Apostila de Computação Gráfica
(com ênfase em síntese de imagens)

Bruno de Oliveira Schneider
Departamento de Ciência da Computação
Universidade Federal de Lavras

Versão: Março de 2020

Sumário

Lista de Figuras

Capítulo 1

Introdução

A Computação Gráfica nasceu da necessidade de se apresentar dados processados por um computador de uma maneira mais facilmente assimilável por uma pessoa do que uma simples lista de números. Dispositivos conhecidos como *plotters* já desenhavam gráficos em papel numa época em que computadores recebiam dados em papel perfurado e exibiam resultados em papel impresso. Mais tarde surgiu a ideia de se conectar um tubo de raios catódicos (CRT - o tubo de imagem das televisões) a um computador, o que abriu grandes possibilidades neste campo, permitindo a apresentação rápida de informações que representam os mais variados modelos, dos mais variados campos de conhecimento, associados a estruturas físicas, fenômenos naturais ou mesmo entidades abstratas.

Podemos definir a Computação Gráfica como a ciência que "estuda os métodos que permitem a visualização de informações armazenadas na memória do computador" [4]. Como existe uma enorme variedade de aplicações nesta área, pode ser interessante dividí-la em sub-áreas menores, de acordo com a natureza das informações usadas no processo de computação, levando em consideração se essas informações são dados de entrada ou de saída (resultados).

Assim, ainda conforme [4], podemos dividir a computação gráfica em 4 sub-áreas:

- *Modelagem Geométrica*: Trata do problema de descrever e estruturar dados geométricos no computador
- *Síntese de Imagens*: Processamento de dados gerados pelo sistema de modelagem, para produzir uma imagem que pode ser vista através de algum dispositivo de saída gráfica.
- *Processamento de Imagens*: Tem por objetivo identificar e realçar algum tipo de informação contida numa imagem, produzindo uma nova imagem como resultado.
- *Análise de Imagens*: Tem por objetivo extrair algum tipo de informação (topológica, física, etc.) de uma imagem, produzindo dados que serão armazenados na memória para processamento posterior.

Neste texto, estaremos preocupados principalmente com a *síntese de imagens*. Vamos supor que os dados necessários para criar a imagem já existem ou podem ser criados de alguma maneira. Precisamos agora converter estes dados de alguma maneira num dispositivo de apresentação, como o vídeo, por exemplo. A estratégia preferida para alcançar este objetivo é imitar a formação de imagens reais. Vamos procurar entender o processo da criação de uma fotografia numa câmera para implementar uma simplificação deste processo no computador. O processo de criação de uma foto é muito parecido com o processo de sensibilização da retina de um olho, de maneira que ele será chamado simplesmente de processo de "criação de imagens".

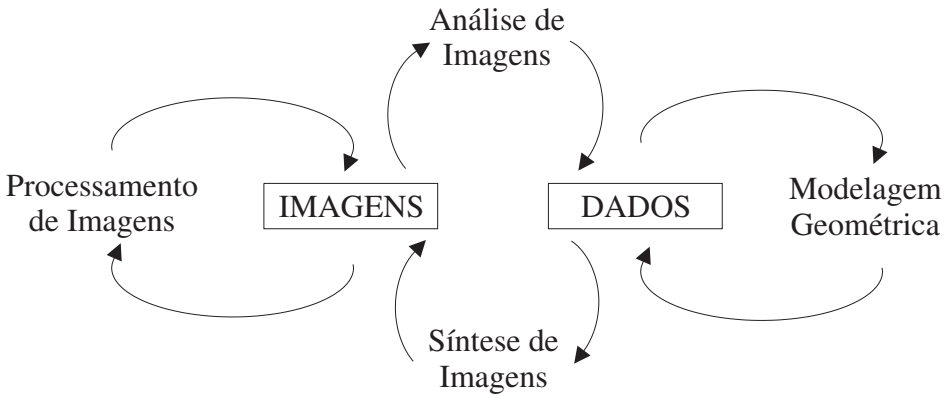


Figura 1.1: A Computação Gráfica dividida em quatro sub-áreas

1.1 A Natureza dos Dados Numa Imagem

Podemos classificar a representação de uma imagem de duas maneiras, dependendo da natureza dos dados utilizados na representação:

- Imagem vetorial: é uma imagem cuja representação é de natureza geométrica, ou seja, ela é definida em função de elementos geométricos e parâmetros (linhas, circunferências, pontos, preenchimentos, etc.)
- Imagem matricial: é uma imagem cuja representação é de natureza discreta, ou seja, a imagem é formada de elementos independentes, dispostos na forma de uma matriz, cada um contendo uma unidade de informação da imagem.

Apesar de se dizer "imagem vetorial" ou "imagem matricial", o que se está classificando não é a imagem em si, mas os dados usados para representá-la.

(a) Imagem Vetorial (b) Imagem Matricial

Figura 1.2: Comparação entre imagem vetorial e imagem matricial

Na Figura 1.2, pode-se observar uma porção da imagem matricial ampliada onde percebe-se claramente os elementos discretos que formam a imagem (chamados *pixels*[1]). O modelo matricial de

[1]Apesar de não haver uma fonte de referência formal, é usualmente aceito que a palavra *pixel* deriva de "picture element" (elemento de imagem).

representação de imagens pode representar qualquer imagem, o que explica o fato dos dispositivos matriciais de apresentação serem hoje muito mais comuns do que os dispositivos vetoriais. Imagens vetoriais, no entanto costumam ser pequenas (em relação à quantidade de *bytes*) e podem ter suas características (dimensões, cores, espessura de linhas, etc.) facilmente alteradas. A conversão de uma imagem vetorial para uma matricial pode ser vista como uma operação simples, entretanto, a conversão inversa é um processo complexo que dificilmente alcança os objetivos que se espera de uma conversão dessa natureza.

Por necessitarem de menos espaço em memória os dispositivos de apresentação vetoriais eram os mais comuns antigamente. Por possibilitarem a representação de mais informação do que a imagem propriamente dita, as imagens vetoriais não serão deixadas de lado na Computação Gráfica. De maneira geral, a representação matricial é mais adequada para imagens naturais (fotos, por exemplo), para imagens artificiais (logotipos, gráficos comparativos, esquemas explicativos, etc.) a representação vetorial costuma ser mais indicada.

1.1.1 Formatos de Arquivo

Existems formatos de arquivo para armazenar imagens matriciais, para vetoriais e para ambos. Os formatos que podem armazenados os tipos representações de imagens são frequentemente chamados de containers e em geral, podem também armazenar dados de outras naturezas, ou seja, dados que não representam imagens.

Alguns formatos de arquivo importantes são:

- JPEG - representação matricial usando compactação com perdas;
- PNG - representação matricial usando compactação sem perdas;
- SVG - representação vetorial;[2]
- PDF - container para imagens matriciais, vetoriais e outros tipos de dados.

1.2 O Processo de Criação de Imagens

Os dois elementos fundamentais no processo de criação de imagens são a câmera e o objeto do qual desejamos criar uma imagem. O objeto em questão não precisa necessariamente ser uma única entidade, entretanto, estaremos usando sempre o singular, já que a existência de vários objetos no processo de criação de imagens não afetará o processo descrito.

A câmera afeta a imagem criada em vários aspectos. Podemos citar, por exemplo, que a lente usada num máquina fotográfica influencia o tamanho da imagem criada, ou em outras palavras, a quantidade de informações que são convertidas em imagem. É o que chamamos de *zoom* da máquina. Dentre os aspectos da câmera, aquele mais facilmente identificável é a posição da câmera em relação ao objeto. A foto de prédio, por exemplo, certamente será diferente, quando for feita de frente para o prédio ou de dentro dele.

De maneira geral, podemos dizer que os fatores principais para definir uma imagem são a posição da câmera e a descrição física do objeto. É verdade que a luz presente no momento da criação da imagem também é um fator fundamental para se definir qual será a imagem formada. Sem luz, não

[2]Por ser um formato extensível, vários programas permitem a descrição de bitmaps dentro de um SVG, tornando o formato efetivamente um container.

seria possível criar uma imagem mesmo com uma câmera e um objeto, uma vez que o objeto seria todo preto e impossível de ser notado.

Outro fator importante a ser observado uma foto de um objeto é sempre 2D, mesmo que o objeto representado seja 3D. Quando chegar a hora de modelar matematicamente o processo de criação de imagens, vamos procurar entender o que causa essa eliminação de uma das dimensões físicas. Por hora, basta lembrar que uma transformação matemática que pega valores num conjunto de dimensão n e os associa a valores num conjunto de dimensão menor que n é chamada de **projeção**.

1.3 O Papel da Luz

Uma alternativa possível para modelar o processo de criação de imagens é tentar imitar o processo de interação entre luz, objeto e câmera. Sabemos que um objeto torna-se visível numa foto porque existe alguma fonte de luz que emite raios de luz. Estes raios por sua vez partem da fonte de luz deslocando-se em linha reta, eventualmente atingindo o objeto em questão. Após atingir o objeto, este raio sofre algum tipo de interação com o objeto e parte de sua energia deixa o objeto, na forma de um novo raio de luz que eventualmente irá atingir e interagir com a câmera, formando a imagem.

Uma fonte de luz pode ser representada por um ponto no espaço, de onde saem raios de luz em todas as direções. Destes raios, uma pequena porção acaba sensibilizando a câmera para formar a imagem. Existem várias alternativas para um caminho de um raio de luz que deixa a fonte de luz e chega na câmera. Vamos analisar algumas:

Uma forma de se implementar este modelo de criação de imagens seria gerar raios de luz que partem da fonte de luz e seguem numa direção qualquer até atingir a câmera. Este método, entretanto, é inviável, pois a fonte de luz emite raios em todas as direções e, além disso, um raio de luz segue seu caminho indefinidamente até que sua energia acabe, o que pode levar uma eternidade. Mesmo que estes dois problemas possam ser simplificados de maneira que possam ser implementados no computador ainda existe o problema de que somente um pequena parcela dos raios de luz que deixam a fonte, chegam a interagir com a câmera, o que significaria um enorme desperdício de processamento.

Precisamos usar algumas simplificações matemáticas e outros artifícios diversos para poder implementar um processo de geração de imagens, mas para isso, vamos analisar o processo de formação de imagens com mais detalhes.

1.4 Um modelo para a formação de imagens

É possível construir uma máquina fotográfica utilizando uma caixa de sapatos (ou lata com tampa) e papel fotográfico. Para tal é necessário fazer um pequeno furo com um prego numa das extremidades da caixa e colocar o papel fotográfico do outro lado. O furo deve ficar tampado (com fita adesiva, por exemplo) até o momento em que se deseja tirar a fotografia. Neste momento o furo deve ser destampado, mantendo-se a câmera parada por alguns instantes, e depois o furo deve ser novamente tampado. Durante o tempo que o furo fica destampado, a luz do ambiente pode entrar na caixa, sensibilizando o papel fotográfico que pode ser revelado mais tarde.

Este equipamento simples ajuda a entender e modelar o processo de criação de imagens. Veja a figura 1.4.

Para construir um modelo matemático desta câmera, vamos supor que o furo da caixa é tão pequeno que, a partir de um ponto na posição (x, y, z), só pode passar um único raio de luz, pelo furo, de tal

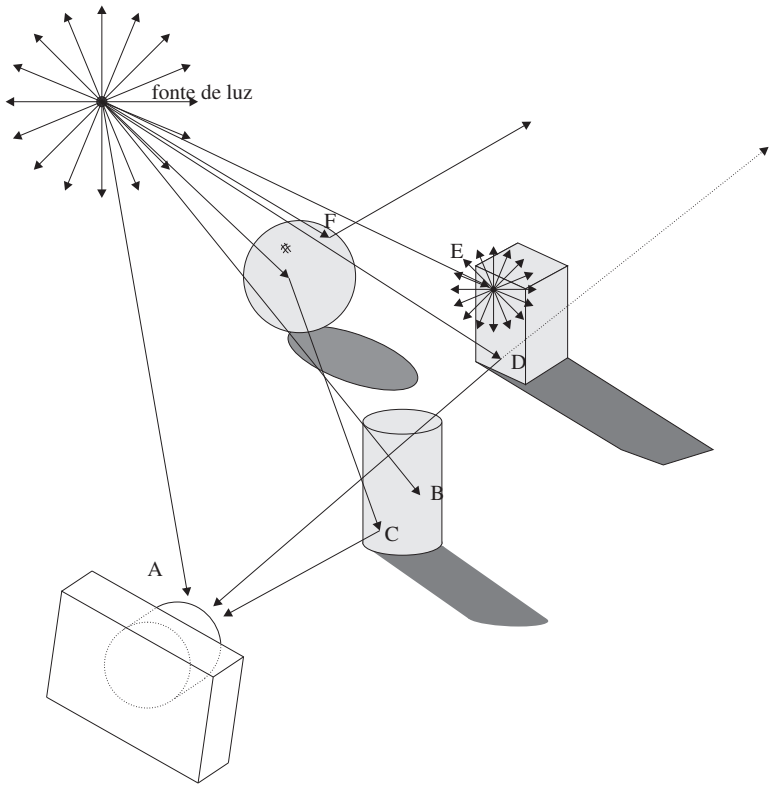


Figura 1.3: Algumas situações possíveis de caminhos de luz num ambiente
A) raio vai direto da fonte de luz à câmera; B) o raio de luz é absorvido pelo objeto; C) o raio de luz é refletido na esfera, depois no cilindro, indo depois para a câmera; D) parte da energia do raio de luz é refletida e parte sofre refração, atravessando o paralelepípedo; E) o raiz de luz atinge uma superfície e sua energia é dividida em infinitos novos raios de luz; F)o raiz de luz é refletido mas não contribui para a formação da imagem.

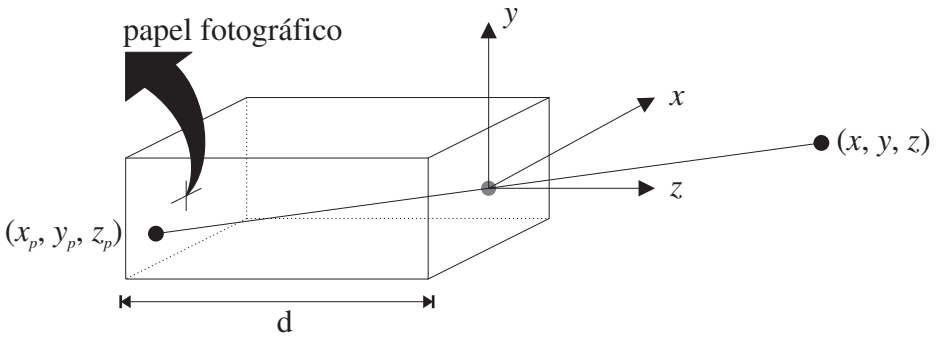


Figura 1.4: Esquema de uma câmera estenopeica

forma que este raio atinge o papel fotográfico no ponto que está na posição (*xp*, *yp*, *zp*). A transformação do valor (*x*, *y*, *z*) em (*xp*, *yp*, *zp*) nos fornecerá as coordenadas (*xp*, *yp*), que são as coordenadas do ponto na imagem (foto). Este processo é chamado de projeção. Todos os pontos visíveis do objeto são projetados no papel fotográfico, passo pelo furo da caixa, assim, chamamos o papel fotográfico de plano de projeção e o furo de centro de projeção. Este modelo matemático é usualmente chamado de *modelo da câmera sintética*.

Matematicamente, não existem grandes diferenças entre uma projeção onde o plano de projeção está entre o objeto e o centro de projeção e uma projeção onde o centro de projeção está entre o plano de projeção e o objeto.

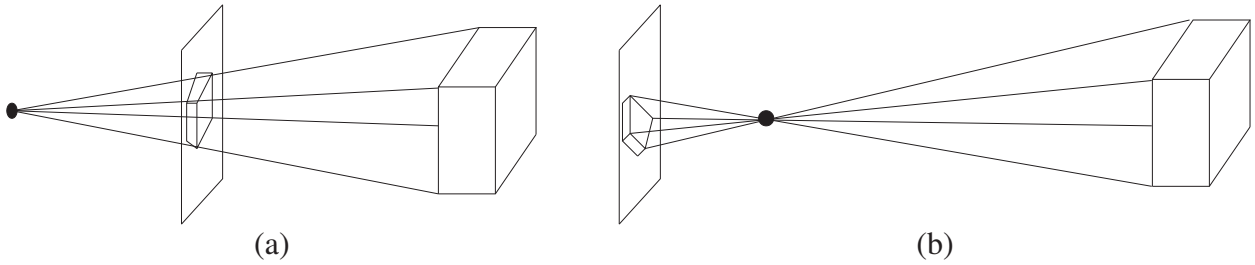


Figura 1.5: Projeções perspectivas
Em (a) O centro de projeção encontra-se depois do plano de projeção e em (b) o centro de projeção encontra-se antes do plano de projeção.

A projeção perspectiva mostrada na Figura 1.5 (b), é mais fiel ao que acontece fisicamente numa máquina fotográfica ou em nossos olhos, entretanto a imagem formada é invertida (de cima para baixo, da direita para a esquerda). A projeção mostrada em (a) além de formar uma imagem com a mesma orientação do objeto, ela pode representar um modelo onde uma pessoa, sentada em sua cadeira à frente de uma tela de vídeo, observa um objeto atrás do vídeo.

Se considerarmos que a imagem formada é uma matriz de pontos (*pixels*) discretos, podemos calcular a cor de cada ponto da imagem, analisando qual raio poderia passar pelo furo da caixa até chegar no papel fotográfico.

1.5 Um modelo físico: *Ray Tracing*

A ideia de seguir os raios de luz em seu caminho inverso, ou seja, do papel fotográfico até um objeto e possivelmente do objeto até uma fonte de luz, elimina a necessidade de se seguir raios de luz que não contribuem para a formação da imagem. Outras dificuldades advindas da quantidade de cálculos relativos à interação da luz com o objeto também podem ser reduzidas pela simplificação do modelo físico. Pode-se por exemplo, limitar o número de reflexões que o raio pode sofrer.

Além de tornar viável uma implementação que faz análise dos raios de luz, essa ideia pode ser aplicada independentemente da posição do plano de projeção em relação ao centro de projeção (ver Figura 1.5). Para cada *pixel* na imagem deve sair um raio que passa pelo centro de projeção. Deve-se então verificar quais são os pontos de interseção do raio com a superfície do objeto e então fazer o cálculo da cor do pixel.

Esta é uma visão geral e simplificada do algoritmo conhecido como *ray tracing* ou *ray casting*. A criação de imagens usando *ray tracing* é conhecida pela realismo das imagens criadas, entretanto este é um processo lento, que normalmente não é usada para gerar gráficos interativos.

1.6 Um modelo geométrico: Câmera Estenopeica.

Uma maneira mais rápida de se criar uma imagem a partir da descrição matemática de um objeto é calcular a projeção deste objeto e identificar a cor de cada *pixel* a ser exibido. Esse método é consideravelmente mais rápido que o *ray-tracing*, visto anteriormente. Por outro lado, esse método dificulta o cálculo do efeito da luz interagindo com vários objetos.

Vamos supor que desejamos criar a imagem de um cubo. Este cubo é definido matematicamente como um conjunto de 6 lados (chamados de *faces*) cada lado do cubo é definido em relação a 4 pontos 3D (com coordenadas x, y e z). Como algumas faces compartilham pontos, existe um total de 8 pontos 3D na definição do cubo. Devemos então calcular a projeção de cada um dos pontos, obtendo 8 pontos 2D, que definem os mesmos 6 lados. Com a projeção alguns pontos podem se sobrepor, ou seja, podem ser projetados nas mesmas coordenadas (xp, yp) mas todos os pontos continuam existindo.

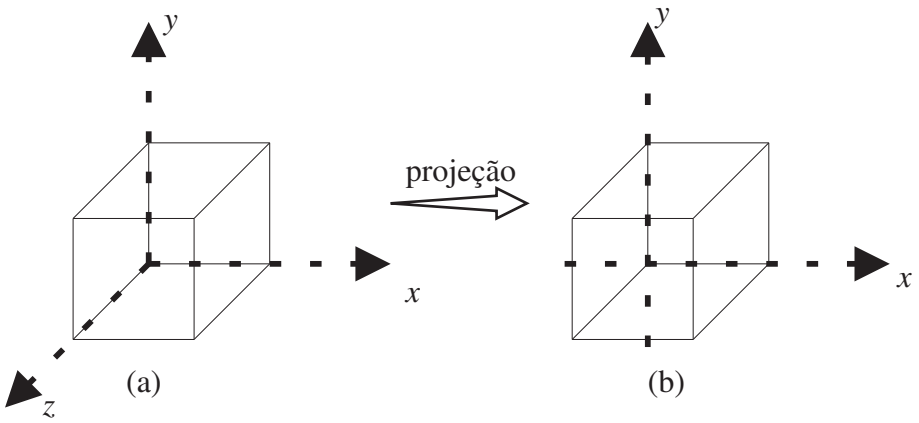


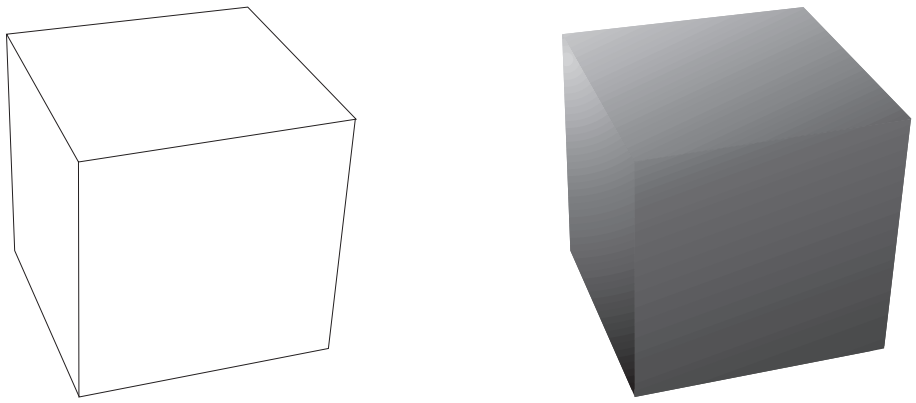
Figura 1.6: Projeção dos vértices de um cubo

Mas não basta desenhar os pontos projetados para se ter a imagem do cubo, também não basta desenhar as linhas entre os pontos que formam as arestas de cada face. É necessário saber quais arestas não estão visíveis (porque estão atrás de uma ou mais faces). Além disso, precisamos também definir a cor de cada *pixel* dentro de cada face. A eliminação das arestas que não estão visíveis vai melhorar a sensação de profundidade e a definição de cor de cada *pixel* do cubo vai aumentar a sensação de realismo, esse realismo será tão bom quanto o modelo matemático utilizado no cálculo. Modelos precisos significam mais cálculos, que por sua vez significam um gasto maior de tempo no processo.

Veremos como gerar a imagem da Figura 1.7a na seção "Tratamento de Linhas e Superfícies Escondidas" e depois veremos como gerar a imagem da Figura 1.7b na seção "Iluminação".

A determinação da cor de cada pixel é um processo complexo e existem várias abordagens para resolver este problema. Se tentarmos simular a realidade com muita precisão, certamente seremos obrigados a gastar muito tempo com cálculos. Não adianta pensar que um dia os computadores serão tão rápidos que este problema será irrelevante. Este processo pode ser tão complicado quanto se queira. É necessário encontrar uma região de equilíbrio entre o realismo proporcionado pelo método de criação de imagens e tempo disponível para cálculos.

A Computação Gráfica Interativa e a Realidade Virtual enfrentam o desafio de tentar sintetizar imagens de boa qualidade em muito pouco tempo. Para se ter uma ideia melhor do problema do tempo, imagine que o ser humano consegue distinguir eventos e imagens distantes aproximadamente 100ms no tempo. Isso quer dizer que um sistema interativo de computação (como um sistema de realidade virtual) deve exibir o estado de uma operação qualquer no mínimo dez vezes por segundo, o que lhe

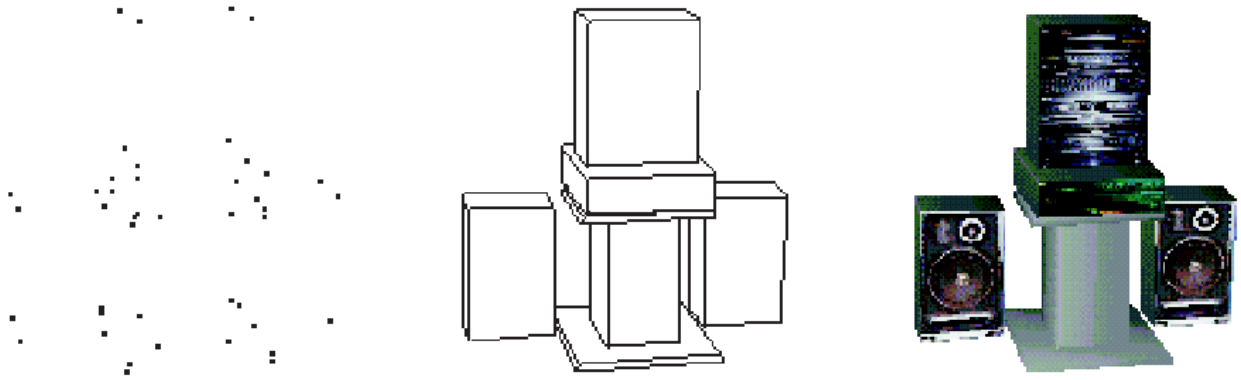


(a) (b)

Figura 1.7: Um cubo projetado com determinação das faces visíveis (a) e o mesmo cubo após a determinação da cor de cada pixel (b)

deixa um tempo de 100ms para processar toda a informação colhida da interação do usuário e apresentar resultados. Agora olhe ao seu redor e imagine como seria possível sintetizar a imagem que você vê em menos de 100ms.

Uma boa técnica para se acrescentar realismo a uma imagem sintetizada é o uso de texturas. Uma textura é uma imagem mapeada sobre um polígono plano (uma face) que por sua vez é parte de um objeto 3D. A função da textura é proporcionar realismo a um objeto que tenha modelo geométrico simples. Apresentar a imagem de um objeto com textura é um processo mais lento que apresentar a imagem de um objeto sem textura. Entretanto, apresentar a imagem de um objeto com textura é mais rápido que apresentar a imagem de um objeto com modelo geométrico complexo, que dispense a textura.



(a) (b) (c)

Figura 1.8: (a) Projeção dos pontos que descrevem o objeto; (b) determinação de superfícies escondidas e (c) aplicação de texturas e modelo de iluminação

Além de todo o processo já discutido, a criação de imagens em tempo interativo envolve ainda a eliminação de porções não visíveis do objeto na tentativa de evitar o processamento de dados que não irão contribuir para a formação da imagem. Neste instante, por exemplo, você não vê aquilo que está atrás de você. Um sistema que simula o processo de visão de alguém deve ser capaz de eliminar o processamento dos objetos fora de alcance da visão. Essa eliminação é chamada de *recorte*.

De maneira geral o processo de síntese de imagens em tempo interativo pode ser representado pelo diagrama da figura 1.9. Esse process é conhecido como *pipeline* gráfico tradicional.

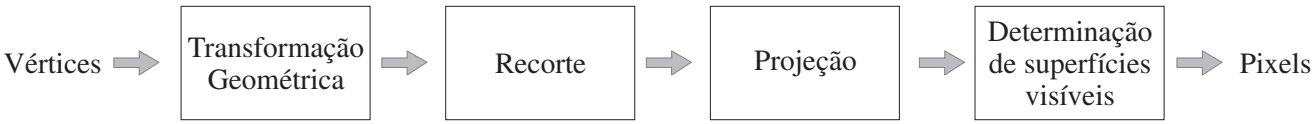


Figura 1.9: Visão geral do processo de síntese de imagens

1.7 Revisão Matemática

Os elementos dessa seção são parte do conteúdo programático de disciplinas que são pré-requisitos para Computação Gráfica ou são abordados no ensino médio. Porém, a experiência mostra que é necessário que eles sejam revistos antes da apresentação dos tópicos posteriores.

1.7.1 Círculo trigonométrico

É importante lembrar conceitos de trigonometria. Por definição, seno é o cateto oposto dividido pela hipotenusa; cosseno é o cateto adjacente dividido pela hipotenusa e portanto a posição de um ponto qualquer sobre o círculo é dada por $(r \cdot cos(\alpha), r \cdot sen(\alpha))$. A figura 1.10 ajuda na visualização do significado do seno, cosseno, tangente e cotangente, além de mostrar alguns ângulos notáveis em graus e em radianos.

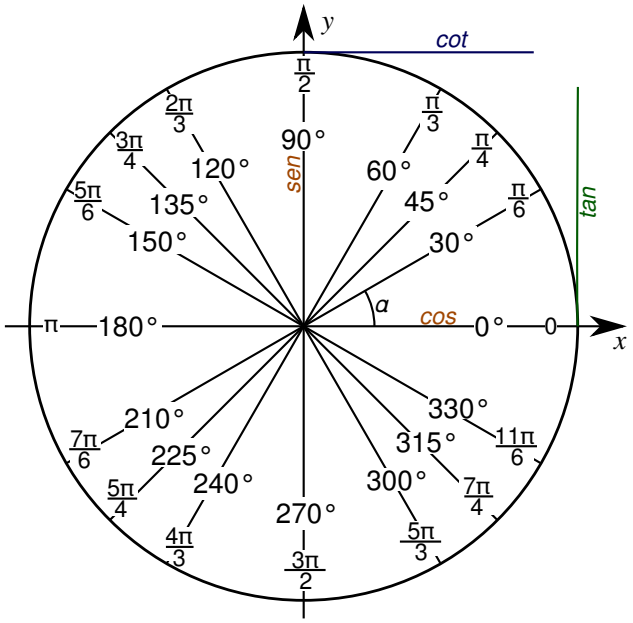


Figura 1.10: Círculo trigonométrico.

Para fazer referência a um quadrante ou um octante do círculo, seguimos a ordem convencionada para o ângulo, assim o primeiro quadrante é compreendido pelo intervalo angular [0°, 90°[, o segundo pelo intervalo [90°, 180°[e assim por diante.

Várias bibliotecas de programação usam ângulo expressos em radianos. É importante lembrar que valor π (pi) é a razão entre a circunferência e o diâmetro de qualquer círculo. Este valor é um número

irracional e só pode ser representado de maneira aproximada. Uma boa aproximação em programas de Computação Gráfica é $\pi \approx 3,14159265358$. Várias linguagens de programação tem bibliotecas que definem essa constante.

Para converter de graus para radianos e vice-versa, basta estabelecer uma relação entre valores correspondentes nas duas métricas.

$$\frac{\pi}{180} = \frac{radianos}{graus}$$
$$\Rightarrow radianos = graus \cdot 0,017453292519943$$
$$\Rightarrow graus = radianos \cdot 57,29577951308232$$

1.7.2 Operações com matrizes

A multiplicação de matrizes é feita em linhas e colunas correspondentes conforme mostrado na figura 1.11. O número de colunas do primeiro operando deve ser igual ao número de linhas do segundo operando. Mais formalmente, a multiplicação de matrizes é dada por:

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + \cdots + a_{in} b_{nj}$$

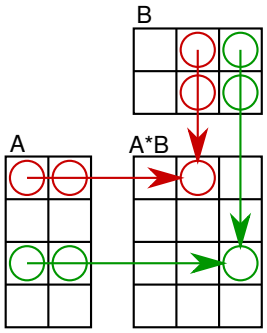


Figura 1.11: Multiplicação de matrizes.

A multiplicação de matrizes é associativa ($ABC = (AB)C = A(BC)$) mas não é comutativa ($AB \neq BA$). Nem toda matriz é inversível. Toda matriz que é inversível é quadrada. A multiplicação de matrizes pode ser implementada conforme o algoritmo 1.1.

Algoritmo 1.1 Multiplicação de matrizes

Dadas duas matrizes compatíveis, m1 e m2, o produto m3 = m1*m2 pode ser calculado assim:
1. Para cada linha da matriz m1:
1.1. Para cada coluna da matriz m2:
1.1.1. soma←0
1.1.2. Para cada índice de coluna de m1 (i):
1.1.2.1. soma ←m1(linha, i) + m2(i, coluna)
1.1.3. m3(linha, coluna) ←soma

1.7.3 Regra da mão direita

Para fazer operações no espaço, geralmente precisamos de alguma convenção que determine o que é positivo e o que é negativo. Uma referência muito comum de orientação é "sentido horário" ou anti-horário, porém essa referência costuma causar confusão num espaço 3D. Na Álgebra Linear existe o conceito de bases[3] com orientação positiva (bases positivas) e bases com orientação negativa (bases negativas). Essas duas classes de bases podem ser distinguidas pela regra da mão direita.

A regra da mão direita é recurso mnemônico para se determinar uma orientação positiva no espaço. Ela é usada em Física, Álgebra, Computação Gráfica e outras áreas do conhecimento. Para determinar o sentido positivo no espaço, use o dedão da mão direita espalmada, enquanto alinha os outros dedos com os elementos verificados. Com ela fica fácil determinar a orientação dos vetores de uma base positiva (figura 1.12 - a) e o sentido de rotação em relação a um eixo (figura 1.12 - b).

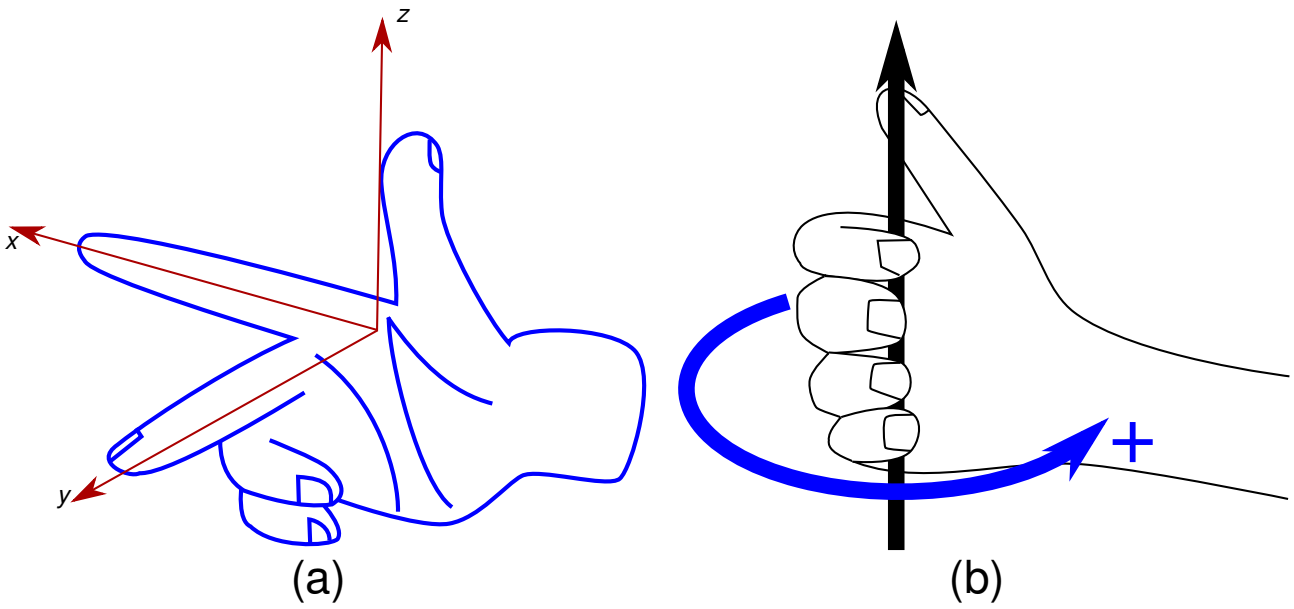


Figura 1.12: A regra da mão direita: orientação de vetores de uma base (a) e sentido positivo de rotação (b).

1.7.4 Operações com pontos

1.7.4.1 Subtração

A subtração de dois pontos (P_1 - P_2) pode ser definida com um segmento de reta orientado que vai de de P_2 para P_1, ou também como a direção (vetor) que leva a posição P_2 para P_1.

$$P_1 - P_2 = (x_1 - x_2,\ y_1 - y_2,\ z_1 - z_2)$$

[3]Uma base é um conjunto de vetores *linearmente independentes* que *geram* todos os vetores possíveis de um espaço vetorial.

1.7.5 Operações com vetores

1.7.5.1 Soma

A soma de dois vetores pode ser entendida como o segmento de reta orientado que une a base do primeiro vetor à ponta do segundo vetor.

$$P_1 + P_2 = (x_1 + x_2,\ y_1 + y_2,\ z_1 + z_2)$$

1.7.5.2 Produto Escalar

Dados dois vetores $u = (u_x, u_y, u_z)$ e $v = (v_x, v_y, v_z)$, o produto escalar é:

$$u \cdot v = u_x v_x + u_y v_y + u_z v_z$$

Ele tem a propriedade geométrica de estar relacionado com o ângulo entre os dois vetores, conforme mostra a figura 1.13. Para vetores normalizados, o produto escalar é o cosseno do ângulo entre os vetores.

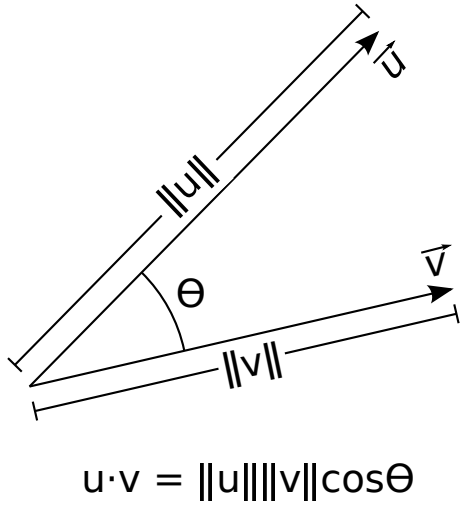


Figura 1.13: A relação entre produto escalar e ângulo.

Como o cosseno de um ângulo no intervalo $[0°, 90°)$ é positivo e o cosseno de um ângulo no intervalo $(90°, 180°]$ é negativo, podemos usar o sinal do produto escalar para saber se dois vetores tem direções opostas ou não.

1.7.5.3 Produto Vetorial

Dados dois vetores $u = (u_x, u_y, u_z)$ e $v = (v_x, v_y, v_z)$, produto vetorial é:

$$u \times v = (u_y v_z - u_z v_y,\ u_z v_x - u_x v_z,\ u_x v_y - u_y v_x)$$

Ele tem a propriedade geométrica de que o resultado é um vetor perpendicular tanto a u quanto a v. Dados dois vetores u e v, existem dois vetores que são perpendiculares a eles: $u \times v$ e $-(u \times v)$. Para saber qual dos dois é resultado, pode-se usar a regra da mão direita, colocando o indicador na direção do primeiro operando e o dedo médio na direção do segundo, temos o resultado na direção do

polegar. O produto vetorial é anticomutativo, ou seja $u \times v = -v \times u$. A norma do produto vetorial é proporcional à norma de seus operandos.
A fórmula do produto vetorial 3D é mais facilmente lembrada se associada ao determinante da matriz 3x3 em que a primeira linha tem os vetores unitários que compõe um sistema ortogonal de coordenadas, a segunda tem as coordenadas do primeiro operando e a terceira tem as coordenadas do segundo operando, conforme visto na figura 1.14.

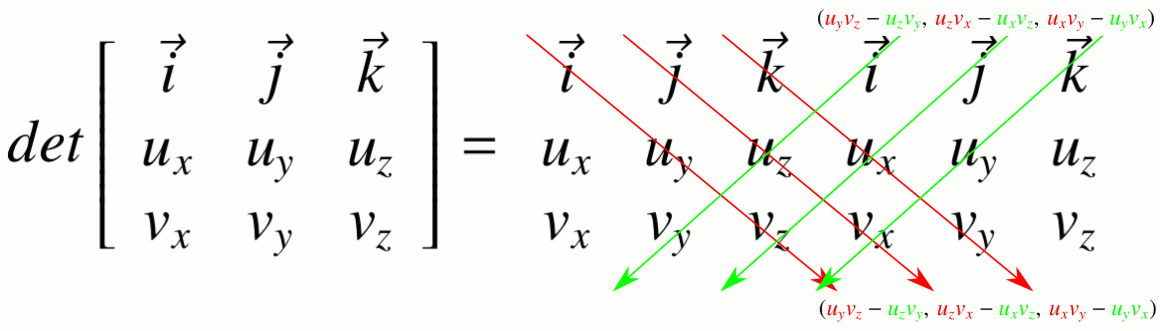


Figura 1.14: Mnemônico da fórmula do produto vetorial.

1.7.6 Equação da reta

Uma reta é um conjunto infinito de pontos ao longo de uma dimensão. Esse conjunto de pontos pode ser descrito no plano (deixaremos o espaço 3D para depois) de várias formas. A forma mais lembrada é a equação explícita, em que a coordenada y é dada em função da coordenada x.

$$y = ax + b$$

Com frequência, deseja-se encontrar a equação a partir de dois pontos (P_1 e P_2) da reta. Nesse caso, os parâmetros são dados por:

$$a = \frac{y_2 - y_1}{x_2 - x_1} \qquad b = y_1 - ax_1$$

A equação explícita não pode ser usada para representar retas verticais, ou seja, cuja coordenada x é constante ao longo de toda a reta. Neste caso, a equação implícita é mais útil:

$$Ax + By + C = 0$$

Os parâmetros podem ser encontrados a partir de dois pontos:

$$A = y_1 - y_2 \qquad B = x_2 - x_1 \qquad C = x_1y_2 - x_2y_1$$

Por fim, podemos descrever uma reta em função de um parâmetro que é um número real e será denominado t. Para isso, precisamos de dois pontos (P_1 e P_2) da reta. Portanto, dados dois pontos, podemos representar todos os outros infinitos pontos da reta em função de t pela equação paramétrica:

$$P(t) = P_1 + (P_2 - P_1)t$$

A equação paramétrica faz uso de algumas das operações vistas nas seções 1.7.5 e 1.7.4 e tem a vantagem de ser independente do número de dimensões do espaço.

1.7.7 Cálculo de uma posição na reta

Com alguma frequência, temos uma reta (ou segmento de reta), e queremos saber qual o valor de x para determinado y (ou vice-versa). A seção 4.1 tem algumas dessas situações.

Esse problema é facilmente resolvido por semelhança de triângulos, ou *regra de três*. Por exemplo: suponha a situação da Figura 1.15 em que dados dois pontos P0 e P1, queremos calcular o valor de x para um dado y.

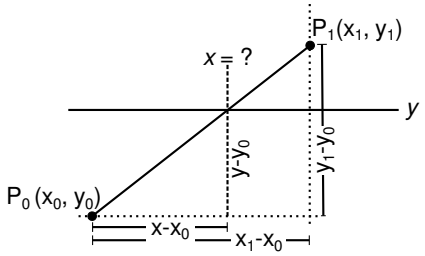


Figura 1.15: Cálculo de uma posição na reta

Temos que:

$$\frac{y_1 - y_0}{x_1 - x_0} = \frac{y - y_0}{x - x_0} \Longrightarrow x = x_0 + \frac{(x_1 - x_0)(y - y_0)}{y_1 - y_0}$$

1.7.8 Equação do plano

Na seção 6.3 vamos precisar encontrar as constantes A, B, C e D que descrevem um plano por meio de sua equação implícita

$$Ax + By + Cz + D = 0$$

dados três pontos $P_0 = (x_0, y_0, z_0)$, $P_1 = (x_1, y_1, z_1)$ e $P_2 = (x_2, y_2, z_2)$. Os três pontos estão em sentido positivo, conforme a *regra da mão direita*. Assim, a normal ao plano é:

$$\vec{N} \quad = \quad (P_1 - P_0) \times (P_2 - P_1) = (n_x, n_y, n_z)$$

Dado um ponto $P = (x, y, z)$ qualquer no plano definido por P_0, P_1 e P_2sabemos que que um vetor que vai de P a P_1 forma um ângulo de 90° com o vetor normal, portanto:

$$(x - x_1, y - y_1, z - z_1) \cdot \vec{N} = 0$$

$$xn_x - x_1n_x + yn_y - y_1n_y + zn_z - z_1n_z = 0$$

$$n_xx + n_yy + n_zz - (x_1n_x + y_1n_y + z_1n_z) = 0$$

Assim, concluímos que:

$$A = n_x$$
$$B = n_y$$
$$C = n_z$$
$$D = -(x_1 n_x + y_1 n_y + z_1 n_z)$$

Capítulo 2

Representação de Sólidos

Para produzir imagens de objetos é preciso ter alguma forma computacionalmente eficiente para representá-los. Na Orientação à Objetos aprendemos a determinar quais são os elementos importantes de um objeto para possibilitar a computação daquele tipo de informação. Quais são os elementos importantes para sintetizar a imagem de objeto?

Vamos tomar como exemplo uma cadeira. Atributos de uma cadeira, poderiam ser o peso, a cor predominante, o número de patrimônio, o nome da empresa fabricante, etc. Esses atributos podem ser úteis para um sistema de controle de inventário, mas certamente não contribuem para produzir uma imagem da cadeira. Para produzir uma imagem da mesma, é preciso uma descrição geométrica da mesma, além de propriedades de material que sejam importantes para determinar como o objeto é visto (uma mesa de vidro não é vista da mesma forma que uma mesa de madeira, mesmo que a geometria das duas seja equivalente).

Por hora, a preocupação é a representação da geometria. As propriedades de material serão abordadas no capítulo 7.

Estamos interessados na representação da geometria de sólidos, pois na natureza não existem objetos planos. Existem objetos sem geometria bem definida como fogo, fumaça, etc., mas os solídos são suficientes para representar uma grande quantidade de situações reais ou imaginárias.

Existem diversas formas para representar sólidos. Algumas das mais diretas e comumente usadas são apresentadas aqui.

2.1 Enumeração de ocupação espacial

Provavelmente a forma mais direta de representação de sólidos é dividir o espaço em elementos discretos que podem ou não fazer parte do solido. A divisão mais simples seria uma divisão regular, em que todas as células têm o mesmo volume. Assim, uma matriz 3D (considerando um espaço 3D) binária pode ser usada para representar uma cadeira ou qualquer outro sólido. Neste tipo de representação, cada célula é chamada de *voxel* (de *volume element*).

2.2 Geometria sólida construtiva (*Constructive Solid Geometry* - CSG)

Conisiste em determinar uma série de sólidos primitivos (cubo, cone, esfera, etc.) e combiná-los por operações matemáticas (união, interseção, diferença, etc.) para produção de outros sólidos.

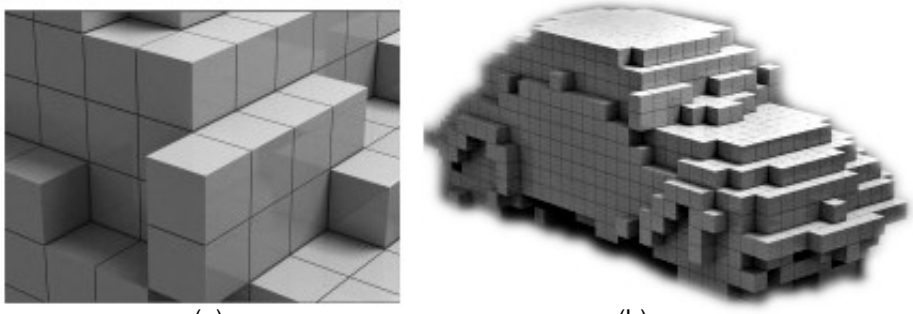


Figura 2.1: (a) Elementos discretos segundo divisão regular (b) Conjunto de voxels representando um carro. Imagem retirada de http://www.bilderzucht.de/blog/3d-pixel-voxel/.

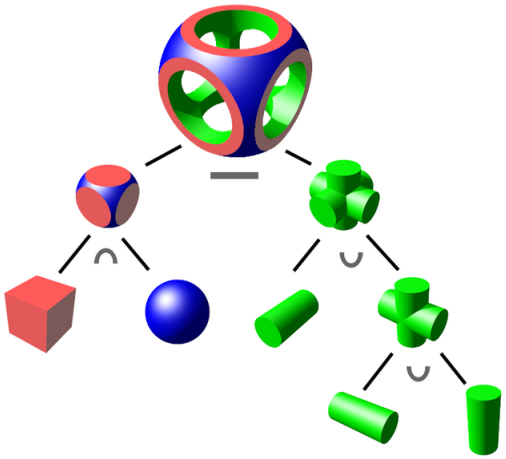


Figura 2.2: Exemplo de representação de um sólido por meio de operações aplicadas a sólidos primitos. Imagem extraída de http://en.wikipedia.org/wiki/File:Csg_tree.png

2.3 Modelagem baseada em pontos (*Point based modeling*)

Com o aumento da complexidade dos objetos cuja imagem pretende-se produzir, a modelagem baseada em pontos tem ganhado força recentemente. A ideia fundamental é reproduzir uma quantidade massiva de detalhes de geometria. Todos esses detalhes de geometria serão eventualmente transformados em pixels e a quantidade de pixels num dispositivo de apresentação não tem crescido tanto quando a informação geométrica. Assim, existem vantagens a representação de sólidos a partir de conjuntos de pontos procura tirar proveito dessa situação. Para mais informações sobre modelagem baseada em pontos, veja [6].

O barateamento de sensores espaciais, baseados em laser, ultrassom, ou infravermelho e luz visível também tem ajudado a popularizar o uso e processamento de modelos de sólidos baseados em pontos, visto que estes aparelhos discretizam o espaço sendo escaneado e localizam uma quantidade de pontos, produzindo um conjunto, frequentemente chamado de nuvem de pontos que representa o objeto digitalizado.

Figura 2.3: Imagem formada a partir de pontos da superfície de um objeto. A quantidade de pontos é pequena e o espaço entre pontos é grande para destacar a natureza discreta da imagem. Extraído de http://graphics.ethz.ch/publications/tutorials/eg2002/.

2.4 Modelagem tradicional

A maneira tradicional para representação de sólidos envolve a descrição da sua superfície a partir de um conjunto de polígonos chamados de *faces*. As faces são frequentemente triângulos, que são os polígonos mais simples. Um modelo geométrico formado por um conjunto de triângulos que compartilham arestas, determinando a superfície de um sólido é frequentemente chamado de *malha de triângulos*.

Esse é tipo mais comum de modelagem de objeto e é foco deste curso. Os objetos usados para síntese de imagens serão sempre representados por malhas de polígonos, a menos que algo contrário seja

explícito.

Vários polígonos representam uma superfície qualquer de maneira aproximada. Quanto maior a quantidade de polígonos, melhor será a aproximação da representação em relação ao objeto. A figura 2.4 apresenta um mesmo objeto aproximados por quantidades diferentes de triângulos. Nota-se que são necessários vários triângulos para representar detalhes de uma superfície.

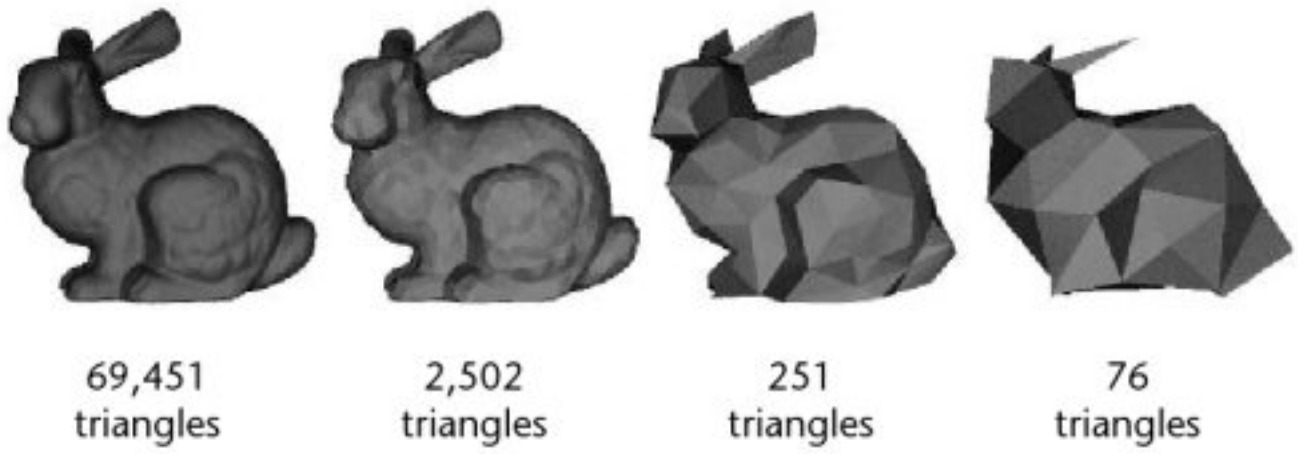


Figura 2.4: Quatros aproximações diferentes para a superfície de um objeto. Extraído de http://polygon-reducer.pc-guru.cz/reducing-level-of-detail.

Para representar um objeto precisamos então de polígonos. Para representar polígonos, precisamos de pontos. Pontos são posições no espaço e não tem tamanho. Três pontos não colineares representam um triângulo. Atenção para o fato de que quatro pontos não colineares não necessáriamente representam um quadrilátero. Para tanto, é preciso que os pontos sejam também coplanares.

Queremos fazer programas de computador que produzem imagens de objetos, então precisamos representar esses objetos de forma que um programa possa fazer a leitura dos dados de maneira simples e eficiente. Vamos "inventar" um formato de arquivo que representa sólidos. Para guiar este exercício, vamos supor que queremos representar um cubo, que tem uma das pontas (chamadas de *vértices*) na origem do sistema de coordenadas e aresta de tamanho 1, todas as três se extendendo no sentido positivo do sistema de coordenadas. Assim, o objeto em questão tem 6 faces (quadrados) definidas por 8 vértices, conforme apresentado na figura 2.5.

Para representar o cubo, podemos usar um arquivo texto, descrevendo cada uma das seis faces, a partir de seus vértices. As faces precisam conter informação suficente para determinar qual o lado de fora e qual o lado de dentro do sólido. Para isso vamos usar a convenção da *regra da mão direita* (seção 1.7.3) e descrever os vértices em sentido positivo com o dedão apontando para o lado de fora. Dessa forma sabemos não apenas qual polígono define a face mas também sabemos qual o lado visível (o lado interno de uma face de um sólido não é visível). Na seção 1.7.8 foi mencionado o cálculo de uma normal a partir de três pontos em um plano. Num sólido, existe a convenção de que a normal de uma face aponta para fora.

A face1, por exemplo, é determinada pelos vértices 3, 8, 4 e 7. Devemos colocá-los em sequência e de acordo com a regra da mão direita. Assim, a face1 pode ser descrita de quatro formas possíveis: {7, 3, 4 e 8 }, {3, 4, 8 e 7}, {4, 8, 7 e 3} ou {8, 7, 3 e 4}. Vamos escolher essa última. Sabemos que o vértice 8 tem coordenadas (0, 1, 1), o vértice 7 tem coordenadas (1, 1, 1), o vértice 3 tem coordenadas (1, 1, 0) e o vértice 4 tem coordenadas (0, 1, 0).

Assim, a face 1 poderia ser descrita por:

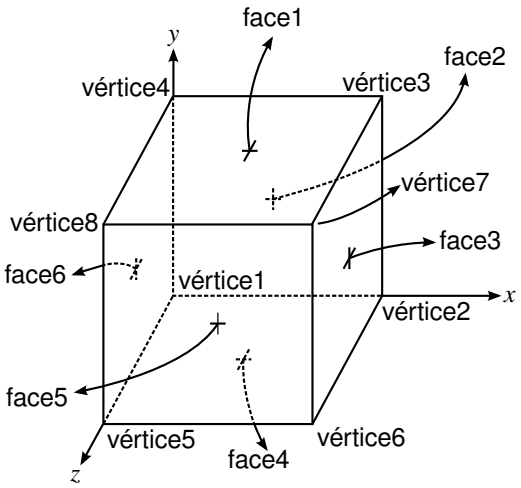


Figura 2.5: As 6 faces e 8 vértices de um cubo.

```
(0, 1, 1)(1, 1, 1)(1, 1, 0)(0, 1, 0)
```

Supondo que o sólido é um conjunto de faces e que cada face é uma linha no arquivo texto, o cubo seria:

```
(0, 1, 1)(1, 1, 1)(1, 1, 0)(0, 1, 0)
(0, 0, 0)(0, 1, 0)(1, 1, 0)(1, 0, 0)
(1, 0, 0)(1, 1, 0)(1, 1, 1)(1, 0, 1)
(0, 0, 1)(0, 0, 0)(1, 0, 0)(1, 0, 1)
(1, 1, 1)(0, 1, 1)(0, 0, 1)(1, 0, 1)
(0, 1, 0)(0, 0, 0)(0, 0, 1)(0, 1, 1)
```

Porém, esse arquivo seria muito difícil de ser lido por um programa com todos esses parênteses e vírgulas. Vamos simplificar isso usando apenas números separados por espaço. Outro problema é que existe muita redundância no arquivo. Cada vértice participa de três faces e sua descrição aparece três vezes. Além de ocupar mais espaço do que é necessário, isso traria dificuldade para realizar alterações nos dados. Se alterarmos um vértice, será necessário procurar e alterar todas as suas outras ocorrências. Para evitar isso, cada vértice será descrito uma única vez no início do arquivo. Depois, cada face será descrita por índices que representam os vértices, começando em 0.

Precisamos de algo que separe as descrições de vértices das descrições de faces. Para essa separação, os dados vão começar um um número que indica a quantidade de vértices no sólido. Como cada vértice tem três coordenadas, fica fácil saber quantos números devem ser lidos representando vértices. Os outros representarão faces.

Para indicar o final de cada face, ao invés de usar o final de linha, usaremos um número: um índice inválido, isso vai facilitar a leitura já que cada face poderia ter uma quantidade indeterminda de vértices. Assim, usaremos o valor -1 para indicar um final de face. O final de linha ao final de cada face será mantido apenas para facilitar a visualização do arquivo, já que finais de linha são facilmente descartados na leitura de números em qualquer linguagem de programação. O arquivo ficaria portanto:

```
8
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
7 6 2 3 -1
0 3 2 1 -1
1 2 6 5 -1
4 0 1 5 -1
6 7 4 5 -1
3 0 4 7 -1
```

Com essas regras simples, conseguimos descrever a geometria de um objeto qualquer. Nenhum recurso para descrever mais de um objeto foi "inventado". Formatos reais de representação de objetos contém muito mais informações que somente a geometria dos sólidos. Outros elementos importantes serão vistos ao longo do curso.

Capítulo 3

Fundamentos Matemáticos

Vamos estudar métodos para gerar imagens a partir da descrição matemática de um objeto. Essa descrição matemática deverá conter, entre outras coisas, uma descrição geométrica do objeto, ou seja uma descrição de sua forma física. Mesmo que o objeto seja algo imaginário, é necessário que ele tenha forma para que se possa gerar sua imagem.

Em Computação Gráfica, trabalharemos com três grupos básicos: escalares, pontos e vetores. Estes três grupos podem ser definidos de pelo menos três maneiras diferentes: matematicamente, geometricamente ou do ponto de vista de estruturas de dados. Vamos tentar misturar estas três definições.

Um escalar representa uma quantidade, seguindo algum padrão de medição. Ele é um número.

Um ponto representa uma posição no espaço. Ele não tem dimensão e é definido (no espaço tridimensional) por três coordenadas (três números), medidas em relação à uma referência pré-estabelecida que chamaremos de *sistema de coordenadas* ou *orientação*.

Um vetor representa uma direção no espaço. Ele não possui posição associada. Neste texto iremos evitar diferenciar os termos *direção* e *sentido*, aos quais no acostumamos por causa das aulas de física. Estaremos usando *direção* para significar uma *direção e sentido*. Vetores dos quais diríamos que tem mesma *direção* e *sentidos* opostos, serão ditos como tendo *direções* opostas, o que se aproxima mais da linguagem popular. Por não possuir posição associada, vetores que estávamos acostumados a chamar de paralelos, serão tratados neste texto como sendo vetores equivalentes, ou seja, vetores que indicam a mesma *direção* e que portanto são, na verdade, o mesmo vetor. Um vetor (no espaço tridimensional) pode ser definido por três coordenadas. Por exemplo: o vetor (0,1,0) pode ser entendido como o vetor que vai da origem (0,0,0) até o ponto (0,1,0) e, normalmente, poderíamos chamar essa direção de "para cima".

3.1 Transformações Geométricas

3.1.1 Definição

As transformações geométricas têm papel de destaque na computação gráfica. Elas são a base matemática que permite a obtenção de resultados importantes em programas que trabalham com dados 2D e 3D. Dentre as várias aplicações das transformações geométricas, temos interesse especial em utilizá-las para fazer projeções seguindo o modelo da câmera estenopeica. Antes de utilizá-las, no entanto, é importante definí-las formalmente para que seu significado fique bem claro.

3.1.1.1 Translação

Podemos entender a transformação de translação como o sendo um deslocamento em linha reta. Podemos associar um fator de deslocamento a cada dimensão do espaço onde estamos trabalhando, de forma que uma translação pode ser representada por vetor. Não faz sentido aplicar uma translação a vetor, já que não estamos associando posição a ele.

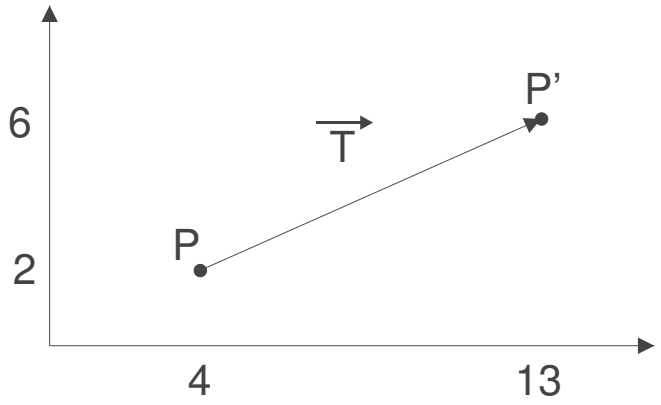


Figura 3.1: Exemplo de translação

Na figura acima, vemos o ponto P(4, 2) onde foi aplicada uma translação (9, 4) transformando-o no ponto P' (13, 6).

Muitas pessoas estranham o fato da translação ser um deslocamento em linha reta ou numa *direção*. Isso se deve à associação do nome com movimento de deslocamento da Terra em torno do Sol ou da Lua em torno da Terra. Esses movimentos não são em linha reta, mas isto deve ao fato deste movimento ser analisado em relação ao tempo. Como a direção do deslocamento muda a cada instante, o resultado final do deslocamento é curvo. Matematicamente falando, uma translação não envolve tempo.

3.1.1.2 Rotação

A rotação pode ser definida como um movimento de giro, num plano, em relação a um referencial. O resultado é um deslocamento que preserva a distância entre o ponto rotacionado e o referencial. Esta transformação preserva ângulos e distâncias.

No espaço tridimensional, o referencial pode ser definido através da associação de um ponto com um vetor, que chamaremos de *eixo*. Assim, podemos dizer que um *eixo* é um vetor ao qual existe uma posição associada ou, "um vetor que passa por algum lugar". No espaço bidimensional, o referencial pode ser definido por um ponto.

A rotação, portanto, pode ser definida por um ângulo e um referencial (ponto ou eixo).

3.1.1.3 Escala

A transformação de escala pode ser entendida como a mudança de unidade de medida em uma ou mais dimensões. Ela produz um efeito que pode ser associado ao *zoom*. Normalmente, a mudança de escala é igual em todas as dimensões, de forma a preservar ângulos, mas isso não é obrigatório. Associaremos um fator de escala a cada dimensão. Apesar de não preservar ângulos e distâncias, uma transformação de escala preserva o paralelismo entre linhas. Na figura 3.3, temos duas transformações

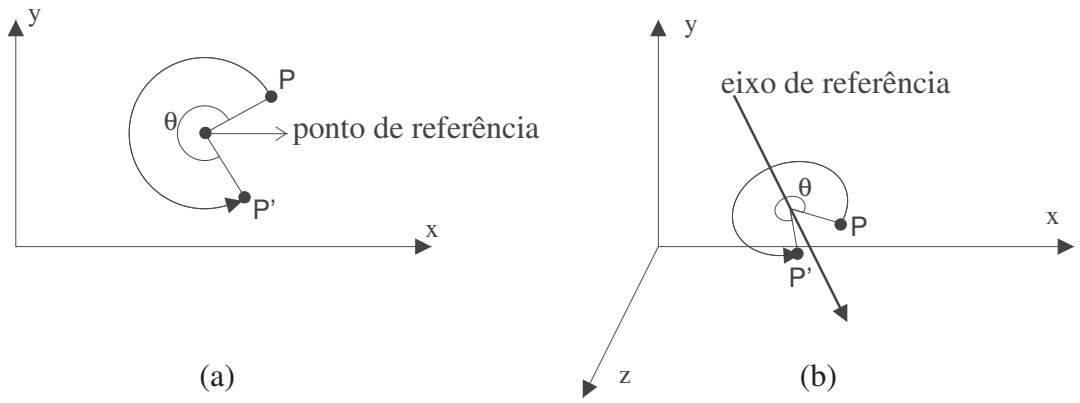


Figura 3.2: Exemplo de rotação 2D (a) e 3D (b)

de escala no plano. Em (a) temos um fator de escala em *x* igual a 2 e fator de escala em *y* igual a 1. Note que não só as dimensões da casa mudaram, mas a também a sua posição relativa à origem. Em (b) temos um fator de escala de 1.5 em *x* e em *y*. Note que os ângulos foram preservados.

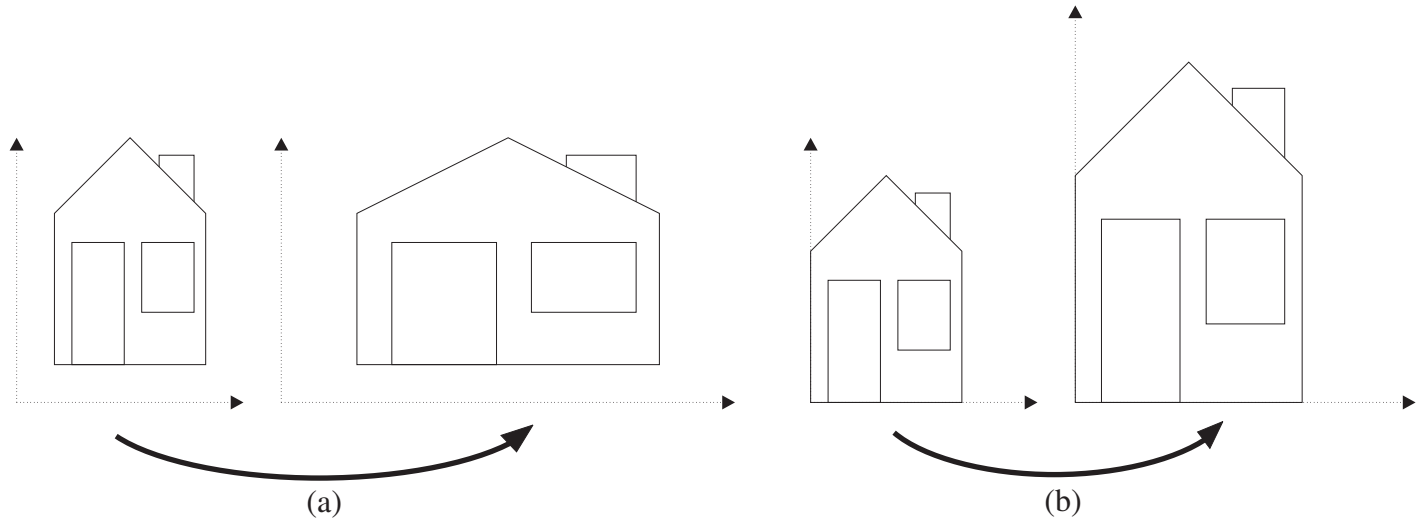


Figura 3.3: Transformação de escala

Quando um ou mais fatores de escala são negativos, dizemos que há espelhamento.

A transformação de escala, portanto, é representada por um conjunto de escalares (um para cada dimensão do espaço).

3.1.1.4 Cisalhamento

A transformação de cisalhamento (*shear*) não é um transformação primária, ela pode ser definida em termos das transformações anteriores [1], entretanto isso não é uma tarefa trivial e devido à sua utilidade, ela será vista separadamente. Uma transformação de cisalhamento tem o efeito de "puxar" um objeto em uma ou mais dimensões e não deve ser confundida com a rotação. Esta transformação é definida por um escalar (fator de cisalhamento) em cada dimensão que indica o quanto uma coordenada será transladada em função de outra coordenada em outra dimensão. Apesar de não ser

normalmente especificado em termos de um ângulo, esse fator pode ser facilmente associado a um ângulo, conforme mostra a figura 3.4:

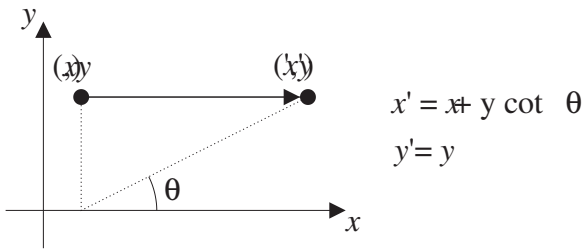


Figura 3.4: cisalhamento em *x*, associado a um ângulo

A figura 3.4 mostra um cisalhamento em *x*, em relação a *y*. Notamos que um ponto sofre uma translação em *x* que é diretamente proporcional à sua coordenada *y*.

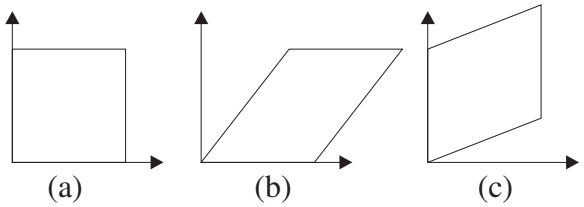


Figura 3.5: Efeito da transformação de cisalhamento 2D

Na figura 3.5, podemos observar o efeito do cisalhamento sobre o um quadrado (a), comparando-o com o resultado do cisalhamento em *x* (b) e em *y* (c).

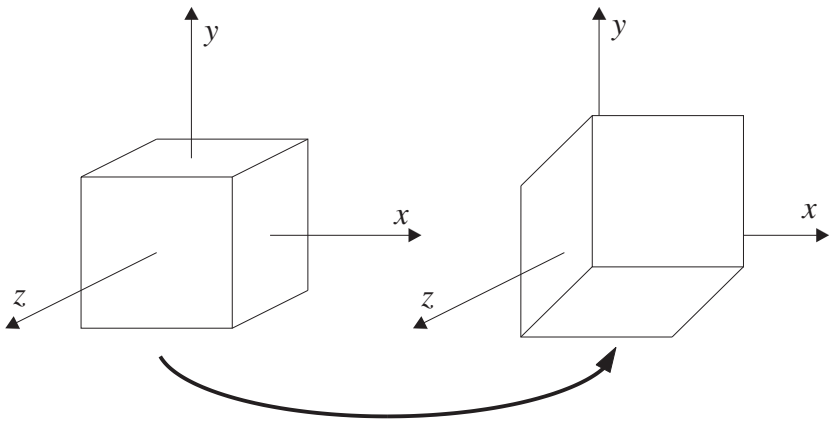


Figura 3.6: Efeito da transformação de cisalhamento 3D

Na figura 3.6, observa-se o efeito do cisalhamento em *x* e *y* em relação a *z*, sobre um cubo de centro na origem.

3.1.2 Representação

As transformações apresentadas são representáveis de maneiras diferentes, o que é ruim para sistemas de computador pois, no final, todas representam um mesmo tipo de informação: uma transformação.

Seria interessante que uma transformação pudesse ser tratada num programa como um objeto. Uma transformação deveria ser capaz de interagir com um ponto ou um vetor, gerando um novo ponto (ou vetor) que representaria a entidade inicial com suas características modificadas.

Vetores e pontos são um conjunto de coordenadas e portanto podem ser facilmente associados à estrutura de dados *vetor* (daí o nome ser o mesmo). As transformações poderiam ser representadas por matrizes (vetores de duas dimensões). Essas matrizes poderiam interagir com os vetores, produzindo novos vetores, o que nos leva à ideia de representar os pontos (e vetores) como matrizes de uma única linha ou coluna que interagem com as matrizes por meio de operações aritméticas tradicionais. Uma translação poderia ser representada pela soma de uma matriz de transformação com um ponto, enquanto que uma transformação de escala poderia ser representada pela multiplicação de uma matriz por um ponto.

É fácil imaginar a translação como soma de matrizes, enquanto a rotação e a escala podem ser facilmente imaginadas como multiplicação de matrizes. Entretanto, é desejável que todas as matrizes de transformação sejam de um mesmo formato e que todas as transformações possam interagir com pontos da mesma maneira (com a mesma operação aritmética).

Este objetivo pode ser alcançado se usarmos coordenadas homogêneas para representar os pontos e os vetores. Desta forma, temos quatro coordenadas para representar um ponto no espaço tridimensional. Em coordenadas homogêneas, convencionamos que o ponto (x, y, z, w) é o mesmo ponto no espaço que (x/w, y/w, z/w, 1), ou seja, (2, 1, 3, 1), (4, 2, 6, 2) e (1, 0.5, 1.5, 0.5) são representações do mesmo ponto. Um ponto cuja coordenada w é 0, pode ser entendido com um ponto no infinito. Podemos dizer que não existe posição associada a um ponto no infinito, então tal ponto passa a ser uma direção, ou seja, um vetor.

Usando coordenadas homogêneas, podemos representar qualquer transformação através de uma matriz 4x4 e a *pré-multiplicação* dessa matriz por um ponto (representado por uma matriz 4x1) resulta numa nova matriz 4x1 que representa o ponto depois da transformação. A grande vantagem dessa representação é que duas transformações podem ser combinadas pela multiplicação de suas matrizes, o que resulta numa nova matriz 4x4 que representa as duas transformações iniciais ao mesmo tempo (ver seção 3.1.4).

Alguns textos convencionam representar os vetores por matrizes linha que podem ser multiplicados à direita por matrizes de transformação. Essas representações para transformações são conhecidas por *matrizes de pós multiplicação*.

3.1.2.1 Matriz de Translação

Sabemos que, na transformação de translação, temos 3 fatores de translação (Tx, Ty, Tz), de tal forma que um ponto P (x, y, z), após sofrer uma translação T (Tx, Ty, Tz), vira o ponto P' (x', y', z') de tal forma que:

$x' = x + Tx$

$y' = y + Ty$

$z' = z + Tz$

Isso pode ser representado da seguinte forma:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.1.2.2 Matriz de Rotação

Para encontrar a fórmula da rotação, vamos primeiro analisar a figura 3.7, que demostra uma rotação no plano, onde o ponto P (x, y), após sofrer uma rotação em relação à origem vira o ponto P' (x', y'):

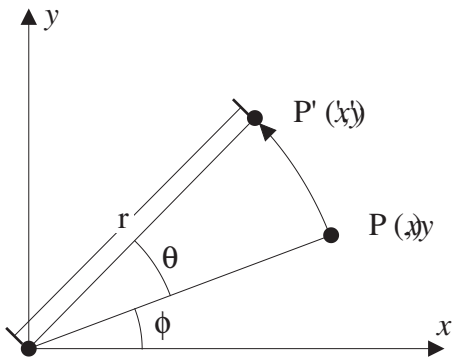


Figura 3.7: Rotação no plano, em torno da origem

Podemos identificar um segundo ângulo, na figura 3.7 (ϕ). Observa-se que:

$x = r \cdot \cos(\phi)$

$y = r \cdot \sin(\phi)$

e também que:

$x' = r \cdot \cos(\theta + \phi)$

$y' = r \cdot \sin(\theta + \phi)$

Logo:

$x' = r \cdot (\cos(\theta) \cdot \cos(\phi) - \sin(\theta) \cdot \sin(\phi))$

$y' = r \cdot (\sin(\theta) \cdot \cos(\phi) + \sin(\phi) \cdot \cos(\theta))$

Donde:

$x' = r \cdot \cos(\phi) \cdot \cos(\theta) - r \cdot \sin(\phi) \cdot \sin(\theta)$

$y' = r \cdot \cos(\phi) \cdot \sin(\theta) + r \cdot \sin(\phi) \cdot \cos(\theta)$

Portanto:

$x' = x \cdot \cos(\theta) - y \cdot \sin(\theta)$

$y' = x \cdot \sin(\theta) + y \cdot \cos(\theta)$

No espaço, esta é a rotação em z, que pode ser representado da seguinte forma:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Analogamente, temos a rotação em x pode ser representada assim:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

E a rotação em y:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.1.2.3 Matriz de Escala

Da transformação de escala, sabemos que:

$x' = s_x \cdot x$

$y' = s_y \cdot y$

$z' = s_z \cdot z$

De maneira que a transformação de escala pode ser representada assim:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.1.2.4 Matriz de Cisalhamento

Estamos interessados no cisalhamento (*shear*) no espaço, mas somente nas dimensões x e y. Cujo efeito será importante no processo de projeção. Sabemos que:

$x' = x + sh_x \cdot z$

$y' = y + sh_y \cdot z$

Esta transformação pode ser representada assim:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.1.3 Aplicação de transformações a objetos

A aplicação transformações a pontos ou vetores é uma operação calculada pela multiplicação de matrizes. Dado um vetor e uma transformação, podemos encontrar o vetor transformado. O uso de coordenadas homogêneas permite uma representação única para pontos e vetores, além de permitir uma representação única para qualquer transformações e uma única maneira de aplicar transformações. Observe que mesmo em casos mais extremos como a aplicação de uma translação a um vetor (direção)[1] esta abordagem é válida.

A aplicação de transformações à objetos (polígonos, sólidos, etc.) é feita aplicação da transformação à todos os elementos (pontos e vetores) que constituem o objeto. Considerendo que os objetos são tradicionalmente representados por descrições vetoriais, ou seja, uma série de parâmetros para primitivas, basta aplicar as transformações nesses parâmetros. Casos existam parâmetros que não são posições nem direções, é preciso encontrar uma representação para eles usando esses dois elementos. Por exemplo: distâncias podem ser entendidas como a magnitude de um vetor.

[1] Direções não "estão" em lugar algum e portanto transladar uma direção não teria nenhuma utilidade prática. Observe que qualquer translação aplicada a uma direção resulta na mesma direção.

3.1.4 Combinação de transformações

As matrizes de transformação vistas na seção 3.1 podem ser combinadas para produzir outras transformações mais sofisticadas. Assim, a partir de algumas poucas operações cujas matrizes são facilmente produzidas, conseguimos produzir qualquer transformação desejada.

Suponha que desejamos apliacar ao ponto P uma translação T seguida de uma rotação R, produzindo o ponto P'. Existe uma transformação que aplicada a P produz P'? Sabemos que $P' = R(TP)$ e pela associatividade da multiplicação de matrizes, temos que $P' = (RT)P$. Assim, se chamarmos RT de M, temos que $P' = MP$ e portanto a matriz M representa uma transformação equivalente à transladar e depois rotacionar.

Assim, a multiplicação de duas ou mais matrizes de transformação produz uma matriz que representa toda a sequência de transformações. Como usamos matrizes de pré-multiplicação, a sequência de transformações pode ser entendida como acontecendo da direita para a esquerda.

3.1.4.1 Rotação (2D) em torno de um ponto arbitrário

3.1.4.2 Escala fixando um ponto arbitrário

3.1.4.3 Transformação de instância

Com frequência queremos produzir uma sequência de transformações para acomodar um objeto modelado num sistema de coordenadas em outro sistema de coordenadas. Para tanto, pode-se esperar a necessidade de uma transformação de escala (para deixar o objeto com o tamanho adequado), uma ou mais transformações de rotação (para deixar o objeto com a orientação adequada) e uma transformação de translação (para deixar o objeto no local adequado).

A sequência mais adequada para produzir os resultados esperados é:

1. transformação de escala,
2. transformação (ou transformações) de rotação.
3. transformação de translação.

Essa sequência permite que os parâmetros sejam encontrados de forma "independente", sem necessidade de muita conta, provalvelmente mentalmente. Por exemplo, suponha que carro apresentado na figura 3.8-a deve ser colocado numa determinada posição do mundo (figura 3.8-b) com um tamanho adequado (bem menor nessa imagem) e com uma orientação adequada (diferente da original). Fazendo a transformação de instância na sequência indicada, sabemos que a escala é somente uma proporção entre o tamanho original e o tamanho final, a rotação é em torno da origem e é dada pelo ângulo entre a posição inicial e a final, além de que a translação é dada pela nova posição do carro (supondo a posição do carro como sendo a origem do sistema de coordenadas original).

Outras sequências de transformações seriam menos independentes e mais difíceis de calcular. Por exemplo, se a translação for feita primeiro e a escala depois, a escala também vai alterar a posição do carro.

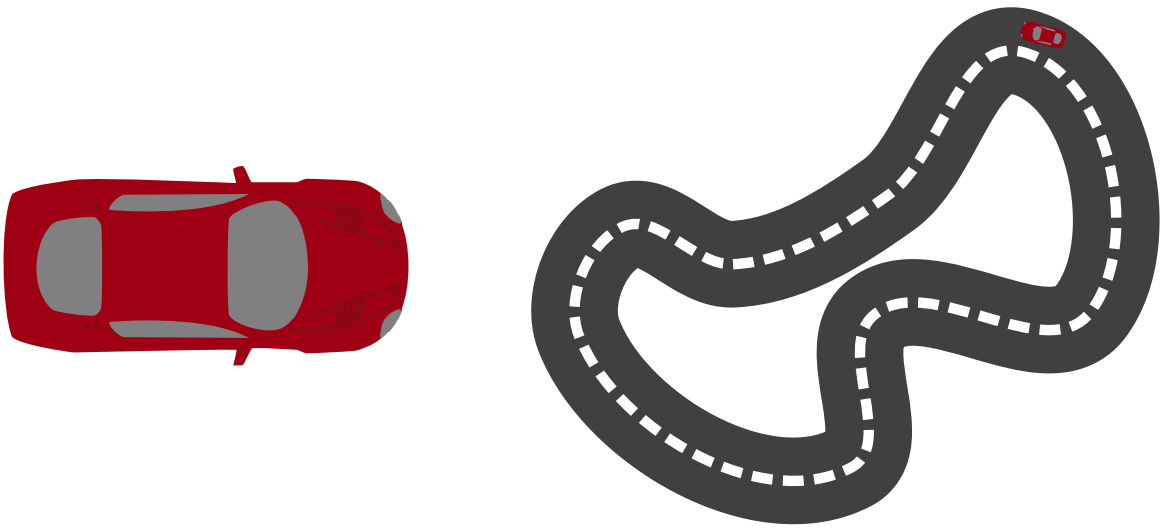


Figura 3.8: Caso de uso da transformação de instância.

3.1.4.4 Rotação (3D) em torno de um eixo arbitrário

3.1.5 Grafo de Cena

3.2 Mudanças Entre Sistemas de Coordenadas

As transformações vistas anteriormente podem ser vistas como mudanças de um sistema de coordenadas.

Na figura 3.9, vemos dois sistemas de coordenadas (SC1 e SC2), além de uma rotação em torno da origem que transforma SC2 em SC1. Essa mesma rotação transforma o ponto P no ponto P'. Além disso, notamos que as coordenadas de P em relação a SC2 são as mesmas de P' em relação a SC1.

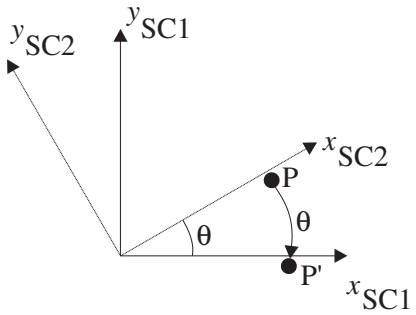


Figura 3.9: Uma mudança de sistema de coordenadas representada por uma transformação geométrica

Assim temos uma maneira prática de transformar coordenadas medidas em SC1 para coordenadas medidas em SC2: basta encontrar uma ou mais transformações geométricas que transformem SC2 em SC1. No exemplo acima, para saber as coordenadas de P de acordo com SC2, aplicamos em P, com relação à SC1, transformações que levariam SC2 em SC1.

Essas transformações são facilmente visíveis no plano, entretanto queremos trabalhar com sistemas de coordenadas de três dimensões. Um sistema de coordenadas de três dimensões pode ser transformado num outro sistema qualquer através de 6 transformações simples: uma translação em cada

dimensão mais uma rotação em cada dimensão. Isso porque trabalhamos com sistemas de coordenadas onde as 3 dimensões são perpendiculares entre si com sinal que respeita a regra da mão direita. Se pensarmos numa escala usada para fazer medições no sistema de coordenadas, precisaríamos de mais três transformações de escala (uma em cada dimensão), mas isto não será importante por enquanto. As três translações podem ser reduzidas numa só com fatores de translação nas três dimensões mas as três rotações serão analisadas separadamente, de forma que possamos utilizar a matrizes de rotação vistas na seção 3.1.2.2.

Para transformar um sistema de coordenadas em outro qualquer vamos então utilizar a seguinte sequência de transformações:

- Uma translação que leve a origem de SC2 na origem de SC1;
- uma rotação no eixo *y*, em torno da origem (relativa a SC1) que leve a direção de *x* em SC2 no plano *xy* de SC1. O valor absoluto do ângulo dessa rotação é dado pelo ângulo entre a projeção de *x* de SC2 no plano *xz* de SC1 e a direção de *x* em SC1;
- uma rotação no eixo *z*, em torno da origem (relativa a SC1) que leve a direção de *x* em SC2 (já transformada pelo passo 2) a ficar coincidente com a direção de *x* em SC1. O valor absoluto do ângulo dessa rotação é dado pelo ângulo entre a direção de *x* em SC2 (já transformada pelo passo 2) e a direção de *x* em SC1;
- uma rotação no eixo *x*, em torno da origem (relativa a SC1) que leve a direção de *y* em SC2 (já transformada pelos passos 2 e 3) a ficar coincidente com a direção de *y* em SC1. O valor absoluto do ângulo dessa rotação é dado pelo ângulo entre a direção de *y* em SC2 (já transformada pelos passos 2 e 3) e a direção de *y* em SC1.

Uma observação importante é que as direções das dimensões de SC2 devem ser conhecidas em coordenadas medidas em SC1, para que seja possível calcular os ângulos mencionados acima.

Outras sequências de transformações (com outros valores) também podem transformar um sistema de coordenadas em outro (sempre uma translação e três rotações no máximo). Qualquer sequência pode ser usada.

3.3 Projeção

Conforme comentado anteriormente, para gerar a imagem de um objeto 3D, precisamos converter as coordenadas 3D em coordenadas 2D que correspondem a uma visão específica do objeto. Esse processo é chamado de projeção.

Existem dois tipos de projeção: a perspectiva e a paralela. A projeção perspectiva é aquela que acontece no processo de formação de imagens em nossos olhos ou numa câmera fotográfica (ver Figura 1.5), por isso é a que gera imagens mais realistas. A projeção paralela pode ser vista como uma projeção perspectiva onde o centro de projeção está no infinito.

Uma projeção pode, portanto, ser definida por um plano de projeção e um centro de projeção (ponto) ou direção de projeção (vetor). O plano de projeção no entanto, precisa estar relacionado com um sistema de coordenadas, ou seja, não basta que o plano esteja definido, é preciso ser capaz de se representar qualquer posição nesse plano em relação a um referencial próprio.

Conforme podemos observar na figura 3.10, na projeção paralela, as linhas que unem os pontos A e B às suas projeções A' e B' são paralelas, isto faz com que o segmento projetado tenha o mesmo tamanho para qualquer distância entre o plano de projeção e o objeto. Esta propriedade da projeção

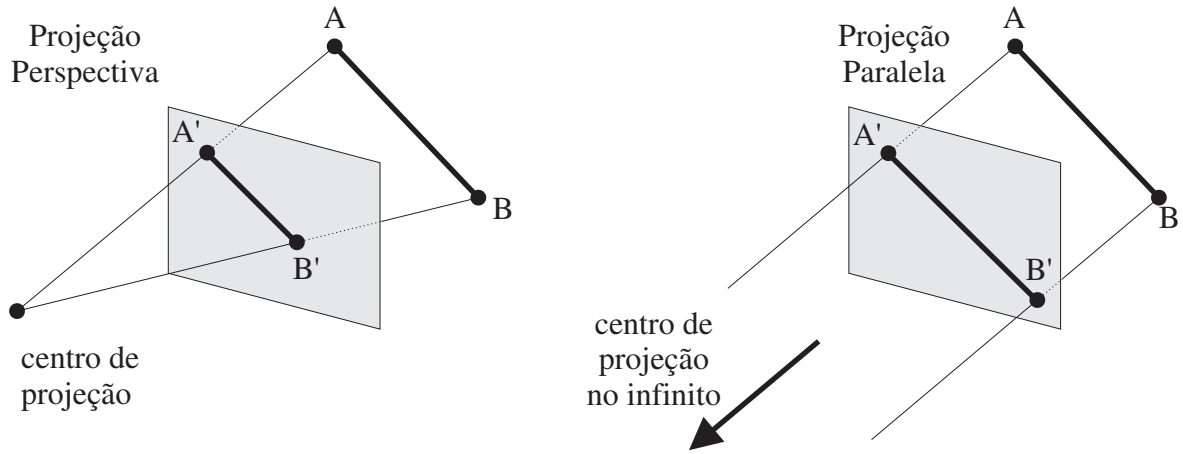


Figura 3.10: Projeção perpsectiva e projeção paralela

paralela é bastante desejável em áreas como engenharia ou arquitetura porque torna possível que alguém possa calcular as dimensões do objeto desenhado a partir de seu desenho.

No processo de síntese de imagens, o plano de projeção será uma representação de algum dispositivo de apresentação (ex.: vídeo ou papel num impressora), por isso é importante que exista um sistema de coordenadas embutido no plano de projeção (PP), de maneira que, a partir da coordenadas de pontos de um objeto, descritas no sistema de coordenadas do mundo (SCM), seja possível encontrar coordenadas das projeções desses pontos num sistema de coordenadas de apresentação (SCA). Essas novas coordenadas serão usadas para enviar comando de desenho para dispositivo de apresentação, formando a imagem desejada.

3.3.1 Projeção Paralela

Para encontrar uma fórmula que nos permita fazer uma projeção paralela qualquer, vamos primeiro imaginar um projeção paralela simples o suficiente para que o cálculo da projeção seja trivial.

Tal projeção poderia ser aquela definida por um PP que coincide com o plano XY do SCM de tal forma que a origem do PP coincida com a origem do SCM, a direção de X e de Y do SCA coincidam com a direção de X e de Y do SCM, e a escala nos dois sistemas seja a mesma. A direção de projeção seria paralela à direção do eixo Z do SCM, com o vetor (0, 0, -1), por exemplo.

Neste caso fica visível que a projeção do ponto (x, y, z) com coordenadas medidas no SCM tem coordenadas (x, y) no SCA.

Na figura 3.11 observamos que P tem as mesmas coordenadas x e y de P', mesmo considerando que as coordenadas de P são medidas de acordo com SCM e as coordenadas de P' são medidas em SCA.

Esta não é a única projeção paralela cujo resultado é trivial, entretanto, ela é suficiente para desenvolvermos uma fórmula para uma projeção qualquer. Quando uma projeção paralela não estiver de acordo com este caso especial, aplicaremos em P uma série de transformações que transformariam SCA em SCM. Os valores das coordenadas x e y de P após essas transformações serão os valores das coordenadas x e y de P' em SCA, conforme visto na seção 3.2.

Existem várias maneiras de aplicar um conjunto de transformações geométricas simples, numa determinada ordem, de maneira a resolver esse problema. Na figura 3.12 vemos dois dos caminhos mais usados:

Uma seqüência de transformações pode ser representada pela multiplicação das matrizes de transfor-

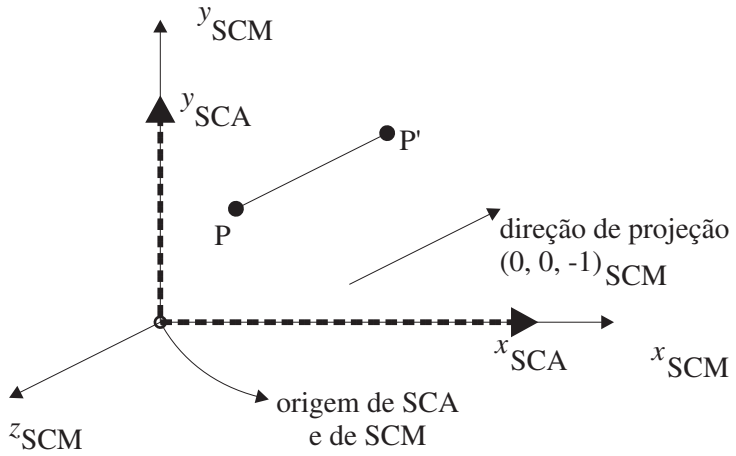


Figura 3.11: Projeção Paralela Especial

mação. Com isso obtemos uma matriz 4x4 que representa a projeção paralela desejada. Essa matriz será aplicada em cada ponto do objeto cuja imagem deseja-se criar.

3.3.2 Projeção Perspectiva

Da mesma forma que no caso da projeção paralela, pretendemos encontrar uma projeção perspectiva com características especiais que permitam encontrar seu resultado de maneira trivial. Tal projeção poderia ser:

Na figura 3.13 (a), as coordenadas (x', y') que representam o ponto P' de acordo com SCA podem ser encontradas por semelhança de triângulos como mostrado em (b), que representa a mesma situação, vista de cima do eixo y. Portanto, sabemos que:

$x' = x \cdot z' / z$

$y' = y \cdot z' / z$

O que pode ser representado pela matriz de transformação:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/z' & 0 \end{bmatrix}$$

A matriz acima altera o valor da coordenada w de um ponto e portanto altera as coordenadas x, y e z. Infelizmente a alteração da coordenada z não era desejável, entretanto, essa coordenada não tem importância no resultado da projeção.

Queremos agora, encontrar um conjunto de transformações simples que transformariam uma projeção perspectiva qualquer na projeção perspectiva especial que acabamos de ver. Na figura 3.14, observamos duas das maneiras mais utilizadas:

Assim como na projeção paralela, podemos obter uma matriz de projeção perspectiva, multiplicando as matrizes de cada transformação mais simples, sem esquecer da matriz de projeção perspectiva especial que eqüivale à aplicação da regra de três.

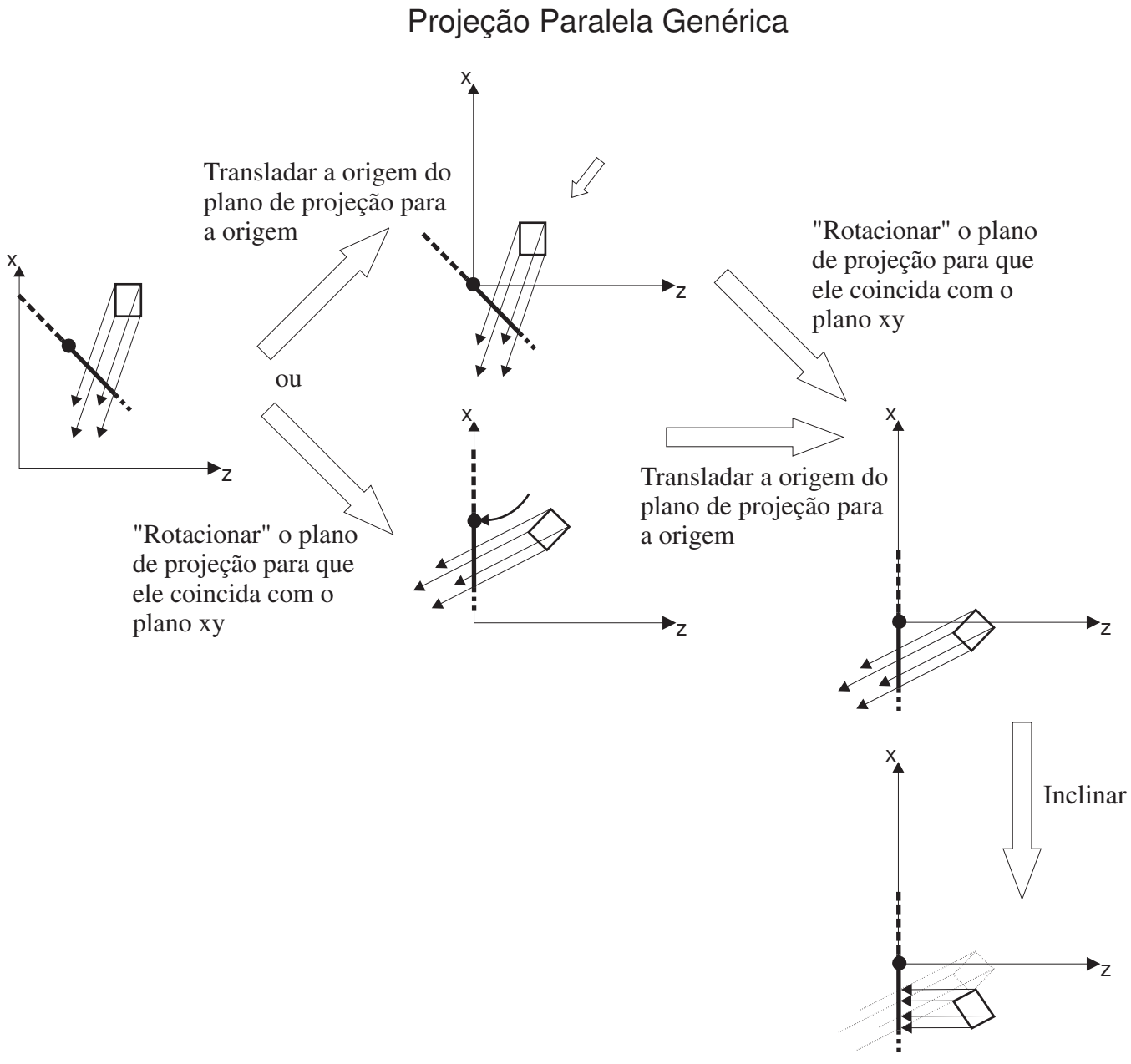


Figura 3.12: Transformação de uma projeção paralela genérica numa projeção paralela especial

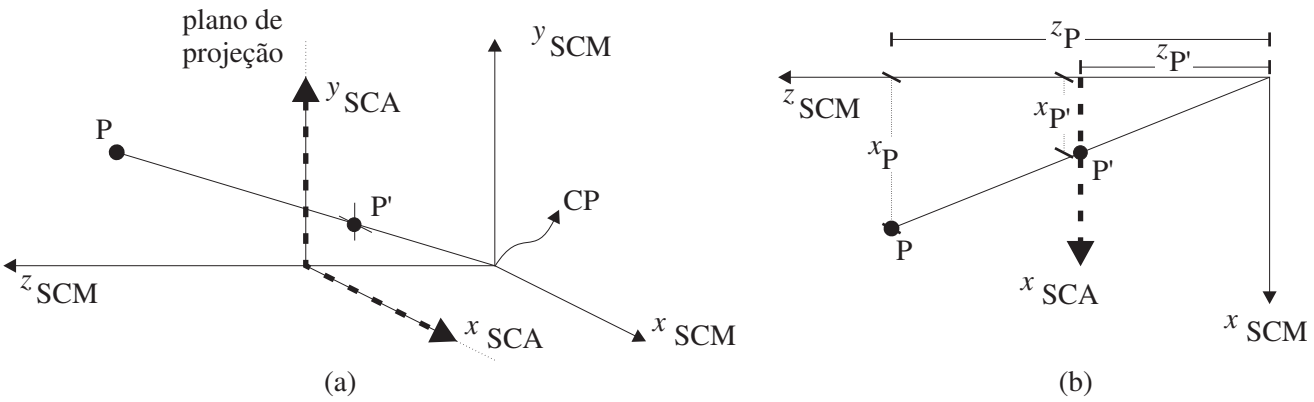


Figura 3.13: Projeção Perspectiva Especial

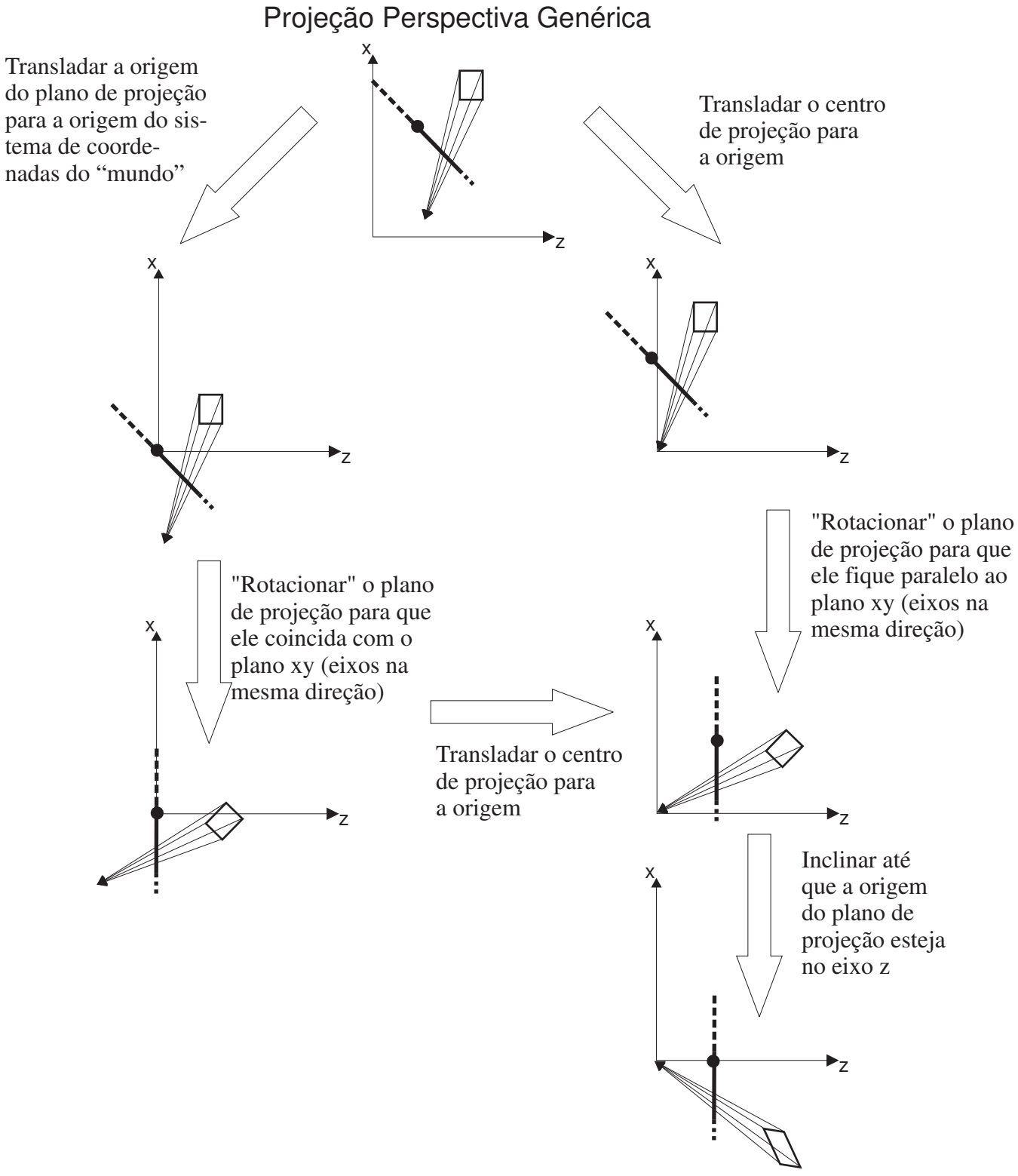


Figura 3.14: Transformação de uma projeção perspectiva genérica numa projeção perspectiva especial

Capítulo 4

Métodos de Recorte

Recorte é o nome dado ao processo de eliminação de partes de um desenho que estão fora de uma área de interesse. Por exemplo: Pode ser necessário apresentar uma imagem de maneira que ela não caiba totalmente na janela de apresentação. Para evitar erros que podem ser causados pela tentativa de se desenhar algo numa região inválida, determina-se o que é que pode realmente ser desenhado.

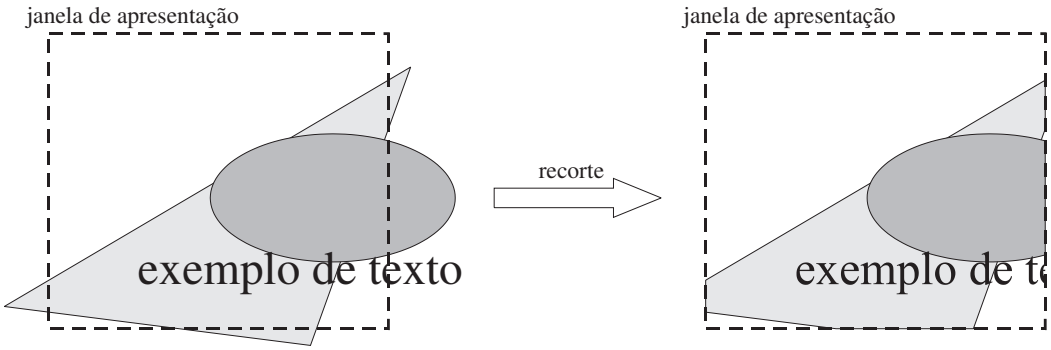


Figura 4.1: Exemplo de recorte

A primitiva de desenho mais importante é o segmento de reta, pois qualquer outra forma ou primitiva pode ser aproximada por um conjunto de segmentos de reta. Abaixo veremos alguns do principais algoritmos para recorte de segmentos de reta.

4.1 Algoritmos de Recorte para Segmentos de Reta

Conjuntos de segmentos de reta podem ser usados como aproximação para outros elementos gráficos. Sabendo recortar segmentos de reta, grande parte dos problemas de recorte podem ser resolvidos.

4.1.1 Algoritmo de Cohen-Sutherland

Este algoritmo realiza testes iniciais para verificar quando é necessário calcular pontos de interseção com o retângulo de recorte. Alguns casos onde o segmento de reta está totalmente dentro do retângulo de recorte ou totalmente fora são facilmente reconhecidos, não havendo necessidade de se realizar cálculos, o que pode significar um grande ganho de tempo, nos casos em que o retângulo de recorte é muito pequeno ou muito grande. Se ele for muito pequeno, poucas interseções precisam ser calculadas, se ele for muito grande, acontece o mesmo.

Para realizar esses testes iniciais, divide-se o plano do retângulo de recorte em 9 regiões. Cada região é rotulada com um código binário conforme a figura 4.2:

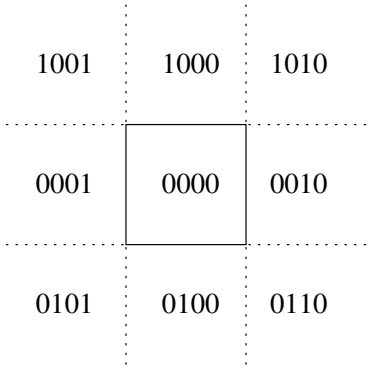


Figura 4.2: Códigos das regiões de recorte.

O primeiro bit (o bit mais à esquerda) será 1 se e somente se a região rotulada estiver acima do retângulo de recorte. O segundo bit será 1 quando a região estiver abaixo do retângulo de recorte. O terceiro bit será 1 se ela estiver à direita e quarto será 1 se ela estiver à esquerda. Para realizar o recorte de um segmento de reta, classifica-se as duas extremidades do segmento de acordo com o rótulo da região onde a extremidade se encontra.

Se ambos os códigos forem zero, o segmento está todo dentro da região de recorte e não é necessário calcular nenhum ponto de interseção. Além disso, se o resultado de um operação de "e" lógico bit a bit for diferente de zero, então os dois códigos tem um bit em comum e portanto o segmento está totalmente fora do retângulo de recorte (veja Figura 4.3).

Se nenhum desses dois testes for verdadeiro, o segmento pode estar interceptando o retângulo de recorte ou pode estar totalmente fora dele. Neste caso, devemos calcular uma interseção do segmento de reta com um dos lados do retângulo de recorte. Os dois novos segmentos de retas resultantes do cálculo desta interseção serão recortados recursivamente pelo algoritmo, até que cada segmento resultante esteja totalmente dentro ou totalmente fora do retângulo de recorte.

No pior dos casos, quatro interseções deverão ser calculadas para se encontrar o recorte de um segmento, além disso a maneira como os limites são escolhidos para cálculo da interseção afeta o desempenho do algoritmo, por exemplo, no caso da 4.3, calcular a interseção com o lado de baixo (ao invés do lado de cima), seria um desperdício. Porém, o cálculo da interseção com o lado direito poderia ser feito, já que a interseção está dentro do segmento de reta.

Para escolher um lado da área de recorte que forneça um ponto dentro do segmento, para consultar os bits do código de cada ponto. Sabemos que um dos códigos é diferente de zero, então olharemos cada bit desse código até encontrar um que seja igual a 1. O lado correspondente a esse bit é um bom candidato a produzir uma interseção útil. Se você não lembra como calcular uma interseção, reveja a seção 1.7.7.

Calculada a interseção existe um problema de classificação. Ainda conforme o exemplo da figura, existem duas possibilidades para a classificação da interseção:

1. a interseção é classificada como dentro da área de recorte, fazendo com que o segmento superior da subdivisão precise ser recortado com o lado superior novamente e

2. a interseção é classificada como acima da área de recorte, fazendo com que o segmento inferior do da subdivisão precise ser recortado com o lado superior novamente.

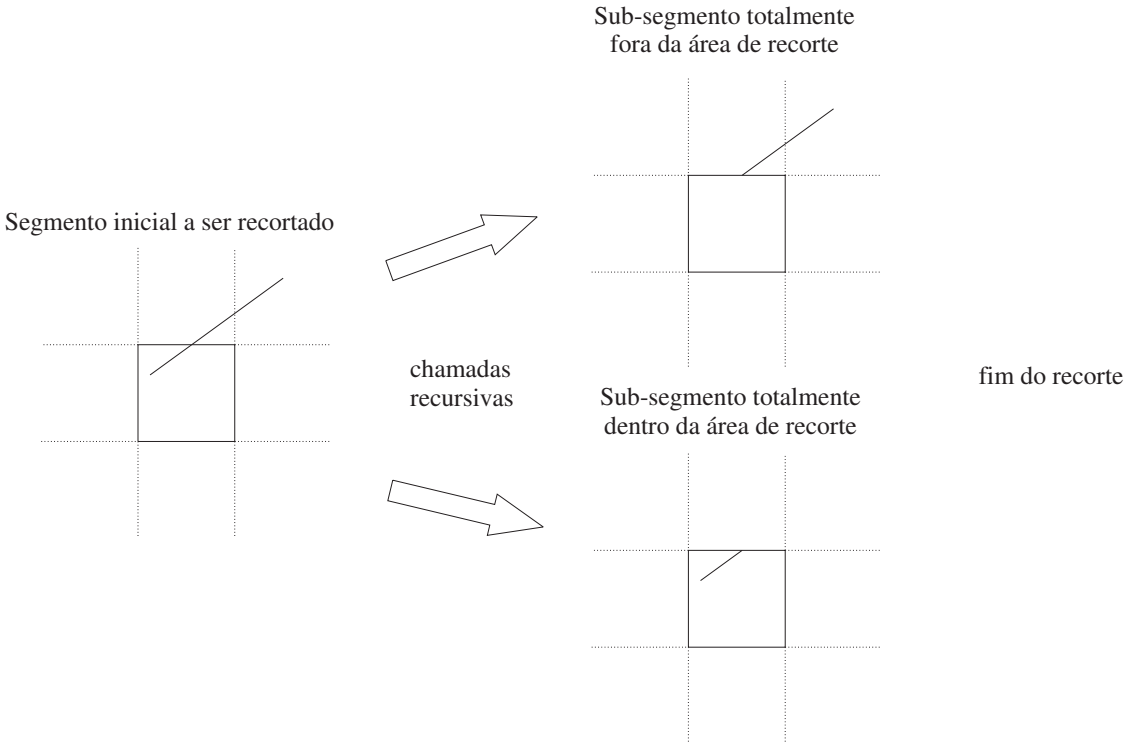


Figura 4.3: Procedimento de recorte de acordo com o algoritmo Cohen-Sutherland.

Dessa forma, uma implementação desenvolvida com pouca atenção nunca terminaria. Esse problema pode ser resolvido fazendo com que a interseção calculada seja classificada com dois códigos diferentes. Ao ser associada com o ponto acima da área de recorte, o bit correspondente da interseção seria 1, proporcionando o descarte da parte superior da subdivisão. Ao ser associada com o ponto abaixo do limite superior da área de recorte, o bit correspondente da interseção seria 0, proporcionando a classificação da parte inferior da subdivisão como totalmente dentro da área de recorte.

Entretanto, é facil ver que com o cálculo de uma interseção, uma das subdivisões do segmento de reta pode ser diretamente descartada (aquela que a interseção forma com o ponto cujo código é diferente de zero e foi usado para escolha do lado a ser usado no cálculo da interseção). Assim, percebe-se que cada segmento de reta gera um único novo segmento, menor, a ser recortado e a implementação pode ser facilmente transformada numa iteração.

4.1.2 Recorte Paramétrico

Em 1978, Cyrus e Beck propuseram um outro algoritmo, geralmente mais eficiente do que o algoritmo de Cohen-Sutherland. Este algoritmo pode ser usado no recorte de segmentos de reta no plano, e também no recorte de segmentos de retas no espaço, em relação a um sólido convexo. Veremos o caso 2D.

Vamos representar o segmento de reta que vai do ponto P_0 a P_1 pela equação paramétrica da reta:

$P(t) = P_0 + (P_1 - P_0)t$

Onde t varia de 0 a 1, sendo 0 no ponto P_0 e 1 no ponto P_1. Seja P_{Li} um ponto aleatório sobre um dos lados da área de recorte (lado Li). Um ponto qualquer do segmento a ser recortado pode estar em uma de três situações possíveis: fora da área de recorte, sobre o lado Li ou dentro da área de recorte. Podemos determinar em qual dessas três situações um ponto $P(t)$ está, calculando o produto escalar

entre o vetor que vai de P_{Li} até $P(t)$ e a *normal ao lado i* (N_i), conforme mostra a figura 4.4[1]. O valor do produto escalar será positivo se $P(t)$ estiver fora da área de recorte, será zero se estiver sobre o lado e será menor que zero se estiver dentro da área de recorte.

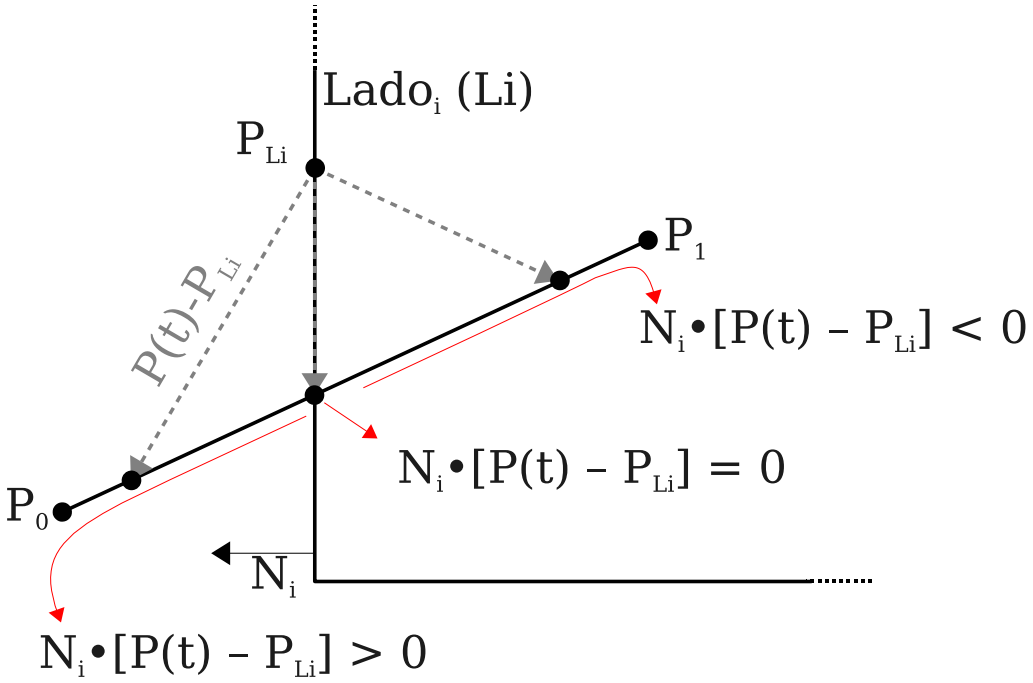


Figura 4.4: Determinação da interseção de um segmento com um lado da área de recorte.

Mais ainda, podemos usar o produto escalar para determinar qual é o ponto que está sobre o lado *Li*, ou seja, qual a interseção entre o segmento e o lado da área de recorte. Para tanto, basta isolar o valor de t na equação do ponto que está na interseção. Desta forma, temos:

$N_i \cdot [P(t) - P_{Li}] = 0$

Substituindo o valor de P(t):

$N_i \cdot [P_0 + (P_1 - P_0)\, t - P_{Li}] = 0$

A seguir, agrupamos termos:

$N_i \cdot [(P_0 - P_{Li}) + (P_1 - P_0)\, t] = 0$

Distribuindo o N_i, temos:

$N_i \cdot [P_0 - P_{Li}] + N_i \cdot [P_1 - P_0]\, t = 0$

Chamamos o vetor $(P_1 - P_0)$ de D:

$N_i \cdot [P_0 - P_{Li}] + N_i \cdot Dt = 0$

E isolamos o t do lado esquerdo:

$$t = \frac{N_i \cdot [P_0 - P_{Li}]}{-N_i \cdot D}$$

Observe que a equação acima é válida somente se o denominador da equação for diferente de zero. Isso poderia acontecer se o segmento a ser recortado for paralelo à linha de recorte (ou se $P_0 = P_1$). Se o denominador for igual a zero, podemos verificar o sinal do numerador. Se ele for positivo, então P_0 está fora da área de recorte (reveja a Figura 4.4), portanto todo o segmento está fora da área de

[1]Note que essa classificação é relativa a um dos lados da área de recorte, e não relativa à área toda.

recorte e deve ser descartado. Se o numerador for negativo, deixaremos que o cálculo das interseções com os outros lados da área de recorte prossigam.

Temos portanto uma fórmula para calcular a interseção entre um segmento qualquer e um dos lados da área de recorte. Esta fórmula produz a descrição de um ponto por meio de seu parâmtero t e não por meio de suas coordenadas. Poderíamos calcular todos os valores de t e usá-los para calcular as coordenadas dos pontos de interseção e consequentemente o segmento recortado, porém ainda é preciso diferenciar quais valores de t são úteis e quais não são. Teremos até quatro de valores de t resultantes dos cálculos, mas só alguns deles são realmente úteis para determinar o resultado do recorte. Queremos que essa determinação seja feita com um mínimo de processamento.

O primeiro passo é classificar cada ponto de interseção como *potencialmente entrando* (PE) ou *potencialmente saindo* (PS), ou seja, queremos saber se o segmento de reta sendo recortado parece entrar ou sair da área de recorte pelo lado em questão. Isso pode ser feito pelo sinal do denominador, que indica se o ângulo dos vetores em questão é maior ou menor que 90°. O sinal do denominador tem sinal contrário ao do produto escalar e portanto contrário ao seno do ângulo. O segmento está potencialmente saindo se o denominador for menor que zero e vice-versa.

Valores de t PE devem ser descartados se forem anteriores a um ponto de entrada conhecido, já que o segmento não pode entrar duas vezes na área de recorte. Se o valor de t for posterior à um ponto de saída do segmento, o segmento *todo* está fora da área de recorte, já que o segmento não pode primeiro sair da área de recorte e depois entrar.

De forma análoga, valores de t PS devem ser descartados se forem posteriores a um ponto de saída conhecido e o segemento *todo* deve ser descartado se um t PS for anterior a um ponto de entrada do segmento.

Os pontos de entrada e saída do segmento na área de recorte podem ser inicializados respectivamente com valores de 0 e 1 para t. Assim somente valores entre 0 e 1 serão usados e os pontos P_0 e P_1farão parte do resultado a menos que pontos mais internos ao segmento sejam encontrados.

Ao final do recorte em relação aos quatros lados, teremos um valor de t para o ponto de entrada, cujas coordenadas devem ser calculadas se $t > 0$ (se $t = 0$, o ponto de entrada é P_0 e portanto tem coordenadas conhecidas). Teremos também um valor de t para o ponto de saída, cujas coordenadas devem ser calculadas se $t < 1$. Assim não são calculadas coordenadas que não façam parte do resultado. O algoritmo 4.1 formaliza uma forma de estabelecer a sequência dos testes e cálculos.

A fim de fixar a ideia deste algoritmo, a figura 4.5 apresenta vários casos possíveis de recorte de um segmento de reta em relação a um retângulo de recorte, junto com uma classificação PE/PS para cada interseção. Procure verificar mentalmente a sequência de testes em cada caso para garantir que você entendeu o processo.

Em 1984 este algoritmo foi aperfeiçoado por Liang e Barsky tornando-se mais eficiente nos casos em que o volume de recorte é um retângulo alinhado com os eixos do sistema de coordenadas, que são justamente os casos que nos interessam. Considerando esta situação especial, sabemos que a normal N_i sempre tem uma coordenada com valor 0, o que significa que o produto escalar $N_i \cdot (P_0 - P_{Li})$ não precisa considerar a coordenada correspondente no vetor $(P_0 - P_{Li})$, justamente a coordenada que precisaria ser aleatóriamente escolhida. Exemplo: para o limite esquerdo do retângulo de recorte, $N_i = (-1, 0)$, logo não precisamos considerar a coordenada y (indeterminada para uma linha vertical). O numerador do cálculo de t fica então $-1(x_0 - x_{min}) = x_{min} - x_0$.

Da mesma forma, o denominador do cálculo de t, fica reduzido a $\pm dx$ ou $\pm dy$. A tabela 4.1 mostra a fórmula para o cálculo de t em cada um dos lados de um retângulo alinhado com o sistema de coordenadas. Essas fórmulas representam uma boa economia de operações aritméticas em relação à formula geral apresentada antes.

Algoritmo 4.1 Recorte Paramétrico.

1. Inicializar a posição de início com 0.
2. Inicializar a posição de fim com 1.
3. Para cada lado da área de recorte:
3.1. Calcular o númerador e o denominador da fórmula de t.
3.2. Se o segmento for paralelo ao lado de recorte, então
3.2.1. Se o segmento estiver fora da área em relação ao lado, então
3.2.1.1. Responder com segmento vazio.
senão:
3.2.2. Calcular o valor de t.
3.2.3. Se o segmento está potencialmente entrando, então
3.2.3.1. Se a interseção for posterior à posição de fim, então
3.2.3.1.1. Responder com segmento vazio.
3.2.3.2. Se a interseção for posterior à posição de início, então
3.2.3.2.1. Atualizar a posição de início.
senão:
3.2.3.3. Se a interseção for anterior à posição de início, então
3.2.3.3.1. Responder com segmento vazio.
3.2.3.4. Se a interseção for anterior à posição de fim, então
3.2.3.4.1. Atualizar a posição de fim.
4. Calcular as coordenadas da posição de início, se não forem as de P_0.
5. Calcular as coordenadas da posição de fim, se não forem as de P_1.
6. Responder com um segmento dado pelas coordenadas calculadas em (4) e (5).

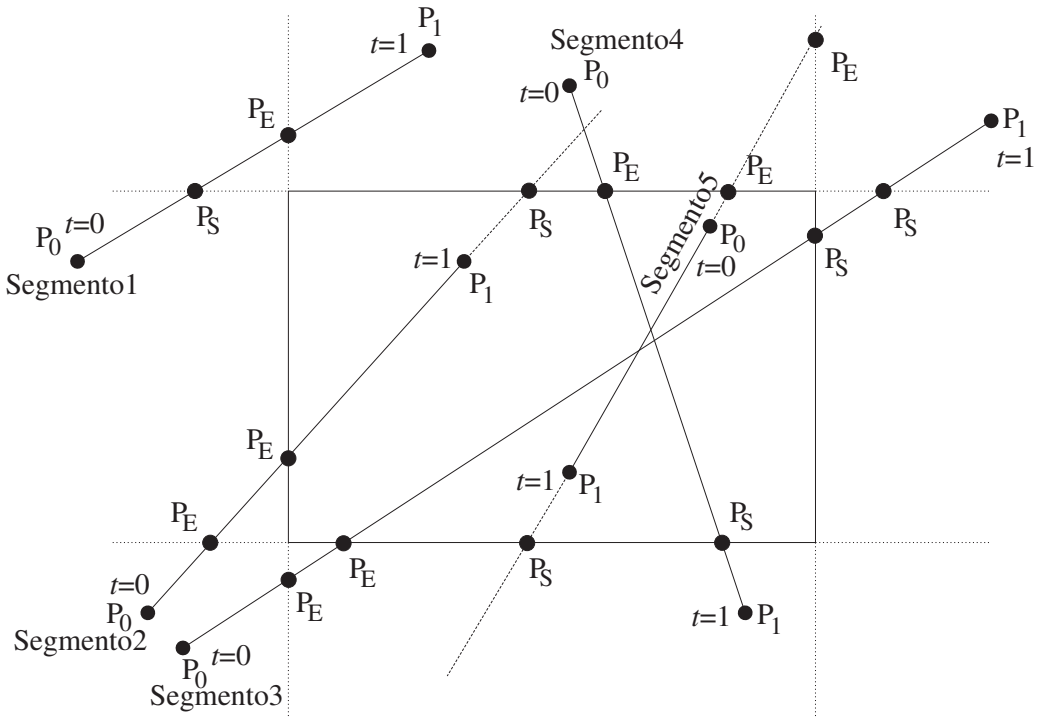


Figura 4.5: Exemplos de recorte paramétrico.

lado esquerdo	lado direito	lado superior	lado inferior
$\frac{x_{min}-x_0}{x_1-x_0}$	$\frac{x_0-x_{máx}}{x_0-x_1}$	$\frac{y_0-y_{máx}}{y_0-y_1}$	$\frac{y_{min}-y_0}{y_1-y_0}$

Tabela 4.1: Fórmulas para o cálculo das interseções contra lados alinhados com os sistemas de coordenadas.

4.2 Recorte de Círculos e Elipses

O recorte de círculos e elipses é bem complexo, e por isso pode ser interessante fazê-lo ponto a ponto. Em casos onde apenas uma pequena parte encontra-se dentro do retângulo de recorte, é interessante implementar testes simples para verificar se círculo esta todo dentro ou todo fora da área de recorte, se estes testes falharem o círculo novos testes pode ser feitos para cada quadrante, sendo possível refiná-los até o nível de octantes. O mesmo vale para as elipses, exceto que os testes ao nível de octante não seriam válidos.

4.3 Recorte de Polígonos

Um algoritmo para recorte de polígonos deve tratar uma grande variedade de casos. O recorte de um polígono côncavo pode dividí-lo em dois ou mais polígonos resultantes. Esta tarefa deve ser divida em tarefas mais simples, de maneira que ao final da resolução de vários problemas menores a tarefa principal esteja terminada.

4.3.1 Algoritmo de Sutherland-Hodgman

Este algoritmo resolve o problema do recorte de um polígono em relação a uma reta. O recorte do polígono em relação ao retângulo de recorte pode então ser resolvido através do recorte em relação a cada um de seus quatro lados, conforme mostra a figura 4.6.

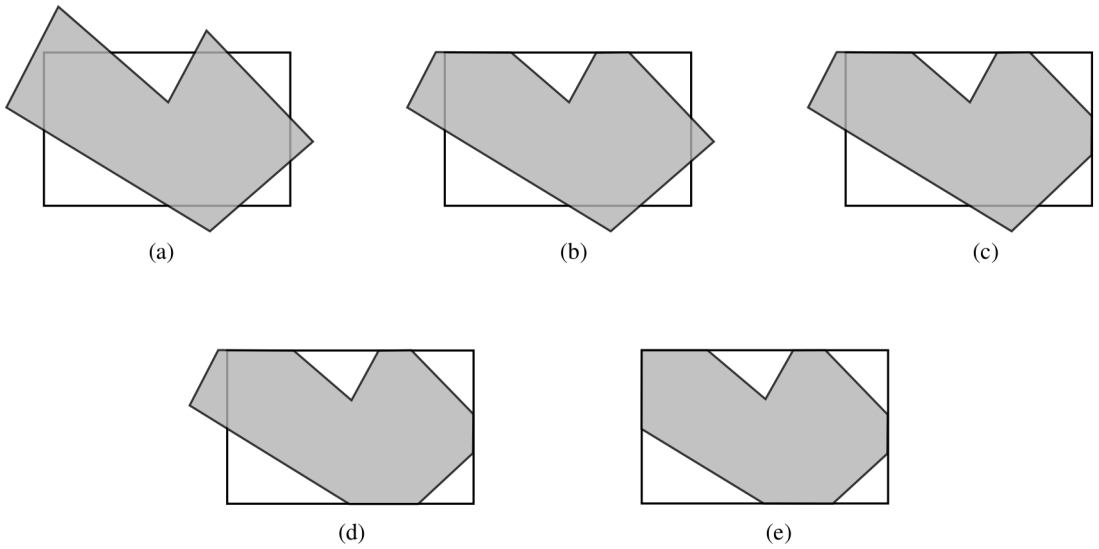


Figura 4.6: (a) Polígono original. (b) Recorte do lado superior. (c) Recorte do lado direito. (d) Recorte do lado inferior. (e) Recorte do lado esquerdo.

O recorte de um polígono com uma reta é feito através da análise de cada face do polígono. Cada face é analisada por suas duas extremidades, de maneira que o polígono todo é analisado vértice a vértice, assim: primeiro os vértices 1 e 2, depois os vértices 2 e 3, etc... Desta maneira, cada face do polígono pode ser classificada como “entrando”, “dentro”, “saindo” ou “fora” da área de recorte, conforme visto na figura 4.7.

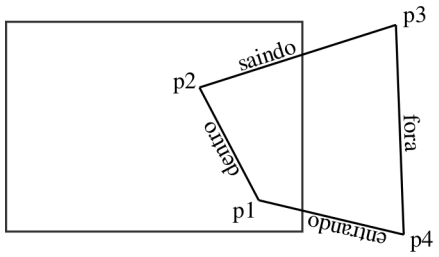


Figura 4.7: Classificação dos lados de um polígono em relação à área de recorte.

O algoritmo recebe uma lista de pontos e retorna uma nova lista de pontos que define o polígono. Para cada dois pontos analisados, se a face do polígono for classificada como “dentro” o ponto de destino é colocado na nova lista de vértices (a). Se a face estiver “saindo”, o ponto de interseção é colocado na nova lista de vértices (b). Se a face estiver “fora”, nenhum ponto é colocado na nova lista de vértices (c). Finalmente, se a face estiver “entrando”, tanto o ponto de interseção quanto o de destino são colocados na nova lista de vértices, conforme a figura 4.8.

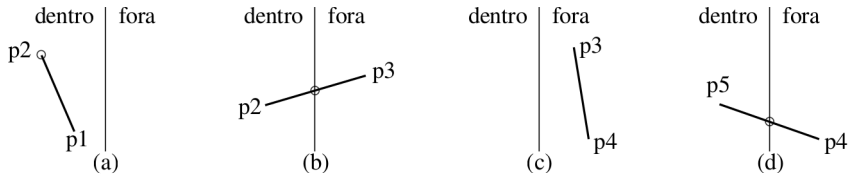


Figura 4.8: Pontos que resultam do recorte de um lado do polígono contra um lado da área de recorte.

O problema deste algoritmo é que se o polígono for dividido em novos polígonos, surgirão lados indesejáveis que precisam ser removidos por algum processamento posterior. A classificação e o cálculo das interseções podem ser feitos como no algoritmo de Liang-Barsky. O algoritmo não precisa de espaço temporário para o armazenamento de recortes parciais no polígono se implementado numa espécie de *pipeline*, onde os valores resultantes do primeiro corte são usados como entrada para o segundo e assim por diante. Esta característica torna interessante a implementação deste algoritmo por *hardware*.

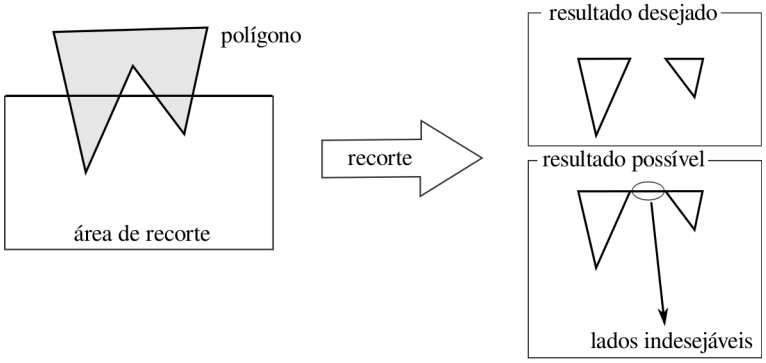


Figura 4.9: O recorte de polígonos côncavos pode criar o surgimento de lados indesejáveis.

4.4 Recorte em 3D

Para a visualização de objetos 3D no plano, pode ser feita uma projeção 3D $\rightarrow$ 2D seguida de um recorte ou pode ser feito um recorte 3D seguido de uma projeção. A segunda alternativa é preferível porque não se perde tempo fazendo a projeção de segmentos que serão descartados mais tarde no processo de recorte.

O recorte em 3D pode ser feito tanto pelo algoritmo de Cohen-Sutherland quanto pelo algoritmo de Cyrus-Beck, estendidos em uma dimensão a mais.

Capítulo 5

Conversão de Primitivas Gráficas em Imagens Matriciais

A conversão de primitivas gráficas em imagens matriciais é informalmente chamada de *rasterização*, porque imagens matriciais são chamadas de *raster images* em inglês. Este processo está intimamente ligado com o hardware, pois suas características físicas (como a distância entre *pixels*) afetam o desenho formado. Para facilitar o desenvolvimento de algoritmos, vamos supor que o *pixel* de coordenadas (x, y) é um círculo cujo centro está em (x, y), que o limite de um *pixel* toca o limite de seus quatro vizinhos horizontais e verticais, e que todos os *pixels* tem o mesmo tamanho.

5.1 Segmentos de Reta

É desejável produzir segmentos de reta que sejam:

- tão finos quanto possível,
- simétricos,
- tão contínuos quanto possível,
- de espessura aparente constante.

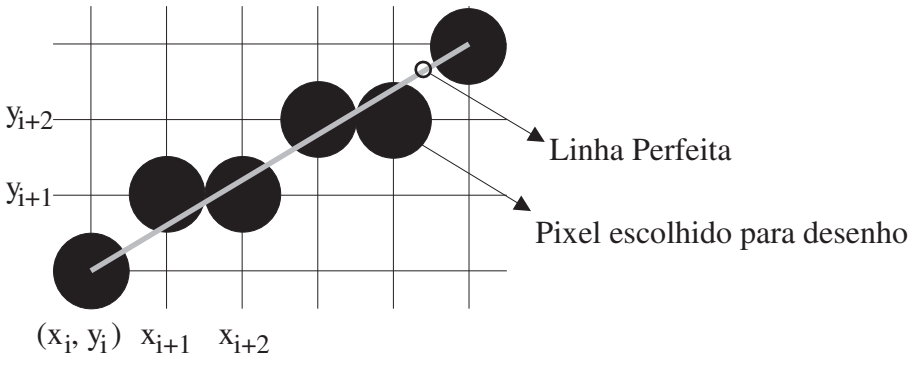


Figura 5.1: Escolha de pixels para desenhar uma reta.

Considerando que a linha está no primeiro octante (ela começa embaixo, à esquerda e vai até à parte de cima à direita, com inclinação entre 0 e 1) e seu ponto inicial é (x_i, y_i), começamos desenhando o seu ponto inicial e então resta escolher entre acender o ponto ao lado (x_{i+1}, y_i) e o ponto da diagonal (x_{i+1}, y_{i+1}). Escolheremos aquele que está mais próximo da reta, matematicamente falando.

O problema é conseguir um algoritmo eficiente para calcular essa distância e então fazer a escolha. Bresenham, em 1965, propôs um algoritmo incremental que usa somente variáveis inteiras, tornado-o bastante eficiente. Esse algoritmo sofreu pequenas modificações para abranger casos que vão além das simplificações adotadas aqui sobre o dispositivo de apresentação.

Para explicar a ideia geral do algoritmo, vamos considerar a linha da Figura 5.1 acima, supondo que o *pixel* (x_i, y_i) já foi desenhado. Sabemos que o próximo *pixel* a ser escolhido deve ser o *pixel* da direita P_E (x_{i+1}, y_i) ou o *pixel* acima deste P_{NE} (x_{i+1}, y_{i+1}). Portanto, a escolha do próximo pixel pode ser resumida à incrementar ou não o valor de y no próximo valor de x.

Vamos considerar um ponto médio M entre P_E e P_{NE}. Vamos considerar também um ponto Q que é a interseção da linha ideal com a reta vertical que passa em x_{i+1}, conforme a figura abaixo.

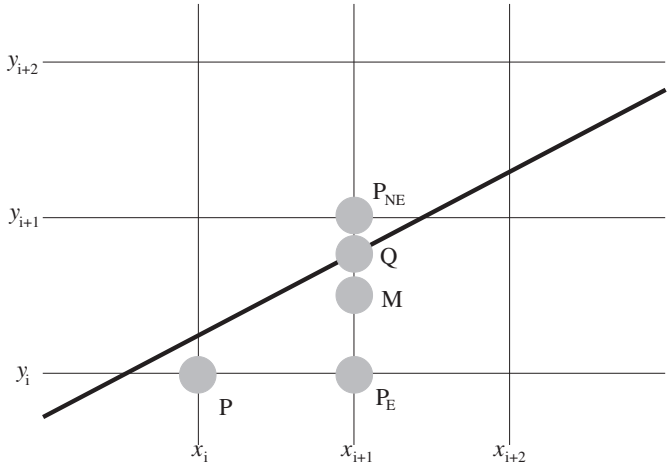


Figura 5.2: Pontos envolvidos na escolha do próximo *pixel*, pelo algoritmo do ponto médio para segmentos de reta.

Podemos observar que se o ponto M estiver abaixo da reta, então o ponto Q está mais próximo de P_{NE} do que P_E, portanto o valor de y deve ser incrementado. Por outro lado, se M estiver acima da reta, o valor de y não deve ser incrementado.

Sabemos que a reta em questão pode ser descrita pela equação $y = ax + B$, onde a é a inclinação da reta que pode ser descrita como dy / dx para um intervalo qualquer da reta. Podemos então descrever a reta pela equação:

$$F(x, y) = dy \cdot x - dx \cdot y + b \cdot dx = 0.$$

Sabemos também que esta equação classifica os pontos do espaço 2D em três categorias: acima da reta, sobre a reta, abaixo da reta. Aplicando valores de x e y nesta equação, o resultado será 0 se o ponto (x, y) estiver sobre a reta, maior que 0 se o ponto estiver abaixo e menor que 0 se estiver acima conforme mostrado na figura 5.3.

Vamos então calcular o valor de $F(x_{i+1}, y_i+½)$ e testar seu sinal para decidir entre escolher o ponto P_{NE} e P_E. Criaremos uma variável de decisão d tal que $d = F(x_{i+1}, y_i+½)$. Pela equação implícita da reta $F(x, y) = ax + by + c = 0$, temos que $d = a(x_i+ 1) + b(y_i + ½) + c$. Onde $a = dy$, $b = -dx$ e $c = B \cdot dx$, conforme vimos anteriormente.

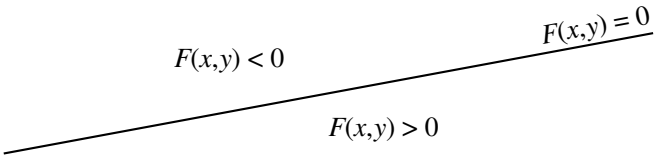


Figura 5.3: Classificação proporcionada pela equação implícita da reta.

Depois de calcular o valor de d e analisar seu sinal, repetiremos o processo para o próximo valor de x. Neste caso, se o último *pixel* escolhido foi P_E, sabemos que $d_{novo} = F(x_i + 2, y_i + ½)$, mas se o último *pixel* escolhido foi P_{NE}, $d_{novo} = F(x_i + 2, y_i + {}^3/_2)$. Se chamarmos o último valor encontrado para d de d_{velho}, podemos jogar seu valor nessas duas fórmulas de d_{novo}, encontrando $d_{novo} = d_{velho} + a$ (onde $a = dy$) para P_E e $d_{novo} = d_{velho} + a + b$ (onde $a + b = dy - dx$) para P_{NE}.

Resumindo, dado um valor de d, o próximo valor de d pode ser encontrado somando dy ao valor antigo quando o último ponto escolhido foi P_E, ou somando $dy - dx$ em caso contrário. O valor inicial de d será $F(x_0 + 1, y_0 + \frac{1}{2}) = dy - \frac{dx}{2}$ onde (x_0, y_0) é o ponto inicial da reta.

Observe que o B (o valor de y para x igual a 0) sumiu. Ele envolveria cálculos como $y_0 - x_0\frac{dy}{dx}$. Detalhando as contas que levam à fórmula anterior, temos:

$$\begin{aligned} F(x_0 + 1, y_0 + \frac{1}{2}) &= a(x_0 + 1) + b(y_0 + \frac{1}{2}) + c \\ &= ax_0 + a + by_0 + \frac{b}{2} + c \\ &= ax_0 + by_0 + c + a + \frac{b}{2} \end{aligned}$$

entretanto,

$$ax_0 + by_0 + c = F(x_0, y_0) = 0$$

pois o ponto (x_0, y_0) está sobre a reta a ser desenhada, logo,

$$\begin{aligned} F(x_0 + 1, y_0 + \frac{1}{2}) &= a + \frac{b}{2} \\ &= dy - \frac{dx}{2} \end{aligned}$$

Para eliminar a fração do valor inicial de d, vamos redefinir $F(x, y)$, multiplicando a antiga função por 2. Desta forma o valor da função em um ponto qualquer (x, y), terá o mesmo sinal que na função antiga, e o sinal é a única coisa que nos interessa para fazer a escolha. Com a multiplicação os valores ficam:

$$d_{inicial} = 2dy - dx, \qquad \Delta_E = 2dy, \qquad \Delta_{NE} = 2(dy - dx)$$

Com esses novos valores, não é necessário o uso de variáveis do tipo ponto flutuante. O algoritmo para desenhar segmentos do segunto octante fica conforme o algoritmo 5.1.

Os segmentos de reta em outros octantes podem ser desenhados de maneira análoga. Procure na Internet algumas implementações desse algoritmo. Você verá que existem muitas variações, especialmente no diz respeito a como lidar com a simetria. Antes da implementação deste algoritmo é bom lembrar que o sistema de coordenadas do dispositivo de apresentação pode ser diferente do usado nas ilustrações deste texto e portanto os vários octantes podem estar em posições diferentes.

Algoritmo 5.1 Algoritmo do Ponto Médio para segmentos de reta.

Desenhar o segmento de $P_0 = (x_0,\ y_0)$ a $P_1 = (x_1,\ y_1)$ no primeiro octante:
1. d ← 2*dy - dx
2. Δ_E ← 2*dy
3. Δ_{NE} ← 2 * (dy-dx)
4. x ← x0
5. y ← y0
6. Desenhar o pixel (x,y).
7. Enquanto x < x1:
7.1. Se d ≤ 0:
7.1.1. Incrementar d de Δ_E.
senão:
7.1.2. Incrementar d de Δ_{NE}.
7.1.3. Incrementar y.
7.2. Incrementar x.
7.3. Desenhar o pixel (x,y).

5.2 Círculos

Bresenham também desenvolveu um algoritmo incremental para desenhar círculos que trabalha somente com variáveis inteiras. Assim como o algoritmo para segmentos de retas, vamos analisar aqui um algoritmo com pequenas variações do algoritmo original, conhecido como algoritmo do ponto médio. Este algoritmo também usa o teste do ponto médio para decidir qual *pixel* (dentre as escolhas possíveis) está mais perto do círculo ideal.

Qualquer algoritmo para desenhar um círculo num dispositivo matricial pode fazer uso da simetria dessa figura para reduzir o tempo gasto com processamento. A partir dos pontos de um dos octantes do círculo, podemos rapidamente encontrar as coordenadas dos pontos restantes do círculo, conforme pode-se ver na figura 5.4.

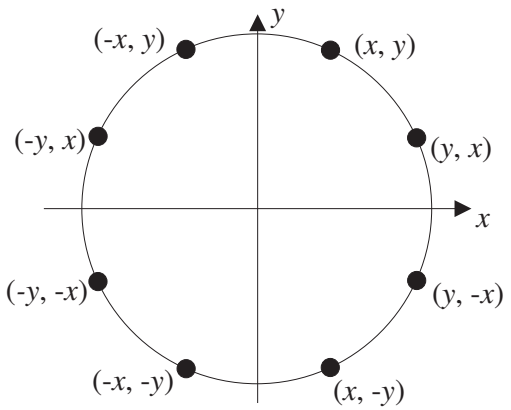


Figura 5.4: Simetria de pontos num círculo

Vamos analisar aqui o problema relativo à geração dos pontos do segundo octante de um círculo. Neste caso, a partir de um último *pixel* desenhado, existem duas alternativas possíveis para o próximo *pixel*, conforme vemos na figura 5.5.

As duas alternativas possíveis para o pixel depois de P são P_E e P_{SE}. Se o ponto médio M estiver

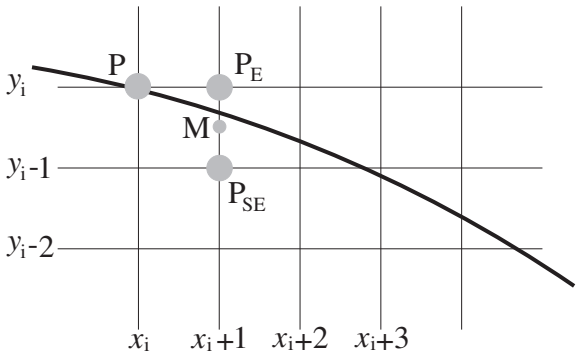


Figura 5.5: Pontos envolvidos na escolha do próximo *pixel*, pelo algoritmo do ponto médio para círculos

abaixo da circunferência, P_E, será o escolhido. Caso contrário, o ponto escolhido será P_{SE}. Considerando que o círculo tem centro na origem, ele pode ser descrito por:

$$F(x, y) = x^2 + y^2 - R^2.$$

Aplicando as coordenadas do ponto M a esta função, o resultado será zero para ponto sobre o círculo, menor que zero para pontos dentro do círculo e maior que zero para pontos fora dele; conforme mostrado na figura 5.6.

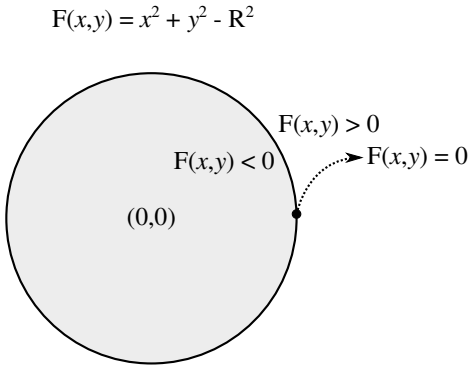


Figura 5.6: Função para determinar a posição de um ponto (x,y) em relação a um círculo de centro na origem e raio R

Devemos então aplicar as coordenadas de M à equação do círculo e escolheremos P_E se o resultado for menor que zero ou P_{SE} em caso contrário. Vamos então criar uma variável de decisão

$$d = F(x_{i+1}, y_{i-½}) = (x_i + 1)^2 + (y_i - ½)^2 - R^2.$$

Caso P_E seja escolhido, o próximo valor da variável de decisão será

$$d_{novo} = F(x_{i+2}, y_{i-½}) = (x_i + 2)^2 + (y_i - ½)^2 - R^2.$$

Neste caso,

$$d_{novo} = d_{velho} + 2x_i + 3,$$

portanto o incremento da variável de decisão será $2x_i$+3. Caso P_{SE} seja escolhido, o próximo valor da variável de decisão será

$$d_{novo} = F(x_{i+2}, y_{i-\frac{3}{2}}) = (x_i + 2)^2 + (y_i - \tfrac{3}{2})^2 - R^2.$$

Nesta situação, $d_{novo} = d_{velho} + 2x_i - 2y_i + 5$, e o incremento será $2x_i - 2y_i + 5$. Observe que, em qualquer um dos dois casos, o valor do incremento da variável de decisão depende das coordenadas do *pixel* P (o último a ser desenhado). Outro detalhe importante é que todas as multiplicações tem um multiplicador igual a 2, o que pode ser implementador de maneira eficiente com uma operação de deslocamento à esquerda (*left-shift*) dos bits da variável.

Resta agora encontrar o valor inicial de d. Supondo que o círculo tem centro na origem e que o raio é um valor inteiro, o primeiro *pixel* a ser escolhido será (0,R). O primeiro ponto médio será (1, R-½), logo, o valor inicial de d será F(1, R-½) = 1 + (R^2 - R + ¼) - R^2 = $^5/_4$ - R. Infelizmente esse valor não é inteiro mas podemos então uma nova variável de decisão d' tal que $d' = d$ - ¼. O valor inicial desta nova variável é então 1 - R e testar se $d < 0$ é a mesma coisa que testar se $d' <$ -¼. Como d' é inicializado com um valor inteiro e é sempre incrementado com um valor inteiro, esse teste pode ser substituído por $d' < 0$. A grosso modo, o que fizemos foi simplesmente ignorar a fração que ficaria sempre sobrando na variável de decisão. Como os incrementos são inteiros, teríamos sempre um número negativo terminando com .75 ou um número positivo terminando com .25. A variavél d' só muda o sinal de 0.25 que fica sendo zero. Uma implementação deve então tratar o zero como sendo positivo.

Este algoritmo pode ser melhorado ainda mais usando diferenças incrementais de segunda ordem. A implementação que desenha círculos com centro fora da origem pode ser feita facilmente. Círculos são um tipo especial de elipse, mas são justamente suas peculiaridades que permitem uma implementação eficiente. Isto pode justificar a implementação de teste inicial (numa função que desenha elipses) para escolher entre o algoritmo do círculo ou da elipse.

5.3 Elipses

Uma elipse pode descrita pela equação F(x, y) = $b^2x^2 + a^2y^2 - a^2b^2 = 0$, onde a é o raio horizontal e b é o raio vertical da elipse. Usaremos novamente o teste do ponto médio para escolher *pixels* ao longo da elipse mas, desta vez, precisamos calcular valores para um quadrante completo. Vamos considerar uma elipse com centro na origem. Da mesma forma que nos casos das outras primitivas, vamos aplicar os valores das coordenadas de M na equação. Se o resultado for menor que zero, teremos M dentro da elipse. Se for maior que zero, M estará fora da elipse.

Para o primeiro quadrante, considerando o ponto inicial P na posição (0, b), teremos uma escolha possível entre P_E e P_{SE}, até um determinado ponto onde a escolha passa a ser entre P_{SE} e P_S, conforme a figura abaixo.

O ponto de separação entre essas duas regiões é aquele onde a tangente tem inclinação -1, e o vetor gradiente (ortogonal à tangente) tem componente x igual à componente y. É possível saber que estamos passando da região 1 para a região 2 no momento em que $a^2(y_i - ½) \leq b^2(x_i + 1)$.

Dentro da região 1, teremos uma situação semelhante à vista no algoritmo do círculo. O próximo M depois de P será (x_i + 1, y_i - ½), o que nos dá um valor para a variável de decisão assim: $d_{velho} = b^2(x_i+1)^2 + a^2(y_i-½)^2 - a^2b^2$. Se P_E for o próximo *pixel* escolhido, então $d_{novo} = b^2(x_i+2)^2 + a^2(y_i-½)^2 - a^2b^2$, portanto o incremento de d fica sendo $b^2(2x_i+3)$. Por outro lado, se P_{SE} for o *pixel* escolhido, então $d_{novo} = b^2(x_i+2)^2 + a^2(y_i-^3/_2)^2 - a^2b^2$, o que torna o incremento de d $b^2(2x_i+3) + a^2(-2y_i+2)$.

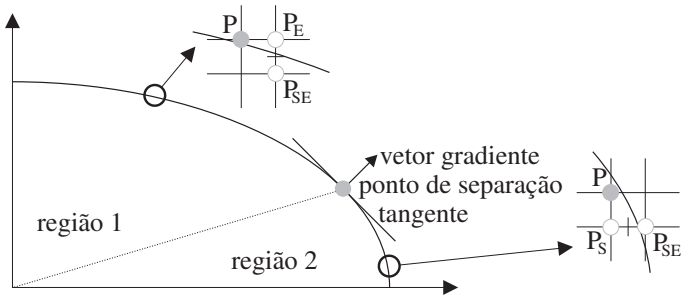


Figura 5.7: Pontos envolvidos na escolha do próximo *pixel*, pelo algoritmo do ponto médio para elipses em suas duas regiões distintas.

Na região 2, as coordenadas de M serão (x_i+½, y_i-1), então o valor da variável de decisão será: $d_{velho} = b^2(x_i+½)^2 + a^2(y_i-1)^2 - a^2b^2$. Se P_{SE} for o próximo pixel escolhido, então $d_{novo} = b^2(x_i+{}^3/_2)^2 + a^2(y_i-2)^2 - a^2b^2$, portanto o incremento de d fica sendo $b^2(x_i+2) + a^2(3-2y_i)$. Por outro lado, se P_S for o *pixel* escolhido, então $d_{novo} = b^2(x_i+½)^2 + a^2(y_i-2)^2 - a^2b^2$, e o incremento de d fica sendo $a^2(3-2y_i)$. O valor inicial de d será $F(1, b-½) = b^2 + a^2(b-½)^2 - a^2b^2 = b^2 + a^2(-b+¼)$. Depois de calculado o valor inicial de d, a parte fracionária do número pode ser desprezada, já que todos os incrementos são inteiros (mais ou menos como fizemos com o círculo).

A cada iteração do *loop*, o algoritmo deve não só testar o sinal de d, como também deve verificar a passagem da região 1 para a região 2. Neste momento, o valor de d, deve passar a ser $F(x_i+½, y_i-1)$. Assim como no caso do círculo, podemos eliminar as multiplicações do cálculo do incremento, calculando incrementos de segunda ordem. O cálculo das componentes do vetor gradiente (o teste que indica passagem da região 1 para a região 2) também pode ser feito de maneira incremental. Para desenhar elipses cujo centro esta fora da origem, usamos o mesmo procedimento do círculo.

5.4 Triângulos

A maioria dos textos de computação gráfica trata da rasterização de polígonos em geral. Um algoritmo que desenha polígonos quaisquer pode desenhar triângulos. Porém, sabendo que estamos lidando com um caso especial de polígono, várias coisas podem ser simplificadas e por isso um metódo específico para triângulos será abordado.

Para encontrar os *pixels* de um triângulo, é preciso mais do que ser capaz de desenhar segmentos de reta. Primitivas geométricas com preenchimento são tradicionalmente desenhadas linha por linha. Monitores de tubo de raios catódicos (CRT) desenham pixels linha por linha e também elementos consecutivos numa linha de matriz costumam ser consecutivos na memória, tornando possível um acesso otimizado. Algoritmos de rasterização que processam linha por linha costumam conter o termo “scan line” no nome em inglês.

Um problema importante na rasterização de primitivas preenchidas é a questão dos *pixels* das extremidades. Se duas primitivas se tocam nas extremidades (vértices ou arestas), não existe uma forma perfeita de associar os *pixels* das extremidades com as primitivas. É desejável que cada *pixel* seja desenhado uma única vez e que não existam *pixels* não associados a nenhuma primitiva. Uma forma comum para lidar com esse problema é desenhar pixels num intervalo que é fechado numa direção e aberto na outra. Conforme pode ser visto na figura 5.8.

Essa solução não resolve todos os casos do problema de um vértice compartilhado entre vários polígonos, mas já funciona como um aviso para o fato de que nem todas as linhas discretas de um polígono

preenchido precisam ser realmente desenhadas. O problema do *pixel* compartilhado não será abordado em mais detalhe neste texto. Um exemplo de *pixel* compartilhado entre vértices pode ser visto na figura 5.9.

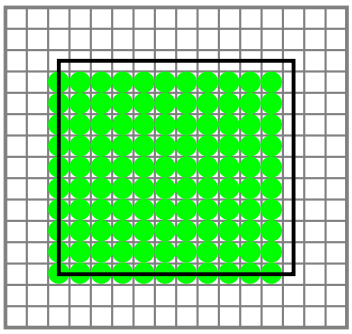


Figura 5.8: Retângulo rasterizado em intervalo aberto à direita e acima.

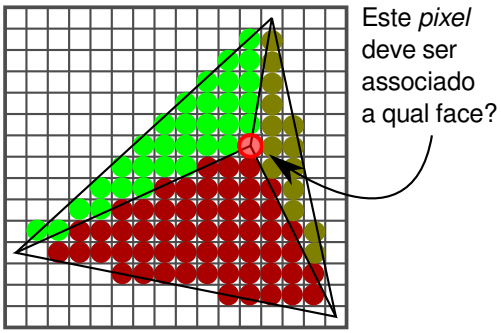


Figura 5.9: *Pixel* compartilhado entre vértices.

Para escolher os *pixels* de um triângulo, vamos primeiro computar valores de x nas extremidades e depois desenhar todos os valores de x para o mesmo y do triângulo, depois incrementamos o valore de y e repetimos o processo.

Para garantir uma escolha de valores limítrofe sem muitos testes, vamos classificar cada aresta de um triângulo como "aresta longa", "aresta 2" ou "aresta 3". A aresta longa será a que cobre o maior intervalo em y. Em caso de empate, qualquer uma das arestas pode ser classificada como a aresta longa. Exemplos de classificação de arestas podem ser vistos na figura 5.10.

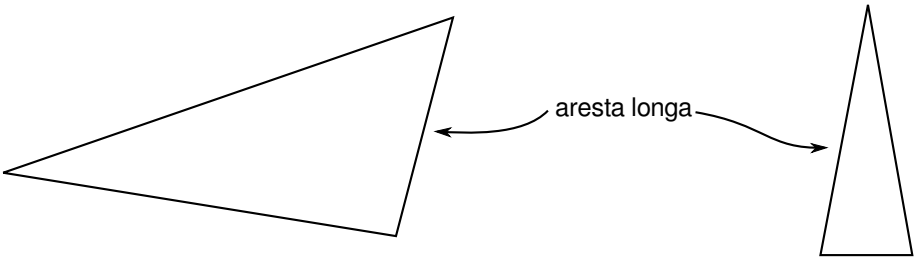


Figura 5.10: Classificação de arestas num triângulo.

Com classificação, sabemos que o triângulo pode ser desenhado entre os limites da aresta 2 e da aresta longa. Depois entre os limites da aresta 3 e da aresta longa. O cálculo dos limites esquerdo e direito em cada y deve preferencialmente ser feito apenas com aritmética inteira, de maneira semelhamente à feita na rasterização de segmentos de reta. Assim, o algoritmo para desenhar um triângulo poderia

ser semelhante ao apresentado no algoritmo 5.2 [2]. No algoritmo, os incrementos para *xdir* e *xesq* estão entre aspas porque podem ser um ou dois números dependendo de como o controle é feito.

Algoritmo 5.2 Rasterização de um triângulo.

1. Calcular ymin, ymax, xesq, xdir, "incremento xesq" e "incremento xdir" usando a aresta2 e a aresta longa.
2. Inicializar y com ymin.
3. Enquanto y for menor que ymax, faça:
3.1. Desenhar todos os pixels entre xesq e xdir.
3.2. Atualizar o valor de xesq e xdir.
3.3. Incrementar o y.
4. Calcular ymin, ymax, xesq, xdir, "incremento xesq" e "incremento xdir" usando a aresta3 e a aresta longa.
5. Inicializar y com ymin.
6. Enquanto y for menor que ymax, faça:
6.1. Desenhar todos os pixels entre xesq e xdir.
6.2. Atualizar o valor de xesq e xdir.
6.3. Incrementar o y.

Capítulo 6

Determinação de Superfícies Visíveis

No processo de apresentação de imagens 3D, podemos alcançar uma boa representação de profundidade eliminando da apresentação as linhas ou superfícies que estão escondidas por outras linhas ou superfícies da imagem.

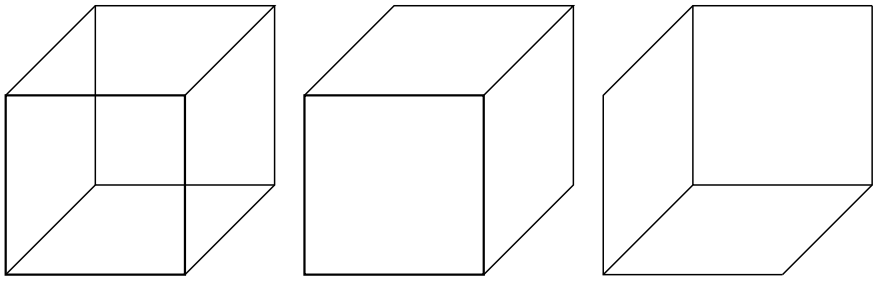


Figura 6.1: A projeção das seis faces de um cubo (a), que poderia ser interpretada como a geometria em (b) ou em (c).

Existem dois tipos principais de algoritmos: a) aqueles que tratam os objetos através de sua representação, modificando-os de forma que eles possam ser apresentados sem problemas; b) aqueles que tratam os objetos em sua apresentação, desenhando os pixels de maneira que somente sejam visíveis aqueles que estejam mais próximos do observador.

6.1 Remoção de faces invisíveis num sólido (*back-face culling*)

Quando um objeto representado é um sólido, normalmente desejamos apresentá-lo para um observador externo, ou seja, não nos interessa a visão de dentro do sólido. Considerando que este sólido é representado por um conjunto de faces planas, cujo vetor normal aponta para fora do sólido, podemos eliminar do processo de projeção e apresentação todas as faces cuja normal forma um ângulo maior do que 90^o com a direção de projeção. Este algoritmo é conhecido como *back-face culling*.

No exemplo acima, somente as faces em negrito precisam ser consideradas na apresentação do sólido, pois as outras encontram-se "de costas" para o observador. As faces que restam ainda precisam ser processadas por algum outro método.

6.2 Algoritmo do pintor

Uma ideia geral para determinação de superfícies visíveis é ordenar todos os polígos da cena em relação à distância entre eles e o observador, depois desenhar os polígonos em ordem, do mais distante do observador até o mais próximo.

Isso fará com que partes da imagem sejam desenhadas mais de uma vez, nos lugares em que mais de um polígono aparecer na imagem, o que não é um problema se desenho for num dispositivo virtual (ex.: *frame buffer*).

O nome "algoritmo do pintor" é uma referência à estratégia usada por pintores há séculos, que consiste em pintar inicialmente os elementos mais distantes e, depois da tinta secar, pintar por cima elementos mais próximos do observador. A figura 6.2 ilustra essa estratégia.

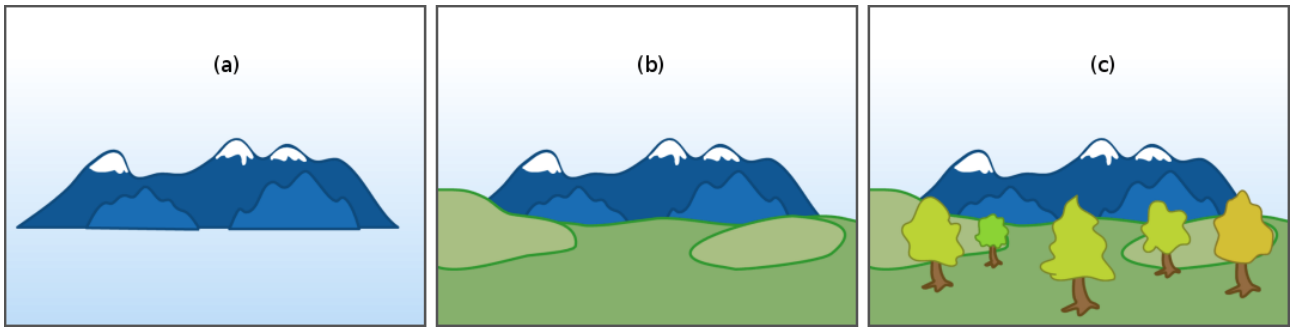


Figura 6.2: Pintura do mais distante para o mais próximo. (a) o céu e montanhas no horizonte foram pintados (b) morros e grama foram pintados depois (c) arvores foram pintadas por último - imagem retirada de https://pt.wikipedia.org/wiki/Ficheiro:Painter's_algorithm.png

O grande problema em desenhar polígonos do mais distante ao mais próximo do observador é que tal distância varia para cada ponto do polígono. Não é portanto possível estabelecer essa ordem em qualquer situação, como pode ser visto na figura 6.3. O "algoritmo do pintor" não é, portanto, um algoritmo propriamente dito, é mais uma ideia geral. Para realmente resolver o problema da determinação de superfícies visíveis usando essa ideia, algoritmos costumam processar a representação dos polígonos, produzindo novos polígonos que podem efetivamente ser ordenados e então desenhados de traz para frente, como em [7].

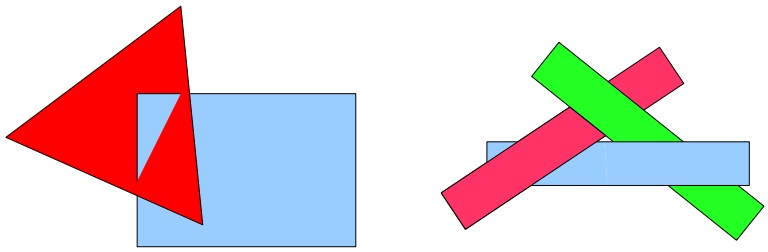


Figura 6.3: Situações em que uma ordem dos polígonos não pode ser determinada.

6.3 Algoritmo do "z-buffer"

Também conhecido por "*depth-buffer*", este algoritmo foi desenvolvido por Ed Catmull, em seu doutorado publicado no ano de 1974. Este é um algoritmo bastante difundido e frequentemente implementado em *hardware* em placas gráficas. Ele requer, além da tabela que indica a cor de cada pixel

(que pode ser a memória de vídeo) uma nova tabela (ou *buffer*) que indica a profundidade de cada pixel da imagem. Esta nova tabela é chamada de *z-buffer* porque normalmente a projeção de imagem é feita em relação ao eixo z, ou seja, são os valores de z que indicam profundidade na imagem.

A ideia geral é desenhar somente os pixels que estejam na frente (do ponto de vista da câmera) de pixels previamente desenhados. O algoritmo 6.1fornece mais detalhes.

Algoritmo 6.1 z-buffer

1. Todas as posições da "memória de vídeo" são incializadas com a cor de fundo.
2. Todas as posições do *z-buffer* são inicializados com 0 ou outro valor indicando a maior distância possível até o observador.
3. Para cada polígono:
3.1 Para cada pixel na projeção do polígono:
3.1.1 Calcular a profundidade do pixel.
3.1.2 Se o pixel não está mais afastado que o anterior (valor atual no *z-buffer*):
3.1.2.1 Atualizar o *z-buffer.*
3.1.2.2 Atualizar a memória de vídeo.

Para calcular a profundidade do pixel (x,y) em um polígono, podemos usar a fórmula $Ax+By+Cz+D = 0$, de onde temos que:

$$z = \frac{-D - Ax - By}{C}$$

Entretanto, esta conta requer muito tempo de processamento, portanto vamos usar a técnica do cálculo incremental. Considerando que sabemos o valor de z_1 para o ponto (x_1, y_1) e queremos calcular o valor de z no próximo x, ou seja, z_2 para o ponto (x_2,y_2) onde $x_2= x_1+1$ e $y_2= y_1$, temos:

$$z_2 = \frac{-D - Ax_2 - By_2}{C} \Rightarrow z_2 = \frac{-D - A\,(x_1 + 1) - By_1}{C} \Rightarrow$$

$$z_2 = \frac{-D - Ax_1 - A - By_1}{C} \Rightarrow z_2 = \frac{-D - Ax_1 - By_1}{C} - \frac{A}{C} \Rightarrow$$

$$z_2 = z_1 - \frac{A}{C}$$

Como o valor de -(A/C) é constante, uma única subtração será necessária para encontrar o próximo valor de z no eixo x. De maneira análoga, o próximo valor de z no eixo y é dado por:

$$z_2 = z_1 - \frac{B}{C}$$

A seção 1.7.8 mostra como encontrar os valores de A, B e C a partir de três pontos, que é a informação disponível na estrutura de dados que representa o sólido cuja face está sendo desenhada. Isso nos proporciona elementos suficientes para uma implementação aparentemente correta do algoritmo, ou ainda, que funciona em situações particulares, mas não temos uma analise suficiente para uma implementação genérica e correta de z-buffer ainda. A projeção perspectiva dificulta a implementação do z-buffer pois 1) destrói a coordenada z dos vértices e 2) destrói a linearidade da variação de z em relação às coordenadas do dispositivo de apresentação (x', y'). Esses obstáculos serão vencidos na seção 9.2. Podemos adiantar que:

1. A profundidade dos vértices pode ser recuperada antes da normalização das coordenadas dos vértices, ou seja, antes da divisão por w.

2. A profundidade de cada pixel pode ser calculada pela interpolação linear de $1/z$ ao invés da interpolação de z.

Uma das maiores vantagens deste algoritmo é que ele não funciona apenas com polígonos planos, ele pode ser usado para determinar a superfície visível de qualquer objeto onde o valor de z e a cor de cada pixel pode ser determinada em função de x e y. Além disso, é um algoritmo rápido e paralelizável em relação às faces desenhadas.

Entre as desvantagens deste algoritmo, podemos citar a grande quantidade extra de memória necessária (pouco importante nos dias de hoje) e também a precisão finita do *z-buffer*, que faz com a ordem de valores muito próximos possa ser computada incorretamento, produzindo um problema muito comum geralmente chamado de *z-fighting*. O *z-fighting* pode ser mitigado usando tipos de dados com muitos bits e também por técnicas diversas mas não pode ser completamente eliminado.

O processo de análise de cada *pixel* da imagem também pode ser estendido para determinar a cor de cada pixel, baseado em sua posição. O *z-buffer* costuma ser implementado de forma integrada aos algoritmos de iluminação.

Capítulo 7

Iluminação

Como mencionado anteriormente a luz é um elemento fundamental na criação de imagens. Sem a luz, saber as propriedades da câmera e do objeto não teria valor, pois a imagem seria sempre preta, sem qualquer informação. Os processos físicos que definem a interação entre luz e objetos são bastante complexos e precisam de um certo grau de simplificação para serem implementados. Sem dúvida, aquele que parece mais preciso e produz os melhores resultados são os algoritmos baseados em *ray-tracing*. Entretanto, é possível implementar modelos bem mais simples, que produzam resultados satisfatórios para a maioria das aplicações de computação gráfica. Antes de apresentá-los vamos apresentar alguns conceitos básicos sobre intensidade de luz e cor.

7.1 Intensidade de Luz

A intensidade de luz é o principal fator a ser considerado na síntese de imagens. Um possível modelo simplificado de iluminação para geração de imagens poderia considerar que algumas parte do objeto cuja imagem será gerada refletem (ou emitem) mais luz que outras partes. Essa variação definirá no cérebro do observador a noção de formas que permitirão identificar o aspecto físico (aparência) do objeto.

As partes que refletem mais luz do objeto serão percebidas como mais claras (mais brancas) e as que refletem menos luz serão percebidas como mais escuras (mais pretas). Neste caso temos o que chamamos de imagem em tons de cinza. É o que acontece quando usamos filme preto e branco, onde existe uma única camada de elementos químicos, sensível à quantidade de luz que lhe incide.

7.2 Cor

Nosso sistema visual, no entanto, reage à intensidade de luz de forma mais complexa. A luz é uma forma de radiação eletromagnética (assim como as ondas de rádio ou microondas, só que numa frequência bem mais alta) cujo comprimento de onda pode variar, aproximadamente, de 350 (violeta) a 780nm (vermelho). Fora desta faixa, a radiação não consegue sensibilizar nossos olhos. Na retina do olho humano existem dois tipos de células nervosas foto-sensíveis: os cones e os bastonetes. Existem três tipos de cones, cada tipo é especificamente sensível a luz, numa determinada faixa de comprimento de onda. Um tipo é mais sensível à luz por volta de 440nm (entre o azul e o violeta), outro tipo é mais sensível à por volta de 545nm (verde) e o terceiro é mais sensível à luz por volta de 580nm (entre amarelo e vermelho — mais próximo do amarelo que do vermelho). Essa diferença

entre os tipos existentes de cones é que nos possibilita enxergar uma grande variedades de cores, permitindo que nosso cérebro consiga extrair muito mais informação dos estímulos da visão do que o cérebro de um gato, que não pode distinguir as cores.

Sabemos que podemos conseguir qualquer cor através da mistura do vermelho, verde e azul, isso porque os receptores em nossos olhos são sensíveis à essas cores (aproximadamente falando), ou seja, nós percebemos as cores como combinações dessas três frequências. Por este motivo, na computação, vamos representar uma cor pela quantidade de vermelho, verde e azul que a compõe. Um sistema digital de identificação de cores deve usar 24 bits ou menos para identificar cada cor (8 bits para o vermelho, 8 para o verde e 8 para o azul) isso porque esses 24 bits nos proporcionam 16777216 cores diferentes o que já é um pouco mais do que o cérebro humano é capaz de distinguir. Alguns sistemas usam 32 bits para identificar um pixel na tela, neste caso, os 8 bits restantes possivelmente servem para identificar transparência, possibilitando que, ao ser desenhado, a cor de um pixel na tela dependa a cor que estava previamente naquela posição.

Você pode estar se perguntando porque a sua professora dizia que as cores primárias eram o vermelho, o amarelo e o azul, quando você era criança. Agora o amarelo virou verde. Por quê?

Existem dois processos de combinação de cores: o processo aditivo e o processo subtrativo. Podemos enxergar o processo aditivo como uma operação de união, sendo o que acontece ao misturarmos luz. É o processo que acontece se usarmos holofotes coloridos projetando luz sobre uma superfície perfeitamente branca, é também o caso em que colocamos pequenas fontes de luz coloridas tão próximas uma da outra que nosso olho não seja capaz de distinguir uma da outra (como no caso do monitor de vídeo). O processo subtrativo, por outro lado, é o processo inverso. É a interseção de cores. É o que acontece quando misturamos tintas.

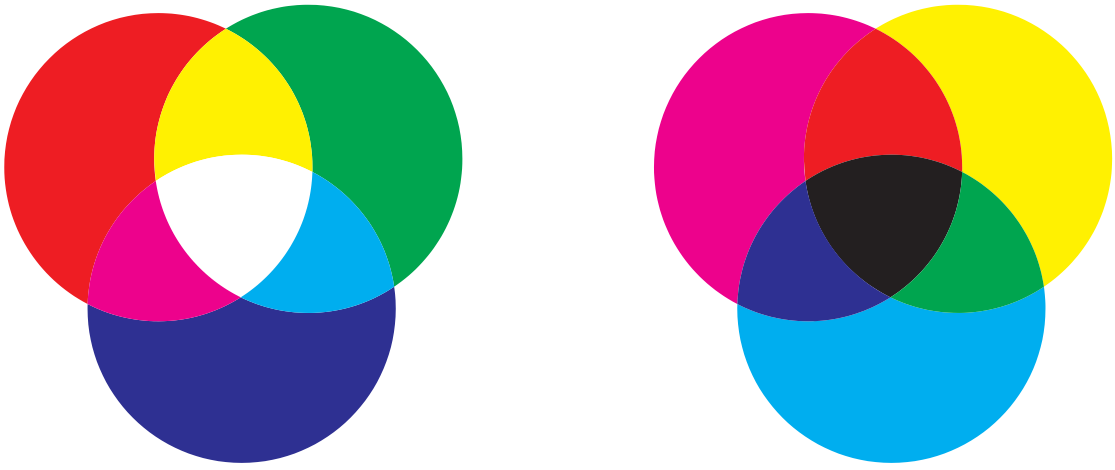
Uma folha de uma árvore absorve um pouco da radiação luminosa do Sol e reflete a parcela restante. Entretanto, a radiação luminosa refletida fica normalmente concentrada na frequência que identificamos por verde, o que faz com que a folha pareça verde para nós. Uma folha de papel branco reflete quase toda radiação luminosa que recebe, sendo que as várias frequências da luz são refletidas mais ou menos de maneira uniforme. Ao aplicarmos tinta verde no papel, estamos aplicando pigmentos químicos que refletem luz cuja frequência esta concentrada na região da cor verde. Colocando mais tinta sobre a mesma região do papel, estamos acrescentando mais pigmentos, alguns vão absorver certas frequências de luz e outros vão absorver outras frequências de luz. Isso faz com que esta região do papel reflita menos luz do antes (somente as frequências que as duas tintas refletem.

Assim, as cores primárias do processo subtrativo são outras:

- ciano (uma cor meio azulada, que é a ausência de vermelho);
- amarelo (que é a ausência de azul);
- magenta (uma cor meio violeta, que é a ausência de verde).

Se os pigmentos presentes nas tintas fossem perfeitos, a mistura dessas três cores absorveria as 3 cores primárias, o que resultaria na cor preta. Como isso não acontece na prática (consegue-se obter um marrom bem escuro) usa-se tinta preta para auxiliar no processo de combinação de cores pelo processo subtrativo.

As crianças aprendem o processo subtrativo de combinação de cores porque trabalham com tinta. Para simplificar o processo, o magenta é substituído pelo vermelho e o ciano é substituído pelo azul, o que gera resultados parecidos, devido às imperfeições dos pigmentos das tintas, livrando professores da tarefa de explicar o que é a cor magenta e o ciano para as crianças.



(a) (b)

Figura 7.1: (a) processo aditivo de composição de cores e (b) processo subtrativo.

Para criar um modelo de representação de cores, que é uma forma de representar uma grande quantidade de cores a partir de elementos mais simples, é interessante considerar qual sistema de combinação de cores será usado. O modelo RGB (Red, Green, Blue) é muito comum e bem apropriado para representar cores usando o processo aditivo (como num monitor LCD). O modelo CMYK (Cian, Magenta, Yellow, blacK) também é muito comum e bem apropriado para representar cores usando o processo subtrativo (como numa impressora jato de tinta). Existem vários outros modelos para descrever cores. Alguns como HSV (Hue, Saturation, Value) foram criados para serem mais perceptivos (se aproxima da forma como mentalizamos a combinação de cores) e facilitar uma escolha interativa de cor. Alguns modelos com YUV consideram a capacidade humana de percepção de diferenças em cores, favorecendo a compactação de dados e minimizando problemas perceptivos em caso de erro de transmissão. Modelos desse tipo são comuns em padrões de TV, aparelhos audivisuais e formatos de arquivo de vídeo. Existem ainda sistemas mais baseados em física como o CIE 1931 XYZ que são usado para formalizar as cores possíveis e calibrar disposivos de apresentação.

Apesar dos muitos modelos de representação de cores, a conversão de um para outro é teoricamente sempre possível. Na prática os espaços de representação podem mudar, o intervalo entre cores representáveis pode ser diferente e os algoritmos para transformação podem ser baseados em observação, sendo diferentes em programas diferentes. Na prática é comum haver perda de informação na conversão entre modelos de cores.

7.3 Modelos de Iluminação

Na tentativa de sintetizar imagens realistas, precisaremos de fórmulas que representem a quantidade de luz associada a um ponto qualquer, numa face do objeto a respeito do qual queremos gerar uma imagem. Essas fórmulas devem se referir a variáveis que representam características das fontes de luz e das propriedades do material do objeto modelado.

7.3.1 Luz Ambiente

A luz que incide num objeto e em seguida segue para a câmera pode ter sua origem numa fonte de luz ou num outro objeto que recebeu luz. Um raio de luz pode sofrer inúmeras reflexões ou refrações,

interagindo com vários objetos até que consiga chegar à câmera. Podemos simplificar a imitação desse processo considerando que cada ambiente possui uma pequena quantidade de luz, de origem desconhecida, independente de qualquer fonte de luz, que ilumina todas as faces um objeto de maneira uniforme.

Cada objeto na cena pode refletir mais ou menos dessa luz ambiente, de acordo com as propriedades do material que o compõe. Dessa forma, podemos dizer que a intensidade luminosa de um ponto arbitrário num objeto é dada por:

$$I = I_a k_a$$

Onde I é a intensidade da luz no ponto analisado; I_a é a intensidade da luz ambiente na cena e k_a é o coeficiente de reflexão da luz ambiente, ou seja, a quantidade de luz ambiente que o objeto reflete.

O coeficiente k_a é uma característica do objeto, um número que pode variar entre 0 e 1, indicando quanto da luz ambiente ele é capaz de refletir.

7.3.2 Reflexão Difusa

A luz ambiente ilumina objetos de maneira uniforme, o que proporciona uma imitação insuficientemente precisa do que acontece no mundo real. Um modelo um pouco mais preciso tenta imitar o que acontece com superfícies toscas ou ásperas, como o carvão, por exemplo. Superfícies assim parecem igualmente brilhantes de qualquer ponto de vista, pois a superfície é tão irregular que não existe uma direção para onde os raios de luz são mais frequentemente refletidos. Entretanto, as faces onde os raios de luz incidem de maneira mais perpendicular parecem mais brilhantes que aquelas nas quais a luz incide de maneira mais inclinada, isso porque se a luz incide de maneira mais perpendicular, ela atinge uma área menor.

Imagine um feixe de luz incidindo sobre duas superfícies em ângulos diferentes (Figura 22). A quantidade de energia luminosa que atinge as duas superfícies é a mesma. Entretanto, a área que a luz atinge é maior na superfície mais inclinada. A grosso modo, podemos dizer que uma face de um objeto será brilhante quando a sua normal formar um ângulo pequeno em relação à direção da fonte de luz. Neste modelo, não interessa a posição do observador.

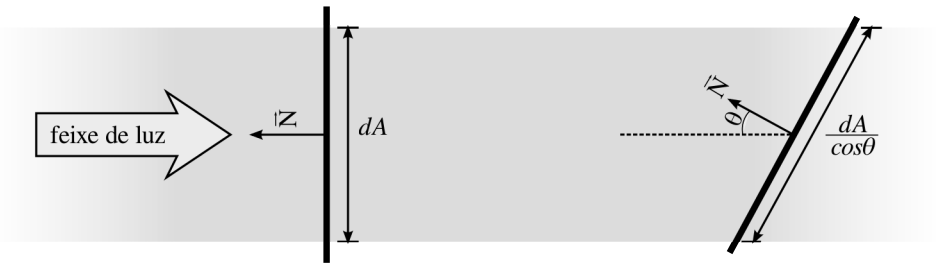


Figura 7.2: Um feixe de luz atinge uma superfície maior quando ela é inclinada.

Dessa forma, podemos dizer que a intensidade luminosa de um ponto arbitrário num objeto é dada por:

$$I = I_d k_d cos\theta$$

Onde I é a intensidade da luz no ponto analisado; I_d é a intensidade da luz difusa na cena, k_d é o coeficiente de reflexão da luz difusa, ou seja, a quantidade de luz difusa que o objeto reflete e θ é o ângulo entre a normal da superfície e a direção da luz.

Juntando a iluminação difusa com a luz ambiente, temos um modelo mais preciso do que usando a iluminação difusa somente. Neste caso a luz ambiente representa aqueles raios de luz que passaram por várias refrações e reflexões cujos resultados exigiriam muito tempo de computação. Neste caso, nossa fórmula de iluminação seria:

$$I = I_d k_d cos\theta + I_a k_a$$

7.3.3 Reflexão Especular

Para um objeto cuja superfície é polida, os raios de luz tendem a ser refletidos de maneira previsível. Uma superfície com polimento perfeito é um espelho. Neste caso o ângulo entre um raio de luz que incide sobre a superfície e a normal é igual ao ângulo entre a normal e o raio de luz refletido.

Na realidade, nenhuma superfície é um espelho perfeito, mas a luz refletida num objeto polido fica mais concentrada numa direção simétrica (em relação à normal) à direção de incidência (ver Figura 7.3). Quanto mais polida for a superfície, mais concentrada estará a luz refletida.

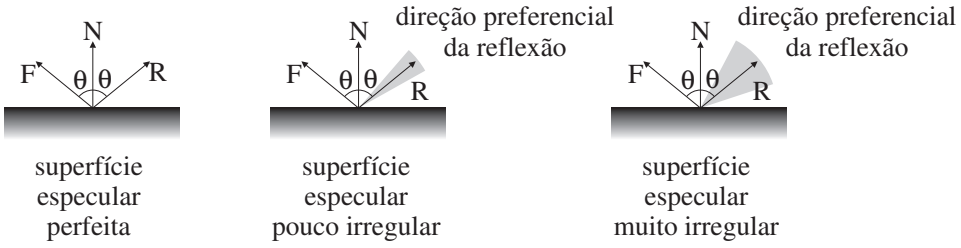


Figura 7.3: Reflexão especular numa superfície.

Por causa da grande quantidade de luz refletida em direções próximas à direção preferencial (R), uma região do objeto olhada numa direção próxima de R, parece muito pouco com o objeto propriamente dito, sendo semelhante à fonte de luz dos raios refletidos. Dizemos que nesta região não vemos o objeto e sim o reflexo de algum outro objeto ou uma fonte de luz.

Desta maneira, um objeto verde, de superfície bastante polida, iluminado por uma luz vermelha, observado da direção preferencial de reflexão será vermelho e será verde quando observado de uma direção muito diferente da direção preferencial de reflexão.

Modelar esse efeito com precisão seria computacionalmente bastante complicado. Entretanto, Phong Bui-Tuong, em 1975, propôs uma aproximação que se tornou bastante popular. Ele considerou que a quantidade de luz refletida é diretamente proporcional ao co-seno do ângulo (α) entre a direção preferida de reflexão (R) e a direção do observador, mas com um índice (n) que determina a importância desse ângulo, chegando a seguinte fórmula:

$$I = I_e k_e cos^n \alpha$$

Onde I é a intensidade da luz no ponto analisado; I_e é a intensidade da luz especular na cena, k_e é o coeficiente de reflexão da luz especular, ou seja, a quantidade de luz especular que o objeto reflete, α é o ângulo entre a direção do observador e direção preferida de reflexão e n é um índice que indica o quanto a superfície é polida. Um valor de n deve ser maior ou igual a 1. Quanto maior o valor de n, mais concentrado será o reflexo observado na região do objeto onde a direção do observador coincide com a direção preferida de reflexão. Valores entre 100 e 500 produzem o efeito de uma superfície metálica polida.

Talvez não faça muito sentido falar em *intensidade de luz especular*, *intensidade de luz difusa* e *intensidade de luz ambiente*, afinal, uma fonte de luz produz raios que são refletidos de maneira que depende quase exclusivamente das propriedades do objeto onde ela incide e sua intensidade é que define a quantidade de luz ambiente, a quantidade de luz refletida de maneira difusa e de maneira especular. Voltaremos a esse assunto na seção 7.3.5.

Juntando os três modelos de reflexão, obtemos um modelo capaz de gerar resultados mais realísticos. A fórmula de iluminação fica:

$$I = I_e k_e cos^n \alpha + I_d k_d cos\theta + I_a k_a$$

7.3.4 Atenuação Atmosférica

Sabemos que os gases e partículas sólidas presentes na atmosfera interagem com raios de luz em seus caminhos de uma fonte de luz até um observador. Ao invés de fazer uma análise detalhada das alterações a que um raio luz está sujeito ao percorrer grandes distâncias na atmosfera, vamos fazer uma análise mais geral e superficial do que ocorre.

Num dia nublado, notamos que objetos distantes tendem a ficar esbranquiçados a medida que se afastam do observador. Ao pôr do Sol, objetos distantes tornam-se mais alaranjados. Durante a noite, objetos distantes tornam-se mais escuros. Em todos os casos, as cores de objetos distantes tendem a ficar uniformes, aproximando-se de uma cor dominante que pode ser vista como a cor do horizonte ou "cor de fundo".

Para representar isso, criamos uma *função de atenuação*, que transforma a cor de um ponto na *cor de fundo*, à medida que a distância entre o observador e o objeto se aproxima do infinito. Utilizando um modelo finito onde os objetos visíveis encontram-se dentre de um *volume visível*, em relação ao qual realizamos *recorte*, podemos definir uma função de atenuação que leve em conta a distância relativa entre o ponto analisado, o plano de frente do volume de visão e o plano de fundo do mesmo.

Funções de atenuação lineares são rápidas de calcular e não requerem a escolha de índices especiais que buscam alcançar um efeito realístico. Entretanto, as funções quadráticas proporcionam efeitos mais realísticos com um pouco mais de processamento. Usando a função de atenuação

$$I = \frac{I'}{a+bd+cd^2}$$

Podemos aproximar a intensidade de luz num ponto (I') para 0, de acordo com a distância (d) do observador até esse ponto. Estamos assim tornando pontos distantes mais escuros, já que o preto é frequentemente usado como *cor de fundo*.

7.3.5 Criando um Modelo Geral

Podemos dizer que a cor de um ponto qualquer é uma função de vários efeitos relacionados com a interação entre a luz e os objetos. Destes efeitos, vimos a reflexão de luz ambiente, de reflexão difusa, reflexão especular e a atenuação atmosférica. A maneira como estes efeitos afetam a imagem de um objeto é chamada de *modelo de iluminação*. Consideramos efeitos distintos porque não seria possível fazer uma simulação física de cada raio de luz. Um modelo de iluminação pode levar em conta mais ou menos efeitos que alteram a maneira como enxergamos um objeto, o custo computacional será maior quanto mais efeitos forem considerados. A fórmula que agrega todos os efeitos também pode ser mais ou menos precisa. É preciso encontrar um equilíbrio entre a qualidade desejada nos resultados e o custo computacional.

Todos os quatro fatores citados anteriormente podem definir um modelo de iluminação, que chamamos de *modelo tradicional de iluminação*, descrito pela fórmula:

$$I = \frac{I_e k_e cos^n \alpha + I_d k_d cos\theta + I_a k_a}{a + bd + cd^2}$$

A intensidade da luz num ponto qualquer do objeto seria calculada de acordo com os fatores comentados anteriormente. Este modelo considera que a *cor de fundo* é o preto já que considera a redução da participação da luz refletida pelo objeto conforme aumenta a distância entre ele e o observaodor, mas não considera o aumento de participação da luz refletida pelo ambiente. Acrescentar elementos para considerar a cor de fundo levaria a outro modelo de iluminação, e fica como exercício para o leitor.

A fórmula do modelo de iluminação seria calculada para cada uma das três cores primárias da luz (considerando o uso do modelo RGB), em cada ponto do objeto, para cada fonte de luz definida. Cada um dos índices da fórmula pode ser calculado em função de características dos objetos e das luzes, mas de maneira geral, as caraterísticas da fórmula são iguais em qualquer implementação, levando em conta esses quatro fatores apresentados. Mesmo quando queremos representar oclusão de fontes de luz (projeção de sombras) e outros efeitos, podemos tomar esse modelo como base, calculando índices e coeficientes iterativamente ao invés de tratá-los como constantes, por exemplo.

7.4 Aplicação de Modelos de Iluminação a Malhas Poligonais

O modelo de iluminação apresentado anteriormente é suficientemente genérico para ser aplicado em qualquer superfície (curva ou facetada), ponto a ponto, proporcionando valores para algoritmos como o *z-buffer*. Entretanto, essa abordagem é bastante lenta. Pretendemos melhorar nosso modelo de iluminação para que ele fique mais rápido (mesmo à custa de perda de qualidade dos resultados).

7.4.1 Iluminação Constante

A alternativa mais simples de todas é aplicar o modelo de iluminação em um único ponto do polígono e utiliza o resultado obtido em todos os outros pontos. Este método é também chamado de "*flat shading*" ou "*constant shading*". Ele não resulta em perda de qualidade se:

- todas as fontes de luz estão no infinito, e portando o ângulo entre a normal do polígono e a direção de cada fonte de luz é constante em todo o polígono,
- o observador está no infinito e
- o polígono é uma representação fiel da superfície do objeto modelo e não uma aproximação.

Quando essas condições não são verdadeiras, temos perda de qualidade, o que não seria um grande problema se cada polígono desenhado fosse bem pequeno.

A real desvantagem desse método, que o torna indesejável mesmo para objetos compostos de polígonos pequenos é o cérebro humano tende a destacar as variações de intensidade de luz. Em outras palavras, uma variação abrupta de intensidade, mesmo que seja pequena, chama nossa atenção para a linha divisória onde a intensidade muda.

Na figura 7.4 podemos notar como o contorno dos retângulos fica destacado em relação à figura toda. Esse efeito é indesejável quando queremos gerar a imagem de uma superfície contínua.

Figura 7.4: A variação abrupta de intensidade é destacada pelo cérebro.

7.4.2 Método de Gouraud

Para resolver o problema da iluminação constante, precisamos fazer distinção entre as cores nos diversos pontos de um polígono. Gouraud propôs, em 1971, que fossem calculadas as cores de cada vértice do polígono e as cores dos outros pontos fossem definidas por interpolação.

Para atingir um efeito mais realístico esse método deve ser usado num modelo geométrico onde as normais são definidas vértice a vértice, ao invés de serem definidas polígono a polígono. O ideal é que a normais sejam definidas durante o processo de criação do modelo geométrico, de maneira analítica. Quando isso não for possível, Gouraud propôs que a normal de um vértice qualquer seja calculada como a média das normais de todos os polígonos que compartilham aquele vértice. Um exemplo seria o cálculo de um vetor normal n, calculado pela média entre quatro outros vetores $n1$, $n2$, $n3$ e $n4$, cujo valor é:

$$n = \frac{n1 + n2 + n3 + n4}{|n1 + n2 + n3 + n4|}$$

A desvantagem deste modelo é que ele não representa bem o caso em que a reflexão especular criaria uma pequena região de brilho no interior do polígono cuja imagem está sendo gerada (ver figura 7.5).

7.4.3 Método de Phong

Para melhorar o efeito obtido pela interpolação, Phong propôs em 1975, que as normais fossem interpoladas no lugar das cores, o que elimina uma série de problemas relacionados à perda de qualidade em termos de reflexão especular. Neste caso, o modelo de iluminação precisa ser aplicado a cada pixel do polígono.

Mesmo usando modelos que não consideram efeitos da reflexão especular, o método Phong produz resultados melhores, porque ele é um meio-termo entre aplicar o modelo de iluminação em cada ponto e o método de Gouraud (note que apenas o vetor normal do polígono varia de pixel para pixel).

Apesar do ganho de qualidade, o método de Phong requer um aumento substancial no processamento necessário para o processo de síntese de imagens, além disso, implementações em *hardware* eram raras até o início dos anos 10, normalmente encontradas somente em equipamentos profissionais e com limitações, por exemplo, em relação à quantidade de fontes de luz que poderiam ser representas[5]. Essas limitações transparecem no sistema tradicional (sem uso de *shaders*) de renderização de APIs como OpenGL em que a limitação do número máximo de luzes costuma ser oito.

Em 1986, Bishop e Weimer propuseram uma aproximação para o método de Phong usando séries de Taylor, conseguindo ganhos expressivos em velocidade.

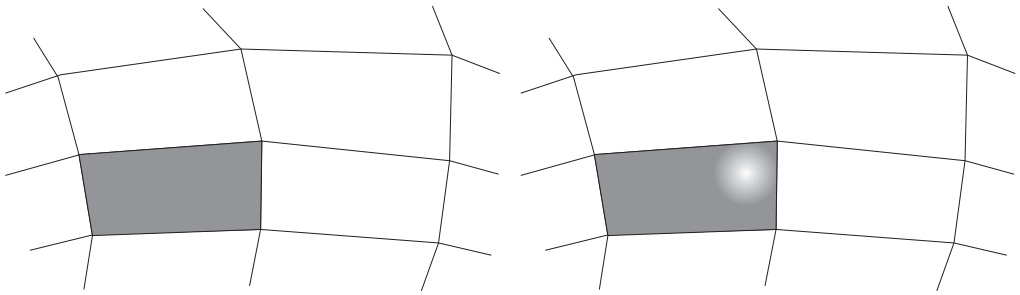


Figura 7.5: Comparação entre o método de Gouraud e o de Phong quando a área de reflexão especular está dentro do polígono.

Capítulo 8

Antialiasing (Técnicas anti-serrilhado)

O resultado da conversão de primitivas geométricas (de natureza contínua) em imagens matriciais (de natureza discreta) é que as linhas (retas ou curvas) podem aparecer como uma escada, ou seja, podemos discernir entre os vários elementos discretos da imagem. Esse efeito é conhecido como *aliasing* e é o resultado da perda de informação na conversão de informações de natureza contínua em informações discretas. Neste caso, a perda de informação é de natureza espacial, mas o termo *aliasing* pode aplicado em transformações de outra natureza (temporal, por exemplo, como no caso de uma animação). Um exemplo clássico do *aliasing* temporal pode ser visto em filmes onde a imagem de uma roda de carro girando dá a impressão de que ela está girando no sentido contrário do movimento. Aliasing espacial também é resposável pelo aparecimento de *padrões moiré*, comuns em função da popularização de fotos e filmes digitais.

(a) alta resolução - sem moiré

(b) baixa resolução - com moiré

(c) padrão moiré em destaque

Figura 8.1: Aparecimento de *padrões moiré*.

Do ponto de vista de *hardware*, o efeito do *aliasing* pode ser resolvido aumentando a resolução do dispositivo, ou seja, através do aumento da quantidade de elementos discretos presentes no espaço de apresentação da imagem. Isso faz com que os vários elementos discretos sejam menores, e portanto mais difíceis de serem notados. Do ponto de vista de *software*, podemos evitar a mudança abrupta de cores entre elementos discretos vizinhos, o que torna mais difícil a distinção entre eles. Infelizmente isso não é possível num dispositivo monocromático de apresentação, mas esses dispositivos são raros hoje em dia. Dispositivos de impressão em massa costumam ser monocromáticos (observe os fotolitos de uma gráfica), entretanto esses dispositivos costumam ser de natureza contínua.

Na figura acima notamos um segmento de reta (b), melhorado por aumento da resolução do dispositivo

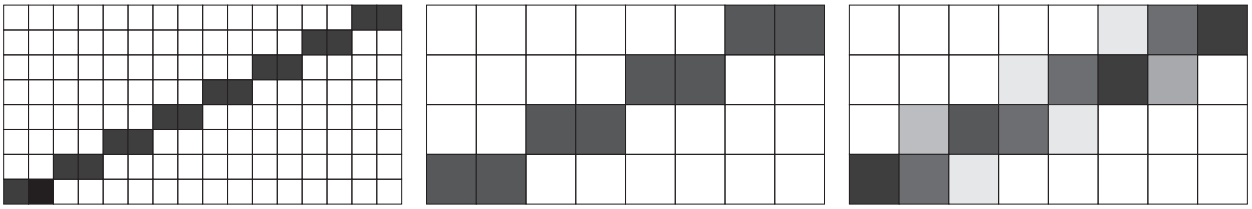


Figura 8.2: Antialiasing.

(a) e pelo uso de intensidades diferentes em cada *pixel* (c). Podemos observar que o aumento da resolução do dispositivo melhora a aparência do segmento de reta, entretanto a complexidade do *hardware* aumenta significativamente (para dobrar a resolução vertical e a horizontal, precisamos de 4 vezes mais memória, 4 vezes a velocidade de transmissão de informações e de um dispositivo de apresentação 4 vezes mais preciso) e ainda podemos observar o "efeito escada" no segmento, dessa vez com duas vezes mais cantos do que antes.

Para resolver o problema de *aliasing*, precisamos de uma fórmula para calcular o quanto da intensidade da cor deve ser usada em cada *pixel*. O sistema de cores RGBα pode facilitar bastante o processo de combinação de cores necessário no processo de *antialiasing*. Neste caso, intensidades menores da cor da primitiva gráfica são obtidas com valores mais altos da componente α da cor.

8.1 Amostragem Simples (*Unweighted)*

Vamos supor que deseja-se desenhar um segmento de reta entre dois pontos. Um representação matricial desse segmento deverá ser tão fina quanto possível. Uma vertical ou horizontal teria então a largura de um *pixel*. Vamos considerar então que uma primitiva gráfica qualquer tem uma determinada espessura, e que essa espessura não pode ser menor que um *pixel*. Considerando um *pixel* como um quadrado cujo centro está nas coordenadas que o identificam e que os vários pixels do dispositivo de apresentação não se sobrepõe, mas também não existe espaço vazio entre eles, podemos considerar uma primitiva gráfica cobre diversos *pixels*, sendo que a área coberta pela primitiva vária entre os vários pixels sob a primitiva gráfica. Podemos observar isso na figura abaixo.

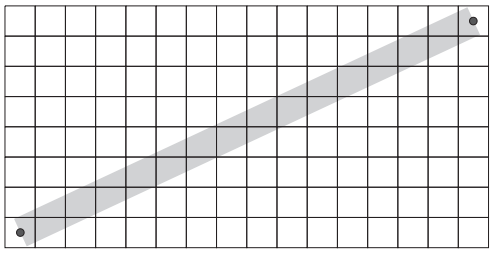


Figura 8.3: Uma reta com um *pixel* de espessura entre dois pontos cobre diferentes áreas dos *pixels* por onde passa.

Podemos associar a intensidade de cada um desses *pixels* à área coberta pela primitiva gráfica. Por exemplo, um *pixel* completamente coberto por uma reta preta, desenhada sobre um fundo branco, seria desenhado com intensidade 100% preta. Por outro lado, um *pixel* cuja área coberta pela reta é de 30% de sua área total, será desenhado com um tom de cinza 30% preto. A figura abaixo mostra como o segmento de reta acima poderia ser desenhado.

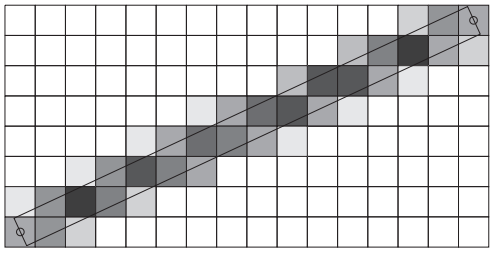


Figura 8.4: Segmento de reta desenhado usando amostragem simples.

De acordo com esta ideia, não importa onde a primitiva gráfica está passando pelo *pixel*. Desta maneira, se o segmento de reta "acerta em cheio" um *pixel*, mas cobre 90% de sua área, então a intensidade desse *pixel* será a mesma de outro em relação ao qual o segmento passa ao lado, mas que também cobre 90% de sua área.

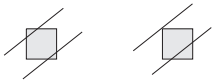


Figura 8.5: Pixels com a mesma área coberta, com o segmento passando em posições diferentes.

Entre os principais efeitos dessa característica da amostragem simples, é que aparecem pedaços mais claros no meio da primitiva. Isso é um efeito indesejável, pois dá ideia de descontinuidade. A amostragem ponderada produz efeitos bem melhores.

8.2 Amostragem Ponderada (*Weighted)*

Seria desejável considerar não somente a área ocupada, mas também sua posição. Nos casos em que a área atingida é mais próxima do centro do pixel, a intensidade deveria ser maior que nos casos em que a área está mais distante. Assim, poderíamos dividir a área de um *pixel* em várias pequenas áreas, de maneira essas pequenas áreas no centro "pesam" mais sobre a intensidade final do *pixel*.

Seria interessante também que essa área se estendesse além da área do *pixel*, de forma os *pixels* que muito próximos da primitiva que não são realmente atingidos por ela sejam usados no processo de desenho, dando a impressão de uma passagem mais contínua dos tons.

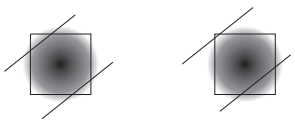


Figura 8.6: Pixels com a mesma área coberta, influenciados de maneiras difrentes

Na figura acima, temos quadrados que representam *pixels*. Esses *pixels* foram divididos em áreas menores, infinitamente pequenas, cada uma com um tom de cinza. As pequenas áreas mais próximas do centro estão mais escuras para indicar que elas pesam mais na intensidade do *pixel*. As pequenas áreas mais distantes estão mais claras, pois pesam menos na intensidade do *pixel*. Em (a), a região coberta pelo segmento é mais escuro que em (b), portanto o *pixel* em (a) seria desenhado com intensidade maior.

Para definir o peso da cada uma das pequenas áreas, precisamos de função de peso de tal forma que dada uma posição (x, y) no plano, obtemos um peso W(x, y). Assim cada uma das pequenas áreas contribui para a intensidade total do *pixel* com a porcentagem de sua área coberta vezes o seu peso. Esse total pode ser calculado pela integral da função W na área coberta pela primitiva.

Uma função W simples seria uma tal que a intensidade é máxima no centro do *pixel* e essa cai linearmente de acordo com a distância entre o ponto analisado e o centro do *pixel*. Desenhando essa função z = W(x, y) teríamos um cone cuja base está centrada no cento do *pixel* em questão. Um bom valor para o raio da base deste cone é o comprimento de um *pixel*, assim as pequenas áreas de cada *pixel* se sobrepõe de forma que um pixel, pode ser influenciado pelo desenho da primitiva mesmo que ela esteja diretamente sobre ele; um pequeno pedaço da primitiva pode influenciar mais de um *pixel* e não sobre nenhum pedaço dos vários *pixels* que não seja afetado pela função.

Dado o conceito da amostragem, é necessário agora prover formas eficientes de se realizar esses cálculos e incorporá-las a um algoritmo de desenho de primitivas gráficas.

Capítulo 9

Mapeamento de Texturas

Descrever a geometria e as propriedades de um objeto com o objetivo de se conseguir resultados realísticos é uma tarefa grandiosa. Imagine a quantidade de informação necessária para descrever uma parede de tijolos, uma parede de azulejos, um tapete ou um teclado de computador. Seriam necessários milhões de polígonos para se descrever o gramado de um campo de futebol e, apesar disso, um observador na arquibancada não é capaz de notar a grande maioria dessa informação. É verdade que uma descrição detalhada do gramado permitiria um grande realismo numa aplicação onde o observador tivesse a liberdade de se deslocar da arquibancada para dentro do gramado e então observar algumas folhas de grama bem de perto, mas a um custo enorme. Existem técnicas apropriadas para isso, técnicas que permitem o uso de grandes bases de dados geométricos mas que impõe um controle sobre o processo de síntese de imagens, de forma que a qualquer instante somente um mínimo necessário de informação é processada. Tais técnicas fogem ao escopo deste texto. Por ora, estamos interessados em descrever detalhadamente a superfície de um objeto, usando um modelo geométrico simples, que permita a formação de imagens cheias de detalhes (como os da casca de uma laranja, por exemplo) a um baixo custo computacional.

A ideia que prevalece nos sistemas de hoje é mapear uma imagem matricial sobre uma superfície de forma que a cor de cada elemento da imagem matricial (chamados *texels*, do inglês *texture elements*) contribua no processo de determinação da cor dos pixels nas regiões correspondentes. Essa técnica foi apresenta por Ed Catmull em sua tese de doutorado em 1974. Ela é conhecida como mapeamento de texturas (*texture mapping*). A textura pode ser um arquivo externo produzido a partir de uma foto (por exemplo) ou pode ser gerada de forma temporária e procedural pelo próprio programa que vai usá-la.

O mapeamento de texturas pode parecer uma técnica simples a princípio pois envolveria apenas interpolação bilinear, mas a natureza discreta dos elementos numa textura e a distorção causada pela projeção perspectiva associada à grande quantidade de informações numa textura, faz com resultados de qualidade sejam conseguidos apenas através de métodos sofisticados e à custa de uma grande quantidade de cálculos. Este texto aborda os métodos mais simples, de qualidade reduzida, mas que são base para se compreender o problema e as soluções sofisticas que já existem.

A referência aos vários elementos discretos de uma textura envolve a definição de um sistema de coordenadas próprio da textura, independente dos demais sistemas envolvidos na síntese de imagens. Uma imagem discreta pode ser considerada como uma matriz m x n de elementos discretos. Podemos associar à matriz vetores ortogonais u e v cujos índices indicam respectivamente a coluna e a linha da matriz. Sem perda de generalidade, podemos supor que dois escalares para u e v, no intervalo [0,1], representam todo o espaço da textura. De forma análoga, vamos associar à projeção da superfície na qual a textura será mapeada, vetores s e t de tal forma que os elementos discretos da imagem também encontram-se no intervalo de [0, 1] nessas duas dimensões.

Dado um polígono plano qualquer, podemos associar a cada um de seus vértices, coordenadas s e t que indicariam como a textura foi "grudada e esticada" sobre a superfície do polígono. Durante o processo de definição da cor dos *pixels* (rasterização), podemos fazer uso de uma função que mapeie coordenadas x e y do dispositivo de apresentação em coordenadas u e v da textura, de maneira que seja possível encontrar a cor do(s) *texel(s)* que influencia a cor do *pixel* analisado.

Dentre as dificuldades de se criar uma função assim, podemos ressaltar que a natureza discreta dos *pixels* e dos *texels* requer uma função que mapeie **áreas** do sistema de coordenadas de apresentação em **áreas** das coordenadas u e v, possivelmente passando por superfícies no sistema de coordenadas do mundo. Outro fator importante é que pode acontecer *aliasing* (em níveis muito elevados) nesse processo. A maneira como esse mapeamento é feito vai determinar a qualidade e a eficiência do processo de mapeamento.

Os sistemas de representação $s \times t$ e $u \times v$ são usados de forma alternada em fontes distintas. Em [1], por exemplo, $s \times t$ é o sistema de representação da textura enquanto $u \times v$ é o sistema da superfície do sólido. A ideia é apenas ter dois sistemas, proporcionando facilidades de mapeamento mas, para fins de simplificação, vamos supor que $s = u$ e $t = v$.

Seria possível mapear as coordenadas (u,v) da textura para o objeto (coordenadas do mundo) e depois projetar essas coordenas para (x', y') do sistema de coordenadas de apresentação. Entretanto como o sistema de coordenadas de apresentação é discreto ficaria inviável determinar quais coordenadas (u,v) deveriam ser usadas para encontrar a cor de cada coordenada (x', y'). Queremos fazer o mapeamento inverso, ou seja, queremos encontrar (u,v) para cada (x', y'), possibilitando a determinação da cor de cada *pixel* uma e uma única vez.

9.1 Mapeamento Linear

A maneira mais simples de se fazer esse mapeamento é sem dúvida a interpolação bilinear. Como os vértices dos polígonos tem coordenadas (u, v) determinadas durante sua modelagem, podemos rapidamente encontrar coordenadas (u, v) para todos os outros *pixels* dentro da área do polígono após a sua projeção. Para isso podemos interpolar ao longo das arestas do triângulo e depois interpolar cada linha sendo desenhada, conforme mostrado na figura 9.1.

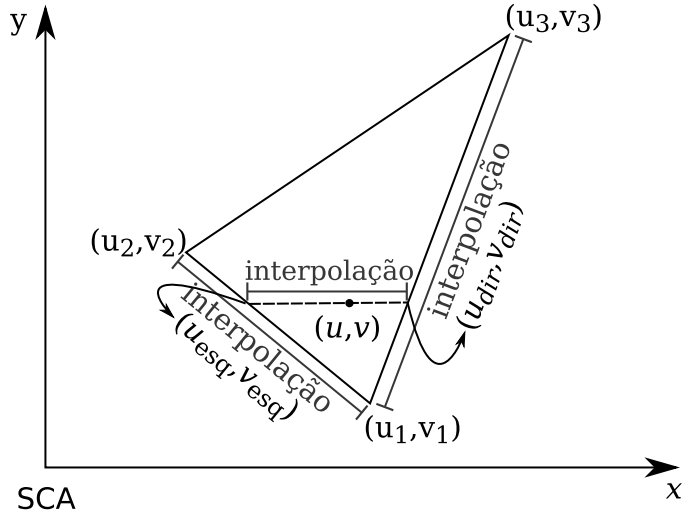


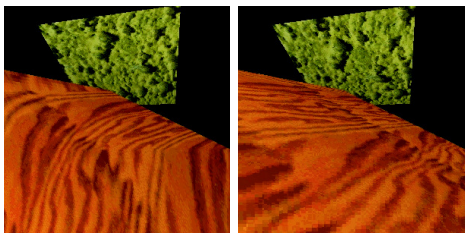
Figura 9.1: Interpolação bilinear das coordenadas de textura num triângulo.

A grande desvantagem desta técnica é que a projeção perspectiva não é linear e os efeitos de usar um mapeamento linear para exibir seus resultados causa distorções indesejáveis na imagem. Um

mapeamento linear tem a forma $y = ax + b$ e a fórmula que associa uma característica qualquer, linear em relação ao polígono (sistema de coordenadas do mundo) a cada posição (x', y') na projeção do mesmo não tem esse formato. Por exemplo, a equação 9.1 na página seguinte mostra que a função da profundidade num triângulo projetado não é linear.

Esse problema também existia no mapeamento linear de cores no algoritmo de Gouraud. Entretanto, naquela situação, a quantidade de informações usadas pelo cérebro (para perceber as três dimensões na imagem 2D) era relativamente pequena e portanto insuficiente para que o cérebro pudesse perceber que algo estava errado. No caso das texturas, a quantidade de informações é muito maior e é suficiente para que os resultados incorretos sejam notados pelo cérebro nos casos em que a distância de cada ponto do polígono ao centro de projeção varia muito (em termos significativos). Esse é o caso em que uma parte do polígono está bem perto do observador e outra está longe.

Nos casos em existe pouca variação de z ao longo de x e y, a distorção pode não ser notada. Nesses casos talvez seja interessante usar o mapeamento linear, já que a sua natureza simples o torna também eficiente. Nas figuras 9.2 são mostrados dois polígonos. O de baixo, com uma textura semelhante à madeira tem muita variação de z ao longo de x e y enquanto que o de cima não. Comparando as duas, podemos notar a distorção causada pelo mapeamento linear (a) em contraposição ao mapeamento não-linear (b) que veremos a seguir. As diferenças são facilmente percebidas no polígono de baixo, mas dificilmente percebidas no polígono de cima.



(a) linear (b) não-linear

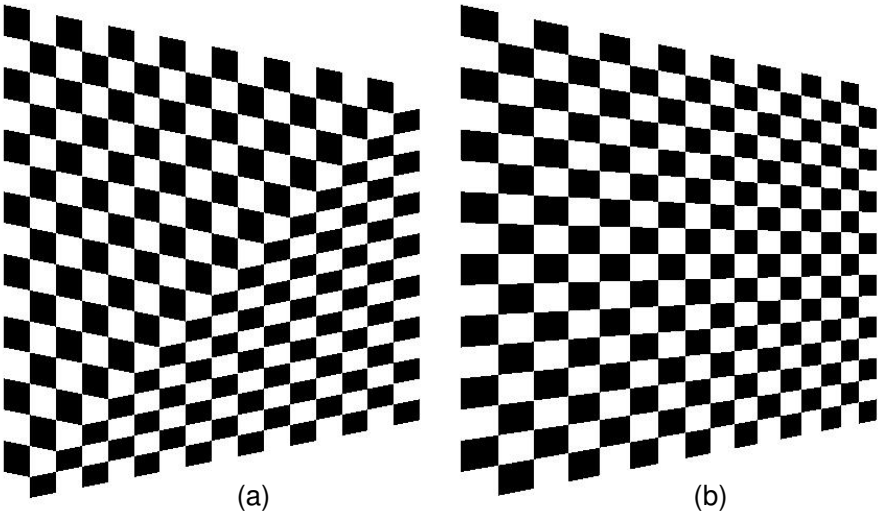
Figura 9.2: Comparação entre mapeamentos de textura (padrões naturais).

Na figura 9.3, que apresenta um padrão mais previsível, fica mais fácil perceber a natureza da distorção do mapeamento linear.

9.2 Mapeamento Não-Linear

Vimos pelos resultados do mapeamento linear que os valores de (u,v) não variam linearmente no sistema de coordenadas de apresentação. Queremos entender qual é a forma de variação de uma característica linear no polígono, depois da projeção do mesmo. Essa característica pode ser as coordenadas de textura ou a cor ou alguma outra característica. Vamos começar pela profundidade, que é evidentemente importante para a síntese de imagens e é linear no polígono.

Na seção 6.3 já foi adiantado que calcular a profundidade de cada pixel é uma tarefa dificultada pela projeção perspectiva. Podemos contornar a perda da profundidade usando 3 coordenadas para cada pixel do dispositivo de apresentação, calculadas a partir das 4 coordenadas dos vértices no sistema de coordenadas do mundo, como resultado da multiplicação das matrizes de transformação e projeção. Uma estratégia é transformar de coordenadas homogêneas para euclidianas assim:



mapeamento linear (a) e não linear (b)

Figura 9.3: Comparação entre mapeamentos de textura (padrão xadrez).

$$x' = x/w$$
$$y' = y/w$$
$$z' = w$$

A coordenada z após a projeção seria a distância entre o plano de projeção e a origem. A transformação anterior produz uma profundidade que é diretamente proporcional à profundidade antes da projeção ou equivalente, no caso do plano de projeção estar em $z = 1$.

Ainda relembrando a fórmula da projeção, podemos relacionar as coordenadas (x', y') com a profundidade (z) dos pontos de um polígono. As coordenadas de apresentação (x', y') de um ponto projetado eram dadas por $x' = z'\frac{x}{z}$ e $y' = z'\frac{y}{z}$. Podemos supor, sem perda de generalidade que $x' = \frac{x}{z}$ e portanto:

$$z = \frac{x}{x'} \tag{9.1}$$

Podemos notar que z não é uma função linear de x', ou seja, algo que varia linearmente num objeto não varia linearmente na sua projeção. Entretanto, invertendo a fórmula conseguimos algo que é linear em *x'*:

$$\frac{1}{z} = \frac{x'}{x}$$

Essa equação sugere que algo linear no sistema de coordenadas do mundo tem uma proporção com a profundidade que é linear no sistema de coordenadas de apresentação.

Observe a figura 9.4. Se a profundidade fosse constante ao longo do segmento $\overline{P_1P_2}$, ela seria linear (na verdade, mais do que linear, seria constante) em relação à variação de y. Porém como a profundidade é uma característica linear em relação a y, ela não é linear em relação a *y'*. Com o segmento $\overline{P'_1P'_2}$ dividido em quatro seções iguais, notamos que seções correspondentes de $\overline{P_1P_2}$ são proporcionais a z. Temos portanto mais indício de que a proporção com z pode fornecer os cálculos que resolvem nosso problema de mapeamento. Suponha que existe uma característica qualquer c que é linear em relação

à x e y no polígono. Sendo linear, sabemos que existem constantes α e β que permitem escrever a fórmula:

$$c = \alpha x + \beta$$

Substituindo x por $x'z$, temos:

$$c = \alpha x' z + \beta$$

Finalmente, investigando como seria a proporção entre c e z, temos:

$$\frac{c}{z} = \alpha x' + \frac{\beta}{z}$$

Podemos notar então que qualquer característica linear no polígono tem uma proporção com a profundidade que é linear na projeção. Sabemos que as coordenadas de textura variam linearmente no polígono e portanto u/z e v/z variam linearmente com relação a *x'* e *y'*. Sabendo o valor de u/z, de v/z e de $1/z$ em cada pixel, sabemos as coordenadas (u,v) do pixel e portanto sabemos escolher uma cor para o pixel baseada na cor correspondente da textura.

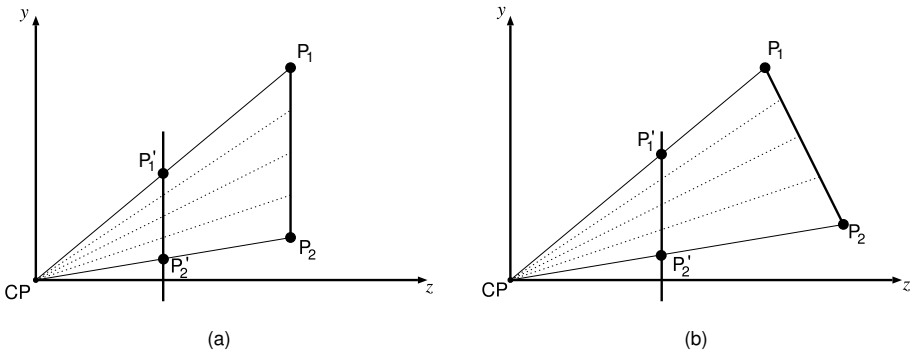


Figura 9.4: Variação de x e y na projeção perspectiva.

Você deve se lembrar que o algoritmo do *z-buffer* também precisa determinar a profundidade de cada pixel da imagem, portanto, o mapeamento de textura fica mais eficiente se implementado junto com a determinação de superfícies vizíveis, que por sua vez deve ser implementada junto com a rasterização de triângulos. Dessa forma todo cálculo realizado pixel a pixel não é somente incremental, mas é também realizado uma única vez.

A disponibilidade de hardware especializado permite fazer todo esse processamento em tempo interativo, mas existem várias técnicas para acelerar a rasterização em situações específicas ou mitigando os erros em situações genéricas. Uma estratégia clássica é calcular o mapeamento não linear de textura somente em alguns pixels e fazer mapeamento linear entre os valores corretos, por exemplo, considerando a perspectiva somente a cada 16 pixels e interpolando linearmente entre eles.

Também é possível ganhar um pouco de velocidade e ainda simplificar a implementação desse algoritmo se calcularmos o gradiente das 3 características que precisamos ao invés de fazer várias interpolações lineares. O gradiente indica o quanto cada característica varia com x ou com y no triângulo em coordenadas de apresentação. Suponha que existe um triângulo em que uma característica varia linearmente. Conhecemos o valor da característica nos pontos P_1, P_2 e P_3 do triângulo. Vamos então determinar dois novos pontos: P_4 que difere de P_1 apenas em y e P5 que difere de P1 apenas em x, conforme a figura 9.5.

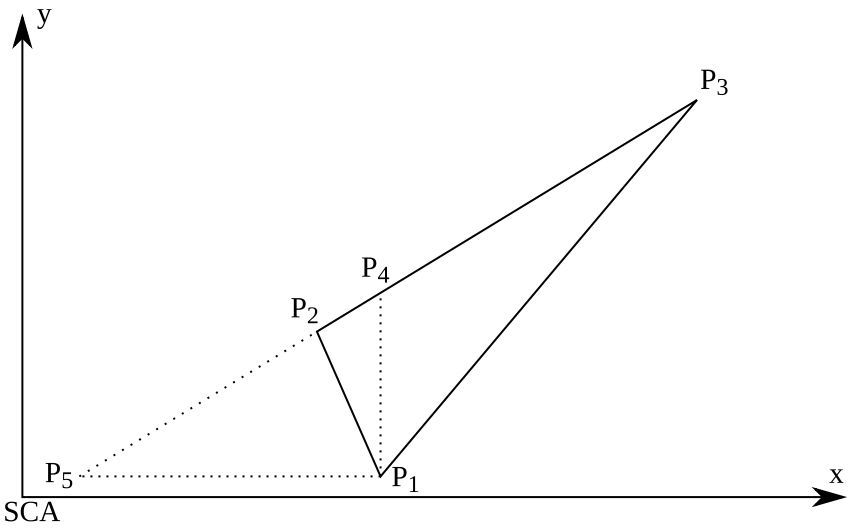


Figura 9.5: Esquema para calcular gradiente de um triângulo

Sabemos, por semelhança de triângulos, que:

$$\frac{y_3 - y_4}{y_3 - y_2} = \frac{x_3 - x_4}{x_3 - x_2}$$

Como $x_4 = x_1$podemos escrever:

$$\frac{y_3 - y_4}{y_3 - y_2} = \frac{x_3 - x_1}{x_3 - x_2}$$

E então, com um pouco de manipulação algébrica, temos:

$$y_4 = \frac{(y_2 - y_3)(x_3 - x_1)}{x_3 - x_2} + y_3$$

Esta fórmula é análoga para qualquer característica c linear no triângulo, então:

$$c_4 = \frac{(c_2 - c_3)(x_3 - x_1)}{x_3 - x_2} + c_3$$

Para calcular o quanto a característica c varia com y no triangulo, vamos calcular a proporção das duas variações ao longo do segmento $\overline{P_1 P_4}$:

$$\begin{aligned}
\frac{\Delta c}{\Delta y} = \frac{c4 - c1}{y4 - y1} &= \frac{\frac{(c_2 - c_3)(x_3 - x_1)}{x_3 - x_2} + c_3 - c_1}{\frac{(y_2 - y_3)(x_3 - x_1)}{x_3 - x_2} + y_3 - y_1} \\
&= \frac{\frac{(c_2 - c_3)(x_3 - x_1) + (x_3 - x_2)(c_3 - c_1)}{x_3 - x_2}}{\frac{(y_2 - y_3)(x_3 - x_1) + (x_3 - x_2)(y_3 - y_1)}{x_3 - x_2}} \\
&= \frac{(c_2 - c_3)(x_3 - x_1) + (x_3 - x_2)(c_3 - c_1)}{(y_2 - y_3)(x_3 - x_1) + (x_3 - x_2)(y_3 - y_1)}
\end{aligned}$$

De maneira análoga,

$$\frac{\Delta c}{\Delta x} = \frac{c5 - c1}{x5 - x1} = \frac{\frac{(c_3 - c_2)(y_3 - y_1)}{y_2 - y_3} + c_3 - c_1}{\frac{(x_3 - x_2)(y_3 - y_1)}{y_2 - y_3} + x_3 - x_1}$$
$$= \frac{(c_3 - c_2)(y_3 - y_1) + (y_2 - y_3)(c_3 - c_1)}{(x_3 - x_2)(y_3 - y_1) + (y_2 - y_3)(x_3 - x_1)}$$

Observe que os dois denominadores são iguais e portanto o denominador só precisa ser calculado uma única vez para todas as características que se deseja interpolar.

Concluindo, o mapeamento não linear de texturas pode ser implementado com base no algoritmo de rasterização de triângulos da seguinte forma:

Algoritmo 9.1 Mapeamento não linear de texturas

1.Calcular os gradientes $\frac{\Delta 1/z}{\Delta x}$, $\frac{\Delta 1/z}{\Delta y}$, $\frac{\Delta u/z}{\Delta x}$, $\frac{\Delta u/z}{\Delta y}$, $\frac{\Delta v/z}{\Delta x}$ e $\frac{\Delta v/z}{\Delta y}$.
2. Para cada metade do triângulo:
2.1 Calcular ymin, ymax, xesq, xdir, "incremento xesq" e "incremento xdir".
2.2. Inicializar y com ymin.
2.3. Inicializar $1/z$, u/z e v/z usando os valores do vértice de menor y.
2.4. Enquanto y for menor que ymax, faça:
2.4.1. Inicializar x com xesq.
2.4.2. Enquanto x for menor que xdir, faça:
2.4.2.1. Calcular z, u e v.
2.4.2.2. Calcular a cor do pixel baseado em (u,v).
2.4.2.3. Desenhar o pixel.
2.4.2.4. Incrementar o valor de u/z, v/z e $1/z$ com base em seus gradientes em x.
2.4.2.5. Incrementar o x.
2.4.3. Incrementar o valor de xesq e de xdir com base em seus respectivos incrementos.
2.4.4. Incrementar o valor de u/z, v/z e $1/z$ com base em seus gradientes em x e em y.
2.4.5. Incrementar o y.

Entretanto, ainda resta o problema de *aliasing*. Uma solução para isso é fazer o mapeamento de áreas do SCA em áreas do SCT (em oposição a mapear pontos). Neste caso podemos fazer uma amostragem ponderada da cor dos *texels* envolvidos (conforme visto na seção que trata de *antialising*). Para fazer o mapeamento de áreas, podemos considerar cada *pixel* como um quadrado. Cada uma de suas quatro extremidades pode ser mapeada no SCT, definindo assim uma área nele. Evidentemente, existe um custo computacional alto para fazer isso.

Capítulo 10

Representação de Sólidos Revisitada

Após a apresentação dos conceitos de representação de sólidos no capítulo 2, várias técnicas foram discutidas e algumas delas exigem mais informação do que havia sido considerado inicialmente.

O cálculo de iluminação trouxe a necessidade da descrição de normais, uma para cada vértice, que permitem uma melhor aproximação da imagem de uma superfície curva através de uma malha poligonal. No caso de uma superfície contínua, cada vértice com sua normal pode ser compartilhado entre várias faces. Para uma superfície descontínua, o vértice pode ser compartilhado entre várias faces mas a normal não. Um mesmo vértice de um objeto deve poder ser associado com normais diferentes, dependendo da face onde ele aparece.

O mapeamento de texturas cria uma situação semelhante. Cada vértice deve estar associado a uma coordenada de textura, porém, em superfícies descontínuas, um mesmo vértice pode estar associado a diferentes coordenadas de textura, dependendo da face onde ele aparece.

Nosso "formato de arquivo" para descrição de sólidos precisa ser repensado. Ele precisa ter descrições de normais e coordenadas de texturas, além de outros elementos. Vamos separar os dados em quatro partes:

- coordenadas de vértices (pontos 3D),
- normais (vetores 3D),
- coordenadas de textura (pontos 2D) e
- faces (relações entre os elementos anteriores, da forma vista anteriormente, mais a possibilidade de associar um mesmo vértice à normais ou coordenadas de textura diferentes).

Considere que o objeto a ser descrito é um dado (cubo com textura que faz cada face ter um desenho diferente). O dado poderia ser representado assim:

```
vertices 8
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
normais 6
0 0 1
0 0 -1
0 1 0
0 -1 0
1 0 0
-1 0 0
textura 14
0 0
0.25 0
```

```
0 0.33
0.25 0.33
0.5 0.33
0.75 0.33
1 0.33
0 0.66
0.25 0.66
0.5 0.66
0.75 0.66
1 0.66
0 1
0.25 1
faces 6
7 2 2 6 2 3 2 2 13 3 2 12 -1
0 5 5 3 5 10 2 5 9 1 5 4 -1
1 0 4 2 0 9 6 0 8 5 0 3 -1
4 3 2 0 3 0 1 3 1 5 3 3 -1
6 4 8 7 4 7 4 4 2 5 4 3 -1
3 1 10 0 1 5 4 1 6 7 1 11 -1
```

As quatro partes do arquivo são identificadas por nomes. A palavra "textura" identifica a parte de coordenadas de textura. Depois do nome de uma parte existe um número inteiro que indica quantos elementos daquele tipo existem. Isso facilitaria uma implementação ao permitir ler uma certa quantidade de números antes de ler outra *string*. A parte de vértices contém várias triplas de números, a parte de normais também contém várias triplas de números. A parte de coordenadas de texturas contém várias duplas de números e finalmente a parte de faces contém algo mais sofisticado. Cada face é um conjunto de triplas de números inteiros, cada tripla representa uma posição da face. Uma face tem três ou mais vértices, cada um deles é representado por três índices que fazem associação, respectivamente, com a coordenadas do vértice, as coordenadas do vetor normal e as coordenadas de textura. Para sinalizar o final dos vértices de uma face (já que a quantidade é variável), o valor -1 no lugar de um índice de coordenadas de vértice (que é um índice inválido) indica o final dos vértices de uma face.

Identifique todos os pontos da seção de coordenadas de textura na figura 10.1 para ver como eles se relacionam aos vértices.

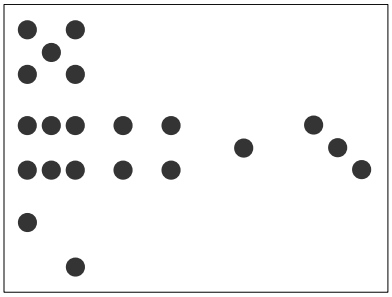


Figura 10.1: Textura de um dado.

Esse sistema proposto permite que vértices, normais e texturas sejam reusados ou multiplicados conforme a necessidade do modelo. O dado modelado tem 6 faces, que formam uma superfície descontínua. Cada índice de coordenadas de vértice aparece três vezes na parte de faces, já que cada vértice é compartilhado por três faces num cubo. Porém em cada face, eles podem aparecer com normais e

coordenada de textura diferentes. O vértice 6, por exemplo, aparece associado com a normal 2 na face 0 e aparece associado com a normal 0 na face 2. De maneira análoga, o vértice 2 aparece associado com a posição 9 da textura na face 2, mas aparece associado com a posição 13 da textura na face 0.

Numa situação real ainda faltariam elementos como a textura propriamente dita e as características do material do sólido.

Referências Bibliográficas

[1] ANGEL, E. Interactive Computer Graphics: a top-down approach with Open-GL. 2nd ed., Addison-Wesley, 2000.

[2] BEAM, J. Tutorial - Introduction to Software-based Rendering: Triangle Rasterization. Janeiro 2009. Disponível em <http://joshbeam.com/articles/triangle_rasterization/>. Acesso em agosto de 2014.

[3] FOLEY, J. D. *et al. Computer Graphics: principles and practice*. 2nd ed. in C, Addison-Wesley, 1996.

[4] GOMES, J. & VELHO, L. *Computação Gráfica*. Vol. 1, Rio de Janeiro, IMPA, 1998.

[5] GRIMSDALE, R. L. e KAUFMAN, A. *Advances in Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*. Springer Science & Business Media, 2012. p.29.

[6] KOBBELT, L & BOTSCH, M. A survey of point-based techniques in computer graphics. *Computers & Graphics*. v.28, n.6, p. 801-814. 2004.

[7] NEWELL, M.E; NEWELL, R.G e SANCHA, T.L. A Solution to the Hidden Surface Problem. In: *Proceedings of the ACM National Conference 1972*. p. 443-450.

[8] SAVCHENKO, S. *3D Graphics Programming: Games and Beyond*. Sams, 2000.