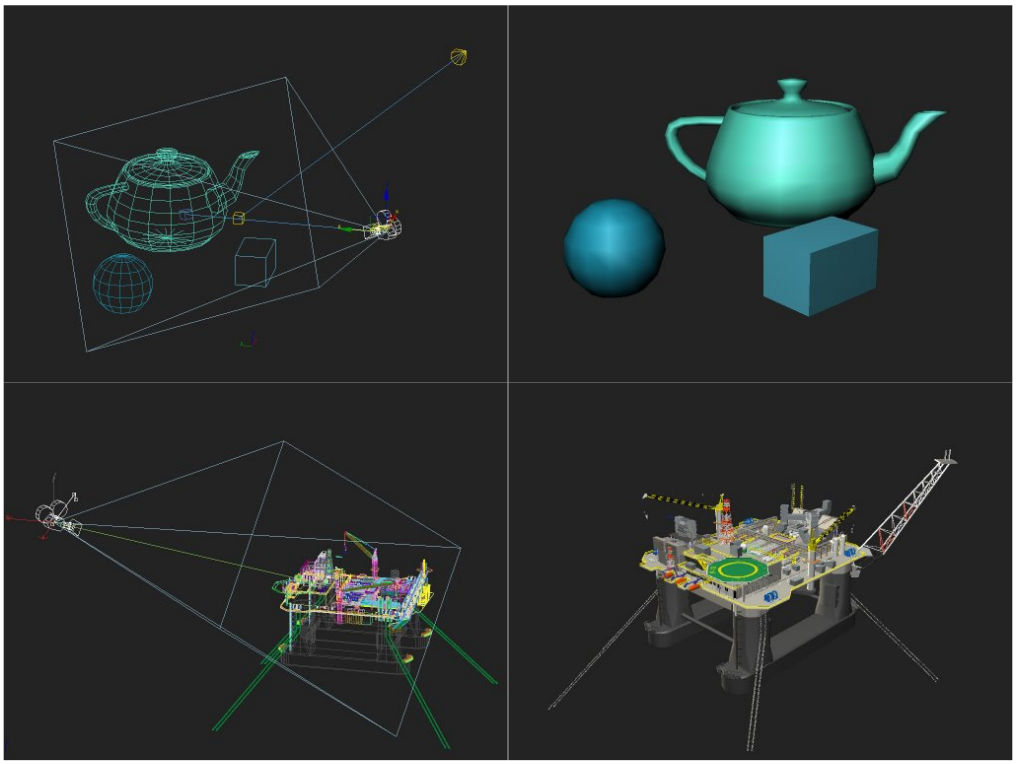
7. Sistemas Gráficos 3D

Num jogo de computador onde um personagem percorre um cenário virtual, a tela do computador precisa exibir, a todo instante, o que ele estaria vendo do cenário naquele momento do jogo. Este requisito de eficiência também aparece nos programas que criam modelos e exibem graficamente resultados de simulações numéricas para áreas técnico-científicas como a Engenharia e a Medicina. Nestes programas a imagem que aparece na tela tem que refletir o momento corrente na simulação computacional.

Programas gráficos com fortes requisitos de eficiência utilizam placas de vídeo especializadas que possuem funções para gerar imagens a partir de descrições de cenas como ilustra a Fig. 7.1. O lado esquerdo desta figura mostra as malhas poligonais que definem os objetos, a luz e a câmera sintética. No lado direito vemos a imagem gerada por estas placas.



(a) modelos (b) imagens produzidas

Fig. 7.1 – Entrada e saída do algoritmo de mapas de profundidade

As imagens da Fig. 7.1(b) e (c) não são tão realistas quanto às produzidas pelo algoritmo de Rastreamento de Raios. Elas são produzidas pelo algoritmo de mapa de profundidades ou *ZBuffer* que é mais eficiente. Esta eficiência é obtida à custa de um realismo visual mais

limitado e do armazenamento na memória de um mapa (*buffer*) com valor da profundidade (*z*) da superfície que contribui para a cor armazenada em cada *pixel* da imagem gerada.

Biblioteca de funções, denominadas de Sistemas Gráficos 3D, rodam eficientemente nas placas gráficas e oferecem ao programador uma abstração de alto nível que permite, em essência, definir as luzes e uma câmera e a partir daí desenhar polígonos no espaço tridimensional. Estas funções se encarregam de gerar a imagem que é vista pela câmera.

Na linguagem da literatura da Computação Gráfica os programas que utilizam estes sistemas são chamados de programas de aplicação ("aplicam" o sistema gráfico para alguma área do conhecimento) e a interface de programação por eles oferecida de API (*Application Programmers Interface*). Os programadores que escrevem os programas de aplicação também são referenciados nesta literatura como programadores de aplicação.

O Sistema Gráfico 3D mais bem sucedido no mercado é OpenGL™ (*Open Graphics Library*) que foi inicialmente desenvolvido pela Silicon Graphics™ para suas estações de trabalho. Posteriormente se tornou um produto de aberto e hoje ele roda em praticamente todos os computadores.

Até recentemente, o OpenGL™ rodando na unidade de processamento da placa, GPU (*Graphical Processing Units*), era fornecido ao programador de aplicação como uma caixa preta. O programador de aplicação fazia uso desta biblioteca OpenGL sem poder modificar nenhum passo dos algoritmos embutidos nela. Com o advento das placas gráficas programáveis o programador passou a poder interferir nos diversos passos destes algoritmos de forma a gerar códigos mais eficientes e a produzir efeitos especiais que não existiam na biblioteca padrão. Com esta nova oportunidade o conhecimento detalhado do algoritmo interno do OpenGL™ se tornou mais relevante. Como agora é possível modificarmos as etapas do algoritmo, precisamos conhecer como elas são normalmente implementadas antes de fazermos qualquer modificação. O algoritmo básico dos Sistemas Gráficos 3D atuais é o algoritmo denominado de *ZBuffer* ou Mapa de Profundidade.

Este capítulo procura apresentar como um Sistema Gráfico 3D pode ser implementado, dando ênfase em dar uma visão geral e explicar ponto chaves da implementação do *ZBuffer*. Esperamos que com este conhecimento o leitor possa fazer também um bom uso do OpenGL. A evolução deste sistema ou a substituição dele por outro, como o DirectX da MicroSoft™, não torna inútil o investimento de adquirir o conhecimento deste capítulo. Eles são muito semelhantes e se fundamentam nas mesmos conceitos. Por isto, achamos que esta é a forma mais permanente de aprender a programar Sistemas Gráficos, tentando entender como ele pode ser implementado.

Rendering pipeline

Um aspecto que destinge o algoritmo de mapa de profundidade do algoritmo de rastreamento de raios é que ele permite que objetos sejam anexados a cena um de cada vez e após cada inserção temos uma imagem correta da cena parcial. O algoritmo de traçado de raios, por outro lado, precisa da definição de todos os objetos da cena antes de calcular a cor de qualquer *pixel*. Se um novo objeto é acrescentado à cena todos os cálculos do algoritmo de rastreamento de raios precisam ser refeitos. Isto não ocorre no algoritmo de mapas de profundidade.

A idéia geral do algoritmo de mapa de profundidade é que cada triângulo, da malha que compõe a fronteira dos objetos[1], seja projetado na superfície da tela e transformado em fragmentos[2] correspondentes a cada *pixel* em que ele se projeta. Os fragmentos, que tiverem valor de profundidade menor que o armazenado no mapa de profundidades, alteram a cor e a profundidade do *pixel* de sua posição. Ou seja, como o algoritmo de mapas de profundidade tem tanto o mapa de cores quanto o de profundidades da cena, ao introduzirmos um novo objeto na cena podemos determinar quais *pixels* da imagem são afetados pelo novo objeto sem termos armazenado os objetos que vieram antes dele.

O processamento de cada um dos polígonos que compõe a superfície de fronteira de cada um dos objetos de uma cena segue uma seqüência de passos em que a saída de um passo é a entrada do passo subseqüente. Mais ainda, o atraso de um passo atrasa a toda a cadeia. Por isto o nome "processo de produção de síntese de imagens", ou *rendering pipeline* como já se tornou conhecido em nosso idioma.

A Fig. 7.2 mostra na parte superior os três macro-passos de produção de uma imagem com base no algoritmo de mapa de profundidades: aplicação, geometria e rastreio[3]. Abaixo de cada uma destes macro-passos temos passos menores que ocorrem dentro dele. As seções que se seguem neste capítulo procuram detalhar um pouco mais o que ocorre em cada um destes passos.

[1] Os modelos implícitos de objetos, do tipo da esfera dada pelo centro e pelo raio, precisam ser convertidos em malhas de polígonos que representam a superfície para serem tratados por este algoritmo.

[2] Fragmentos são os quadradinhos correspondentes aos *pixels* acrescidos de informações como profundidade, cor, textura, etc... O OpenGL™ define vários procedimentos que podem ser aplicados a estes fragmentos para produzir efeitos especiais e controlar melhor a imagem gerada.

[3] Rastreio é utilizada aqui como uma tradução do termo inglês *rasterization*.

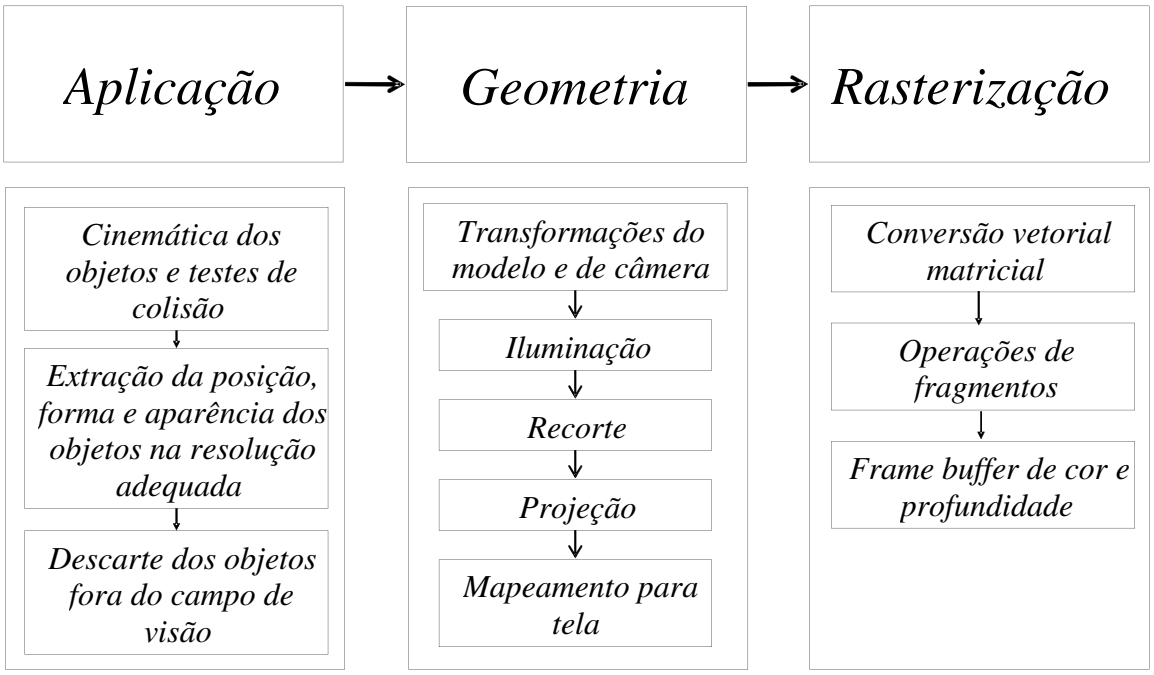


Fig. 7.2 – Passos do *rendering pipeline*

Aplicação

A primeira etapa do processo da *rendering pipeline* consiste no programa de aplicação percorrer suas estruturas de dados e extrair dela uma descrição da cena observada pela câmera naquele momento da simulação computacional. A geometria extraída das estruturas de dados do programa de aplicação é passada para o próximo passo em termos de primitivas da biblioteca gráfica que são: polígonos (triângulos e quadriláteros), linhas e pontos. Ou seja, qualquer que seja a forma de representar a geometria dos objetos da aplicação nesta fase ela geralmente se transforma em malhas de polígonos do tipo das ilustradas na Fig. 7.3.

As primitivas geométricas do tipo faixa de triângulos (*triangular strip*) são de certa forma redundante, uma vez que poderiam ser descritas por um conjunto de triângulos isolados. Elas existem para otimizar o processamento. A descrição das primitivas se faz com base nos seus vértices, e, como veremos a seguir neste capítulo, neles se concentram a maioria dos cálculos da *rendering pipeline*. Quando um vértice, compartilhado entre vários triângulos, é passado uma só vez estes cálculos não são repetidos e sistema fica mais eficiente.

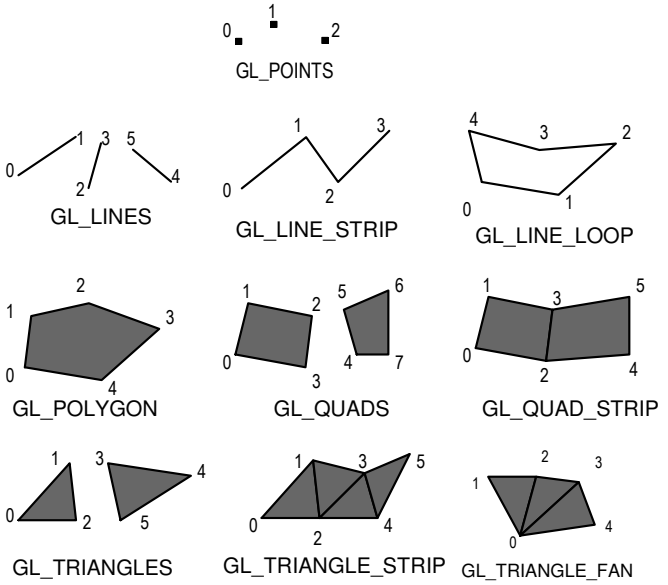


Fig. 7.3 – Tipos de primitivas gráficas do OpenGL™

A etapa "aplicação" é muito dependente do programa que estamos desenvolvendo e de como armazenamos nele a informação geométrica dos objetos visíveis. Discutimos aqui apenas algumas técnicas que impactam diretamente na performance da componente gráfica dos programas. O objetivo do resto desta seção é o de dar apenas uma breve introdução a esta primeira etapa do *rendering pipeline*. O leitor interessado deve procurar os artigos mais recentes que tratam de colisão, multi-resolução, descarte e oclusão. Estes assuntos ainda são temas abertos para pesquisa.

Colisão

Em aplicações onde os objetos da cena, a câmera ou a iluminação muda de posição precisamos recalcular, a cada quadro da animação, a nova posição dos objetos e as possíveis interações geométricas que ocorreram entre eles. Testes de colisão entre muitos objetos podem se tornar uma tarefa muito cara para sistemas tempo real uma vez que necessitaríamos testar a possível interseção da geometria de cada objeto com todos os demais. A combinação de um conjunto de n objetos dois a dois leva a um algoritmo de complexidade $O(n^2)$. Quando n é grande é necessário utilizarmos estruturas de dados com informações espaciais da posição dos objetos de forma a reduzir o número de pares de objetos candidatos a colidir em cada instante.

Multi-resolução

Em aplicações que lidam com grandes quantidades de polígonos é também comum termos a geometria armazenada em diversas resoluções que são utilizadas dependendo da distância do objeto a câmera. Objetos distantes, que aparecem pequenos, são exibidos com uma geometria aproximada e objetos próximos, que ocupam grande parte da superfície de visualização são mostrados com mais detalhes. A Fig. 7.4 mostra quatro modelos de um coelho em resoluções diferentes, cada modelo é apropriado para ser exibido a uma certa distância do observador. A estrutura de dados da aplicação pode simplesmente armazenar

os diversos níveis de resolução como o modelo do coelho e neste caso temos o que se chama de níveis de detalhe discretos (*discrete levels of detail*).

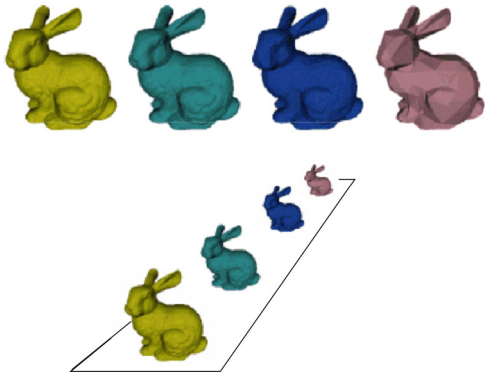


Fig. 7.4 – Modelo de um coelho em 4 resoluções

Existem estruturas de dados que em tempo de exibição são capazes de fornecer um modelo numa resolução especificada, sem causar impacto na performance do programa. Ou seja, a medida que o objeto e a câmera se afastam a malha que representa o objeto vai ficando cada mais simples. São os modelos de multi-resolução dinâmicos.

Devemos destacar aqui que a resolução necessária num ponto da malha que representa um objeto é, em última análise dependente do erro perceptual que a malha menos refinada produz. Se ele for aceitável a malha simplificada deve ser utilizada. Este erro é função da geometria local do objeto (objetos planos precisam de poucos triângulos), da projeção deste erro no plano de projeção (erros distantes são menos perceptivos) e até das características de iluminação e textura local que podem maquiar um erro. A Fig. 7.5 mostra uma malha que representa um terreno onde as duas primeiras características (geometria e distância) foram consideradas para gerar a imagem de um pedaço de terreno.

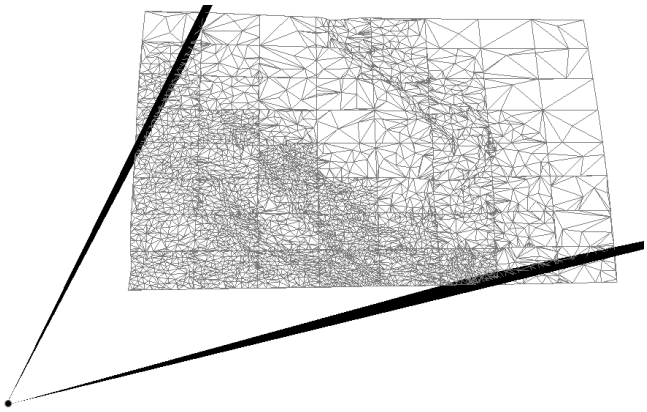


Fig. 7.5 – Modelo em multi-resolução dependente do erro projetado
(extraída da dissertação de Rodrigo Toledo)

Descarte

Um último processo que pode ocorrer na etapa aplicação consiste em não enviar para o *rendering pipeline* objetos que de antemão sabemos não serem visíveis pela câmera. Estes objetos podem simplesmente estar fora do campo de visão da câmera como ilustra a

Fig. 7.6. Note que nesta figura as esferas podem ser os objetos em si ou podem ser volumes auxiliares que envolvem objetos complexos. Neste último caso a idéia consiste em testar estes volumes envolventes simples com o prisma da câmera. Se eles estiverem fora do campo de visão da câmera o objeto complexo que eles envolvem também não é visível. Esta idéia pode ser incrementada criando-se hierarquicamente uma árvore de volumes envolventes onde cada pai envolve todos os nós filhos. Com esta estrutura podemos testar a árvore a partir da raiz. Quando um nó está fora do campo de visão não precisamos nos preocupar com seus filhos.

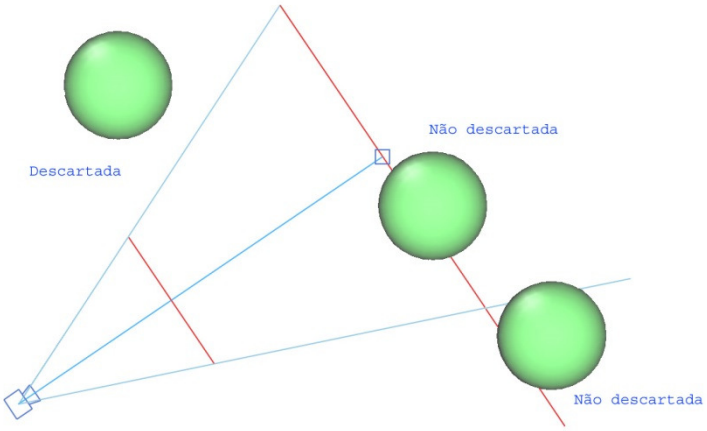


Fig. 7.6 – Descarte de esferas que não são visíveis pela câmera
(extraída da dissertação de Maurício Hofmam).

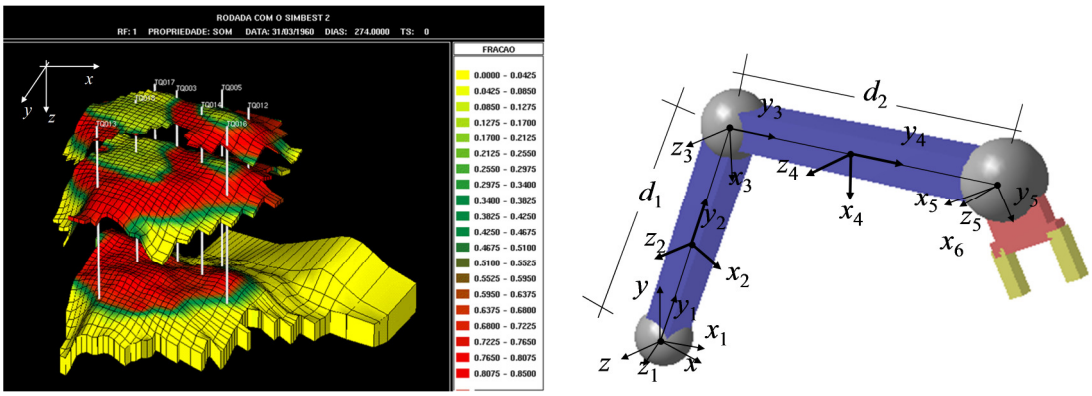
Geometria

As primitivas gráficas chegam a etapa de geometria com o programador de aplicação fornecendo ao sistema gráfico listas de coordenadas de seus vértices. A primeira questão a ser discutida diz respeito ao sistema de coordenadas estes vértices estão descritos.

As coordenadas das primitivas de um modelo podem ser naturalmente fornecidas com relação a um único sistema de coordenadas, como ilustra o modelo de reservatório mostrado na Fig. 7.7(a). O modelo numérico de Volumes Finitos deste reservatório tem as coordenadas dos vértices escritas em relação ao sistema de eixos mostrado no canto superior da figura. Em outras simulações numéricas importantes, como Elementos Finitos, as coordenadas dos vértices também são fornecidas em um único sistema de eixos.

Por outro lado, em aplicações onde temos peças com vínculos e movimentos relativos, do tipo do braço mecânico mostrado na Fig. 7.7(b), necessitamos de diversos sistemas de coordenadas para descrever sua geometria, como discutimos no capítulo sobre transformações geométricas.

Um Sistema Gráfico geral deve dar suporte a ambos tipos de aplicações. Ou seja, os vértices das primitivas podem ser dados em um sistema único para toda a cena ou cada objeto da cena deve poder ser descrito em um sistema próprio de coordenadas.



(a) reservatório (b) braço mecânico

Fig. 7.7 – Sistemas de coordenadas das aplicações

Transformação do sistema dos objetos para o da câmera (*model view*)

O OpenGL™ prevê que as coordenadas fornecidas para descrever as primitivas gráficas de um objeto sofrem primeiramente uma transformação geométrica denominada *model-view*. Esta transformação leva do sistema de coordenadas dos objetos (*object coordinate system*) na qual o modelo este descrito para o sistema de coordenadas do olho (*eye coordinate system*), chamado aqui de sistema da câmera, que é comum para todos os objetos.

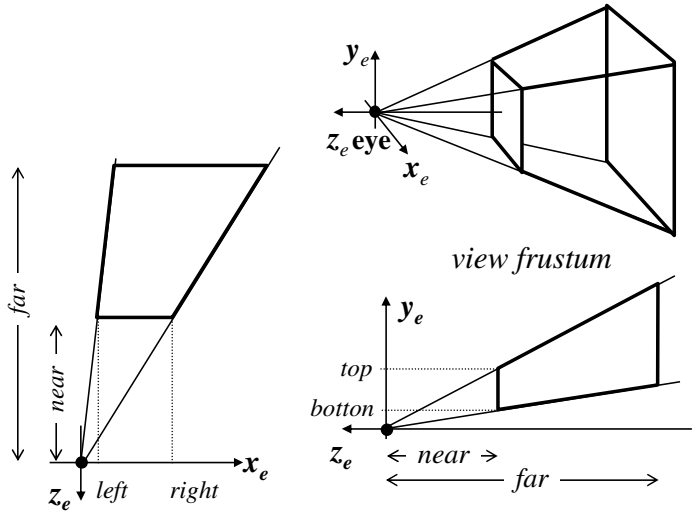
Para entendermos quem é o sistema de coordenada da câmera, a Fig. 7.8(a) ilustra a sua posição padrão, segundo a especificação do OpenGL™. O centro de projeção (***eye***) está na origem com a câmera voltada para a direção negativa do eixo ***z***. O eixo ***y*** indica a direção vertical da câmera. Esta posição padrão pode parecer estranha à primeira vista mas ela existe para simplificar a implementação dos próximos passos e, além disto, a câmera pode ser re-posicionada como veremos logo a seguir.

Para os paramentos internos da câmera, adotamos aqui o modelo de câmera da função `glFrustum` que é mais geral que a `gluPerspective` que utilizamos quando apresentamos o algoritmo de Rastreamento de Raios. Nesta câmera mais geral o centro óptico não fica necessariamente no meio da janela do plano de projeção. O plano de projeção, entretanto, permanece ortogonal ao eixo *z*.

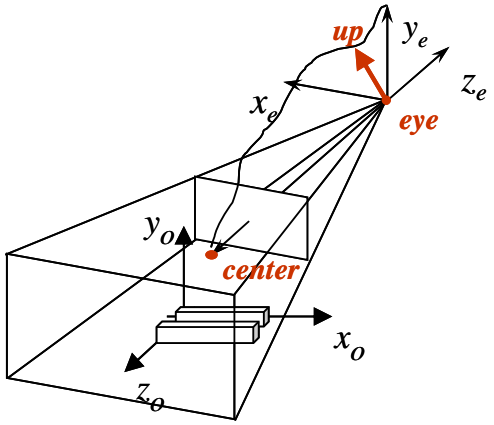
Apesar da generalidade dos parâmetros internos desta nova câmera, a posição padrão mostrada na Fig. 7.8(a) é pouco conveniente para a maioria das aplicações. Muitas aplicações exigem que a câmera possa ser posicionada em uma posição qualquer da cena como se fosse mais um objeto. Com esta flexibilidade o programador pode facilmente mudar a posição da câmera de um quadro para outro simulando, por exemplo, uma navegação na cena.

Para atender ao requisito de tratar a câmera como se fosse apenas mais um objeto, o OpenGL™ oferece uma função utilitária chamada `glLookAt` que permite que a câmera

possa ser instanciada na cena como se fosse mais um objeto. A função `glLookAt` que tem como parâmetros os vetores **eye, center** e **up,** destacados na Fig. 7.8(b). No capítulo de Rastreamento de Raios utilizamos estes mesmos parâmetros para facilitar a comparação entre estes dois enfoques e para escrevermos programas que possam alternar entre estes dois algoritmos.



(a) posição padrão da câmera mais geral do OpenGL™



(b) posição arbitrária definida através da função gluLookAt

Fig. 7.8 –Sistema de coordenadas da câmera

A partir dos vetores **eye, center** e **up** podemos computar os unitários na direção dos eixos da câmera, $\mathbf{x}_e\mathbf{y}_e\mathbf{z}_e$, através das seguintes relações[4]:

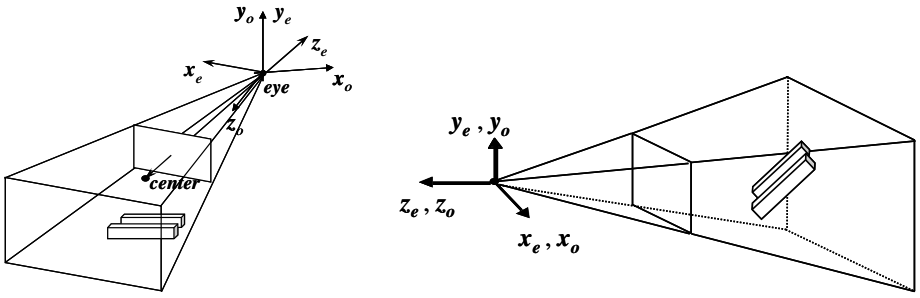
$$\mathbf{z}_e = \frac{1}{\|\mathbf{eye} - \mathbf{center}\|}(\mathbf{eye} - \mathbf{center}) \qquad (7.1.a)$$

[4] Estas equações são as mesmas derivadas no capítulo de Rastreamento de Raios. Foram repetidas aqui apenas para facilitar a leitura.

$$\mathbf{x}_e = \frac{1}{\|\mathbf{up}\times\mathbf{z}_e\|}(\mathbf{up}\times\mathbf{z}_e) \tag{7.1.b}$$

$$\mathbf{y}_e = \mathbf{z}_e \times \mathbf{x}_e \tag{7.1.c}$$

A transformação das coordenadas do mundo em coordenadas da câmera dos vértices das primitivas pode ser obtida compondo-se duas transformações: uma translação que leve o vetor **eye** para origem, seguida de uma rotação que alinhe os eixos da câmera com os eixos do mundo (Fig. 7.9).



(a) após a translação (b) após a rotação

Fig. 7.9 –Sistema de coordenadas da câmera

A matriz que representa esta composição é dada por:

$$\mathbf{L}_{at} = \mathbf{RT} = \begin{bmatrix} x_{ex} & x_{ey} & x_{ez} & 0 \\ y_{ex} & y_{ey} & y_{ez} & 0 \\ z_{ex} & z_{ey} & z_{ez} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.2}$$

onde eye_x, x_{ex}, y_{ex}, z_{ex} são as coordenadas x dos vetores ***eye***, $\boldsymbol{x}_e$, $\mathbf{y}_e$ e $\mathbf{z}_e$, respectivamente. A notação para as coordenadas de y e z segue a mesma regra. A translação nesta equação é trivial, mas a rotação é mais complexa e requer uma explicação.

Para entendermos a matriz de rotação da equação (7.2) devemos primeiro considerar a rotação inversa que leva os pontos que estão sobre os vetores da base $\boldsymbol{x}_o\mathbf{y}_o\mathbf{z}_o$ para a base $\boldsymbol{x}_e\mathbf{y}_e\mathbf{z}_e$. A matriz desta transformação pode ser facilmente obtida se lembrarmos que as colunas de uma matriz que representa uma transformação linear são as transformadas dos vetores da base. Ou seja, a primeira coluna desta matriz da transformada inversa contém as coordenadas de $\boldsymbol{x}_e$, a segunda de $\mathbf{y}_e$ e a terceira de $\mathbf{z}_e$. Como esta matriz é ortonormal, a sua inversa é a sua transposta, justificando a expressão que aparece na equação (7.2).

Resumindo a discussão desta seção, ao desenharmos uma objeto que tem uma matriz de instanciação $\mathbf{M}_{\text{obj}}$ os vértices passam pela transformação composta:

$$\mathbf{M}_{view} = \mathbf{L}_{at}\mathbf{M}_{obj} \tag{7.3}$$

Se na implementação do Sistema Gráfico acumularmos as matrizes multiplicando pela direita, como faz o OpenGL™, no programa de aplicação a matriz *model view* seria

definida primeiro como $\mathbf{L}_{at}$ através da `glLookAt` e depois, para cada objeto da cena, acumularíamos a matriz $\mathbf{M}_{\text{obj}}$. Caso a cena tenha vários objetos com matrizes de instanciação diferentes a implementação do OpenGL™ fornece o mecanismo de pilha para armazenar matrizes que serão posteriormente recuperadas, como discutimos no capítulo sobre Transformações Geométricas.

Iluminação dos vértices

Uma cena contém, além dos objetos e da câmera, as luzes e os modelos de material incluindo cor e texturas. As posições das luzes passam pelas mesmas transformações discutidas acima. São instanciadas na cena como se fossem objetos que não vemos diretamente. Só vemos os seus efeitos sobre os objetos geométricos. Ou seja, no OpenGL™ a instanciação de luzes não gera primitivas geométricas. Se quisermos vê-las na cena temos que criar uma forma geométrica para elas.

Após a transformação *modelview* que coloca toda a cena num mesmo sistema de coordenadas, é conveniente calcularmos logo as componentes difusa e especular da reflexão das luzes nos vértices. Os cálculos da iluminação dos vértices segue o modelo de iluminação local apresentado no algoritmo de Rastreamento de Raios (sem os efeitos de sombra, transparência e reflexão especular). A cor de cada vértice pode então ser calculada por:

$$\begin{pmatrix} I_r \\ I_g \\ I_b \end{pmatrix} = \sum_{luzes} \left(\begin{pmatrix} l_{ar} \\ l_{ag} \\ l_{ab} \end{pmatrix} \otimes \begin{pmatrix} k_{ar} \\ k_{ag} \\ k_{ab} \end{pmatrix} + \begin{pmatrix} l_r \\ l_g \\ l_b \end{pmatrix} \otimes \begin{pmatrix} k_{dr} \\ k_{dg} \\ k_{db} \end{pmatrix} \left(\hat{\mathbf{n}} \cdot \hat{\mathbf{L}}\right) + \begin{pmatrix} l_r \\ l_g \\ l_b \end{pmatrix} \otimes \begin{pmatrix} k_{sr} \\ k_{sg} \\ k_{sb} \end{pmatrix} \left(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}\right)^n \right) \tag{7.4}$$

onde $(l_{ar},l_{ag},l_{ab})^{\mathrm{T}}$ são as intensidades RGB da luz ambiente, $(k_{ar}, k_{ag}, k_{ab})^{\mathrm{T}}$ é a cor ambiente do material, $(l_r,l_g,l_b)^{\mathrm{T}}$ são as intensidades RGB da luz, $(k_{dr}, k_{dg}, k_{db})^{\mathrm{T}}$ é cor difusa do material e $(k_{sr}, k_{sg}, k_{sb})^{\mathrm{T}}$ e n são a cor e o coeficiente especular, como discutimos anteriormente . A Fig. 7.10 ilustra os vetores $\hat{\mathbf{n}},\hat{\mathbf{r}},\hat{\mathbf{v}}$ e $\hat{\mathbf{L}}$.

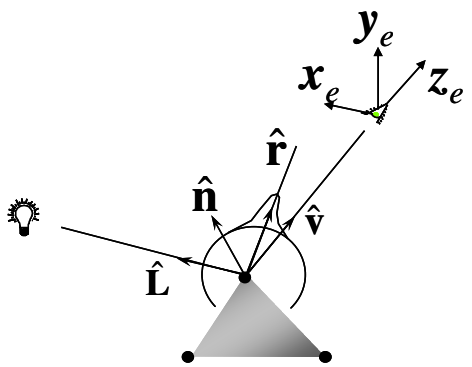


Fig. 7.10 – Iluminação dos vértices

O OpenGL™ modifica a equação (7.4) para levar em conta outros efeitos como luzes direcionais, atenuação para simular neblina (*fog*) e textura. As duas primeiras modificações

são variações na fórmula e a textura, já foi introduzida no capítulo sobre Rastreamento de Raios. Optamos por omitir este discussão para não estender demasiadamente este capítulo.

Transformação de projeção

A posição padrão da câmera ilustrada na Fig. 7.8(a) facilita o cálculo da projeção cônica. Para calcularmos a projeção de um ponto no plano *near*, como ilustra a Fig. 7.11, basta escalarmos as coordenadas do ponto por um fator que reduzisse a coordenada z para a posição $-n$. O sinal negativo é necessário para transformar uma distância positiva em uma coordenada negativa (a câmera só vê os pontos com coordenadas z negativas). Ou seja, a projeção de um ponto **p** pode ser escrita como:

$$\mathbf{p}_p = \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = \frac{n}{-z_e} \begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix} \tag{7.5}$$

Esta equação pode também ser deduzida através da semelhança de triângulos também ilustrada na Fig. 7.11.

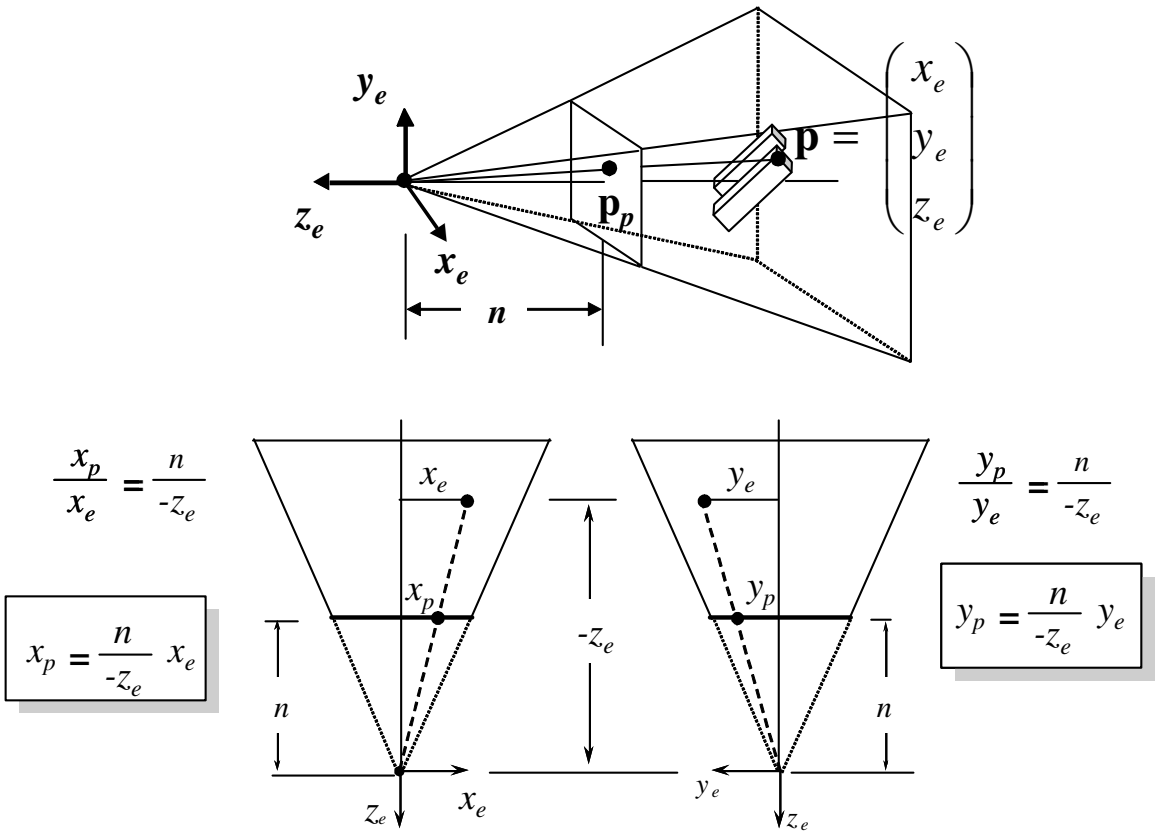


Fig. 7.11 –Projeção cônica simples

Esta projeção pode ser escrita como sendo uma transformação no espaço homogêneo definida pela uma matriz 4×4:

$$\begin{bmatrix} wx_p \\ wy_p \\ wz_p \\ w \end{bmatrix} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} nx_e \\ ny_e \\ nz_e \\ -z_e \end{bmatrix} \rightarrow \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = \frac{1}{-z_e} \begin{pmatrix} nx_e \\ ny_e \\ nz_e \end{pmatrix} \tag{7.6}$$

Ocorre, entretanto, que se projetarmos os vértices das primitivas gráficas perdemos a informação sobre sua profundidade e não podemos decidir mais que primitiva é visível e qual está oculta por outra.

Para preservar a profundidade no processo de projeção o algoritmo de mapa de profundidades implementado no OpenGL™ realiza uma transformação homogênea que distorce o mundo da forma ilustrada na Fig. 7.12. Nesta transformação o centro de projeção, **eye**, vai para o infinito e a projeção cônica se transforma numa projeção paralela ortográfica onde os raios projetores se transformam em retas perpendiculares ao plano de projeção. As coordenadas dos vértices paralelas ao plano de projeção, ***x*** e ***y***, ficam com os seus valores corretos projetados, uma vez que a projeção ortográfica não vai mais alterá-los, e as profundidades relativas, coordenadas ***z***, são preservada.

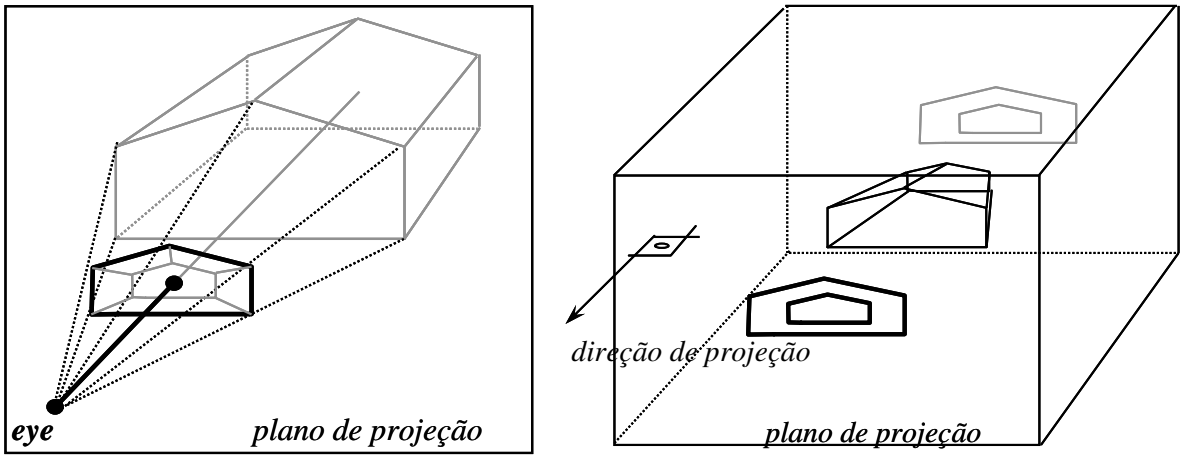
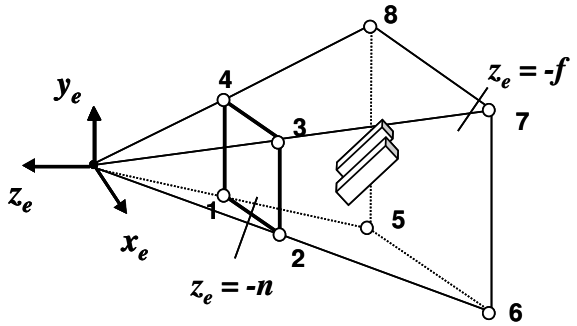
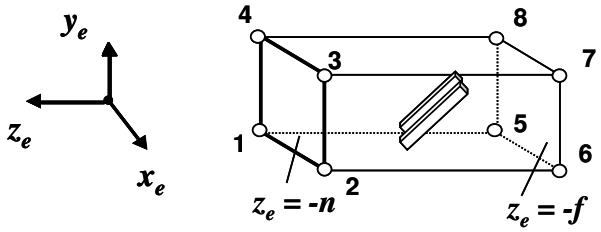


Fig. 7.12 –Simplificação da projeção cônica

Para derivarmos a transformação que faz a simplificação mostrada na Fig. 7.12 vamos primeiramente calcular a matriz que transforma o tronco de pirâmide de visão num paralelepípedo como ilustra a Fig. 7.13.



(a) pirâmide antes da transformação



(b) paralepípedo depois da transformação

Fig. 7.13 –Antes e depois da transformação projetiva desejada

Podemos determinar a matriz desta transformação partindo de uma matriz genérica de coeficientes m_{ij} desconhecidos. A transformação em coordenadas homogêneas pode ser escrita por:

$$\begin{bmatrix} w_i x_i' \\ w_i y_i' \\ w_i z_i' \\ w_i \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \tag{7.7}$$

onde $\begin{pmatrix} x_i & y_i & z_i \end{pmatrix}^T$ são as coordenadas cartesianas de um ponto i qualquer antes da transformação e $\begin{pmatrix} x_i' & y_i' & z_i' \end{pmatrix}^T$ são as coordenadas do mesmo ponto depois da transformação. Devemos notar que a coordenada w pode ser diferente de ponto para ponto. Assim não temos apenas as 14 incógnitas m_{ij} para resolver, cada ponto que utilizamos cria uma incógnita a mais.

Poderíamos determinar os m_{ij} colocando as condições de que os pontos 1,2,3,..8 sejam transformados da forma indicada na Fig. 7.13. Ocorre, entretanto, que podemos adotar um procedimento mais simples utilizando condições geométricas mais convenientes. Considere, por exemplo, que a origem, ***eye***, deve ir para o infinito na direção positiva de z. Isto se escreve em coordenadas homogêneas como:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ \alpha \\ 0 \end{bmatrix} \tag{7.8}$$

onde α é um fator positivo desconhecido. Aplicando esta condição na equação (7.7) temos:

$$\begin{bmatrix} 0 \\ 0 \\ \alpha \\ 0 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{03} \\ m_{13} \\ m_{23} \\ m_{33} \end{bmatrix} \tag{7.9}$$

Com isto determinamos a última coluna a menos do fator α. Ou seja, se **H** é a matriz procurada, temos:

$$\mathbf{H} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & \alpha \\ m_{30} & m_{31} & m_{32} & 0 \end{bmatrix} \tag{7.10}$$

Outra condição forte sobre a matriz de projeção **H** é que todos os pontos do plano de projeção devem permanecer com suas posições inalteradas. Em coordenadas homogêneas isto se escreve como sendo:

$$\begin{bmatrix} x \\ y \\ -n \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} \beta x \\ \beta y \\ -\beta n \\ \beta \end{bmatrix} \quad \forall x, y \in R \tag{7.11}$$

onde β é um segundo fator desconhecido. Esta condição aplicada a matriz homogênea resulta em:

$$\begin{bmatrix} \beta x \\ \beta y \\ -\beta n \\ \beta \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & \alpha \\ m_{30} & m_{31} & m_{32} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00}x + m_{01}y - m_{02}n \\ m_{10}x + m_{11}y - m_{12}n \\ m_{20}x + m_{21}y - m_{22}n + \alpha \\ m_{30}x + m_{31}y - m_{32}n \end{bmatrix} \quad \forall x, y \in R \tag{7.12}$$

Esta condição é forte porque ela é válida para todo x e y, e as quatro igualdades mostradas na equação (7.12) são na realidade identidades de polinômios. Isto é, todos os termos têm que ter os mesmos coeficientes, reduzindo a matriz **P** a:

$$\mathbf{H} = \begin{bmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \beta + \dfrac{\alpha}{n} & \alpha \\ 0 & 0 & -\dfrac{\beta}{n} & 0 \end{bmatrix} \tag{7.13}$$

A última condição para determinar esta matriz pode ser obtida observando-se que os pontos do plano *far* permanecem na mesma profundidade com as coordenadas x e y escaladas de um fator n/f, como ilustra a Fig. 7.13. Em coordenadas homogêneas isto se escreve como:

$$\begin{bmatrix} x \\ y \\ -f \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} \gamma\frac{n}{f}x \\ \gamma\frac{n}{f}y \\ -\gamma f \\ \gamma \end{bmatrix} \quad \forall x, y \in R \tag{7.14}$$

onde γ é o terceiro fator desconhecido. Esta condição aplicada a transformação procurada coma matriz de (7.13) resulta em:

$$\begin{bmatrix} \gamma\frac{n}{f}x \\ \gamma\frac{n}{f}y \\ -\gamma f \\ \gamma \end{bmatrix} = \begin{bmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \beta+\frac{\alpha}{n} & \alpha \\ 0 & 0 & -\frac{\beta}{n} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} \beta x \\ \beta y \\ -\beta f - \frac{\alpha f}{n} + \alpha \\ \beta\frac{f}{n} \end{bmatrix} \quad \forall x, y \in R \tag{7.15}$$

Novamente temos aqui quatro identidades de polinômios, uma em cada uma das linhas, resultando em:

$$\beta = \gamma\frac{n}{f} \tag{7.16a}$$

$$-\gamma f = \alpha\left(1-\frac{f}{n}\right) - \beta f \quad \Rightarrow \alpha = \gamma n \tag{7.16b}$$

Substituindo os valores de α e β na matriz (7.13) chegamos a:

$$\mathbf{H} = \begin{bmatrix} \gamma\frac{n}{f} & 0 & 0 & 0 \\ 0 & \gamma\frac{n}{f} & 0 & 0 \\ 0 & 0 & \gamma\frac{n}{f}+\gamma & \gamma n \\ 0 & 0 & -\gamma\frac{1}{f} & 0 \end{bmatrix} \tag{7.17}$$

Com isto determinamos a transformação desejada, uma vez que matrizes que, se uma matriz representa uma transformação homogênea, qualquer produto dela por um escalar representa a mesma transformação. Basta observar que o escalar vai multiplicar tanto as coordenadas x_h y_h z_h quanto a coordenada w. Como a coordenada cartesiana é resultante da divisão de ambas o resultado não se altera se elas estão multiplicadas pelo mesmo fator.

Para simplificar podemos tomar $\gamma=f$ de forma a eliminar os denominadores, resultando na matriz:

$$\mathbf{H} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & nf \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.18}$$

que se parece bastante com a matriz da equação (7.6) a menos da coordenada z que agora não é mais "achatada" no plano de projeção. Aliás, uma maneira menos rigorosa, comumente encontrada na literatura para derivar a matriz **P**, consiste em partir de uma forma relaxada da matriz da equação (7.6) onde a profundidade z é definida por dois coeficientes desconhecidos a e b:

$$\mathbf{H} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.19}$$

Esta mudança estabelece que a profundidade z passe a ser calculada por:

$$z' = -a - \frac{b}{z} \tag{7.20}$$

Se aplicarmos nesta matriz a condição de que os pontos do plano *near* e *far* não saiam destes planos teremos duas condições:

$$-n = -a + \frac{b}{n} \tag{7.21a}$$

$$-f = -a + \frac{b}{f} \tag{7.21b}$$

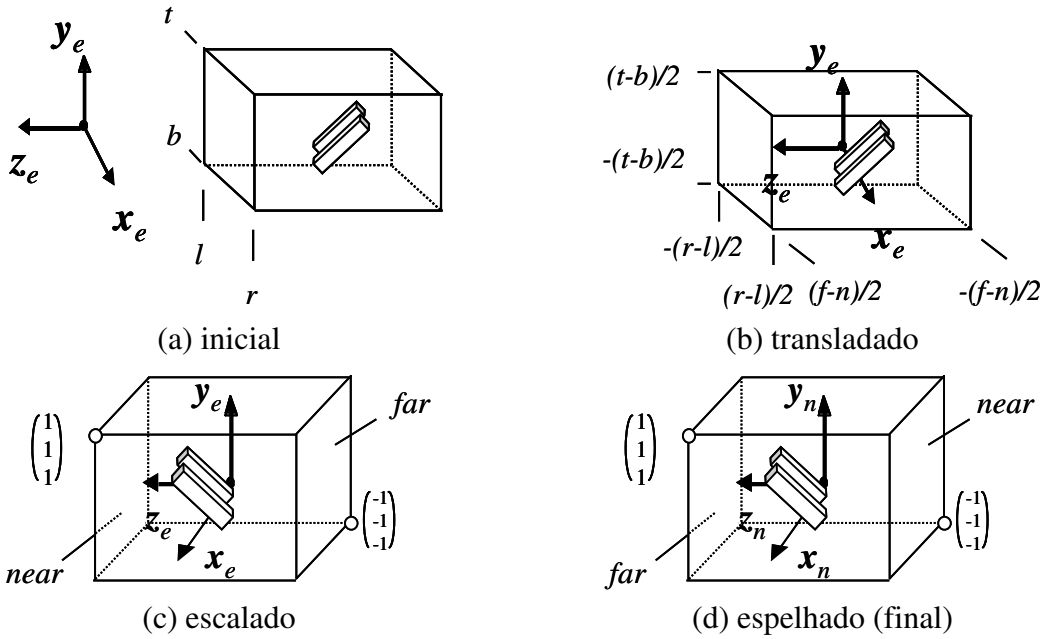
Resolvendo estas duas equações temos:

$$\begin{cases} a = f + n \\ b = f \cdot n \end{cases} \tag{7.22}$$

O que novamente resulta na matriz dada em (7.18).

Para tornar os próximos passos mais simples o *rendering pipeline* do OpenGL transforma o paralelepípedo de visão, indicado na Fig. 7.13, no cubo normalizado definido por [-1,1]×[-1,1] ×[-1,1] como ilustra a Fig. 7.14(d).

A matriz que realiza esta normalização do paralelepípedo de visão pode ser deduzida através da composição de três outras da forma ilustradas na Fig. 7.14. A primeira consiste em transladar o centro do paralelepípedo de visão para a origem (Fig. 7.14(a) para Fig. 7.14(b)). A segunda escala o cubo de forma a torná-lo unitário (Fig. 7.14(b) para Fig. 7.14(c)). Finalmente, a terceira espelha o cubo de forma que o plano *near* passe a ter o menor valor da coordenada z (Fig. 7.14(c) para Fig. 7.14(d)). A intenção deste espelhamento é que as profundidades estejam na mesma ordem que a coordenada z. Ou seja, que menores z's representem pontos mais próximos.



(a) inicial (b) transladado

(c) escalado (d) espelhado (final)

Fig. 7.14 –Ajuste do paralelepípedo de visão para o cubo [-1,1]×[-1,1] ×[-1,1]

A matriz que representa a normalização do paralelepípedo de visão pode então ser escrita pela seguinte composição:

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2/(r-l) & 0 & 0 & 0 \\ 0 & 2/(t-b) & 0 & 0 \\ 0 & 0 & 2/(f-n) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -(r+l)/2 \\ 0 & 1 & 0 & -(t+b)/2 \\ 0 & 0 & 1 & (f+n)/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.23)$$

ou:

$$\mathbf{N} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.24)$$

Se combinarmos esta matriz com a matriz da transformação projetiva **H** (eq. (7.16)) temos a matriz de projeção **P** que transforma as coordenadas de um ponto do sistema da câmera para este espaço normalizado. A expressão desta matriz é então:

$$\mathbf{P} = \mathbf{NH} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.25}$$

Esta matriz é a mesma que a especificação do OpenGL™ apresenta como sendo a matriz correspondente a função `glFrustum`. Ou seja, ela define a transformação de projeção que leva as coordenadas de um ponto do sistema da câmera apresentada na Fig. 7.9(a) para o sistema de coordenadas normalizadas ilustrado na Fig. 7.14(d).

A Fig. 7.15 resume os sistemas coordenados e as transformações que deduzimos até aqui. Como dito na seção anterior, no OpenGL™ a primeira matriz de transformação é a *model view* que compõe as transformações de instanciação dos objetos, $\mathbf{M}_{obj}$, com a matriz de instanciação da câmera na cena, $\mathbf{L}_{at}$.

Um ponto importante que devemos destacar aqui é que apesar da derivação destas transformações ser complexa, a implementação computacional delas é simples e direta, justificando o esforço de entendermos toda esta álgebra. Outro ponto importante a destacar é que a transformação de projeção modifica os ângulos entre direções que existem no espaço da câmera. Daí a necessidade de fazermos os cálculos do modelo de iluminação no espaço da câmera, antes da deformação que muda os ângulos da cena.

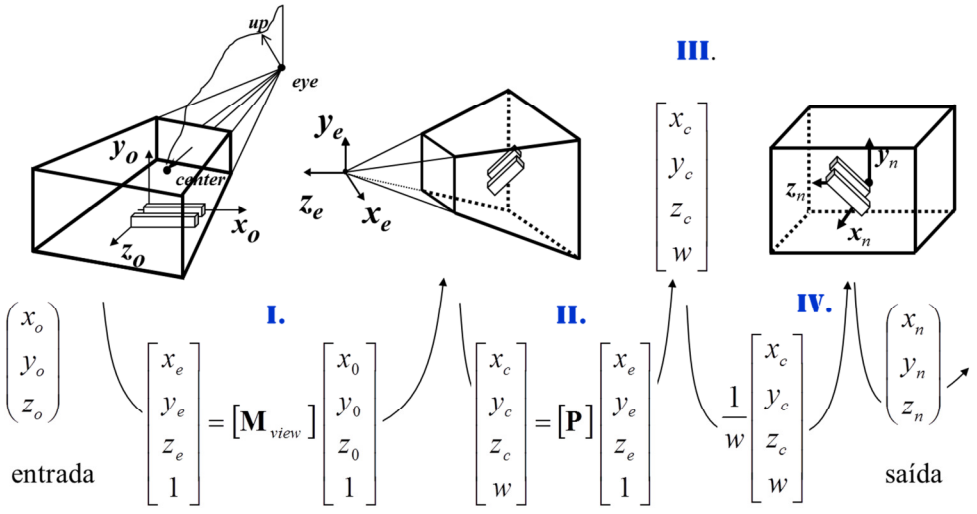


Fig. 7.15 – Resumo dos passos de geometria até o momento

Projeção ortogonal

A partir das equações apresentadas nesta seção podemos também derivar a matriz que o OpenGL™ utiliza para projeção paralelas definidas pelas funções `glOrtho` e `glOrtho2D`. Nestas funções a câmera segue um modelo ilustrado na Fig. 7.16, onde os

raios projetores não convergem para o **eye** mas são paralelos ao eixo *z*. Como estes raios são ortogonais ao plano de projeção, *z=near*, este tipo de projeção é chamado de projeção ortográfica.

Um ponto importante a destacar é que o paralelepípedo da Fig. 7.16 é exatamente o mesmo paralelepípedo que resulta do prisma da câmera `glFrustum` depois que ele sofre a transformação **H**, como ilustra a Fig. 7.13. A matriz de projeção paralela do OpenGL™ é matriz que leva deste prisma para o cubo no espaço normalizado. Ou seja, ela é simplesmente a matriz **N** derivada acima e mostrada na equação (7.24). A especificação do OpenGL™ apresenta esta matriz sendo a matriz corresponde às funções `glOrtho` e `glOrtho2D`, como seria de esperarmos.

A projeção paralela ortográfica não tem o realismo da projeção cônica mas é muito útil nas Engenharias e Arquitetura porque elas preservam o paralelismo de linhas e permitem a definição de escala. Com o desenho em escala podemos fazer medidas dos tamanhos das peças diretamente sobre sua projeção apresentada numa planta.

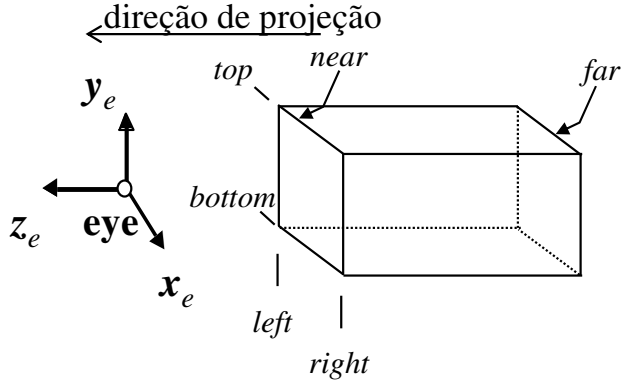


Fig. 7.16 – Modelo de câmera das funções `glOrtho` e `glOrtho2D` do OpenGL™

O material apresentado nesta seção pode ainda ser útil se desejarmos deduzir outros modelos de câmera que não os dois apresentados acima. A Fig. 7.15 ilustra esquematicamente uma estação de Realidade Virtual composta de diversas telas. O modelo de câmera para cada uma das telas deve acompanhar a posição da cabeça do observador. Quando ele está olhando na direção de uma das câmeras, as demais ficam inclinadas com relação ao eixo ótico que vai do seu olho na direção para frente. Para renderizar corretamente um cenário virtual nestas telas é necessário calcularmos uma matriz que faça corretamente esta projeção. Para isto precisamos acrescentar no processo descrito nesta seção mais alguns passos na derivação da matriz **P** (exercício 7.1). O OpenGL™ permite que o programador forneça diretamente para ele tanto a matriz de projeção, quanto a matriz de instanciação (*model view*) através da função `glLoadMatrix`. Ou seja, podemos escrever as matrizes que quisermos e pedir para o OpenGL™ para que as coordenadas dos vértices sejam transformadas por elas e não por outras internas ao sistema.

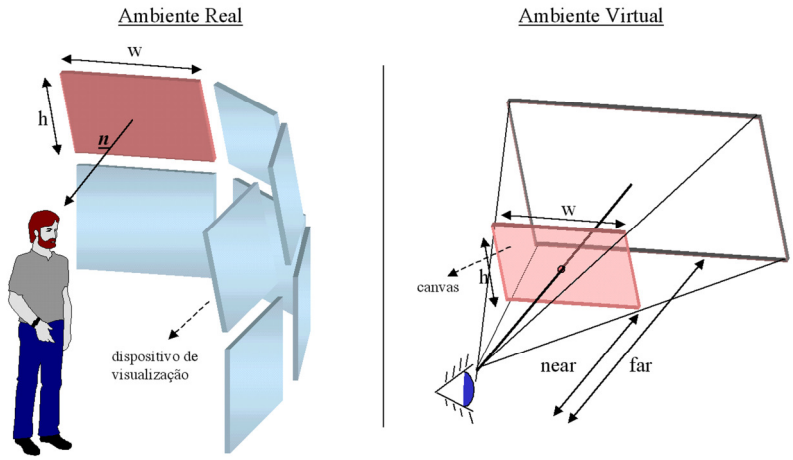


Fig. 7.17 – Requisitos de uma câmera de uma estação de Realidade Virtual (adaptado da dissertação de Alexandre G. Ferreira)

Recorte

O próximo passo no *rendering pipeline* é o recorte (*clipping*) das primitivas para desenharmos apenas a parte visível delas. Esta parte é dada pela interseção das primitivas com o tronco de pirâmide de visão (ou com o paralelepípedo, no caso da projeção paralela).

Para ilustrar os algoritmos envolvidos no recorte de primitivas, vamos considerar quatro problemas diferentes de recorte mostrados na Fig. 7.16: (a) segmentos de reta, (b) polígonos quaisquer, (c) triângulos e (d) malhas de triângulos. Cada um destes problemas tem uma certa particularidade. O recorte de polígonos não convexos contra uma janela convexa pode resultar em mais de uma região, como ilustra a Fig. 7.16(b). O recorte de triângulos num sistema, como o OpenGL™ que é otimizado para triângulos, não deve gerar um polígono qualquer mas sim vários sub-triângulos. Quando temos uma malha de triângulos (Fig. 7.16(d)) o resultado do recorte deve ser uma nova malha com os novos vértices compartilhados, de forma a evitar a duplicação cálculos relativos a eles.

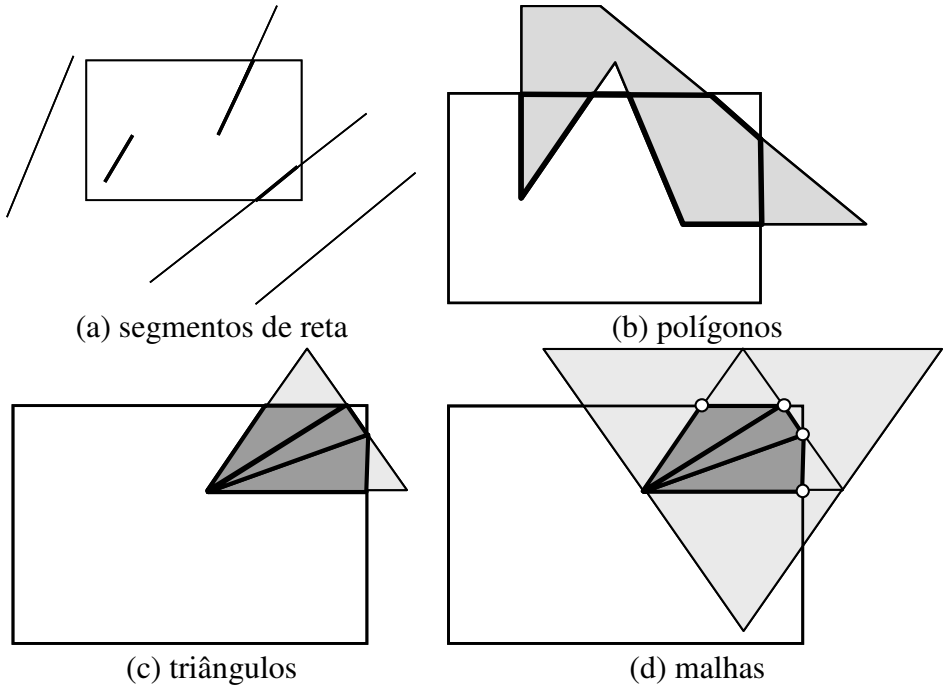


Fig. 7.18 - Diferentes problemas de recorte

Para facilitar a compreensão dos algoritmos descritos aqui as áreas de recorte mostradas na Fig. 7.18 são retangulares. Elas correspondem a parte visível do desenho através uma janela no R^2. Na realidade temos que recortar as primitivas no R^3. Os volume de recorte candidatos são ou o troco de pirâmide das coordenadas da câmera ou é o cubo das coordenadas normalizadas. É uma questão de decidirmos quando nas transformações ilustradas na Fig. 7.15 queremos fazer o recorte.

Antes de escolhermos o volume de recorte, devemos notar que de qualquer forma todas as nossas opções são regiões convexas limitadas por hiperplanos. Mais ainda estes hiperplanos têm equação da forma $ax+by+c=0$, $ax+by+cz+d=0$, ou $ax+by+cz+dw=0$, dependendo se estamos tratando do problema do R^2, no R^3 ou no PR^3, respectivamente.

De uma forma geral podemos definir os hiperplanos que formam as fronteiras das regiões de recorte através da equação $\mathbf{n}\cdot\mathbf{p} = 0$. Onde $\mathbf{n}=(a,b,c,d)^T$ e $\mathbf{p}=(x,y,z,w)^T$. A Fig. 7.17 mostra a condição de pertinência de um ponto genérico a uma região de recorte convexa a partir da avaliação deste ponto na equação dos seus hiperplanos. Se orientarmos as normais dos hiperplanos para fora da região, os produtos internos homogêneos dos pontos interiores com estas normais sempre resultam negativos[5].

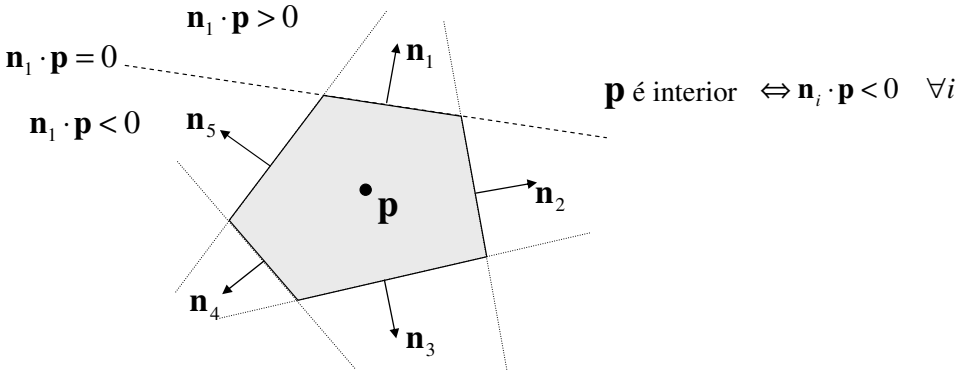


Fig. 7.19 – Condição algébrica de um ponto pertencer a uma região convexa

Um outro aspecto comum aos problemas de recorte diz respeito ao ponto em que um segmento de reta ou uma aresta de um polígono, $\mathbf{p}_1\mathbf{p}_2$, intercepta um hiperplano que é uma fronteira da região de recorte cuja normal é $\mathbf{n}$. O produto interno homogêneo $d_i=\mathbf{n}\cdot\mathbf{p}_i$ diz se o ponto i (1 ou 2) está fora ou dentro da região de recorte. Mais ainda, o valore de d_i é uma medida de distância com sinal do ponto i ao plano. Se a normal for unitária esta distância é a distância cartesiana convencional e os valores positivos medem distância dos pontos exteriores e os negativos interiores. Se a normal não for unitária estes valores são escalados pelo inverso de sua norma.

Se o produto d_1d_2 for maior que zero os dois pontos estão do mesmo lado, logo o segmento $\mathbf{p}_1\mathbf{p}_2$ não intercepta o hiperplano. Se o produto d_1d_2 for menor que zero, o segmento intercepta o hiperplano e o ponto de interseção pode ser calculado por (ver Fig. 7.20):

$$\mathbf{p}_i = \frac{d_1}{d_1 - d_2}\mathbf{p}_2 - \frac{d_2}{d_1 - d_2}\mathbf{p}_1 \qquad (7.26)$$

Devemos notar que esta equação vale tanto para $d_1>0$ e $d_2<0$ quanto para vice-versa. Ela pode ser deduzida por semelhança de triângulos analisando-se estas duas situações. Uma curiosidade algébrica é que o denominador é sempre a soma dos módulos de d_1 e d_2 com um sinal adequado para a equação (7.26) calcular o ponto de interseção. Deixamos a prova disto a cargo do leitor.

[5] O produto interno homogêneo aqui é: $\mathbf{n}\cdot\mathbf{p}=ax+by+cz+dw$ mesmo que w seja igual a 1 e/ou z seja igual a zero.

De posse de um processo algébrico para classificar um ponto e outro para calcular a interseção de um segmento de reta com relação a um hiperplano qualquer podemos considerar estes passos como básicos e descrever alguns algoritmos para recorte de segmentos de reta e de polígonos.

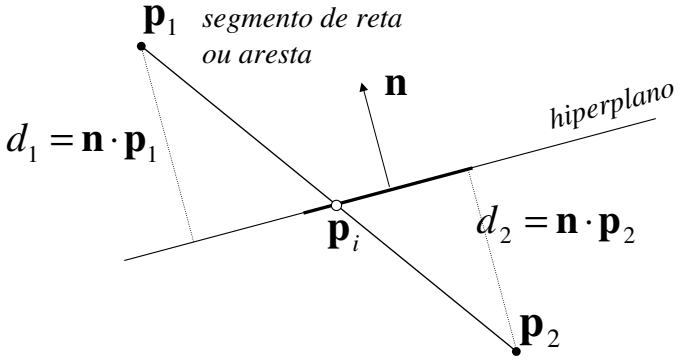


Fig. 7.20 - Interseção de aresta × hiperplano

Recorte de segmento de reta

Provavelmente um dos mais difundidos e utilizados algoritmos de recorte de retas é o algoritmo de Dan Cohen e Ivan Sutherland que remontam ao início da Computação Gráfica no MIT na década de 60.

A idéia geral do algoritmo consiste em classificar os dois vértices do segmento de reta com relação a cada um dos hiperplanos da fronteira de recorte. Se os dois vértices estiverem no lado positivo (externo) de qualquer um dos hiperplanos, então o segmento de reta pode ser descartado, uma vez que ele está fora da região de interesse. Por outro lado, se os dois vértices estivem dentro (negativo ou zero) de todos os hiperplanos, então o segmento não precisa ser recortado e pode seguir no *rendering pipeline*. Estes dois casos, ditos triviais, são os pontos de parada do algoritmo e estão ilustrados pelos segmentos A e B, respectivamente, da Fig. 7.19.

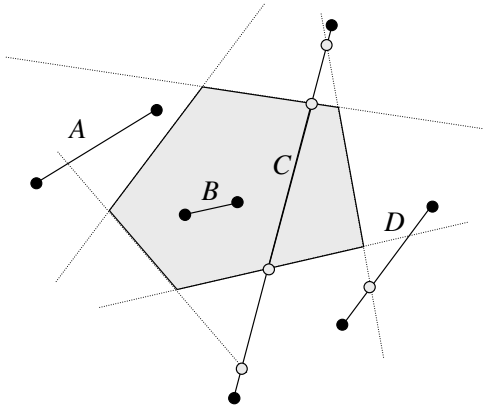


Fig. 7.21 - Recorte de segmentos com base no algoritmo Cohen-Sutherland

Quando os vértices do segmento não se classificam em nenhum dos dois casos triviais a idéia o algoritmo é ir recortando o segmento jogando fora as extremidades que ficam fora dos hiperplanos. Ou seja, a cada passo o algoritmo escolhe um hiperplano em que um dos vértices esteja classificado como fora, calcula a interseção do segmento com este hiperplano, descarta a parte do segmento que fica do lado positivo e recalcula a posição do novo vértice com relação a todos os hiperplanos. Os segmentos C e D da Fig. 7.19 ilustram as duas evoluções possíveis. O segmento C vai sendo recortado até ficar todo dentro região de recorte finalizando no caso trivial semelhante ao segmento B. O segmento D, por sua vez, quando recortado fica todo do lado positivo de dos hiperplanos.

Cyrus e Beck propuseram em 1978 uma estratégia diferente para o recorte de linhas que resulta geralmente num algoritmo mais eficiente. Esta proposta se baseia na equação paramétrica do segmento com dois valores do parâmetro t_e e t_s que indicam o ponto de entrada e saída do segmento na região. Inicialmente estes dois parâmetros são iguais a zero e um, respectivamente, correspondendo ao segmento que vai do vértice estabelecido como inicial ao final (existe uma orientação implícita na parametrização t).

Para cada um dos hiperplanos o algoritmo de Cyrus–Beck calcula o valor de t_i correspondente a interseção do segmento com ele. Cada hiperplano é classificado como sendo uma possível fronteira de entrada, ou de saída, do segmento na região de recorte em função das orientações do segmento e da normal. Ou seja, se o produto interno do vetor que vai do ponto inicial ao final do segmento com a normal do hiperplano indicar que eles formam um ângulo maior que 90^o, este hiperplano é uma fronteira de entrada do segmento na região de recorte. Sendo assim o valor de t_i calculado é candidato a substituir o valor de t_e desde que seja maior que o valor atual dele. Caso a orientação relativa do segmento com a normal seja ao contrário o valor calculado de t_i é um candidato a substituir o valor de t_s desde ele seja menor que o valor atual dele.

O algoritmo termina quando todos os hiperplanos que fazem a fronteira da região de recorte são visitados, ou quando ao atualizar t temos um valor de $t_s<t_e$. No primeiro caso enviamos para a próxima etapa do *rendering pipeline* o segmento que vai da posição correspondente a t_e até a posição de t_s. No segundo caso o segmento é descartado, pois não intercepta a região de recorte. A Fig. 7.20 ilustra o caso de três segmentos A, B e C sendo analisados contra as fronteiras de uma região de recorte retangular[6].

[6] O algoritmo de Cyrus-Beck trata uma região de recorte convexa qualquer. Ou seja, a figura é só um exemplo. Já o algoritmo de Cohen-Sutherland foi originalmente proposto para regiões de recorte retangulares (janelas) mas pode, como foi feito aqui, ser estendido para tratar de uma região convexa qualquer.

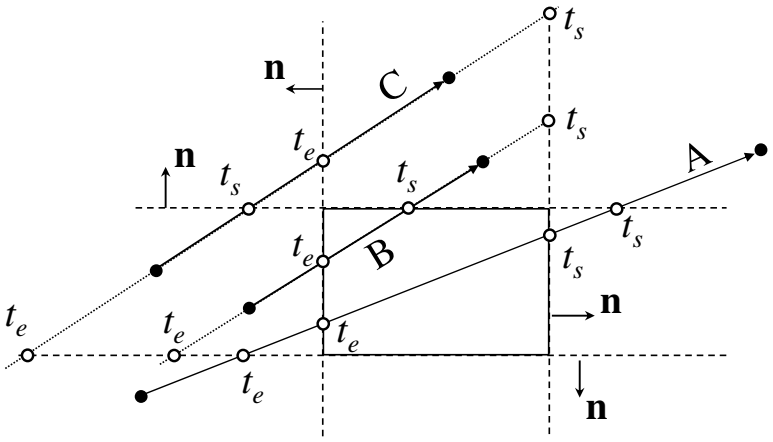


Fig. 7.22 – Recorte de segmentos com base no algoritmo de Cyrus-Beck

Recorte de polígonos

O algoritmo de recorte de polígonos de Sutherland e Hodgman (1974) contorna a dificuldade de partirmos um polígono não convexo contra uma fronteira convexa de *n* lados em *n* problemas mais simples de recortar um polígono contra um hiperplano infinito como ilustra a Fig. 7.21.

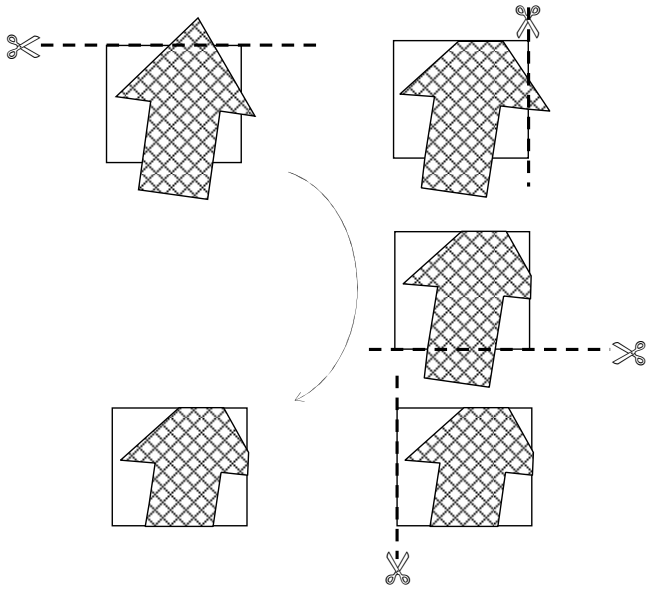
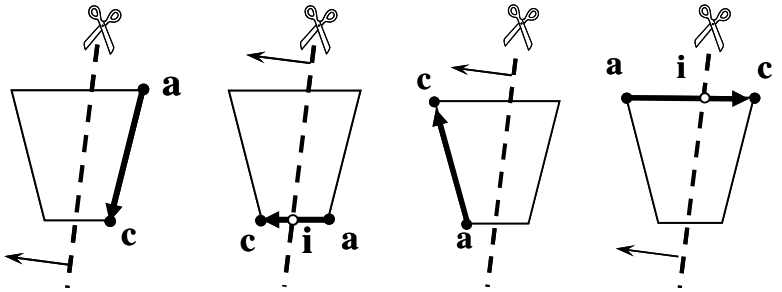


Fig. 7.23 - Recorte de polígonos com base no algoritmo de Hodgman-Sutherland

O algoritmo de recorte de um polígono qualquer contra um hiperplano infinito é razoavelmente simples. Ele percorre a lista de vértices do polígono classificando-os em interno ou externo a aquela fronteira. A partir do segundo vértice, o segmento que vai do vértice anterior ao vértice corrente é analisado para saber se ele: (a) é interno, (b) está

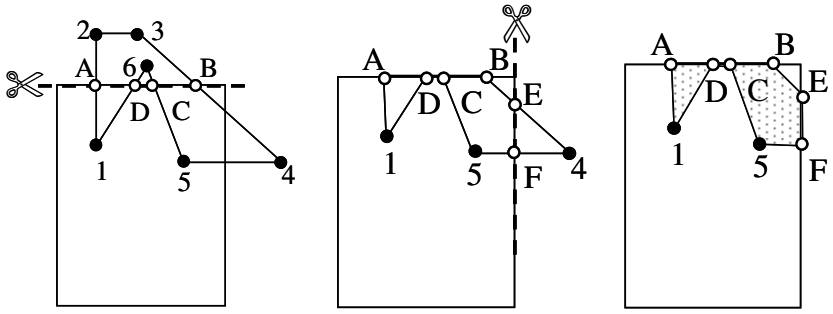
saindo, (c) é externo, ou (d) está entrando. Estas as quatro situações possíveis são caracterizadas pela classificação dos vértices anterior e corrente e estão ilustradas na Fig. 7.22. Nesta figura **a** é o ponto anterior, **c** o ponto corrente, **i** o ponto de interseção e o hiperplano de recorte está indicado com a tesoura com a normal para fora. Imaginando o algoritmo como um filtro com entrada **c**, a Fig. 7.22 mostra também as ações tomadas para cada um dos quatro casos.



Saída: **c**	Saída: **i**	nada	Saída: **i, c**
(a) interno	(b) saindo	(c) externo	(d) entrando

Fig. 7.24 - Classificação de um segmento no algoritmo de Hodgman-Sutherland

A Fig. 2.25 mostra um exemplo dos passos do algoritmo Hodgman-Sutherland aplicado aos dois hiperplanos que recortam um polígono qualquer. A figura mostra também duas tabelas que exemplificam as análises de classificação ilustradas na Fig. 7.24. Ao final do recorte no hiperplano superior o polígono 1,2,3,4,5,6 é transformado em A,B,4,5,C,D,1. Devemos notar que, para esta fronteira o ponto 4 é interior. Depois do recorte no hiperplano da direita o polígono passa a ser B,E,F,5,C,D,1,A. Hodgman e Sutherland apresentaram também uma implementação de uma função em C re-entrante que implementa este algoritmo que se tornou bastante difundida.



a	**c**	saída
1	2	A
2	3	
3	4	B e 4
4	5	5
5	6	C
6	1	D e 1

a	**c**	saída
A	B	B
B	4	E
4	5	F e 5
5	C	C
C	D	D
D	1	1
1	A	A

Fig. 7.25 - Exemplo do algoritmo de Hodgman-Sutherland

Recorte em coordenadas homogêneas

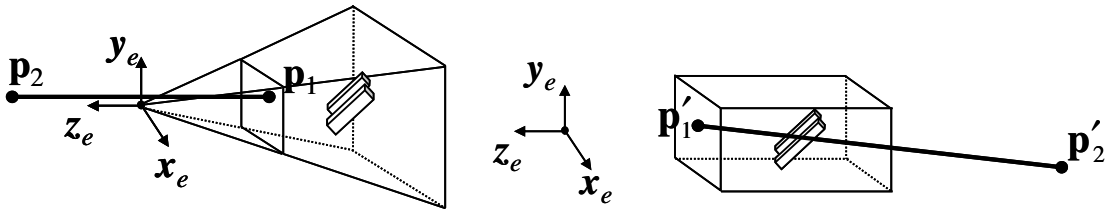
Vamos agora retornar a questão da escolha da região de recorte. Se examinarmos as transformações iniciais do *rendering pipeline* vamos encontrar diversas opções para realizarmos o recorte. Na Fig. 7.13 vemos quatro possíveis escolhas: o tronco de pirâmide genérico em coordenadas dos objetos, o tronco de pirâmide do sistema de coordenadas da câmera, um cubo no espaço homogêneo de dimensão quatro, PR^3 que resulta da transformação projetiva, antes da divisão por w e, finalmente, o cubo do espaço normalizado. Este último seria uma escolha mais natural uma vez que nele os hiperplanos são dados por: $x = \pm 1$, $y = \pm 1$ e $z = \pm 1$, o que simplificaria bastante as expressões do algoritmos de recorte.

Ocorre, entretanto, que a transformação projetiva **H** mostrada na seção anterior leva pontos que não estavam no tronco de pirâmide de visão para o espaço normalizado [-1,1]×[-1,1] ×[-1,1]. A Fig. 7.24 mostra um exemplo onde a transformação **H** leva um segmento de reta que não deveria ser visível para uma posição intercepta o volume de visão normalizado. Nesta figura o ponto $\mathbf{p}_1$ tem coordenadas cartesianas $(1,1,-n)^T$ e o ponto $\mathbf{p}_2=(1,1,n)^T$. A transformação destes pontos pode ser calculada por:

$$\mathbf{p}_1' = \mathbf{H}\mathbf{p}_1 = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & nf \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} n \\ n \\ -n^2 \\ n \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -n \\ 1 \end{bmatrix} \tag{7.27a}$$

$$\mathbf{p}_2' = \mathbf{H}\mathbf{p}_2 = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & nf \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ n \\ 1 \end{bmatrix} = \begin{bmatrix} n \\ n \\ n^2+2nf \\ -n \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -n-2f \\ 1 \end{bmatrix} \tag{7.27b}$$

Ou seja, a divisão por $-n$ traz o ponto $\mathbf{p}_2$ que estava atrás da câmera para uma posição a frente.



(a) antes (b) depois

Fig. 7.25 – Exemplo de problema do recorte no R^3

A solução para evitar este efeito indesejável consiste em descartar os pontos que estão atrás da câmera antes de fazer esta divisão. Estes pontos são caracterizados pelo valor de suas

coordenadas z positivas. Quando estes pontos são transformados pela transformação de projeção eles ficam caracterizados pelo valor das suas coordenadas homogêneas, w, negativas.

Ou seja, a solução mais indicada consiste em recortar no espaço homogêneo PR^4 acrescentando a condição que só nos interessa os pontos com $w>0$. A dificuldade nesta escolha está em escrevermos a equação destes hiperplanos com normal para fora.

Para deduzirmos a equação do hiperplano do PR^4 correspondente a fronteira "à esquerda", podemos partir desta condição coordenadas do espaço normalizado:

$$x_n \geq -1 \tag{7.28}$$

Se voltarmos para o espaço antes da divisão por w temos:

$$\frac{x_c}{w} \geq -1 \tag{7.29}$$

Como a multiplicação de uma desigualdade por um valor positivo é diferente da multiplicação por um valor negativo esta equação se divide em duas:

$$x_c \geq -w \quad se \quad w > 0 \tag{7.30a}$$

e

$$x_c \leq -w \quad se \quad w < 0 \tag{7.30b}$$

Somente a primeira equação é válida uma vez que não estamos interessados na região do PR^4 em que w é menor que zero. Com isto resolvemos parte do nosso problema, temos uma inequação que define nossa fronteira. A questão agora consiste em escolhermos a equação que resulta com a normal para fora. Isto porque podemos escrever equação (7.28a) de duas maneiras:

$$0 \geq -w - x_c \tag{7.31a}$$

e

$$+w + x_c \geq 0 \tag{7.31b}$$

Tomando a igualdade em cada uma delas temos duas equações para o mesmo hiperplano:

$$-w - x_c = 0 \tag{7.32a}$$

e

$$+w + x_c = 0 \tag{7.32b}$$

Qual a diferença entre eles? A direção da normal. Uma é o negativo da outra. Para fazermos a escolha vamos consideras as distâncias que elas medem para um ponto qualquer de coordenadas $(x_c,y_c,z_c,w)^T$ ao hiperplano "à esquerda". A primeira resulta em:

$$d = -w - x_c \tag{7.33a}$$

e a segunda em:

$$d = +w + x_c \tag{7.33b}$$

Resta agora escolher a equação em que os pontos internos, que estão à esquerda, resultem em valores negativos (na nossa convenção a normal aponta para fora). A partir das equações (7.29) podemos ver que esta condição implica em escolhermos a equação (7.30a). A partir de um desenvolvimento semelhante ao descrito acima podemos encontrar as equações que medem as distâncias aos outros cinco hiperplanos.

Resumindo as equações que medem distância aos planos à esquerda, à direita, abaixo, acima, à frente, atrás podem serem escritas como:

$$d_l = -w - x_c \tag{7.34a}$$

$$d_r = -w + x_c \tag{7.34a}$$

$$d_b = -w - y_c \tag{7.34a}$$

$$d_t = -w + y_c \tag{7.34a}$$

$$d_n = -w - z_c \tag{7.34a}$$

$$d_f = -w - x_c \tag{7.34a}$$

A função `distancia` mostrada no Quadro 7.1 ilustra quão simples e eficiente podemos codificar os cálculos de distância com relação a estes pontos, justificando o esforço deste desenvolvimento razoavelmente complexo. Ou seja, mais uma vez encontramos no *rendering pipeline* uma dedução complexa que resulta em uma implementação simples e eficiente.

```
double distancia(double x, double y, double z, double w, int plano)
{
 switch( plan0 )
 {
  case 0: return( -w - x );  /* esquerda */
  case 1: return( -w + x );  /* direita */
  case 2: return( -w - y );  /* abaixo */
  case 3: return( -w + y );  /* acima */
  case 4: return( -w - z );  /* perto */
  case 5: return( -w + z );  /* longe */
 }
 return( 0.0 );   /* erro */
}
```

Quadro 7.1 – Função de distância aos hiperplanos de recorte no PR^3

Este espaço homogêneo PR^3, onde o recorte é feito é denominado pelo OpenGL™ de epaço de recorte (*clipping coordinates*).

Divisão por w e mapeamento para a tela

Os vértices das primitivas recortadas, convertidos para o espaço cartesiano normalizado por:

$$x_n = \frac{x_c}{w} \tag{7.35a}$$

$$y_n = \frac{y_c}{w} \tag{7.35b}$$

$$z_n = \frac{z_c}{w} \tag{7.35c}$$

tem valores entre -1 e 1. Estes valores são então transformados em coordenadas da tela. Seguindo as convenções da função `glViewport` do OpenGL™ a região da tela onde as primitivas são desenhadas corresponde a um retângulo $w \times h$ *pixels* posicionado da forma mostrada na Fig. 7.26. Devemos notar que, diferente do Windows™, o sistema de eixos de uma janela no OpenGL™ tem origem no canto inferior esquerdo e o eixo y vai de baixo para cima.

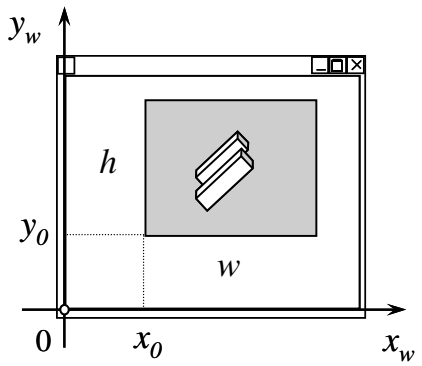


Fig. 7.26 – *Viewport* numa janela da tela segundo o OpenGL™

A transformação linear que leva os intervalos [-1,+1] das coordenadas do sistema normalizado para as coordenadas da janela é dada por:

$$x_w = x_0 + w\left(\frac{x_n + 1}{2}\right) \tag{7.36a}$$

$$y_w = y_0 + h\left(\frac{y_n + 1}{2}\right) \tag{7.36b}$$

Um ponto importante é que todas estas transformações preservaram a orientação da definição do *frustum* de visão. Ou seja, o usuário vê o ponto do plano *near* de coordenadas *left* e *bottom* no canto inferior esquerdo da janela e o *right* e *top* no canto superior direito, como era de se esperar.

Como já foi dito anteriormente as próximas etapas do *rendering pipeline* necessitam também da coordenada z_w dos vértices para resolver problemas como o de visibilidade. Como estes valores são armazenados para todos os *pixels* é natural procurarmos armazená-los com inteiros pequenos para economizar memória. Como os valores de $z_n \in [-1,1]$, a transformação:

$$z_w = z_{\max}\left(\frac{z_n + 1}{2}\right) \tag{7.36}$$

leva os valores de z_v para o intervalo [0, z_{max}]. Se, para cada *pixel*, o mapa de profundidade armazenar um número inteiro, não negativo, de *n bits*, $z_{max} = 2^n$-1.

Um ponto importante neste mapa de profundidade é a escala não uniforme da coordenada z depois da transformação **P** definida na equação (7.23). Para discutirmos melhor este efeito devemos analisar a transformação que leva do espaço da câmera (*eye* coordinates) para as coordenadas normalizadas que é onde a distorção ocorre.

Utilizando a expressão de **P** da equação (7.23) a transformação das coordenadas da câmera para as coordenadas de recorte pode ser escrita por:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} \tag{7.38}$$

Desta equação podemos tirar as coordenadas homogêneas z e w através de:

$$\begin{cases} z_c = -\dfrac{f+n}{f-n} z_e - \dfrac{2fn}{f-n} \\ w = -z_e \end{cases} \tag{7.39}$$

Convertendo z_c para o espaço cartesiano temos a coordenada normalizada z_n:

$$z_n = \frac{z_c}{w} = \frac{f+n}{(f-n)} + \frac{2fn}{(f-n)}\frac{1}{z_e} \tag{7.40}$$

Para entender esta função a Fig. 7.27 mostra três gráficos onde o plano *far* está a uma distância arbitrária 100 metros e o plano *near* a 1, 10 e 30 metros, respectivamente. Quando n tem valores pequenos quando comparado a f (1%, por exemplo), o mapeamento fica bastante distorcido e profundidades diferentes no espaço da câmera podem resultar em profundidades muito próximas no espaço normalizado. Isto pode ser confirmado se analisarmos quanto um intervalo de z no espaço da câmera fica reduzido no espaço normalizado para valores de z mais próximos do plano *far*, como ilustra a figura.

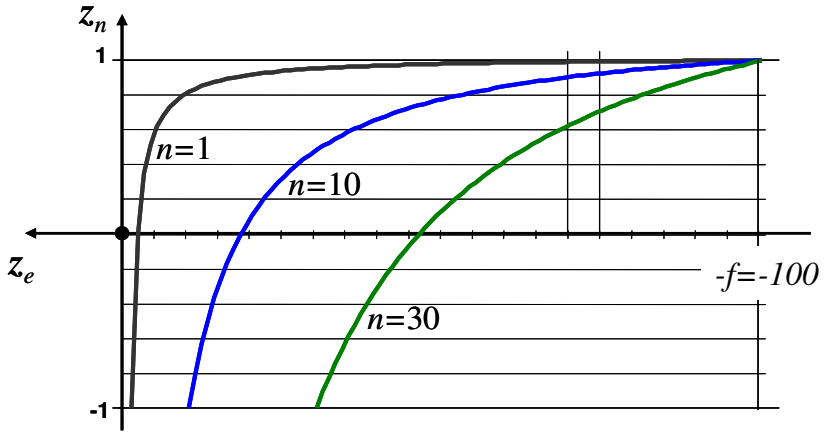


Fig. 7.27 – Mapeamento de profundidade em função dos valores de *near* e *far*

Para tornar esta discussão mais concreta, vamos incorporar a transformação do sistema de coordenadas de recorte para o sistema de coordenadas da janela, substituindo a equação (7.38) na equação (7.35):

$$z_w = \frac{z_{\max}}{2}\left(\left(\frac{f+n}{(f-n)} + \frac{2fn}{(f-n)}\frac{1}{z_e}\right)+1\right) \tag{7.41}$$

Como dito anteriormente, estes valores estão no intervalo $[0, z_{max}]$ e devem ser armazenados com inteiros no mapa de profundidade. Podemos, ainda inverter esta função e escrever as coordenadas da câmera em função das coordenadas do mapa de profundidade. Após alguns passos chegamos a:

$$z_e = \frac{fn}{\frac{z_w}{z_{\max}}(f-n) - f} \tag{7.42}$$

Se adotarmos 16 *bits* para armazenar a profundidade (z_{max} = 65535), f = 1000 e n= 0.01, podemos investigar qual o valor de z_e que mapeia para 65534. Ou seja, a que profundidade no espaço da câmera apenas uma unidade a menos do que o valor que mapeia para z_e=-1000. A equação (7.40) tomaria os seguintes valores e resultaria em:

$$z_e = \frac{1000\times 0.01}{\frac{65534}{65535}(1000-0.01)-1000} = -396$$

Destas contas podemos verificar que todos os pontos que estão a uma profundidade maior que 39,6% do tamanho do prisma de visão na direção z mapeiam para o mesmo valor no mapa de profundidade! Isto corresponde a um arredondamento grosseiro que pode não distinguir corretamente quais pontos são visíveis.

Numa animação em que a posição da câmera muda continuamente nós temos que ser cuidadosos com a escolha dos valores de n e f. Uma escolha exagerada, como a mostrada acima, leva resultados de oclusão muito ruins. Mais ainda, mesmo com escolhas razoáveis de valores de n e f é comum termos, nas animações feitas com mapa de profundidade em

placas de 16 *bits*, objetos próximos entre si aparecendo alternadamente de um quadro para outro durante a animação. Este efeito conhecido como *alias* temporal degrada bastante a qualidade da animação.

Rastreio

Rastreio[7] é o processo pelo qual primitivas geométricas, do tipo linha e polígonos, são convertidas em imagens como ilustra a Fig. 7.28. No algoritmo de mapa de profundidades, cada ponto nestas imagens contém pelo menos a informação de cor e de profundidade.

O processo de rastreio pode ser visto como ocorrendo em duas etapas. Na primeira descobrimos quais quadradinhos do sistema de coordenadas da janela, são ocupados pela primitiva e no segundo atribuímos uma cor e uma profundidade a este quadrado. Estas informações acrescidas de outras, como as coordenadas de textura, são chamadas de fragmentos. A próxima etapa do *rendering pipeline* utiliza estes fragmentos para atualizar os mapas de cor e profundidade.

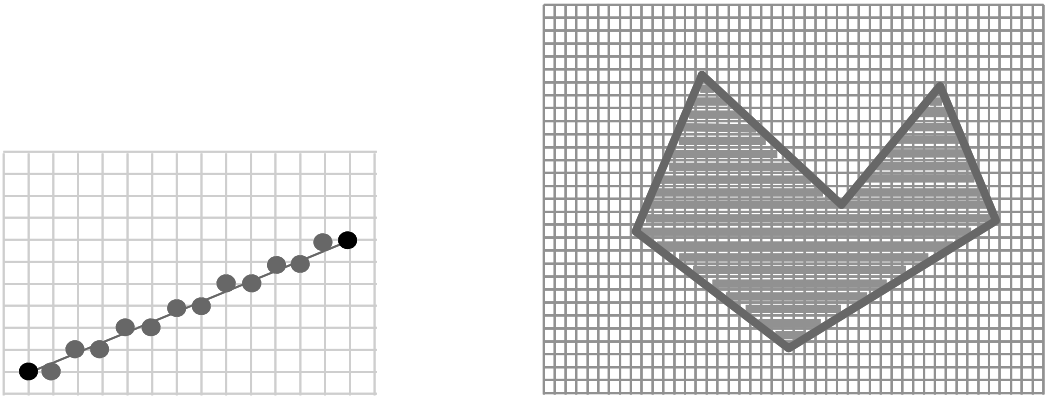


Fig. 7.28 – Rastreio de linhas e polígonos

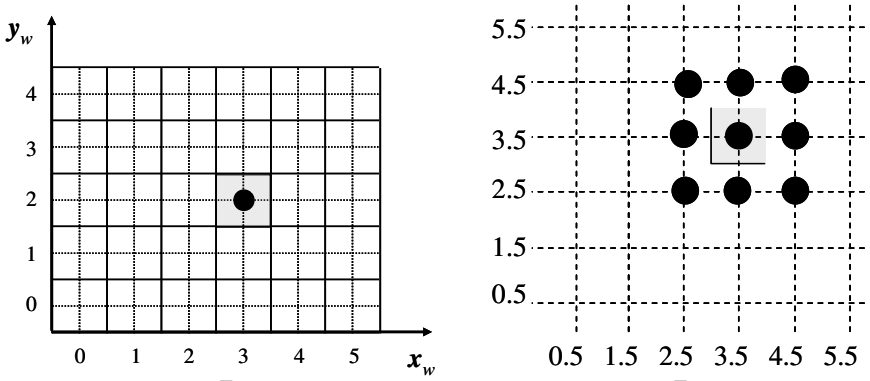
Pontos

Para elucidar melhor o sistema de coordenadas da janela vamos considerar inicialmente a rastreio de pontos que é mais simples. A Fig. 7.29(a) ilustra qual quadradinho é afetado por um ponto de coordenadas $(3,2)^T$ numa janela de largura 6 e altura 6. A Fig. 7.29(b) mostra os nove quadradinhos que são interceptados por um ponto de tamanho 3. As coordenadas na grade do centro de um ponto de tamanho ímpar são dadas por:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \lfloor x_w \rfloor + \frac{1}{2} \\ \lfloor y_w \rfloor + \frac{1}{2} \end{pmatrix} \tag{7.43}$$

[7] Rastreio é utilizado aqui como a tradução da palavra inglesa *rasterization*.

As figuras desta seção utilizam uma bola preta para marcar esta posição.



(a) Ponto $(3,2)^T$ de tamanho 1 (b) Ponto $(3,3)^T$ de tamanho 3

Fig. 7.29 – Rastreio de linhas e polígonos

Segmentos de reta

A Fig. 7.30 ilustra três casos triviais de rastreio de segmentos de reta: horizontais, verticais e inclinados à 45^o. Nestes casos a escolha dos quadradinhos gerados pelo segmento de reta é óbvia. Na realidade existem detalhes, como segmentos com espessura diferente de um e técnicas de antialias, que omitimos aqui para não sobrecarregar o assunto.

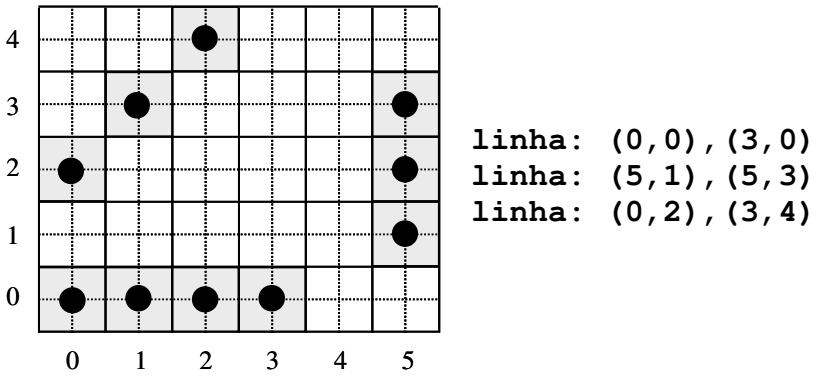


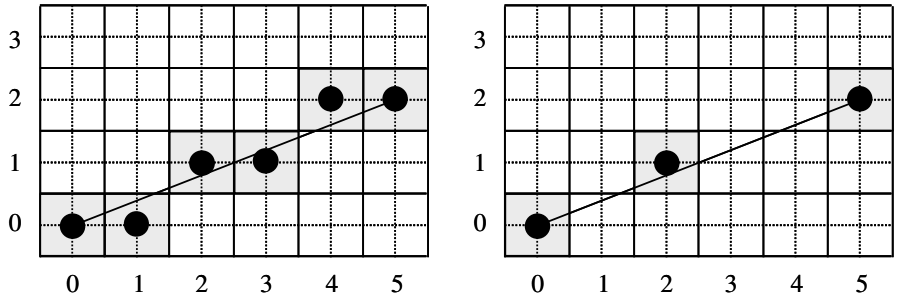
Fig. 7.30 – Casos triviais de linhas

A escolha dos quadradinhos interceptados por um segmento de reta de inclinação qualquer não é tão óbvia. Bresenham (1965) propôs um critério geométrico para definir quais quadradinhos são afetados por um segmento de reta de acordo com o coeficiente angular m dele ($y=mx+b$).

Se o segmento de reta estiver numa posição mais próxima da horizontal do que da vertical, $m \in [-1,1]$, dizemos que ele é x-dominante e escolhemos um quadradinho para cada coluna da imagem (Fig. 7.31(a)). Caso contrário, dizemos que o segmento é y-dominante e escolhemos um quadradinho por linha.

A Fig. 7.31 ilustra um segmento que é x-dominante com as duas escolhas: uma correta e a outra errada. A opção errada da direita ilustra a importância de um bom critério

geométrico. O critério de Bresenham não deixa o segmento de reta ficar falhado, sem engrossá-lo desnecessariamente.



(a) correto (b) errado

Fig. 7.31 – Critério geométrico de Bresenham para segmentos de reta

A Fig. 7.32 ilustra o critério geométrico dos losangos de altura um adotado pelo OpenGL™ para segmentos de reta. Segundo este critério, apenas os losangos em que o segmento de reta sai deles são interceptados. Este critério seleciona os mesmos quadradinhos que o critério de Bresenham a menos do último. No OpenGL™ o último quadradinho de um segmento de reta não gera um fragmento para evitar duplicação deste fragmentos caso venhamos a fazer o rastreio de outro segmento reta que parte do ponto final do anterior.

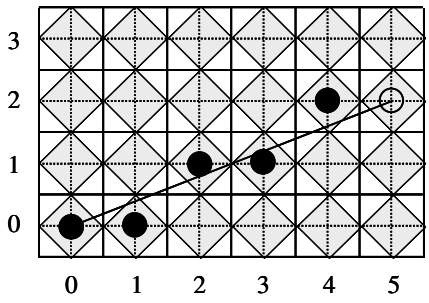


Fig. 7.32 – Critério geométrico do OpenGL™ para segmentos de reta

A função escrita em C e apresentada no Quadro 7.2 gera os fragmentos interceptados por um segmento de reta que vai de (x_1,y_1) até (x_2,y_2) segundo o critério geométrico de Bresenham.

```
void linha(int x1, int y1, int x2, int y2)
{
  float m = (y2-y1)/(x2-x1);
  float b = y1 - m*x1;
  float y;

  fragmento(x1,y1);   /* cria um fragmento com esta posicao */
  while( x1 < x2 )
     {
      x1++;
      y = m*x1 + b;

      fragmento(x1,ROUND(y));
     }
```

```
}
```

Quadro 7.2 – Função simples de segmento de reta segundo o critério de Bresenham

A macro `ROUND(y)` arredonda o valor de y para o inteiro mais próximo. Ela pode, por exemplo, ser escrita como sendo:

```
#define ROUND(x) (int)floor((x)+0.5)
```

A função do Quadro 7.2 é pouco eficiente. Uma das principais razões para sua ineficiência é a necessidade de uma multiplicação a cada passo do laço. Uma maneira simples de contornarmos esta multiplicação consiste em observarmos que, dada a linearidade da relação entre x e y, quando x varia em intervalos constantes (e iguais a um) y também varia de forma uniforme. Ou seja, se para um dado valor de x_i o valor de y é:

$$y_i = mx_i + b \tag{7.44a}$$

O valor de y para o próximo x é dado por:

$$y_{i+1} = m(x_i + 1) + b \tag{7.44b}$$

Se subtrairmos (7.42b) de (7.42a) temos a variação de y para incrementos de x:

$$y_{i+1} - y_i = m \tag{7.45}$$

A função do Quadro 7.2 pode então ser escrita de forma um pouco mais eficiente através de:

```
void linha(int x1, int y1, int x2, int y2)
{
 float m = (y2-y1)/(x2-x1);
 float b = y1 - m*x1;
 float y=y1;          /* inicializa o valor de y */

 pixel(x1,y1);
 while( x1 < x2 )
    {
     x1++;
     y += m;           /* incremento constante */

     fragmento(x1,ROUND(y));
    }
}
```

Quadro 7.3 – Função incremental de segmento de reta segundo o critério de Bresenham

A função linha do Quadro 7.3 é baseada nas operações com números ponto flutuante que são menos eficientes com números inteiros. Para evitar as operações de ponto flutuante devemos observar que o rastreio de um segmento de reta consiste basicamente em varrer todos os valores inteiros de x e y entre os limites dos valores de seus vértices. Como, num segmento x-dominante, temos mais valores em x que em y, os primeiros variam de um em um enquanto que os segundos tomam alguns valores repetidos. A questão então consiste em saber quando os valores de y variam e quando repetem. Bresenham propôs um

algoritmo que utiliza uma variável de controle que indica o erro de mantermos o valor de y constante em cada passo do algoritmo. Este algoritmo, por ser pioneiro, ficou famoso mas não pode ser facilmente generalizado para outras curvas, como círculos e elipses.

O algoritmo do ponto médio tem a mesma idéia que o algoritmo de Bresenham mas a variável de controle tem um significado geométrico o que permite que o algoritmo seja generalizado como mostramos a seguir.

Considere a equação de um segmento de reta que vai de (x_1,y_1) até (x_2,y_2) como sendo:

$$F(x, y) = a.x + b.y + c = 0 \qquad (7.46)$$

onde: $a = y_2 - y_1$, $b = x_1 - x_2$ e c tem um valor que não vem ao caso. Para uma reta no primeiro octante, como mostram as figuras acima, com estas escolhas de a e b, a função $F(x, y)$ separa os ponto do plano em três conjuntos: os que estão acima da reta suporte do segmento, $F(x, y) < 0$, os que estão no segmento, $F(x, y) = 0$, e os que estão abaixo, $F(x, y) > 0$. Devemos notar que se invertermos o sinal de a e b estes sinais trocam e a dedução tem que ser refeita com algumas trocas no algoritmo final.

A Fig. 7.33 mostra uma análise do algoritmo indo do ponto x_p para x_p+1. Nas condições do segmento estar no primeiro octante, existem duas escolhas para o próximo ponto: leste ou nordeste (marcados com **e** e **ne** na figura). O algoritmo evolve para nordeste caso o valor de F no ponto médio, **m**, seja maior que zero como ilustra a figura da esquerda. Caso o valor de F em **m** seja menor que zero o ponto **e** se torna o próximo do algoritmo. O caso de F igual a zero pode ser incluído em qualquer uma das escolhas anteriores, desde que consistentemente.

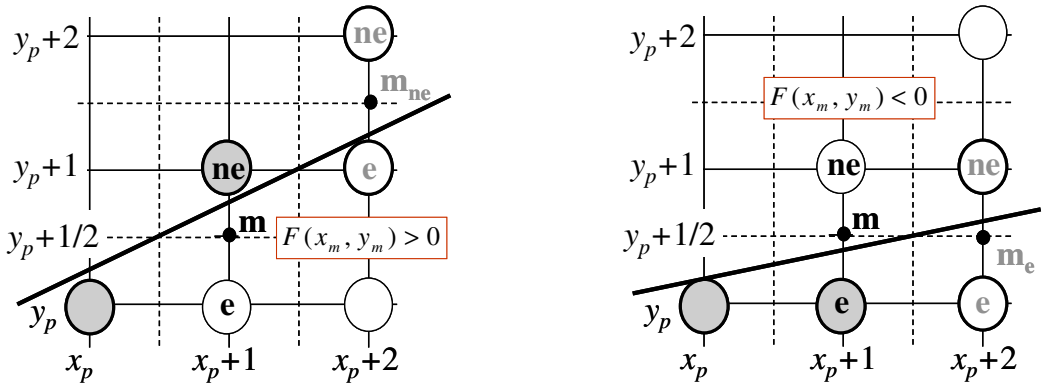


Fig. 7.33 – Critério de escolha do valor de y do próximo ponto do segmento de reta

Resumindo, o algoritmo do ponto médio para segmento de reta no primeiro octante consiste em varrer todas as colunas da esquerda para direita partindo do ponto (x_1,y_1). O valor de y ou permanece o mesmo, escolha **e**, ou é incrementado de um, escolha **ne**, em função do sinal da função F no ponto **m**.

O único problema que ainda temos que resolver é custo do cálculo da função $F(x,y)$. Como ela é uma função linear podemos fazer este cálculo de forma incremental, como fizemos no Quadro 7.3, mas continuaríamos com pontos flutuantes. Para eliminar a necessidade de ponto flutuante vamos criar no algoritmo uma variável d de valor $2F$. Esta variável tem o mesmo sinal que F logo pode substituí-lo no teste de escolha **e** ou **ne.** A vantagem desta

variável sobre F é que o seu valor no ponto pode ser calculado com aritmética inteira. No ponto inicial ela vale:

$$d_{ini} = 2F(x_1 + 1, y_0 + ½) = 2a(x_0 + 1) + 2b(y_0 + ½) + 2c \qquad (7.47a)$$

ou

$$d_{ini} = 2(ax_1 + by_1 + c) + 2a + b = 2F(x_1, y_1) + 2a + b = 2a + b \qquad (7.47b)$$

Quando o algoritmo evolve para o leste ou para o nordeste seu valor pode ser, respectivamente, calculado por:

$$d_{novo} = 2F(x_p + 2, y_p + ½) = 2a(x_p + 2) + 2b(y_p + ½) + 2c = d_{ant} + 2a \qquad (7.48a)$$

$$d_{novo} = 2F(x_p + 2, y_p + ⅔) = 2a(x_p + 2) + 2b(y_p + ⅔) + 2c = d_{ant} + 2(a + b) \qquad (7.48b)$$

O Quadro 7.4 mostra uma implementação do algoritmo do ponto médio para rastreio de um segmento de reta feita apenas com números inteiros. Devemos observar que esta função gera os mesmo fragmentos que a função do Quadro 7.2 e é bem mais eficiente.

```
 1 void linhaPM(int x1, int y1, int x2, int y2)
 2 {
 3   int a = y2-y1;
 4   int b = x1-x2;
 5   int d=2*a+b;            /* valor inicial da var. decisao */
 6   int incrE = 2*a;        /* incremento p/ mover E */
 7   int incrNE = 2*(a+b);   /* incremento p/ mover NE */
 8
 9   fragmento(x1,y1);
10   while (x1<x2) {
11       x1++;
12       if (d<=0)     /* escolha E */
13          d+=incrE;
14       else {        /* escolha NE */
15          d+=incrNE;
16          y1++;
17       }
18       fragmento(x1,y1);
19    }
20
21 }
```

Quadro 7.4 – Algoritmo do ponto médio para segmentos de reta

Estilos de linha podem ser facilmente implementados nos algoritmos desta seção. Se no código do Quadro 7.4, por exemplo, substituirmos as linhas 8 e 18 por:

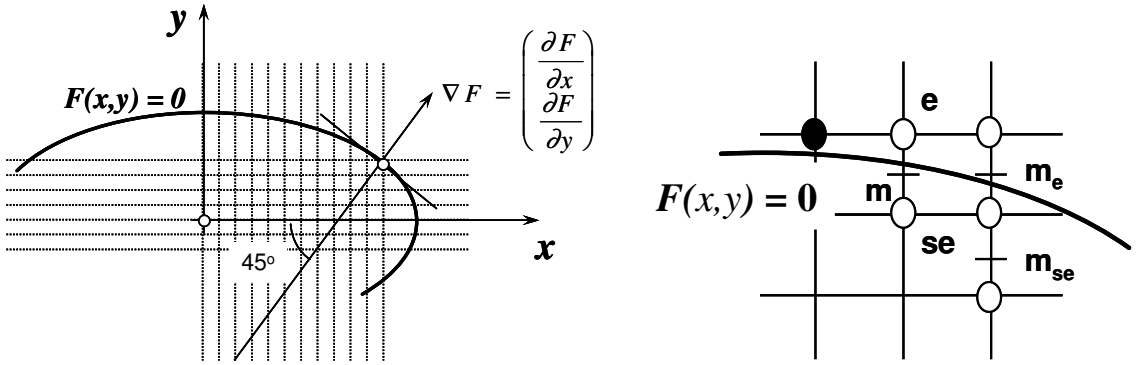
```
08  int estilo [8]={1,1,0,0,1,1,0,0}; int k=1;
18     if (estilo[(++k)%8]==1) fragmento(x1,y1);
```

passaríamos a gerar linhas tracejadas com os traços de comprimento de dois *pixels*. Constantes inteiras de também podem ser utilizadas para definirmos estilos, basta percorrermos a constante *bit* a *bit* de forma circular como fizemos com os elementos do vetor acima.

Círculos e elipses

A Fig. 7.34 ilustra a extensão do algoritmo do ponto médio para o rastreio de elipses. A Fig. 7.34(a) mostra o critério geométrico de Bresenham para elipses. Segundo este critério quando a curva está próxima da horizontal tomamos de um *pixel* por coluna, quando ela esta mais próxima da vertical tomamos um *pixel* por linha. O ponto de mudança também está ilustrado na figura e corresponde ao gradiente da curva a 45°.

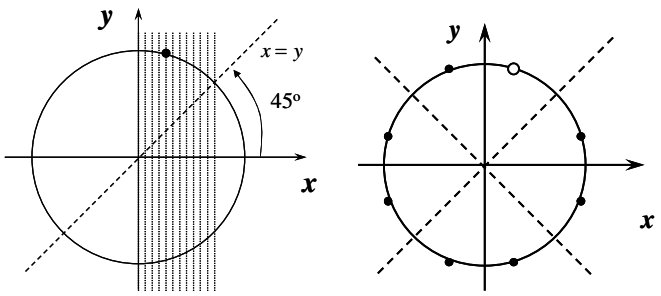
Na Fig. 7.34(b) temos o critério do ponto médio aplicado ao segundo octante com o algoritmo iniciando no ponto do eixo *y* e caminhando para direita até o ponto de gradiente 45°. Aqui as escolhas são sul e sudeste como ilustra a figura. O algoritmo é bastante semelhante ao apresentado para reta. A maior diferença vem do fato de que a função *F* é quadrática neste caso. Para computar seus valores eficientemente é necessário calcularmos os incrementos dos incrementos de *y* quando *x* é acrescido de 1. Ou seja, como incremento varia linearmente temos que utilizar diferenças de segunda ordem para calculá-lo sem uso da multiplicação, como fizemos no algoritmo de segmentos de retas.



(a) (b)

Fig. 7.34 – Critérios geométricos para rastreio de elipses

O círculo é um caso particular da elipse com duas simplificações importantes. A primeira diz respeito ao ponto de mudança de *x* dominante para *y* dominante. Como ilustra a Fig. 7.35(a) este ponto é simplesmente o ponto em *x* é igual *y*. A segunda, que dadas as simetrias ilustradas na Fig. 7.35(b) o algoritmo só necessita percorrer o segundo octante. Ou seja, quando calculamos um ponto (x,y) como pertencente ao círculo, os pontos: $(-x,y)$, $(x,-y)$, $(-x,-y)$, (y,x), $(-y,x)$, $(y,-x)$, $(-y,-x)$ também pertencem a ele e os fragmentos correspondentes podem ser automaticamente gerados.



(a) Critério geométrico (b) simetrias

Fig. 7.35 –Rastreio de círculos

Quadro 7.5 ilustra um algoritmo para rastreio de um círculo centrado na origem. Para adaptá-lo para um circulo qualquer basta transladá-lo do valor da posição de seu centro.

```
x=0,y=raio;
fragmento(x,y);
while (x<y) {
     x++;
     if (F(M)<0)
        escolha E;
    else
        escolha SE;
    fragmento(E ou SE);
}
```

Quadro 7.5 – Rastreio de círculos

Polígonos

O rastreio de polígonos quaisquer requer que definamos primeiro o conceito de interior e exterior. Quando o polígono é fechado por uma curva que não tem auto-interseção este conceito é natural e intuitivo, mas quando o sistema gráfico recebe uma seqüência de vértices como o ilustrado na Fig. 7.36(a) precisamos de um critério para definir interior.

Um dos critérios mais utilizados para definir interior de polígono consiste em lançar um raio do ponto em questão em uma direção qualquer e calcular o número de vezes em que o raio intercepta a curva da fronteira. Se este número for par (0,2,4,...) significa que o ponto é exterior. A explicação segue o seguinte racional: como os polígonos são regiões fechadas, o último segmento do raio está fora ou seja é externo. Cada interseção representa uma mudança de estado. Como temos um numero par, temos conjuntos de entrada e saída acoplados. Daí ou o raio nunca entrou no polígono (zero interseções) ou entrou e saiu uma ou mais vezes (2,4,6... interseções). Seguindo o mesmo raciocínio se o número de interseções for ímpar o ponto é interior. Esta regra é chamada de regra *par-ímpar*.

As Fig. 7.36 (b) e (c) ilustram a regra par-ímpar aplicada a polígonos simples. A Fig. 7.36(d) mostra o interior do polígono da Fig. 7.36(d) quando resultado desta regra é aplicada.

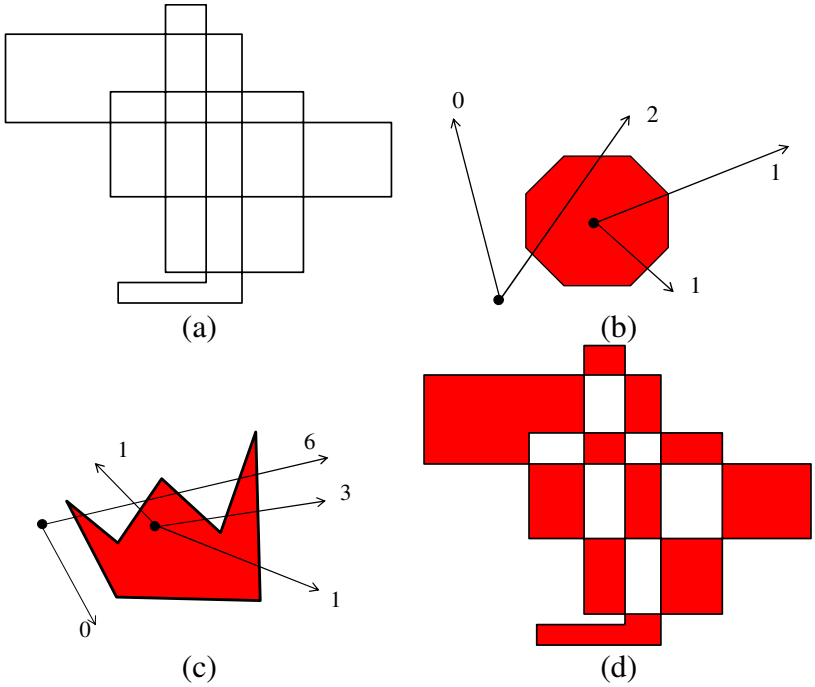
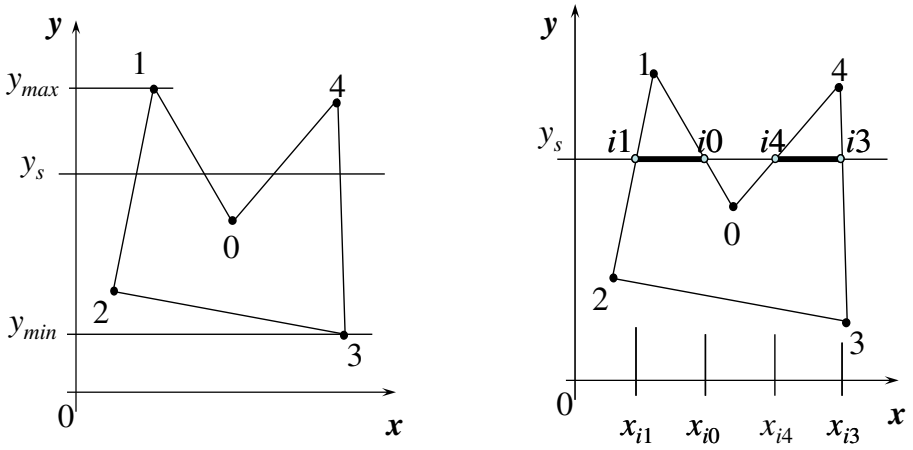


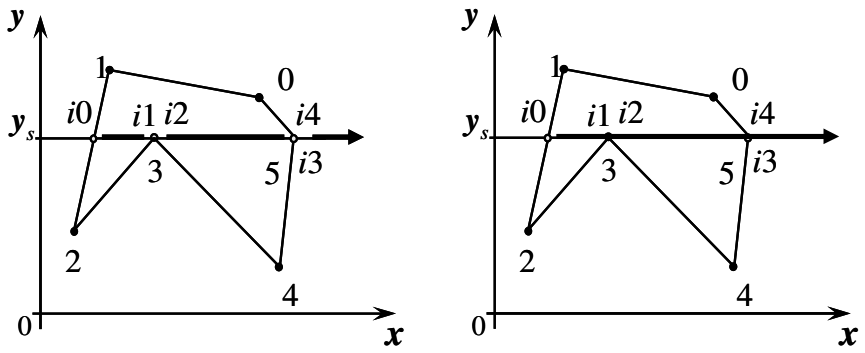
Fig. 7.36 –Conceito de interior de polígonos

Com base no critério par-ímpar podemos desenvolver um algoritmo para preenchimento de um polígono qualquer, como ilustrado na Fig. 7.37. O polígono é fornecido ao sistema gráfico como uma lista de coordenadas inteiras de vértices. No caso da figura: (x_0,y_0), (x_1,y_1), (x_2,y_2), (x_3,y_3) e (x_4,y_4). A primeira parte do algoritmo consiste em determinarmos o escopo das linhas afetadas pelo polígono, ou seja, y_{max} e y_{min}. Para cada valor inteiro de y neste intervalo temos uma linha de rastreio (*scan line*) denominada na figura de y_s. Para cada um destas linhas temos que determinar um ou mais intervalos de x que representam a interseção da linha de rastreio com o polígono. O cálculo da interseção da linha de rastreio com a aresta segue, normalmente, a ordem das arestas do polígono que é implicitamente definidas pela ordem dos vértices. No caso da figura em questão a interseção resultaria na x_{i0}, x_{i1}, x_{i3} e x_{i4}, mostrados na Fig. 7.37(b). Colocando estes valores em ordem crescente em x temos: x_{i1}, x_{i0}, x_{i2}, x_{i4} e x_{i3}. Tomando estes valores dois a dois podemos gerar os fragmentos que estão entre eles.



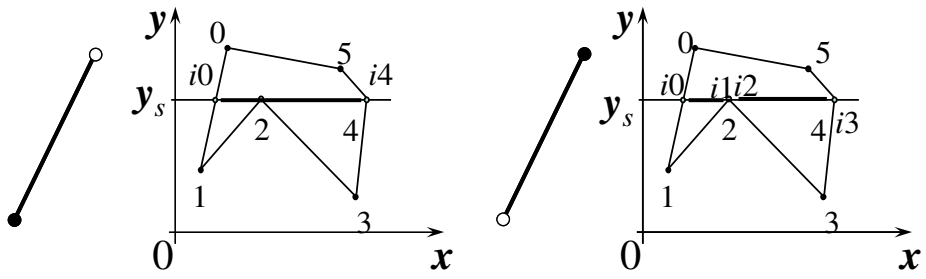
(a) Escopo das linhas de rastreio (b) Interseção de uma linha com o polígono
Fig. 7.37 – Preenchimento de polígonos quaisquer

Como os polígonos são fechados e, o que deve ser exibido, está dentro da janela o algoritmo deve sempre calcular um número par de interseções. Um problema ocorre, entretanto, quando as linhas de rastreio que passam pelos vértices (e elas sempre passam uma vez que estamos utilizando inteiros). Como calculamos a interseção da linha de rastreio com cada aresta do polígono ficamos num dilema: se incluirmos as interseções com os vértices das arestas, temos excesso de pontos (ver Fig. 7.38(a)). Se não incluirmos os vértices, temos falta (ver Fig. 7.38(b)).



*i*0-*i*1, *i*2-*i*3, *i*4-**?** *i*0-**?**
(a) incluindo os vértices (b) não incluindo os vértices
Fig. 7.38 – Problema com as interseções da linha de rastreio com os vértices

Uma solução simples para este problema consiste em incluirmos a interseção com o vértice somente se ele for o de maior (ou menor) ordenada *y* na aresta. Assim quando duas arestas cruzam a linha de rastreio contamos apenas na que está abaixo (ou acima). Quando as arestas tocam a linha de rastreio tanto faz não contarmos ou contarmos duas vezes. No primeiro caso o trecho passa direto e no segundo ele para mas gera outro conectado, ou seja, gera os mesmos fragmentos.



*i*0- *i*4 *i*0-*i*1, *i*2-*i*3
(a) incluindo os vértices y_{min} (b) incluindo os vértices y_{max}
Fig. 7.39 – Soluções para o problema das interseções da linha de rastreio com os vértices

O problema de duplicação de fragmentos mencionado no algoritmo de rastreio de segmentos de retas que tem vértices em comum também aparece aqui. Se gerarmos os

fragmentos da interseção à esquerda até, inclusive, a interseção à direita, em todas as linhas de rastreio, incluindo a mais alta e a mais baixa, teremos duplicação de fragmentos nas arestas comuns de polígonos adjacentes. As soluções esboçadas tanto na Fig. 7.39(a) quanto na (b), já reduzem as ocorrências desta duplicação. Para completar podemos adotar dois critérios a mais: desconsiderar as arestas horizontais e não gerar o fragmento mais à direita em cada intervalo da linha de rastreio. Estes critérios, em conjunto com um dos dois critérios da Fig. 7.39, definem a que polígono devem ser atribuídos os fragmentos de uma fronteira comum sem gerar duplicações ou buracos. O leitor pode verificar isto através da analise exaustiva de casos.

Este algoritmo tem dois passos caros: o cálculo para cada linha de rasteio da interseção com todas as arestas do polígono, e a ordenação destas interseções. Para facilitar a discussão que se segue vamos definir que a linha de rastreio vai de baixo para cima. Ou seja, uma linha de rastreio tem o valor de y igual ao da linha anterior mais um.

Podemos reduzir o custo do cálculo da interseção das linhas de rastreio com uma aresta se observarmos que o valor de x na interseção de uma linha de rastreio é igual ao da linha de rastreio anterior mais um valor constante (Fig. 7.40).

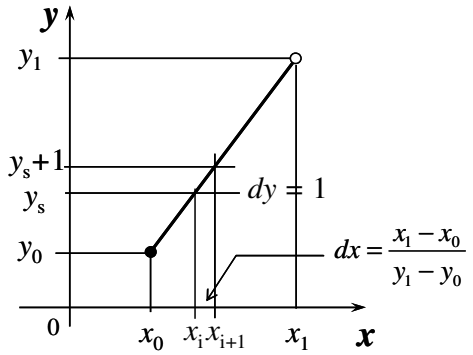


Fig. 7.40 – Otimização no cálculo da interseção da linha de rastreio com uma aresta

Triângulos

Os algoritmos de rastreio de triângulo podem ser implementados de uma forma muito mais eficiente que os algoritmos de polígonos. Por isto o OpenGL™ adota como estratégia básica fazer o rastreio de polígonos decompondo-os em triângulos. Na biblioteca básica ele, inclusive, só trata de polígonos convexos que são triviais de serem decompostos em triângulos, como ilustra a Fig. 7.41. Polígonos côncavos só são tratados em bibliotecas auxiliares.

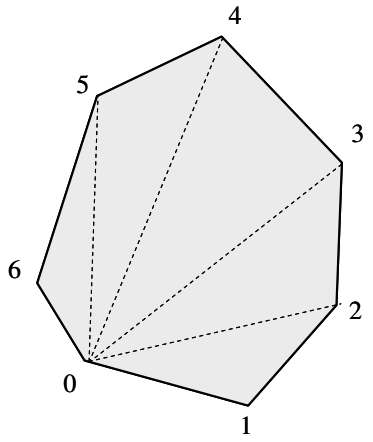


Fig. 7.41 – Decomposição de um polígono convexo em triângulos

Para apresentar o algoritmo de rastreio de triângulos adotamos aqui a seguinte notação: a letra A corresponde ao vértice de maior y, B corresponde ao intermediário e C ao de menor y ($y_A \geq y_B \geq y_C$). As arestas são nomeadas com a letra minúscula correspondente ao vértice oposto.

Um triângulo qualquer pode estar posicionado nas duas posições mostradas na Fig. 7.42. Dadas as coordenadas dos vértices podemos descobrir qual o caso estamos tratando fazendo o produto vetorial:

$$\vec{c} \times \vec{b} = (x_B - x_A)(y_C - y_A) - (x_C - x_A)(y_B - y_A) \tag{7.49}$$

O sinal deste produto identifica a posição do triângulo conforme mostra a Fig. 7.42.

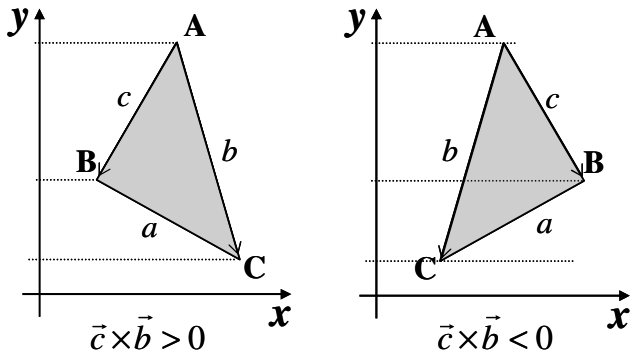


Fig. 7.42 – Posições possíveis de um triângulo dado $y_A \geq y_B \geq y_C$

Por simplicidade vamos considerar apenas o caso em que o ponto B está a esquerda. O algoritmo para o outro caso segue o mesmo raciocínio. No caso do rastreio de triângulos temos apenas dois laços: um que leva a linha de rastreio de y_C, inclusive, até y_B, exclusive. Outro que leva a linha de rastreio de y_B, inclusive, até y_A, exclusive. A Fig. 7.43 ilustra estas duas etapas. Na primeira etapa a linha de rastreio vai de x_a, inclusive, até x_b, exclusive. Estes valores podem ser eficientemente calculados se observarmos que para a primeira linha de rastreio eles têm valor igual à x_C. Nas linhas subseqüentes cada um deles

é incrementado pelo seu correspondente *dx* (ver Fig. 7.40). Este processo incremental também pode ser usado na segunda etapa resultando num algoritmo bastante eficiente.

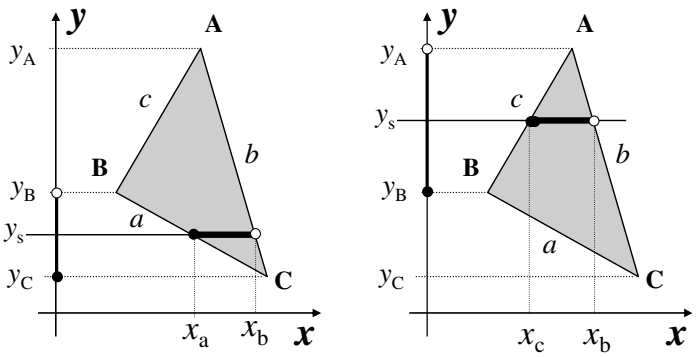


Fig. 7.43 – Etapas do rastreio de triângulos

Interpolação de atributos

Infelizmente a interpolação linear da Fig. 7.40 não é correta para interpolarmos atributos como cor ou coordenadas de textura entre os vértices. A Fig. 7.44(a) mostra um exemplo simples onde uma linha qualquer, dividida uniformemente em 4 cores, é projetada segundo uma projeção cônica. A Fig. 7.44(b) mostra em destaque como estas cores se projetam ao lado de uma interpolação linear para ressaltar as diferenças.

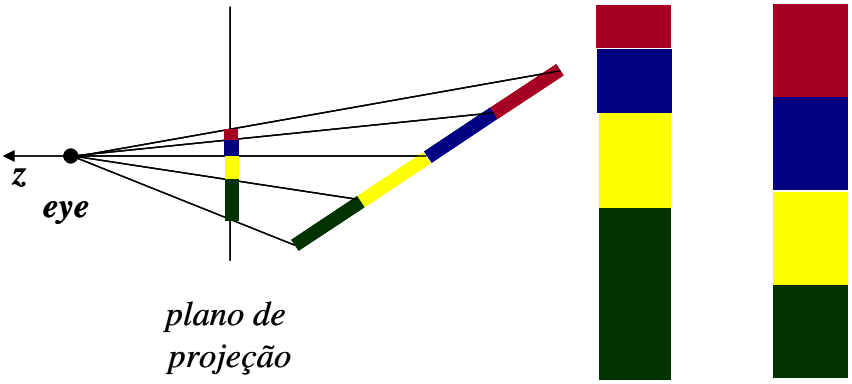


Fig. 7.44 – Projeção de atributos de cor e textura

Na geração de fragmentos a partir dos vértices temos que levar em conta efeitos como este para obtermos realismo visual. Quando olhamos os dormentes ao longo de uma linha de trem ou em uma longa parede de tijolos esperamos ver este efeito. Se ele não estiver presente a imagem não parece real.

Os algoritmos de rastreio interpolam entre vértices ao longo de retas no espaço projetado. Ou seja, capturam pontos igualmente espaçados neste espaço. Para atribuirmos cor ou coordenadas de textura para estes pontos precisamos correlacionar as parametrizações do

espaço projetado e do espaço cartesiano. A idéia básica é que para atribuirmos uma cor ou outro atributo a um ponto no espaço projetado nós possamos saber o valor do atributo interpolando no espaço cartesiano onde a variação é linear.

Na nomenclatura do material apresentado neste capítulo o espaço projetado é o espaço normalizado depois da divisão por *w*, e o espaço cartesiano é o espaço da câmera. Para mantermos a discussão que segue simples e focada no problema da projeção cônica vamos adotar nesta seção os termos cartesiano e projetado, ao invés de da câmera e normalizado. Vamos também adotar a convenção onde pontos no espaço cartesiano são notados por letras maiúsculas em negrito e, seus correspondentes projetados em letras minúsculas, também em negrito. A ausência do negrito caracteriza valores escalares como no resto do livro.

A Fig. 7.45 mostra um segmento $\mathbf{P_0P_1}$ projetado em $\mathbf{p_0p_1}$ num plano que dista um ale das coordenadas *w* de cada um dos vértices, como sendo uma medida da distância de profundidade como deduzimos nas equações (7.18) ou (7.25).

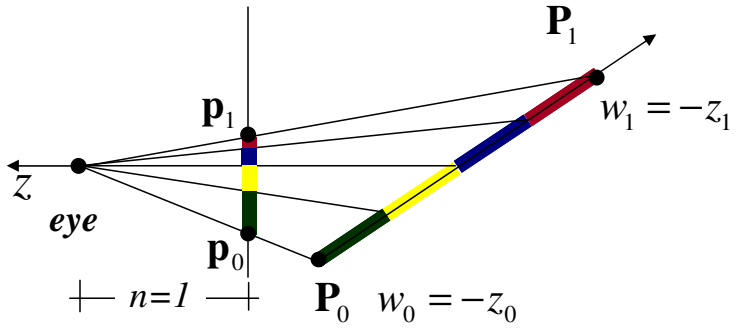


Fig. 7.45 – Interpolação perspectiva

Sejam *t* e *s* os parâmetros de interpolação linear nos espaços projetados e cartesianos, respectivamente. Um ponto genérico nestes segmentos pode ser escrito como sendo:

$$\mathbf{p}(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \tag{7.50}$$

$$\mathbf{P}(s) = (1-s)\mathbf{P}_0 + s\mathbf{P}_1 \tag{7.51}$$

Como a profundidade também varia linearmente ao longo do segmento podemos escrevê-la como:

$$w(s) = (1-s)w_0 + sw_1 \tag{7.52}$$

A projeção cônica do ponto genérico do espaço cartesiano por ser escrita por[8]:

$$\mathbf{p}(s) = \frac{\mathbf{P}(s)}{w(s)} = \frac{(1-s)\mathbf{P}_0 + s\mathbf{P}_1}{(1-s)w_0 + sw_1} \tag{7.53}$$

Se fizermos coincidir este ponto com o interpolado pela equação (7.50) temos a relação entre as coordenadas paramétricas que estamos procurando:

[8] Notem que o denominador é uma grandeza escalar.

$$(1-t)\frac{\mathbf{P}_0}{w_0}+t\frac{\mathbf{P}_1}{w_1}=\frac{(1-s)\mathbf{P}_0+s\mathbf{P}_1}{(1-s)w_0+sw_1} \tag{7.54}$$

Desta equação podemos facilmente explicitar *t* em função de *s*:

$$t=\frac{sw_1}{(1-s)w_0+sw_1} \tag{7.55}$$

A relação que queremos é a inversa desta, que é dada por:

$$s=\frac{tw_0}{(1-t)w_1+tw_0} \tag{7.56}$$

Se um atributo qualquer, por exemplo, uma intensidade luminosa varia linearmente no espaço cartesiano podemos escrevê-lo como:

$$I=(1-s)I_o+sI_1 \tag{7.57}$$

Para capturarmos esta variação no espaço projetado substituímos o valor de *s* pela equação em *t* dada em (7.56), o que resulta em:

$$I=\left(1-\frac{tw_0}{(1-t)w_1+tw_0}\right)I_o+\frac{tw_0}{(1-t)w_1+tw_0}I_1 \tag{7.58}$$

Após algumas manipulações esta equação pode ser simplificada para:

$$I=\frac{(1-t)\left(I_0/w_0\right)+t\left(I_1/w_1\right)}{(1-t)\left(1/w_0\right)+t\left(1/w_1\right)} \tag{7.59}$$

Para calcularmos os valores de *I* para valores de *t* que sofrem incrementos constantes podemos calcular da forma ilustrada na Fig. 7.40 o numerador e o denominador, independentemente. Para obtermos *I* precisamos, para cada fragmento, fazer a divisão do primeiro pelo segundo.

Ou seja, ainda podemos utilizar interpolações lineares para os atributos dos fragmentos para fazer uma interpolação correta sob a transformação cônica. O custo é basicamente uma divisão por fragmento.

Os atributos de textura no OpenGL™ podem ser fornecidos em coordenadas homogêneas, que antes de serem utilizados, são divididos pela coordenada *q*, que corresponde ao *w* do espaço geométrico. Ou seja, se *u* e *q* são as coordenadas homogêneas de textura de um ponto a coordenada de textura cartesiana dele seria *u/q*. A interpolação linear deste atributo no espaço cartesiano também segue a mesma regra. Assim a coordenada de textura de um ponto qualquer no espaço cartesiano entre os pontos $\mathbf{P_0P_1}$ é dada por:

$$\frac{u}{q}=\frac{(1-s)u_o+su_1}{(1-s)q_o+sq_1} \tag{7.60}$$

Se substituirmos o valor de *s* pela equação em *t* dada em (7.56) chegamos, após algumas simplificações, em:

$$\frac{u}{q} = \frac{(1-t)\,u_0/w_0 + t\,u_1/w_1}{(1-t)\,q_0/w_0 + t\,q_1/w_1} \tag{7.61}$$

Esta equação consta da especificação do OpenGL™ como sendo a maneira correta de interpolarmos atributos fornecidos em coordenadas homogêneas ao longo de uma linha no processo de rastreio. Devemos notar esta equação é um caso geral da equação (7.59) em que q_0=q_1=1.

Um outro caso importante é a interpolação da informação de profundidade z ao longo de um segmento de reta qualquer. Ou seja, se substituirmos o atributo I da equação (7.57) pela profundidade z temos:

$$z = (1-s)z_o + sz_1 \tag{7.62}$$

A relação entre as parametrizações do espaço projetado e do espaço cartesiano, dada em função de w na equação (7.56) pode ser re-escrita em:

$$s = \frac{tz_0}{(1-t)z_1 + tz_0} \tag{7.63}$$

dado que $w = -z$. Substituindo esta equação na equação (7.62) encontramos a seguinte expressão:

$$z = (1 - \frac{tz_0}{(1-t)z_1 + tz_0})z_o + \frac{tz_0}{(1-t)z_1 + tz_0} z_1 \tag{7.64}$$

Simplificando esta expressão chegamos a:

$$z = \frac{(1-t)z_1z_0 + tz_0z_1}{(1-t)z_1 + tz_0} = \frac{z_1z_0}{(1-t)z_1 + tz_0} \tag{7.65}$$

Uma maneira mais conveniente de escrevermos esta relação é dada por:

$$\frac{1}{z} = (1-t)\frac{1}{z_0} + t\frac{1}{z_1} \tag{7.66}$$

Ou seja, no espaço projetado o inverso da coordenada z varia linearmente com o inverso das profundidades dos vértices do segmento. Podemos implementar esta expressão com uma interpolação linear seguida de uma inversão (divisão).

Depois de toda discussão desta seção cabe um comentário que, como o custo da divisão por fragmento é significativo, a especificação do OpenGL™ só recomenda interpolar a coordenada z e as coordenadas de textura de forma correta. A interpolação de cores, na maioria das implementações da biblioteca, utiliza o esquema linear simples ilustrado na Fig. 7.40.

A Fig. 7.46 mostra o efeito de interpolação linear[9] sobre a cor de forma a produzir um efeito de superfície suave. Devemos notar que, se compararmos com o elipsóide sem suavização o efeito desta interpolação geometricamente incorreta não é ruim.

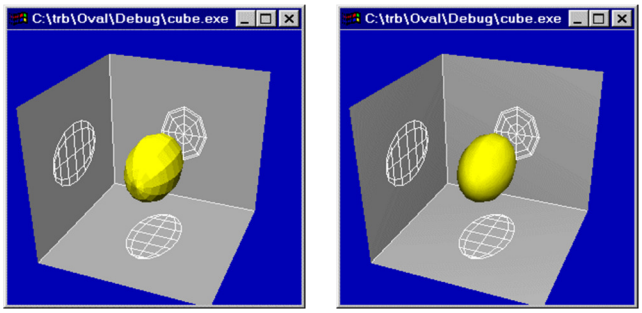


Fig. 7.46 – Interpolação linear do atributo cor

Operação com fragmentos

O último passo do *rendering pipeline* do OpenGL™ consiste em compor os fragmentos oriundos do rastreio nos mapas (*buffers*) de cor e profundidade em função de outros mapas como o mapa de estêncil (*stencil buffer*) e de operações como as operações de composição.

O mapa de profundidade (*ZBuffer*) serve para identificar se o fragmento que está chegando no *pipeline* é ou não visível. Ou seja, somente os fragmentos mais próximos da câmera são considerados, descartando assim aqueles que estão oclusos por outros. Este mapa é inicializado com o valor de z máximo. A medida que um fragmento mais próximo se apresenta sua profundidade fica registrada neste mapa, além é claro dele ser enviado para o mapa de cor. Os próximos fragmentos já testam contra ele. Com este recurso podemos garantir que apenas os fragmentos das primitivas mais próximas da câmera aparecem no mapa de cor, independente da ordem que as primitivas são enviadas para o *rendering pipeline*.

O mapa de estêncil serve para controlarmos quais os *pixels* da janela devem receber fragmentos, e quais devem ser deixados como estão. Um uso típico do mapa de estêncil consiste em criar uma janela de forma arbitrária dentro da área retangular da *viewport*. Assim, por exemplo, podemos definir uma janela escotilha incializando o estêncil com 1's e 0's distinguindo a área circular do resto da janela. A medida que um fragmento é testado contra o estêncil podemos eliminar aqueles que estão fora do círculo desejado. Podemos também criar comportamentos mais complexos se utilizarmos as opções do OpenGL™ que permitem mudar dinamicamente o valor do mapa de estêncil em função dos fragmentos que chegam da etapa de rastreio.

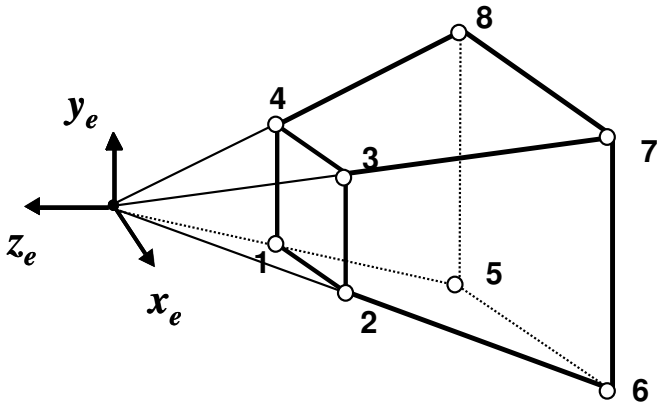
Finalmente, depois de um fragmento passar por todos os testes sua cor pode ser atribuída ao *pixels* correspondente ou podemos fazer composições entre eles. Uma composição comum

[9] Num modelo melhor de iluminação as cores intermediárias entre dois vértices deveriam ser computadas com base na equação de iluminação, ou seja estimando valores de normais e de coeficientes de reflexão nos pontos interpolados. A interpolação linear das cores no algoritmo de *ZBuffer* é conhecida como interpolação de Gouraud que se contrapõe com a proposta de Phong de interpolar normais.

é a correspondente ao operado *over* na qual o canal alfa da cor do fragmento é entendido como sendo a transparência dele, como discutimos no final do capítulo de imagens.

Exercícios Resolvidos

1. Mostre que a matriz de projeção da função glFrustrum do OpenGL transforma o tronco de pirâmide mostrado abaixo num cubo de lado 2. (Sugestão: Calcule a transformada dos oito vértices do tronco de pirâmide).



	xe	ye	ze
1	-2	-1	-2
2	3	-1	-2
3	3	2	-2
4	-2	2	-2
5	-100	-50	-100
6	150	-50	-100
7	150	100	-100
8	-100	100	-100

Resp.:

$n = 2,\ f = 100,\ l = -2,\ r = 3,\ b = -1,\ t = 2$

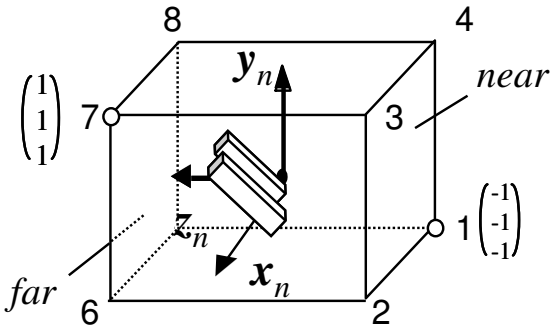
$$\mathbf{P} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0.8 & 0 & 0.2 & 0 \\ 0 & 1.333 & 0.333 & 0 \\ 0 & 0 & -1.0408 & -4.0816 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0.8 & 0 & 0.2 & 0 \\ 0 & 1.333 & 0.333 & 0 \\ 0 & 0 & -1.0408 & -4.0816 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -2 \\ -1 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ -2 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \equiv \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}$$

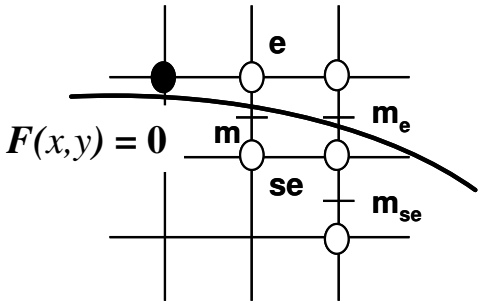
$$\begin{bmatrix} 0.8 & 0 & 0.2 & 0 \\ 0 & 1.333 & 0.333 & 0 \\ 0 & 0 & -1.0408 & -4.0816 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -3 \\ -1 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ -2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \equiv \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$$

…

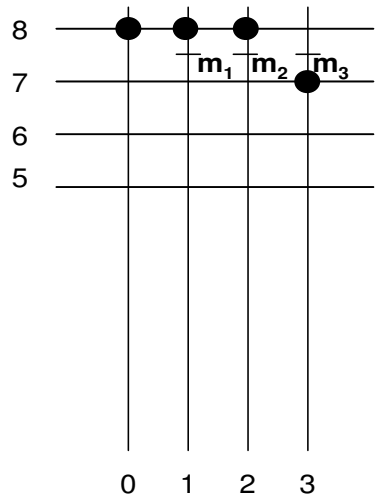
$$\begin{bmatrix} 0.8 & 0 & 0.2 & 0 \\ 0 & 1.333 & 0.333 & 0 \\ 0 & 0 & -1.0408 & -4.0816 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -100 \\ 100 \\ -100 \\ 1 \end{bmatrix} = \begin{bmatrix} -100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \equiv \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}$$



2. Considere um círculo de raio 8 centrado na origem. Partindo do *pixel* (0,8) calcule os três próximos *pixels* do primeiro quadrante que o algoritmo de rasterização de círculos baseado na avaliação do ponto médio faria. (Sugestão: escreva a equação do círculo e avalie o valor no ponto m mostrado na figura abaixo. Justifique sua escolha com base neste valor).



Resp.:



$$F(x, y) = x^2 + y^2 - 64$$

$$F(\mathbf{m}) \begin{cases} > 0 \rightarrow escolha & \textbf{se} \\ \leq 0 \rightarrow escolha & \textbf{e} \end{cases}$$

$$F(\mathbf{m}_1) = F(1,7.5) = -6.75 \leq 0 \rightarrow escolha \quad \textbf{e}$$

$$F(\mathbf{m}_2) = F(2,7.5) = -3.75 \leq 0 \rightarrow escolha \quad \textbf{e}$$

$$F(\mathbf{m}_3) = F(3,7.5) = 1.25 \geq 0 \rightarrow escolha \quad \textbf{se}$$

Logo os três próximos *pixels* são: (1,8), (2,8) e (3,7)

3. Determine os vetores da base do sistema de coordenadas do olho, xe ye ze, correspondente a chamada:

```
gluLookAt(10,10,10,  0,0,0, 0,0,1);
```

Resp:

$$\mathbf{z}_e = \frac{1}{\|\mathbf{eye} - \mathbf{center}\|}(\mathbf{eye} - \mathbf{center}) = \frac{1}{10\sqrt{3}}\begin{pmatrix}10\\10\\10\end{pmatrix} = \frac{1}{\sqrt{3}}\begin{pmatrix}1\\1\\1\end{pmatrix}$$

$$\mathbf{x}_e = \frac{1}{\|\mathbf{up} \times \mathbf{z}_e\|}(\mathbf{up} \times \mathbf{z}_e) = unit\left(\begin{pmatrix}0\\0\\1\end{pmatrix} \times \begin{pmatrix}1\\1\\1\end{pmatrix}\right) = unit\begin{pmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\0 & 0 & 1\\1 & 1 & 1\end{pmatrix} = unit\begin{pmatrix}-1\\1\\0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}-1\\1\\0\end{pmatrix}$$

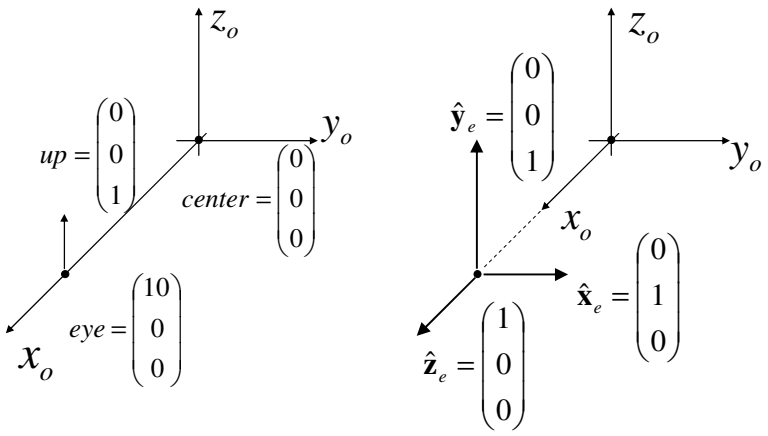
$$\mathbf{y}_e = \mathbf{z}_e \times \mathbf{x}_e = \frac{1}{\sqrt{6}}\begin{pmatrix}1\\1\\1\end{pmatrix} \times \begin{pmatrix}-1\\1\\0\end{pmatrix} = \frac{1}{\sqrt{6}}\begin{pmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\1 & 1 & 1\\-1 & 1 & 0\end{pmatrix} = \frac{1}{\sqrt{6}}\begin{pmatrix}-1\\-1\\2\end{pmatrix}$$

4. Determine a matriz de modelagem e visualização ("modelview") que resulta do bloco de programa mostrado abaixo.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(10,0,0, 0,0,0, 0,0,1);
```

Resp.:

A determinação do sistema do **eye** pode ser feita visualmente diretamente na figura.



$$\mathbf{L}_{at} = \mathbf{RT} = \begin{bmatrix} x_{ex} & x_{ey} & x_{ez} & 0 \\ y_{ex} & y_{ey} & y_{ez} & 0 \\ z_{ex} & z_{ey} & z_{ez} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{L}_{at} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5. Determine a matriz de projeção que resulta do bloco de programa mostrado abaixo.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60,800/600,1,11);
```

Resp.:

Partindo da matriz do glFrustum $\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$,

Devemos substituir *right*, *left*, *top*, *bottom* por *fovy* e *aspect*. O ângulo *fovy* é definido sobre as variáveis top e bottom. E *right* e *left* são encontrados pelo *aspect*.

$$\text{fovy} = 2\tan^{-1}\left(\frac{\text{top} - \text{bottom}}{2\text{near}}\right)$$

$$top - bottom = 2.near.\tan\left(\frac{fovy}{2}\right)$$

$$\text{aspect} = \frac{\text{right} - \text{left}}{\text{top} - \text{bottom}}$$

$$\text{right} - \text{left} = \text{aspect}.(\text{top} - \text{bottom}) = 2.\text{aspect}.\text{near}.\tan\left(\frac{\text{fovy}}{2}\right)$$

Sendo o *fovy* simétrico, temos que $\text{top} = -\text{bottom}$ e $\text{right} = -\text{left}$.

A matriz resultante fica assim

$$\text{M} = \begin{bmatrix} \frac{2\,\text{near}}{2.\text{aspect}.\text{near}.\tan\left(\frac{\text{fovy}}{2}\right)} & 0 & \frac{\text{right} - \text{right}}{2.\text{aspect}.\text{near}.\tan\left(\frac{\text{fovy}}{2}\right)} & 0 \\ 0 & \frac{2\,\text{near}}{2.\text{near}.\tan\left(\frac{\text{fovy}}{2}\right)} & \frac{\text{top} - \text{top}}{2.\text{near}.\tan\left(\frac{\text{fovy}}{2}\right)} & 0 \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2\,\text{far}\,\text{near}}{\text{far} - \text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

ou

$$\text{M} = \begin{bmatrix} \frac{1}{\text{aspect}.\tan\left(\frac{\text{fovy}}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\text{fovy}}{2}\right)} & 0 & 0 \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{-2\text{far} * \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$M = \begin{bmatrix} \frac{\sqrt{3}}{4} & 0 & 0 & 0 \\ 0 & \sqrt{3} & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

6. Considere o algoritmo de recorte de Cohen-Sutherland aplicado a segmentos de retas em 2D onde os pontos são classificados em códigos com 4 dígitos na seguinte ordem: **abaixo; acima; à esquerda; à direita** (assuma x orientado da esquerda para a direita, y de baixo para cima). Considere ainda na tabela abaixo as coordenadas dos vértices de 4 segmentos de retas e o retângulo de recorte definido por: $x_{min} = 2$, $x_{max} = 5$, $y_{min} = -1$, $y_{max} = 5$. Complete a tabela com os códigos de cada vértice dos segmentos e responda na coluna direita da tabela qual das condições se aplica a cada um dos segmentos:

   descarte = está todo fora, pode ser descartado;
   desenhe = está todo dentro, pode ser desenhado;
   recorte = está parcialmente dentro, deve ser recortado e desenhado;
   indefinido = não é possível decidir só com base nos códigos.

Vértice $\mathbf{p}_1$	Vértice $\mathbf{p}_2$	Código $\mathbf{p}_1$	Código $\mathbf{p}_2$	Condição do segmento $\mathbf{p}_1\mathbf{p}_2$
(1, 6)	(8, 7)	0110	0101	descarte
(3, 4)	(4, 0)	0000	0000	desenhe
(0, 4)	(1, 7)	0010	0110	descarte
(1, 2)	(7, 7)	0010	0101	indefinido

   Resp:

   Segmento de reta de (1,6) a (8,7):
   $\mathbf{p}_1$: $x < 2 \Rightarrow$ à esquerda, $y > 5 \Rightarrow$ à cima, então o código é 0110
   $\mathbf{p}_2$: $x > 5 \Rightarrow$ à direita, $y > 5 \Rightarrow$ à cima, então o código é 0101
   o segmento deve ser **descartado** por estar **acima** e 0110 & 0101 = 0100 ≠ 0.

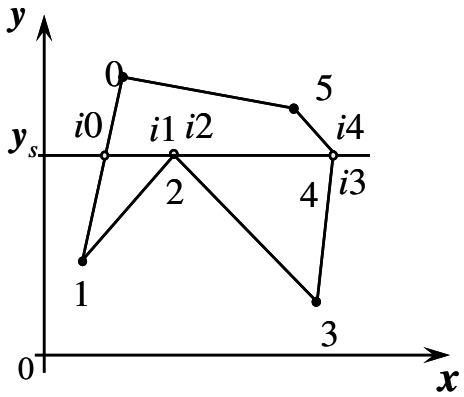
   Segmento de reta de (3,4) a (4,0):
   $\mathbf{p}_1$: $x \in [2,5]$, $y \in [-1,5] \Rightarrow$ dentro, então o código é 0000
   $\mathbf{p}_2$: $x \in [2,5]$, $y \in [-1,5] \Rightarrow$ dentro, então o código é 0000
   o ponto deve ser **desenhado** por estar **dentro** e 0000 && 0000.

   Segmento de reta de (0,4) a (1,7):
   $\mathbf{p}_1$: $x < 2 \Rightarrow$ à esquerda, $y \in [-1,5] \Rightarrow$dentro, então o código é 0010
   $\mathbf{p}_2$: $x < 2 \Rightarrow$ à esquerda, $y > 5 \Rightarrow$ à cima, então o código é 0110
   o segmento deve ser **descartado** por estar **à esquerda** e 0010 & 0110 = 0010 ≠ 0.

   Segmento de reta de (1,2) a (7,7):
   $\mathbf{p}_1$: $x < 2 \Rightarrow$ à esquerda, $y \in [-1,5] \Rightarrow$dentro, então o código é 0010

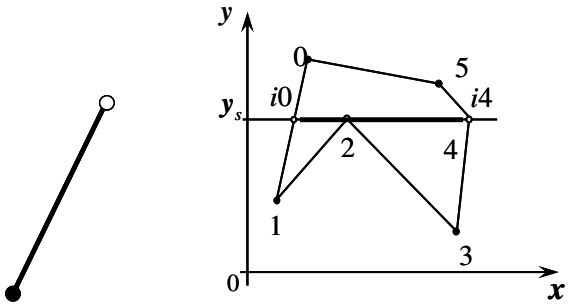
$\mathbf{p}_2$: $x > 5 \Rightarrow$ à direita, $y > 5 \Rightarrow$ à cima, então o código é 0101
o segmento está **indefinido**. 0010 & 0101 não é $\neq 0$ e eles não são ambos 0.
Esta é uma situação que o somente com o código não podemos afirmar se ele está fora ou parcialmente dentro, portando a solução é: indefinido.

7. Um dos problemas do algoritmo de preenchimento de polígonos é o cálculo da interseção da linha de rastreamento (*scan line*) com a fronteira do polígono representada por uma lista de arestas que vão de um vértice a outro (os polígono é definido como uma lista circular de vértices). A figura abaixo mostra uma linha de rastreamento com este problema. Explique como podemos evitar este problema no algoritmo e exemplifique como o algoritmo resolveria o caso abaixo.



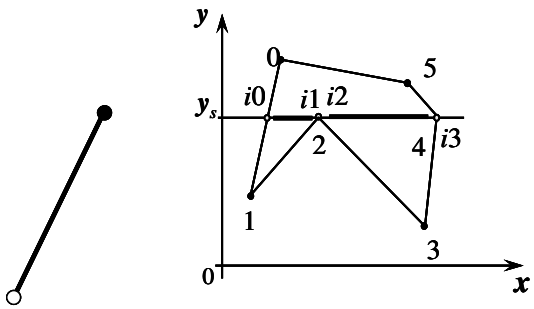
Resposta:

Para evitar de contar 2 vezes quando a situação é de entrada ou saída e para contar 0 ou 2 vezes quando o vértice apenas tangencia a linha de rastreio adotamos como critério que cada segmento é fechado no vértice inferior e aberto no superior. Ou seja:



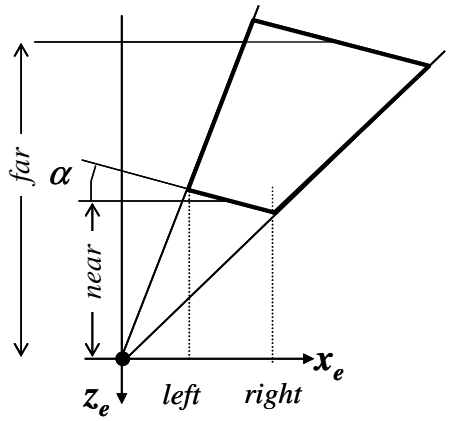
Neste caso teríamos as interseções 0 e 4.

Poderíamos ter também optado por considerar fechado o segmento no topo e aberto na base. Neste caso teríamos as interseções 0, 1, 2 e 3, que resultaria correto da mesma maneira.


y
0
5
y_s
i0
i1 i2
2
4
i3
1
3
0
x

Exercícios

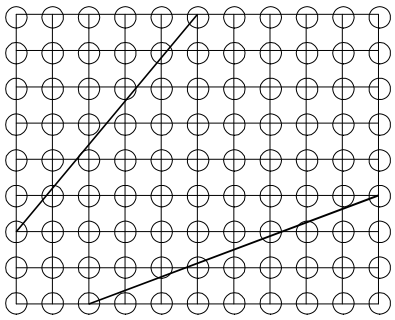
1. Derive a matriz de projeção de uma câmera semelhante a definida pela `glFrustum` (Fig. 7.8) exceto pelo fato dos planos *near* e *far* estarem inclinados de um ângulo α conforme mostra a figura abaixo.



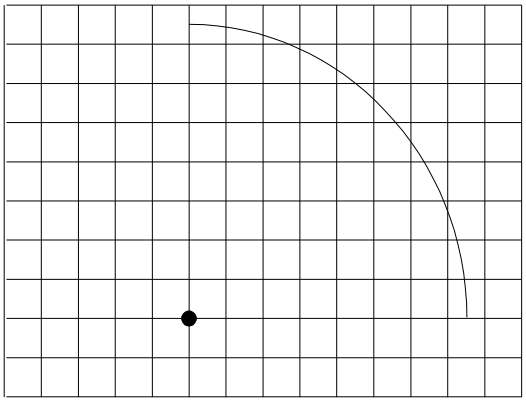
2. Faça uma família de gráficos mostrando a relação entre o valor de z no mapa de profundidade e o valor de z no sistema de coordenadas do olho quando o programa utiliza a função:
```
glPerspective( 60, 4.0/3,  n , f );
```
para os valores de f iguais a 10., 100., 1000, e valores de n iguais a 0.01, 0.1 e 1.0. Quando temos o problema de *alias* de profundidade? Como evitá-lo?

3. Determine os vetores da base do sistema de coordenadas do olho, $\boldsymbol{x_e}\ \boldsymbol{y_e}\ \boldsymbol{z_e}$, correspondente a chamada:
```
gluLookAt(10,10,10,  0,0,0, 0,0,1);
```
Determine também a matriz $\mathbf{L}_{at}$ correspondente.

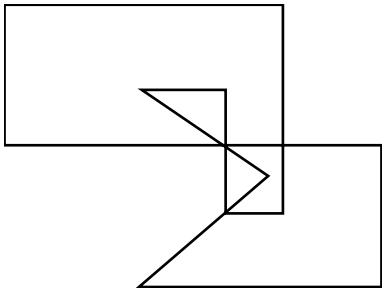
4. Seguindo o critério geométrico do algoritmo de Bresenham, determine quais fragmentos (identificados com os círculos) são gerados no rastreio das duas linhas mostradas na figura abaixo.



5. Marque na figura os fragmentos que um algoritmo de rastreio de círculos, seguindo o critério de Breseham, deveria gerar para o arco mostrado na figura. Explique o critério adotado.

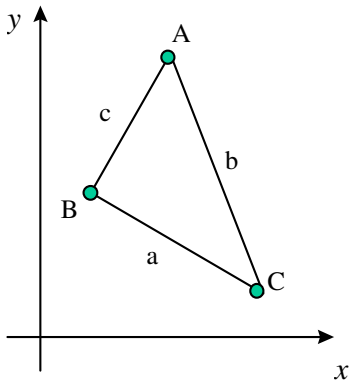


6. Faça uma hachura na região interior, segundo a regra par-ímpar (*even-odd rule*), do polígono cuja fronteira está mostrada na figura abaixo.



7. Determine, segundo o algoritmo de preenchimento de triângulos apresentado acima, quem são os vértices A, B e C do triângulo (20, 120), (200,5), (80,300). Determine também a posição relativa do vértice B segundo o critério da Fig. 7.43, com base no produto vetorial indicado.

8. Os algoritmos de preenchimento de polígonos usam um princípio de coerência para calcular as coordenadas x do início e do fim de cada linha de rastreio (*scan line*). Determine os incrementos dx_a, dx_b e dx_c do triângulo mostrado na figura abaixo, assumindo que o rastreio se dê de cima para baixo (dy=-1).



	x	*y*
A	15	25
B	5	10
C	20	5

9. A tabela abaixo mostra os códigos do algoritmo de Cohen-Suterland dos pontos $\mathbf{p_1}$ e $\mathbf{p_2}$ de diversos segmentos de reta para uma janela de recorte retangular e alinhada

com os eixos. Os códigos de quatro bits significam que o ponto está à cima, abaixo, à esquerda, e a direita, respectivamente. Ou seja, um ponto de código 0010 significa que o ponto está a esquerda da janela e não está nem acima nem abaixo. Responda na coluna direita da tabela qual das condições se aplica a cada um dos segmentos:

[A] está fora, pode ser descartado;
[B] está dentro, pode ser desenhado;
[C] está parcialmente dentro, deve ser clipado e desenhado;
[D] não posso dizer se está fora ou parcialmente dentro;
[E] não pode existir um segmento com este código.

Códigos		RESPOSTAS
P0	P1	(A|B|C|D|E)
0010	0000	
1100	1000	
1010	0010	
0101	1010	
0100	0001	
0000	0000	

10. Considere o algoritmo de recorte de Cohen-Sutherland aplicado a segmentos de retas em 3D onde os pontos são classificados em códigos com 6 dígitos na seguinte ordem: atrás; à frente; abaixo; acima; à esquerda; à direita (assuma x orientado da esquerda para a direita, y de baixo para cima e z de trás para a frente). Considere ainda na tabela abaixo as coordenadas dos vértices de 4 segmentos de retas e o paralelepípedo de recorte definido por: $x_{min} = 2$, $x_{max} = 6$, $y_{min} = -1$, $y_{max} = 5$, $z_{min} = 0$, $z_{max} = 8$. Complete a tabela com os códigos de cada vértice dos segmentos e responda na coluna direita da tabela qual das condições se aplica a cada um dos segmentos:

descarte = está todo fora, pode ser descartado;
desenhe = está todo dentro, pode ser desenhado;
recorte = está parcialmente dentro, deve ser recortado e desenhado;
indefinido = não é possível decidir só com base nos códigos.

Vértice $\mathbf{p}_1$	Vértice $\mathbf{p}_2$	Código $\mathbf{p}_1$	Código $\mathbf{p}_2$	Condição do segmento $\mathbf{p}_1\mathbf{p}_2$
(1, 6, 4)	(8, 7, 3)			
(3, 4, 7)	(4, 0, 2)			
(0, 4, 6)	(1, 7, 6)			
(1, 2, 9)	(7, 7,-1)			

11. Refaça o recorte dos segmentos de reta do problema anterior calculando os valores t_s e t_e do algoritmo de Cyrus-Beck.

12. Determine a expressão que calcula o parâmetro *t* do algoritmo de Cyrus-Beck para o plano x+2y+3z=6 para o segmento de reta $\mathbf{p_0p_1}$, $\mathbf{p_0}=(x_0,y_0,z_0)^T$ e $\mathbf{p_1}=(x_1,y_1,z_1)^T$. Supondo que a normal aponte para fora, determine também a expressão que determina se o ponto correspondente ao parâmetro *t* está entrando ou saindo da região delimitada pelo plano.

13. Seguindo a lógica do algoritmo de recorte de polígonos de Sutherland-Hodgman, determine a lista de vértices que passa de uma etapa para outra para o polígono 0,1,2,3,4,5,6 e 7 da figura abaixo. Responda na ordem indicada na tabela-resposta. Ou seja, coloque na coluna "Esquerda" o resultado após o recorte na fronteira esquerda, coloque na coluna "Direita" o resultado após o recorte nas fronteiras esquerda seguida da direita e assim por diante.

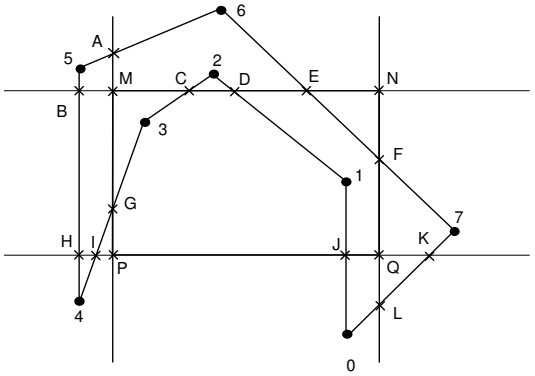


Tabela-resposta:

Entrada	Esquerda	Direita	Abaixo	Acima
0				
1				
2				
3				
4				
5				
6				
7				

14. Explique porque o algoritmo de recorte de primitivas do OpenGL não é implementado após a transformação de projeção, quando o volume de recorte é um cubo do R^3. Dê um exemplo do tipo de problema que ocorreria.