UNIPAC
UNIVERSIDADE PRESIDENTE ANTÔNIO CARLOS
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO E
COMUNICAÇÃO SOCIAL

CURSO DE CIÊNCIA DA COMPUTAÇÃO

Darlan Ferreira Maia

BIBLIOTECA GRÁFICA MATRIX

BARBACENA
DEZEMBRO DE 2004

Darlan Ferreira Maia

BIBLIOTECA GRÁFICA MATRIX

Trabalho de conclusão de curso apresentado
à Universidade Presidente Antônio Carlos,
como requisito parcial para obtenção do grau
de Bacharel em Ciência da Computação.

ORIENTADOR: Prof. Mestre Reinaldo Silva Fortes

BARBACENA
DEZEMBRO DE 2004

Darlan Ferreira Maia

BIBLIOTECA GRÁFICA MATRIX

Trabalho de conclusão de curso apresentado
à Universidade Presidente Antônio Carlos,
como requisito parcial para obtenção do grau
de Bacharel em Ciência da Computação.

Aprovado em ____ / ____ / _____

BANCA EXAMINADORA

Prof. Mestre Reinaldo Silva Fortes (Orientador)
Universidade Presidente Antônio Carlos

Prof. Cesário José Ferreira
Universidade Presidente Antônio Carlos

Prof. Leandro Alves Rezende
Universidade Presidente Antônio Carlos

Agradeço a Deus e a todos aqueles que de alguma forma colaboraram para que eu atingisse meu objetivo.

RESUMO

A partir dos anos 80, com o surgimento dos monitores CRT, a Computação Gráfica desenvolveu-se rapidamente. Como o interesse em Computação Gráfica cresceu, passou a ser importante desenvolver aplicações que pudessem funcionar em diferentes plataformas. Um padrão para desenvolvimento de aplicativos gráficos facilita esta tarefa, eliminando a necessidade de escrever código para um *driver* gráfico distinto para cada plataforma na qual a aplicação deve funcionar. Neste cenário, a Biblioteca Gráfica Matrix aparece como uma alternativa para um padrão de desenvolvimento de aplicativos gráficos. Este trabalho visa a criação de uma estrutura para esta biblioteca, que consiste basicamente da definição, descrição e especificação de um conjunto de funções gráficas para a Matrix. Visando validar a estrutura proposta, um protótipo operacional foi desenvolvido.

Palavras-chave: Computação Gráfica, Biblioteca Gráfica, Matrix, estrutura.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

Até o início dos anos 80, a Computação Gráfica era uma disciplina restrita e altamente especializada. Devido principalmente ao alto custo do *hardware*, poucos aplicativos utilizavam gráficos. O advento dos computadores pessoais de baixo custo, como o IBM-PC e o Apple Macintosh, com monitores CRT (Tubo de Raios Catódicos), popularizou o uso de gráficos na interação usuário-computador. [1]

Os monitores CRT de baixo custo possibilitaram o desenvolvimento de inúmeros aplicativos de baixo custo e de fácil utilização, que dispunham de interfaces gráficas: processadores de texto, planilhas, aplicativos de desenho, jogos, etc.

A Computação Gráfica desenvolveu-se de modo bem diverso: de simples aplicativos gráficos para computadores pessoais a aplicativos de modelagem e de visualização em *workstations* (estações de trabalho) e supercomputadores. Como o interesse em Computação Gráfica cresceu, passou a ser importante desenvolver aplicações que pudessem funcionar em diferentes plataformas. Um padrão para desenvolvimento de aplicativos gráficos facilita esta tarefa, eliminando a necessidade de escrever código para um *driver* gráfico distinto para cada plataforma na qual a aplicação deve funcionar.

1.2 Contexto do Trabalho

Para padronizar a construção de aplicativos que utilizam recursos gráficos e torná-los o mais independentes possível das máquinas, e portanto facilmente portáveis, foram desenvolvidas as chamadas Bibliotecas Gráficas.

Dentre as Bibliotecas Gráficas que já foram desenvolvidas, vale ressaltar as seguintes:

- XWindow: se tornou padrão para o desenvolvimento de interfaces gráficas 2D em *workstations* UNIX. Uma outra facilidade do XWindow é o uso de redes de computadores, onde um aplicativo pode funcionar em uma *workstation* e ler a entrada e ser exibido em outra *workstation,* mesmo de outro fabricante;

- GKS (*Graphical Kernel System*): padronizado pela ANSI (American National Standards Institute) e ISO (International Organization for Standardization) em 1985. Suporta um conjunto de primitivas gráficas interrelacionadas, tais como: desenho de linhas, polígonos, caracteres, etc., bem como seus atributos. Mas não suporta agrupamentos de primitivas hierárquicas de estruturas 3D e mapeamento de textura;

- PHIGS (*Programmer's Hierarchical Interactive Graphics System*): baseado no GKS, PHIGS é um padrão ANSI. PHIGS e seu descendente, PHIGS+, provêem meios para desenhar e manipular objetos 3D encapsulando descrições de objetos e atributos em uma *display list*. A *display list* é utilizada quando o objeto é exibido ou manipulado. Uma vantagem é a possibilidade de descrever um objeto complexo uma única vez mesmo exibindo-o várias vezes. Isto é especialmente importante se o objeto a ser exibido deve ser transmitido por uma rede de computadores. Uma desvantagem da *display list* é a necessidade de um esforço considerável para especificar novamente um objeto que está sendo modelado interativamente pelo usuário. Uma desvantagem do PHIGS e PHIGS+ é que eles não têm suporte a recursos avançados como mapeamento de textura;

- PEX: extensão do XWindow para o PHIGS, de modo que o XWindow pudesse desenhar e manipular objetos 3D. Entre outras extensões, PEX soma *modo imediato* ao PHIGS, assim um objeto pode ser exibido durante a sua definição sem a necessidade da *display list*. PEX também não suporta recursos avançados e só está disponível aos usuários do XWindow.

Com a evolução das Bibliotecas Gráficas surgiu o conceito de API (Application Programming Interface) onde é feita uma "tradução" da comunicação do *software* com o *hardware.* Assim, o programador escreve código para o sistema operacional, onde um *driver* traduzirá o que o aplicativo quer e acessará o *hardware* de acordo com o que foi pedido. A vantagem desta técnica é que o programador não precisa conhecer todas as placas de vídeo existentes no mercado, porque o *driver* converte o que ele quer em um comando compatível com a placa instalada no computador. Alguns exemplos de API's são: [2]

- OpenGL: provê características avançadas e pode ser utilizado em *modo imediato* ou com *display list*. É um padrão relativamente novo, sua primeira versão é de 1992, e é baseado na biblioteca gráfica GL (Graphics Library) das *workstations* IRIS da Silicon Graphics. Atualmente um consórcio de indústrias é responsável pelo gerenciamento da evolução do OpenGL. Existe uma implementação livre do OpenGL com código fonte disponível, conhecida com MesaGL ou Mesa3D. Consiste em um conjunto de 150 comandos, entre procedimentos e funções, que permitem a um programador especificar os objetos e as operações que os envolvem produzindo imagens de alta qualidade. As especificações do OpenGL não descrevem as interações entre OpenGL e o sistema de janelas utilizado como: Windows, XWindow, etc. Assim, tarefas comuns em uma aplicação, tais como criar janelas gráficas, gerenciar eventos provenientes de mouse e teclado, e apresentação de menus ficam a cargo de bibliotecas próprias de cada sistema operacional. Pode-se utilizar a biblioteca GLUT (OpenGL ToolKit) para gerenciamento de janelas. O OpenGL provê controle direto sobre operações gráficas fundamentais em 2D e 3D, incluindo a especificação de parâmetros como matrizes de transformação e coeficientes de iluminação, métodos de *antialiasing,* operações sobre pixels. Também incorpora um grande conjunto de funções de renderização de texturas, de efeitos especiais, e de outras poderosas funções de visualização. Mas não provê mecanismos para descrever ou modelar objetos geométricos complexos. Diante das funcionalidades providas, esta Biblioteca Gráfica tem se tornado um padrão amplamente adotado na indústria de desenvolvimento de aplicações gráficas. Este fato tem sido encorajado também pela facilidade de aprendizado, pela estabilidade das rotinas, pela boa documentação disponível e pelos resultados visuais consistentes para qualquer sistema de exibição concordante com este padrão; [3]

- DirectX: criado pela Microsoft em 1995, logo após o lançamento do Windows 95 em resposta a pergunta de muitos desenvolvedores de jogos, sobre como seria desenvolver jogos para o Windows 95. Disponibilizado como um kit de desenvolvimento de software (SDK ou Software Development Kit), ele foi comprado originalmente de uma companhia da cidade de Londres chamada RenderMorphics e distribuído como DirectX 2. Porém, o DirectX 2 era mal projetado, com pouca documentação, tornando a tarefa dos programadores muito difícil. Já o DirectX 3, era provavelmente a primeira distribuição "séria" da Microsoft, e que realmente iria começar a mudar o mundo dos jogos. Sendo que a Microsoft era a maior empresa de *software* do mundo, e que 90% dos computadores no mundo usavam seu sistema operacional, ficou tudo mais fácil. Os fabricantes de *hardware* perceberam rapidamente que seguir o caminho da Microsoft era a coisa prudente a fazer, e todos começaram a produzir

drivers para o DirectX. O DirectX foi projetado primeiramente para jogos, mas atualmente pode ser usado no desenvolvimento de qualquer aplicação gráfica. [2]

1.3 Objetivos

Este trabalho consiste no desenvolvimento de uma Biblioteca Gráfica doravante denominada Matrix, que visa em sua essência a reunião de funções para o desenho e manipulação de objetos gráficos.

A Matrix deve ser simples e de fácil utilização, para proporcionar o desenvolvimento de aplicações gráficas. É necessário que o aplicativo desenvolvido com ela seja portável, podendo funcionar em diferentes plataformas. Além destas questões, deve-se estruturar a Matrix para uma fácil manutenção e atualização no futuro.

1.3.1 Objetivos específicos

- Estudar técnicas da Computação Gráfica para:

a) desenhar uma reta;
b) desenhar uma circunferência;
c) desenhar uma elipse;
d) desenhar um polígono 2D;
e) preencher uma circunferência;
f) preencher uma elipse;
g) preencher um polígono 2D;
h) *anti-aliasing*;
i) transformações geométricas em 2D;
j) desenhar um polígono 3D;
k) transformações geométricas em 3D;
l) determinação de superfície visível; e,
m) mapeamento de *textura*.

- Estudar um método para a leitura de imagens no formato PCX.
- Estudar a estrutura e as características das Bibliotecas Gráficas:

a) OpenGL versão 1.2;
b) DirectX versão 8.0; e,
c) Allegro versão 3.12.

- Especificar uma estrutura para a Matrix, definindo:

a) quais informações obter sobre a placa de vídeo;

b) o que é modo gráfico;

c) como desenhar um *pixel* na tela do monitor;

d) as resoluções de vídeo suportadas; e,

e) os números de cores suportados.

- Implementar um prótotipo operacional que valide a estrutura proposta:

a) definindo as linguagens de programação mais adequadas para sua implementação;

b) descrevendo as funções implementadas; e,

c) descrevendo o funcionamento da Matrix.

1.4 Organização da Monografia

O Capítulo 2, intitulado "Técnicas da Computação Gráfica", aborda o estudo de técnicas de desenho e manipulação de objetos gráficos. Também é visto neste capítulo, conceitos relativos à Computação Gráfica.

Tendo em vista que este trabalho consiste no desenvolvimento de uma Biblioteca Gráfica, o Capítulo 3, intitulado "Bibliotecas Gráficas", aborda o estudo da estrutura e das características das bibliotecas já desenvolvidas OpenGL, DirectX e Allegro.

O Capítulo 4, intitulado "Matrix", apresenta a estrutura da Biblioteca Gráfica definida neste trabalho, que consiste basicamente dos passos necessários para utilização de gráficos em computador. E também, é apresentada uma implementação de um protótipo operacional da Matrix, realizada para validar a estrutura proposta.

No Capítulo 5, intitulado "Considerações Finais", os resultados alcançados neste trabalho são apresentados e discutidos. Além disso, possíveis direcionamentos para trabalhos futuros e a conclusão do trabalho são apresentados.

No Apêndice A, intitulado “Imagens PCX”, o estudo de um método sobre como ler imagens no formato PCX de um arquivo é apresentado.

O Anexo A, intitulado “Material Prático”, é um CD-ROM que contém o arquivo texto desta monografia e os arquivos do protótipo operacional.

2 TÉCNICAS DA COMPUTAÇÃO GRÁFICA

2.1 Conceitos

Para o melhor entendimento deste capítulo e dos seguintes, serão apresentados aqui alguns conceitos, são eles: [1]

a) *Software*: programa de computador;
b) *Hardware*: parte física de um computador;
c) 2D: bidimensional;
d) 3D: tridimensional;
e) *Pixel*: ponto colorido na tela do monitor;
f) *Textura*: imagem pré-digitalizada contida em um arquivo;
g) *Texel*: estrutura que armazena a posição e a cor de um ponto de uma *textura*;
h) Tabela *Hashing*: vetor de dados, onde cada posição do vetor armazena uma lista encadeada;
i) Pilha: estrutura que pode ser representada como um vetor, onde o primeiro elemento a ser colocado no vetor é o último a sair, pode-se fazer uma analogia com uma pilha de pratos; e,
j) *Driver*: arquivo de um sistema operacional que recebe um pedido de um *software* e "traduz" este pedido em um comando, enviando depois este comando para o *hardware* executá-lo. [2]

2.2 Primitivas Gráficas

Primitivas gráficas são objetos gráficos elementares como retas, circunferências, elipses, polígonos. O traçado de primitivas gráficas é feito por algoritmos que determinam na

matriz de *pixels* da tela do monitor quais *pixels* devem ser alterados de forma a simular a aparência do objeto gráfico desejado. [1]

2.3 Algoritmo do "Ponto-Médio" para Retas

Foi proposto por Bresenham em 1965. É um algoritmo clássico que utiliza apenas variáveis inteiras. O algoritmo permite fazer os cálculos de (x_{i+1}, y_{i+1}) incrementalmente, usando os cálculos já feitos para (x_i, y_i).

O algoritmo assume que a inclinação está entre 0 e 1 no 1º octante (outras inclinações podem ser usadas por simetria). O ponto (x_1, y_1) é o inferior esquerdo, e (x_2, y_2) é o superior direito.

Partindo do ponto $P(x_1, y_1)$, a escolha do próximo ponto a ser desenhado é analisando o ponto médio M. Se M está acima da reta, o próximo ponto a ser desenhado é o da direita (E), e se M estiver abaixo da reta, o próximo a ser desenhado é o ponto acima e a direita (NE), conforme apresentado na Figura 2.1.

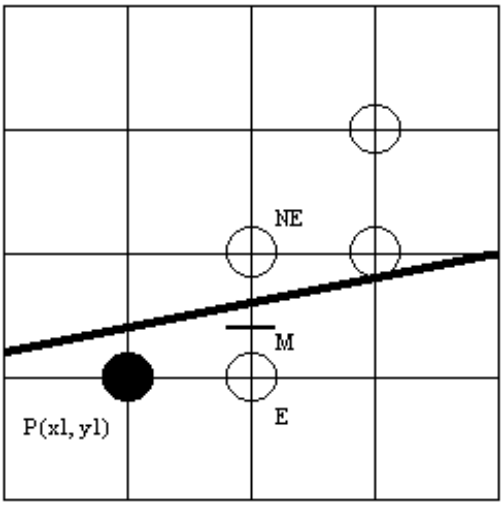

Figura 2.1 – Matriz de pixels para o ponto médio M e as escolhas E e NE.

2.4 Algoritmo do "Ponto-Médio" para Circunferências

A equação da circunferência com centro na origem e raio R, no plano cartesiano é dada por: $x^2 + y^2 = R^2$

Seja a função $F(x, y) = x^2 + y^2 - R^2 = 0$, o valor 0 é sobre a circunferência, positivo fora dela e negativo dentro. Considerando um arco de 45º, onde o ponto inicial é dado por P(x, y) e sendo (x = 0) e (y = R), a escolha do próximo ponto a ser desenhado é analisando o ponto médio M. Se M está dentro da circunferência, o ponto escolhido é o próximo a direita

(E), e se M está fora ou sobre a circunferência, o ponto escolhido é o próximo a direita e abaixo (SE), de acordo com a Figura 2.2.

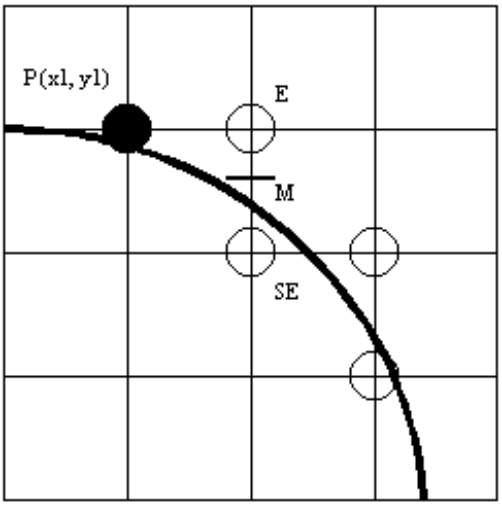

Figura 2.2 – Matriz de pixels para o ponto médio M e as escolhas E e SE.

Como os incrementos são calculados para o 2º octante, deve-se usar simetria de ordem 8 para desenhar os pontos dos demais octantes, formando a circunferência. A Figura 2.3 ilustra esta simetria.

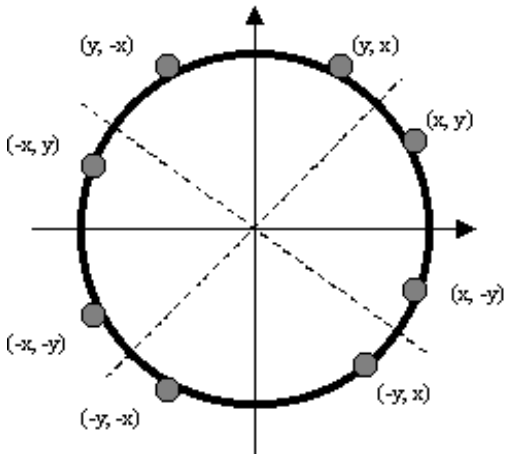

Figura 2.3 – Oito pontos simétricos em uma circunferência.

2.5 Algoritmo do "Ponto-Médio" para Elipses

A equação da elipse centrada em (0, 0) é dada por: $F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$, onde 2a é o comprimento do eixo maior (eixo x) e 2b é o comprimento do eixo menor (eixo y), conforme apresentado na Figura 2.4.

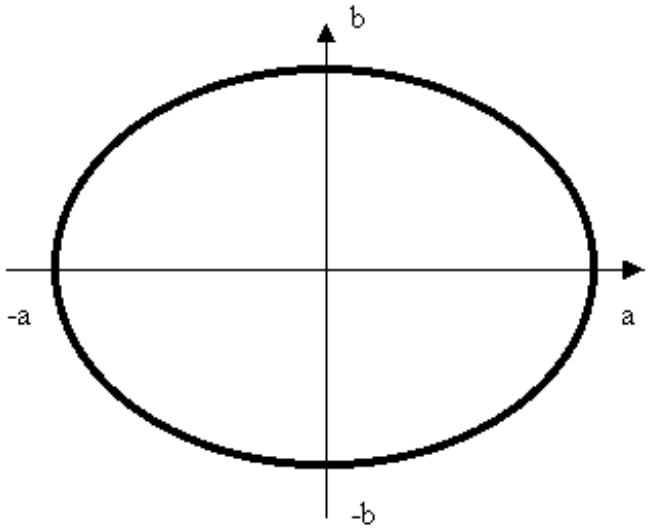

Figura 2.4 – Elipse padrão centrada na origem.

Como no traçado da elipse ocorre mudança na inclinação, divide-se o 1º quadrante em duas regiões: o limite entre as duas regiões é o ponto da curva cuja tangente tem inclinação igual a –1. Assim, enquanto $a^2y > b^2x$ permanece-se na região 1, caso contrário, muda-se para a região 2. A Figura 2.5 ilustra a divisão do quadrante.

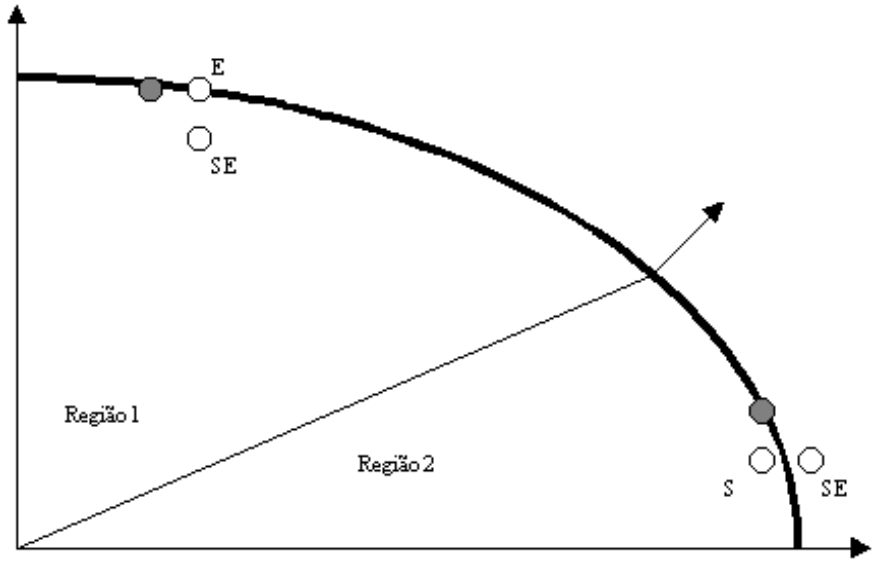

Figura 2.5 – As duas regiões adotadas, definidas pela tangente a 45º.

Seja a função $F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$, o valor 0 é sobre a elipse, positivo fora dela e negativo dentro. Considerando um arco de 45º, onde o ponto inicial é dado por P(x, y) e sendo (x = 0) e (y = b), a escolha do próximo ponto a ser desenhado é analisando o ponto médio M. Na região 1, se M está dentro da elipse, o ponto escolhido é o próximo a direita (E), e se M está fora ou sobre a elipse, o ponto escolhido é o próximo a direita e abaixo (SE). Na região 2, se M está dentro da elipse, escolhe-se (SE), e se M está fora ou sobre a elipse, escolhe-se (S).

Como os incrementos são calculados para o 1º e 2º octantes, deve-se usar simetria de ordem 4 para desenhar os pontos dos demais octantes, formando a elipse.

2.6 Desenho de Polígonos 2D

Para polígonos 2D é criada uma lista de vértices (pontos) e, retas devem ser traçadas de um vértice a outro, de acordo com a Figura 2.6, em uma determinada ordem: [4]

(V_i a V_{i+1}), (V_{i+1} a V_{i+2}), (V_{n-1} a V_n) e (V_n a V_i).

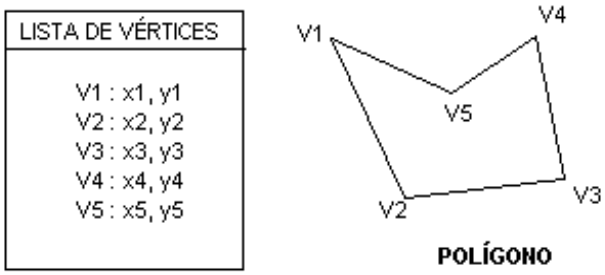

Figura 2.6 – Lista de vértices e polígono de 5 vértices.

2.7 Preenchimento de Primitivas Gráficas

Preencher primitivas gráficas consiste em decidir quais *pixels* devem ser desenhados dentro da primitiva. O preenchimento pode ser feito com uma cor, uma *textura* ou um padrão. Mas neste trabalho será estudo o preenchimento com cor.

2.8 Preenchimento de Circunferências e Elipses

Para preencher uma circunferência ou uma elipse, utiliza-se um processo denominado preenchimento por saturação. A partir de um *pixel* que é desenhado num ponto (x, y) do interior da primitiva e chamado de “gérmen” ou “semente”, verifica-se através da cor da borda e de preenchimento, quais dos 4 *pixels* ao seu redor também fazem parte do interior. Os *pixels* que satisfazerem a essas condições são desenhados e suas coordenadas são inseridas numa pilha (*stack*) para serem considerados novas “sementes”. A Figura 2.7 ilustra este processo. [5]

O processo continua verificando novas “sementes” até que todo o interior esteja preenchido. Nesse processo, *pixels* que já foram desenhados são verificados novamente.

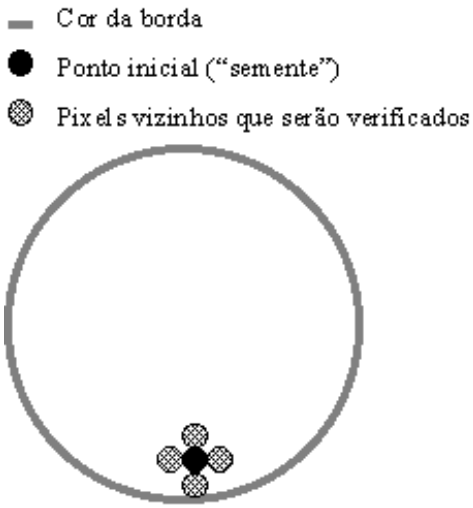

Figura 2.7 – Representação de um preenchimento por saturação.

2.9 Preenchimento de Polígonos 2D

Para polígonos 2D, decidir quais *pixels* devem ser desenhados, consiste em percorrer (varrer, *scan*) sucessivas linhas que interceptam o polígono, desenhando os *pixels* em blocos (*span*) que estão dentro do polígono, da esquerda para a direita. [4]

2.9.1 Preenchimento de retângulos

O retângulo exibe coerência de linha de varredura (*scan-line coherence*), no sentido de que linhas de varredura consecutivas que interceptam o retângulo são idênticas.

Sendo assim, preencher um retângulo consiste em desenhar cada *pixel* de $\mathbf{x}_{min}$ a $\mathbf{x}_{max}$, para cada linha de varredura **y**, que varia de $\mathbf{y}_{min}$ a $\mathbf{y}_{max}$, de acordo com a Figura 2.8.

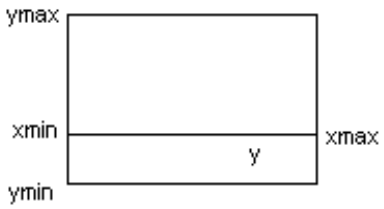

Figura 2.8 – Linha de varredura para o retângulo.

2.9.2 Preenchimento de polígonos 2D arbitrários

O algoritmo de *scan-line* é utilizado para o preenchimento de polígonos arbitrários e, contempla tanto polígonos côncavos quanto polígonos convexos, mesmo que eles tenham auto-interseção e buracos em seu interior. Neste algoritmo, calcula-se blocos de

pixels que estão entre os lados esquerdo e direito do polígono. O bloco extremo é calculado por um algoritmo auxiliar, que através de incrementos calcula a linha de varredura/lado de interseção a partir da interseção com a linha de varredura anterior. Repetindo este processo para cada linha de varredura que intercepta o polígono, o polígono inteiro é analisado. A Figura 2.9 exemplifica uma linha de varredura e suas interseções. [1]

É preciso determinar que *pixels* da linha de varredura estão dentro do polígono, pois não se pode desenhar *pixels* que estejam fora polígono, pois pode-se estar invadindo a região de outro polígono.

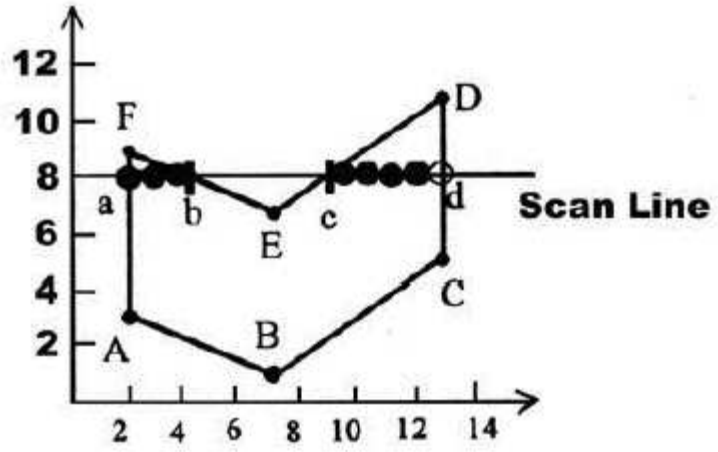

Figura 2.9 – Linha de varredura para um polígono arbitrário e interseções da linha de varredura 8 com os lados FA e CD.

O processo de preencher polígonos pode ser dividido em três passos:

1) Obter a interseção da linha de varredura com todos os lados do polígono;
2) Ordenar os pontos de interseção, do menor valor de **x** para o maior; e,
3) Desenhar os *pixels* entre pares de pontos de interseção do polígono que são internos a ele. Para determinar quais os pares que são internos ao polígono, pode-se usar a regra de Paridade: a paridade inicialmente é par, e a cada interseção encontrada, o *bit* (valor) de paridade é invertido e, o *pixel* é desenhado quando a paridade é ímpar, e não é desenhado quando é par.

Arestas horizontais são desenhadas ou não, de acordo com o *bit* de paridade. Se for aresta do topo não é desenhada.

Alguns polígonos possuem lados muitos próximos criando uma região denominada *sliver*. Essa região é tão estreita que seu interior não contém um bloco de *pixels* para cada linha de varredura, sendo assim, podem existir linhas de varredura com um único *pixel* ou sem nenhum.

O algoritmo de *scan-line* utiliza coerência de aretas, que ocorre quando muitos lados interceptados por uma linha de varredura **y**, também o são pela linha de varredura **y+1**.

Deverá ser criada uma tabela global de arestas denominada ET (*Edge Table*), conforme a Figura 2.10, contendo todas as arestas do polígono em ordem crescente, de sua menor (ymin) coordenada **y** para sua maior (ymax). A ET é uma tabela *hashing* aberta, com **p** posições (*buckets* ou "cestos"), uma para cada linha de varredura. A cada "cesto" **y** é associada uma lista encadeada com os vértices onde $\mathbf{y_{min}} = \mathbf{y}$. As arestas na lista estão ordenadas em ordem crescente da coordenada **x** de sua extremidade inferior (xmin). Cada nó da lista armazena ainda a coordenada $\mathbf{y_{max}}$ da aresta, e o incremento em **x** utilizado para passar para a próxima linha de varredura.

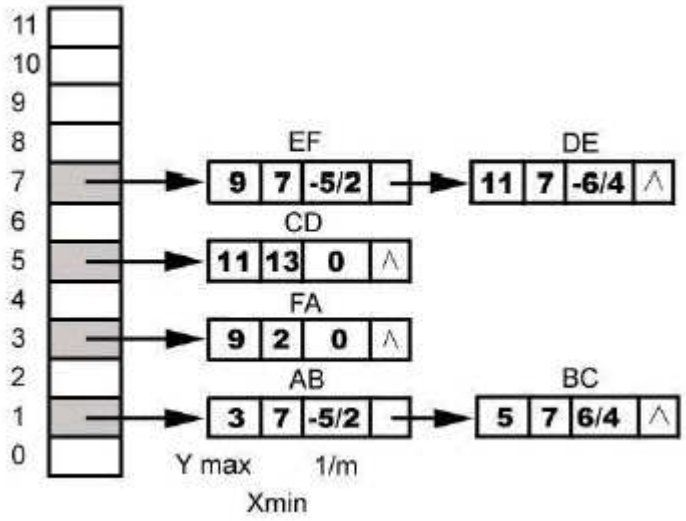

Figura 2.10 – Tabela ET para o polígono da Figura 2.9.

Também deve ser montada uma outra tabela onde, para cada linha de varredura, armazena-se o conjunto de arestas por ela interceptadas numa estrutura de dados denominada AET (*Active-Edge Table*, Tabela de arestas ativas), como ilustra a Figura 2.11. As arestas na AET devem ser ordenadas de acordo com as coordenadas **x** dos pontos de interseção, de forma que se possa preencher os blocos definidos por pares de valores de interseção que definem as extremidades dos blocos. A medida que se passa para a próxima linha de varredura, **y+1**, a AET é atualizada. Primeiramente, arestas que estão na AET mas não são interceptadas por esta próxima linha de varredura, ou seja, aquelas para as quais **ymax = y**, são removidas. Depois, quaisquer novas arestas interceptadas por essa linha, ou seja, aquelas para as quais **ymin = y+1**, são incluídas na AET. Finalmente, para as arestas que ainda estão na AET, os novos valores de **x** das interseções são calculados.

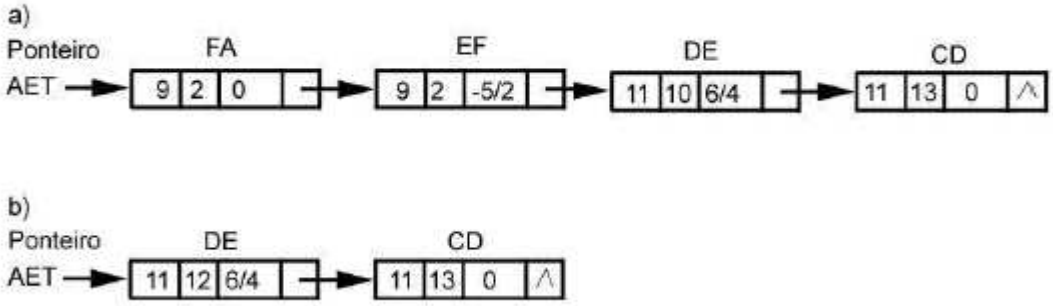

Figura 2.11 – Tabela AET para o polígono da Figura 2.9.

O algoritmo de scan-line utiliza as tabelas apresentadas anteriormente da seguinte maneira:

1) Obtém a menor coordenada **y** armazenada na ET, ou seja, o valor de **y** do primeiro "cesto" não vazio;
2) Inicializa a AET como vazia;
3) Repete os próximos passos até que ET e AET estejam vazias:
4) Transfere do cesto **y** na ET para a AET as arestas cujo $\mathbf{y_{min}} = \mathbf{y}$ (lados que estão começando a ser percorridos), mantendo a AET ordenada em **x**, e retira os lados que possuem $\mathbf{y_{max}} = \mathbf{y_{min}}$ para algum lado que está sendo inserido;
5) Desenha os pixels desejados na linha de varredura **y** usando pares de coordenadas **x** da AET;
6) Retira da AET as entradas para as quais $\mathbf{y} = \mathbf{y_{max}}$ (arestas que não serão atingidas pela próxima linha de varredura);
7) Incrementa **y** de 1 (coordenada da próxima linha de varredura);
8) Para cada aresta não vertical que permanece na AET, atualiza **x** para o novo **y**; e,
9) Como o passo anterior pode ter desordenado a AET, reordena a AET.

2.10 Anti-Aliasing

As retas desenhadas utilizando-se o algoritmo da seção 2.2, e que não são horizontais ou verticais, podem apresentar uma aparência "serrilhada" ou "efeito escada", como ilustra a Figura 2.12. Este efeito indesejável é ocasionado pela técnica do algoritmo para desenho de retas, que escolhe o *pixel* a ser desenhado sem levar em consideração a consequência dessa escolha na aparência da reta. [6]

Então, *anti-aliasing* consiste na aplicação de uma técnica que reduz ou elimina o efeito mencionado anteriormente.

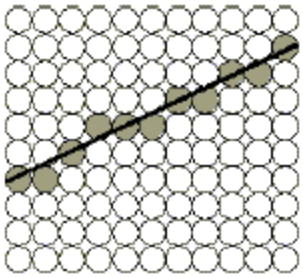

Figura 2.12 – Reta com aparência “serrilhada”.

Apesar de uma primitiva ideal, como uma reta, ter espessura zero, a reta desenhada tem espessura não nula, pois ocupa uma área finita da tela do monitor. Assim, pode-se pensar em uma reta como um retângulo com uma certa espessura que cobre uma região da matriz de *pixels*, como ilustrado na Figura 2.13.

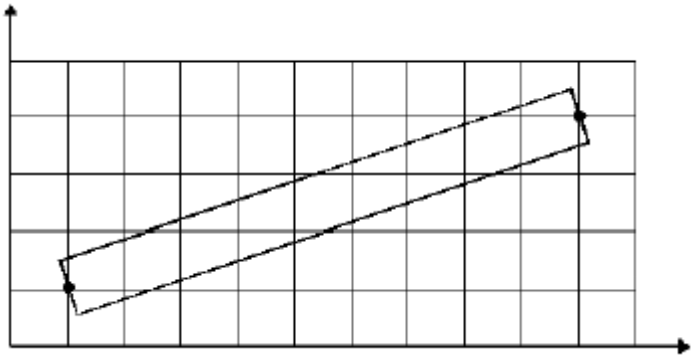

Figura 2.13 – Reta definida com uma espessura diferente de zero.

A técnica de *anti-aliasing* que será apresentada aqui é denominada amostragem por áreas não ponderada (*unweighted area sampling*). De acordo com esta técnica, deve-se atribuir diferentes intensidades de cor a cada *pixel* interceptado pela área coberta pelo retângulo. *Pixels* são tratados como “quadrados” com centro nas interseções da matriz de *pixels*. Usando a cor cinza como exemplo, se o retângulo cobre todo o *pixel*, ele recebe a intensidade máxima da cor, e se cobre parte do pixel, ele recebe um tom de cinza cuja intensidade é proporcional à parte coberta, conforme apresentado na Figura 2.14. O objetivo desta técnica é “suavizar” a definição da reta, melhorando sua aparência visual quando observada à distância.

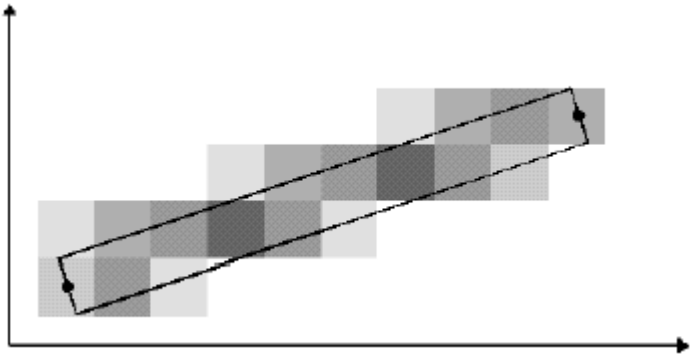

Figura 2.14 – A intensidade da cor do pixel é proporcional à parte coberta.

2.11 Transformações Geométricas em 2D

Consistem de operações para manipular objetos, onde a Translação altera a posição do objeto, a Rotação através de um ângulo muda a visão de um objeto e o tamanho de um objeto é alterado pela Escala. [7]

2.11.1 Translação

Pode-se efetuar a translação de pontos no plano XY adicionando-se quantidades inteiras às suas coordenadas. Assim, cada ponto P(x, y) pode ser movido por **dx** unidades em relação ao eixo **x**, e por **dy** unidades em relação ao eixo **y**, como mostra a Figura 2.15.

Logo, o ponto P'(x', y'), pode ser escrito como:

$$x' = x + dx \qquad y' = y + dy$$

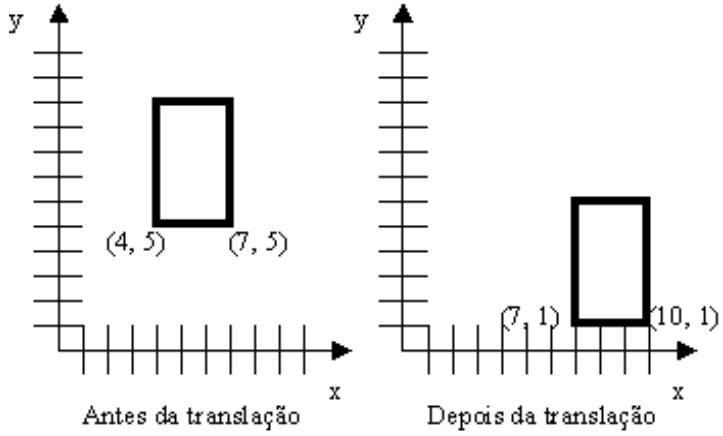

Figura 2.15 – Translação de um objeto.

2.11.2 Rotação

A rotação de um ponto P(x, y) em relação à origem, através de um ângulo, é efetuada multiplicando suas coordenadas pelo **seno** e **cosseno** do ângulo (θ) e, somando ou subtraindo os resultados das multiplicações.

Sendo assim, o ponto P'(x', y') é definido como:

$$x' = x.\cos(\theta) - y.\sen(\theta) \qquad y' = x.\sen(\theta) + y.\cos(\theta)$$

Os ângulos positivos são definidos quando a rotação é feita no sentido anti-horário, e ângulos negativos quando a rotação é feita no sentido horário, como exemplifica a Figura 2.16.

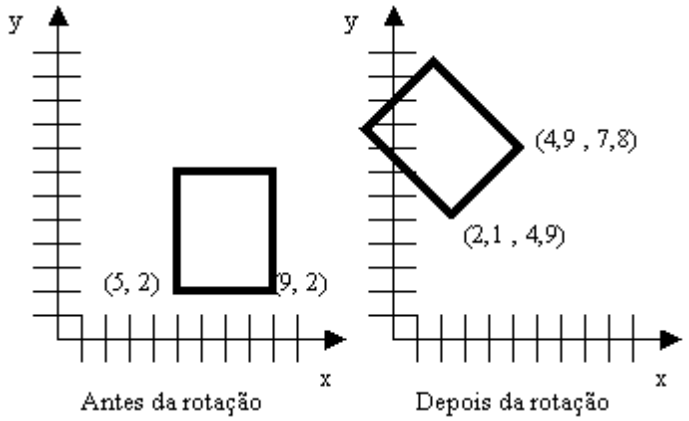

Figura 2.16 – Rotação de 45º de um objeto.

2.11.3 Escala

Pode-se efetuar mudanças de escala, ou apenas Escala, de um ponto P(x, y) pelo eixo **x** ou eixo **y**, através das multiplicações de suas coordenadas pelas unidades de escala **sx** e **sy**, conforme apresentado na Figura 2.17.

Assim, o ponto P'(x', y'), pode ser escrito como:

$$x' = sx.x \qquad y' = sy.y$$

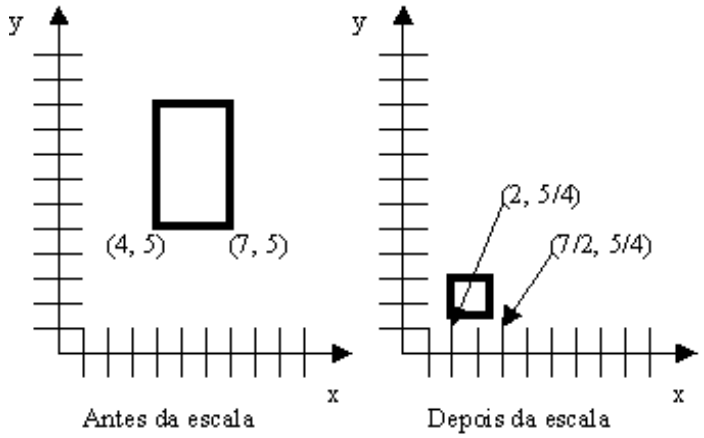

Figura 2.17 – Mudança de escala de um objeto.

Deve-se observar que a escala é feita em relação à origem, sendo assim, o objeto fica menor e mais próximo da origem. Também as proporções do objeto são alteradas, isto é, **sx** é diferente de **sy**. Porém ao utilizar escalas uniformes, **sx = sy**, as proporções não são afetadas.

2.12 Desenho de Polígonos 3D

Polígonos 3D (tridimensionais), que serão tratados aqui como objetos 3D, possuem coordenadas do tipo P(x, y, z) e são representados no eixo XYZ, por um conjunto de superfícies poligonais que delimitam o interior do objeto. O eixo XYZ permite visualizar a largura, altura e profundidade de um objeto, como mostram as Figuras 2.18 e 2.19. [8]

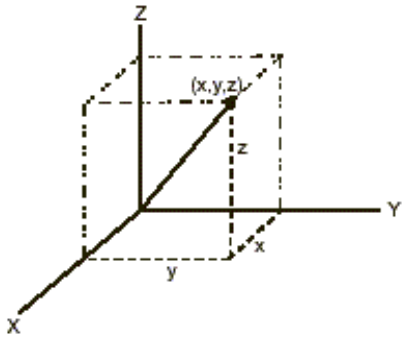

Figura 2.18 – Representação no eixo XYZ.

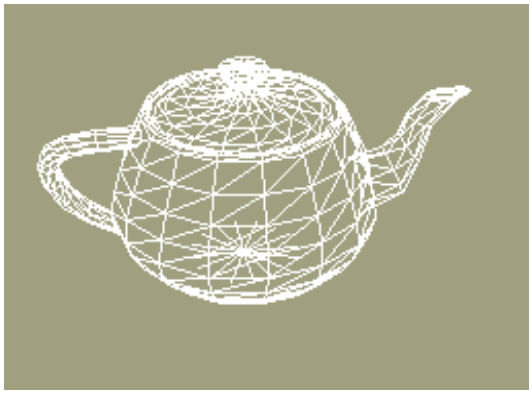

Figura 2.19 – Objeto 3D.

A vantagem deste método é que as superfícies planas formadas permitem um tratamento através de equações lineares, o que acelera o cálculo para a representação em 3D. Um poliedro pode ser facilmente representado por superfícies poligonais. Outros objetos, no entanto, precisam ser transformados numa rede de pontos que formarão os poliedros da superfície. Com este processo, as figuras podem ser apresentadas facilmente na forma de estrutura de “arame”. Além disso, o grau de detalhamento da superfície pode ser aumentado desde que a superfície seja dividida num número maior de polígonos.

Como ilustra a Figura 2.20, um polígono pertencente a um dado objeto é caracterizado pelo conjunto de vértices que o constitui e pelos atributos da sua superfície.

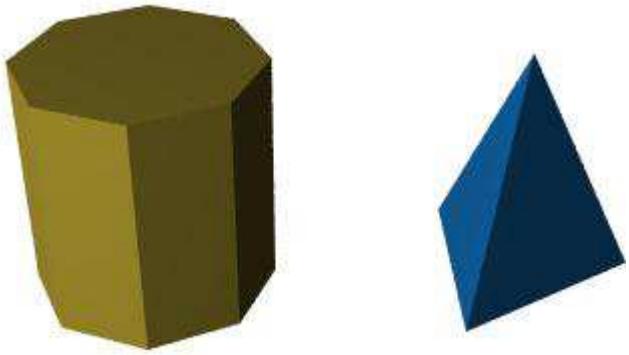

Figura 2.20 – Objetos formados por polígonos.

Uma das formas mais utilizadas na representação de objetos 3D é através de 3 listas, contendo respectivamente uma lista de vértices, uma lista de arestas e uma lista de polígonos, de acordo com a Figura 2.22. As coordenadas de todos os vértices do objeto são armazenadas na lista de vértices. A lista de arestas faz referência à lista de vértices para definir todas as arestas do objeto. De forma análoga, a lista de polígonos utiliza a lista de arestas para formar as superfícies poligonais do objeto, como mostra a Figura 2.21.

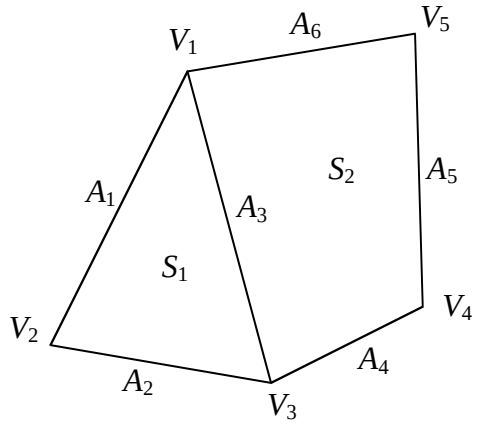

Figura 2.21 – Representação de superfícies poligonais.

Tabela de vértices			
V_1	x_1	y_1	z_1
V_2	x_1	y_1	z_1
V_3	x_1	y_1	z_1
V_4	x_1	y_1	z_1
V_5	x_1	y_1	z_1

Tabela de arestas		
A_1	V_1	V_2
A_2	V_2	V_3
A_3	V_1	V_3
A_4	V_3	V_4
A_5	V_4	V_5
A_6	V_1	V_6

Tabela de polígonos				
S_1	A_1	A_2	A_3	
S_2	A_3	A_4	A_5	A_6

Figura 2.22 – Listas (tabelas) para as superfícies poligonais da Figura 2.21.

2.12.1 Projeção perspectiva

No processo de desenho de um objeto 3D depara-se com o problema de representar um objeto 3D num meio 2D (bidimensional), que é a tela do monitor. Para fazer esta representação utiliza-se um processo denominado projeção perspectiva. [4]

As técnicas utilizadas em projeção perspectiva são derivadas daquelas utilizadas pelos artistas e desenhistas profissionais. Pode-se dizer que o olho do observador coloca-se no centro de projeção, e o plano que deve conter o objeto projetado transforma-se no plano de projeção. Dois segmentos de reta que saem do centro de projeção e atingem o objeto projetado no plano de projeção, são chamados de projetores. A Figura 2.23 ilustra a projeção de uma linha.

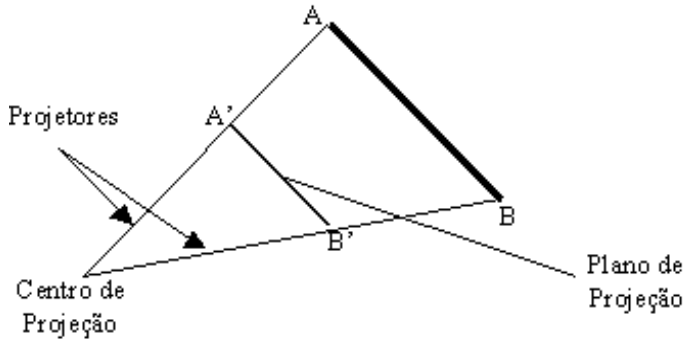

Figura 2.23 – Linha AB e sua projeção A'B'.

Projeções perspectivas são categorizadas pelo seu número de pontos de fuga principais, como mostra a Figura 2.24. Denominam-se pontos de fuga principais, quando se tem a aparência de haver uma interseção entre um conjunto de retas pararelas com um dos eixos principais **Ox**, **Oy** ou **Oz**. O número de pontos de fuga principais é determinado pelo número de eixos principais intersectados pelo plano de projeção.

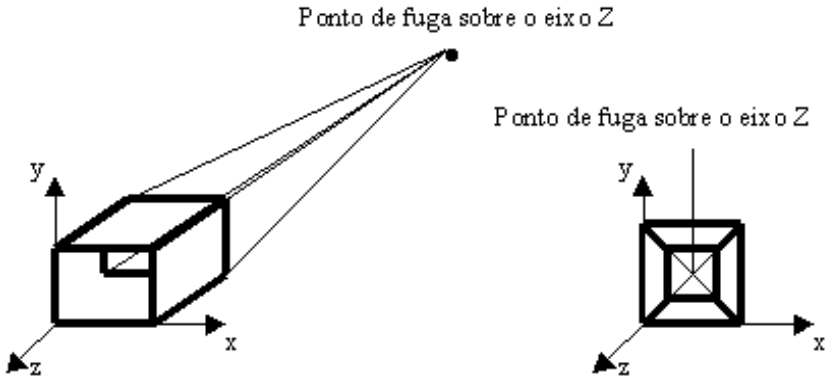

Figura 2.24 – Projeções de um cubo com 1 ponto de fuga sobre um plano cortando o eixo Z.

Para obter uma projeção perspectiva de um objeto 3D, são transformados os pontos ao longo dos projetores que se encontram no centro de projeção. As coordenadas do ponto P'(xp, yp) projetado no plano de observação são obtidas a partir da divisão das coordenadas homogêneas **x** e **y** do ponto a ser projetado P(x, y, z) por **h**:

$$xp = x / h \qquad yp = y / h$$

onde **h** é definido por:

$$h = (z + d) / d$$

A valor de **d** corresponde à distância do olho do observador em relação ao plano de projeção.

2.13 Transformações Geométricas em 3D

A capacidade para representar um objeto 3D é fundamental para a percepção de sua forma. Porém, em muitas situações é necessário mais do que isto, ou seja, poder manipular o objeto, transformando-o através de translações, rotações e escalas. [4]

Para tal, o sistema de coordenadas 3D utilizado será o da regra da mão direita, com o eixo **Z** perpendicular ao papel e saindo em direção ao observador, como mostra o eixo XYZ da Figura 2.25.

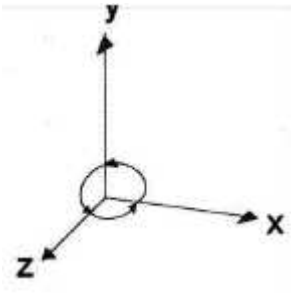

Figura 2.25 – Sistema de coordenadas 3D.

A forma de efetuar transformações no sistema 3D é semelhante a do sistema 2D, com a diferença que um ponto em 3D possui três coordenadas.

Sendo assim, a translação de um ponto P(x, y, z) para o ponto P'(x', y', z') é feita a partir de:

$$x' = x + dx \qquad y' = y + dy \qquad z' = z + dz$$

A rotação, por sua vez, pode ser efetuada em relação a qualquer um dos três eixos, e a coordenada do ponto correspondente ao eixo a ser utilizado permanece inalterada.

Logo, a rotação do ponto P(x, y, z) em relação ao eixo **x** é dada por:

$$y' = y.\cos(\theta) + z.(-\text{sen}(\theta)) \qquad z' = y.\text{sen}(\theta) + z.\cos(\theta)$$

Já em relação ao eixo **y** temos que:

$$x' = x.\cos(\theta) + z.\text{sen}(\theta) \qquad z' = x.(-\text{sen}(\theta)) + z.\cos(\theta)$$

E em relação ao eixo **z**, a rotação é feita por:

$$x' = x.\cos(\theta) + y.(-\text{sen}(\theta)) \qquad y' = x.\text{sen}(\theta) + y.\cos(\theta)$$

Para a rotação, o sentido positivo é dado quando observando um eixo positivo em direção à origem, e aplicando uma rotação de 90º, um eixo positivo é levado em direção a outro. Sendo assim, para o eixo **x**, o sentido positivo é de **y** para **z**, no caso do eixo **y**, é de **z** para **x**, e com relação ao eixo **z**, é de **x** para **y**.

Por último, a escala de um ponto P(x, y, z) para o ponto P'(x', y', z') é efetuada através de:

$$x' = sx.x \qquad y' = sy.y \qquad z' = sz.z$$

2.14 Determinação de Superfície Visível

O problema da visibilidade consiste, essencialmente, na determinação das superfícies mais próximas ao observador, que, consequentemente, estarão visíveis. Dado um objeto 3D, devemos definir que superfícies do objeto são visíveis para o centro de projeção. [6]

Então, para determinar as superfícies visíveis será utilizado o algoritmo *back-face culling*, que tem a característica de funcionar com precisão de cena, calculando a solução exatamente.

De acordo com este algoritmo, se a superfície de um objeto é aproximada por uma superfície poligonal sem borda, então as faces são polígonos e englobam completamente o volume do objeto. Assumindo-se que as normais (N) de todas as faces apontam para fora do volume, tem-se que as faces, cujas normais (N) apontam na direção contrária ao observador, estão numa parte do objeto cuja visibilidade é bloqueada por outras faces mais próximas ao observador. Deve-se assumir, também, que o objeto é convexo.

Analisando as coordenadas do objeto, uma face obstruída pode ser identificada pela não negatividade do produto escalar da sua normal (N) com o vetor definido pelo centro de projeção e qualquer ponto do polígono, como mostra a Figura 2.26.

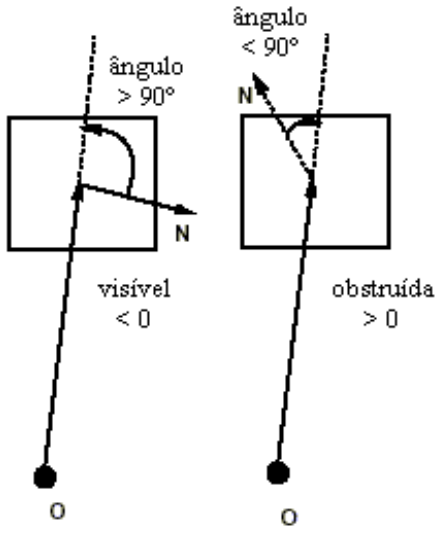

Figura 2.26 – Visibilidade de faces.

2.15 Mapeamento de Textura

O termo *textura* refere-se a uma imagem pré-digitalizada contida em um arquivo. O mapeamento de *textura* pode ser utilizado na representação de objetos 3D para diminuir os efeitos de "aparência plástica" causados pelo uso de modelos de iluminação, para permitir que objetos tenham superfícies com outras propriedades além da cor e, para dar a ilusão de uma superfície mais complexa. [9]

O mapeamento de *textura* consiste em reproduzir sobre a superfície de um objeto as propriedades de uma *textura*, como ilustra a Figura 2.27.

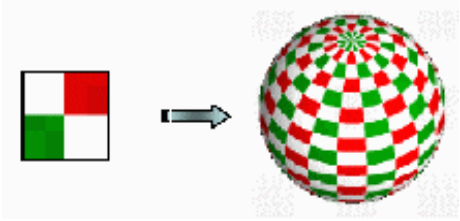

Figura 2.27 – Esfera mapeada com textura.

Texturas são funções T(s, t) cujo domínio é um espaço bidimensional (Figura 2.28) e o contradomínio pode ser cor, opacidade, etc. É comum ajustar a escala da imagem de tal forma que a imagem toda se enquadre no intervalo **0 <= s, t <= 1**. Normalmente a função em si é derivada de alguma imagem capturada, e se a imagem está armazenada numa matriz [0..N-1, 0..M-1], então, T(s, t) = [(1 – t).N, s.M].

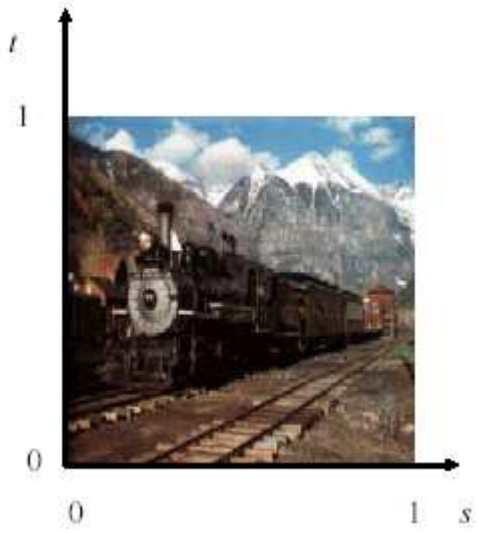

Figura 2.28 – Espaço de textura.

Então, no mapeamento de *textura* é preciso obter cada ponto P(u, v) da superfície a ser mapeada, correspondente a um ponto do espaço de textura, o que pode ser feito através das funções: [4]

$$u = fu(s, t) = au.s + bu.t + cu$$

$$v = fv(s, t) = av.s + bv.t + cv$$

As coordenadas (au, bu, cu) e (av, bv, cv) pertencem respectivamente aos vetores **u** e **v** da superfície a ser mapeada. Estes vetores originam do vértice de menores coordenadas, como mostra a Figura 2.29. [10]

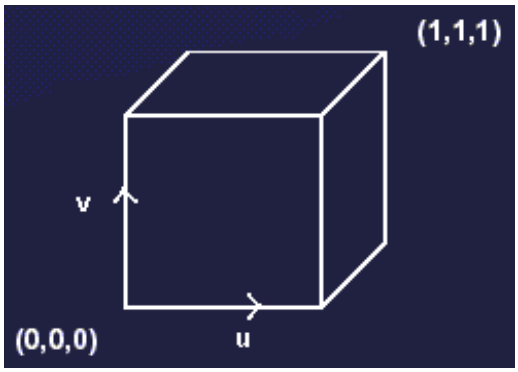

Figura 2.29 – Vetores U e V.

De acordo com a função de mapeamento apresentada, a *textura* é usada para "embrulhar" (*wrap*) o objeto.

3 BIBLIOTECAS GRÁFICAS

3.1 OpenGL

O OpenGL foi projetado para funcionar de forma independente do *hardware*. Sendo assim, em seu funcionamento o OpenGL envia um pedido para o sistema operacional onde um *driver* converte este pedido em um comando compatível com o *hardware* (placa de vídeo), conforme apresentado na Figura 3.1. [11]

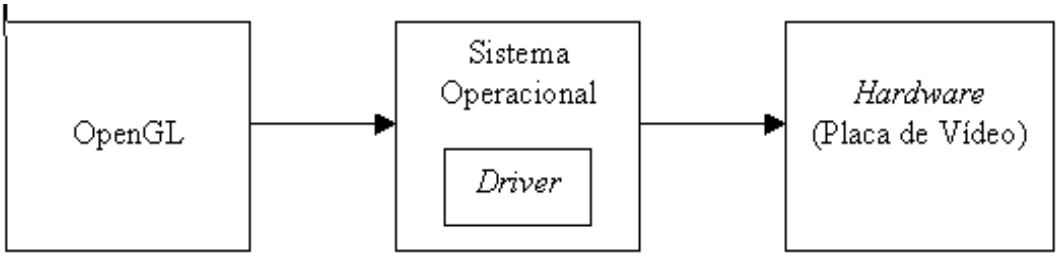

Figura 3.1 - Funcionamento do OpenGL.

3.1.1 Opengl como máquina de estados

O OpenGL é uma máquina de estados. Todos os estados ou modos habilitados nas aplicações gráficas que o utiliza têm efeito enquanto os mesmos estiverem ligados ou forem modificados. Além disto, todas as características do OpenGL são configuráveis através de variáveis, tais como: objetos que estão sendo desenhados, posições, transformações, cores, propriedades de *texturas*. [12]

3.1.2 O pipeline do opengl

Todos os estados e configurações do OpenGL são utilizados através de funções. A maioria dessas funções seguem uma ordem ao serem executadas no *hardware*, formando assim, uma série de estágios de execução, denominado *pipeline* do OpenGL. O diagrama da Figura 3.2 ilustra como o OpenGL obtém e processa os dados no *pipeline*. [11]

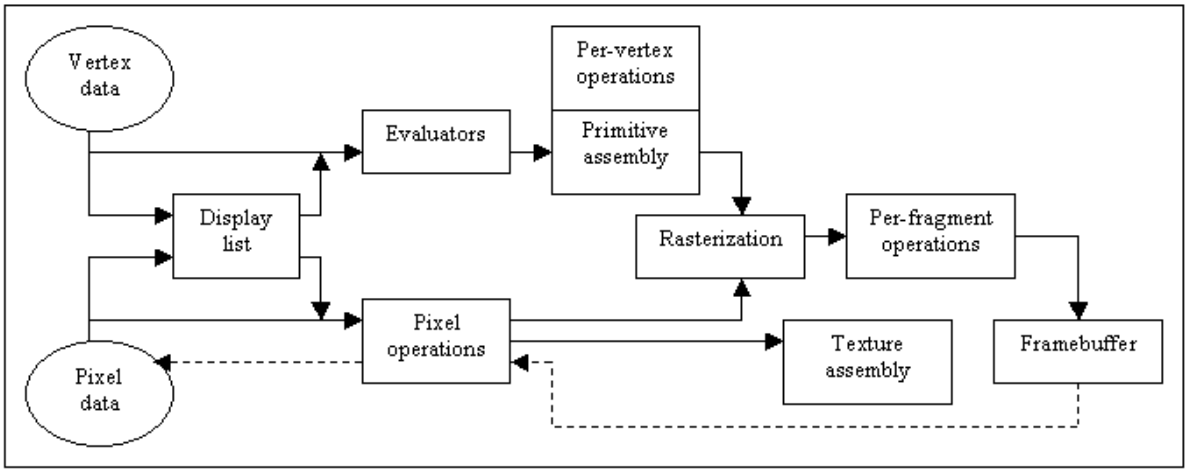

Figura 3.2 - Pipeline do OpenGL.

3.1.3 Componentes do pipeline

Com exceção do Vertex data, Pixel data e Display list, os componentes do *pipeline* que serão descritos aqui, são implementados no *hardware* e executam suas funções através de pedidos do OpenGL, são eles:

- Vertex data (Dados dos vértices): componente onde são armazenados os dados geométricos dos objetos, como por exemplo, os vértices;

- Pixel data (Dados dos *pixels*): componente onde são armazenados os dados dos *pixels*, como cores e posições;

- Display list (Lista de exposição): lista onde os dados que descrevem um objeto, como vértices e cores, podem ser amazenados para uso corrente ou para serem usados num momento posterior;

- Evaluators (Avaliadores): derivam os vértices dos objetos 3D para poderem ser usados em suas representações;

- Per-vertex operations (Operações com vértices) e Primitive assembly (Montagem de primitivas): operações com as coordenadas dos vértices de um objeto 3D, geram e transformam as coordenadas de uma *textura* para serem usadas no mapeamento de

textura do objeto 3D. As operações com vértices nas transformações geométricas de translação, rotação e escala também são efetuadas neste componente. A montagem de primitivas é responsável pela montagem de objetos 2D antes de seu desenho. O processo principal da montagem de primitivas é o *clipping*, onde é feita a eliminação das partes do objeto que estão fora do plano de visualização (tela do monitor). O *clipping* de um vértice simplesmente aceita ou rejeita o vértice; o *clipping* da linha, circunferência, elipse ou polígono pode adicionar vértices dependendo de como estes objetos são interligados;

- Pixel operations (Operações de pixels): agrupam e processam os dados dos *pixels* criando um mapa de *pixels*;

- Texture assembly (Montagem de texturas): componente responsável pela montagem e armazenamento de *texturas*, facilitando assim, a comutação entre elas e acelerando o desempenho na sua utilização;

- Rasterization (Rasterização): converte os vértices e os dados dos *pixels* em fragmentos. Cada quadrado do fragmento corresponde a um *pixel* no *framebuffer* (memória da placa de vídeo). Diante disto, os cálculos para suportar o *anti-aliasing* são feitos e o valor da cor é atribuído a cada quadrado do fragmento;

- Per-fragment operations (Operações fragmentadas): realiza a operação de "texturização", onde um *texel* é gerado da *textura* para cada fragmento e aplicado ao fragmento. Também executa uma operação para determinar as superfícies visíveis de um objeto 3D. Estas operações podem ser ativadas ou não;

- Framebuffer: memória da placa de vídeo para onde dados processados no *pipeline* são enviados para depois serem exibidos pela placa na tela do monitor.

3.1.4 Modo imediato

Modo utilizado pelo OpenGL para processar os dados dos objetos assim que eles forem enviados para o *pipeline*, ou seja, os dados não terão que ser armazenados na Lista de exposição para aguardar o processamento. [13]

3.1.5 Desenho 2D

O desenho de um objeto 2D, como uma reta, uma circunferência, uma elipse, um polígono, é realizado pelo OpenGL através de um estágio do *pipeline* onde várias funções são executadas pelos componentes do *pipeline* em uma determinada ordem. Ao serem enviados para o *pipeline*, os dados dos objetos percorrem o caminho apresentado na Figura 3.3. [11]

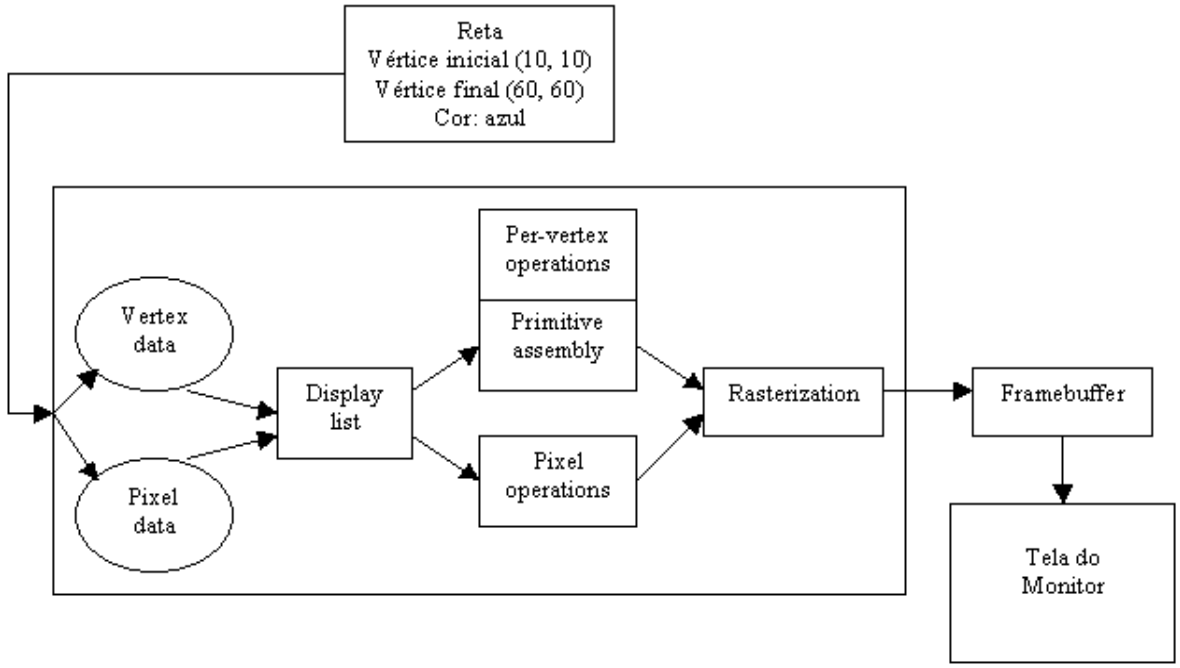

Figura 3.3 - Desenho 2D no OpenGL.

3.1.6 Desenho 3D

Para o desenho de objetos 3D, os dados dos objetos percorrem um caminho diferente, passando também pelos componentes Evaluators, Per-vertex operations e Per-fragment operations, como mostra a Figura 3.4.

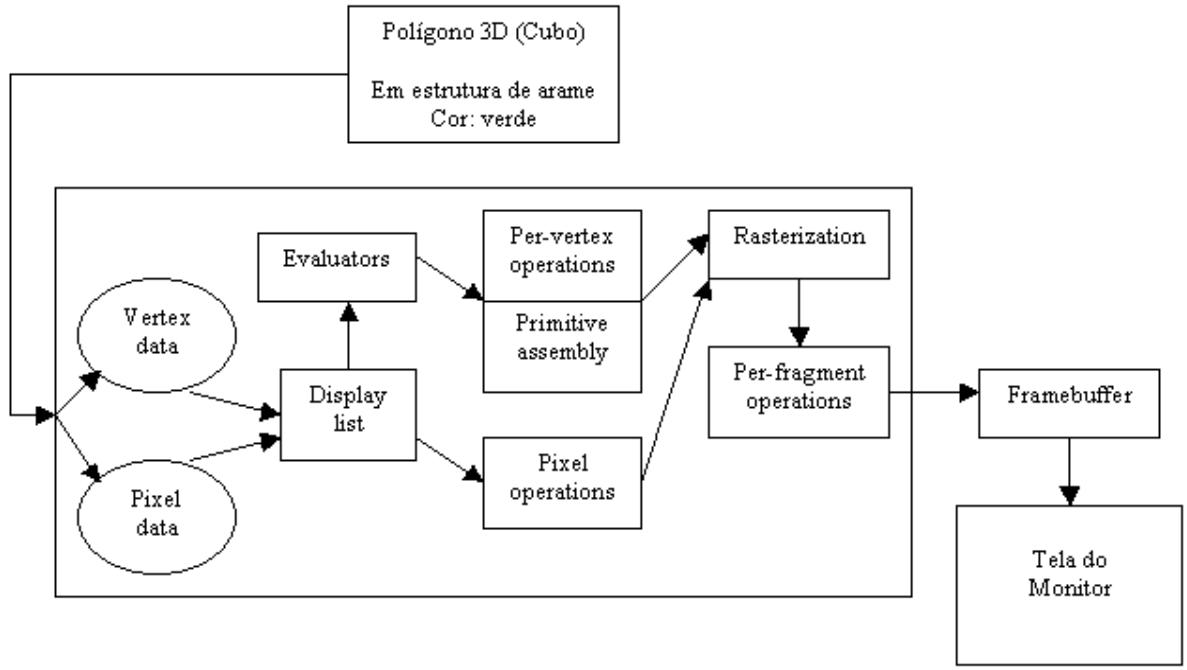

Figura 3.4 - Desenho 3D no OpenGL.

3.1.7 Mapeamento de textura

Na utilização do mapeamento de *textura* no desenho de objetos 3D, os dados percorrem o mesmo caminho apresentado da Figura 3.4, sendo adicionado apenas o componente Texture assembly, que recebe as informações da *textura* contida em um arquivo. A Figura 3.5 ilustra o que foi dito. [14]

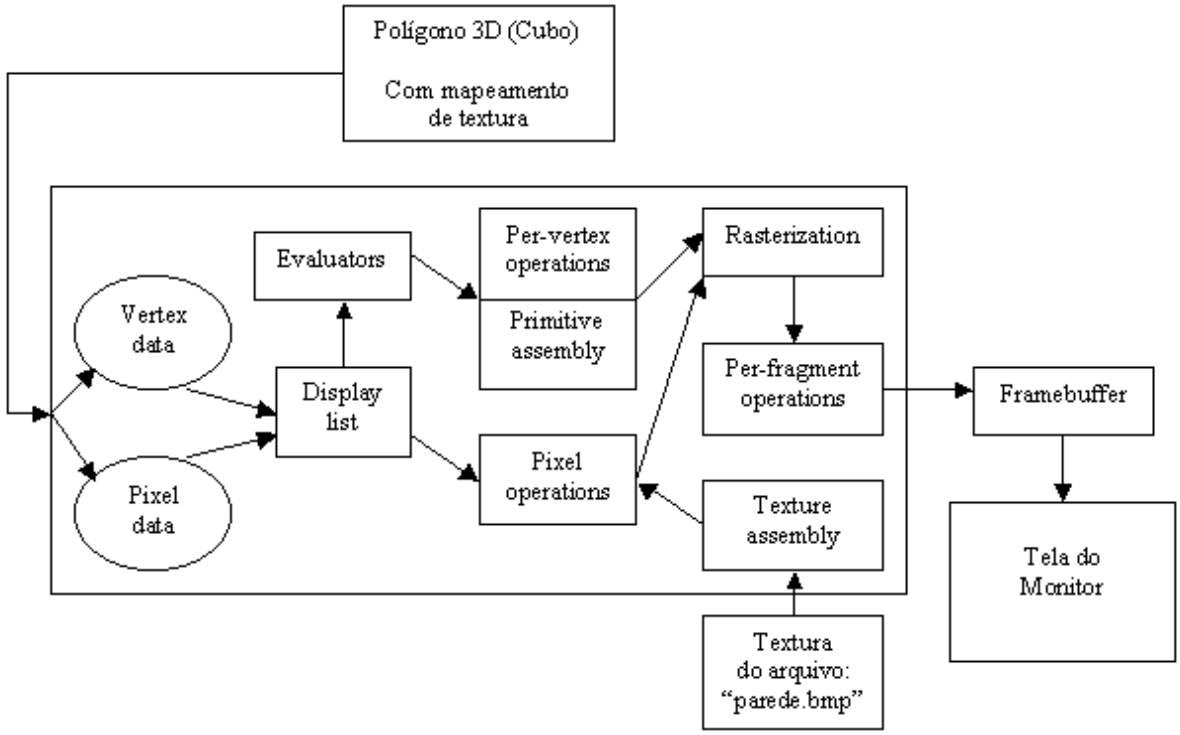

Figura 3.5 - Mapeamento de textura no OpenGL.

3.1.8 Características do opengl

O OpenGL obtém informações da placa de vídeo (hardware) através de uma requisição feita ao *driver* do sistema operacional. Estas informações são usadas pelo OpenGL para manipular a placa através do *driver*. Entre elas, estão: [11]

a) Processador: informações sobre o processador gráfico da placa, como por exemplo, velocidade;
b) Memórias de vídeo e *textura*: informações como tamanho e velocidade; e,
c) Recursos: informações sobre as funções gráficas disponíveis.

Para utilizar gráficos em computador é preciso iniciar o modo gráfico, e o OpenGL faz esta inicialização através da função glbegin( ) e finaliza este modo com glend( ). Todas as demais funções de um código escrito usando OpenGL devem estar entre glbegin( ) e glend( ).

Entre os números de cores suportados pelo OpenGL, estão:

a) 8 bits: 256 cores;
b) 16 bits: 65.536 cores;
c) 24 bits: 16.777.216 cores; e,
d) 32 bits: 16.777.216 cores, onde 24 bits são para as cores e 8 bits para o valor de alpha que é usado na geração de objetos com transparência.

3.1.9 Glut

O GLUT é um conjunto de ferramentas que podem ser usadas com o OpenGL no desenvolvimento de aplicações gráficas. Ele implementa um sistema de janelas permitindo todos os tipos de operações com janelas. [13]

3.1.10 Sintaxe de comandos

Os comandos do OpenGL obedecem a um padrão bem definido para especificação dos nomes de funções e de constantes. Todos os comandos utilizam-se do prefixo **gl** em letras minúsculas. Similarmente, as constantes são definidas com as iniciais **GL_**, em letras maiúsculas, e usam um caracter *underscore* (_) para separar as palavras. Ex.: glbegin( ) e GL_COLOR_BUFFER_BIT.

3.2 DirectX

O DirectX é formado por um conjunto de componentes denominado SDK (Software Development Kit), onde cada componente tem uma finalidade específica. Entre eles estão: [15]

a) Direct3D: responsável pelo desenho e manipulação de objetos 2D e 3D;
b) DirectSound: executa sons em aplicações;
c) DirectMusic: executa músicas em aplicações;
d) DirectShow: executa animações em aplicações;
e) DirectPlay: auxilia o desenvolvimento de aplicações para redes; e,
f) DirectInput: manipula as entradas e saídas de periféricos.

De acordo com contexto deste trabalho será estudado apenas o componente Direct3D.

3.2.1 Direct3d

O Direct3D é componente gráfico de baixo-nível que nos permite desenhar e manipular objetos usando funções gráficas. Assim como o OpenGL, o Direct3D pode ser visto como um mediador entre a aplicação gráfica e o *hardware* (placa de vídeo), como mostra a Figura 3.6.

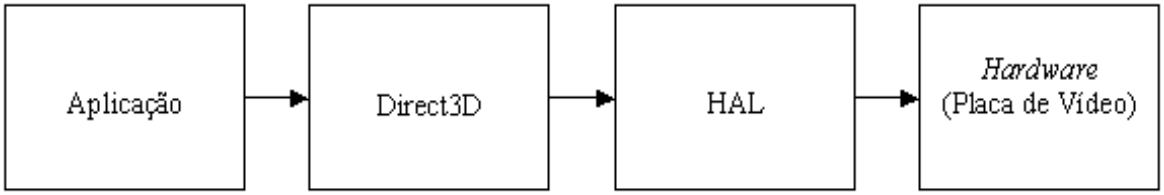

Figura 3.6 - Relação entre a aplicação, o Direct3D e o hardware.

Como pode ser visto na Figura 3.6, o Direct3D define interfaces e funções disponíveis para a aplicação. Estas interfaces e funções representam todas as características disponíveis pelo Direct3D. Há de se ressaltar ainda que, assim como acontece com o OpenGL, o uso do Direct3D depende das capacidades do *hardware*.

Como mostra a Figura 3.6, há um dispositivo intermediário entre o Direct3D e o *hardware*, a HAL (Camada de Abstração do Hardware). O Direct3D não pode interagir diretamente com o *hardware* porque há uma grande variedade de placas de vídeo existentes, e cada placa têm diferentes características e formas de implementação. Por isto, o Direct3D depende dos fabricantes de *hardware* para a implementação da HAL. A HAL é um código específico que diz ao *hardware* qual função deve ser realizada. Desta forma, o Direct3D pode funcionar sem conhecer os detalhes específicos das placas de vídeo, e sua especificação pode ser feita independente delas.

Os fabricantes de *hardware* implementam todas as funções que a placa de vídeo pode realizar na HAL. Alguns recursos que são disponibilizados pelo Direct3D, mas que a placa não tem, não são implementados na HAL. Tentar utilizar uma função do Direct3D que não está implementada na HAL resulta num erro, a não ser que seja uma função para o processamento de vértices, caso em que a funcionalidade pode ser emulada por *software*. Para evitar estes erros, o Direct3D permite verificar se as funções estão disponíveis na placa.

3.2.2 O dispositivo ref

Às vezes, é preciso desenvolver aplicações que utilizam funções do Direct3D, mas que a placa não tem implementadas. Para este propósito, o Direct3D provê um dispositivo de

referência denominado REF (Reference Device), o qual emula todas as funções do Direct3D por *software*. Isto permite usufruir de muitas das últimas tecnologias que as placas anteriores a estas tecnologias não têm. [2]

3.2.3 D3ddevtype

No desenvolvimento de uma aplicação com o Direct3D, deve-se especificar o dispositivo a ser utilizado, atribuindo ao enumerador D3DDEVTYPE o membro D3DDEVTYPE_HAL para utilização do HAL e o membro D3DDEVTYPE_REF se o dispositivo for o REF.

3.2.4 Interfaces

Interfaces são estruturas associadas aos objetos gráficos. O Direct3D usa as interfaces e suas funções para desenhar e manipular os objetos. As funções que uma interface disponibiliza são específicas daquele objeto associado a ela.

3.2.5 Arquitetura do direct3d

Assim como o OpenGL, o Direct3D é dividido em componentes formando um *pipeline* de processamento, onde cada componente tem sua função específica. São eles: [15]

- Vertex Data: armazena os vértices dos objetos;

- Vertex Shaders: realiza operações geométricas com os vértices, preparando os vértices de objetos 3D para suas representações e calculando as coordenadas das *texturas*;

- Transformation Engine: onde são feitas as transformações geométricas como translação, rotação e escala;

- Primitive Operations: componente onde as primitivas (objetos) são montados, e parâmetros são definidos, como por exemplo, se o objeto vai ter preenchimento ou não. São definidas aqui, as primitivas que pertencem a um objeto 3D e se este vai ser mapeado com uma *textura* ou não. Também, é feita a determinação das superfícies vísiveis dos objetos 3D;

- Pixel Shader: aqui as cores são atribuídas aos *pixels* e estes são agrupados. Os *pixels* de uma *textura* que serão usados, também são definidos neste componente;

- Rasterization: no processo de rasterização, a cada ponto de um objeto é associado um *pixel*, e a técnica de *anti-aliasing* é aplicada no objeto;

- Framebuffer: memória da placa de vídeo para onde os objetos processados são enviados para depois serem exibidos pela placa na tela do monitor.

3.2.6 Desenho com o direct3d

O Direct3D desenha e manipula um objeto 2D ou 3D, enviando os dados do objeto para serem processados pelos componentes no *pipeline*. Os dados são processados na ordem ilustrada na Figura 3.7.

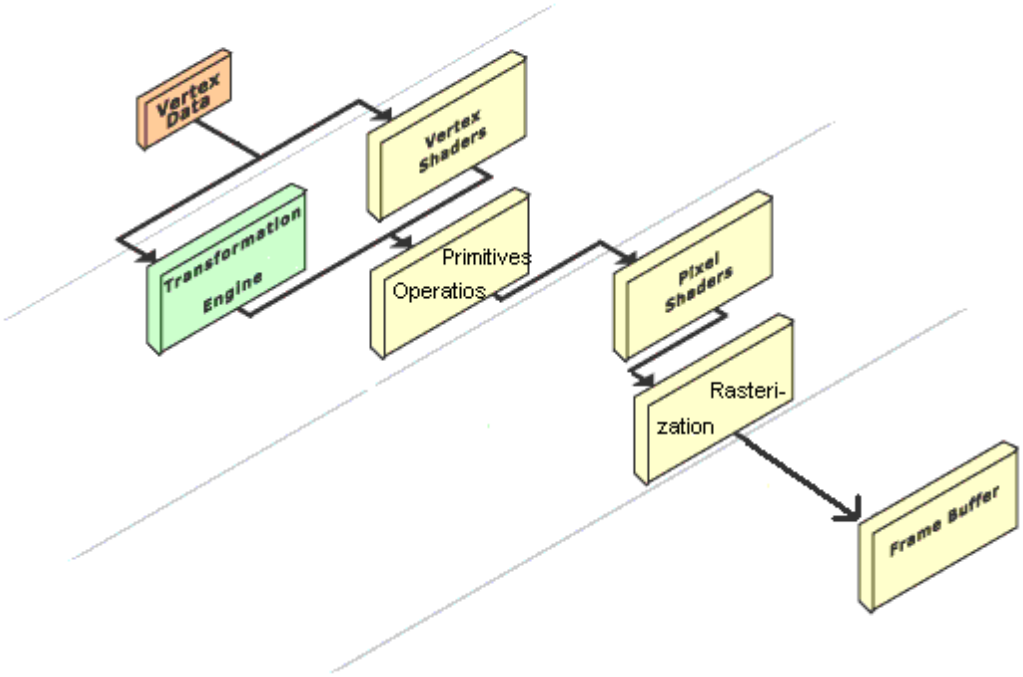

Figura 3.7 - Pipeline do Direct3D.

3.2.7 Características do direct3d

Antes de utilizar as funções gráficas, o Direct3D também obtém informações do *hardware*, para saber qual dispositivo deve ser inicializado, se o HAL ou o REF. [2]

Após a inicialização do dispositivo, o modo gráfico é ativado através da função BeginScene( ) e desativado por EndScene( ). Todas as demais funções de um código escrito usando Direct3D devem estar entre estas funções.

Os números de cores suportados pelo Direct3D são os mesmos do OpenGL, citados na seção 3.1.8.

3.2.8 Com

Component Object Model (COM), é uma tecnologia que permite ao DirectX ser independente da linguagem de programação utilizada no desenvolvimento de uma aplicação e ter compatibilidade com versões prévias de aplicações desenvolvidas com o DirectX.

Usualmente, refere-se a um objeto COM como uma interface, a qual propõe-se que poderá ser usada como uma classe da linguagem de programação C++. A maioria dos detalhes do COM são transparentes quando se programa DirectX com C++. Um detalhe importante é que se deve obter apontadores às interfaces COM através de funções especiais ou métodos de outra interface COM. Os objetos COM têm sua própria gerência de memória, e as interfaces COM têm como prefixo um letra **I** maiúscula. Por exemplo, a que representa uma superfície é IDirect3DSurface9.

3.3 Allegro

A Allegro é uma biblioteca gráfica diferente do OpenGL e do DirectX, no que diz respeito ao caminho que os dados dos objetos têm que percorrer até o desenho do objeto na tela do monitor. Não existe na Allegro um pipeline de obtenção e processamento de dados dos objetos, sua estrutura de funcionamento é bem simples, se comparado ao OpenGL e DirectX. Em suma, a Allegro é formada por conjuntos de funções gráficas, onde a combinação de várias funções de vários conjuntos, é utilizada para atingir um objetivo específico, como por exemplo, desenhar um objeto. O importante é que na Allegro não é feito o agrupamento de funções semelhantes em um componente. Portanto, o Allegro não gerencia o processamento dos dados dos objetos, como fazem o OpenGL e o DirectX. Ela apenas disponibiliza as funções, que são executadas pelo computador como uma outra função qualquer, ou seja, sem a supervisão da Allegro. A Figura 3.8 ilustra o que foi dito. [16]

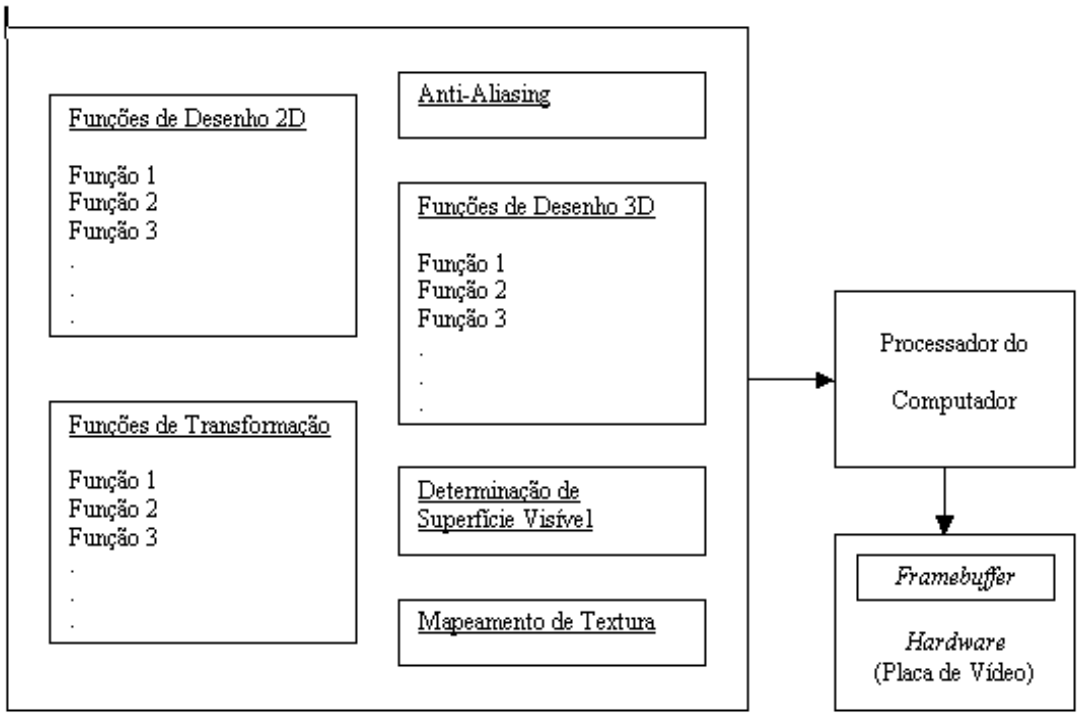

Figura 3.8 - Estrutura da Allegro.

É importante ressaltar também que, todas as funções da Allegro são executadas pelo processador do computador, denominado processamento por *software*. A única relação que a Allegro tem com o *hardware* (placa de vídeo), é o envio dos dados processados para o *framebuffer* (memória de vídeo) da placa, para poderem ser exibidos.

3.3.1 Desenho com a allegro

O desenho de um objeto, como por exemplo, um objeto 2D com transformação de rotação é feito pela Allegro, enviando as funções específicas para tal, para o processador, como mostra a Figura 3.9. [17]

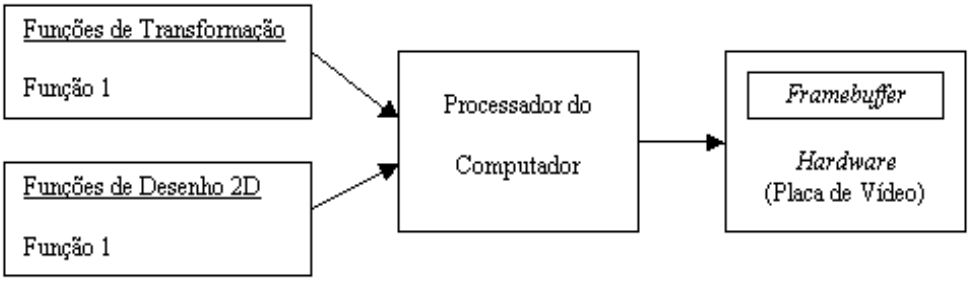

Figura 3.9 - Desenho de um objeto.

Como pode ser visto na Figura 3.9, os vértices do objeto são transformados e o objeto é montado no processador, para depois ser enviado para o *framebuffer*.

3.3.2 Características da allegro

As únicas informações que a Allegro obtém do *hardware* (placa de vídeo), são o tamanho do *framebuffer* (memória de vídeo) e o endereço desta memória no computador.

Assim, como o OpenGL e o DirectX, a Allegro utiliza duas funções para iniciar e finalizar o modo gráfico, que são, respectivamente: Allegro_Init( ) e Allegro_Exit( ). Todas as demais funções devem estar entre elas. A Allegro suporta 8, 16, 24 e 32 bits de cor.

3.4 Aplicações

Como exemplo de algumas aplicações gráficas, mais especificamente jogos, que utilizam o OpenGL e o DirectX, será mostrado aqui algumas imagens dos jogos Quake 3 Arena e Unreal Tournament. As Figuras 3.10 e 3.11 mostram o Quake 3 Arena utilizando o OpenGL, e as Figuras 3.12 e 3.13 mostram o Unreal Tournament utilizando o DirectX.

Figura 3.10 – Imagem 1 do Quake 3 Arena.

Figura 3.11– Imagem 2 do Quake 3 Arena.

Figura 3.12 – Imagem 1 do Unreal Tournament.

Figura 3.13 – Imagem 2 do Unreal Tournament.

4 MATRIX

A Matrix deve ter uma estrutura simples, semelhante a da Allegro. Para tal, não existirá a presença de componentes que agrupam funções semelhantes. As funções devem estar dispostas na Matrix sem nenhuma relação estrutural entre elas. Portanto, os dados dos objetos na Matrix não percorrerão sempre o mesmo caminho em seu processamento.

A execução de todas as funções da Matrix será feita pelo processador do computador, sendo assim, a Matrix não fará uma gerência da execução, que vai desde o fornecimento dos dados para as funções até a exibição dos objetos na tela do monitor. Assim como, na Allegro, será feito o processamento por *software*. A Matrix só comunicará com o *hardware* (placa de vídeo), através do processador, para o envio dos dados processados para o *framebuffer* (memória de vídeo) da placa, e para iniciar e finalizar o modo gráfico. A Figura 4.1 ilustra a estrutura da Matrix.

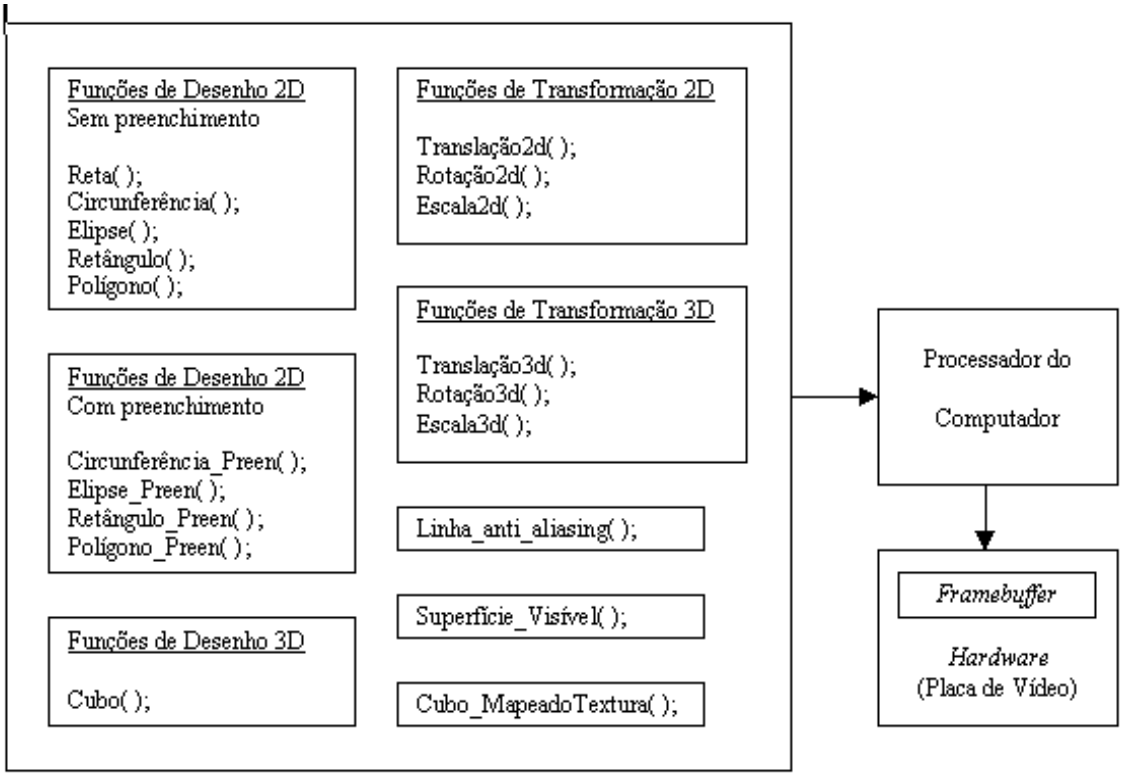

Figura 4.1 - Estrutura da Matrix.

Como foi dito anteriormente, as funções para desenho e manipulação de objetos são enviadas para o processador, e os resultados desse processamento são enviados para o *framebuffer* da placa, como mostra a Figura 4.2.

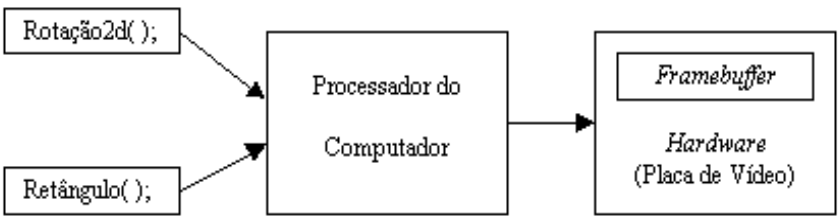

Figura 4.2 - Processamento dos objetos pela Matrix.

4.1 Informações do Hardware

A Matrix obtém informações do *hardware* (placa de vídeo) para poder enviar os dados processados para o *framebuffer* da placa. Estas informações são:

a) Endereço do *framebuffer*: utilizado para saber onde se localiza o *framebuffer*;

b) Tamanho do *framebuffer*: utilizado como referência a quantidade de dados que podem ser enviados;

c) Resoluções suportadas: refere-se as resoluções de vídeo que são suportadas pela placa de vídeo; e,

d) Números de cores: refere-se aos números de cores que podem ser utilizados com a placa.

4.2 Modo Gráfico

Modo gráfico é um estado de um sistema de computador que quando iniciado, permite a utilização de gráficos neste computador. Para iniciá-lo, a Matrix deve enviar para o *hardware* (placa de vídeo) um comando dizendo que o modo gráfico deve ser iniciado. A partir disso, o *framebuffer* do *hardware* vai estar preparado para receber os dados dos objetos enviados pela Matrix, e depois exibí-los na tela do monitor.

Quando a Matrix não precisar mais do *hardware*, ele deve enviar um comando dizendo que o modo gráfico deve ser finalizado.

4.3 Desenho de um Pixel

Todos os objetos gráficos são desenhados na tela do monitor, colorindo-se os pontos que descrevem a geometria do objeto com uma cor. Sendo assim, *pixel* é um ponto colorido.

O *framebuffer* é a memória de vídeo do *hardware* (placa de vídeo) e é estruturado como um vetor. Como a tela do monitor é formada por várias linhas consecutivas, de cima para baixo ou vice-versa, e cada linha possui vários pontos, pode-se abstrair que se estas linhas forem colocadas uma do lado da outra, formarão um vetor.

Sabendo-se que como cada posição no *framebuffer* equivale a uma posição na tela do monitor, a Matrix desenha um *pixel* armazenando um valor de cor em uma posição do *framebuffer*. A partir disto, o *hardware* vai identificar a posição correspondente na tela do monitor e colorir este ponto com a cor desejada, conforme apresentado na Figura 4.3.

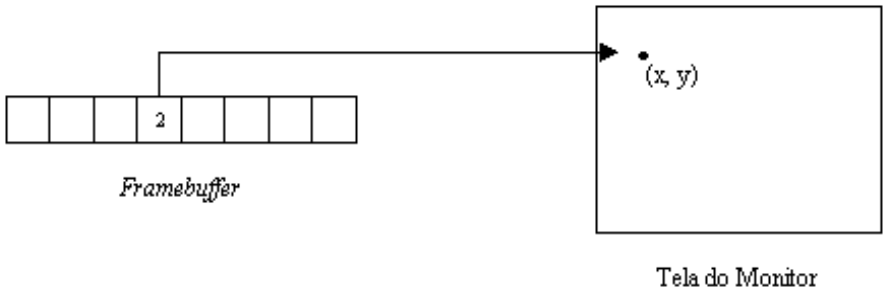

Figura 4.3 - Relação entre o framebuffer e a tela do monitor.

4.4 Resoluções de Vídeo

Antes de desenhar qualquer *pixel*, a Matrix deve definir a resolução de vídeo a ser utilizada. A resolução de vídeo corresponde ao número de *pixels* que podem ser desenhados na tela do monitor, ou seja, para uma resolução de 640 x 480, pode-se desenhar 640 *pixels* em uma linha horizontal (resolução **X**) e 480 *pixels* em uma linha vertical (resolução **Y**). Assim, ao multiplicar estes dois valores, sabe-se que podem ser desenhados 307.200 *pixels* na tela do monitor. A Figura 4.4 ilustra o que foi dito.

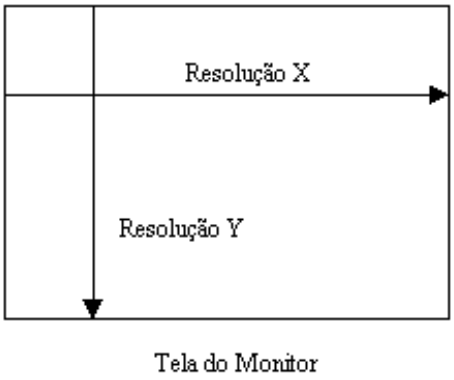

Figura 4.4 - Resolução de vídeo.

A Matrix poderá utilizar as resoluções de vídeo apresentadas abaixo, se estas forem suportadas pelo *hardware* (placa de vídeo). São elas:

a) 320 x 200;
b) 640 x 480;
c) 800 x 600; e,
d) 1024 x 768.

4.5 Números de Cores

Quando se quer desenhar um *pixel*, deve-se atribuir um valor de cor a este *pixel*. Este valor corresponde a um número inteiro, como o número 2 (dois) por exemplo, que num sistema de 4 bits (16 cores) representa a cor azul. Como o *hardware* associa cada cor a um valor, antes de desenhar qualquer *pixel*, a Matrix deve definir o sistema de cores a ser utilizado para que o *hardware* faça esta associação. Por exemplo, ao definir o sistema como sendo de 4 bits, e fazendo a pontenciação 2 elevado a 4, obtém-se que o número de cores disponíveis é 16.

Quanto ao número de cores, a Matrix poderá utilizar os números apresentados abaixo, se estes forem suportados pelo *hardware* (placa de vídeo). São eles:

a) 8 bits: 256 cores;
b) 16 bits: 65.536 cores; e,
c) 24 bits: 16.777.216 cores.

4.6 Protótipo Operacional

4.6.1 Linguagens de programação

O desenho e manipulação de objetos gráficos em computador envolvem muitos cálculos, e estes cálculos devem ser efetuados rapidamente para não prejudicar a exibição dos objetos. Para tal, é importante utilizar linguagens de programação que permitem a geração de um aplicativo compacto e de rápida execução. Além disto, estas linguagens devem permitir a programação em baixo-nível, ou seja, uma comunicação direta com o *hardware* (placa de vídeo). A portabilidade também é um fator importante, pois o aplicativo deve executar em diferentes computadores. O código escrito para a Matrix deve ser estruturado e composto por procedimentos e funções.

Sendo assim, as linguagens de programação mais adequadas para a implementação da Matrix, são as linguagens C e Assembly. A linguagem C foi escolhida por gerar aplicativos de rápida execução com código estruturado, e a linguagem Assembly por permitir a programação em baixo-nível. O código escrito para a Matrix será uma mescla da utilização das duas linguagens.

Como exemplo de utilização destas linguagens, temos praticamente, quase todos os aplicativos gráficos existentes, entre jogos e aplicativos de desenho. As Bibliotecas Gráficas OpenGL, DirectX e Allegro, também foram implementadas usando-se as linguagens C e Assembly.

4.6.2 Funções implementadas

Nesta seção, serão descritas as funções implementadas na Matrix, utilizando as técnicas de Computação Gráfica apresentadas no Capítulo 2. As funções são:

- void iniciar_matrix(short w, short h, short bits_pixel): função utilizada para iniciar o modo gráfico, com os parâmetros da resolução de vídeo e do sistema de cor a ser utilizado;

- void finalizar_matrix(void): usa-se esta função para finalizar o modo gráfico;

- void plot_pixel(short x, short y, unsigned long cor): utilizada para armazenar o valor da cor de um *pixel* em uma posição do *buffer* (memória) de *pixels*, que corresponde a um ponto (x, y) da tela do monitor;

- void esvaziar_buffer(void): utilizada para enviar os dados do *buffer* de *pixels* para o *framebuffer* (memória de vídeo);

- void limpar_buffer(unsigned long cor): usa-se esta função para armazenar um único valor de cor em todas as posições do *buffer* de *pixels*;

- unsigned long ler_pixel(short x, short y): usa-se esta função para retornar um valor de cor de uma posição do *buffer* de *pixels*;

- void reta(short x1, short y1, short x2, short y2, unsigned long cor): função usada para desenhar uma reta, do ponto (x1, y1) ao ponto (x2, y2);

- void circunferencia(short xc, short yc, short raio, unsigned long cor): desenha uma circunferência com centro no ponto (xc, yc) e com raio definido pelo parâmetro "raio";

- void elipse(short xc, short yc, short rx, short ry, unsigned long cor): desenha uma elipse com centro no ponto (xc, yc) e raios de tamanho "rx" e "ry";

- void retangulo(short x1, short y1, short x2, short y2, unsigned long cor): utilizada para desenhar um retângulo com comprimento de "x1" a "x2", e altura de "y1" a "y2";

- void poligono2d(struct ponto lista[ ], short num_pontos, unsigned long cor): através de uma lista que contém os vértices (pontos) do polígono, desenha um polígono 2D;

- void circunferencia_preen(short xc, short yc, short raio, unsigned long cor): desenha uma circunferência preenchida com uma cor;

- void elipse_preen(short xc, short yc, short rx, short ry, unsigned long cor): desenha uma elipse preenchida com uma cor;

- void retangulo_preen(short x1, short y1, short x2, short y2, unsigned long cor): desenha um retângulo preenchido com uma cor;

- void poligono2d_preen(struct ponto lista[ ], short num_pontos, unsigned long cor): desenha um polígono com preenchimento por uma cor;

- struct ponto translacao2d(struct ponto p, short dx, short dy): translada o ponto "p", "dx" e "dy" unidades, e retorna o ponto transladado;

- struct ponto rotacao2d(struct ponto p, float angulo): efetua a rotação do ponto "p" em relação ao ângulo "angulo", e retorna o ponto rotacionado;

- struct ponto escala2d(struct ponto p, float sx, float sy): efetua a escala do ponto "p", através dos parâmetros "sx" e "sy";

- void poligono3d(struct ponto3d lista[ ], short num_pontos3d, short xc, short yc, unsigned long cor, short sup_vis): através de uma lista que contém os vértices 3D (pontos 3D) do poligono, desenha um polígono 3D (cubo) com centro no ponto (xc, yc) e, com determinação de superfícies visíveis se o valor do parâmetro “sup_vis” for 1 e sem se o valor for diferente de 1;

- struct ponto3d translacao3d(struct ponto3d p, short dx, short dy, short dz): translada o ponto 3D "p", "dx", "dy" e "dz" unidades, e retorna o ponto transladado;

- struct ponto3d rotacao3d_x(struct ponto3d p, float angulo): efetua a rotação do ponto 3D "p" em relação ao eixo **x** e ao ângulo "angulo", e retorna o ponto rotacionado;

- struct ponto3d rotacao3d_y(struct ponto3d p, float angulo): efetua a rotação do ponto 3D "p" em relação ao eixo **y** e ao ângulo "angulo", e retorna o ponto rotacionado;

- struct ponto3d rotacao3d_z(struct ponto3d p, float angulo): efetua a rotação do ponto 3D "p" em relação ao eixo **z** e ao ângulo "angulo", e retorna o ponto rotacionado;

- struct ponto3d escala3d(struct ponto3d p, float sx, float sy, float sz): efetua a escala do ponto 3D "p", através dos parâmetros "sx", "sy" e "sz";

- short back_face_culling(struct face_cubo f): através do parâmetro “f” que contém os vértices 3D que formam a face de um objeto (cubo), determina se a face é visível ou não, retornando o valor 1 para sim, e 0 para não;

- void poligono3d_mapeado(struct ponto3d lista[ ], short num_pontos3d, short xc, short yc, struct pcx textura): usando os vértices 3D do polígono (cubo), presentes em uma lista, desenha um polígono 3D (cubo) mapeado com uma textura fornecida pelo parâmetro "textura" e, com centro no ponto (xc, yc) e determinação de superfícies visíveis.

Os termos "ponto" e "ponto3d" que aparecem em algumas funções descritas, referem-se a duas estruturas, onde "ponto" armazena duas coordenadas (x, y), e "ponto3d" armazena três coordenadas (x, y, z).

4.6.3 Funcionamento da matrix

O diagrama de fluxo de dados (DFD) da Figura 4.5 apresenta o funcionamento da Matrix de forma genérica, cada passo é descrito a seguir:

1) O programa ou aplicativo gráfico que utiliza a Matrix, passa as informações de vídeo e os dados geométricos dos objetos, como parâmetros das funções da Biblioteca Gráfica Matrix;
2) A Matrix processa as informações e os dados recebidos, e depois envia para a tela do monitor as características com que o modo gráfico deve ser utilizado, e o objeto ou cena a ser exibido; e,

3) Os objetos estão visíveis na tela do monitor.

Figura 4.5 - DFD nível 0 da Matrix.

Na Figura 4.6, o DFD apresenta o funcionamento da Matrix, mostrando a ordem e quais funções podem ser utilizadas. Cada passo é descrito a seguir:

1) O programa ou aplicativo gráfico que utiliza a Matrix, passa as informações de vídeo como parâmetros da função que inicia o modo gráfico, e que foi descrita na seção 4.6.2 como Iniciar_Matrix;
2) A função Iniciar_Matrix processa estas informações e inicia o modo gráfico enviando para a tela do monitor as características desejadas;
3) Os dados geométricos dos objetos são passados pelo programa gráfico, como parâmetros para as funções de transformação;
4) As funções de transformação, como translação, rotação e escala são executadas, e os dados transformados são enviados como parâmetros para as funções de desenho;
5) As funções utilizam os dados transformados para montar os objetos, e depois envia os objetos para a tela do monitor; e,
6) Os objetos estão visíveis na tela do monitor.

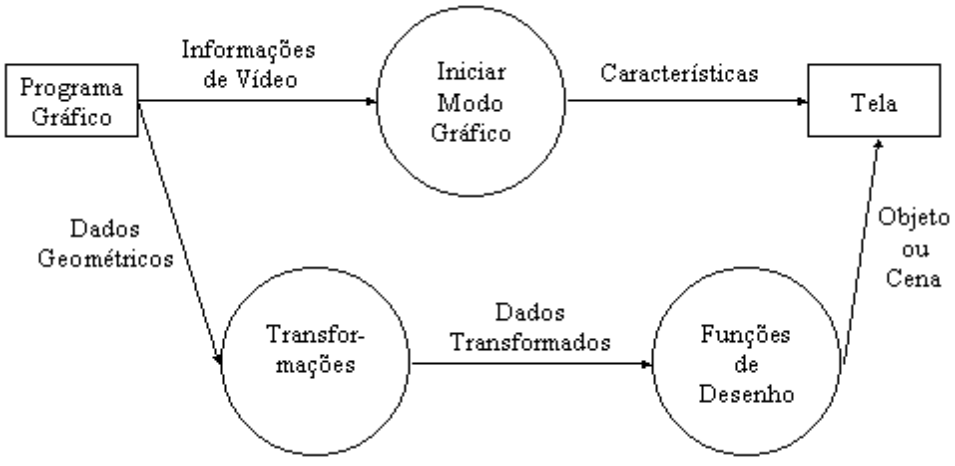

Figura 4.6 – DFD nível 1 da Matrix.

5 CONSIDERAÇÕES FINAIS

5.1 Resultados Gerais

O principal resultado deste trabalho foi a elaboração de uma estrutura para a Biblioteca Gráfica Matrix, apresentada no Capítulo 4, baseada no estudo de Bibliotecas Gráficas já existentes, levando em consideração como elas desenham e manipulam objetos gráficos.

Esta estrutura consiste basicamente da definição, descrição e especificação de um conjunto de funções gráficas para a Matrix.

5.1.1 Resultados específicos

Os estudos iniciais sobre técnicas da Computação Gráfica demostraram que existe uma relação muito próxima entre elas, e que sua utilização nem sempre é uma tarefa fácil.

Ficou evidente que as Bibliotecas Gráficas OpenGL e DirectX, agrupam as funções gráficas em componentes para permitir que os dados dos objetos sempre percorram um caminho pré-determinado em seu processamento. E que a Allegro, por ter uma estrutura mais simples, não faz este agrupamento.

A ausência de componentes permite uma estruturação mais simples, onde não há necessidade de uma comunicação intensa com o *hardware* (placa de vídeo), e de uma gerência do processamento dos dados dos objetos.

Para validar a estrutura proposta foi implementado um protótipo operacional da Matrix, que foi testado em diferentes computadores, apresentando um resultado satisfatório. Neste protótipo, não está disponível a função de *anti-aliasing*, devido à complexidade de sua

implementação e ao pouco tempo disponível para a realização deste trabalho. Não foi possível apresentar neste trabalho as telas de execução do protótipo, porque ele não é executado num sistema de janelas, não permitindo assim, a captura de imagens.

5.2 Trabalhos Futuros

Os resultados alcançados na realização deste trabalho não determinam o fim da Matrix, mas na verdade, abrem novas possibilidades para a realização de trabalhos futuros, dentre elas podemos ressaltar:

- Realizar estudos e implementações de técnicas para o desenho de esferas, cilindros, cones, e outros objetos 3D;

- Realizar estudos e implementações de técnicas de iluminação e sombreamento de objetos;

- Pesquisar, definir e implementar um pipeline de renderização de objetos.

5.3 Conclusão

Observa-se com a conclusão deste trabalho, que foi desenvolvida uma Biblioteca Gráfica simples e de fácil utilização.

A estrutura da Matrix e seu conjunto de funções gráficas, permitem o desenho e a manipulação de objetos gráficos em computador. A inclusão de novas funções gráficas pode ser feita sem modificar a estrutura, e estas novas funções podem utilizar as funções já existentes permitindo assim, uma integração entre elas.

A análise das execuções do protótipo operacional implementado valida a estrutura proposta, mostrando que seu funcionamento é eficaz e eficiente.

Espera-se com estes resultados que os futuros trabalhos a serem realizados na expansão e aprimoramento da Matrix sejam úteis no contexto da Computação Gráfica de nosso país, em que a pesquisa e desenvolvimento nesta área ainda são pequenos, disponibilizando um material teórico e prático sobre Bibliotecas Gráficas.

BIBLIOGRAFIA

[1] FOLEY, James D.; VAN DAM, Andries; FEINER, Steven K.; HUGHES, John F. **Computer Graphics**: Principles and Practice. 2. ed.
[2] Grupo de Desarrollo en Allegro - DirectX. Disponível em <http://gda.utp.edu.co:9673/gda/documentacion/programacion_3d/directx> Acesso em 24 set. 2004.
[3] Introdução à computação gráfica com OpenGL. Disponível em <http://www.dca.ufrn.br/~ambj/ele435/opengl/> Acesso em 10 set. 2004.
[4] HEARN, Donald; BAKER, M. Pauline. **Computer Graphics**: C Version. New Jersey: Prentice Hall. 1997.
[5] Computação Gráfica - FASP. Disponível em <http://www.fasp.br/docente/graduacao/anacristina/cg.html> Acesso em 9 ago. 2004.
[6] Grupo de Base de Dados e Imagens – ICMC / USP. Disponível em <http://gbdi.icmc.sc.usp.br/documentacao/apostilas/cg/downloads/apostilas.pdf> Acesso em 2 ago. 2004.
[7] COPPE-UFRJ. Disponível em <http://orion.lcg.ufrj.br/compgraf1/downloads/IntroCG.pdf> Acesso em 2 ago. 2004.
[8] Programa Especial de Treinamento - PET/AGRO - UFSC. Disponível em <http://www.inf.ufsc.br/~awangenh/CG/apostilas/apostilaport.pdf> Acesso em 9 ago. 2004.
[9] LCG / UFRJ - Laboratório de Computação Gráfica. Disponível em <http://orion.lcg.ufrj.br/compgraf2/downloads/lcg/LCG_Texturas.pdf> Acesso em 2 ago. 2004.
[10] Grupo de Computação Gráfica - UFRGS. Disponível em <http://www.inf.ufrgs.br/cg/teaching/inf01009/2002/Textura.pdf> Acesso em 9 ago. 2004.
[11] OpenGL. Disponível em <http://www.opengl.org> Acesso em 10 set. 2004.
[12] Introdução à OpenGL. Disponível em <http://www.inf.pucrs.br/~manssour/OpenGL/index.html> Acesso em 10 set. 2004.
[13] Tutorial do OpenGL. Disponível em <http://www.ingleza.com.br/opengl/1.html> Acesso em 10 set. 2004.
[14] Biblioteca Gráfica OpenGl. Disponível em <http://www.inf.pucrs.br/~pinho/CG/Aulas/OpenGL/OpenGL.html> Acesso em 10 set. 2004.
[15] DirectX. Disponível em <http://www.microsoft.com/directx/> Acesso em 24 set. 2004.
[16] Allegro. Disponível em <http://www.talula.demon.co.uk/allegro/> Acesso em 11 out. 2004.
[17] Allegro.cc. Disponível em <http://www.allegro.cc/> Acesso em 11 out. 2004.
[18] Wotsit's Format. Disponível em <http://www.wotsit.org/> Acesso em 8 nov. 2004.

APÊNDICE A – IMAGENS PCX

O formato de imagem PCX foi uma das primeiras tentativas da indústria de PCs (Personal Computer) em produzir uma padrão para troca e armazenamento de imagens. O formato PCX é um exemplo clássico de um padrão de fato criado a partir de uma iniciativa da indústria. [18]

Os arquivos PCX possuem um cabeçalho de tamanho fixo (*128 bytes*), seguido de um número variável de *bytes* contendo os valores dos *pixels* que compõem a imagem. Ao final do arquivo pode existir uma estrutura destinada a armazenar informações relativas à paleta de cores em uso.

A declaração da estrutura PCX_File exemplifica e detalha todas as variáveis necessárias para possibilitar a leitura de um cabeçalho PCX, como descrito abaixo:

```
struct PCX_File
{
   BYTE Header;
   BYTE Version;
   BYTE Encode;
   BYTE BitPerPix;
   WORD X1;
   WORD Y1;
   WORD X2;
   WORD Y2;
   WORD Hres;
   WORD Vres;
   PCXCOLOR EGAPalette[16];
   BYTE VideoMode;
   BYTE ColorPlanes;
   WORD BytesPerLine;
   BYTE Unused[60];
};
```

Descrição das variáveis:

- BYTE Header: deve sempre conter o valor "10" (dez) decimal para identificar o arquivo como sendo um PCX válido;

- BYTE Version: contém o número da versão utilizada para codificar o arquivo. O valor 0 (zero) indica que a versão é a *2.5*, o valor 2 indica versão *2.8* com paleta estendida, o valor 3 indica versão *3.0* sem paleta e o valor 5 indica versão *3.0* com paleta;

- BYTE Encode: contém a informação sobre o tipo de compressão utilizada na área de imagem. Se a imagem estiver comprimida no esquema RLL (Run Length Limited), o valor deste campo será sempre diferente de 0 (zero);

- BYTE BitPerPix: determina o tipo de codificação de cores utilizado no arquivo, ou seja, o número de bits por pixel;

- WORD X1 e WORD Y1: sempre serão de valor 0 (zero), a origem destes campos remonta ao formato PCC, ora em desuso, que permitia a localização de imagens parciais em áreas específicas da tela;

- WORD X2 e WORD Y2: contém o tamanho em pixels da imagem contida no arquivo;

- WORD Hres e WORD Vres: foram criados em função das diversas relações de *altura x largura* disponíveis nas placas de vídeo do início dos anos 80, tendo agora, conteúdo irrelevantes;

- BYTE ColorPlanes: as antigas placas EGA (Enhanced Graphics Adapter) e VGA (Video Graphics Adapter) possuíam um intrincado esquema de formação de cores baseado em planos, assim uma placa EGA operando em 16 cores seria capaz de apresentar imagens codificadas em 1 bit por pixel em 4 planos. Tornou-se obsoleto, pois os arquivos atuais contém quase sempre 8, 16 ou 24 bits por pixel e 1 plano de visualização;

- BYTE VideoMode e WORD BytesPerLine: tornaram-se obsoletos e continham respectivamente o modo padrão de operação da placa de vídeo bem como o número de *bytes* por linha descomprimida.

Sendo o PCX um padrão de fato e não regulamentado por nenhuma Instituição, alguns fabricantes optaram por utilizar alguns campos obsoletos ou até mesmo a área livre de 60 *bytes* ao final do cabeçalho para armazenar informações proprietárias e exclusivas de seus aplicativos.

A estrutura PCXCOLOR representa a estrutura de armazenamento das cores da paleta EGA e VGA estendida, contidas no cabeçalho ou no final do arquivo PCX respectivamente. Atualmente alguns destes campos caíram em desuso em virtude dos novos ambientes operacionais ou pela evolução tecnológica das placas de vídeo dos PCs.

```
struct PCXCOLOR
{
```

```
   BYTE Red;
   BYTE Green;
   BYTE Blue;
};
```

Esta estrutura deve estar presente e conter informações válidas para imagens de 16 cores (no cabeçalho) e para imagens de 256 cores (no final do arquivo). Para imagens de 16.777.216 (24 bits) cores, as informações contidas nesta estrutura são irrelevantes.

O campo PCXCOLOR EGAPalette[16], da struct PCX_File contém as informações de cor para imagens de 16 cores. As cores são definidas pela combinação de valores RGB (Red, Green, Blue) que podem variar de intensidade 0 a 63. A leitura dos 48 *bytes* do cabeçalho carrega a tabela de cores relativas aos pixels da imagem, assim sendo uma tabela que contenha os seguintes valores:

```
struct PCXCOLOR
{
BYTE Red;          BYTE Green;          BYTE Blue;
          63                   0                    0
          0                    63                   0
          0                    0                    63
          28                   32                   14
          ...                  ...                  ...
};
```

Esta leitura indica ao aplicativo que os pixels de imagem de valor 0 (zero) devem ser apresentados na tela com a cor vermelha, os de valor 1 (um) devem ser apresentados em verde, os de valor 3 em azul, e assim sucessivamente para todos os 16 valores da tabela de cores EGA.

O mesmo raciocínio é válido para a paleta estendida, localizada ao final do arquivo e que contém o mesmo tipo de informação para imagens codificadas em 8 bits (256 cores). A paleta estendida inicia-se com uma *tag* (valor) de cheque, de valor decimal 12, indicando a presença de uma paleta estendida válida logo ao final da imagem. Seguidamente à esta *tag* encontram-se 768 bytes relativos às 256 cores (RGB) de cada um dos bytes da imagem. Diferentemente da paleta EGA, a paleta estendida é codificada em valores de 0 a 255 para intensidade de cores.

O formato PCX incorpora um padrão simples de compressão RLE (Run Length Encoding) que, tipicamente, é capaz de reduzir o tamanho dos arquivos de imagem em 50%. A compressão RLE é normalmente bastante eficiente, sua performance porém é altamente

dependente do tipo da imagem podendo, em casos extremos, aumentar o tamanho teórico do arquivo.

Na compressão RLE cada linha da imagem é comprimida separadamente. O formato PCX acomoda a representação de imagens por planos de cor, isto é, codifica a imagem colorida em três imagens separadas e distintas para as componentes RGB. Quando a imagem está armazenada em possui múltiplos planos a compressão RLE se dará para cada linha de cada plano separadamente. O formato PCX suporta até 4 planos, a saber: BGRI, respectivamente Blue, Green, Red e Intensity. Se a imagem possui apenas 1 plano então os dados da imagem referem-se ao índice de cores da paleta estendida ao final do arquivo, pois só pode ser de 256 cores (8 bits).

No esquema RLE o valor original de um pixel é substituído por um código de repetição, o valor a ser repetido, e o número de repetições. Assim se uma imagem possui grandes áreas contíguas, com a mesma tonalidade, o esquema de compressão RLE trará bons resultados. Por outro lado, se a imagem possui poucas áreas de tonalidade igual, o esquema RLE será pouco eficiente. O sinalizador de repetição e o contador de repetição são implementados num único *byte*. Os dois *bits* mais significativos são setados em 1 e os restantes 6 *bits* indicam o número de repetições no intervalo de 1 a 63. Se um *bit* de dado possui seus dois *bits* mais significativos setados (> 192), então obrigatoriamente ele será codificado como RLE mesmo que possua apenas uma repetição. A seqüência na Figura A.1 ilustra este esquema de compressão e seus resultados.

RLL		RLL binário		Expandido
197	200	11000101	11001000	200 200 200 200 200
194	56	11000010	00111000	56 56
void	120	void	01111000	120
193	289	11000001	100100001	289
201	12	11001001	00001100	12 12 12 12 12 12 12 12 12

Figura A.1 – Tabela de compressão RLE.

ANEXO A – MATERIAL PRÁTICO

Este anexo é um CD-ROM onde estão armazenados o arquivo texto desta monografia, no formato DOC do Microsoft Word 2000, e os arquivos executáveis e com código-fonte do protótipo operacional, nas extensões “.exe”, “.h” e “.c”.