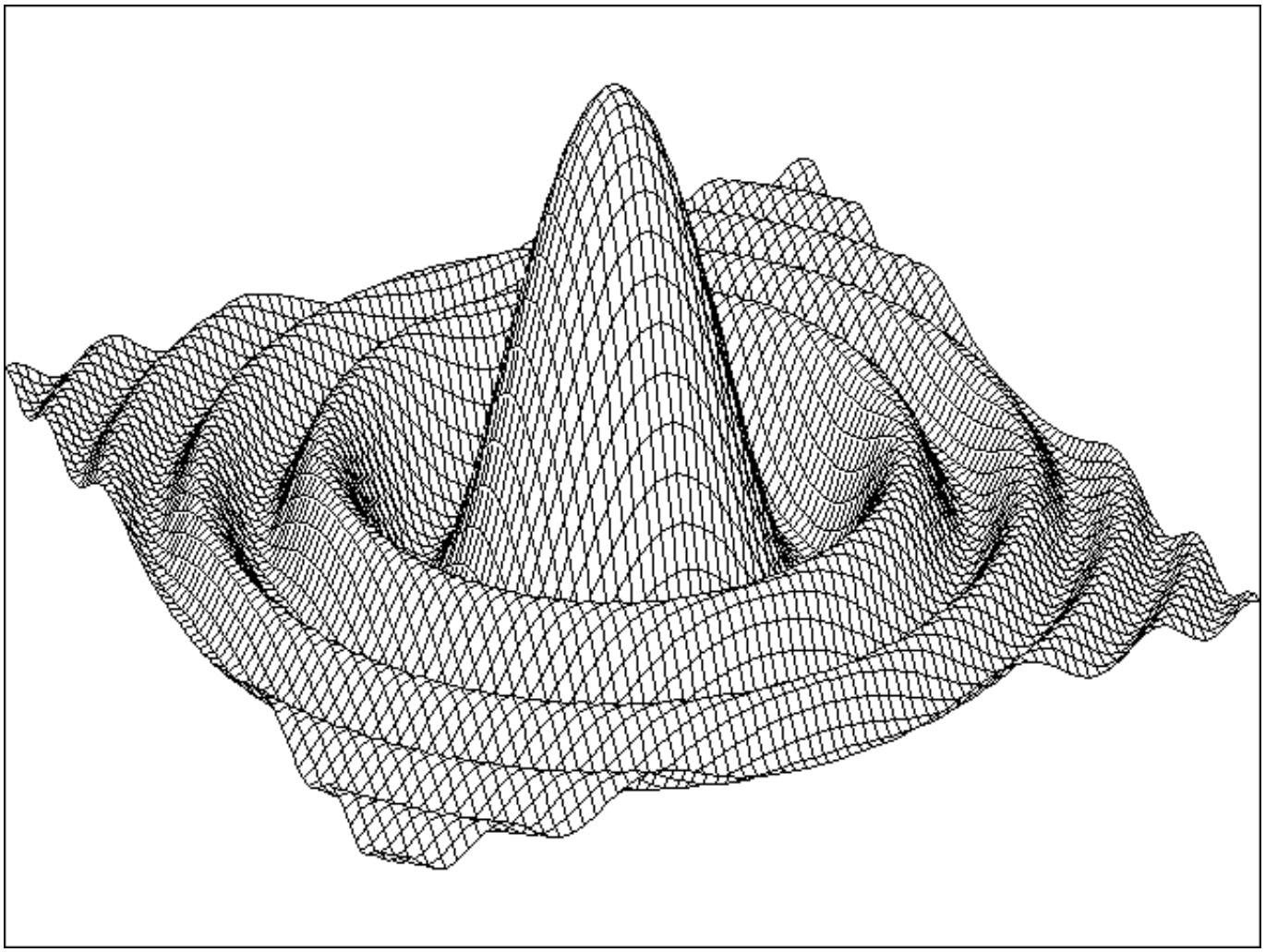


Figure 6.16: Imagem gerada pelo algoritmo horizonte flutuante

Este algoritmo não funciona se o efeito perspectiva é muito acentuado. Isto acontece quando o observador está muito próximo ao objeto. Sugerimos que se coloque sempre o observador a uma distância razoável do objeto para diminuir o efeito perspectiva ou então que se utilize a projeção ortogonal. O algoritmo de horizonte flutuante foi implementado pelo autor e o resultado pode ser visto na figura 6.16.

Chapter 7

Eliminação de Superfícies Escondidas

Vimos no capítulo anterior algumas técnicas para criar imagens armação de arame. Uma imagem mais elaborada pode ser conseguida "pintando" as faces. Também costuma-se utilizar o termo "sombrear[1]". Para se conseguir uma imagem sombreada são essenciais um modelo de iluminação e um algoritmo de eliminação de superfícies escondidas.

7.1 Modelo de Iluminação de Phong Local

Definição 7.1 *Um* **modelo de iluminação** *é uma fórmula utilizada para calcular o espectro luminoso de um pixel da imagem.*

Definição 7.2 *O* **espectro luminoso** *é uma função $f(\lambda)$ que devolve a intensidade de luz para cada comprimento de onda visível λ. O espectro também pode ser descrito fornecendo a intensidade das três cores primárias: vermelha, verde e azul.*

Um modelo de iluminação pode ser local ou global.

Definição 7.3 *Um* **modelo de iluminação local** *considera somente a luz incidente diretamente da fonte, a orientação da superfície, as características da superfície e a posição do observador para determinar o espectro da luz refletida para o observador.*

Definição 7.4 *Um* **modelo de iluminação global** *considera, além dos fatores acima mencionados, a luz que chega ao ponto pela reflexão ou pela transmissão a partir de outros objetos do ambiente.*

Definição 7.5 *As* **características da superfície** *são um conjunto de parâmetros que determinam o aspecto visual da superfície. Nelas estão incluídas a cor, o grau de polimento e o grau de transparência do objeto.*

[1] Shading ou rendering em inglês.

Nesta seção, veremos apenas um modelo de iluminação local: o modelo de Phong. Os modelos de iluminação global serão apresentados na parte de rastreamento de raio. Supondo que no ambiente não existem objetos transparentes, só precisamos considerar as reflexões. Normalmente, num modelo de iluminação local, três termos contribuem para a intensidade luminosa de um ponto: reflexão especular, reflexão difusa e iluminação ambiente.

7.1.1 Reflexão especular

A luz refletida especularmente pode ser considerada como se refletisse nas camadas superficiais do objeto. Por exemplo, ao encerarmos um carro, ele torna-se brilhante. Este brilho é devido aos raios de luz que se refletem especularmente na camada de cera. A cor do carro é determinada pela reflexão difusa da lataria, pois normalmente, a cor da reflexão especular independe da cor do objeto. Numa reflexão especular, todos os comprimentos de onda são refletidos igualmente (ou quase igualmente). Por exemplo, um objeto azul refletido numa bola de Natal amarela continua sendo azul, apesar de estar misturado com a cor amarela proveniente da reflexão difusa da bola de Natal. Outro exemplo: o brilho do Sol refletido na lataria de um carro vermelho é amarelo. A seguinte equação descreve o espectro da reflexão especular:

$$I_s(\lambda) = K_s(\lambda, \theta) L(\lambda) (\vec{N} \circ \vec{L'})^e \tag{7.1}$$

onde (veja a figura 7.1),

- $I_s(\lambda)$ é a intensidade de luz refletida especularmente de comprimento λ.
- $\vec{N}$ é o vetor unitário normal à superfície.
- $\vec{L'}$ é o vetor unitário que indica a direção intermediária entre a direção da fonte de luz $\vec{L}$ e do observador $\vec{I}$. Ou seja, $\vec{L'} = versor(\vec{L} + \vec{I})$.
- θ é o ângulo formado entre $\vec{L}$ e $\vec{L'}$ (ou entre $\vec{I}$ e $\vec{L'}$).
- $K_s(\lambda, \theta)$ é a função que descreve a quantidade de luz refletida (um número entre 0 e 1) quando recebe um raio de luz de comprimento de onda λ num ângulo θ,
- A constante e indica o grau de espalhamento da luz refletida especularmente. Quanto maior for o valor de e, mais direcional será a reflexão especular. O valor de e é da ordem de 60 para superfícies metálicas polidas.
- $L(\lambda)$ é o espectro da fonte de luz. Descreve a intensidade de luz emitida pela fonte para cada comprimento de onda λ.
- A operação $\circ$ é um produto escalar entre dois vetores.

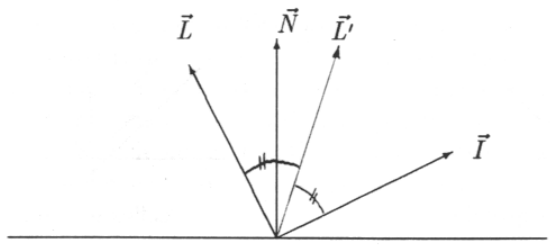


Figure 7.1: Geometria de reflexão da luz de fonte.

Observação 7.1 *Para sermos rigorosos, deveríamos dividir a equação 7.1 por d^2, onde d é a distância entre o ponto ao qual estamos aplicando o modelo de iluminação ($\dot{P}$) e a fonte de luz ($\dot{L}$). Porém, na prática, a equação 7.1 funciona muito bem sem essa divisão, criando imagens inclusive mais "realistas" do que se efetuasse tal divisão. Alguns autores sugerem que efetue a divisão por d ([ProcElem]) e outros sugerem que não se efetue nenhuma divisão ([Whitted, 80]).*

Simplificando a equação acima, para podermos colocá-lo no computador, temos:

$$\bar{I}_s = \bar{K}_s \otimes \bar{L}(\vec{N} \circ \vec{L'})^e \tag{7.2}$$

A barra em cima das letras indica variáveis com três componentes: vermelho (r), verde (g) e azul (b). Chamaremos variáveis deste tipo de COR. As notações utilizadas são:

- O produto $\otimes$ de duas variáveis tipo COR resulta uma COR. Por exemplo, $\bar{p} = \bar{q} \otimes \bar{r}$ significa $p_r = q_r r_r$, $p_g = q_g r_g$ e $p_b = q_b r_b$.
- O produto de uma variável COR por um número real resulta uma COR. Por exemplo, $\bar{p} = \bar{q}r$ significa $p_r = q_r r$, $p_g = q_g r$ e $p_b = q_b r$.

Cada componente de $\bar{K}_s$ deve ser um número entre 0 e 1. Eles são da ordem de 0.9 para objetos muito espelhados e polidos. Num objeto normal, K_{sr}, K_{sg} e K_{sb} são aproximadamente iguais. Se existem mais de uma fonte de luz, digamos k fontes, a equação 7.2 deve ser alterada para:

$$\bar{I}_s = \bar{K}_s \otimes (\sum_{i=1}^{k} \bar{L}_i(\vec{N} \circ \vec{L'_i})^e)$$

7.1.2 Reflexão difusa

A luz refletida difusamente pode ser considerada como tendo penetrado nas camadas mais profundas da superfície do objeto, tendo sido absorvido e então reemitido. Como consequência, a luz refletida

Reflexão especular Reflexão difusa

Figure 7.2: Reflexão especular e difusa.

difusamente é emitida com igual intensidade em todas as direções. Considere um papel. Ele apresenta o mesmo "padrão de brilho", independentemente da posição do observador. Pois um papel apresenta reflexão predominantemente difusa. Já não se pode dizer o mesmo de um metal. Uma pequena variação na posição do observador faz com que apareça um outro padrão de brilho numa folha de alumínio. Isto porque a reflexão predominante é especular.

A cor de um objeto é determinado principalmente pela reflexão difusa. Pois, como já vimos, a reflexão especular costuma refletir igualmente bem luzes de todos os comprimentos de onda e, portanto, não determina a cor do objeto. A reflexão difusa, por outro lado, apresenta um espectro de reflexão que varia de objeto para objeto. O espectro da luz refletida difusamente pode ser descrita pela equação:

$$I_d(\lambda) = K_d(\lambda)L(\lambda)(\vec{N} \circ \vec{L}) \tag{7.3}$$

onde (veja a figura 7.1),

- $I_d(\lambda)$ é a intensidade de luz refletida difusamente de comprimento de onda λ.
- $K_d(\lambda)$ é a curva de reflexão difusa. Assume um valor entre 0 e 1, para cada comprimento λ.
- $L(\lambda)$ é o espectro da fonte de luz.
- $\vec{N}$ é o vetor unitário normal à superfície.
- $\vec{L}$ é o vetor unitário que indica a direção da fonte de luz.

Observe que nesta equação a posição do observador não aparece. Pois a reflexão difusa é independente da posição do observador.

$\vec{N} \circ \vec{L}$ é um produto escalar entre dois vetores unitários e assume o valor um se ambos vetores são idênticos e o valor vai se aproximando de zero à medida que o ângulo formado pelos dois vetores aproxima-se de 90 graus. Na verdade, $\vec{N} \circ \vec{L} = \cos\alpha$, onde α é o ângulo formado pelos dois vetores, o que

indica que a reflexão difusa será máxima se a luz incidir perpendicularmente à superfície e, quanto mais obliquamente a luz incidir, a quantidade de luz refletida difusamente se tornará menor. Simplificando a equação 7.3 para que possa ser implementado num computador, temos:

$$\bar{I}_d = \bar{K}_d \otimes \bar{L}(\vec{N} \circ \vec{L}) \tag{7.4}$$

O valor K_d determina a cor do objeto. Por exemplo,

- $K_{dr} = 0.5$, $K_{dg} = 0.5$ e $K_{db} = 0$ determina um objeto amarelo.
- $K_{dr} = 0.1$, $K_{dg} = 0.1$ e $K_{db} = 0.1$ determina um objeto preto.
- $K_{dr} = 0.9$, $K_{dg} = 0.9$ e $K_{db} = 0.9$ determina um objeto branco.

Se existem k fontes de luz, a equação 7.4 deve ser alterada para:

$$\bar{I}_d = \bar{K}_d \otimes (\sum_{i=1}^{k} \bar{L}_i(\vec{N} \circ \vec{L}_i))$$

7.1.3 Reflexão da luz ambiente

Se no nosso modelo de iluminação houvesse apenas os dois termos vistos nas subseções 7.1.1 e 7.1.2 os objetos que estivessem na sombra apareceriam como completamente negros. Porém, numa cena real, um objeto também recebe luz refletido difusamente de paredes da sala e dos outros objetos à sua volta. Esta luz ambiente representa uma fonte de luz difusa que faz com que mesmo os objetos que estão na sombra recebam alguma luz. O algoritmo de radiosidade consegue calcular corretamente este fenômeno. Num modelo de iluminação simples, porém, a luz ambiente é considerada constante:

$$I_a(\lambda) = K_a(\lambda)L_a(\lambda) \tag{7.5}$$

onde, $K_a(\lambda)$ é a curva de reflexão da luz ambiente que muda de objeto para objeto. Normalmente, $K_a(\lambda)$ e $K_d(\lambda)$ têm curvas muito semelhantes, pois ambas referem-se à reflexão difusa. $L_a(\lambda)$ é o espectro da luz ambiente. Esta curva irá variar de acordo com os objetos que fazem parte do ambiente. Por exemplo, se o meu ambiente está cercado de paredes vermelhas, a curva $L_a(\lambda)$ apresentará um pico no espectro vermelho. Simplificando a equação 7.5, temos:

$$\bar{I}_a = \bar{K}_a \otimes \bar{L}_a$$

7.1.4 Um modelo de iluminação local

Juntando os três termos vistos nas subseções 7.1.1, 7.1.2 e 7.1.3, chegamos a um modelo de iluminação local:

$$I(\lambda) = K_s(\lambda,\theta)L(\lambda)(\vec{N} \circ \vec{L'})^e + K_d(\lambda)L(\lambda)(\vec{N} \circ \vec{L}) + K_a(\lambda)L_a(\lambda)$$

Simplificando, obtemos o modelo de iluminação local amplamente utilizado nos programas de Computação Gráfica:

$$\bar{I} = \bar{K_s} \otimes \bar{L}(\vec{N} \circ \vec{L'})^e + \bar{K_d} \otimes \bar{L}(\vec{N} \circ \vec{L}) + \bar{K_a} \otimes \bar{L_a} \tag{7.6}$$

A observação 7.1 vale para os dois primeiros termos da equação acima. Repare que, neste modelo, a fonte de luz deve ser necessariamente pontual. Se existirem k fontes de luz, a equação 7.6 torna-se:

$$\bar{I} = \bar{K_s} \otimes (\sum_{i=1}^{k} \bar{L_i}(\vec{N} \circ \vec{L_i'})^e) + \bar{K_d} \otimes (\sum_{i=1}^{k} \bar{L_i}(\vec{N} \circ \vec{L_i})) + \bar{K_a} \otimes \bar{L_a}$$

7.2 Lançamento de Raio[2]

Uma vez estudado um modelo de iluminação local, vamos passar aos algoritmos de eliminação de superfícies escondidas. O primeiro deles, e o mais simples, é o algoritmo de lançamento de raio que mais tarde deu origem ao algoritmo de rastreamento de raio. O lançamento de raio não utiliza nenhuma coerência do ambiente para acelerar o processamento. Consequentemente, é mais lento que os outros algoritmos de superfícies escondidas. Mas a facilidade de se implementar o lançamento de raio para os mais variados tipos de objetos tornam-no o único algoritmo de superfícies escondidas possível de se usar em muitos casos. O lançamento de raio consegue trabalhar com qualquer tipo de objeto, desde que seja possível calcular a intersecção desse tipo de objeto com uma semi-reta e exista uma maneira de calcular o vetor normal à superfície. A coerência, que o lançamento de raio não usa, pode ser definida como:

Definição 7.6 *A* **coerência** *é a característica do ambiente (ou da imagem) de ser localmente constante.*

Podemos citar as seguintes coerências:

- Coerência quadro a quadro: dois quadros consecutivos de uma animação são semelhantes.
- Coerência de aresta: se um ponto da aresta é visível, a vizinhança deste ponto tende a ser visível.
- Coerência de linha de rastreio: duas linhas de rastreio vizinhas são semelhantes.
- Coerência de área: dentro de uma pequena área contígua da imagem, a cor tende a ser semelhante.

[2] Ray-casting.

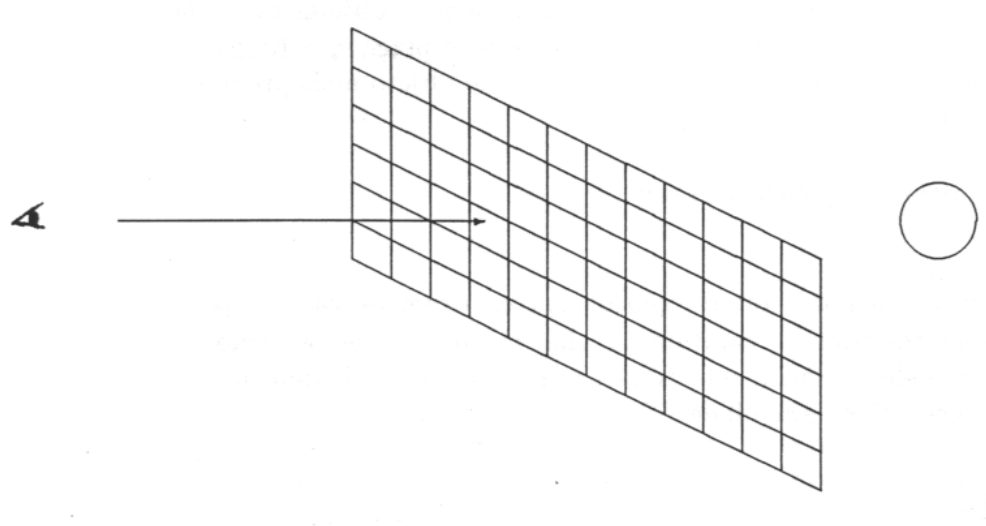


Figure 7.3: Lançamento de raio.

A idéia básica de lançamento de raio (assim como a de rastreamento de raio, como veremos mais adiante) é que o observador enxerga um objeto através dos raios de luz que partem da fonte, batem nesse objeto e chegam ao observador. A luz pode chegar ao observador pela reflexão ou pela refração (também chamada de transmissão). Nesta seção, consideraremos somente a reflexão. Se os raios de luz forem traçados a partir da fonte, muito poucos deles chegarão ao observador. Consequentemente, o processo será computacionalmente ineficiente. [Appel, 68] sugeriu que os raios deveriam ser traçados na direção oposta, isto é, do observador aos objetos. Os raios são lançados do $\dot{E}$ para objeto através de um anteparo imaginário quadriculado. Cada quadrado desse anteparo corresponde a um pixel da tela. Um raio é lançado para o centro de cada quadrado do anteparo. A cor do pixel será calculada aplicando o modelo de iluminação ao objeto mais próximo que esse raio atingir. Observe que, para podermos aplicar o modelo de iluminação, é necessário que saibamos calcular o vetor normal à superfície. Mais tarde, [Whitted, 80] teve a idéia de implementar o lançamento de raio junto com um modelo de iluminação global, obtendo o rastreamento de raio. O lançamento de raio pode ser esquematizado como segue:

Algoritmo lançamento de raio;
  Leia $\dot{E}$, $\vec{F}$, $\dot{L}$, $\bar{L}$ e a descrição do ambiente;
  Coloque os objetos do ambiente na coordenada do observador;
  Para $y = 0$ até $altura - 1$ faça
    Para $x = 0$ até $largura - 1$ faça
      Converta $P_s = (x, y)$ para a coordenada do observador E (chamaremos de P_e);
      Calcule o raio R que parte de $\dot{E}$ e passa pelo ponto P_e;
      Calcule as intersecções do raio R com todos os objetos do ambiente;

Se R não intercepta nenhum objeto então o pixel $P_s = (x, y)$ assume a cor de fundo;
Senão o pixel $P_s = (x, y)$ assume a cor do objeto mais próximo ao ponto $\dot{E}$;
FimPara
FimPara
Fim do Algoritmo lançamento de raio.

Os métodos para calcular a intersecção de um raio com os vários tipos de objetos (plano, esfera, polígono, etc) serão descritos na capítulo 10. Lá, também veremos como calcular o vetor normal. O lançamento de raio pode ser facilmente adaptado para calcular algumas propriedades físicas dos sólidos, como o volume e o centro de gravidade.

7.3 Z-Buffer[4]

Z-buffer é um dos algoritmos de eliminação de superfícies escondidas mais simples, juntamente com o lançamento de raio. A técnica foi proposta originalmente por [Catmull, 75] e, mesmo sendo tão simples, consegue gerar trivialmente as imagens de ambientes muito complexos. Ao mesmo tempo, é bastante rápido. A grande desvantagem deste algoritmo é por exigir um gasto de memória muito grande, da ordem do número de pixels da tela. Como veremos mais adiante, este problema pode ser minimizado dividindo a tela em unidades menores que chamaremos de janelas. Seja $T_{l \times c}$ a tela do computador. Para se executar o algoritmo z-buffer, necessita-se de uma matriz $Z_{l \times c}$ de variáveis em ponto flutuante.

Na execução, o valor z de um novo pixel a ser colocado na tela é comparado com o valor antigo armazenado na matriz Z. Se esta comparação indicar que o novo pixel está à frente do pixel armazenado na tela T então a cor do novo pixel é escrito na tela e o seu valor z é escrito na matriz Z. Caso contrário, nada é feito. Para poder implementar o algoritmo z-buffer, é fundamental conhecer os algoritmos de preenchimento de polígono (veja a seção 5.5). O algoritmo z-buffer para polígonos pode ser esquematizado como segue:

1 Algoritmo z-buffer;
2 Preencha a matriz T com a cor de fundo e a matriz Z com $+\infty$.
3 Leia a descrição do ambiente, $\dot{E}$, $\vec{F}$, $\dot{L}$ e $\bar{L}$;
4 Coloque os objetos na coordenada do observador;
5 Calcule a cor de cada polígono;
6 Coloque os objetos na coordenada da U^3;
7 Elimine as faces irrelevantes;
8 Para todo polígono G do ambiente faça
9 Para cada pixel P de G faça
10 Se $P_z < Z[P_x, P_y]$ então

[4]Buffer significa pára-choque. Portanto, a tradução literal seria pára-choque z.

```
11              Z[P_x, P_y] := P_z;  T[P_x, P_y] := cor do pixel P;
12          FimSe
13      FimPara
14  FimPara
15 Fim do algoritmo z-buffer.
```

7.3.1 Memória necessária

Vamos estimar a memória necessária para rodar z-buffer. Suponha que a tela possui 800×1000 pontos. Se utilizássemos variáveis em ponto flutuante (4 bytes) para armazenar a profundidade z, necessitaríamos de $800 \times 1000 \times 4 = 3200000$ bytes, ou seja, 3 MBytes! Nem sempre dispomos de tanta memória. Uma possível solução para a falta de memória seria armazenar a matriz Z na memória secundária, como o disco magnético. Com isto, o algoritmo tornar-se-ia muito mais lento.

Uma outra solução é dividir a tela em janelas menores e processar uma janela de cada vez. Se nós dividíssemos a tela de 800×1000 pontos em 10 janelas de 80×1000 pontos, a matriz Z passa a ocupar somente 320000 bytes, isto é, 320 KBytes. A desvantagem de dividir a tela em n janelas é que um mesmo objeto pode ser processado mais de uma vez. Isto, evidentemente, torna o algoritmo mais lento. Porém, se o tamanho de cada polígono é pequeno, o aumento de tempo de processamento não será grande. No algoritmo com a tela dividida em janelas, antes de processar z-buffer, cria-se uma lista de polígonos candidatos para cada janela[5]. Depois, cada janela é processada sequencialmente, preenchendo somente os polígonos candidatos. Se o tamanho dos polígonos é grande, é interessante efetuar o recorte (veja a subseção 5.6.2) para determinar exatamente quais partes dos polígonos ficam dentro da janela.

Quando o tamanho das janelas torna-se o menor possível, isto é, quando se reduz a um único pixel, obtemos o algoritmo de lançamento de raio. E quando a janela é uma linha da tela, o algoritmo recebe o nome especial de z-buffer de linha de rastreio[6].

Z-buffer pode ser paralelizado simplesmente fazendo com que cada processador seja responsável por uma janela. Todos os processadores recebem o mesmo programa mas cada processador recebe apenas a lista de polígonos candidatos da janela sob a sua responsabilidade. Com isso, cada processador tem todas as informações necessárias para processar uma janela.

O algoritmo z-buffer foi implementado pelos alunos Evaldo H. de Oliveira, Carlos C. Tanaka e Paulo P. da Silveira S. como projeto do curso Introdução à Computação Gráfica (1990). A imagem gerada por este programa pode ser vista na figura 7.4. A saída do programa foi tratada pelo algoritmo de difusão de erro (veja a seção 8.1) para que pudesse ser mostrada numa tela CGA de um computador PC/XT.

[5] Polígonos cuja intersecção com a janela é não nula.
[6] Scan line z-buffer.

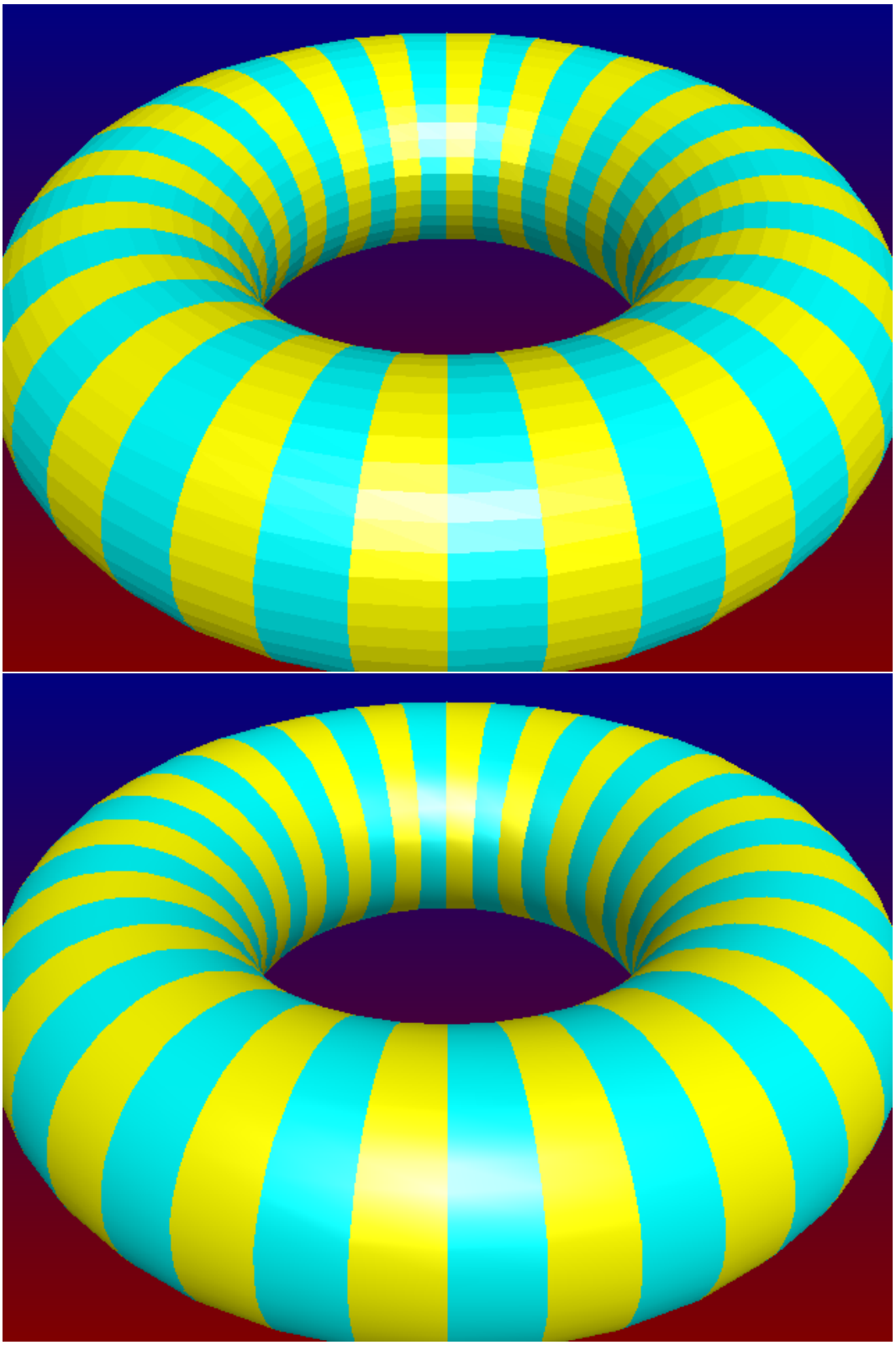


Figure 7.4: Imagens geradas pelo algoritmo z-buffer sem e com suavização de Gouraud.

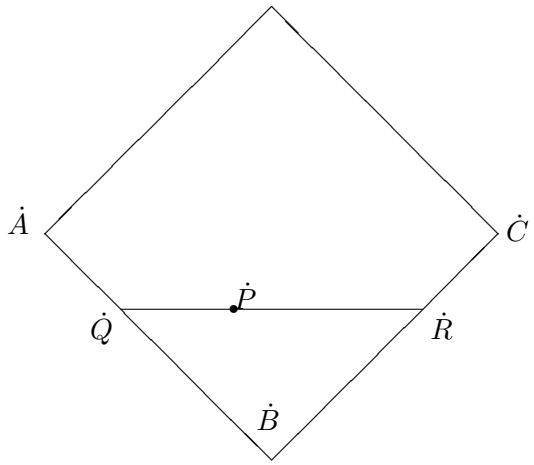


Figure 7.5: Sombreamento de Gouraud e de Phong

7.3.2 O modelo de iluminação no z-buffer

Uma dificuldade para se implementar z-buffer é que o vetor normal e a posição do polígono devem ser calculadas na coordenada do observador para que se possa aplicar corretamente o modelo de iluminação (linha 11 do algoritmo) mas o algoritmo de preenchimento do polígono (linhas 9 a 13) só funciona se o polígono estiver na coordenada da tela. Uma solução simples seria calcular a cor de cada polígono no pré-processamento (linha 5). Assim, cada polígono seria pintado inteiramente com uma única cor. Isto funciona bem quando o tamanho de cada polígono é pequeno. Ultimamente, tem-se utilizado da ordem de 50000 polígonos para representar um objeto, o que torna os polígonos muito pequenos.

Se queremos mostrar polígonos relativamente grandes, devemos aplicar o modelo de iluminação para cada pixel. Isto deve ser feito convertendo o ponto $\dot{P} = (P_x, P_y, P_z)$ da linha 11 para a coordenada do observador antes de se aplicar o modelo de iluminação. Os vetores normais dos polígonos podem ser calculados no pré-processamento (linha 5), pois para cada polígono existe um único vetor normal.

Se o nosso modelo poligonal é uma aproximação de uma superfície curva (por exemplo, o rosto humano aproximado por polígonos) então existem dois métodos de sombreamento que geram as imagens que imitam a superfície original: [Gouraud, 71] e [Phong, 75]. Em ambos os métodos, deve-se calcular o vetor normal em cada um dos vértices do poliedro. Isto pode ser feito tirando a média dos vetores normais das faces que incidem no vértice.

Em [Gouraud, 71], aplica-se o modelo de iluminação em cada vértice no pré-processamento. Com isso, para cada vértice do polígono fica determinada a sua cor. Depois, no preenchimento de polígono (linha 11), calcula-se uma interpolação linear das cores dos vértices. Por exemplo, considere o segmento

$\dot{Q}\dot{R}$ da superfície poligonal da figura 7.5. A intensidade em $\dot{Q}$ é determinada interpolando linearmente a intensidade dos vértices $\dot{A}$ e $\dot{B}$:

$$I_Q = u\, I_B + (1-u)\, I_A, \quad 0 \leq u \leq 1$$

onde $u = AQ/AB$. De modo análogo, as intensidades dos vértices $\dot{B}$ e $\dot{C}$ são interpoladas para se obter a intensidade em $\dot{R}$:

$$I_R = w\, I_B + (1-w)\, I_C, \quad 0 \leq w \leq 1$$

onde $w = CR/CB$. Finalmente, a intensidade em $\dot{P}$ é obtida pela interpolação linear de I_Q e de I_R:

$$I_P = t\, I_R + (1-t)\, I_Q, \quad 0 \leq t \leq 1$$

onde $t = QP/QR$.

Em [Phong, 75], em vez de se interpolar a intensidade luminosa, interpola-se o vetor normal. O modelo de iluminação é então aplicado para cada pixel, utilizando o vetor normal interpolado. Esta técnica dá uma melhor aproximação da curvatura da superfície e, como consequência, gera as imagens mais realistas. Em particular, a reflexão especular da fonte de luz aproxima-se mais da imagem real do que no sombreamento de Gouraud. Utilizando novamente a figura 7.5, o vetor normal em $\dot{Q}$ é determinado interpolando linearmente os vetores normais em $\dot{A}$ e em $\dot{B}$. O vetor normal em $\dot{R}$ é determinado pela interpolação linear dos vetores normais em $\dot{B}$ e $\dot{C}$. Finalmente, o vetor normal em $\dot{P}$ é calculado pela interpolação linear de vetores normais em $\dot{Q}$ e $\dot{R}$:

$$\begin{array}{llll} \vec{n}_Q = u\,\vec{n}_B + (1-u)\vec{n}_A, & 0 \leq u \leq 1, & \text{onde} & u = AQ/AB \\ \vec{n}_R = w\,\vec{n}_B + (1-w)\vec{n}_C, & 0 \leq w \leq 1, & \text{onde} & w = CR/CB \\ \vec{n}_P = t\,\vec{n}_R + (1-t)\vec{n}_Q, & 0 \leq t \leq 1, & \text{onde} & t = QP/QR \end{array}$$

7.3.3 Z-buffer para modelo de partículas

[Norton, 82] utilizou z-buffer para criar as imagens de superfícies fractais modeladas como um conjunto de pontos no espaço (da ordem de um milhão de pontos). As sombras dos objetos são criadas executando z-buffer duas vezes. Na primeira vez, executa-se z-buffer na coordenada da fonte de luz para determinar as sombras e calcular a intensidade luminosa de cada ponto. Suponha que os pontos estão armazenados num arquivo sequencial. Na primeira leitura, determinam-se quais são os pontos que recebem a luz direto da fonte. Na segunda leitura, calculam-se as intensidades luminosas de cada ponto. O vetor normal à superfície em cada ponto é determinado comparando as suas coordenadas com as coordenadas dos pontos vizinhos. Na segunda execução de z-buffer, o arquivo de pontos é lido uma única vez e eliminam-se os pontos escondidos. Esta segunda execução é feita na coordenada da tela U^3. Esta técnica é bastante interessante, pois objetos arbitrariamente complexos podem ser representados por um conjunto de pontos. Isto inclui, além de todos os tipos de superfícies imagináveis, objetos como nuvens, fumaças, etc. Além disso, as sombras dos objetos podem ser calculadas sem dificuldades. O algoritmo do pintor poderia ser utilizado, em vez de z-buffer, para criar as imagens de modelo de partículas.

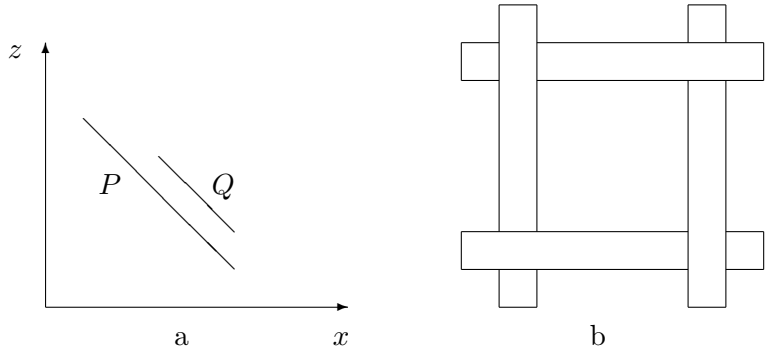


Figure 7.6: Casos em que o algoritmo do pintor falha.

7.4 Algoritmo do pintor[9]

Este algoritmo, também conhecido pelo nome de algoritmo de lista de prioridade, procura ordenar os objetos pelas suas coordenadas z. Depois, os objetos que estão mais longe do observador serão pintados primeiro. Os objetos mais próximos são escritos na tela apagando os objetos previamente pintados. O método recebe o nome de algoritmo do pintor pela analogia à técnica usada pelos artistas para pintar. O artista pinta primeiro o cenário de fundo, depois os objetos que ficam numa distância intermediária e por fim os elementos próximos.

Se os objetos são simples, como no modelo de partículas, uma ordenação pelas coordenadas z basta para calcular a ordem correta em que os objetos devem ser mostrados na tela. Isto pode não ser verdade quando os objetos não são simplesmente pontos. Digamos que queremos mostrar polígonos. Então, primeiro ordene esses polígonos pelas suas coordenadas z mínimas (ou máximas). Normalmente, isto é suficiente para achar a solução correta, mas nem sempre. Veja a figura 7.6a. Se tivéssemos ordenado os polígonos pelas coordenadas z máximas, pintaríamos o polígono P antes de desenhar Q e a imagem gerada seria incorreta. Uma maneira de detectar isto seria testar, depois de ter ordenado os polígonos pelas suas coordenadas z máximas, se os polígonos assim ordenados também estão em ordem crescente de coordenadas z mínimas. Se não estão, então detecte os polígonos problemáticos (que serão como os da figura 7.6a) e aplique testes mais complexos como aqueles descritos em [ProcElem]. Um outro problema que pode aparecer é polígonos que se sobrepõe ciclicamente, como os da figura 7.6b. Felizmente, este caso é difícil de aparecer quando os objetos são muito pequenos como uma superfície curva aproximada por muitos polígonos. [ProcElem] também descreve uma solução para este caso: dividir polígonos cíclicos em outros menores. Observe que esta solução poderia ter sido aplicada também para os polígonos do tipo da figura 7.6a.

[9]Painter's algorithm.

Algumas vezes, conhece-se de antemão que os polígonos problemáticos como os da figura 7.6 não podem aparecer. É o caso de funções tipo $z = f(x, y)$ aproximadas por polígonos. Nós vimos na seção 6.5 que para funções deste tipo existiam algoritmos mais simples para eliminar as arestas escondidas. O mesmo ocorre para a eliminação de superfícies escondidas.

O algoritmo de pintor permite criar objetos transparentes. Estes objetos devem ter paredes finas, para que o desvio dos raios de luz devido ao efeito de refração seja pequeno. Para isso, quando for pintar os objetos transparentes, deve-se tirar a média ponderada entre a cor do objeto transparente e a cor do cenário de fundo armazenada no pixel. Observe que a imagem dos objetos transparentes criada desta forma não obedece à lei de Snell.

7.5 Algoritmo de Subdivisão da Tela

Esta técnica foi apresentada originalmente por [Warnock, 68]. Consiste em dividir a tela em janelas cada vez menores até que cada uma das janelas contenha somente objetos suficientemente simples que podem ser processados facilmente. As implementações específicas deste algoritmo variam no método de subdividir as janelas e nos detalhes de critério utilizados para se decidir quando o conteúdo deve ser considerado suficientemente simples.

Vamos descrever uma implementação assumindo que a tela é quadrada com a largura e a altura de tamanhos iguais a 2^k , para algum inteiro k. Uma janela complexa demais para poder ser mostrada é subdividida em quatro janelas iguais. O processo se repete até chegar nas janelas simples. Se a resolução da tela é 512×512, repetindo-se o processo 9 vezes chegamos à janela de tamanho um ($2^9 = 512$). Este processo pode ser implementado utilizando a recursão ou uma pilha e implicitamente acaba gerando uma estrutura de dados chamada "quadtree". O algoritmo pode ser descrito como:

```
Procedimento SubDiv( x, y, tamanho: inteiro );
  Se tamanho ≤ 1 então
    Se nenhum polígono intersecta o pixel então
      Coloque a cor de fundo no pixel
    Senão
      Coloque a cor do polígono mais próximo no pixel
    FimSe
  Senão
    Se nenhum polígono intersecta a janela então
      Pinte a janela inteira com a cor de fundo
    SenãoSe um único polígono intersecta a janela então
      Desenhe o polígono e pinte o resto da janela com a cor de fundo
    SenãoSe o polígono mais próximo ao observador envolve a janela completamente então
      Desenhe esse polígono
    Senão
      tamanho := tamanho div 2;
```

```
            SubDiv( x+tamanho, y+tamanho, tamanho );
            SubDiv( x, y+tamanho, tamanho );
            SubDiv( x+tamanho, y, tamanho );
            SubDiv( x, y, tamanho );
        FimSe
    FimSe
FimProcedimento;
```

Quando a janela se reduz a um único pixel (tamanho=1), utiliza-se o mesmo método do lançamento de raio: Ache o polígono mais próximo que intersecta o pixel e pinte-o com a cor desse polígono. Se não existe tal polígono então pinte-o com a cor de fundo.

Se a janela é maior do que um pixel (tamanho > 1) então três situações estão previstas:

1. Se nenhum polígono intersecta a janela então pinte-a com a cor de fundo.
2. Se um único polígono intersecta a janela então desenhe-o e pinte o resto da janela com a cor de fundo.
3. Se o polígono mais próximo que intersecta a janela envolve-a completamente então pinte toda a janela com a cor desse polígono.

Se nenhuma das três situações acima acontece então subdivida a janela em quatro janelas menores. Falta explicar como se pode descobrir se um polígono intersecta uma janela ou não. E, caso intersecte, como descobrir se o polígono envolve completa ou parcialmente a janela (veja a figura 7.7). O problema pode ser resolvido da seguinte maneira:

1. Se um dos vértices do polígono pertence à janela então o polígono intersecta a janela (casos 2 ou 3).
2. Senão podemos ter caso 1 ou 4. Se um dos cantos da janela pertence ao polígono então o polígono envolve a janela (caso 4). Caso contrário, são disjuntos (caso 3). Observe que um dos cantos da janela pertence ao polígono se, e somente se, todos os cantos da janela pertencerem ao polígono. Como consequência, basta testar somente um canto da janela.

7.6 Algoritmos para modelos CSG

Nesta seção, vamos apresentar a estrutura básica dos algoritmos que trabalham com modelos CSG. Esta estrutura pode ser utilizada para criar algoritmos de superfícies escondidas para CSG e é, pela própria propriedade da árvore CSG, do tipo "dividir e conquistar". Outros algoritmos para CSG podem ser encontrados em [SolidMod], [IntrRay] e [Atherton, 83].

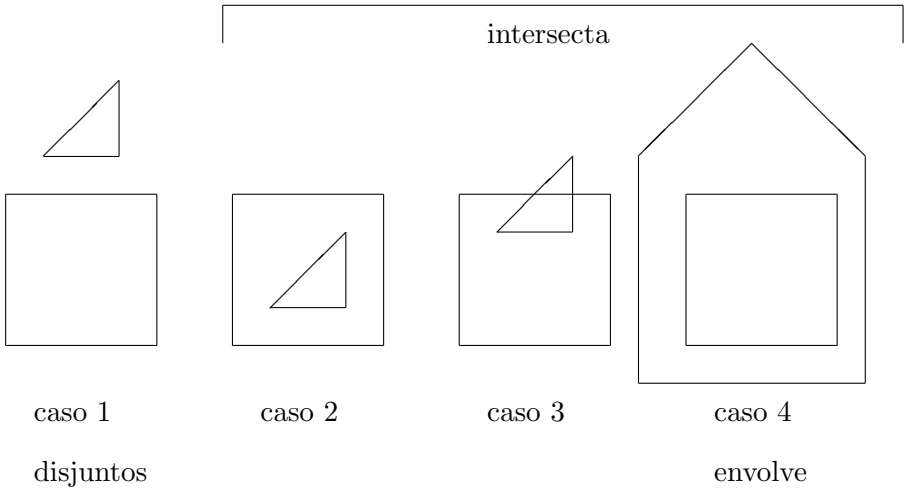


Figure 7.7: Casos de polígono x janela.

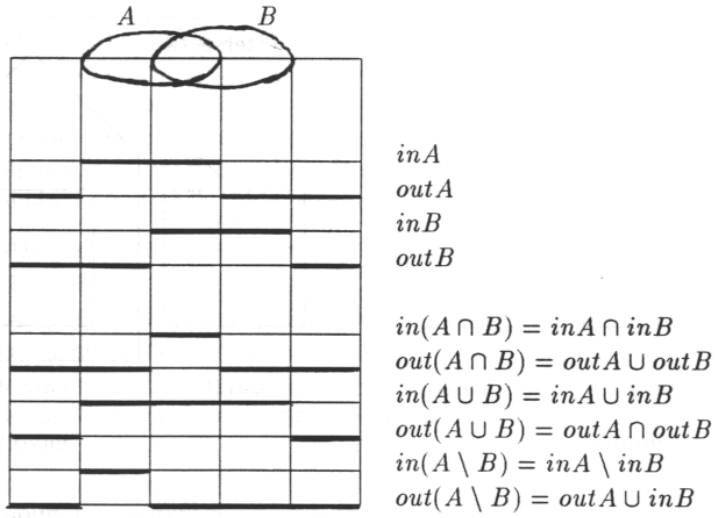


Figure 7.8: Combinações booleanas de duas classificações

Vamos descrever, primeiro, a função `CÁrvore` que, dado um segmento de reta e e uma árvore CSG S, classifica quais pedaços de e estão dentro de S e quais estão fora. Considere o tipo:

classificação = record $EinS$, $EoutS$: ConjuntoDeSegmentos end;

Então a função `CÁrvore` pode ser esquematizada como segue:

```
function CArvore(S:^ArvoreCSG; e:segmento):classificacao;
{ Classifica a aresta e em EinS e EoutS. }
begin
  if S^.op=primitiva then CArvore:=CPrim(S,e)
  else CArvore:=CCombine(CArvore(S^.left,E), CArvore(S^.right,E), S^.op);
end;
```

Falta descrever as funções CPrim e CCombine, que têm os seguintes cabeçalhos:

```
function CPrim(S:^ArvoreCSG; e:segmento):classificacao;

function CCombine(esquerda, direita:classificacao; Op:operacao): classificacao;
```

A função `CPrim` classifica a aresta e em $EinS$ e $EoutS$, respectivamente os pedaços de e que pertencem ao S e os que não lhe pertencem. S deve ser um modelo primitivo de CSG. Como todos os modelos primitivos de CSG são combinações booleanas de semi-espaços, a tarefa básica para escrever `CPrim` é classificar e contra um semi-espaço, o que é fácil.

A função `CCombine` combina duas classificações de acordo com a operação booleana Op para criar a classificação resultante. Com as regras da figura 7.8 conseguimos combinar qualquer operação booleana entre duas classificações.

Por fim, dispondo do procedimento `CÁrvore`, fica fácil implementar o algoritmo de lançamento de raio, rastreamento de raio e o cálculo das propriedades integrais como o volume, o centro de massa, etc.

Observe que as transformações geométricas devem ser aplicadas aos modelos primitivos (que ficam nas folhas da árvore CSG) antes de chamar a função `CÁrvore`.

Chapter 8

Técnicas Auxiliares

8.1 Meio tom[1]

No capítulo anterior, vimos várias técnicas para gerar imagens coloridas utilizando todas as cores que uma variável tipo COR é capaz de representar. Ora, uma variável tipo COR é um registro (structure em C e record em Pascal) com 3 campos em ponto flutuante: vermelho, verde e azul. Portanto, uma variável tipo COR é capaz de representar um número praticamente infinito de cores diferentes. Porém, a tela de um computador muitas vezes não consegue reproduzir tantas cores. Muitos computadores ainda possuem telas com apenas dois estados diferentes: aceso ou apagado. Mesmo as estações de trabalho coloridas como Sparc Station conseguem mostrar somente 256 cores diferentes ao mesmo tempo. Isto é muito pouco, pois consegue mostrar somente 8 tonalidades diferentes de vermelho, 8 de verde e 4 de azul ($8 \times 8 \times 4 = 256$). A razão desta limitação é a enorme quantidade memória necessária para a tela. Considere um computador com a resolução 1000×1000, capaz de mostrar simultaneamente 256 cores. Será necessário gastar 8 bits (1 byte) para cada pixel, pois $2^8 = 256$. Logo, a memória de vídeo ocupará $1000 \times 1000 \times 1$ bytes, ou seja, 1 megabyte! Portanto, toda vez que não trabalharmos com as placas gráficas especiais (de 24 bits ou 48 bits por pixel) teremos que executar um programa especial que cause ao observador a impressão de que o computador possui mais cores do que realmente tem. Este programa está baseado em algoritmos chamados de meio tom. Um algoritmo de meio tom típico, como é apresentado nos livros e revistas especializadas, destina-se a mostrar imagens não coloridas com várias tonalidades de cinza num dispositivo binível, por exemplo, uma impressora matricial, tela de computador tipo PC, impressora a laser, etc. Devemos fazer algumas adaptações nesses algoritmos para que funcionem nos dispositivos com mais de dois níveis de tonalidade ou nos dispositivos coloridos. Os algoritmos de meio tom para dispositivos biníveis podem ser encontrados em [Knuth, 87] e em [ProcElem]. O problema de meio tom, para dispositivos com 2^k níveis de tonalidade, pode ser formalizado:

Definição 8.1 *Dada uma matriz $A_{m \times n}$ de números reais no intervalo $[0 \ldots 1[$, queremos construir uma*

[1] Half-tone. Às vezes traduzido para português como meia tonalidade.

matriz $B_{m\times n}$ de números inteiros entre 0 e $2^k - 1$ tal que o valor médio de entradas $B[i,j]$ quando (i,j) são próximas a (i_o, j_o) seja igual a $2^k\, A[i_o, j_o] - 0.5$.

O valor k indica o número de bits utilizado para cada pixel da imagem. Quando $k = 1$, temos um dispositivo binível. Se a imagem é colorida, devemos aplicar o algoritmo três vezes, uma vez para cada cor primária.

8.1.1 Difusão de erro

Uma solução interessante para este problema foi apresentado por [Floyd, 75]. Consiste em difundir o erro cometido ao converter o valor real $A[i,j]$ para o valor inteiro $B[i,j]$ nos pixels vizinhos. Com isso, o erro cometido no arredondamento é compensado nos pixels vizinhos. Seja $E = 2^k$. Então o algoritmo de difusão de erro é:

```
for i:=1 to n do
  for j:=1 to m do begin
    B[i,j] := trunc(E*A[i,j]);  {Equivale a round(E*A[i,j]-0.5).}
    err := A[i,j] - B[i,j];
    A[i,j+1] := A[i,j+1] + err*alpha;
    A[i+1,j-1] := A[i+1,j-1] + err*beta;
    A[i+1,j] := A[i+1,j] + err*gamma;
    A[i+1,j+1] := A[i+1,j+1] + err*delta;
  end;
```

As constantes α, β, γ e δ são escolhidos de modo que a soma seja igual a um. Pode-se usar, por exemplo, $(\alpha, \beta, \gamma, \delta) = (7/16,\ 3/16,\ 5/16; 1/16)$ ou $(3/8,\ 0,\ 3/8,\ 2/8)$. Há autores que recomendam que a soma destas constantes seja um pouco menor que um para que o erro difundido não chegue até aos pixels muito distantes. Este método gera imagens excelentes, melhores que as geradas por qualquer outro método. Ele não pode ser paralelizado, pois o valor $B[m,n]$ depende de todas as mn entradas da matriz A. Para gerar as imagens para as impressoras a laser, existem técnicas especiais, pois os algoritmos convencionais de meio tom não podem ser usados para esta finalidade (veja [Knuth, 87] e [Velho, 91]).

O algoritmo de difusão de erro está implementado no programa MEIOTOM e funciona juntamente com programa RAIOS. Os resultados do programa MEIOTOM podem ser vistos nas figuras 9.7.

8.1.2 Disparo ordenado[2]

Uma outra técnica consiste em introduzir propositadamente erros na imagem, que são somados aos valores dos elementos da matriz $A[i,j]$ antes de se aplicar o algoritmo. O menor padrão de erro é uma matriz 2×2:

[2] Ordered dither.

$$D_2 = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

O padrão de erro é repetido horizontal e verticalmente em toda a imagem formando um quadriculado. O erro a ser somado é um número entre -0.5 e $+0.5$ e é obtido dividindo a entrada da matriz D por l^2 e depois subtraindo 0.5. O valor l é o número de colunas (ou linhas) da matriz D. Os padrões de erros maiores, 4×4, 8×8, etc são obtidos usando a seguinte relação recursiva:

$$D_l = \begin{bmatrix} 4D_{l/2} & 4D_{l/2} + 2U_{l/2} \\ 4D_{l/2} + 3U_{l/2} & 4D_{l/2} + U_{l/2} \end{bmatrix} \quad l = 2^n, \quad n \geq 2$$

Onde U_l é uma matriz $l \times l$ cujos elementos são todos iguais a um. Por exemplo, a matriz de padrão de erro 4×4 é:

$$D_4 = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

O algoritmo de disparo ordenado é:

```
for i:=1 to m do
  for j:=1 to n do
    B[i,j] := trunc( E*A[i,j] + ( D[i mod l,j mod l] / (l*l) - 0.5 ) );
```

Onde $E = 2^k$. Este algoritmo é facilmente paralelizável.

8.2 Enfatização de bordas

As imagens geradas pelos algoritmos de meio tom podem melhorar sensivelmente se os contornos da imagem original são realçados. Isto é especialmente verdade para obter imagens biníveis. A melhora é menos substancial quando o dispositivo gráfico possui mais tonalidades. A idéia é substituir $A[i,j]$ por

$$A'[i,j] = \frac{A[i,j] - \alpha \bar{A}[i,j]}{1 - \alpha} \quad 0 \leq \alpha < 1$$

Onde $\bar{A}[i,j]$ é a média de $A[i,j]$ e de seus oito vizinhos:

$$\bar{A}[i,j] = \frac{1}{9} \sum_{u=i-1}^{i+1} \sum_{v=j-1}^{j+1} A[u,v]$$

Se $\alpha = 0$, $A'[i,j] = A[i,j]$. Quanto mais α aproxima-se de 1, as bordas são mais realçadas. Se a imagem é colorida, basta aplicar uma vez o processo acima para cada uma das três cores primárias.

Part III

Rastreamento de Raio

Chapter 9

Rastreamento de Raio Clássico

As técnicas para sintetizar as imagens bidimensionais a partir da descrição de objetos tridimensionais começaram a ser estudadas desde os meados da década de 60. Um bom resumo dessas técnicas pode ser encontrado no artigo [Sutherland, 74] e no livro [ProcElem]. Mas todas essas técnicas trabalhavam com modelos de iluminação local, isto é, sem levar em consideração os efeitos de reflexão e de refração entre os objetos. Por exemplo, utilizando o modelo de iluminação local, não se pode sintetizar uma imagem de várias bolas de Natal, onde cada uma reflete as outras, nem se pode sintetizar um objeto metálico polido que reflete os objetos à sua volta. A iluminação local também não permite criar objetos verdadeiramente transparentes que obedecem à lei de Snell. Somente é possível sintetizar objetos transparentes de espessura fina, onde o desvio dos raios de luz devido à refração é desprezível.

Tudo isso mudou com a criação do algoritmo de rastreamento de raio, publicado no famoso artigo [Whitted, 80]. O rastreamento de raio permitiu criar as imagens com um realismo impressionante, que não poderiam ter sido sintetizadas utilizando os algoritmos conhecidos até então. Ele consegue calcular, de um modo simples e elegante, as reflexões, refrações e sombras. Este foi o primeiro algoritmo a utilizar um modelo de iluminação global. Mais tarde, apareceu o algoritmo de radiosidade no artigo [Goral, 84] que funciona baseado num princípio totalmente diferente do rastreamento de raio.

O rastreamento de raio funciona muito bem para sintetizar as imagens de ambientes com a incidência de luz direta da fonte, sombras nítidas e objetos com alto grau de reflexão especular, como bolas de Natal, objetos metálicos polidos, etc. O algoritmo de radiosidade, por sua vez, consegue simular bem os ambientes noturnos, com a fonte de luz suave como a lâmpada fluorescente com uma placa isolux e objetos onde predominam a reflexão difusa, como papel, parede com tinta à base d'água, madeira, etc. Atualmente, há uma intensa pesquisa para criar novos algoritmos com as vantagens oferecidas por ambos, veja por exemplo, [Wallace, 87].

Após o artigo [Whitted, 80], apareceu uma grande variedade de versões do rastreamento de raio:

- Ou melhorando o tempo de processamento (veja [Heckbert, 84], [Arvo, 87] e [Shinya, 87]);
- Ou melhorando a qualidade da imagem gerada, permitindo criar alguns efeitos visuais impossíveis

de serem geradas pelo algoritmo original (veja [Amanatides, 84] e [Cook, 84]);

- Ou possibilitando trabalhar com objetos não previstos pelo algoritmo de Whitted (veja [Blinn, 82], [Sederberg, 84], [Kajiya, 84], [Hanrahan, 83] e [Kajiya, 83]).

Muitos dos artigos citados acima estão apresentados de um modo resumido no livro [IntrRay]. O rastreamento de raio tem sido utilizado poucas vezes na animação comercial. Nenhuma rede de televisão brasileira utiliza-o para criar as suas vinhetas, o que é devido ao tempo enorme de processamento de que este algoritmo necessita. Se um programa de rastreamento de raio fosse utilizado para animação, levaria aproximadamente 100 dias de processamento para criar um minuto de filme utilizando uma estação de trabalho tipo Sparc. Por isso, os programas de animação comercial, como por exemplo "Topas" para os computadores da linha PC/XT/AT, continuam utilizando o modelo de iluminação local. Neste capítulo, apresentaremos o algoritmo de rastreamento de raio da maneira concebida por [Whitted, 80].

9.1 Modelo de Iluminação de Phong Global

A intensidade luminosa de um ponto do objeto não depende somente da fonte de luz. Depende também da presença ou não de outros objetos no ambiente, pois estes objetos funcionam como fontes de luz secundárias, gerando as reflexões e as refrações de um objeto no outro. Como podemos calcular os efeitos de reflexão e de refração de um objeto no outro? Uma idéia seria seguir a trajetória dos raios de luz a partir das fontes até chegar ao observador. Este processo consegue simular exatamente o processo físico da iluminação. Ele simula corretamente a reflexão e a refração entre os objetos. Porém, apresenta um inconveniente: é extremamento caro computacionalmente, pois apenas uma fração muito pequena dos raios de luz que partiram das fontes chegarão até o observador.

Um método alternativo é traçar os raios no sentido inverso, do observador para os objetos. Já utilizamos esta técnica no lançamento de raio (veja a seção 7.2). Como é impossível, computacionalmente, calcular as trajetórias de todos os raios de luz, algumas simplificações deverão ser feitas. Podemos utilizar o modelo de iluminação de Phong, que já vimos na seção 7.1, modificando-o para que se torne global. O modelo de Phong é um modelo bastante simplificado da realidade física. Esta simplificação e o fato de estarmos traçando os raios no sentido inverso acabam acarretando uma série de erros na imagem gerada, o que é inevitável. Por exemplo, o rastreamento de raio não consegue calcular corretamente a sombra de um objeto transparente nem calcular as reflexões difusas entre os objetos.

Para facilitar a descrição, suporemos que não existem objetos transparentes e que existe uma única fonte de luz. Além disso, iremos supor que o observador esteja situado na posição $\dot{E} = (0, 0, 0)$, olhando para a direção do vetor $\vec{F} = (0, 0, 1)$. À distância unitária do observador existe uma tela imaginária quadriculada onde a cada quadriculado corresponde um pixel da tela do computador. Um raio de luz é lançado do observador para o centro de cada pixel da tela. Um raio de luz que parte do observador é chamado de raio do observador. Se esse raio não atingir nenhum objeto, então o pixel por onde ele passou recebe a cor de fundo, normalmente preta. Caso contrário, digamos que esse raio do observador

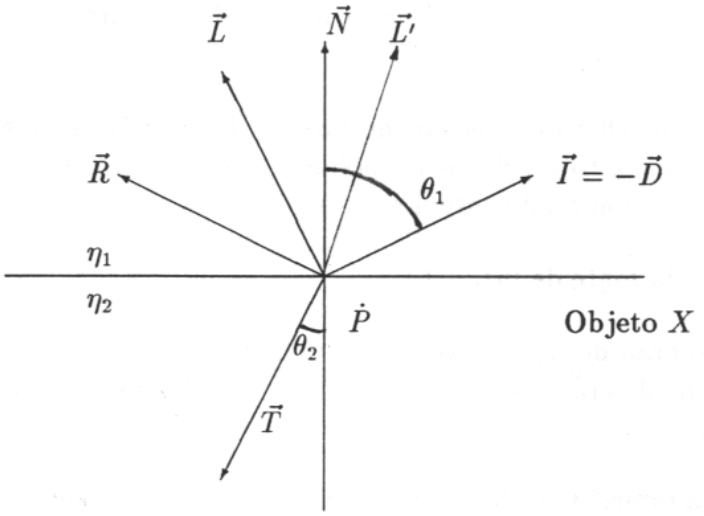


Figure 9.1: Geometria de reflexão.

atingiu um ponto $\dot{P}$ de um objeto X. Então, a intensidade luminosa do ponto $\dot{P}$ é calculada somando-se os seguintes termos:

Termo 1: Quantidade de luz proveniente da reflexão especular da fonte de luz.

Termo 2: Quantidade de luz proveniente da reflexão difusa da fonte de luz.

Termo 3: Quantidade de luz refletida especularmente proveniente de outros objetos.

Termo 4: Quantidade de luz ambiente refletida difusamente.

Vejamos agora como calcular cada um destes quatro termos. Para gerar uma imagem colorida, é preciso aplicar as equações que serão descritas três vezes, uma vez para cada cor primária.

9.1.1 Termo 1: reflexão especular da fonte

A quantidade de luz proveniente da reflexão especular da fonte pode ser calculada pela seguinte equação, que é a mesma vista na subseção 7.1.1.

$$I_s = K_s L(\vec{N} \circ \vec{L'})^e$$

onde (veja a figura 9.1),

- I_s é a intensidade de luz proveniente diretamente da fonte refletida especularmente.
- $\vec{N}$ é o vetor unitário normal à superfície de X no ponto $\dot{P}$.
- $\vec{L'}$ é o vetor unitário que indica a direção intermediária entre a direção da fonte de luz $\vec{L}$ e do observador $\vec{I}$ ($\vec{L'} = versor(\vec{L} + \vec{I})$).
- K_s é a constante de reflexão especular do objeto. É um número real entre 0 e 1 e é da ordem de 0.9 para os objetos muito espelhados e polidos. Existem três constantes K_s, uma para cada cor primária, e as três são normalmente iguais num objeto.
- L é a intensidade da fonte de luz.
- A constante e é o grau de espalhamento da luz refletida especularmente. Quanto maior for o valor de e, mais direcional será a reflexão especular. O valor de e é da ordem de 60 para superfícies metálicas polidas.

O termo I_s entra na intensidade do ponto $\dot{P}$ se, e somente se, o ponto $\dot{P}$ não estiver na sombra. Para descobrir se o ponto $\dot{P}$ está na sombra ou não, um raio é lançado do ponto $\dot{P}$ em direção à fonte de luz. Se esse raio encontrar algum obstáculo então o ponto $\dot{P}$ estará na sombra. Os raios que partem em direção à fonte de luz são chamados de raios de sombra.

9.1.2 Termo 2: reflexão difusa da fonte

A intensidade da reflexão difusa da fonte pode ser descrita pela mesma equação já vista na subseção 7.1.2:

$$I_d = K_d L(\vec{N} \circ \vec{L})$$

onde (veja a figura 9.1),

- I_d é a intensidade de luz proveniente diretamente da fonte refletida difusamente.
- $\vec{N}$ é o vetor unitário normal à superfície de X no ponto $\dot{P}$.
- $\vec{L}$ é o vetor unitário que indica a direção da fonte de luz.
- K_d é a constante de reflexão difusa e assume um valor entre 0 e 1.
- L é a intensidade da fonte de luz.

O termo I_d também só entra na intensidade de luz do ponto $\dot{P}$ se o ponto $\dot{P}$ não estiver na sombra.

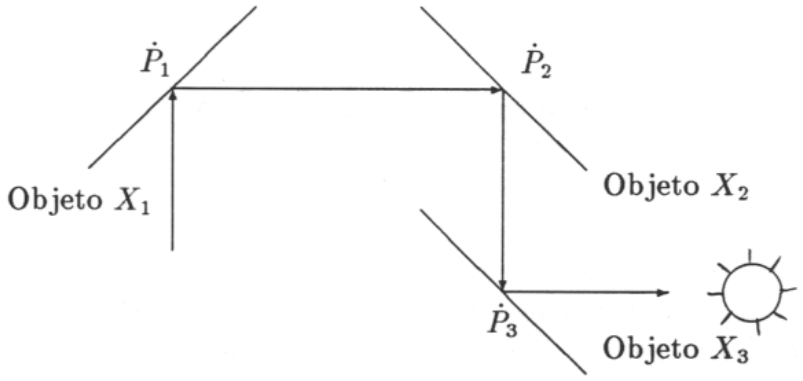


Figure 9.2: Cálculo recursivo das reflexões.

9.1.3 Termo 3: reflexão especular de luz proveniente de outros objetos

Além da luz que chega ao ponto $\dot{P}$ diretamente da fonte que modelamos nos dois primeiros termos, a luz pode provir indiretamente de outros objetos. Veja a figura 9.2, onde mostra a luz que chega ao observador depois de ter refletido especularmente três vezes. O computador calcula a intensidade desta luz seguindo a trajetória dos raios no sentido inverso da trajetória real. Evidentemente, a luz também pode chegar ao observador após dois ou mais de três reflexões. O rastreamento de raio só consegue modelar a luz proveniente de outros objetos que reflete especularmente na superfície, isto é, a reflexão difusa indireta é considerada uma constante, como veremos no termo 4. O algoritmo de radiosidade, por sua vez, só consegue calcular as reflexões difusas. Para o algoritmo de radiosidade, não existe reflexão especular e inclusive a própria fonte de luz é considerada difusa, o que explica porque a fonte de luz do radiosidade deve ser "suave" como um abajur ou uma lâmpada fluorescente com a placa acrílica isolux. Para calcular a intensidade de luz devida à reflexão especular indireta aplica-se a fórmula (veja a figura 9.2):

$$I_{r,i} = K_{r,i}\, I_{i+1} \quad \text{para} \quad i \geq 1$$

Onde $I_{r,i}$ é a intensidade de luz no ponto $\dot{P}_i$ devida à reflexão especular da luz indireta, $K_{r,i}$ é a constante de reflexão do objeto ao qual pertence o ponto $\dot{P}_i$ e I_{i+1} é a intensidade de luz em ponto $\dot{P}_{i+1}$ vista do ponto $\dot{P}_i$. Para calcular I_{i+1} é necessário calcular antes I_{i+2} e assim por diante. I_{i+1}, I_{i+2}, etc. são calculadas recursivamente e nelas estão incluídas todos os quatro termos que contribuem para a intensidade de luz num ponto. A recursão é cortada quando a intensidade de luz do ponto $\dot{P}_n$ contribuir muito pouco (digamos, menos de 10 ou 20%) para a intensidade da luz do ponto $\dot{P}_1$. Para aplicar a fórmula recursivamente e descobrir a posição do ponto $\dot{P}_{i+1}$, devemos lançar um raio de reflexão a partir do ponto $\dot{P}_i$.

Se ignorarmos este termo (I_r) então o rastreamento de raio torna-se lançamento de raio e o modelo

de iluminação passa a ser local. Para calcular as intensidades I_2, I_3, etc. é necessário calcular a direção de reflexão $\vec{R}$ da figura 9.1:

$$\vec{R} = versor(\vec{D} - 2\,(\vec{N} \circ \vec{D})\,\vec{N})$$

Onde $\vec{D} = -\vec{I}$.

Observação 9.1 *Como já explicamos na observação 7.1, para sermos rigorosos, deveríamos dividir os termos I_s e I_d por d^2, onde d é a distância entre a fonte de luz e o ponto $\dot{P}$. Porém, o termo I_s não deve ser dividido de maneira alguma por d^2. Veja a figura 9.3. Suponha que o objeto X reflete especularmente a luz proveniente de um pequeno ângulo sólido de ω esteroradianos*[1] *em torno da semi-reta de reflexão. Suponha que no primeiro caso, o observador enxerga, pela reflexão, um objeto X_1 e que no segundo caso, enxerga o objeto X_2. O objeto X_1 está mais próximo ao observador que o X_2 e ambos os objetos têm a mesma intensidade luminosa. Demonstraremos que o observador enxerga X_1 e X_2 como se tivessem a mesma intensidade luminosa, apesar de X_2 estar mais longe do que X_1.*

Se a reflexão provém do objeto X_1, o observador enxerga através da reflexão especular, uma área ωd_1^2, e se a reflexão provém do objeto X_2, o observador enxerga uma área ωd_2^2, onde d_1 e d_2 são as distâncias do ponto $\dot{P}$ aos objetos X_1 e X_2, respectivamente. Ora, como a intensidade de luz decai com o quadrado da distância, se a reflexão especular provém do objeto X_1, o observador vê uma intensidade proporcional a ω através da reflexão especular ($\omega = \omega d_1^2/d_1^2$). E se a reflexão especular provém do objeto X_2, o observador também vê a mesma intensidade de luz, proporcional a ω. Portanto, concluimos que a intensidade da reflexão especular indireta independe da distância entre os objetos.

9.1.4 Termo 4: luz ambiente

Se no modelo de iluminação houvesse apenas os três termos anteriores, um objeto sem reflexão especular que estivesse na sombra apareceria como se fosse completamente preto. Isto aconteceria, por exemplo, com uma folha de papel que estivesse na sombra. Ora, sabemos que não é isto o que acontece na realidade, pois nesse papel chegam as luzes refletidas (difusa ou especularmente) por outros objetos e que o papel reflete difusamente. O algoritmo de radiosidade consegue calcular corretamente este fenômeno. No rastreamento de raio, porém, a luz ambiente é considerada constante. A seguinte equação, já vista na subseção 7.1.3, calcula a intensidade da luz ambiente:

$$I_a = K_a\, L_a$$

Onde K_a é a constante de reflexão da luz ambiente e L_a é a intensidade da luz ambiente.

[1] Lembre-se de que o radiano é o comprimento do arco da circunferência dividida pelo raio da circunferência. Do mesmo modo, o esteroradiano é a área de um pedaço da esfera dividida pelo raio da esfera ao quadrado.

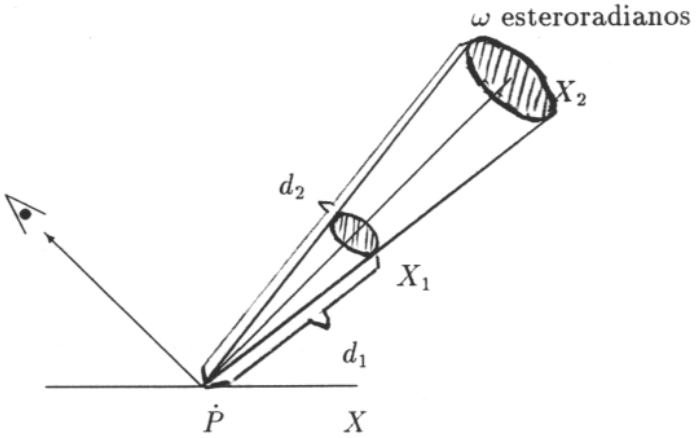


Figure 9.3: Intensidade da reflexão especular independe da distância.

9.1.5 Resumo

Resumindo o que vimos, para cada pixel da tela é lançado um raio do observador. Se esse raio não encontra nenhum objeto, o pixel recebe a cor preta. Senão, suponha que esse raio atinja um ponto $\dot{P}_1$. Neste caso, a seguinte fórmula recursiva calcula a intensidade de luz I_1 do ponto $\dot{P}_1$:

Se $\dot{P}_1$ está na sombra, isto é, se o raio de sombra encontra algum obstáculo
$I_1 = K_{r,1}\, I_2 + K_a\, L_a$
Senão
$I_1 = K_s\, L\, (\vec{N} \circ \vec{L}')^e + K_d\, L\, (\vec{N} \circ \vec{L}) + K_{r,1}\, I_2 + K_a\, L_a$

Onde a intensidade I_2 deve ser calculada recursivamente. Se existem k fontes de luz, substitua a fórmula acima por:

$I_1 = K_{r,1}\, I_2 + K_a\, L_a$
Para i de 1 até k faça
Se $\dot{P}_1$ não está na sombra de L_i então
$I_1 = I_1 + K_s L_i (\vec{N} \circ \vec{L}_i')^e + K_d L_i (\vec{N} \circ \vec{L}_i)$
FimPara

9.1.6 Transparência

Criar objetos transparentes seguindo o modelo físico não é uma tarefa simples. Já vimos que utilizando o algoritmo do pintor é possível criar objetos transparentes de espessura fina, onde o desvio dos raios

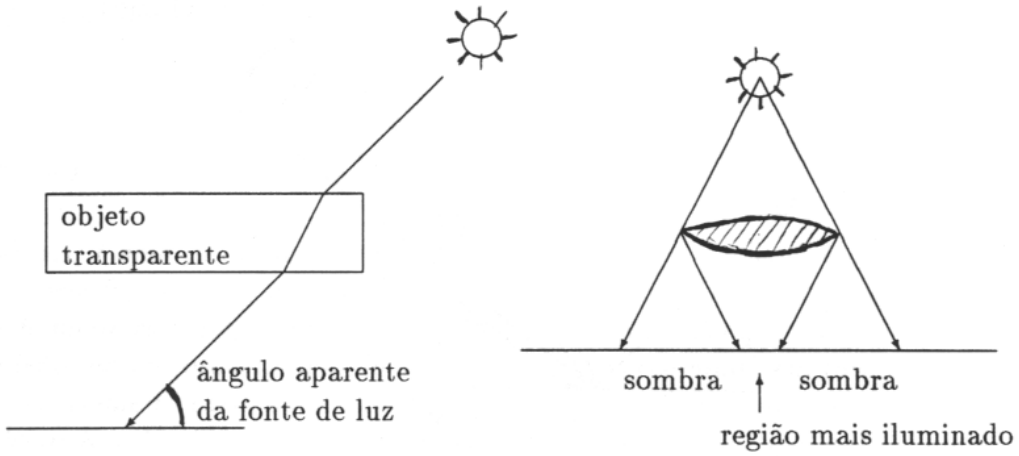


Figure 9.4: Sombra de objeto transparente.

de luz devido à refração é desprezível. Utilizando o rastreamento de raio, é possível criar imagens com objetos transparentes que obedecem à lei de Snell. Porém, ainda restam muitos problemas para serem resolvidos. Por exemplo, como calcular corretamente a sombra de um objeto transparente? Este problema permaneceu aberto até hoje. [Amanatides, 84] afirmou que o seu algoritmo era incapaz de resolver este problema. O autor opinaria que, traçando os raios a partir do observador para os objetos, é impossível resolver este problema para o caso geral, pois para isso seria necessário ter um método para "adivinhar" o ângulo de desvio do raio de sombra devido ao objeto transparente. Se observarmos as imagens geradas por rastreamento de raio que contenham objetos transparentes, repararemos que ou os objetos transparentes não projetam nenhuma sombra ou então eles projetam sombras de intensidade luminosa uniforme, que não obedecem à lei de Snell. Uma sombra verdadeira de um objeto transparente pode inclusive aumentar a intensidade luminosa. Um exemplo típico disto é a sombra de uma lente convexa. No meio da sua sombra aparece uma região intensamente brilhante (veja a figura 9.4). Observe a sombra de um copo de vidro e repare como a sombra deste objeto é complexa. No final deste trabalho (seção 12.2), apresentaremos um algoritmo inédito para calcular as sombras de objetos transparentes e que denominamos de rastreamento bidirecional.

Um outro problema é decidir se deve ou não se deve aplicar o modelo de iluminação no interior do objeto transparente. Veja a figura 9.5. No ponto $\dot{B}$ deve-se ou não se deve aplicar o modelo de iluminação? Existem imagens geradas pelos rastreadores de raio que não aplicam a equação de iluminação nesse ponto. Porém, se observarmos atentamente um objeto real transparente, repararemos que ele apresenta duas (ou mais) reflexões: uma reflexão na superfície externa (ponto $\dot{A}$) e outra na superfície interna (ponto $\dot{B}$). O autor observou a reflexão do filamento de uma lâmpada elétrica (não leitosa) numa janela para verificar este fenômeno. Se, porém, resolvemos aplicar o modelo de iluminação no ponto $\dot{B}$, obedecendo à realidade

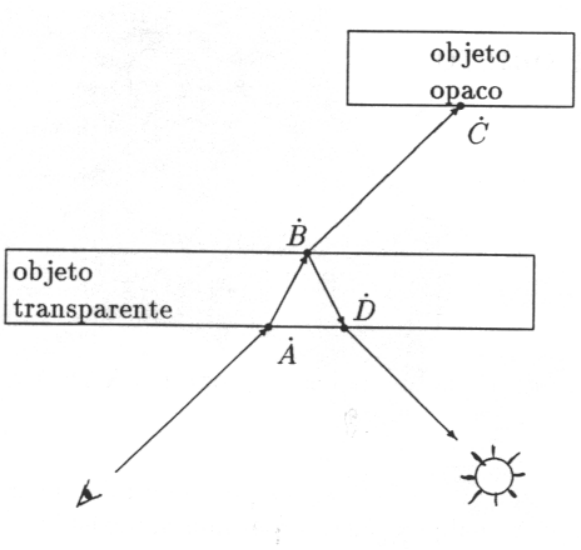


Figure 9.5: Problema de modelo de iluminação no objeto transparente.

física, aparece o mesmo problema de determinar a sombra do objeto transparente: o ângulo aparente da fonte de luz, visto do ponto $\dot{B}$, é indeterminável. Com estas observações, os problemas para sintetizar a imagem correta de um objeto transparente devem ter-se tornado claros.

Se deseja criar objetos transparentes ignorando os problemas acima, é necessário calcular o vetor $\vec{T}$ da figura 9.1. Isto pode ser calculado da seguinte maneira:

$d = -\vec{N} \circ \vec{D}$ (d será sempre maior ou igual a zero).
$\eta = \eta_1/\eta_2$
$b = 1 - \eta^2\,(1 - d^2)$
Se $b < 0$ então não há refração. A reflexão é total.
Senão, calcule $c = \sqrt{b}$.
$\vec{T} = \eta\vec{D} - (c - \eta d)\,\vec{N}$

Onde η_1 e η_2 são os índices de refração dos dois meios. $\vec{T}$ calculado desta forma satisfaz à lei de Snell: $\eta_1\, sen\theta_1 = \eta_2\, sen\theta_2$.

9.2 O Programa RAIOS

O programa RAIOS é uma implementação de rastreamento de raio em linguagem C. Ele foi criado "a partir de zero", isto é, sem adaptar qualquer outro programa pré-existente. Esta implementação gera

Figure 9.6: Fluxo de dados do programa RAIOS.

um arquivo texto contendo as cores com que se deve pintar cada pixel da tela (veja a figura 9.6). Este arquivo é lido e mostrado na tela por um outro programa chamado MEIOTOM, que é a implementação do algoritmo de difusão de erro (veja [ProcElem] e [Knuth, 87]) alterado para gerar as imagens coloridas, e deve ser diferente para cada tipo de computador. Este programa causa a impressão de que o monitor tem mais cores do que de fato possui. No momento, os programas RAIOS e MEIOTOM estão rodando nas estações de trabalho Sparc mas são facilmente transportáveis para outros computadores por terem sido escritos em C segundo a definição de [CProg] e por utilizarem um arquivo texto como meio de comunicação entre eles. As estações de trabalho que utilizamos possuem somente 256 cores simultâneas, o que torna o programa MEIOTOM indispensável para gerar as imagens de qualidade. As imagens geradas pelo RAIOS podem ser vistas nas figuras 9.7. Apresentaremos aqui o esqueleto do programa RAIOS.

Em primeiro lugar, como o programa trabalha constantemente com os vetores, declaramos o tipo VETOR e criamos as funções que o manipulam:

```
typedef struct {double x,y,z;} VETOR;
VETOR versor(v) VETOR v;              /* Calcula o versor do vetor v. */
double distancia(p,q) VETOR p,q;      /* Distancia entre dois pontos. */
VETOR diferenca(u,v) VETOR u,v;       /* Subtracao vetorial. */
VETOR soma(u,v) VETOR u,v;            /* Soma vetorial. */
double escalar(u,v) VETOR u,v;        /* Produto escalar entre u e v. */
VETOR multesc(t,v) double t; VETOR v; /* Multiplica v pelo escalar t. */
```

Acreditamos que o que cada uma destas funções faz fica claro pelos seus cabeçalhos, juntamente com os comentário. Para acelerar a velocidade de processamento, estas funções podem ser substituídas pelos macros. Vamos definir mais três estruturas muito utilizadas no programa:

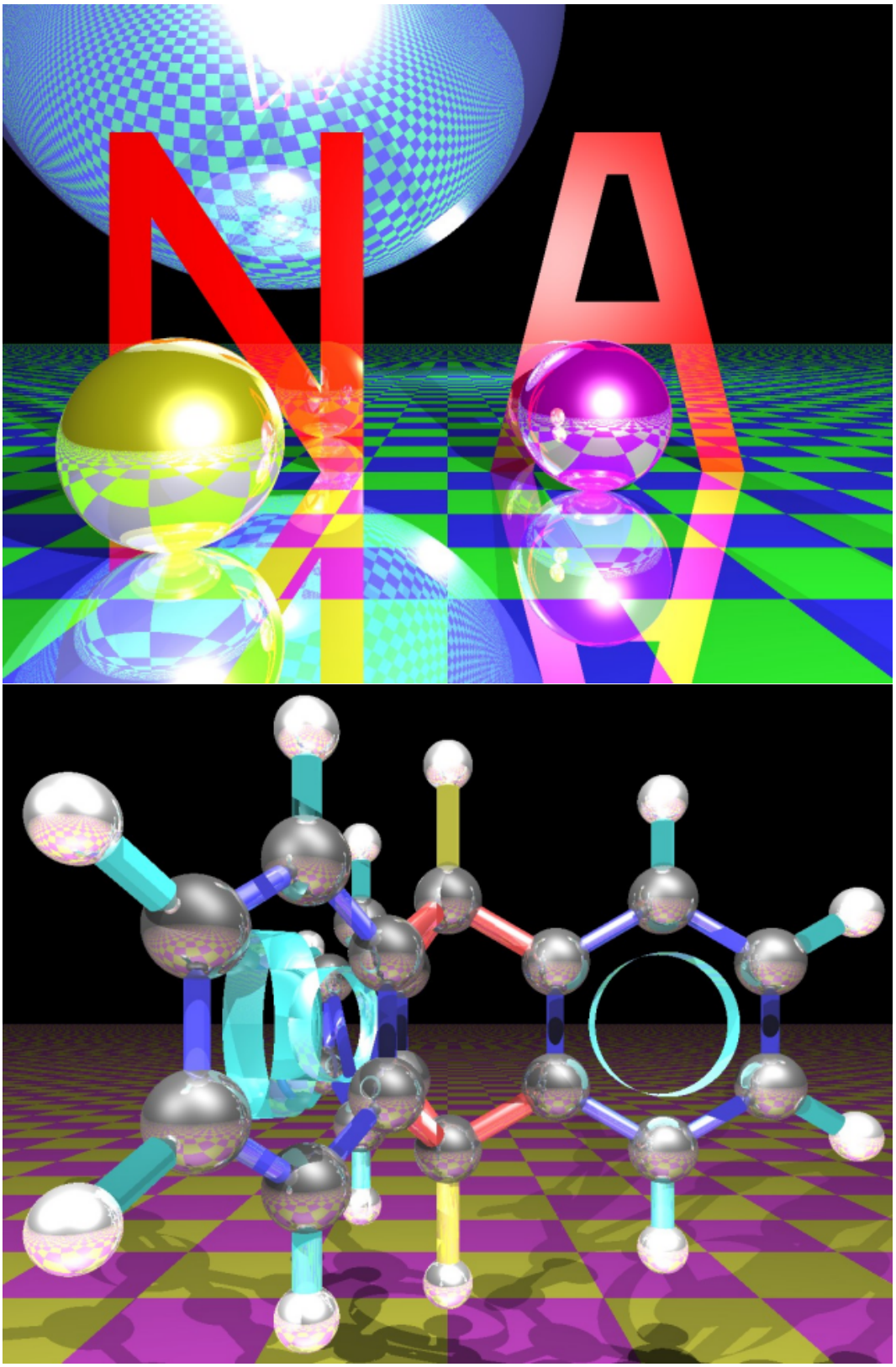


Figure 9.7: Imagens geradas pelo RAIOS e processadas pelo MEIOTOM (de [Kim, 92]).

RAIO: Define uma semi-reta com a origem p e a direção v. A sua definição em C é: `typedef struct {VETOR p,v;} RAIO;`

COR: Uma variável tipo COR armazena a intensidade de luz para cada uma das três cores primárias. Para poder utilizar as mesmas funções (ou macros) que definimos para os vetores, nomeamos os três campos de COR de x-y-z, respectivamente vermelho, verde e azul. A definição de COR fica, portanto, simplesmente: `#define COR VETOR`

CR: É a abreviação de características. Uma variável CR armazena as características visuais da superfície de um objeto. São as constantes utilizadas no modelo de iluminação que vimos na seção anterior. `typedef struct {COR kd,ks,kr,ka; double e;} CR;`

Para cada tipo de objeto é preciso escrever as três funções abaixo (veremos mais detalhes destas funções na seção 10):

1. `void inic(x) OBJETO *x;`
   Pré-processa o objeto x, fazendo alguns cálculos que podem ser efetuados uma única vez no começo do programa. Por exemplo, se o objeto é uma esfera, neste procedimento são calculados E_r^2 e $1/E_r$, onde E_r é o raio da esfera. Estes valores serão usados repetidas vezes para calcular as intersecções.

2. `double inter(r,x) RAIO r; OBJETO *x;`
   Calcula a distância percorrida pelo raio r desde a sua origem r_p até atingir o objeto x pela primeira vez. Devolve um valor negativo se não há intersecção entre r e x.

3. `void carac(r,x,t,cr,n) RAIO r; OBJETO *x; double t; CR **cr; RAIO *n;`
   Esta função recebe o raio r, o objeto x e a distância $t = inter(r, x)$ e devolve um apontador para as características visuais do objeto x (na variável cr), o ponto de intersecção de r com x (na variável $n.p$) e o normal da superfície do objeto x (na variável $n.v$).

Como, para cada tipo de objeto é necessário escrever as três funções acima, aparecem, por exemplo, as funções inicplano, interplano, caracplano, inicpoligono, interpoligono, etc. Agora, é necessário escrever a função recursiva *intensidade* que calcula a intensidade luminosa de cada pixel. O seu cabeçalho é:

```
COR intensidade(r,pct) RAIO r; COR pct;
```

Esta função calcula a intensidade luminosa vista por um observador situado no ponto $r.p$ quando olha na direção $r.v$. A variável *pct* indica qual fração da intensidade calculada irá contribuir para a cor final do pixel. A variável *pct* vale $(1, 1, 1)$ quando r é um raio do observador. Se todos os campos de *pct* tornarem-se menores que um limite dado (por exemplo, 0.2) então a recursão é cortada. A parte central da função main fica:

```
VETOR u, um; COR cor; RAIO r;
r.p.x=r.p.y=r.p.z=0.0; um.x=um.y=um.z=1.0;
for (u.y=MaximoY; u.y>=MinimoY; u.y=u.y-DeltaY) {
  for (u.x=MinimoX; u.x<=MaximoX; u.x=u.x+DeltaX) {
    r.v=versor(u); cor=intensidade(r,um);
    /* Imprima aqui cor.x, cor.y e cor.z no arquivo de saida */
  }
}
```

MinimoX, MaximoX, MinimoY e MaximoY são as dimensões da tela e DeltaX e DeltaY são as distâncias entre um pixel e outro. No próximo capítulo, veremos como calcular a intersecção objeto/raio para os diferentes tipos de objetos.

Chapter 10

Cálculo de Intersecção

No capítulo anterior, descrevemos: "Um raio de luz é lançado do observador para o centro de cada pixel da tela. Se esse raio não atingir nenhum objeto faça ... senão faça ...". Como o computador descobre se um raio atingiu um objeto ou não? Neste capítulo, vamos procurar responder a esta pergunta.

Na nossa mente, imaginamos um raio de luz partindo do seu ponto de origem, atravessando o ar e chocando-se com um objeto. Dentro do computador, porém, não é exatamente isto o que acontece, pois no computador o "raio" não se movimenta no tempo. O que acontece na realidade é que o computador calcula as intersecções de uma semi-reta (um raio de luz é tratado como uma semi-reta pelo computador) com todos os objetos do ambiente, e depois, entre todas essas intersecções, escolhe aquela que estiver mais próxima do ponto de origem do raio. Disso podemos deduzir que para "lançar um raio", na verdade é necessário saber calcular a intersecção de uma semi-reta com cada tipo de objeto. Vamos dividir o processo de cálculo de intersecção em três:

Fase 1 – pré-processamento: Nesta fase, o objeto é pré-processado, fazendo alguns cálculos que podem ser efetuados uma única vez no começo do programa. Os resultados destes cálculos são armazenados em variáveis convenientes. Por exemplo, se uma esfera foi definida pelo seu centro $\dot{E}_p$ e pelo seu raio E_r, no pré-processamento calcula-se $(E_r)^2$ para evitar que seja necessário calculá-lo várias vezes.

Fase 2 – cálculo da distância percorrida: Nesta fase, dispondo das informações do objeto pré-processadas na fase 1, calcula-se a distância percorrida t pelo raio desde a sua origem até se chocar com um objeto. Para alguns objetos, pode ser necessário calcular alguma informação adicional além da distância t. Se o raio não se chocar com nenhum objeto, devolve-se uma distância negativa.

Fase 3 – cálculo das características da superfície: Nesta última fase, utilizando as informações obtidas nas duas primeiras fases, são calculados o ponto de intersecção $\dot{P}$, a normal da superfície $\vec{N}$ e as características visuais do objeto (tais como as constantes de reflexão difusa, especular, etc.).

Esta divisão do cálculo de intersecção em três fases tem motivos de conveniência prática. A conveniência de separar a fase de pré-processamento é clara, pois ela economiza o tempo de processamento. Por outro lado, nem sempre queremos obter as características da superfície. Por exemplo, quando lançamos um raio de sombra, só estamos interessados em descobrir se existe algum objeto entre a origem do raio e a fonte de luz, o que pode ser feito sem executar a fase 3.

Se o objeto é uma superfície implícita, o seguinte método geral calcula a intersecção. Sejam dadas uma semi-reta $R(t) = \dot{R}_p + \vec{R}_v t,\ t \geq 0$ e uma superfície implícita $F(x, y, z) = 0$. Substituindo a equação da semi-reta dentro da função F, obtemos uma nova equação com uma única variável:

$$F(R_{px} + R_{vx}t, R_{py} + R_{vy}t, R_{pz} + R_{vz}t) = 0$$

Basta resolvê-lo para calcular a intersecção. Utilizaremos este método geral para achar as intersecções de um raio com esferas, planos, cilindros, cones, etc.

10.1 Esfera

Sejam dados um raio (isto é, uma semi-reta) R com a origem em $\dot{R}_p$ e a direção $\vec{R}_v$ ($\vec{R}_v$ é um vetor unitário) e uma esfera E de raio E_r e centro $\dot{E}_p$. Queremos descobrir a primeira intersecção do raio R com a esfera E.

Primeiro, vamos transladar a semi-reta e a esfera de modo que o centro da esfera coincida com o ponto (0,0,0). A origem do raio R torna-se, então $\dot{Q} = \dot{R}_p - \dot{E}_p$. A direção $\vec{R}_v$ e o raio da esfera E_r não devem ser alterados. A equação da esfera é, após a translação, $x^2 + y^2 + z^2 = E_r^2$ e a equação da semi-reta é $\dot{Q} + \vec{R}_v t,\ t \geq 0$. Escrevendo explicitamente todos os componentes da equação da semi-reta R, temos:

$$[\ Q_x + R_{vx}t,\ Q_y + R_{vy}t,\ Q_z + R_{vz}t\ ], \quad t \geq 0$$

Substituindo a equação acima na equação da esfera, obtemos:

$$(Q_x + R_{vx}t)^2 + (Q_y + R_{vy}t)^2 + (Q_z + R_{vz}t)^2 = E_r^2$$

Separando os termos e agrupando de acordo com o grau de t:

$$(R_{vx}^2 + R_{vy}^2 + R_{vz}^2)t^2 + 2(Q_xR_{vx} + Q_yR_{vy} + Q_zR_{vz})t + (Q_x^2 + Q_y^2 + Q_z^2 - E_r^2) = 0$$

Sejam, pois:

$a = \vec{R}_v \circ \vec{R}_v$
$b = 2\ (\dot{Q} \circ \vec{R}_v)$
$c = \dot{Q} \circ \dot{Q} - E_r^2$

Ora, $a = 1$ sempre, pois $\vec{R}_v$ é um vetor unitário. Logo,

$$\Delta = b^2 - 4c, \quad t_1 = (-b + \sqrt{\Delta})/2 \quad \text{e} \quad t_2 = (-b - \sqrt{\Delta})/2.$$

Com isto, achamos as duas intersecções da semi-reta com a esfera e estamos em condições de escrever os procedimentos de cálculo de interseção para as esferas:

Pré-processamento

As duas únicas contas que não dependem da semi-reta são E_r^2 e $1/E_r$. Calcule-as e armazene-as em variáveis convenientes.

Cálculo da distância percorrida

a) Tendo em conta que E_r^2 foi avaliada no pré-processamento, calcule:

$$\dot{Q} = \dot{R}_p - \dot{E}_p$$
$$b = 2\,(\dot{Q} \circ \vec{R}_v)$$
$$c = \dot{Q} \circ \dot{Q} - E_r^2$$

b) Calcule $\Delta = b^2 - 4c$. Se $\Delta < 0$, não há intersecção e devolvemos $t = -1$. Senão, calcule $t_1 = 0.5(-b + \sqrt{\Delta})$ e $t_2 = 0.5(-b - \sqrt{\Delta})$.

c) Escolha, entre t_1 e t_2, aquele que tiver o menor valor positivo. Esta é a distância percorrida pelo raio até se chocar com a esfera. Este teste pode ser feito com o seguinte trecho de programa em C:

```
t = (0.0 < t2) ? t2 : t1;
```

Este teste funciona, pois temos sempre $t_1 \geq t_2$.

Cálculo do ponto de intersecção e do normal

a) O ponto de intersecção é: $\dot{I} = \dot{R}_p + t\vec{R}_v$

b) Habitualmente, queremos obter o vetor normal que aponte para fora da esfera. Isto porque, na maioria das vezes, olhamos a esfera do lado de fora e, para aplicar o modelo de iluminação de Phong, necessitamos do normal à superfície que aponte para o lado onde está situado o observador. Mas, se por algum motivo, a esfera deve ser vista do lado de dentro (por exemplo, quando temos uma esfera transparente) então o normal deve apontar para dentro da esfera. Pode-se verificar se o observador está dentro da esfera fazendo o seguinte teste: Se $t_1 < 0$ e $0 < t_2$ então o observador está dentro da esfera e o normal é $\vec{N} = versor(\dot{E}_p - \dot{I}) = (\dot{E}_p - \dot{I})\frac{1}{E_r}$. Caso contrário, $\vec{N} = versor(\dot{I} - \dot{E}_p) = (\dot{I} - \dot{E}_p)\frac{1}{E_r}$. Observe que $\frac{1}{E_r}$ foi calculado no pré-processamento.

Eficiência

No pior caso, efetuamos 11 somas/subtrações, 11 multiplicações, 2 comparações e uma raiz quadrada no cálculo da distância percorrida t. [IntrRay] descreve um outro método para calcular t que gasta somente 10 somas/subtrações, 7 multiplicações, 2 comparações e uma raiz quadrada.

Além disso, efetuamos 6 somas/subtrações, 6 multiplicações e 2 comparações no cálculo do ponto de intersecção e do normal.

10.2 Plano

Sejam dados um plano L e um raio R. O plano L é definido por um ponto $\dot{L}_p \in L$ e pelo vetor unitário $\vec{L}_v$ normal a L. O plano L é o conjunto de pontos $\dot{P}$ que satisfazem

$$(\dot{P} - \dot{L}_p) \circ \vec{L}_v = 0 \tag{10.1}$$

O raio R é definido pelo ponto $\dot{R}_p$ que indica a origem do raio e pelo vetor-direção unitário $\vec{R}_v$. É o conjunto de pontos

$$\dot{R}_p + t\vec{R}_v \quad \text{onde} \quad t \geq 0 \tag{10.2}$$

Para achar a intersecção entre L e R, substitua a equação 10.2 na equação 10.1:

$$(\dot{R}_p + t\vec{R}_v - \dot{L}_p) \circ \vec{L}_v = 0$$

Isolando t, obtemos:

$$t = (\dot{L}_p \circ \vec{L}_v - \dot{R}_p \circ \vec{L}_v) \;/\; (\vec{R}_v \circ \vec{L}_v)$$

Vejamos agora, as três fases do cálculo de intersecção entre uma semi-reta e um plano.

Pré-processamento

Calcule $d = -\dot{L}_p \circ \vec{L}_v$. Observe que este d, que acabamos de calcular, é o mesmo da equação de plano $ax + by + cz + d = 0$ e indica a distância da origem do sistema $(0, 0, 0)$ ao plano L. Além disso, os valores a, b e c da equação do plano são respectivamente L_{vx}, L_{vy} e L_{vz}.

Cálculo da distância percorrida

Calcule $\alpha = \vec{R}_v \circ \vec{L}_v$. Se $\alpha = 0$ então o raio R é paralelo ao plano L e não há intersecção entre ambos. Senão, calcule $t = -(d + \dot{R}_p \circ \vec{L}_v)/\alpha$. Se $t < 0$ então a intersecção fica atrás da origem do raio R.

Cálculo do ponto de intersecção e do normal

a) O ponto de intersecção é $\dot{I} = \dot{R}_p + t\vec{R}_v$.

b) O normal $\vec{N}$ é $\vec{L}_v$ se $\alpha < 0$ e é $-\vec{L}_v$ caso contrário. Com isso, o plano pode ser visto de ambos os lados. Se com certeza o plano será visto sempre de um único lado, pode-se eliminar a comparação $\alpha < 0$ e considerar sempre $\vec{N} = \vec{L}_v$ ou $\vec{N} = -\vec{L}_v$, dependendo do lado onde estará o observador.

Eficiência

No cálculo da distância percorrida são efetuadas 5 somas/subtrações, 2 comparações, 6 multiplicações e 1 divisão. No cálculo do ponto de intersecção e do vetor normal são efetuados 3 somas/subtrações, 3 multiplicações e 1 comparação.

10.3 Polígono

Sejam dados um raio $R = (\dot{R}_p, \vec{R}_v)$ e um polígono $G = (\dot{G}_0, \dot{G}_1, \ldots, \dot{G}_{n-1}, \dot{G}_n = \dot{G}_0)$ no espaço. Para calcular a intersecção de R com G, é preciso antes de mais nada, calcular a equação do plano $L = (\vec{L}_v, \dot{L}_p)$ ao qual G pertence. Para calcular o vetor unitário $\vec{L}_v$ normal ao plano L, faça a seguinte conta, sugerida por Martin Newell e citada em [ProcElem] e em [Sutherland, 74]:

$$\begin{cases} x = \sum_{i=0}^{n-1} (y_i - y_{i+1})(z_i + z_{i+1}) \\ y = \sum_{i=0}^{n-1} (z_i - z_{i+1})(x_i + x_{i+1}) \\ z = \sum_{i=0}^{n-1} (x_i - x_{i+1})(y_i + y_{i+1}) \end{cases}$$

Onde (x_i, y_i, z_i) são as coordenadas do vértice $\dot{G}_i$ e $L_v = versor(x, y, z)$ é o vetor normal exato do plano L se o polígono é planar e, caso contrário, fornece um vetor próximo àquele que melhor aproxima o normal. Para calcular o ponto $\dot{L}_p$, podemos pegar qualquer ponto do plano L. Em particular, podemos escolher simplesmente qualquer um dos vértices do polígono. Mas, se teme que se possam aparecer polígonos que não sejam planares, sugerimos que se escolha o seguinte ponto:

$$\begin{cases} L_{px} = (\sum_{i=0}^{n-1} x_i)/n \\ L_{py} = (\sum_{i=0}^{n-1} y_i)/n \\ L_{pz} = (\sum_{i=0}^{n-1} z_i)/n \end{cases}$$

Observe que, como $\vec{L}_v$ e $\dot{L}_p$ são calculados no pré-processamento, podemos gastar "todo o tempo que quisermos" para calculá-los, pois este tempo será desprezível em comparação ao tempo gasto no rastreamento de raio. Em seguida, calculamos a intersecção $\dot{I}$ do raio R com o plano L utilizando o mesmo método já visto na seção anterior. Resta, então, testar se a intersecção $\dot{I}$ pertence ou não ao polígono G. Para isso, o polígono G e a intersecção $\dot{I}$ são projetados num dos três planos definidos pelos três eixos

das coordenadas. Esta projeção, na prática, é simplesmente jogar fora uma das três coordenadas dos vértices de G e do $\dot{I}$. Entre as três coordenadas (x, y, z), jogue fora aquela com o maior valor absoluto em $\vec{L}_v$. Por exemplo, se $\vec{L}_v = (-0.267, 0.535, -0.802)$ então deve-se jogar fora a coordenada z.

Chamemos de $H = (\dot{H}_0, \dot{H}_1, \ldots, \dot{H}_{n-1}, \dot{H}_n = \dot{H}_0)$ o polígono 2D obtido eliminando uma das coordenadas de G da maneira acima descrita. Cada vértice $\dot{H}_i$ de H tem duas coordenadas $\dot{H}_i = (u_i, v_i)$. E seja $\dot{J} = (J_u, J_v)$ o ponto obtido de $\dot{I}$ eliminando uma das suas coordenadas. Então, nós acabamos caindo no teste de pertinência de um ponto a um polígono no plano que já estudamos na seção 5.4.

Pré-processamento

Calcule o plano L, determinando $\vec{L}_v$ e $\dot{L}_p$. Também calcule $d = -\dot{L}_p \circ \vec{L}_v$, qual das três coordenadas deve jogar fora e o polígono H. Se quiser acelerar o teste de pertinência do ponto ao polígono, calcule também a janela limitante, conforme explicamos na subseção 5.4.3.

Cálculo da distância percorrida

Calcule t usando o mesmo método usado para calcular a distância percorrida para plano. Se não há intersecção entre o raio R e o plano L, também não há intersecção entre o raio R e o polígono G. Senão, calcule o ponto de intersecção $\dot{I}$ entre o raio R e o plano L. Projete $\dot{I}$, obtendo $\dot{J}$. Verifique se $\dot{J}$ pertence ao polígono H, fazendo o teste visto na seção 5.4. Se pertence, devolva t como distância percorrida. Senão, não há intersecção raio/polígono e deve-se devolver -1.

Cálculo do ponto de intersecção e do normal

O ponto de intersecção $\dot{I}$ já foi calculado na fase anterior. O vetor normal é calculado usando a mesma técnica utilizada para calcular o normal do plano.

10.4 Cilindro

Para nós, um cilindro será algo como um pedaço de cano. Não será fechado nas extremidades e a espessura da parede será infinitesimal. Sejam dados, pois, um cilindro $C = (\dot{C}_p, \dot{C}_q, C_r)$ e uma semi-reta $R(t) = \dot{R}_p + \vec{R}_v t,\ t \geq 0$. Os pontos $\dot{C}_p$ e $\dot{C}_q$ são os centros das duas "tampas" circulares de C e C_r é o raio do cilindro (veja a figura 10.1). O ponto $\dot{R}_p$ é a origem do raio e $\vec{R}_v$ é a sua direção.

Vamos aplicar uma transformação geométrica T nos pontos $\dot{C}_p$ e $\dot{C}_q$ para deixar o cilindro na sua forma canônica, obtendo o cilindro D (figura 10.1). O cilindro $D = (\dot{D}_p, \dot{D}_q, D_r)$ é, portanto:

$$\begin{cases} \dot{D}_p = T\dot{C}_p = (0, 0, 0) \\ \dot{D}_q = T\dot{C}_q \\ D_r = C_r \end{cases}$$

Onde a matriz de transformação geométrica T é:

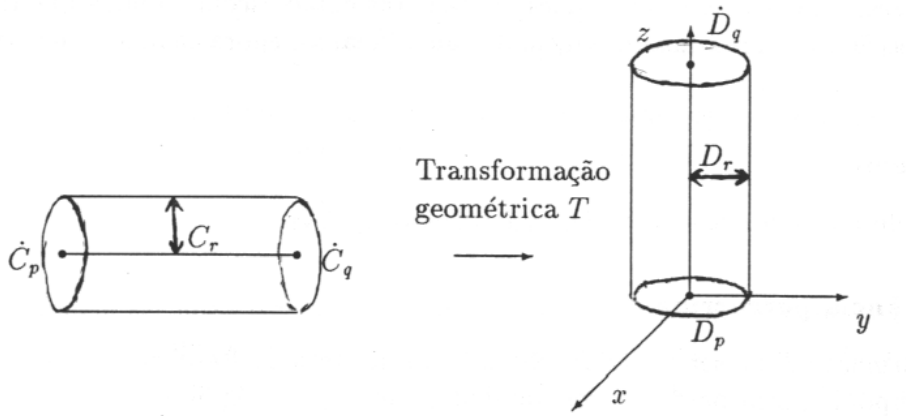


Figure 10.1: Cilindro canônico.

$$T = T_{\dot{C}_p}\, R_{z,-\theta}\, R_{y,-\phi}$$

Os ângulos θ e ϕ são as coordenadas esféricas do vetor $\dot{C}_q - \dot{C}_p$. Aplicando a matriz T também na semi-reta R, obtemos a semi-reta S:

$$T\dot{R}(t) = T\dot{R}_p + T\vec{R}_v t = \dot{S}_p + \vec{S}_v t = \dot{S}(t)$$

Agora, temos que calcular a intersecção entre D e S, que é mais fácil que calcular a intersecção entre C e R. Calcule, pois, aquela intersecção, imaginando que D é um cilindro infinito. A equação do cilindro D, se o imaginarmos infinito, é:

$$x^2 + y^2 - D_r^2 = 0$$

Substituindo a equação da reta $\dot{S}(t)$ na equação do cilindro, obtemos:

$$(S_{px} + S_{vx}t)^2 + (S_{py} + S_{vy}t)^2 - D_r^2 = 0$$

Isolando os termos em função do grau de t, temos:

$$(S_{vx}^2 + S_{vy}^2)t^2 + 2(S_{px}S_{vx} + S_{py}S_{vy})t + (S_{px}^2 + S_{py}^2 - D_r^2) = 0 \tag{10.3}$$

Resolvendo esta equação, podemos achar duas, uma ou nenhuma solução real. Se a equação acima não possui nenhuma raiz então não há intersecção entre a semi-reta R e o cilindro C. Caso contrário,

precisamos descobrir se as raízes da equação representam realmente uma intersecção entre a semi-reta e o cilindro. Uma raiz t representa uma intersecção verdadeira se, e somente se, satisfaz as seguintes condições:

$$\begin{cases} 0 < t \\ 0 \leq S_{pz} + S_{vz}t \\ S_{pz} + S_{vz}t \leq D_{qz} \end{cases}$$

Se a primeira condição não é verdadeira, significa que a intersecção ocorre atrás da origem da semi-reta. Se a segunda ou a terceira condição é falsa, significa que t representa uma intersecção entre a semi-reta e o cilindro infinito mas que esta intersecção não pertence ao cilindro finito. Teste estas condições para cada uma das raízes. Se existirem duas raízes verdadeiras então escolha aquela que tiver o menor valor, pois será a intersecção mais próxima da origem do raio. Vejamos agora cada uma das três fases do cálculo da intersecção:

Pré-processamento

Calcule a matriz de transformação T e $D_{qz} = distancia(\dot{C}_p, \dot{C}_q)$.

Cálculo da distância percorrida

Aplique a transformação T na semi-reta R, obtendo a semi-reta S. Resolva a equação 10.3; sejam t_1 e t_2 as suas raízes e suponha, sem perda de generalidade, que $t_2 \leq t_1$. Aplique o trecho do programa abaixo para calcular a distância percorrida t pelo raio R até se chocar pela primeira vez com o cilindro limitado C.

```
Se (0<t2) entao
  Jz := Spz + Svz * t2;
  Se (0 <= Jz) e (Jz <= Dqz) retorne(t2);
FimSe;
Se (0<t1) entao
  Jz := Spz + Svz * t1;
  Se (0 <= Jz) e (Jz <= Dqz) retorne(t1);
FimSe;
Retorne(-1);
```

Armazene o valor J_z para ser utilizado na próxima fase.

Cálculo do ponto de intersecção e do normal

O ponto de intersecção é: $\dot{I} = \dot{R}_p + t\vec{R}_v$. O vetor normal à superfície é: $\vec{N} = versor(\dot{I} - \dot{C}_p - J_z\vec{C}_v)$.

10.5 Modelo de bolha

Vejamos agora como se pode determinar a intersecção de um raio com um modelo de bolha. Como já vimos na seção 3.7, o modelo de bolha foi introduzido por [Blinn, 82] e consiste em criar uma superfície implícita complexa a partir de somas e subtrações de campos escalares mais simples. Um dos campos escalares mais simples possível é gerada pela função abaixo e o chamaremos de átomo. Um átomo é uma função densidade que decresce exponencialmente com a distância ao centro do átomo:

$$D(x,y,z) = e^{-ar} \ , \ \text{onde} \ r = \sqrt{(x-x_1)^2 + (y-y_1)^2 + (z-z_1)^2} \ \text{e} \ (x_1, y_1, z_1) \ \text{é o centro do átomo.}$$

Os átomos serão o modelo primitivo do modelo de bolha. Podemos representar uma coleção de átomos somando-se as contribuições de cada átomo:

$$D(x,y,z) = \sum_i b_i \, e^{-a_i \, r_i}$$

As constantes b_i podem ser negativas, ou seja, um átomos pode ter um volume negativo. Um átomo com volume negativo é normalmente invisível, mas se um átomo negativo é colocado próximo a um objeto normal, o objeto é "mastigado" pelo átomo negativo. Isto é similar à operação de diferença entre dois sólidos. Para que a função acima possa ser calculada eficientemente no computador, [Blinn, 82] substituiu-a por:

$$D(x,y,z) = \sum_i b_i \, e^{-a_i \, r_i^2}$$

Isto elimina a necessidade de calcular a raiz quadrada. A superfície do modelo de bolha são os pontos do espaço onde a função densidade acima é igual a um valor de disparo T, ou seja, os pontos onde a função F abaixo é igual a zero:

$$F(x,y,z) = D(x,y,z) - T = 0 \tag{10.4}$$

A intersecção da superfície F com um raio de luz pode ser determinada utilizando um método numérico. Para isso, substitua a equação da semi-reta $\dot{R}_p + \vec{R}_v \, t, \ \ t \geq 0$ na função 10.4 obtendo:

$$F(t) = \left(\sum_i b_i \, e^{-a_i \, (\dot{R}_p + \vec{R}_v \, t - \dot{C}_i) \circ (\dot{R}_p + \vec{R}_v \, t - \dot{C}_i)} \right) - T \tag{10.5}$$

onde $\dot{C}_i$ é o centro do i-ésimo átomo. Existem dois métodos numéricos que poderiam ser utilizados para achar as raízes da equação 10.5.

Método de Newton: O método de Newton parte de um "chute" inicial t_1 e iterativamente gera as soluções t_2, t_3, ... cada vez mais próximas da raiz verdadeira. Dada a i-ésima solução t_i, a $(i+1)$-ésima solução pode ser calculada através da fórmula:

$$t_{i+1} = t_i - \frac{F(t_i)}{F'(t_i)}$$

Onde $F'(t_i)$ pode ser calculada algebracamente a partir da equação 10.5. Apesar de não estar citado em [Blinn, 82], o valor aproximado de $F'(t_i)$ também poderia ser obtido numericamente:

$$F'(t_i) \approx \frac{F(t_i + \triangle t) - F(t_i)}{\triangle t}$$

escolhendo $\triangle t$ convenientemente pequeno.

Este método converge muito rapidamente para a raiz da equação mas existem casos em que o método pode divergir. O próximo método tem a garantia de que sempre converge.

Método de busca binária[1]: Este método parte de um intervalo inicial $[t_e, t_d]$ dentro do qual deve estar localizada uma raiz de F. A cada iteração, este intervalo vai-se estreitando pela interpolação linear até que o intervalo seja suficientemente pequeno. Utiliza-se a seguinte fórmula para estreitar o intervalo:

$$t_m = \frac{t_e\, F(t_d) - t_d\, F(t_e)}{F(t_d) - F(t_e)}$$

Se $\text{sinal}(F(t_m)) = \text{sinal}(F(t_d))$ então fazemos $t_d \leftarrow t_m$ senão $t_e \leftarrow t_m$. Este método acha com certeza uma raiz dentro do intervalo $[t_e, t_d]$ mas converge mais lentamente que o método de Newton. Por isso, [Blinn, 82] preferiu utilizar uma solução híbrida que procura aproveitar as vantagens dos ambos métodos.

Método híbrido: Este método calcula o valor t_{i+1} pelo método de Newton. Se este valor estiver fora do intervalo $[t_e, t_d]$ então o valor é recalculado pela fórmula da busca binária. Este processo se repete até que $\mid F(t_{i+1}) \mid$ seja menor que uma tolerância de erro ε.

Cálculo dos intervalos iniciais: Os intervalos iniciais onde pode existir uma raiz são calculados baseados numa heurística que por sua vez se apóia no nosso conhecimento a respeito da função F. Espera-se que a solução esteja próxima da intersecção do raio com os átomos individuais ou esteja próxi ma do máximo local de um átomo, se esse máximo não exceder o valor de disparo T (veja a figura 10.2). Criam-se as listas de "chutes iniciais" e elas são ordenadas em ordem crescente de z. A semi-reta fica então dividida em intervalos iniciais onde uma raiz da função F pode estar localizada.

[1] Regula falsi.

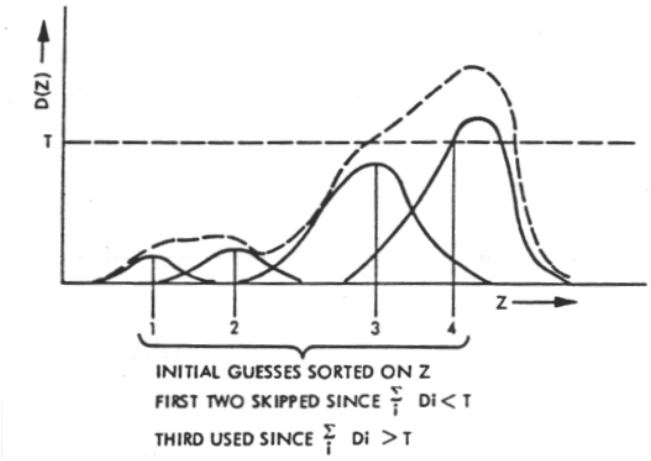


Figure 10.2: Intervalos iniciais (de [Blinn, 82]).

Cálculo garantido da intersecção raio/bolha: [Kalra, 89] descreve um método que garantidamente acha a intersecção mais próxima de um raio com uma superfície implícita qualquer. Este método está baseado nas constantes L e G de Lipschitz ([Kalra, 89] cita [Gear, 71] e [Lin, 74] como referências para estas constantes). As constantes L e G satisfazem as seguintes inequações:

$$\|F(\dot{P}_a) - F(\dot{P}_b)\| \leq L \, \|\dot{P}_a - \dot{P}_b\|$$
$$\|g(t_a) - g(t_b)\| \leq G \, \|t_a - t_b\|$$

onde $g(t) = dF/dt$. Informalmente, podemos dizer que estas duas constantes indicam quão rapidamente a função F pode variar.

[Kalra, 89] descreve um novo algoritmo de intersecção de raio/bolha que utiliza método numérico (de Newton ou de busca binária) mas que tem a garantia de que sempre achará a intersecção mais próxima, qualquer que seja a superfície. Para isto, é necessário conhecer as constantes L e G dessa superfície. [Kalra, 89] também descreve como calcular as constantes L e G para várias superfícies implícitas diferentes.

Cálculo do vetor normal: [Blinn, 82] descreve um método simples para calcular o vetor normal: calcular o gradiente da função F.

$$\vec{N} = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right]$$

Se o cálculo algébrico das derivadas parciais é difícil, uma aproximação numérica também poderia ser utilizada:

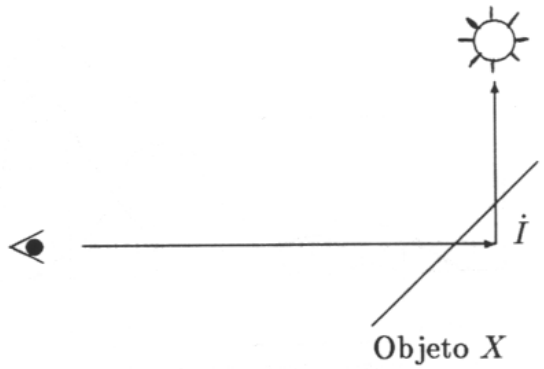


Figure 10.3: Problema de precisão.

$$\vec{N} = \left[\frac{\triangle F}{\triangle x}, \frac{\triangle F}{\triangle y}, \frac{\triangle F}{\triangle z}\right] = \left[\frac{F(x+\triangle x, y, z)}{\triangle x}, \frac{F(x, y+\triangle y, z)}{\triangle y}, \frac{F(x, y, z+\triangle z)}{\triangle z}\right]$$

Onde $\triangle x$, $\triangle y$ e $\triangle z$ devem ser escolhidos convenientemente.

10.6 Problema de Precisão

Todos aqueles que procuram implementar um rastreador de raio acabam encontrando um problema inesperado. O problema aparece devido à precisão finita de números em ponto flutuante e recebe o nome de problema de precisão. Veja a figura 10.3. O raio do observador é lançado do ponto $\dot{E}$ e bate num objeto X no ponto $\dot{I}$. Este ponto $\dot{I}$, pelo problema de precisão, pode estar um pouco abaixo ou um pouco acima da superfície X. Se está abaixo, como na figura 10.3, o rastreador de raio pensará erroneamente que o ponto $\dot{I}$ fica na sombra, quando o raio de sombra for lançado. Problemas semelhantes também ocorrem com os raios de reflexão e de transmissão. Como consequência deste problema, aparece na imagem uma série de pequenos pontos, parecidos com o "ruído" da televisão. [IntrRay] dá uma série de soluções para resolver este problema.

a) O primeiro método é mover o ponto de intersecção $\dot{I}$ para fora e para dentro do objeto X conforme a necessidade. Isto pode ser feito calculando:

$$\dot{I}_f = \dot{I} + \varepsilon\, \vec{N} \quad ou \quad \dot{I}_d = \dot{I} - \varepsilon\, \vec{N}$$

Onde $\vec{N}$ é o vetor normal ao objeto X que aponta para fora, ε é um pequeno número positivo, por exemplo 0.000001 e $\dot{I}_f$ e $\dot{I}_d$ são respectivamente o ponto $\dot{I}$ perturbado para fora e para dentro de

X. Se deseja escolher ε mais "cientificamente", as dimensões dos objetos presentes no ambiente devem ser analisadas e escolher um ε pequeno o suficiente para não alterar a imagem gerada mas grande o suficiente para eliminar o problema de precisão. Como vimos na seção 5.7, basta examinarmos o maior valor absoluto das coordenadas dos pontos para se poder escolher ε. Veja a seção 5.7 ou [Kim, 92] para maiores detalhes. A perturbação deve ser para fora do objeto quando se lançam os raios de sombra e de reflexão e deve ser para dentro quando se lançam os raios de transmissão.

b) O segundo método é fazer com que o programa se lembre o objeto X de onde o raio R está partindo e não permitir que seja aceita como uma solução válida a intersecção de R com o próprio X.

c) Um outro método é ignorar os objetos cujas distâncias à origem do raio sejam menores que um ε dado. [IntrRay, p 47] sugere que, no cálculo da intersecção raio/esfera, se utilize um ε baseado no raio da esfera.

Chapter 11

Técnicas de Aceleração

11.1 Volume Limitante

A técnica de aceleração do rastreamento de raio mais utilizada é o volume limitante. No artigo [Whitted, 80], que introduziu o rastreamento de raio, já havia o conceito de volume limitante. Um volume limitante pode ser definido como segue:

Definição 11.1 *Um* **volume limitante** *é aquele que contém um objeto dado e que permite testar se há uma intersecção de um modo mais simples do que se o teste fosse feito com o próprio objeto.*

Somente há a necessidade de se calcular a intersecção raio/objeto se o raio intersectar o volume limitante do objeto. Pois, caso contrário, o raio não poderá intersectar o objeto que está inteiramente contido no volume. [Whitted, 80] utilizou as esferas como volumes limitantes, observando que elas eram as formas que permitiam calcular a intersecção mais rapidamente.

[Rubin, 80] introduziu a noção de volume limitante hierárquico (VLH[1]) o que fez com que o tempo de processamento para calcular o ponto de colisão caísse de tempo linear no número de objetos para um tempo logarítmico. VLH consiste em colocar uma série de volumes limitantes dentro de um volume maior. Se o raio não intersecta o volume maior, não há necessidade de testar se esse raio intersecta os volumes limitantes filhos, assim como todos os objetos neles contidos. Um volume limitante hierárquico é criado repetindo-se o processo acima. O procedimento IntersecçãoVLH tem o seguinte esqueleto:

```
Procedimento IntersecçãoVLH(raio, nó);
   Se nó é uma folha então
      Intersecção(raio, nó.objeto);
   SenãoSe HáIntersecção(raio,nó.VolumeLimitante) então
      Para cada filho do nó calcule
```

[1] BVH – Bounding Volume Hierarchy.

```
            IntersecçãoVHL(raio, filho);
        FimSe;
    FimProcedimento
```

A estrutura de dados do VLH é uma árvore onde as folhas são os objetos e os nós internos são os volumes limitantes. A criação de VLH para um dado conjunto de objetos não é simples. [Rubin, 80] sugere que esta operação seja feita manualmente, através de um editor próprio. [Goldsmith, 87] descreveu uma técnica para criar automaticamente a estrutura VLH. [Kajiya, 83] descreveu a técnica para construir automaticamente VLH para certos tipos de fractais.

11.2 Subdivisão do Espaço

Uma outra série de técnicas para acelerar o rastreamento de raio está baseada na subdivisão do espaço em cubos (ou paralelepípedos) alinhados aos eixos, denominados de voxels[2]. Um pré-processamento é responsável pela construção de voxels que, somados, constituem um volume que contém todo o ambiente. Para cada voxel é criada uma lista de objetos candidatos que pertencem (total ou parcialmente) àquele voxel. Para se calcular a intersecção raio/objeto mais próxima, calculam-se os voxels por onde esse raio passa e, só são testadas as intersecções do raio com os objetos-candidatos desses voxels. Existem duas técnicas básicas de subdivisão do espaço: subdivisão uniforme e não-uniforme.

11.2.1 Subdivisão espacial uniforme

[Fujimoto, 86] introduziu a técnica de subdivisão espacial uniforme. O espaço é subdividido em voxels de tamanhos iguais. Para cada voxel é criada uma lista de objetos candidatos. Os apontadores para os inícios dessas listas podem ser armazenados numa matriz tridimensional. A criação das listas é facilitada pelo fato de todos os voxels terem o mesmo tamanho. Os voxels por onde o raio passa podem ser calculados de um modo muitíssimo eficiente, utilizando um algoritmo semelhante àquele para desenhar retas na tela (Bresenham ou DDA). [Fujimoto, 86] desenvolveu o algoritmo 3DDDA[3] para esta finalidade. Veja a figura 5.1. O algoritmo 3DDDA deve traçar retas tipo B, pois precisamos identificar todos os voxels por onde o raio passa, sem podermos desprezar os voxels cujas intersecções com o raio são pequenas.

11.2.2 Subdivisão espacial não-uniforme

As técnicas de aceleração do rastreamento de raio pela subdivisão espacial não-uniforme são as que dividem o espaço em regiões de tamanhos variados. Esta variação no tamanho dos voxels permite que mais subdivisões sejam feitas nas regiões densamente povoadas e, por outro lado, cria voxels grandes para cobrir as regiões esparsamente povoadas ou inteiramente vazias. Costumam-se utilizar octrees ou árvores BSP[4] para representar esta subdivisão.

[2] Volume element.
[3] Three-dimensional digital difference analyzer.
[4] Binary space partition trees.

A octree é uma árvore 8-ária utilizada para dividir o espaço em regiões de tamanhos variados. Ela é construída subdividindo recursivamente um volume retangular paralela aos eixos em oito volumes menores, de tamanhos iguais entre si. Esta subdivisão continua até que os volumes resultantes satisfaçam algum critério. [Glassner, 84] introduziu o uso de octree para acelerar o rastreamento de raio. A cada volume é associada uma lista de objetos candidatos que são reconhecidos testando se há intersecção entre as suas superfícies e as seis faces do volume. Se há intersecção, o objeto deve ser adicionado à lista de candidatos do volume. Se o objeto não intersecta nenhuma das faces do volume, escolhe-se um ponto qualquer do objeto e testa se este ponto pertence ao volume, para verificar se o volume contém o objeto totalmente. Também deve-se testar se o objeto contém totalmente o volume. Para isso, escolhe-se um ponto qualquer do volume e verifica se ele pertence ao objeto. O volume é subdividido recursivamente até que cada voxel contenha menos objetos que um número limite. Uma vez construída a octree, o seguinte procedimento calcula a intersecção raio/objeto com o auxílio da octree.

```
Procedimento IntersecçãoOctree(raio);
   Q̇ := raio.origem;
   Repita
      Localize o voxel V que contém Q̇;
      Para cada objeto-candidato do voxel V calcule
         Intersecção(raio, objeto);
      Se nenhuma intersecção foi encontrada então
         Q̇ := um ponto no próximo voxel atravessado pelo raio;
   AtéQue (uma intersecção seja encontrada) ou
      (Q̇ esteja fora do volume limitante do ambiente);
FimProcedimento;
```

Pode-se localizar o voxel V que contém um ponto $\dot{Q}$ fazendo uma busca na octree. Pode-se achar um ponto no próximo voxel atravessado pelo raio calculando o ponto por onde o raio sai do voxel atual e depois movendo esse ponto uma pequena distância para fora do voxel corrente. Esta distância pode ser o comprimento do menor voxel dividido por dois. Cuidados especiais devem ser tomados para lidar com os casos em que o raio sai do voxel corrente atravessando uma aresta ou um vértice.

[Kaplan, 85] utilizou a árvore BSP para acelerar o rastreamento de raio. Uma árvore BSP divide recursivamente o volume em dois volumes menores até chegar aos voxels nas folhas. Esta divisão é feita usando planos ortogonais aos eixos das coordenadas e consequentemente efetua praticamente a mesma subdivisão que o método octree.

[Jansen, 86] introduziu uma nova técnica para achar os voxels que o raio atravessa na árvore BSP. Em vez de achar o próximo voxel calculando um ponto que garantidamente está dentro dele e efetuando uma busca na árvore BSP, esta nova técnica busca recursivamente todos os ramos atravessados pelo raio. Neste método, cada nó é visitado no máximo uma vez. [Jansen, 86] chamou o método anterior de "travessia sequencial" e o novo método de "travessia recursiva". O algoritmo de travessia recursiva está esquematizado abaixo:

```
Procedimento IntersecçãoBSP(segmento, nó);
   Se (segmento é nulo) ou (nó é NIL) então retorne;
   Se (nó é uma folha) então
      Para cada objeto candidato do nó calcule
         Intersecção(segmento,objeto);
   Senão
      Próximo := a parte do segmento cortado pelo no.PlanoDePartição
         que fica mais próximo à origem do raio;
      IntersecçãoBSP(próximo, no.SemiEspaçoMaisPróximo);
      Se (nenhuma intersecção foi encontrada) então
         Distante := a parte do segmento cortado pelo no.PlanoDePartição
            que fica mais distante da origem do raio;
         IntersecçãoBSP(distante, No.SemiEspaçoMaisDistante);
      FimSe
   FimSe
FimProcedimento;
```

Primeiro, calcula-se o segmento de raio que fica dentro do volume limitante que contém todo o ambiente. À medida que a árvore BSP é percorrida, o segmento de raio é cortado recursivamente em segmentos cada vez menores pelos planos de corte da árvore BSP. O segmento mais próximo à origem do raio é processado primeiro. Se neste segmento se encontra uma intersecção, o segmento mais distante é descartado. Caso contrário, o segmento mais distante também é particionado recursivamente. Quando o segmento de raio estiver inteiramente num dos lados do plano de partição, um dos segmentos será vazia, fazendo com que a chamada recursiva termine imediatamente. Este processo, na verdade, percorre a árvore BSP em pré-ordem.

11.3 Técnicas Direcionais

Uma outra série de técnicas utilizadas para acelerar o rastreamento de raio são as técnicas direcionais, que procuram utilizar o fato de que os raios com as origens e as direções próximas irão intersectar provavelmente o mesmo objeto. Um conceito utilizado por todas estas técnicas para discretizar as direções é o cubo direcional.

Definição 11.2 *Um* **cubo direcional** *tem lados de comprimento dois, alinhados aos eixos, e o seu centro coincide com o centro do sistema de coordenadas.*

O cubo direcional é a versão 3D do quadrado direcional utilizado para calcular o pseudo-ângulo (veja a seção 5.3). Vamos denominar as seis faces do cubo direcional de $+X$, $-X$, $+Y$, $-Y$, $+Z$ e $-Z$ (veja a figura 11.1). Cada uma destas faces cobre um ângulo sólido de $2\pi/3$ esteroradianos. O cubo direcional permite representar um vetor-direção 3D em coordenadas 2D. Dado um vetor-direção, podemos construir

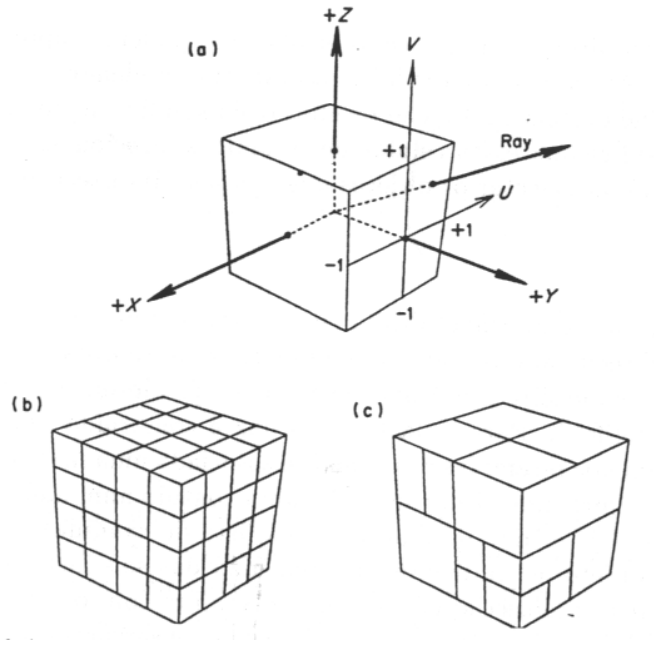


Figure 11.1: Cubo direcional (do livro [IntrRay]). (a) Eixos dominantes. (b) Subdivisão uniforme. (c) Subdivisão adaptativa.

uma representação alternativa determinando qual das faces do cubo direcional ele intersecta e calculando as coordenadas *u-v* do ponto de intersecção. Para determinar qual das faces o vetor-direção irá intersectar, basta procurar o componente do vetor com o maior valor absoluto e verificar o seu sinal. O seguinte procedimento converte um vetor-direção 3D para a representação *u-v*:

```
procedure Direction_to_UV(in direction; out axis, u, v);
begin
  ax:=|direction.x|; ay:=|direction.y|; az:=|direction.z|;
  if (ax>ay) and (ax>az) then begin
    if direction.x>0 then axis:=pos_X else axis:=neg_X;
    u:=direction.y/ax; v:=direction.z/ax;
  end else if (ay>az) then begin
    if direction.y>0 then axis:=pos_Y else axis:=neg_Y;
    u:=direction.x/ay; v:=direction.z/ay;
  end else begin
    if direction.z>0 then axis:=pos_Z else axis:=neg_Z;
    u:=direction.x/az; v:=direction.y/az;
  end;
end;
```

Representando um vetor-direção 3D em coordenadas 2D, podemos aplicar as técnicas de subdivisão no contexto das direções. Assim como na subdivisão espacial, podemos subdividir as direções uniforme ou não-uniformemente. Cada célula de direção resultante da subdivisão do cubo direcional tem o formato de um pirâmide. [IntrRay] chama estas células de pirâmides direcionais. Um raio que pertence a um pirâmide direcional só pode intersectar os objetos que possuem intersecção não nula com esse pirâmide.

11.3.1 Light buffer

O conceito de cubo direcional é aplicável diretamente para os raios que têm a mesma origem. Assim, podemos aplicar o cubo direcional diretamente para os raios de sombra e os raios do observador.

[Haines, 86] utilizou o cubo direcional para acelerar o cálculo de sombra, obtendo o algoritmo que ele denominou de "light buffer". Para cada fonte de luz é construído um cubo direcional subdividido uniformemente e a cada um dos pirâmides direcionais é associada uma lista de objetos-candidatos que têm intersecção não nula com o pirâmide. Um raio de sombra contido num dado pirâmide direcional somente pode ser bloqueado pelos objetos que pertencem à sua lista de candidatos. As listas de candidatos são criadas projetando cada objeto do ambiente em cada uma das seis faces do cubo direcional. Como só há necessidade de calcular a intersecção de um raio de sombra com os objetos-candidatos, o cálculo de intersecção é acelerado. [Haines, 86] faz uma série de outras observações que ajudam acelerar um pouco mais o "light buffer".

Nós podemos aplicar o cubo direcional aos raios do observador. Esta técnica equivaleria, na verdade, à velha técnica de subdividir a tela em janelas e para cada janela construir uma lista de objetos-candidatos.

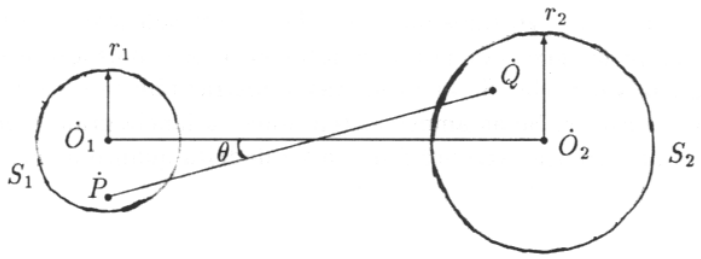


Figure 11.2: Teorema de coerência de raio.

Em vez disso, podemos obter um algoritmo mais geral que pode acelerar o cálculo de intersecção para todos os tipos de raios.

11.3.2 Algoritmo de coerência de raio

[Ohta, 87] conseguiu generalizar o uso da informação direcional para todos os tipos de raios utilizando o teorema de coerência de raio. Este teorema diz:

Teorema 11.1 *Qualquer raio que origina dentro de uma esfera S_1 e termina dentro da esfera S_2 define um ângulo agudo θ com a reta que passa pelos centros das duas esferas que satisfaz a seguinte inequação, segundo [IntrRay]:*

$$cos\,\theta\ > \sqrt{1 - \frac{r_1 + r_2}{d}}$$

onde $d = \parallel \dot{O}_1 - \dot{O}_2 \parallel$. Mas refizemos as contas e obtivemos uma inequação diferente:

$$cos\,\theta\ > \sqrt{\frac{d^2}{(r_1 + r_2)^2 + d^2}}$$

No algoritmo de coerência de raio, um cubo direcional é criado para cada entidade de onde pode se originar um raio. Estas entidades são a posição do observador, fontes de luz e os objetos com reflexão especular ou transparentes. Os cubos direcionais são divididos uniformemente e para cada pirâmide direcional resultante é criada uma lista de objetos-candidatos. Nesta lista pertencem todos os objetos que um raio cuja direção pertence ao pirâmide direcional poderia atingir, qualquer que seja a origem desse raio dentro da esfera limitante da entidade. Para criar esta lista é utilizado o teorema de coerência de raio. Para cada objeto dentro da lista de candidatos calcula-se o limite inferior d da distância do objeto até a entidade. Depois, a lista é ordenada pelos valores d. As intersecções de um raio com os objetos-candidatos são calculadas na ordem crescente de d até que tenha efetuado o teste com todos os objetos-candidatos ou a distância de uma intersecção conhecida até a entidade seja menor que o limite inferior d do próximo objeto a ser testado.

11.4 Eliminar Raios Desnecessários.

Uma outra série de técnicas para acelerar o rastreamento de raio consiste em não lançar aqueles raios que se sabe que contribuirão pouco para a cor final do pixel.

A primeira dessas técnicas é a denominada "controle adaptativa da altura da árvore" introduzida por [Hall, 83]. Em vez da recursão do rastreamento de raio terminar numa profundidade pré-definida ou numa superfície não-reflexiva e opaca, o novo critério de parada da recursão leva em conta a quantidade de contribuição do raio à cor final do pixel. Esta técnica foi utilizada no programa RAIOS. Ajustando o nível de disparo, pode-se eliminar muitos raios sem que a alteração do resultado seja perceptível.

Uma outra técnica que se enquadra nesta categoria é a técnica de otimização estocástica, descrita em [Cook, 86], [Kajiya, 86] e [Lee, 85]. Esta técnica aplica-se principalmente ao rastreamento distribuído e ao rastreamento anti-disfarce. No rastreamento distribuído, podemos obter resultados arbitrariamente precisos aumentando o número de amostras. A otimização estocástica evita que as amostras desnecessárias sejam coletadas. Por exemplo, através da otimização estocástica podemos calcular o número de amostras para obter a estimativa da cor do pixel que difira 5% da cor verdadeira com a probabilidade 90%, sem "chutar" uma solução como "20 raios por pixel devem ser suficientes". Além disso, mais amostras são coletadas somente naqueles pixels onde isso é necessário. A otimização estocástica calcula, além da média das amostras, a sua variância. Se as primeiras amostras têm praticamente o mesmo valor, não é necessário coletar mais amostras. Se as amostras têm uma variância grande, mais amostras deverão ser coletadas a fim de calcular a solução tão próxima da verdadeira quanto queiramos.

Chapter 12

Efeitos Especiais

12.1 Rastreamento de Raio Distribuído

[Cook, 84] introduziu o rastreamento de raio distribuído. Com este algoritmo, tornou-se possível obter alguns efeitos visuais impossíveis de serem obtidos pelo algoritmo original de rastreamento de raio, tais como:

1. Eliminação do "problema de disfarce[1]". O rastreamento de raio original gera arestas em forma de escada devido ao problema de disfarce. Este efeito pode ser eliminado se os pixels das bordas dos objetos receberem uma cor média das cores dos objetos que cobrem este pixel. Esta média deve ser ponderada, colocando pesos proporcionais às áreas do pixel cobertas por cada um dos objetos.

2. Borrão de movimento[2]. Se criarmos uma animação utilizando o rastreamento de raio, aparece o "efeito estroboscópico". Este problema aparece porque o algoritmo considera que a lente da máquina fotográfica fica aberta um intervalo de tempo infinitesimal. Para eliminá-lo, devemos considerar que a lente fica aberta durante um intervalo de tempo finito.

3. Objetos fora de foco[3]. O rastreamento de raio original gera as imagens onde todos os objetos estão perfeitamente enfocados. Isto acontece por utilizar o modelo de máquina fotográfica de papelão, onde se considera que a lente tem uma área infinitésima. Para gerar os objetos fora de foco, devemos considerar que a lente da máquina fotográfica (ou córnea do olho do observador) possui uma área finita.

4. Penumbra. As imagens geradas pelo rastreamento de raio possuem sombras com contornos perfeitamente delimitados, pois considera que a fonte de luz se reduz a um ponto sem dimensões no espaço. Se a fonte de luz tem volume, aparecerão penumbras.

[1] Aliasing.
[2] Motion blur, também chamado de "temporal anti-aliasing" ou "eliminação do efeito estroboscópico".
[3] Depth of field.

5. Translucência. Os objetos transparentes gerados pelo algoritmo de [Whitted, 80] não apresentam translucência. Para criar este efeito, deve considerar que um objeto translúcido transmite os raios de luz também para as direções um pouco diferentes daquela calculada pela lei de Snell.

6. Reflexão nublada. É gerado fazendo a mesma consideração da translucência para reflexão.

O rastreamento de raio distribuído tem-se mostrado como o único algoritmo capaz de gerar de um modo elegante e correto todos estes efeitos. Mas eles são conseguidos às custas de gastar um grande tempo computacional, algumas vezes maior que o algoritmo original de rastreamento. Onde o rastreamento tradicional lançaria um único raio, o rastreamento distribuído lança vários, distribuídos no espaço e/ou no tempo. Para criar os efeitos acima, não é necessário lançar mais raios do que seriam necessários para eliminar o "efeito disfarce", pois esses mesmos raios podem ser utilizados para criar todos os efeitos mencionados, se os distribuirmos convenientemente.

- Distribuindo os raios de reflexão, obtém-se a reflexão especular nublada[4]. Se o número dos raios de reflexão for suficientemente grande, conseguimos simular a reflexão difusa e obter imagens semelhantes àquelas geradas pelo algoritmo de radiosidade.

- Distribuindo os raios de transmissão obtemos a translucência.

- Distribuindo os raios de sombra, obtemos penumbras e estamos simulando uma fonte de luz não pontual.

- Distribuindo os raios do observador sobre a lente da máquina fotográfica (ou sobre a córnea) obtemos os objetos fora de foco.

- Distribuindo os raios no tempo obtemos o borrão de movimento.

12.1.1 Modelo de iluminação usado no rastreamento distribuído

O modelo de iluminação de Phong, apresentado como um modelo local em 7.1.4 e como um modelo global em 9.1, pode ser ainda melhorado. Nesta subseção, apresentaremos o modelo de iluminação de Phong melhorado, utilizado no rastreamento distribuído.

A intensidade I de luz refletida num ponto da superfície é uma integral, sobre a hemisfera de raio unitário que cobre a superfície, da função de iluminação L e da função de reflexão R.

$$I(\phi_r, \theta_r) = \int_{\phi_i=0}^{\pi/2} \int_{\theta_i=0}^{2\pi} L(\phi_i, \theta_i)\ R(\phi_i, \theta_i, \phi_r, \theta_r)\ d\phi_i\, d\theta_i \tag{12.1}$$

[4]Estamos procurando distinguir a reflexão especular nublada da reflexão difusa. A reflexão especular nublada é obtida quando a superfície onde se dá a reflexão não é perfeitamente polida. Como resultado, a reflexão especular fica "embaçada", "nublada" ou "pouco nítida", mas continua tendo uma direção de reflexão. Na reflexão difusa, não existe mais a direção de reflexão, pois a reflexão se dá com a mesma intensidade em todas as direções.

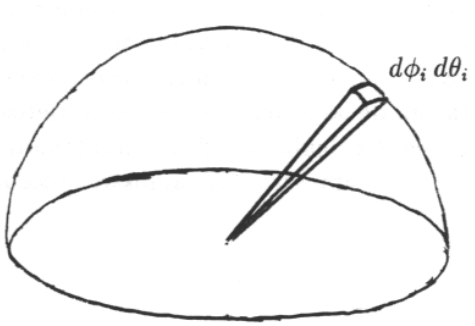


Figure 12.1: hemisfera e ângulo sólido

Onde (ϕ_i, θ_i) é o ângulo de incidência e (ϕ_r, θ_r) é o ângulo de reflexão. Veja a figura 12.1. Para certificar-se de que o modelo acima descrito é uma extensão do modelo Phong, podemos fazer as seguintes considerações:

- Suponha que L é zero exceto para as direções das fontes de luz. Então, a integral é substituída pela somatória e obtemos os termos I_s e I_d do modelo de iluminação de Phong. Esta simplificação leva a gerar imagens com sombras nítidas, sem penumbras.
- Suponha que $L(\phi_i, \theta_i)$ seja uma constante para todas as direções que não sejam as direções das fontes de luz. Então, L é independente dos ângulos ϕ_i e θ_i e pode ser tirado fora da integral. Isto nos dá o termo I_a, isto é, a intensidade de luz ambiente do modelo de Phong.
- Suponha que a função de reflectância R é zero exceto para a direção reflexão. Isto causa reflexões nítidas e dá o termo I_r do modelo de Phong global. A suposição correspondente para a luz transmitida causa refração nítida.

Ora, a integral acima é complexa demais para calculá-la analiticamente. Mas podemos amostrar alguns pontos e calcular uma aproximação da integral acima. Com isso, evitamos fazer as simplificações na equação que levariam a cair no modelo de iluminação de Phong.

12.1.2 Reflexão nublada

O rastreamento de raio cria as imagens com as reflexões especulares nítidas, como num espelho. Porém, na realidade, as reflexões são frequentemente nubladas. Qualquer simulação analítida deste fenômeno deve estar baseada no cálculo da integral sobre algum ângulo sólido. O rastreamento de raio cônico ([Amanatides, 84]) é uma técnica para calcular uma aproximação desta integral. Porém, a aplicação do rastreamento cônico para qualquer tipo de objeto é dificultada pela necessidade de calcular, além da intersecção e do normal, a fração do ângulo sólido do raio cônico bloqueada por cada um dos objetos. O

algoritmo de radiosidade ([Goral, 84]) é uma outra técnica para calcular analiticamente uma aproximação dessa integral mas tem a desvantagem de só funcionar para os objetos perfeitamente difusos.

O rastreamento distribuído consegue calcular a reflexão nublada lançando vários raios de reflexão. A integral é aproximada pela somatória das amostras. As amostras recebem pesos diferentes de acordo com a função de reflexão R.

12.1.3 Translucência

A luz transmitida através de um objeto é descrita por uma equação similar à equação 12.1 mas substituindo a função de reflectância R pela função de transmissão T e a integral é calculada sobre a hemisfera abaixo da superfície.

O rastreamento de raio clássico consegue calcular a transparência mas não a translucência. O rastreamento de raio distribuído calcula a translucência lançando vários raios de transmissão. A função de transmissão T determina como os raios devem ser distribuídos sobre a hemisfera e qual deve ser o peso de cada raio.

12.1.4 Penumbra

A penumbra ocorre quando a fonte de luz é obscurecida parcialmente. Este fenômeno só pode acontecer quando a fonte de luz é extensa no espaço. O rastreamento distribuído calcula a penumbra lançando vários raios de sombra, cada um em direção a algum ponto diferente dentro da fonte de luz. As amostras são somadas depois de receberem pesos diferentes, de acordo com a área projetada e o brilho das diferentes partes da fonte de luz.

12.1.5 Objetos fora de foco

Todos os algoritmos de síntese de imagem que vimos até agora estão baseados no modelo de máquina fotográfica de papelão[5]. Veja a figura 12.2. Em outras palavras, todos esses algoritmos de síntese de imagem geram imagens onde todos os objetos estão perfeitamente enfocados. Ora, tanto a máquina fotográfica verdadeira como o olho humano têm uma abertura de lente finita (e não uma abertura infinitesimal como na máquina fotográfica de papelão) e por isso elas geram imagens onde alguns objetos estão fora de foco. Os objetos fora de foco são normalmente indesejáveis, mas às vezes queremos obtê-los, por exemplo, para destacar um objeto do resto.

[Potmesil, 82] descreve uma técnica para gerar os objetos fora de foco. Nela, primeiro é gerada uma imagem utilizando o modelo de máquina de papelão. Depois esta imagem é pós-processada para criar os objetos fora de foco. Este tipo de pós-processamento não consegue gerar as imagens completamente corretas, pois a visibilidade é calculada num único ponto, o centro da lente. A visibilidade dos objetos muda de acordo com a parte da lente a partir de onde se olham os objetos e isto não pode ser levada em conta por um pós-processamento. O rastreamento de raio distribuído resolve este problema lançando, em

[5]Pinhole camera model.

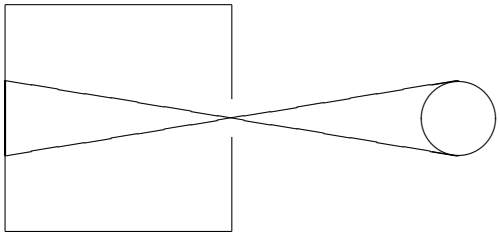


Figure 12.2: Modelo de máquina fotográfica de papelão

vez de um único raio do observador para cada pixel, vários raios distribuídos ao longo da área da lente. Isto resolve também o problema de disfarce, pois ao tirar várias amostras para cada pixel, o problema é atenuado.

12.1.6 Borrão de movimento

Distribuindo os raios no tempo, resolve-se o problema de borrão de movimento. Este efeito é bastante desejável para se gerar uma animação de qualidade. Para gerá-lo, só é necessário saber calcular a posição dos objetos no tempo. As mudanças na visibilidade, a intensidade luminosa do objeto em movimento, as sombras, as reflexões, os objetos fora de foco serão todos corretamente borrados, mesmo que a curva do movimento seja arbitrariamente complexa.

12.2 Rastreamento Bidirecional

O rastreamento bidirecional é um algoritmo inédito que consegue gerar as sombras verdadeiras de objetos transparentes. Basta olharmos para a sombra de um copo ou de um cinzeiro de vidro para verificarmos que as suas sombras são bastante complexas. O rastreamento bidirecional consegue trabalhar com qualquer objeto cuja imagem possa ser gerada por um rastreador de raio convencional. O tempo gasto pelo novo algoritmo é da mesma ordem de um rastreador convencional, mas necessita de uma grande quantidade de memória, da mesma forma que o z-buffer. O autor também conseguiu desenvolver uma técnica que minimiza o uso de memória quando as sombras dos objetos transparentes ocupam uma pequena área da tela. No ambiente podem estar presentes mais de uma fonte de luz e as técnicas de aceleração do rastreamento de raio podem ser aplicadas neste novo algoritmo sem quaisquer alterações.

A idéia básica do algoritmo consiste em lançar os raios em dois sentidos: das fontes para os objetos e do observador para os objetos. Já vimos que lançar os raios das fontes para os objetos é extremamente

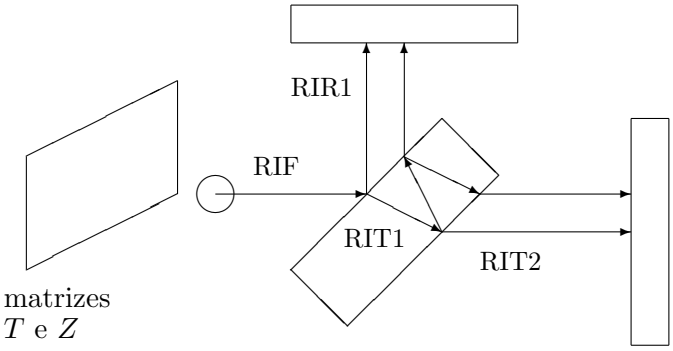


Figure 12.3: Os raios inversos e as matrizes T e Z.

caro computacionalmente, pois muito pouco desses raios chegarão até o observador. Por isso, somente utilizaremos este recurso para aqueles raios que não podem ser traçados em sentido contrário: os que atravessam um objeto transparente. Além disso, procuraremos tirar o maior número de informações possíveis desses raios, mesmo que eles não cheguem a atingir o observador. Os outros tipos de raios serão traçados em sentido convencional: do observador para os objetos. Portanto, o rastreamento bidirecional pode ser dividido em duas fases bem distintas:

Fase 1: Cálculo das sombras dos objetos transparentes.

Fase 2: Rastreamento de raio convencional utilizando as informações geradas pela fase 1.

12.2.1 Fase 1: Cálculo das sombras dos objetos transparentes

Nesta fase, calculamos as sombras dos objetos transparentes e as armazenamos em duas matrizes, que chamaremos de Z e de T (veja a figura 12.3). O número de elementos dessas duas matrizes deve ser igual ao número de elementos da tela do computador. A matriz T pode ser armazenada na própria tela do computador ou numa memória à parte e representará a intensidade das sombras dos objetos transparentes. Cada elemento da matriz Z é um número em ponto flutuante e irá desempenhar o mesmo papel da matriz Z em z-buffer: armazena a profundidade dos pontos.

O cálculo das sombras de objetos transparentes é feito lançando os raios das fontes de luz em direção aos objetos transparentes. Esses raios serão chamados de raios inversos. Assim, podemos ter raio inverso da fonte (RIF), raio inverso de reflexão (RIR) e raio inverso de transmissão (RIT). Pode-se criar um volume limitante para cada objeto transparente e lançar somente os RIFs que com certeza intersectarão o volume limitante. Se algum RIF não intersectar o objeto transparente, apesar de intersectar o volume limitante, será desprezado. A trajetória de um RIF cria uma árvore binária, do mesmo modo que um raio do observador. As arestas desta árvore binária são os raios inversos e os nós são os objetos que os

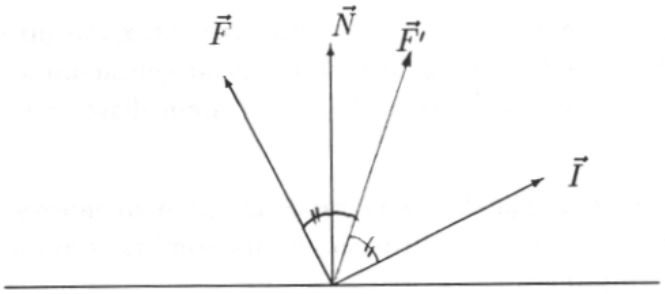


Figure 12.4: Modelo de iluminação na fase 1 do rastreamento bidirecional.

raios inversos intersectam (onde os nós internos são os objetos transparentes e as folhas são os objetos opacos). Um ramo da árvore de raio inverso deve ser podado quando a intensidade do raio for muito fraca ou se o raio se perder no espaço. Abaixo, pode ser vista uma representação da árvore binária criada por um raio inverso.

RIF
  RIR1
    RIR2
    RIT2
  RIT1
    RIR2
      RIR3
      RIT3
    RIT2
      RIR3
      RIT3

Antes de começarmos o processamento da fase 1, devemos preencher a matriz Z com $+\infty$ e a matriz T com a cor de fundo. Depois, lançamos os raios inversos. Quando um raio inverso atingir um objeto opaco num ponto $\dot{P} = (P_x, P_y, P_z)$, podemos ter um dos três casos a seguir:

Caso 1: Se $P_z > Z[P_x, P_y] + \varepsilon$ nada é feito. Pois existe um objeto opaco mais próximo ao observador do que o ponto $\dot{P}$.

Caso 2: Se $P_z < Z[P_x, P_y] - \varepsilon$ então o ponto $\dot{P}$ está mais próximo do observador que aquele ponto armazenado nas matrizes Z e T. O modelo de iluminação é então aplicado ao ponto $\dot{P}$ e a intensidade assim calculada é armazenada na matriz $T[P_x, P_y]$. Além disso, a profundidade P_z é armazenada em $Z[P_x, P_y]$.

Caso 3: Se $Z[P_x, P_y] - \varepsilon \leq P_z \leq Z[P_x, P_y] + \varepsilon$ então o raio inverso chocou-se num ponto já atingido por um outro raio inverso. Neste caso, a intensidade da sombra é aumentada, fazendo: $T[P_x, P_y] \leftarrow T[P_x, P_y]$ + modelo de iluminação.

Podemos usar a idéia de [Norton, 82] para diminuir a quantidade de raios inversos necessários. Isto é, nos casos 2 e 3, em vez de preenchermos somente $Z[P_x, P_y]$ e $T[P_x, P_y]$, também preencheremos os oito pixels vizinhos de (P_x, P_y) nas matrizes Z e T. Sugerimos o seguinte modelo de iluminação para os raios inversos (veja a figura 12.4).

$$I = K_s\, F\, (\vec{N} \circ \vec{F}')^e + K_d\, F\, (\vec{N} \circ \vec{F})$$

Onde F e $\vec{F}$ são respectivamente a intensidade[6] e a direção do raio inverso e $\vec{F}'$ é o meio caminho entre $\vec{F}$ e $\vec{I}$. As outras siglas são as mesmas do modelo de iluminação de Phong. Não é necessário verificar se entre o ponto $\dot{P}$ e o observador existe algum obstáculo, pois isto será testado na fase 2.

Utilizando a técnica de espalhamento[7], podemos economizar a memória quando as sombras dos objetos transparentes ocupam uma área reduzida da tela do computador. Para isso, as matrizes Z e T devem ser substituídas por uma tabela de espalhamento H, indexada por P_x e P_y. No começo da fase 1, a tabela H deve estar vazia. Quando um raio inverso atingir um objeto transparente, procure a entrada da tabela H com a chave (P_x, P_y). Se não existe nenhum ponto com a chave (P_x, P_y) então temos o caso 2 (veja os três casos da página 175). Senão, suponha que a i-ésima entrada da tabela H contém a chave (P_x, P_y). Então:

Se $P_z > H[i].z + \varepsilon$ então temos o caso 1.
Se $P_z < H[i].z - \varepsilon$ então temos o caso 2.
Se $H[i].z - \varepsilon \leq P_z \leq H[i].z + \varepsilon$ então temos o caso 3.

12.2.2 Fase 2: Sombreamento dos objetos.

Esta segunda fase constitui um rastreamento de raio normal. A única diferença é que o modelo de iluminação deve levar em consideração as matrizes T e Z calculadas na fase anterior. Suponha que queremos aplicar o modelo de iluminação a um ponto $\dot{P}$ que pertence ao objeto X.

- Se $P_z < H[i].z - \varepsilon$ então utilize o modelo de iluminação de um rastreador convencional, pois a sombra do objeto transparente estará atrás do ponto $\dot{P}$.

- Se $H[i].z - \varepsilon \leq P_z \leq H[i].z + \varepsilon$ então a intensidade do pixel é: $I =$ modelo de iluminação + $K_b\, T[P_x, P_y]$. Onde K_b é uma constante que depende do número de raios inversos lançados e do ângulo sólido formado por esses raios.

[6] Lembre-se de que um raio inverso perde parte da sua intensidade ao atravessar um objeto transparente ou ao se subdividir em raios de reflexão e de transmissão.

[7] Hashing.

- Se $P_z > H[i].z + \varepsilon$ então provavelmente uma borda do objeto X está localizada no pixel $\dot{P}$. Podemos aplicar só o modelo de iluminação mas lembrando que estaremos gerando o efeito disfarce neste pixel.

Com isso, acabamos a descrição do rastreamento bidirecional. Este algoritmo ainda não foi implementado. Pretendemos implementá-lo tão logo possível.

Chapter 13

Referências Bibliográficas

[Algorithms] Sedgewick, R. *Algorithms.* Addison-Wesley, 1983.

[Amanatides, 84] Amantides, J. *Ray Tracing with Cones.* Computer Graphics, Vol. 18, No. 3, July 1984, pp 129–135.

[Appel, 68] Appel, A. *Some Techniques for Shading Machine Rendering of Solids.* AFIPS 1968 Spring Joint Comput. Conf., pp 37–45.

[Arvo, 87] Arvo, J. and Kirk, D. *Fast Ray Tracing by Ray Classification.* Computer Graphics, Vol. 21, No. 4, July 1987, pp 55–64.

[Atherton, 83] Atherton, P. R. *A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry.* Computer Graphics, Vol. 17, No. 3, July 1983, pp 73–82.

[Barnsley, 88] Barnsley, M. F. and Sloan, A. D. *A Better Way to Compress Images.* Byte, January 1988, pp 215–223.

[Blinn, 82] Blinn, J. F. *Lighr Reflection Functions for Simulation of Clouds and Dusty Surfaces.* Computer Graphics, Vol. 16, No. 3, July 1982, pp 21–29.

[CalcNum, 91] *Exercício Programa de Cálculo Numérico.* Escola Politécnica, todas as turmas, segundo semestre de 1991.

[Catmull, 75] Catmull, E. *Computer Display of Curved Surfaces.* Proc. IEEE Conf. Comput. Graphics Pattern Recognition Data Struct., May 1975, p. 11.

[ConcBas] Gomes, J. M. e Velho, L. C. *Conceitos Básicos de Computação Gráfica.* VII Escola de Computação, São Paulo, 1990.

[Cook, 84] Cook, R. L., Porter, T. and Carpenter, L. *Distributed Ray Tracing.* Computer Graphics, Vol. 18, No. 3, July 1984, pp 137–145.

[Cook, 86] Cook, R. L. *Stochastic Sampling in Computer Graphics.* ACM Trans. Graph., 5(1), January 1986, pp 51–72.

[CProg] Kernighan, B. W. and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, 1978.

[Fournier, 82] Fournier, A., Fussel, D. and Carpenter, L. *Computer Rendering of Stochastic Models.* Comm. of the ACM, June 1982, Vol. 25, No. 6, pp 371–384.

[Fujimoto, 86] Fujimoto, A., Tanaka, T. and Iwata, K. *ARTS: Accelerated Rau-Tracing System.* IEEE Comput. Graph. Appl., 6(4), April 1986, pp 16–26.

[Gear, 71] Gear, C. W. *Numerically Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, Englewood Cliffs, NJ, 1971.

[Glassner, 84] Glassner, A. S. *Space Subdivision for Fast Ray Tracing.* IEEE Comput. Graph. Appl., 8(3), March 1988, pp 60–70.

[Goldsmith, 87] Goldsmith, J. and Salmon, J. *Automatic Creation of Object Hierarchies for Ray Tracing.* IEEE Comput. Graph. Appl., 7(5), May 1987, pp 14–20.

[Goral, 84] Goral, C. M., Torrance, K. E., Greenberg, D. P. and Battaile, B. *Modeling the Interactions of Light Between Diffuse Surfaces.* Computer Graphics, Vol. 18, No. 3, July 1984, pp 213–222.

[Gouraud, 71] Gouraud, H. *Computer Display of Curved Surfaces.* Doctoral thesis, University of Utah, 1971. A condensed version is given in IEEE Trans. Comp., Vol. C-20, 1971, pp 623–628.

[Haines, 86] Haines, F. A. and Greenberg, D. P. *The Light Buffer: A Shadow Testing Accelerator.* IEEE Comput. Graph. Appl., 6(9), September 1986, pp 6–16.

[Hall, 83] Hall, R. A. and Greenberg, D. P. *A Testbed for Realistic Image Synthesis.* IEEE Comput. Graph. Appl., 3(10), November 1983, pp 10–20.

[Hanrahan, 83] Hanrahan, P. *Ray Tracing Algebraic Surfaces.* Computer Graphics, Vol. 17, No. 3, July 1983, pp 83–90.

[Heckbert, 84] Heckbert, P. S. and Hanrahan, P. *Beam Tracing Polygonal Objects.* Computer Graphics, Vol. 18, No. 3, July 1984, pp 119–127.

[IntrRay] Glassner, S. A. *An Introduction to Ray Tracing.* Academic Press, 1989.

[Jansen, 86] Jansen, F. W. *Data Structures for Ray Tracing.* In Data Structures for Raster Graphics, Proceedings Workshop. Eurographics Seminars, Spring Verlag , 1986, pp 57–73.

[Kajiya, 83] Kajiya, J. T. *New Techniques for Ray Tracing Procedually Defined Objects.* Computer Graphics, Vol. 17, No. 3, July 1983, pp 91–102.

[Kajiya, 84] Kajiya, J. T., Herzen, B. P. V. *Ray Tracing Volume Densities.* Computer Graphics, Vol. 18, No. 3, July 1984, pp 165–174.

[Kajiya, 86] Kajiya, J. T. *The Rendering Equation.* Computer Graphics, Vol. 20, No. 4, August 1986, pp 143–150.

[Kalra, 89] Kalra, D. and Barr, A. H. *Guaranteed Ray Intersections with Implicit Surfaces.* Computer Graphics, Vol. 23, No. 3, July 1989, pp 297–306.

[Kamada, 87] Kamada, T. and Kawai, S. *An Enhanced Treatment of Hidden Lines.* ACM Transactions on Graphics, Vol. 6, No. 4, October 1987, pp 308–323.

[Kaplan, 85] Kaplan, M. R. *Space Tracing a Constant Time Ray Tracer.* State of the Art in Image Synthesis (Siggraph'85 Course Notes), Vol. 11, July 1985.

[Kim, 92] Kim, H. Y. *Como Calcular a Probabilidade de Falha do Método de Perturbação.* Anais da Quarta Semana de Informática da UFBA, abril de 1992, pp 13–20.

[Knuth, 87] Knuth, D. E. *Digital Halftones by Dot Diffusion* ACM Transactions on Graphics, Vol. 6, No. 4, October 1987, pp 245–273.

[Lee, 85] Lee, M., Redner, R. A. and Uselton, S. P. *Statiscally Optimized Sampling for Distributed Ray Tracing.* Computer Graphics, Vol. 19, No. 3, July 1985, pp 61–67.

[Lin, 74] Lin, C. C. and Segel, L. A. *Mathematics Applied to Deterministic Problems in the Nature Science.* Macmillan Publishing Co., Inc., New York.

[MathElem] Rogers, D. F. and Adams, J. A. *Mathematical Elements for Computer Graphics.* McGraw-Hill, 1976.

[Muraki, 91] Muraki, S. *Volumetric Shape Description of Range Data Using "Blobby Model".* Computer Graphics, Vol. 25, No. 4, July 1991, pp 227–235.

[Nishimura, 85] Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I. and Omura, K. *Object Modeling by Distribution Function and a Method of Image Generation (in Japanese).* Proc. Eletronics Communication Conference, J68–D(4), 1985.

[Norton, 82] Norton, A. *Generation and Display of Geometric Fractals in 3-D.* Computer Graphics, Vol. 16, No. 3, July 1982, pp 61–67.

[Ohta, 87] Ohta, M. and Maekawa, M. *Ray Coherence Theorem and Constant Time Ray Tracing Algorithm.* Computer Graphics 1987 (Proc. of CG International'87), pp 303–314.

[Overhauser, 68] Overhauser, A. W. *Analytic Definition of Curves and Surfaces by Parabolic Blending.* Technical Report No. SL68–40, Ford Motor Company Scientific Laboratory, May 8, 1968.

[Phong, 75] Phong, B. T. *Illumination for Computer Generated Images.* Doctoral thesis, University of Utah, 1973. A condensed version is given in CACM, Vol. 18, 1975, pp 311–317.

[Potmesil, 82] Potmesil, M. and Chakravarty, I. *Synthetic Image Generation with a Lens and Aperture Camera Model.* ACM Trans. Graph., Vol. 1, No. 2, April 1982, pp 85–108.

[ProgLin] Humes, C. e Humes, A. F. P. C. *Programação Linear – Um Primeiro Curso.* IX Congresso Nacional de Matemática Aplicada e Computacional, 1986.

[ProgPrinc] Ammeraal, L. *Programming Principles in Computer Graphics.* John Wiley & Sons, 1986.

[ProcElem] Rogers, D. F. *Procedural Elements for Computer Graphics.* McGraw-Hill, 1985.

[Reeves, 83] Reeves, W. T. *Particle System – A Technique for Modeling a Class of Fuzzy Objects.* Computer Graphics, Vol. 17, No. 3, July 1983, pp 359–376.

[Requicha, 80] Requicha, A. A. G. *Representations for Rigid Solids: Theory, Methods, and Systems.* ACM Computing Surveys, Vol. 12, No. 4, December 1980, pp 437–464.

[Rubin, 80] Rubin, S. M. and Whitted, T. *A 3-Dimensional Representation for Fast Rendering of Complex Scenes.* Computer Graphics, Vol. 14, No. 3, July 1980, pp 110–116.

[Sederberg, 84] Sederberg, T. W. and Anderson, D. C. *Ray Tracing of Steiner Patches.* Computer Graphics, Vol. 18, No. 3, July 1984, pp 159–164.

[Shinya, 87] Shinya, M., Takahashi, T. and Naito, S. *Principles and Applications of Pencil Tracing.* Computer Graphics, Vol. 21, No. 4, July 1987, pp 45–54.

[SolidMod] Mäntylä, M. *An Introduction to Solid Modeling.* Computer Science Press, 1988.

[Sutherland, 74] Sutherland, I. E., Sproull, R. F. and Schumacker, R. A. *A Characterization of Ten Hidden-Surface Algorithms.* ACM Computing Surveys, Vol. 6, No. 1, March 1974, pp 1–55.

[Velho, 91] Velho, L. and Gomes, J. M. *Digital Halftoning with Wpace Filling Curves.* Computer Graphics, Vol. 25, No. 4, July 1991, pp 81–90.

[Wallace,87] Wallace, J. R., Cohen, M. F. and Greenberg, D. P. *A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods.* Computer Graphics, Vol. 21, No. 4, July 1987, pp 311–320.

[Warnocker, 68] Warnock, J. E. *A Hidden Line Algorithm for Halftone Picture Representation.* University of Utah Computer Science Dept. Rep., TR 4–5, May 1968, NT IS AD 761 995.

[Whitted, 80] Whitted, T. *An Improved Illumination Model for Shaded Display.* Comm. of the ACM, Vol. 23, No. 6, June 1980, pp 343–349.

[Wright, 73] Wright, T. J. *A Two-Space Solution to the Hidden Line Problem for Plotting Functions of Two Variables.* IEEE Transactions on Computers, Vol. C-22, No. 1, January 1973, pp 28–33.

[Wyvill, 86] Wyvill, G., McPheeters, C. and Wyvill, B. *Data Structure for Soft Objects.* The Visual Computer, Vol 2, pp 227–234, 1986.