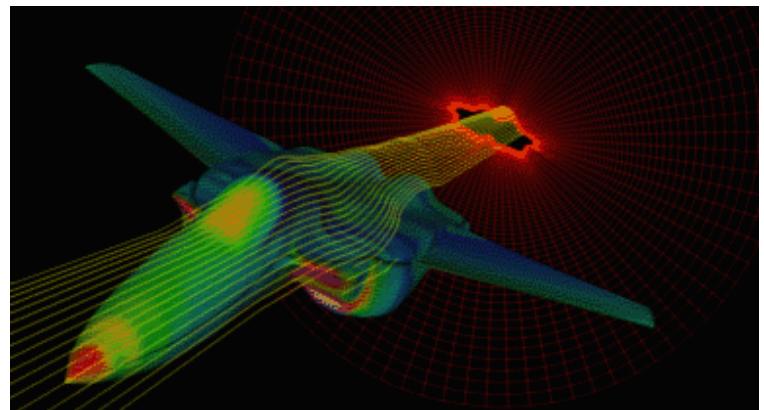
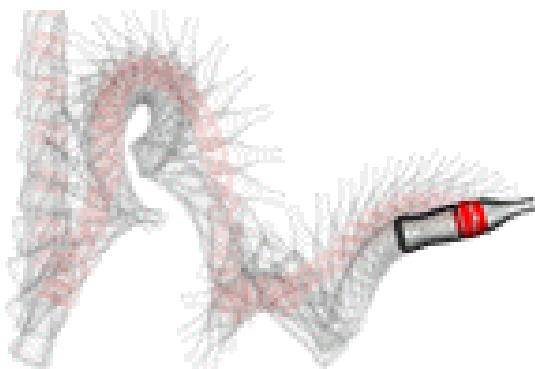
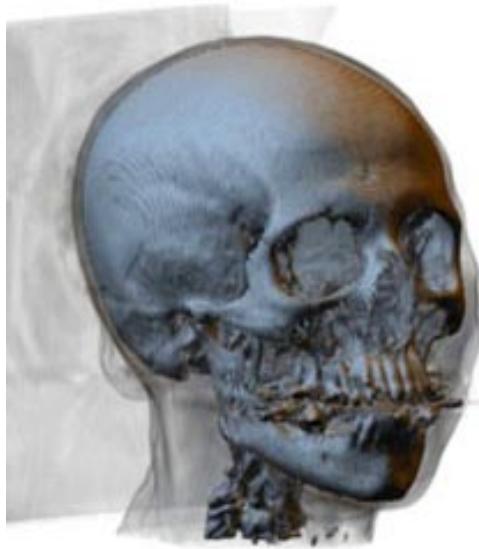


# *Computação Gráfica*



*Antonio Lopes Apolinário Júnior*

# Índice

---

<b>INTRODUÇÃO.....</b>	<b>1</b>
HISTÓRICO .....	1
CLASSIFICAÇÃO DAS APLICAÇÕES.....	1
<i>Projeto Assistido por Computador (Computer Aided Design - CAD).....</i>	2
<i>Visualização Científica.....</i>	4
<i>Processamento de Imagens.....</i>	7
<i>Visão Computacional .....</i>	8
<i>Animação.....</i>	8
<i>Realismo.....</i>	10
<i>Realidade Virtual.....</i>	11
<i>Interfaces Gráficas.....</i>	12
ESTRUTURA DO CURSO .....	13
<b>CAPÍTULO I ESTRUTURA DAS APLICAÇÕES GRÁFICAS .....</b>	<b>15</b>
ARQUITETURA TÍPICA DE UMA APLICAÇÃO GRÁFICA.....	15
<i>Hardware.....</i>	15
<i>Software.....</i>	17
PARADIGMAS DE ABSTRAÇÃO.....	19
<i>Universo Físico.....</i>	25
<i>Espaço Contínuo e Discreto.....</i>	25
<i>Universo Matemático .....</i>	27
<i>Universo de Representação .....</i>	28
<i>Universo de Implementação .....</i>	33
<i>Imagens .....</i>	35
PIPELINE DE UMA APLICAÇÃO GRÁFICA : .....	37
<i>Modelagem dos objetos .....</i>	38
<i>Transformações Geométricas .....</i>	38
<i>Recorte ou Clipping .....</i>	39
<i>Projeções.....</i>	39
<i>Rasterização .....</i>	40
<b>CAPÍTULO II OPENGL PRIMEIROS COMANDOS .....</b>	<b>42</b>
HISTÓRICO .....	42
FILOSOFIA.....	44
IMPLEMENTAÇÃO .....	52
<i>OpenGL, Delphi e Windows .....</i>	53
<i>Primitivas .....</i>	54
<i>Interface .....</i>	54
REFERÊNCIA.....	66
<b>CAPÍTULO III TRANSFORMAÇÕES GEOMÉTRICAS.....</b>	<b>75</b>
DEFINIÇÕES.....	75
TRANSFORMAÇÕES BIDIMENSIONAIS.....	76
<i>Escala.....</i>	76
<i>Rotação .....</i>	77
SISTEMA DE COORDENADAS HOMOGÊNEAS .....	81
TRANSFORMAÇÕES TRIDIMENSIONAIS .....	83
COMPOSIÇÃO DE TRANSFORMAÇÕES .....	84
TRANSFORMAÇÕES DE DEFORMAÇÃO .....	86
<i>Tapering .....</i>	87

<i>Twisting</i> .....	88
TRANSFORMAÇÕES GEOMÉTRICAS NA BIBLIOTECA <b>OPENGL</b> .....	90
<i>Manipulação das transformações</i> .....	90
Via funções específicas .....	91
Via Funções de manipulação direta da matriz corrente .....	91
<i>Estrutura de Pilha de Matrizes</i> .....	91
<i>Exemplo</i> .....	92
REFERÊNCIA.....	98
<b>CAPÍTULO IV SISTEMA DE VISUALIZAÇÃO .....</b>	<b>102</b>
CONSTRUÇÃO DO SISTEMA DE VISUALIZAÇÃO.....	102
<i>Posição da Camera no Espaço</i> .....	104
<i>Direção de Observação</i> .....	104
<i>Orientação da Camera</i> .....	105
<i>Tipo de Lente da Camera</i> .....	106
<i>Sistema de Coordenadas da Camera</i> .....	107
<i>Utilização de um Sistema de Visualização</i> .....	112
PROJEÇÕES GEOMÉTRICAS PLANARES .....	113
<i>Componentes Básicos de uma Projeção</i> .....	114
<i>Tipos de Projeção</i> .....	116
<i>Projeção Paralela Ortogonal</i> .....	118
<i>Matriz de Projeção Paralela Ortogonal</i> .....	120
<i>Projeção Perspectiva</i> .....	121
<i>Matriz de Projeção Perspectiva</i> .....	121
<i>Projeção Perspectiva x Projeção Paralela</i> .....	123
Perspectiva .....	123
Paralela .....	124
APLICAÇÃO NO OPENGL .....	124
REFERÊNCIA.....	133
<b>CAPÍTULO V MODELO DE ILUMINAÇÃO .....</b>	<b>136</b>
INTRODUÇÃO .....	136
MODELO DE ILUMINAÇÃO.....	137
<i>Modelo Físico</i> .....	138
<i>Modelo de Reflexão Local</i> .....	140
Componente Ambiente.....	141
Componente Difusa .....	141
Componente Especular.....	145
Composição da Componentes.....	148
<b>CAPÍTULO VI ALGORITMOS DE COLORAÇÃO .....</b>	<b>150</b>
ALGORITMO FLAT .....	151
ALGORITMO DE GOURAUD.....	153
ALGORITMO DE PHONG.....	157
APLICAÇÃO OPENGL.....	159
REFERÊNCIA.....	163
<b>CAPÍTULO VII REALISMO E MODELOS GLOBAIS.....</b>	<b>167</b>
MODELOS LOCAIS .....	167
<i>Mapeamento de Texturas</i> .....	167
Mapeamento de Ambiente (Environment Mapping) .....	171
<i>Bump Mapping</i> .....	173
<i>MODELO DE ILUMINAÇÃO GLOBAL</i> .....	175
<i>Ray Tracing</i> .....	175
Algoritmo de Ray Tracing.....	177
<i>Radiosidade</i> .....	180

# **INTRODUÇÃO**

---

## **Histórico**

Computação Gráfica é a área da Ciência da Computação preocupada com a manipulação e visualização de objetos reais ou imaginários. Para tanto ela se vale de estruturas de dados, algoritmos e modelos matemáticos e físicos que permitem a representação computacional desses objetos.

No início da década de 80 a Computação Gráfica ainda era uma área muito pouco explorada. Dois fatores básicos levavam a essa situação :

Alto custo dos equipamentos envolvidos;

Custo computacional elevado (para os padrões da época) de seus algoritmos.

Com os avanços tecnológicos a relação custo/capacidade computacional dos recursos exigidos em aplicações gráficas caiu bastante. Hoje temos computadores pessoais capazes de suprir os requisitos de processamento de modelos de média complexidade, com resultados bastante razoáveis. Outro fator que trouxe a Computação Gráfica mais próxima dos usuários “comuns” foi a grande popularidade que os sistemas com interfaces gráficas incorporadas (X-Windows/Unix, Windows, Macintosh, etc) alcançaram. Esses sistemas possibilitaram um grande avanço para os desenvolvedores, visto que grande parte das dificuldades inerentes a manipulação do hardware gráfico já são resolvidas pelo próprio sistema de janelas (via drivers dos fabricantes).

Hoje a Computação Gráfica pode ser vista como uma poderosa ferramenta de auxílio as mais diversas áreas do conhecimento humano. Essa diversidade foi responsável em criar áreas de especialização dentro da Computação Gráfica, cada uma preocupada em atacar classes diferentes de problemas, com requisitos distintos. A seguir apresentamos uma classificação para essas áreas de especialização.

## **Classificação das Aplicações**

Nesse seção a classificação feita para as aplicações não pretende ser completa, mas sim apresentar as áreas mais significativas. É importante também deixar claro que, apesar dessa “separação” em áreas de especialização a Computação Gráfica continua sendo uma área única na Ciência da Computação.

Nas aplicações que serão mostradas como exemplos, ficará claro que várias técnicas tem de ser combinadas para que o resultado final seja satisfatório.

### *Projeto Assistido por Computador (Computer Aided Design - CAD)*

Nessa classe de aplicações a Computação Gráfica tem como função básica fornecer subsídios aos projetistas - tipicamente engenheiros, arquitetos e designers – para a construção e manipulação de modelos de objetos, em sua grande maioria, manufaturáveis. Na figura 1 temos dois exemplos de objetos gerados por pacotes de CAD.

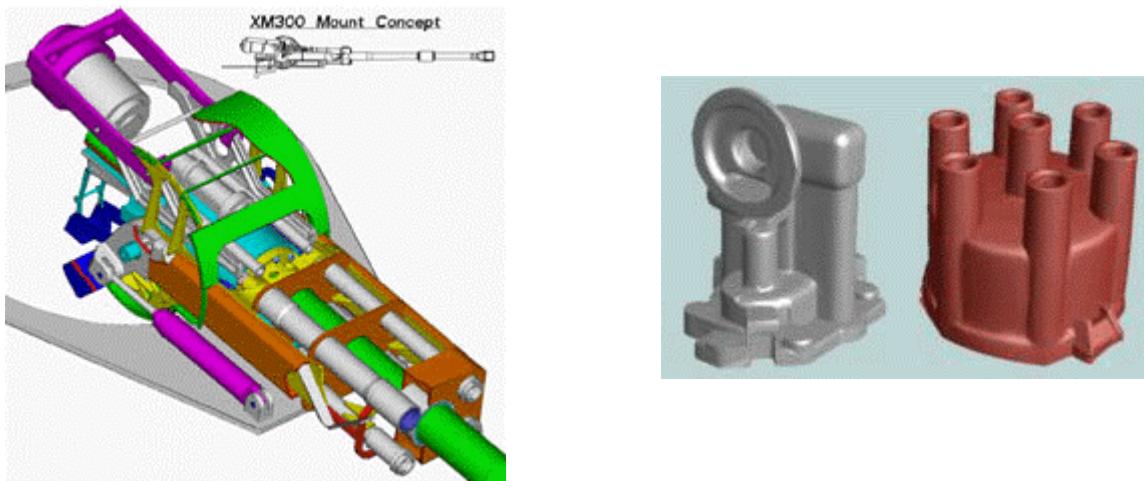
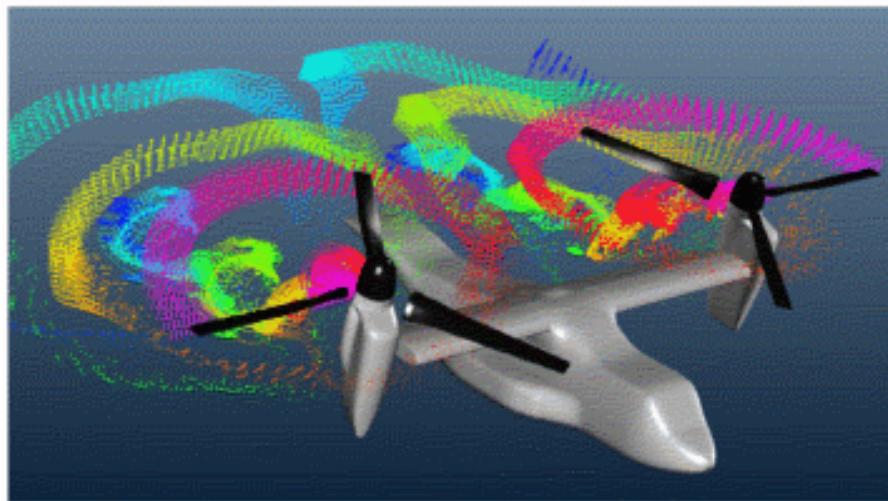
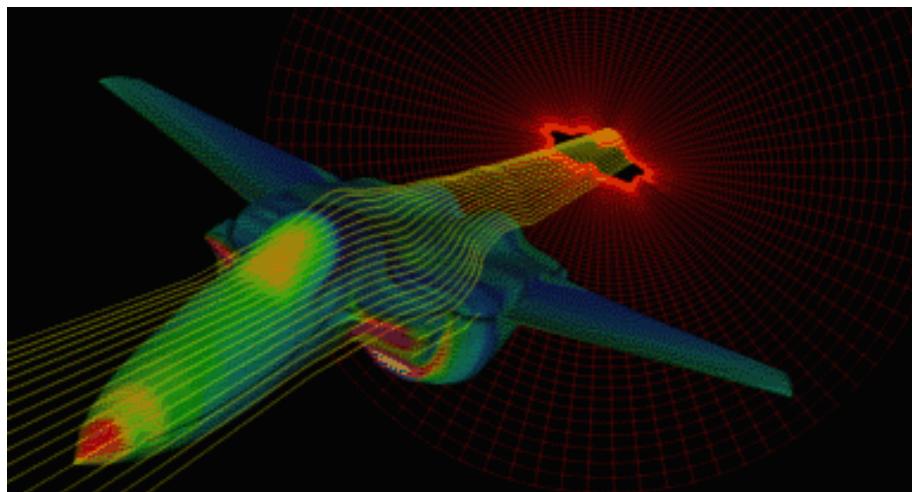


Figura 1 – Exemplos de objetos mecânicos gerados através de aplicações CAD.

Uma aplicação dessa classe deve ser capaz de, com base nos dados armazenados do objeto (geométricos e topológicos), produzir novas informações, relevantes para o usuário. Além disso, a aplicação deve ser capaz de garantir a validade do modelo, visto que este será mais tarde construído fisicamente.

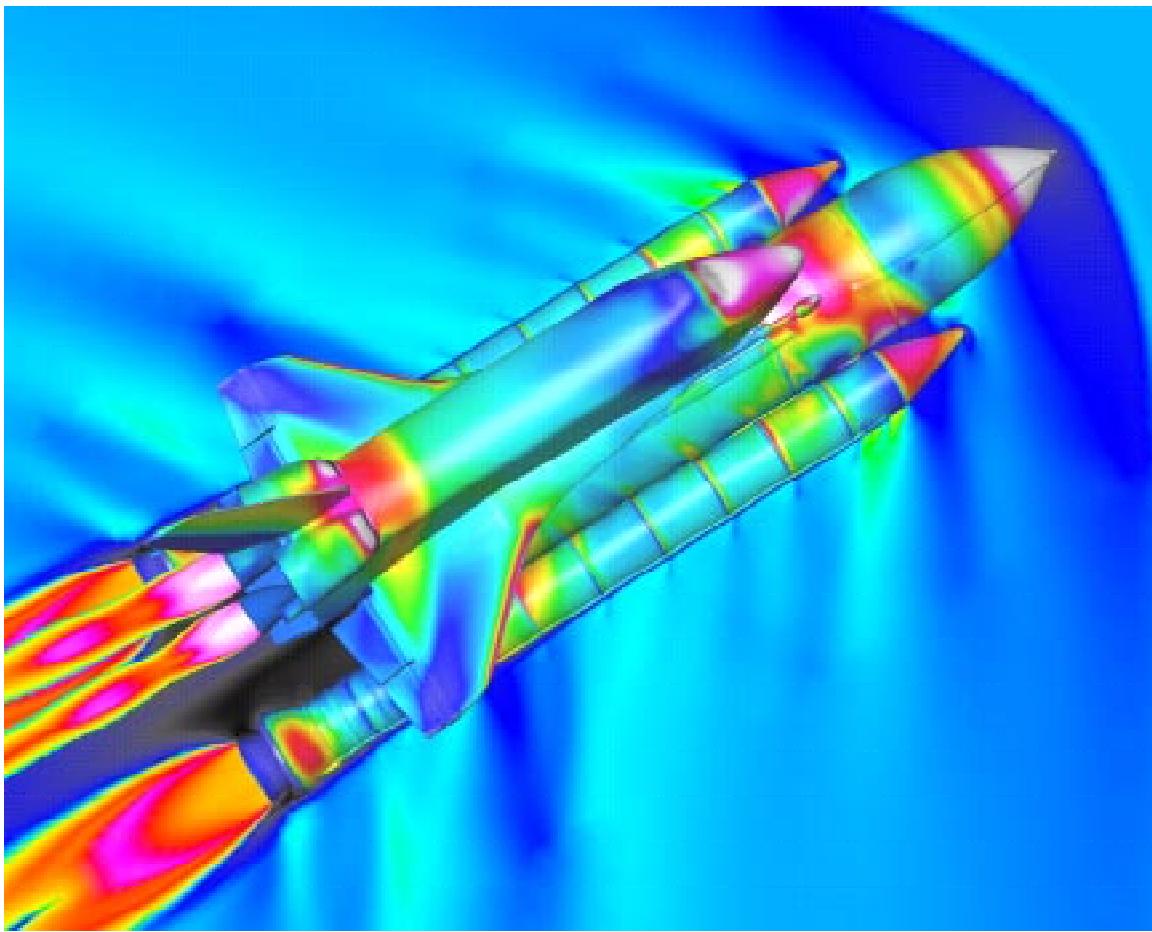
Outro exemplo de aplicação típica dessa área pode ser visto na figura 2. Nela podemos ver dois modelos de aeronaves. Nos dois casos uma análise do comportamento de suas estruturas aerodinâmicas está sendo feita. Com base nas informações da geometria dos objetos pode-se aplicar modelos de engenharia e simular, por exemplo, um estudo sobre do arrasto aerodinâmico das estruturas.



**Figura 2 - Dois exemplos de análises aerodinâmicas, utilizando como base modelos gerados por computador em sistemas CAD.**

Na figura 3 temos um outro tipo de análise, baseado ainda na geometria do modelo. Nela temos realçadas por uma escala de cor as regiões da estrutura do ônibus espacial sujeitas a diferentes graus de tensão.

Deve ficar claro que nesse tipo de aplicação a Computação Gráfica irá participar não apenas na geração das imagens, mas principalmente fornecendo uma estrutura consistente para representação da geometria dos objetos. Essa estrutura deverá interagir com procedimentos específicos da área da aplicação para as análises do usuário.



**Figura 3 - Visualização dos esforços aos quais a estrutura do ônibus espacial é submetida durante a decolagem. O mapeamento de cores da idéia do grau de esforço da estrutura.**

### *Visualização Científica*

A análise dos resultados em modelos científicos em geral é uma tarefa bastante complexa. A Computação Gráfica tem sido uma ferramenta de grande auxílio. O motivo é bastante óbvio : a análise de dados numéricos requer muito mais trabalho, experiência e poder de abstração do que quando feita com base no mapeamento desses mesmos dados em forma de imagem. Esse fato torna-se ainda mais crítico se o fenômeno a ser estudado distribui-se no espaço 3D - a realidade da grande maioria das aplicações na área científica. Exemplos de utilização de técnicas de visualização científica podem ser visto nas figuras 4, 5 e 6.

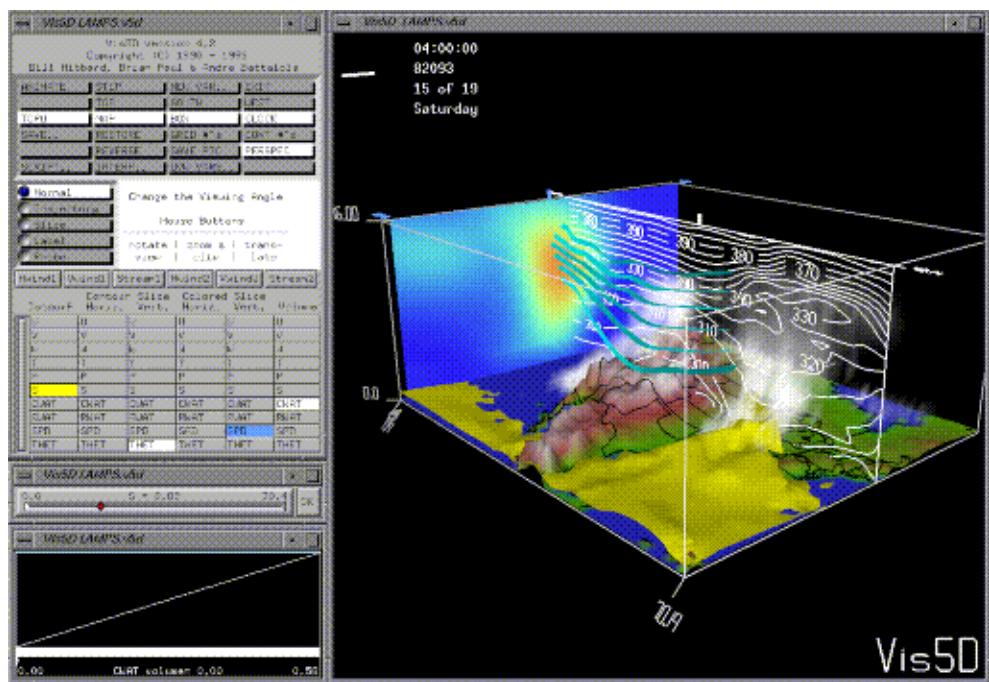


Figura 4 - Exemplo de um sistema para visualização de dados meteorológicos. A janela à direita mostra gráficos superpostos das várias variáveis envolvidas na modelagem meteorológica.

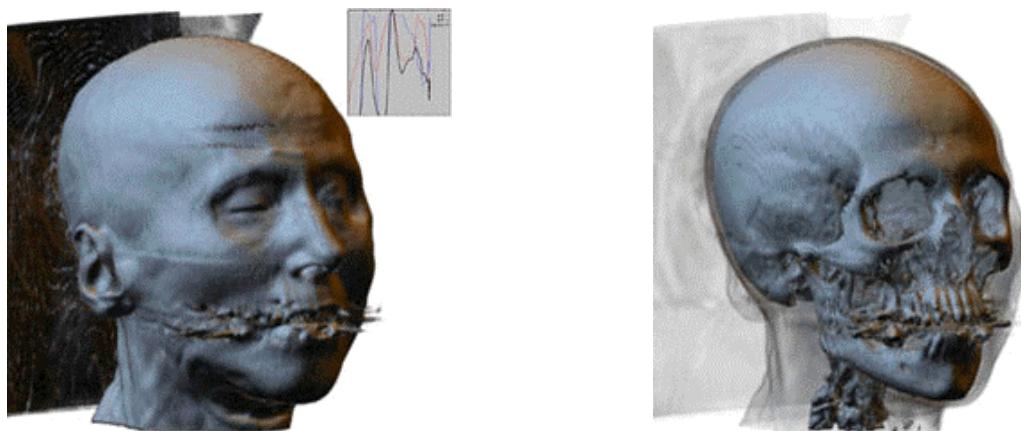
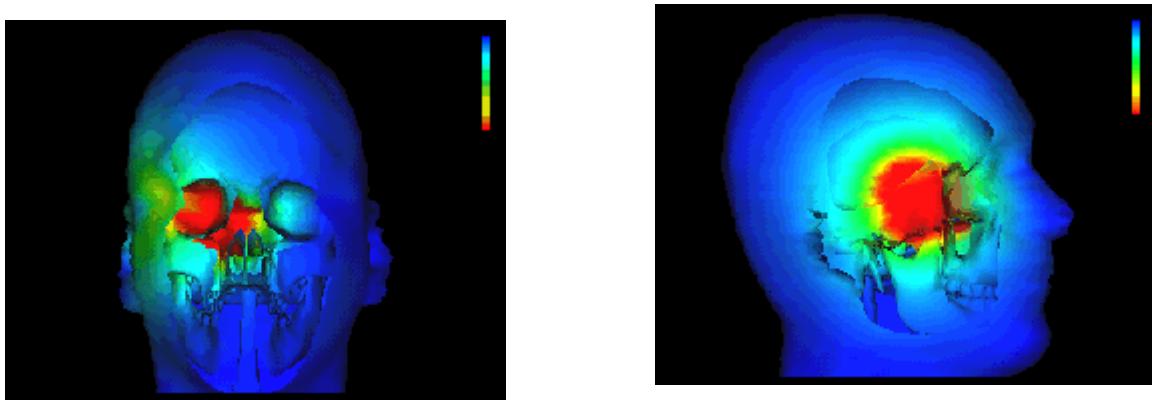


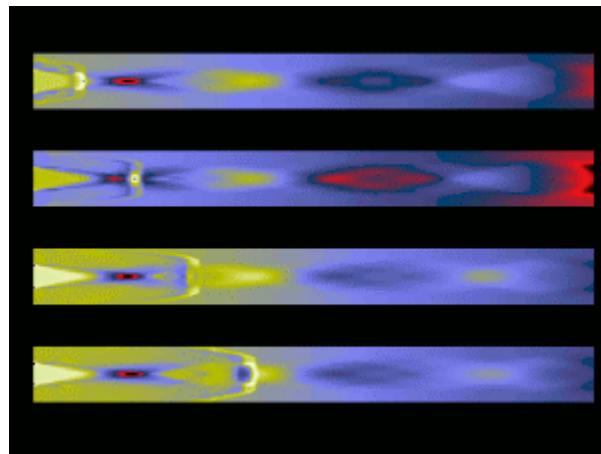
Figura 5 - Duas imagens reconstruídas a partir de resultados colhidos por exames de ressonância magnética. À esquerda é dado um tom opaco a pele, enquanto que à direita a camada de pele possui um certo grau de transparência. Dessa forma é possível visualizar a estrutura óssea .



**Figura 6 - Outro exemplo de visualização na área médica.** Nesse caso dois pontos de vista diferentes do mesmo objeto. A variação das cores é utilizada para diferenciar regiões com propriedades diferentes.

Nessa classe de aplicações o papel da Computação Gráfica é bem menor visto que a geração e manutenção dos modelos não está sob sua responsabilidade. Cabe a área gráfica receber os dados finais do modelo e gerar uma imagem para posterior interpretação.

Além da geração de imagens estáticas, uma forma de análise comum nessa área é baseada em uma série temporal : um fenômeno é acompanhado em um intervalo de tempo e são coletados dados distribuídos no tempo. A Computação Gráfica pode tornar a análise desses dados bem mais interessante modelando essa característica dinâmica em uma seqüência de imagens animadas.

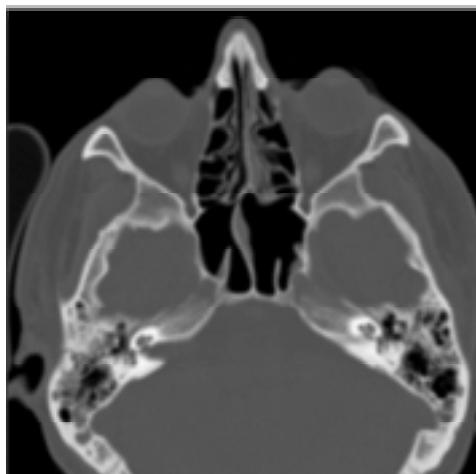


**Figura 7 - Seqüência de imagens que permite o acompanhamento do fenômeno dentro de um intervalo de tempo.**

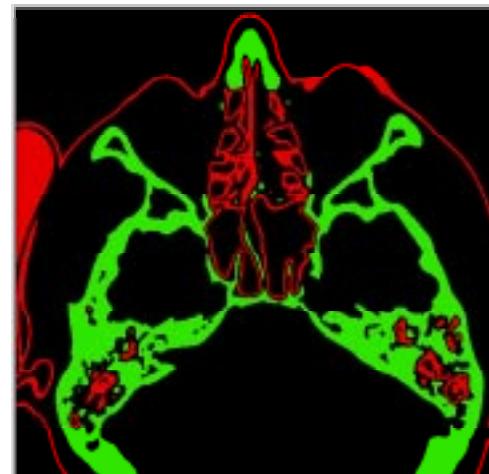
## *Processamento de Imagens*

Áreas com medicina, cartografia, sensoriamento remoto, astronomia entre outras se valem dessa área da Computação Gráfica para extrair informações de imagens.

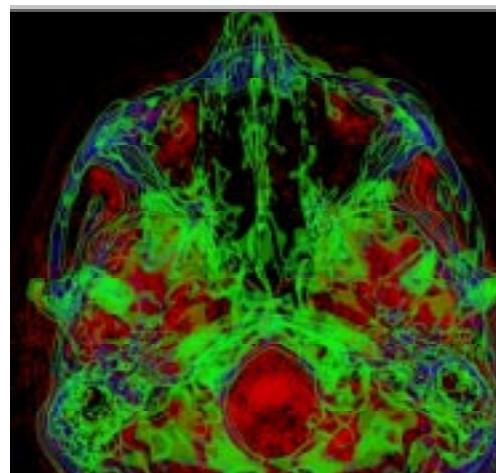
Essas imagens em geral são geradas externamente por outro tipo de aplicação. Com base em técnicas, que tem como base o processamento digital de sinal, essas imagens são tratadas e os resultados apresentados ao usuário para sua avaliação.



(a)



(b)



(c)

Figura 8 - Seqüência de imagens geradas por processamento da imagem. (a) imagem original gerada pelo método de ressonância magnética; (b) a mesma imagem com regiões de interesse realçadas; (c) padrão de cores diferente.

Essa área também é responsável pelo estudo de algoritmos para compressão de imagens.

### *Visão Computacional*

Essa área da Computação Gráfica tem por objetivo tentar reproduzir modelos 3D a partir de imagens bidimensionais. Ou seja, reproduzir computacionalmente o processo que ocorre no sistema visual humano.

A robótica se vale dessas técnicas para, por exemplo, avaliar o ambiente no qual um robô está imerso e poder tomar decisões sobre seu deslocamento dentro dessa área.

A área de cartografia também utiliza técnicas de visão para reproduzir, com base em imagens aéreas, por exemplo, um mapa de relevo de uma determinada região.

### *Animação*

Nessa área a preocupação é simular os controles existentes para a produção de filmes tradicionais através de sistemas automatizados. Manipulação de câmeras e "atores", geração automática de seqüências de quadros são algumas das preocupações nessa área.

O realismo dos movimentos também é importante, principalmente na simulação do comportamento personagens animados.

Essa área tem tido um crescimento bastante acentuado graças à indústria de divertimento. Nas figuras 9 e 10 temos exemplos de aplicações onde os efeitos de animação são aplicados : No cinema (9), na televisão para produção de vinhetas e comerciais (10a e 10b) e em videogames (10c e 10d).



Figura 9 - Cena do filme 'O Segredo do Abismo'. Técnicas de animação, combinadas a modelagem e visualização são empregadas para se obter o efeito da materialização do alienígena em água.

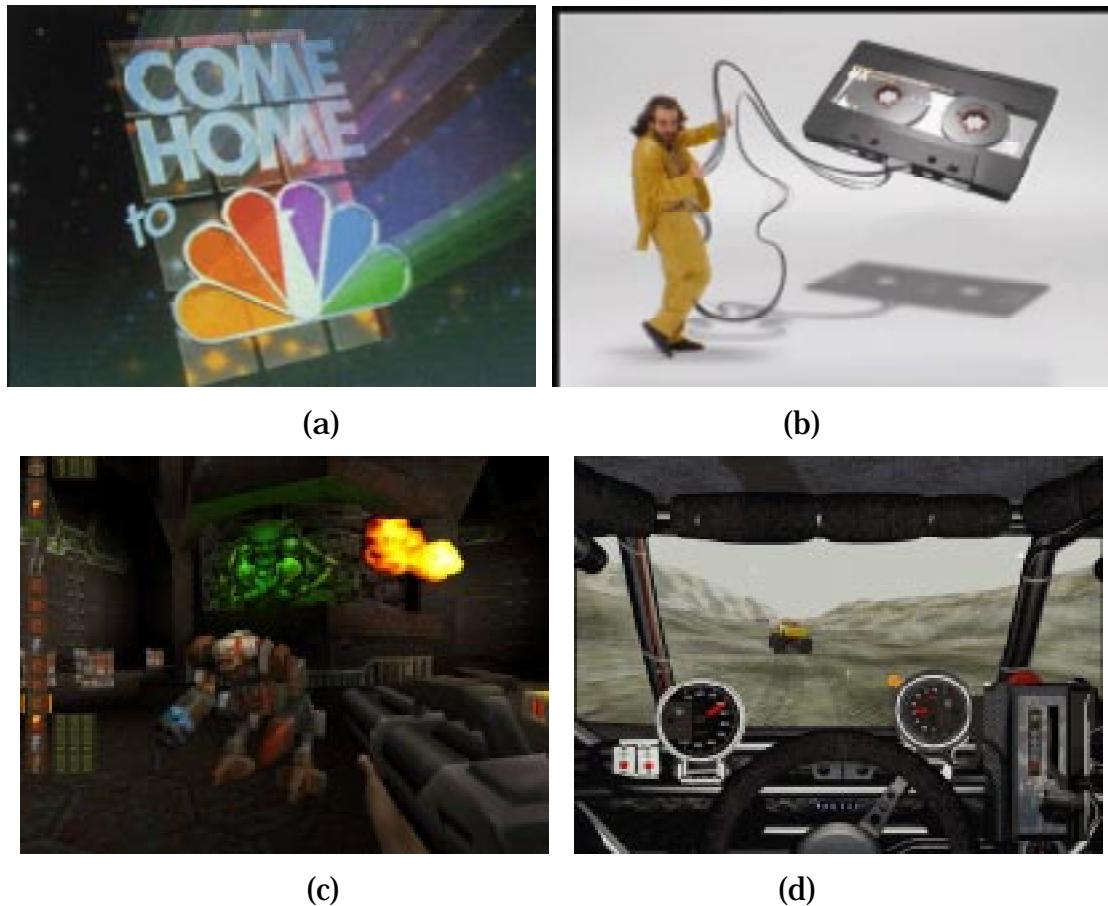


Figura 10 - Exemplos do uso de animação : (a) e (b) vinhetas e comerciais - com ênfase no sincronismo atores reais e sintéticos; (c) e (d) jogos eletrônicos - com ênfase na movimentação de personagens e objetos.

## *Realismo*

A preocupação desse segmento é construir modelos para geração de imagens com a aparência o menos sintética possível.

Para tanto modelos físicos cada vez mais sofisticados vem sendo utilizados para modelar as interações das fontes de luz com os objetos. Representações dos materiais também é importante para a caracterização do realismo de uma imagem.



**Figura 11 - Estudo de iluminação natural simulada pela técnica de radiosidade.**



Figura 12 - Estudo de iluminação de interiores utilizando a técnica de radiosidade.

### *Realidade Virtual*

Apesar de ainda não passar de uma promessa, a realidade virtual tem um relacionamento bastante estreito com a área gráfica. É necessário que uma representação realista desse mundo virtual seja gerada para "convencer" o usuário que ele está dentro de um "outro mundo".

A dificuldade nesse caso é que as máquinas domésticas hoje já tem capacidade para gerar imagens com relativo grau de realismo. No entanto aplicações de realidade virtual tem um requisito muito mais caro : geração de seqüências de imagens realistas em tempo real. Para tanto os equipamentos domésticos típicos ainda deixam muito a desejar.

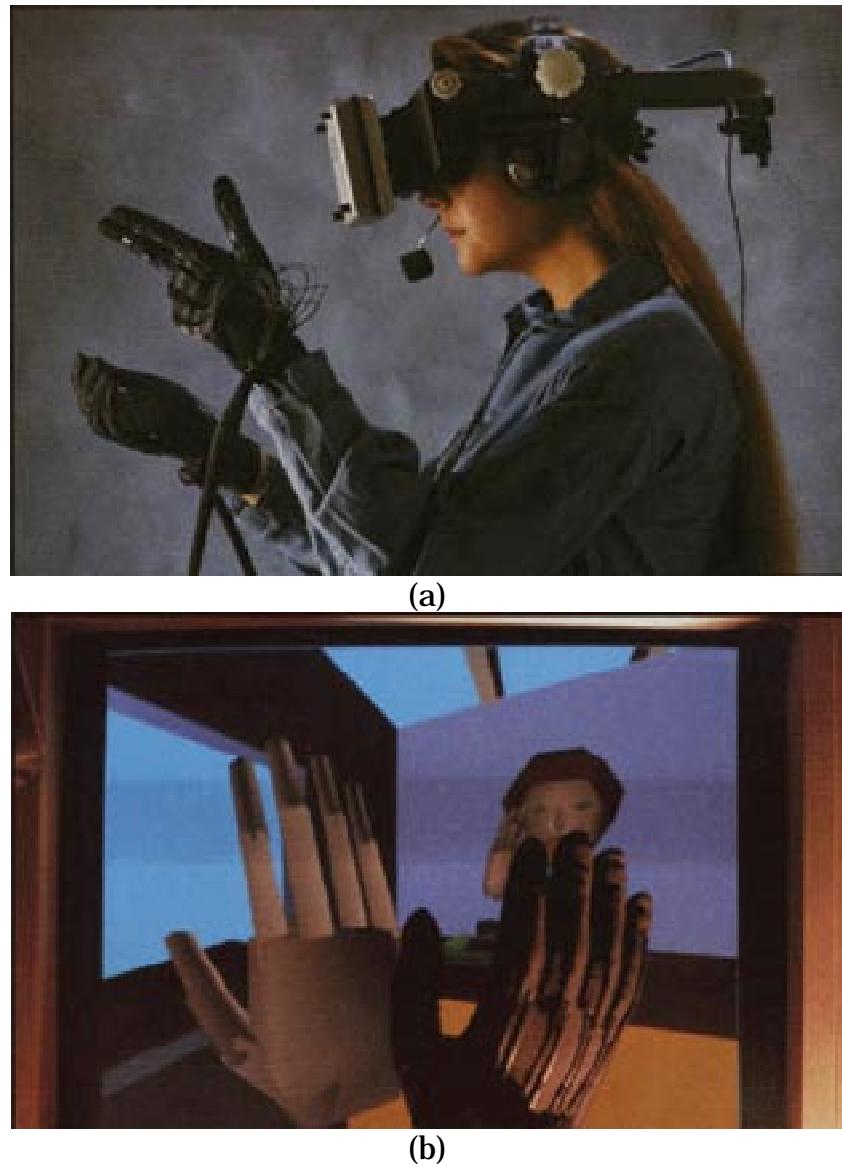
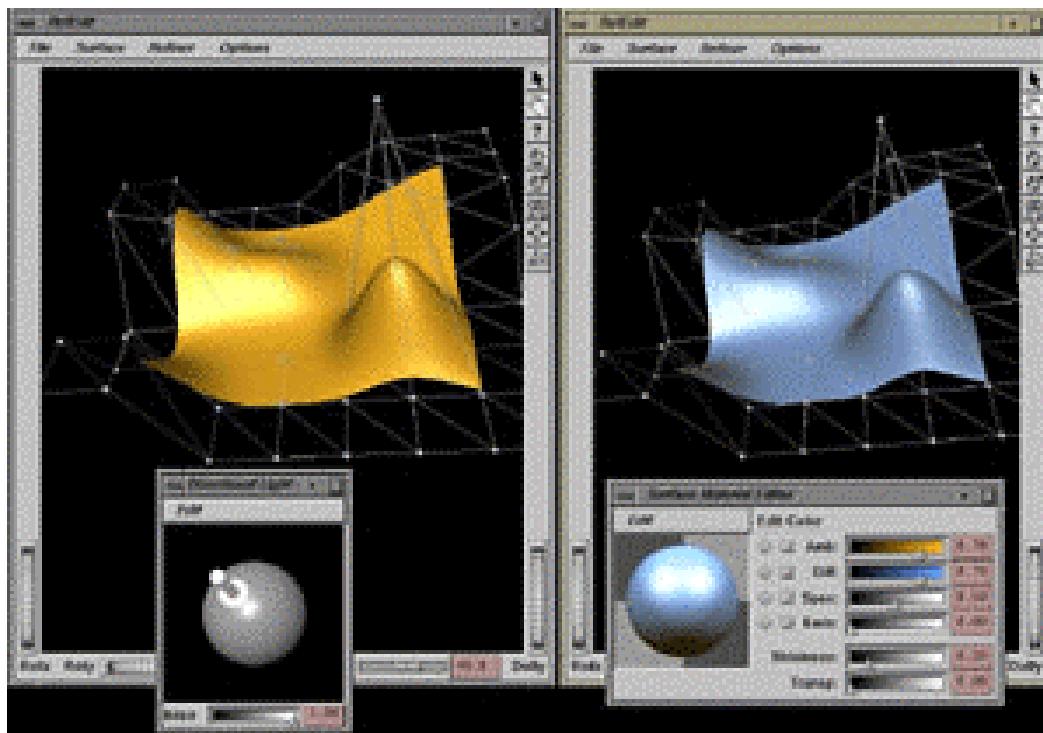


Figura 13 - (a) Dispositivos de hardware voltados para realidade virtual; (b) visualização de um "mundo virtual" onde a luva é mapeada.

### *Interfaces Gráficas*

Sem dúvida essa área contribuiu para viabilizar a Computação Gráfica de forma decisiva. O encapsulamento dos procedimentos de tratamento do hardware gráfico, tornou o trabalho de programação gráfica bem mais simples.



**Figura 14 - Exemplo de interface gráfica com vários tipos diferentes de formas de iteração, em uma aplicação voltada para a modelagem de superfícies.**

## Estrutura do Curso

Como podemos observar a Computação Gráfica hoje está envolvida em uma grande gama de aplicações. Dentro desse quadro tentaremos apresentar os principais processos envolvidos na geração de imagens sintéticas, em particular aquelas resultantes do processo de modelagem de objetos tridimensionais. Em geral esses processos básicos funcionam como alicerce da maioria das aplicações gráficas.

Iniciaremos nosso estudo analisando a organização de uma aplicação gráfica típica, não só sob o ponto de vista do hardware mas também em termos de software. Apresentaremos também um dos modelos mais populares para a representação de objetos tridimensionais : as malhas poligonais. Por fim o processo de geração de imagens de objetos será discutido e caracterizado como um pipeline com 4 estágios básicos.

No segundo capítulo daremos inicio a parte prática do curso. Nela vamos utilizar uma biblioteca gráfica no sentido de simplificar a construção de aplicações - o OpenGL. Nessa etapa apresentaremos a sua filosofia, bem como alguns conceitos básicos para a sua utilização. Um exemplo de aplicação será

discutido, com vistas a caracterizar a sua utilização dentro do ambiente de programação utilizado, nesse caso o Delphi 2.0.

Em seguida retomamos o estudo da teoria, analisando a utilização das transformações geométricas dentro da Computação Gráfica. Iremos apresentar as principais transformações geométricas e sua representação matricial. Em paralelo iremos analisar os dois sistemas de coordenadas mais utilizados : o Cartesiano e o Homogêneo; discutindo suas motivações e utilização. Mostraremos também como os conceitos apresentados são implementados dentro do OpenGL.

Em aplicações que manipulam objetos tridimensionais é comum o de definição de um observador ou uma "camera". Um Sistema de Visualização permite controlar entre outros parâmetros a posição, o tipo de projeção entre outras propriedades da camera. Nessa etapa iremos avaliar quais são esses parâmetros e mapear quais são suas influencias na imagem final. Uma visão teórica (através do estudo das transformações projetivas) e prática (a implementação de cameras virtuais dentro do OpenGL) será apresentada.

Para dar aos objetos uma aparência realista após a sua projeção, um estudo das influencias da iluminação na sua aparência se faz necessário. Nesse capítulo veremos, de forma breve, quais as componentes da luz influenciam de maneira determinante na imagem final dos objetos. Apresentaremos um modelo de reflexão e três variações clássicas para a sua implementação. Veremos que o OpenGL permite a manipulação de fontes de luz de forma simples e rápida.

## CAPÍTULO II

# OPENGL PRIMEIROS COMANDOS

---

### Histórico

A empresa *Silicon Graphics Inc.*, figura hoje como uma das maiores empresas no mercado de hardware / software voltado especificamente para aplicações gráficas. Ela lançou, no início da década de 90, a biblioteca gráfica *IRIS GL (Graphics Library)*, que inicialmente foi desenvolvida especialmente para a arquitetura *IRIS*, de sua propriedade, baseada no sistema operacional *UNIX*.

Devido a boa aceitação entre os seus usuários, a empresa decidiu investir em tornar essa biblioteca um padrão de mercado. Para tanto a desvinculou de arquitetura *IRIS* e tornou-a multiplataforma. Nascia aí a biblioteca ***OpenGL*** – de ***Open Graphics Library***.

Hoje um consórcio, do qual participam grande fabricantes de hardware e software, como *IBM*, *HP*, *Sun*, *Microsoft* e a própria *Silicon Graphics*, entre outras, administra a especificação da biblioteca.

Temos versões da ***OpenGL*** rodando nos principais sistemas operacionais disponíveis no mercado : *UNIX* em todos os seus "sabores" (*AIX*, *HP-UX*, *Solaris*, *Linux* (domínio público), entre outros), *Microsoft Windows* e *Machintosh*. Para alcançar esse grau de portabilidade a ***OpenGL*** não prevê em sua especificação funções que dependam de plataforma. O controle de eventos, manipulação de janelas, interações com o usuário, via teclado ou mouse, são tratados acionando-se as rotinas apropriadas da API do sistema operacional que estiver sendo utilizado. Com isso é garantido que a parte ***OpenGL*** de qualquer aplicação pode facilmente ser portada para qualquer sistema que possua uma implementação da biblioteca.

Reiterando a portabilidade de código, a ***OpenGL*** cria um mapeamento entre os tipos de dados utilizados por suas funções e os tipos de dados disponíveis na instalação. Esse mapeamento é apresentado na tabela 1.

<i>OpenGL</i>	<i>Tipo</i>	<i>C</i>	<i>PASCAL</i>	<i>Símbolo</i>	<i>Tamanho</i>
<b><i>GLbyte</i></b>	signed char		shortint	b	inteiro 8 bits
<b><i>GLshort</i></b>	short		smallint	s	inteiro 16 bits
<b><i>GLint,</i></b> <b><i>GLsizei</i></b>	long		integer	i	inteiro 32 bits
<b><i>GLfloat,</i></b> <b><i>GLclampf</i></b>	float		single	f	float 32 bits
<b><i>GLdouble,</i></b> <b><i>GLclampd</i></b>	double		double	d	float 64 bits
<b><i>GLubyte,</i></b> <b><i>GLboolean</i></b>	unsigned char		byte	ub	unsigned int 8 bits
<b><i>GLushort,</i></b>	unsigned short		word	us	unsigned int 16 bits
<b><i>GLuint,</i></b> <b><i>GLenum,</i></b> <b><i>GLbitfield</i></b>	unsigned long		cardinal	ui	unsigned int 32 bits

Tabela 1 – Tipos de dados abstratos criados pela biblioteca *OpenGL*.

Nas figuras 39 e 40 temos a arquitetura de uma aplicação *OpenGL* em dois ambientes distintos : *MS Windows* e *X-Windows (UNIX)*.

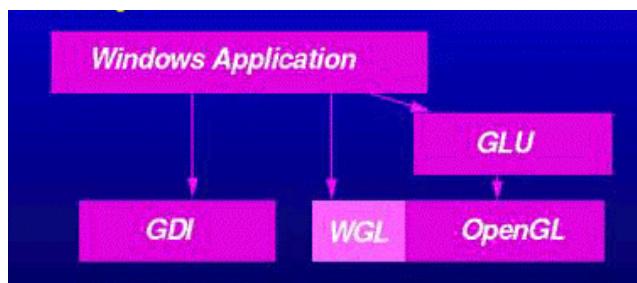


Figura 39- Arquitetura de uma aplicação OpenGL em ambiente MS Windows.

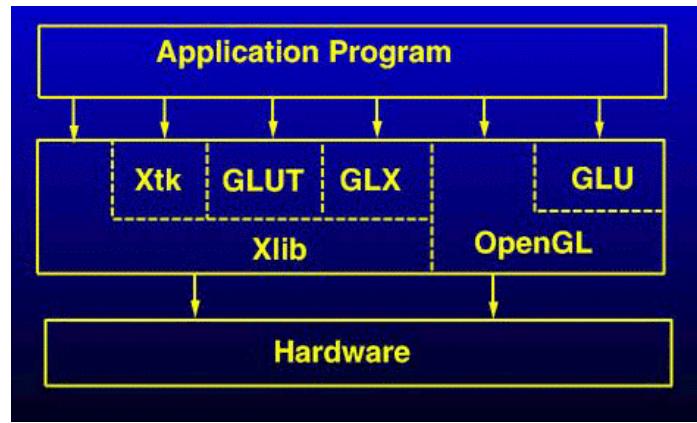


Figura 40 - Arquitetura de uma aplicação OpenGL em um ambiente X-Windows.

Hoje a ***OpenGL*** tornou-se a biblioteca de programação gráfica mais utilizada, principalmente em ambientes e aplicações de médio e grande porte. No ambiente *MS Windows* seu único concorrente é o *Direct3D* da própria *Microsoft*. A *Sun Microsystems* pretende entrar nesse mercado a partir do lançamento da linguagem *Java3D*, extensão da linguagem *Java* com suporte gráfico.

## Filosofia

A ***OpenGL*** trabalha prevendo sua utilização em uma arquitetura cliente/servidor. Nessa modalidade teremos um servidor ***OpenGL*** processando os comandos enviados pelo cliente. No cliente será feita a visualização do resultado que o servidor enviará. Esse suporte a arquitetura cliente/servidor foi herdado do ambiente de janelas *X-Windows (UNIX)* que também possui esse suporte. Essa arquitetura pode ser vista na figura 41.

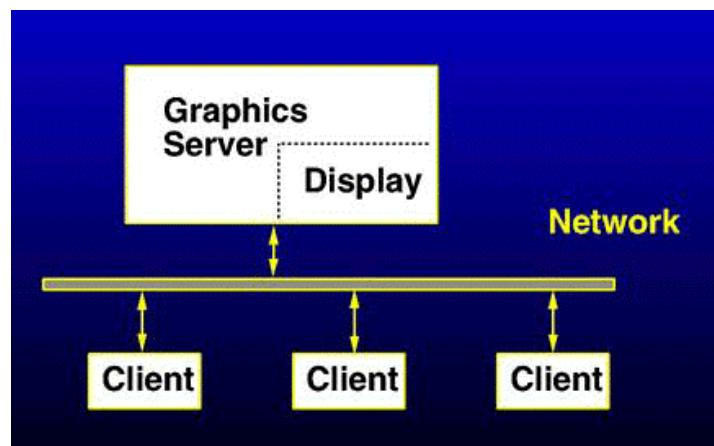


Figura 41 – Arquitetura cliente/servidor de uma aplicação OpenGL.

O projeto da ***OpenGL*** direcionado para caracterizar uma aplicação como uma máquina de estados. O ambiente ***OpenGL*** assume um determinado estado quando da sua criação. Esse estado permanece até que um evento induza uma mudança. A partir desse momento, a máquina assume um novo estado. Esse irá perdurar até que um novo evento ocorra e modifique novamente seu estado.

Um exemplo simples que demonstra esse princípio é a definição da cor de desenho dos objetos na ***OpenGL***. Uma vez definida uma certa cor todos os objetos serão desenhados com essa mesma cor até que um novo comando de definição de cor seja dado.

A vantagem desse enfoque é a redução no número de parâmetros necessários para definição dos objetos. Por *default* os estados correntes serão utilizados para seu desenho. Outra vantagem é diminuir o fluxo de informações que precisam transitar na rede (levando-se em conta a arquitetura cliente/servidor).

Como uma máquina de estados a ***OpenGL*** possui uma série de atributos que correspondem a algumas variáveis de estado. Com o decorrer do curso apresentaremos alguns desses atributos.

A ***OpenGL*** trabalha com o conceito de primitivas gráficas. São elas os elementos geométricos básicos disponibilizados pela biblioteca para que o usuário possa construir seus modelos. As principais primitivas suportadas são mostradas na figura 42. Apesar de estarem representadas em um plano, sua definição pode ser feita também no espaço tridimensional.

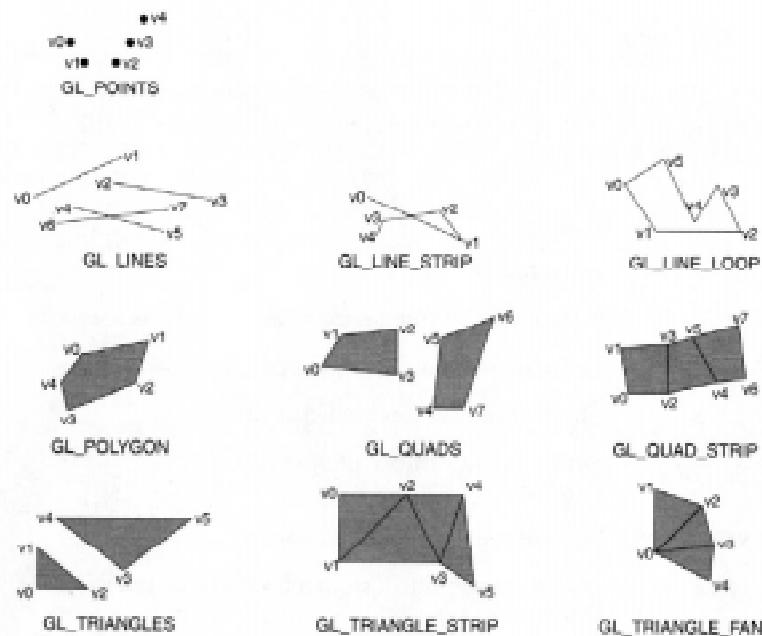


Figura 42 – Primitivas gráficas disponíveis na biblioteca *OpenGL*.

Ao mesmo tempo que esse número reduzido de primitivas facilita o aprendizado e simplifica o tratamento interno da biblioteca, torna-se complicado o trabalho de programação de aplicações que trabalham com modelos mais complexos. Por exemplo, formas básicas como cubos, esferas, cones, etc, devem ser construídas pelo programador com base nas primitivas oferecidas. Para simplificar esse trabalho, bibliotecas complementares foram desenvolvidas. As rotinas dessas bibliotecas auxiliares são construídas com base nas funções da própria ***OpenGL***. Na figura 43 temos algumas primitivas tridimensionais disponíveis em duas dessas bibliotecas auxiliares : ***glu*** e ***glaux***. Outras bibliotecas complementares se destinam a fornecer suporte a manipulação de janelas e eventos (***GLUT***), orientação à objetos (***OpenInventor***), construção de interfaces (***GLUI***), entre outras.

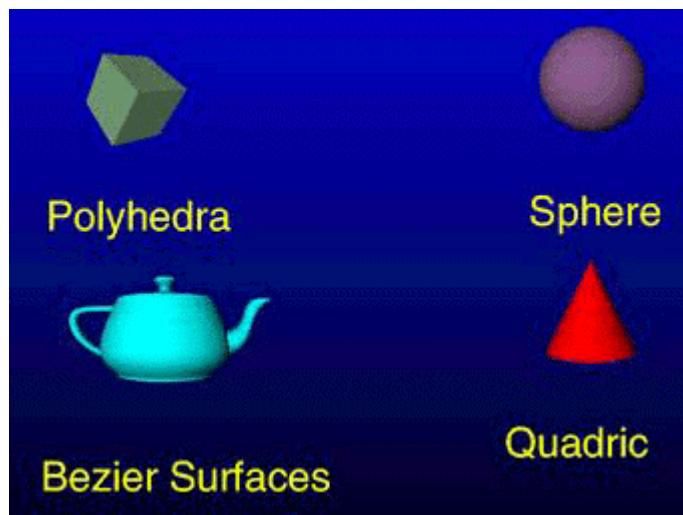


Figura 43 – Primitivas tridimensionais disponíveis nas bibliotecas *glu* e *aux*.

No processo de criação dos modelos temos que posicionar ou mesmo alterar a sua forma. Para tanto nos valemos de transformações geométricas como translações, rotações e escalas, entre outras. Essas transformações podem ser escrita na forma de matrizes. A ***OpenGL*** possui funções para manipulação dessas transformações, tanto em sua forma geral (matrizes) como através de transformações pré-definidas (rotação, translação e escala). Todo o processamento dessas matrizes é encapsulado dentro da biblioteca, de forma que basta o programador chamar a rotina específica, ou no caso mais geral, descrever a forma da matriz da transformação a ser aplicada.

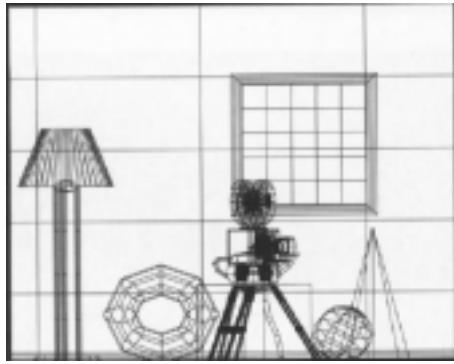
Em Computação Gráfica é comum termos um objeto organizado como um conjunto hierárquico de outros objetos. Um exemplo simples é um modelo de um ser humano. Seus braços, pernas, etc, seriam objetos de um modelo maior - seu corpo. Nesse caso é interessante que certas transformações tenham efeito em apenas grupos de objetos, enquanto outras sejam aplicadas no modelo como um

todo. Por exemplo, um modelo humano correndo deve, ao mesmo tempo que descola o seu corpo para frente, movimentar braços e pernas de forma diferenciada. Para implementar esse tipo de hierarquia nas transformações o **OpenGL** implementa uma estrutura de pilha para manipular as transformações geométricas. Mais tarde no capítulo 3 entraremos em detalhes sobre sua utilização.

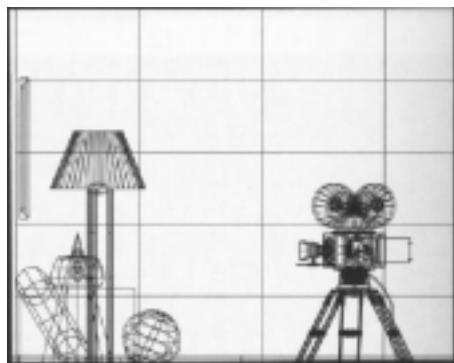
A grande força da **OpenGL** é seu suporte a manipulação de objetos definidos no espaço tridimensional. Nesse espaço o posicionamento do observador é um fator importante na definição da imagem a ser gerada. Nas figuras 44(a), 44(b) e 44(c) podemos ver como uma mesma cena pode ser vista sob diversos pontos de vista. É com base na posição do observador que as funções da biblioteca são capazes de determinar quais são os objetos que podem ser vistos, e desses qual a sua porção visível. O **OpenGL** disponibiliza uma série de funções com o objetivo de definir e manipular uma camera virtual, controlar a sua posição no espaço, tipo de projeção (figuras 44(d) e 44(e)), entre outras facilidades.

Outro fator importante na geração de imagens realistas é o tratamento das fontes de iluminação. A imagem de um objeto ganha muito mais realismo quando a interação das fontes de luz dispostas na cena e o objeto são levadas em conta. Podemos observar a diferença entre a figura 45, onde a interação fontes de luz / objetos não são consideradas, e a figura 46, onde essas interações são consideradas. O **OpenGL** suporta no mínimo 6 fontes de luz. É possível controlar parâmetros como : posição, cor, tipo, forma de interação com os materiais entre outras propriedades.

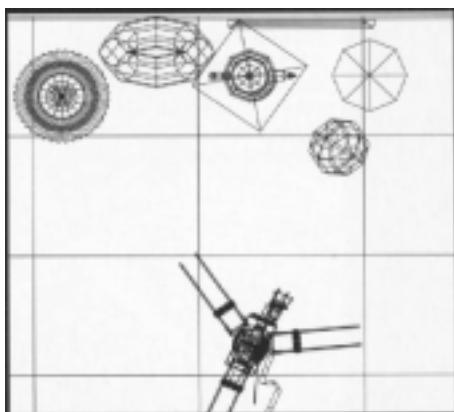
Na seqüência de figuras de 45 a 47 temos exemplos de como a iluminação pode influenciar as imagens gerada para uma mesma cena. As diferenças técnicas entre as imagens serão discutidas com mais detalhes no capítulo 4.



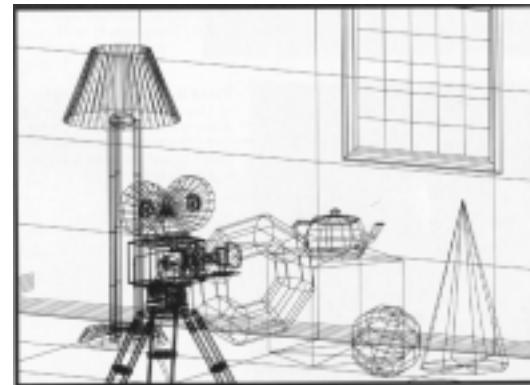
(a)



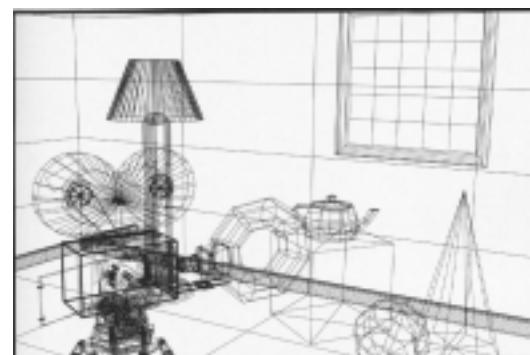
(b)



(c)



(d)



(e)



Figura 45 – Imagem de uma cena onde os objetos são pintados, sem que a interação com as fontes de luz sejam levadas em consideração. Repare que não há noção de volume nos objetos da imagem.



Figura 46 – Outro algoritmo é o de Phong, que dá aos objetos uma aparência metálica

Para gerarmos imagens realistas de objetos de material homogêneo (ouro, prata, cobre, plástico, etc) a simples interação objeto/fonte de luz é suficiente. No entanto, para objetos feitos de materiais heterogêneos, como madeira, granito, mármore, etc, essa solução não se mostra adequada. Nesses casos utilizamos, em conjunto com o modelo de iluminação, uma técnica denominada **mapeamento de textura**. A idéia é, de posse de uma imagem do material a ser simulado, "embrulhar" o objeto com essa imagem. Se esse embrulho for bem feito o objeto irá parecer feito desse material. O efeito dessa técnica pode ser visto na figura 47. O **OpenGL** possui funções para tratamento de textura. É possível controlar, por exemplo, a forma como o "embrulho" será feito e definir a imagem a ser utilizada.

Como sabemos ao final do pipeline de uma aplicação gráfica teremos a geração dos pixels que serão armazenados no *frame buffer*, e que irão formar a imagem final do modelo. O **OpenGL** trabalha com uma série de *buffers* para a geração e manipulação da imagem final. O *buffer* mais importante é o *color buffer*, uma cópia do frame buffer gerada em memória. Esse *buffer* armazena os pixels gerados pelo OpenGL e que serão posteriormente mapeados no *frame buffer*.

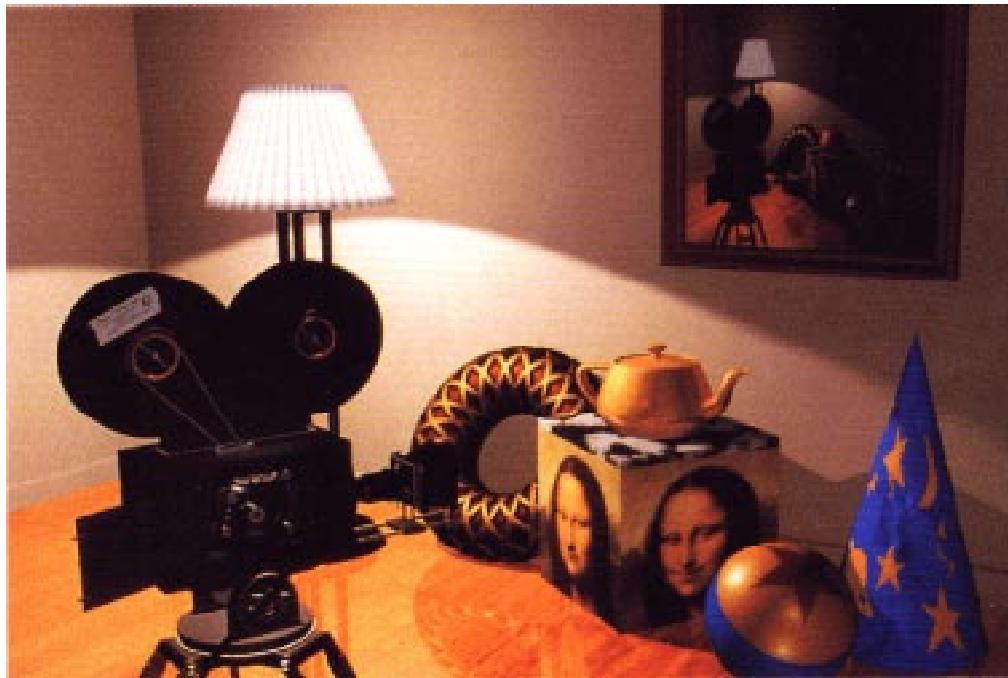


Figura 47 – Imagem gerada com efeitos avançados como sobras, reflexos e iluminação.

Outros *buffers* auxiliares são utilizados para o processamento de algumas rotinas ***OpenGL***:

*depth buffer* - utilizado pelo algoritmo *z-buffer* do ***OpenGL***.

*stencil buffer* - possibilita a criação de efeitos como sombra (figura 47 e 49).

*acummulation buffer* - permite a simulação de objetos em movimento ou fora de foco - figuras 48 e 49, respectivamente.



Figura 48 – Imagem de exemplo do efeito produzido por objetos em movimento.

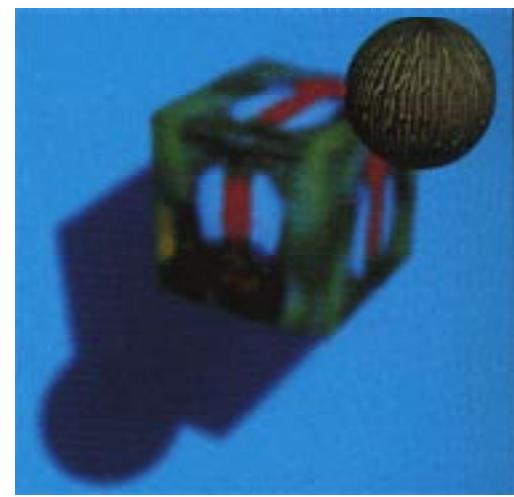
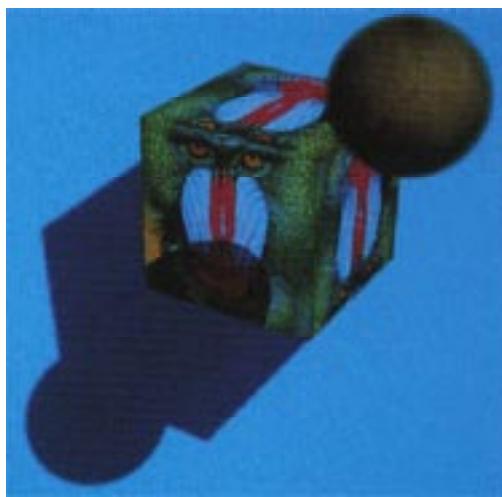


Figura 49 – Exemplos de imagens que utilizam os efeitos de sombra e foco.

Uma técnica bastante utilizada na geração de seqüências animadas, que se vale da existência de *buffers* intermediários para o armazenamento da imagem, é a de ***double buffer***. O objetivo dessa técnica é evitar que o usuário perceba a construção da imagem durante a apresentação da animação. Como o próprio nome sugere, dois *buffers* são utilizados : um "ligado" ao *frame buffer* e outro utilizado durante o processo de geração da nova imagem. Dessa forma quando a aplicação terminar a geração da imagem, uma troca (*swap*) entre os *buffers* é feita. O ***OpenGL*** suporta essa técnica.

## Implementação

A biblioteca ***OpenGL*** é implementada como um conjunto de funções que permitem a criação e manipulação de objetos. Seus estados também podem ser alterados por meio de funções específicas. No ambiente Windows todas as funções estão disponíveis a partir de uma DLL (***dynamic link library - biblioteca de linkagem dinâmica***) denominada *Opengl32.dll*. Outro conjunto de funções utilitárias (*glu*) acompanha o ***OpenGL*** : *glu32.dll*.

Todas as rotinas disponíveis, tanto no ***OpenGL*** quanto na biblioteca *glu*, seguem um padrão de nomeclatura : rotinas do ***OpenGL*** tem prefixo ***gl***, enquanto que rotinas da biblioteca *glu* tem prefixo ***glu***.

Muitas rotinas do ***OpenGL*** possuem variantes. Elas podem se diferir tanto no número de parâmetros (em geral em função das dimensões do espaço de trabalho - 2D, 3D ou 4D) quanto no seu tipo (ver tabela 1). Um exemplo dessa situação pode ser visto na função de criação de vértices. São ao todo 15 variantes. A sua funcionalidade é sempre a mesma, porém seus parâmetros dependem do espaço em que o vértice está definido - 2D, 3D ou 4D- e do tipo de dado utilizado para definir suas coordenadas -inteiros, ponto flutuante de precisão simples ou dupla, etc. Outra possibilidade é fornecer a função um vetor com as coordenadas de cada vértice do modelo.

Outras funções do ***OpenGL*** esperam encontrar uma constante, representada por um tipo enumerado (*GLenum*). Por padronização todas as constantes são escritas em maiúsculas e tem prefixo *GL*. Por exemplo podemos indicar o tipo de primitiva que iremos definir através de constantes como *GL\_POINTS*, *GL\_TRIANGLES* ou *GL\_QUADS* (figura 42).

Um ponto importante na implementação do ***OpenGL*** é a sua interface com o sistema de janelas do ambiente a ser utilizado. Nesse ponto não há uma padronização, já que cada ambiente irá trabalhar com uma arquitetura diferente. No ambiente *Windows* temos que em algum momento "conectar" a biblioteca a janela onde as imagens serão produzidas. Essa ligação deve ser feita explicitamente. Além disso precisamos criar uma estrutura denominada *render*

*context* (similar ao *device context* para uma janela bidimensional da API do *Windows*). As rotinas que fazem essa ligação são definidas dentro da API do *Windows (GDI)* e não na biblioteca ***OpenGL***.

Quando utilizamos um ambiente visual de programação, como o *Delphi*, *Visual Basic* ou *C++ Builder*, essa ligação pode ser encapsulada através de um componente - no nosso caso o ***WaiteGL***. Dessa forma a criação e tratamento do *render context* passa a ser transparente para a aplicação, processada por um método desse componente.

O componente ***WaiteGL*** tem por objetivo não só encapsular o procedimento de conexão ***OpenGL / Windows***, como também mapear algumas das rotinas das DLL's como métodos desse componente. Em nossos exemplos iremos nos restringir a utilização do componente para a primeira finalidade.

Ao incluirmos o componente ***WaiteGL*** no formulário uma janela - semelhante a um painel - é aberta. Ela responde a dois eventos (além daqueles comuns a qualquer componente visual) relacionados com o ***OpenGL***:

### ***OnSetupRC***

Como o próprio nome sugere o evento *OnSetupRC* é disparado quando da criação do componente e cumpre a finalidade de inicializar o *render context* e o próprio ambiente ***OpenGL***. Procedimentos típicos desse evento são a definição da cor de limpeza da tela, definição de camera, fontes de luz, entre outros.

### ***OnRender***

O evento *OnRender* será acionado sempre que for necessário o redesenho da janela ***WaiteGL***. Ele pode ser comparado ao evento *OnPaint* que alguns componentes visuais possuem. Nesse evento serão colocados todos os comandos necessários para o desenho do modelo.

A seguir apresentamos um exemplo de implementação que faz uso do componente ***WaiteGL*** e manipula algumas primitivas definidas no espaço bidimensional. A idéia desse exemplo é apresentar o formato típico de uma aplicação *Delphi/WaiteGL/OpenGL*, dando ênfase na interface entre esses elementos. Além disso, veremos o mecanismo de construção de objetos, com base nas primitivas da biblioteca, e como alterar parâmetros (variáveis de estado) que influenciam as primitivas.

## *OpenGL, Delphi e Windows*

Uma vez que o componente ***WaiteGL*** tenha sido instalado no ambiente *Delphi*, a sua utilização torna-se transparente. No entanto, para termos acesso

direto as funções armazenadas nas DLL's do ***OpenGL***, um protótipo das funções a serem utilizadas deve ser declarado. Uma UNIT (*opengl.pas*) com todos os protótipos das funções ***OpenGL*** é fornecido junto com a versão 3.0 do *Delphi*. Dessa forma basta que na clausula USES do nosso programa seja incluída a declaração dessa UNIT, tal como na listagem a seguir.

A biblioteca ***OpenGL*** necessita ter conhecimento de qual janela será utilizada para desenho, para poder fazer a associação com o *render context* da janela. Para isso dois métodos do componente ***WaiteGL*** são utilizados : *MakeCurrent* e *MakeNotCurrent*. Esses métodos são acionados, nesse caso, pelos eventos *OnActivate* e *OnClose*, respectivamente. Cabe ressaltar que estamos trabalhando com um caso particular, onde apenas uma janela será utilizada pelo ***OpenGL*** para geração de imagens. Em casos mais gerais, em que várias janelas podem ser usadas para desenho, teríamos que acionar esses métodos sempre que uma troca de janela acontecesse. Mais tarde no decorrer do curso veremos um exemplo dessa situação.

### *Primitivas*

Para o ***OpenGL*** qualquer objeto é definido pelo conjunto de seus vértices. Internamente eles serão estruturados de modo a formar primitivas como pontos, linhas, triângulos, quadriláteros e polígonos (figura 42). O ***OpenGL*** identifica como os vértices deverão ser interpretados através do comando *glBegin*. Seu único parâmetros corresponde a uma constante que indica que tipo de primitiva será sendo definida. Todo comando *glBegin* é finalizado por um comando *glEnd*. A definição dos vértices que compõe o modelo é feita através da função *glVertex*. Essa função, que deve aparecer sempre no interior de um par *glBegin/glEnd*, recebe como parâmetros as coordenadas do vértice.

Podemos observar o uso dessas funções nos 6 procedimentos responsáveis pelo desenho do modelo da aplicação. Cada um deles utiliza uma primitiva diferente.

### *Interface*

A aplicação possui uma interface bem simples (figura 50) : na parte superior da tela temos uma janela onde as imagens serão geradas, e na parte inferior temos uma série de componentes de interface que irão controlar alguns parâmetros do ***OpenGL***.

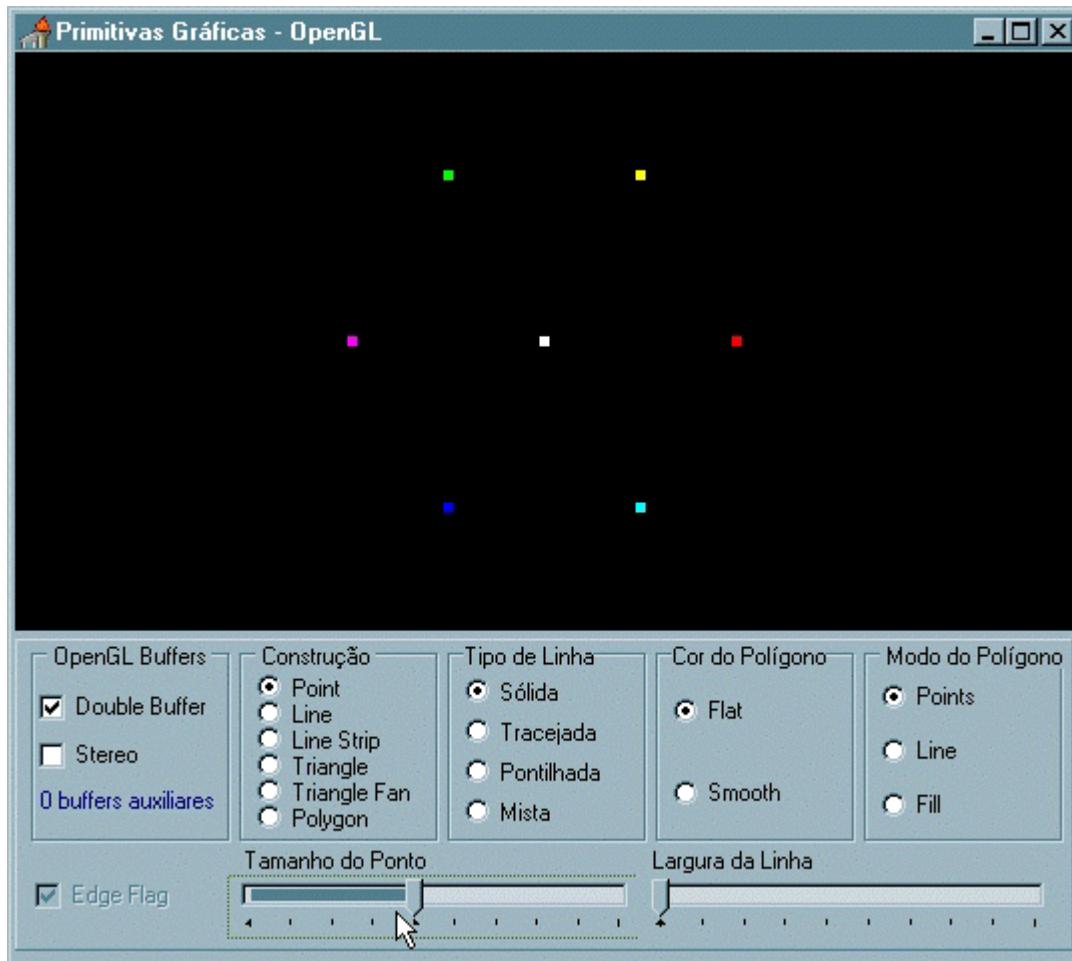


Figura 50 – Interface da aplicação exemplo. Em detalhe o controle do tamanho dos pontos.

Na janela ***OpenGL*** iremos desenhar um hexágono. Dependendo da opção selecionada na interface o hexágono será desenhado através de um tipo específico de primitiva.

O hexágono é inicialmente representado pela primitiva *POINTS*. Seus vértices são pré-calculados, com base nas dimensões da janela, no procedimento *DefinePontosHexagono*. A estrutura de dados que armazena esse modelo é a de uma matriz bidimensional. Os seus índices indicam, respectivamente, número do vértice e a sua coordenada (X ou Y). Uma segunda matriz é utilizada para o mapeamento da cor de cada vértice. De forma análoga o primeiro índice representa o número do vértice, enquanto que o segundo indica a componente de cor (R, G, B ou A).

É possível também controlar, via interface :

Tamanho dos pontos (vértices);

Largura das linhas (arestas);  
 Tipo de Linha (sólida, pontilhada, tracejada ou mista);  
 Modo de desenho do polígono (*points*, *lines* ou *fill*);  
 Forma de preenchimento do polígono (*flat* ou *smooth*).

Cada um desses parâmetros possui uma função ***OpenGL*** correspondente. Os três primeiros são autoexplicativos. Na figura 51 podemos observar o seu efeito prático.

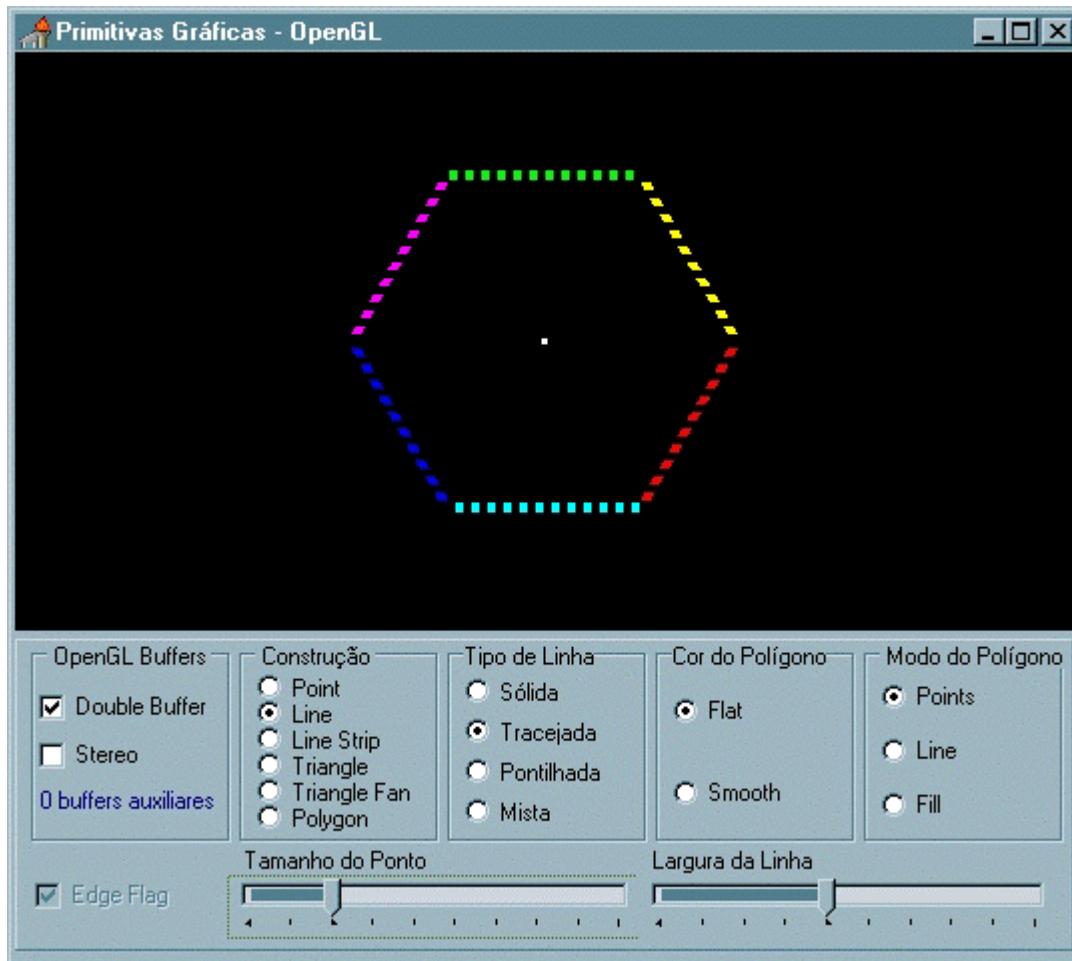


Figura 51 – Definição da largura da linha e tipo de linha.

Os dois últimos merecem comentários adicionais. *O modo de desenho do polígono* controla como ele será desenhado na tela - somente seus vértices, somente arestas ou preenchido. Esse controle só apresenta um resultado visual caso a primitiva utilizada, *POLYGON*, *QUADS* ou *TRIANGLE*.

Conforme já mencionamos, um polígono possui dois lados : o interno e o externo. O controle do modo de desenho dos polígonos pode ser feito para cada

um de seus lados. No caso do exemplo esse modo é aplicado para os dois lados de forma idêntica. Essa possibilidade é interessante para a visualização de objetos tridimensionais. Por exemplo, é possível definir que o lado interno dos polígonos (interior do objeto) será preenchido enquanto que o lado externo será representado apenas pelas arestas. Com isso podemos diferenciar o interior do exterior do objeto. Como estamos trabalhando, por enquanto, no espaço bidimensional essa diferenciação não se faz necessária.

Conjugado ao *modo de desenho do polígono* temos o tipo de *preenchimento do polígono*. Os tipos disponíveis são : *flat* e *smooth*. Quando selecionamos *flat* uma única cor é utilizada para o seu preenchimento - a cor de seu primeiro vértice - como na figura 52.

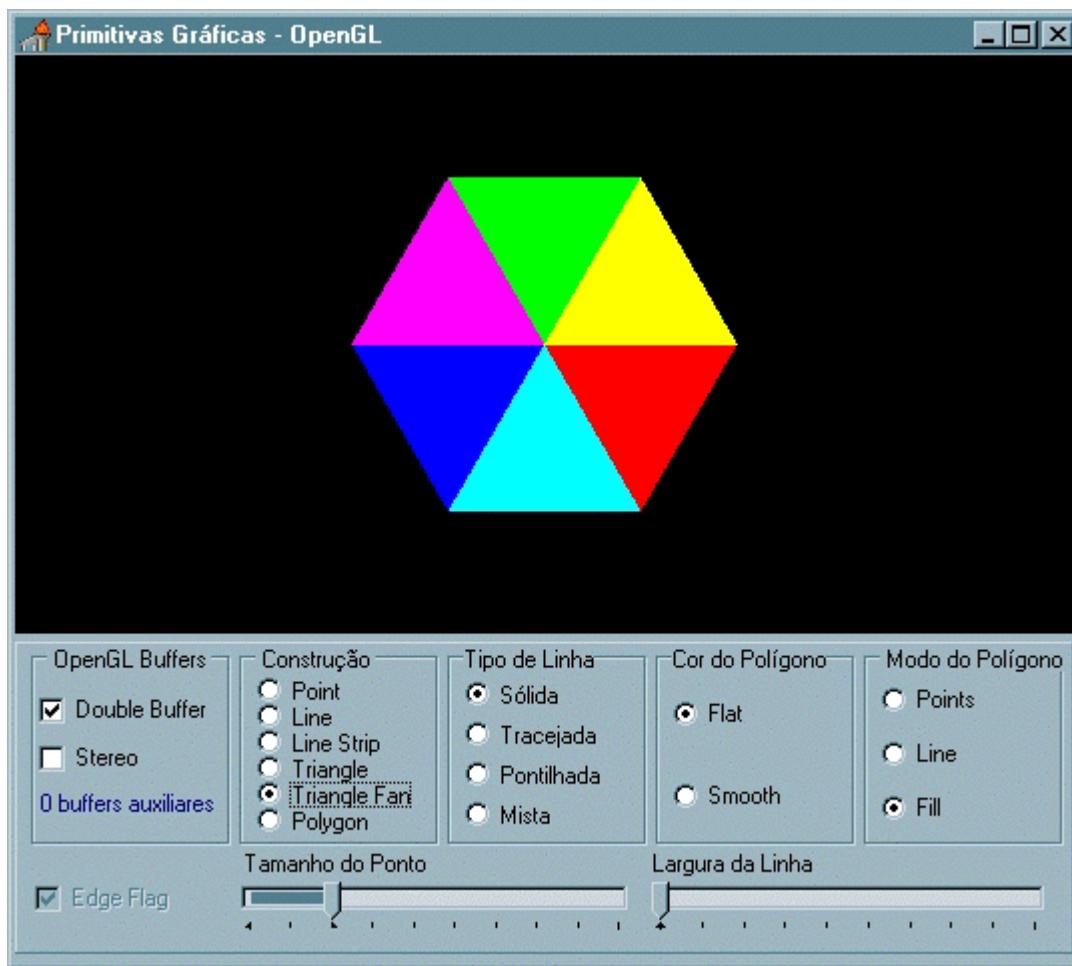


Figura 52 – Exemplo de primitivas em modo preenchido com preenchimento *flat*.

Outra possibilidade é utilizar o modo *smooth*. Nesse caso o polígono será preenchido com base nas cores de ***todos*** os seus vértices. Para garantir a

suavidade na passagem de uma cor para outra, é feita a interpolação das cores dos vértices. O resultado obtido é mostrado nas figuras 53 e 54.

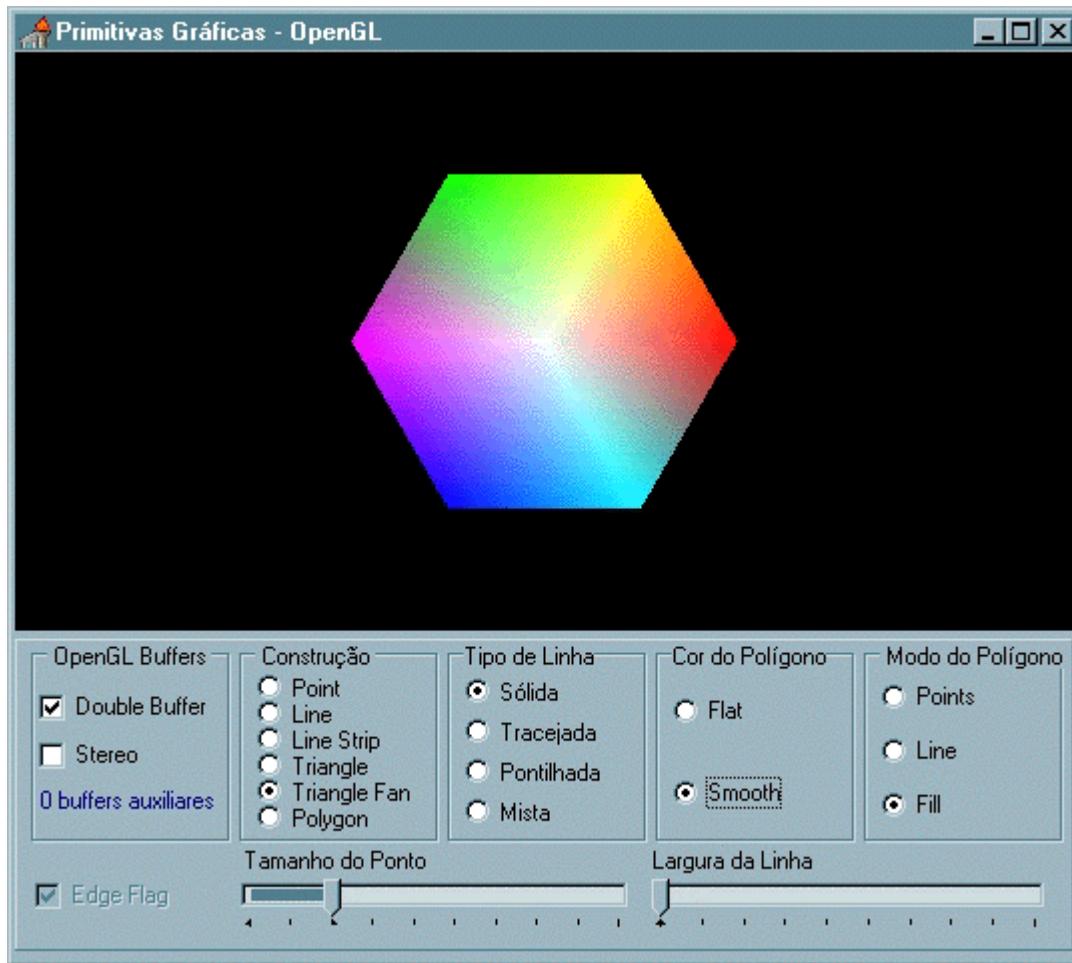


Figura 53 – Exemplo de primitiva (triangulos) com preenchimento do tipo *smooth*.

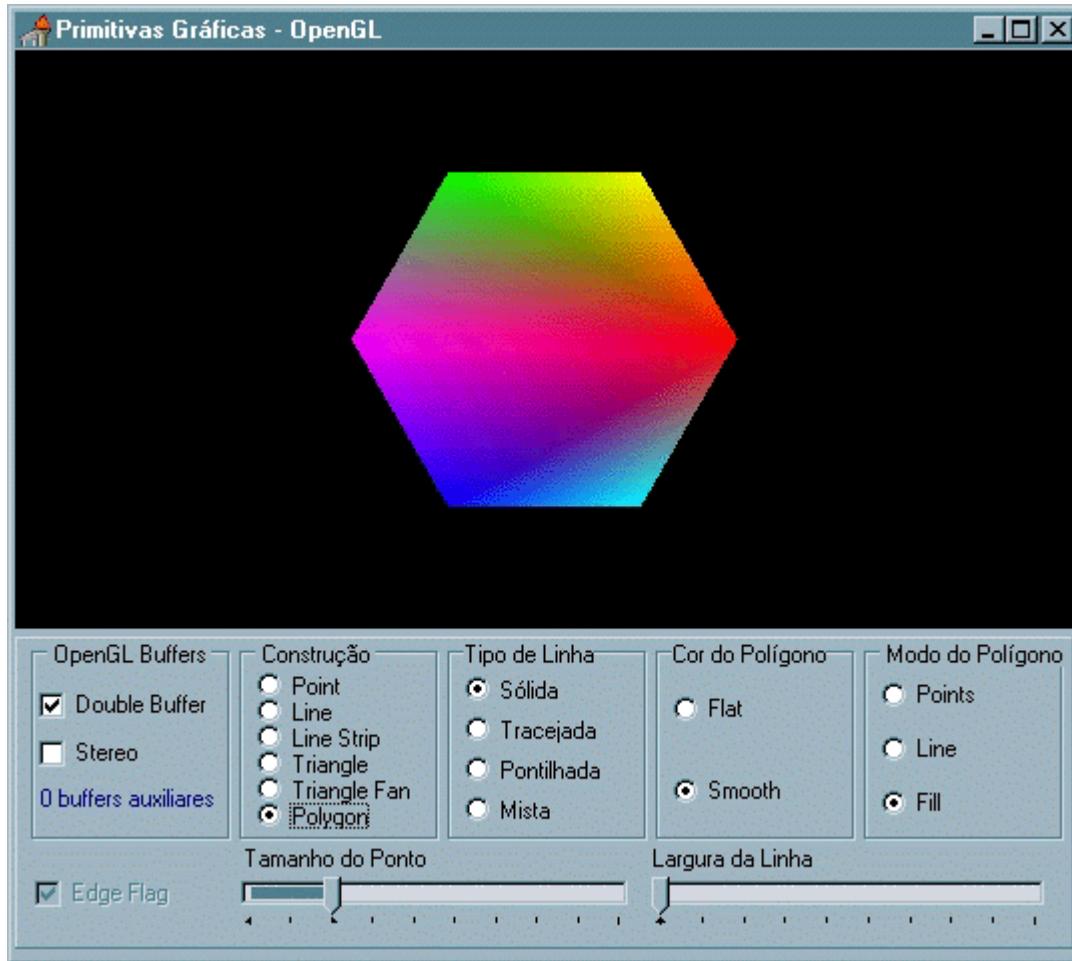


Figura 54 - Exemplo de primitiva (polígono) com preenchimento do tipo *smooth*.

Como exercício para o leitor fica a explicação do porquê da diferença de preenchimento do hexágono em função da primitiva utilizada para sua representação : triângulos (figura 53) e polígono (figura 54).

```

unit Primitivas;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OleCtrls, ComCtrls, StdCtrls, ExtCtrls, Menus,
  // UNIT incluida automaticamente pelo componente WaiteGL.
  WAITEGL,
  // UNIT onde estão declarados os protótipos para as funções OpenGL das dll's
  // opengl32.dll e glu32.dll
  Opengl;

type
  TFrmPrimitivas = class(TForm)
    Panel1: TPanel;
    RGTipoHexagono: TRadioButton;
    CBEedgeFlag: TCheckBox;
    GLWindow: TWaiteGL;
    RGPolygonMode: TRadioButton;
    TBPointSize: TTrackBar;
    Label1: TLabel;
    Label2: TLabel;
    TBLLineWidth: TTrackBar;
    RGLineStipple: TRadioButton;
    RGShadeModel: TRadioButton;
    GroupBox1: TGroupBox;
    CBDoubleBuffer: TCheckBox;
    CBStereo: TCheckBox;
    LblNumBuffers: TLabel;
    procedure GLWindowRender(Sender: TObject);
    procedure GLWindowSetupRC(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormResize(Sender: TObject);
    procedure RGTipoHexagonoClick(Sender: TObject);
    procedure CBEedgeFlagClick(Sender: TObject);
    procedure Fim1Click(Sender: TObject);
    procedure RGPolygonModeClick(Sender: TObject);
    procedure TBPointSizeChange(Sender: TObject);
    procedure TBLLineWidthChange(Sender: TObject);
    procedure RGLineStippleClick(Sender: TObject);
    procedure RGShadeModelClick(Sender: TObject);
    procedure CBDoubleBufferClick(Sender: TObject);
    procedure CBStereoClick(Sender: TObject);
  end;

  private
    { Private declarations }

    Procedure DefinePontosHexagono;

  public
    Procedure DesenhaHexagonoPoints;
    Procedure DesenhaHexagonoLine;
    Procedure DesenhaHexagonoLineStrip;
    Procedure DesenhaHexagonoTriangles;
    Procedure DesenhaHexagonoTriangleFan;
    Procedure DesenhaHexagonoPolygon;
    { Public declarations }
  end;

  // *****
  // * Declaração de dois tipos enumerados para facilitar a *
  // * leitura dos vetores de cores e vértices do hexágono. *
  // *****

  Coordenadas = (X, Y);
  Cores      = (R, G, B, A);
  VetCores   = array [0..6, R..A] of GLfloat;

var
  FrmPrimitivas : TFrmPrimitivas;
  Hexagono     : array [0..6, X..Y] of GLfloat;
  ComStereo,
  ComDoubleBuffer : GLBoolean;
  NumAuxBuffers : GLint;
  const
    Cor : VetCores = ((1.0, 1.0, 1.0, 1.0),
                      (1.0, 0, 0, 1.0),
                      (1.0, 1.0, 0, 1.0),
                      (0, 1.0, 0, 1.0),
                      (1.0, 0, 1.0, 1.0),
                      (0, 0, 1.0, 1.0),
                      (0, 1.0, 1.0, 1.0));
  implementation
  {$R *.DFM}

```

```

// *****
// * Método onde os pontos do hexagon são definidos (com base *
// * nas dimensões da tela) e armazenados no vetor HEXAGONO *
// *****

Procedure TFrmPrimitivas.DefinePontosHexagono;
var i    : integer;
    angulo,
    raio  : real;
Begin
  Hexagono[0,X] := GLWindow.Width / 2;
  Hexagono[0,Y] := GLWindow.Height / 2;
  angulo := 0.0;
  if GLWindow.Width > GLWindow.Height then
    raio := GLWindow.Height / 3
  else
    raio := GLWindow.Width / 3;
  for i:=1 to 6 do
    Begin
      Hexagono[i,X] := Hexagono[0,X] + raio * cos(angulo);
      Hexagono[i,Y] := Hexagono[0,Y] + raio * sin(angulo);
      angulo := angulo + PI / 3;
    end;
End;

// *****
// * Método responsável pelo desenho do hexagono utilizando      *
// * a primitiva POINTS. Só os vértices do hexagono serão desenhados *
// *****

Procedure TFrmPrimitivas.DesenhaHexagonoPoints;
var i : integer;
Begin
// Define o inicio da declaração dos vértices da primitiva POINTS
  glBegin(GL_POINTS);
  for i:=0 to 6 do
    Begin
      glColor4f(Cor[i,R], Cor[i,G], Cor[i,B], Cor[i,A]);
      glVertex2f(Hexagono[i,X], Hexagono[i,Y]);
    end;
  glEnd;
End;

// Finaliza a declaração da primitiva
glEnd;
End;

// *****
// * Método responsável pelo desenho do hexagono utilizando      *
// * a primitiva LINES. As arestas do hexagono serão desenhados   *
// *****

Procedure TFrmPrimitivas.DesenhaHexagonoLine;
var i : integer;
Begin
// Desenha um ponto no centro do hexagono
  glBegin(GL_POINTS);
  glColor4f(Cor[0,R], Cor[0,G], Cor[0,B], Cor[0,A]);
  glVertex2f(Hexagono[0,X], Hexagono[0,Y]);
  glEnd;
// Inicio da declaração dos vértices da primitiva LINES
  glBegin(GL_LINES);
  for i:=1 to 5 do
    Begin
      glColor4f(Cor[i,R], Cor[i,G], Cor[i,B], Cor[i,A]);
      glVertex2f(Hexagono[i,X], Hexagono[i,Y]);
      glColor4f(Cor[i+1,R], Cor[i+1,G], Cor[i+1,B], Cor[i+1,A]);
      glVertex2f(Hexagono[i+1,X], Hexagono[i+1,Y]);
    end;
// Fecha o hexagono ligando o primeiro ao último ponto
  glColor4f(Cor[6,R], Cor[6,G], Cor[6,B], Cor[6,A]);
  glVertex2f(Hexagono[6,X], Hexagono[6,Y]);
  glColor4f(Cor[1,R], Cor[1,G], Cor[1,B], Cor[1,A]);
  glVertex2f(Hexagono[1,X], Hexagono[1,Y]);
  glEnd;
End;

// *****
// * Método responsável pelo desenho do hexagono utilizando      *
// * a primitiva LINESTRIP. As arestas do hexagono serão desenhadas. *
// *****

Procedure TFrmPrimitivas.DesenhaHexagonoLineStrip;
var i : integer;
Begin
// Desenha um ponto no centro do hexagono

```

```

glBegin(GL_POINTS);
  glColor4f(Cor[0,R], Cor[0,G], Cor[0,B], Cor[0,A]);
  glVertex2f(Hexagono[0,X], Hexagono[0,Y]);
glEnd;
// Inicio da declaração dos vértices da primitiva LINESTRIP
glBegin(GL_LINE_STRIP);
  for i:=1 to 6 do
    Begin
      glColor4f(Cor[i,R], Cor[i,G], Cor[i,B], Cor[i,A]);
      glVertex2f(Hexagono[i,X], Hexagono[i,Y]);
    end;
// Fecha o hexagono ligando o primeiro ao último ponto
  glColor4f(Cor[1,R], Cor[1,G], Cor[1,B], Cor[1,A]);
  glVertex2f(Hexagono[1,X], Hexagono[1,Y]);
glEnd;
End;

// *****
// * Método responsável pelo desenho do hexagono utilizando      *
// * a primitiva TRIANGLES. O hexagono será desenhado como um      *
// * conjunto de 6 triangulos com um vértice comum no centro do   *
// * hexagono.                                                 *
// *****

Procedure TFrmPrimitivas.DesenhaHexagonoTriangles;
var i : integer;
  ShowEdges : Boolean;
Begin
// Como base no estado da interface (CBEEdgeFlag) define se as arestas
// dos triangulos que são internas ao hexagono serão mostradas ou não.
  ShowEdges := TRUE;
  if CBEEdgeFlag.State = cbUnChecked then
    ShowEdges := FALSE;
// Inicio da declaração dos vértices da primitiva TRIANGLES
  glBegin(GL_TRIANGLES);
    for i:=1 to 5 do
      Begin
// Liga ou desliga a aresta dependendo da variavel ShowEdges
        glEdgeFlag(ShowEdges);
        glColor4f(Cor[0,R], Cor[0,G], Cor[0,B], Cor[0,A]);
        glVertex2f(Hexagono[0,X], Hexagono[0,Y]);
// Força que a aresta seja desenhada - aresta do hexagono
        glEdgeFlag(TRUE);
        glColor4f(Cor[i,R], Cor[i,G], Cor[i,B], Cor[i,A]);
        glVertex2f(Hexagono[i,X], Hexagono[i,Y]);
      end;
    End;
    // Liga ou desliga a aresta dependendo da variavel ShowEdges
    glEdgeFlag(ShowEdges);
    glColor4f(Cor[i+1,R], Cor[i+1,G], Cor[i+1,B], Cor[i+1,A]);
    glVertex2f(Hexagono[i+1,X], Hexagono[i+1,Y]);
  end;
// Desenha o último triangulo
  glEdgeFlag(ShowEdges);
  glColor4f(Cor[0,R], Cor[0,G], Cor[0,B], Cor[0,A]);
  glVertex2f(Hexagono[0,X], Hexagono[0,Y]);
  glEdgeFlag(TRUE);
  glColor4f(Cor[6,R], Cor[6,G], Cor[6,B], Cor[6,A]);
  glVertex2f(Hexagono[6,X], Hexagono[6,Y]);
  glEdgeFlag(ShowEdges);
  glColor4f(Cor[1,R], Cor[1,G], Cor[1,B], Cor[1,A]);
  glVertex2f(Hexagono[1,X], Hexagono[1,Y]);
  glEnd;
End;

// *****
// * Método responsável pelo desenho do hexagono utilizando      *
// * a primitiva TRIANGLEFAN. O hexagono será desenhado como um      *
// * conjunto de 6 triangulos com um vértice comum no centro do   *
// * hexagono e ligados por arestas comuns.                         *
// *****

Procedure TFrmPrimitivas.DesenhaHexagonoTriangleFan;
var i : integer;
Begin
// Inicio da declaração dos vértices da primitiva TRIANGLEFAN
  glBegin(GL_TRIANGLE_FAN);
    glColor4f(Cor[0,R], Cor[0,G], Cor[0,B], Cor[0,A]);
    glVertex2f(Hexagono[0,X], Hexagono[0,Y]);
    for i:=0 to 6 do
      Begin
        glColor4f(Cor[i,R], Cor[i,G], Cor[i,B], Cor[i,A]);
        glVertex2f(Hexagono[i,X], Hexagono[i,Y]);
      end;
    // Fecha o último triangulo
    glColor4f(Cor[1,R], Cor[1,G], Cor[1,B], Cor[1,A]);
    glVertex2f(Hexagono[1,X], Hexagono[1,Y]);
    glEnd();
  End;
// *****
// * Método responsável pelo desenho do hexagono utilizando      *

```

```

/* * a primitiva POLYGON. O hexágono será desenhado como um polígono.*
// ****
Procedure TfrmPrimitivas.DesenhaHexagonoPolygon;
  var i : integer;
Begin
// Início da declaração dos vértices da primitiva POLYGON
  glBegin(GL_POLYGON);
    for i:=1 to 6 do
      Begin
        glColor4f(Cor[i,R], Cor[i,G], Cor[i,B], Cor[i,A]);
        glVertex2f(Hexagono[i,X], Hexagono[i,Y]);
      end;
    glEnd;
End;

// ****
// * Método responsável pela inicialização do ambiente OpenGL - *
// * via componente WaiteGL. Esse método só é executado na criação *
// * do componente WaiteGL. *
// ****

procedure TfrmPrimitivas.GLWindowSetupRC(Sender: TObject);
Begin
// Define a cor de limpeza da tela
  glClearColor(0.0, 0.0, 0.0, 1.0);
// Habilita o tracado de linhas com "padrão"
  glEnable(GL_LINE_STIPPLE);
// Define o padrão das linhas - no caso linhas sólidas
  glLineStipple(1, $FFFF);
// Define as dimensões da janela de visualização,
// em função do tamanho da janela WaiteGL
  glViewport(0, 0, width, height);
// Define qual a pilha de matrizes de trabalho
  glMatrixMode(GL_MODELVIEW);
// inicializa a matriz corrente
  glLoadIdentity();
// Define o volume de visão, para uma projeção paralela
  glOrtho(0.0, width, 0.0, height, 1.0, -1.0);
// Define como os polígonos serão desenhados na tela
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
// Define a forma de preenchimento dos polígonos
  glShadeModel(GL_FLAT);
// Verifica se o ambiente OpenGL da instalação suporta Double Buffer
  glGetBooleanv(GL_DOUBLEBUFFER, @ComDoubleBuffer);

```

```

CBDoubleBuffer.Checked := ComDoubleBuffer;
// Verifica se o ambiente OpenGL da instalação suporta Stereo
  glGetBooleanv(GL_STEREO, @ComStereo);
  CBStereo.Checked := ComStereo;
// Verifica quantos buffers auxiliares o ambiente OpenGL da instalação suporta
  glGetIntegerv(GL_AUX_BUFFERS, @NumAuxBuffers);
  LblNumBuffers.caption := IntToStr(NumAuxBuffers) + ' buffers auxiliares';
// Força a execução imediata dos comandos pendentes na fila
  glFlush();
End;

// ****
// * Método responsável pela redesenho da janela WaiteGL. *
// * Esse método é acionado toda vez que a janela precisar ser *
// * redesenhada. Pode ser gerado automaticamente. *
// ****

procedure TfrmPrimitivas.GLWindowRender(Sender: TObject);
begin
// Limpa a janela OpenGL - color buffer
  glClear(GL_COLOR_BUFFER_BIT);
// Em função do item selecionado na interface
// chama o procedimento de desenho correspondente
  case RGTipoHexagono.ItemIndex of
    0 : DesenhaHexagonoPoints;
    1 : DesenhaHexagonoLine;
    2 : DesenhaHexagonoLineStrip;
    3 : Begin
      CBEEdgeFlag.Enabled := TRUE;
      DesenhaHexagonoTriangles;
    end;
    4 : DesenhaHexagonoTriangleFan;
    5 : DesenhaHexagonoPolygon;
  end;
  glFlush();
// Promove a troca dos color buffer
  GLWindow.SwapBuffers();
end;

// ****
// * Método acionado na ativação do formulário. *
// ****

```

```

procedure TfrmPrimitivas.FormActivate(Sender: TObject);
begin
  // Promove a ligação entre a biblioteca OpenGL e a
  // janela WaiteGL denominada GLWindow
  GLWindow.MakeCurrent();
  // Monta a matriz de coordenadas do hexágono
  DefinePontosHexagono;
end;

// *****
// * Método acionado na desativação do formulário.
// *****

procedure TfrmPrimitivas.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  // Desfaz a ligação entre a biblioteca OpenGL e a
  // janela WaiteGL denominada GLWindow
  GLWindow.MakeNotCurrent();
end;

// *****
// * Método acionado na mudança de tamanho do formulário.
// *****

procedure TfrmPrimitivas.FormResize(Sender: TObject);
begin
  // Redefine a janela de visualização do OpenGL
  glViewport(0, 0, width, height);
  // Reinicializa a matriz corrente
  glLoadIdentity();
  // Redefine o volume de visão, para uma projeção paralela
  glOrtho(0.0, width, 0.0, height, 1.0, -1.0);
  glFlush();
end;

// *****
// * Os Métodos a seguir são acionados pela alteração de estado da *
// * da interface. *
// *****

procedure TfrmPrimitivas.RGTipoHexagonoClick(Sender: TObject);
begin
  CBEdgeFlag.Enabled := FALSE;
  // Força o redesenho da janela WaiteGL -
end;

// *****
// * Ação do evento OnRender.
// *****

procedure TfrmPrimitivas.OnRender(Sender: TObject);
begin
  // Redesenho da janela WaiteGL
  GLWindow.Invalidate();
end;

procedure TfrmPrimitivas.CBEdgeFlagClick(Sender: TObject);
begin
  GLWindow.Invalidate();
end;

procedure TfrmPrimitivas.Fim1Click(Sender: TObject);
begin
  FrmPrimitivas.Close();
end;

procedure TfrmPrimitivas.RGPolygonModeClick(Sender: TObject);
begin
  // Com base no item selecionado no RadioGroup define
  // o modo como os polígonos serão desenhados
  case RGPolygonMode.ItemIndex of
    0 : glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
    1 : glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    2 : glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  end;
  GLWindow.Invalidate();
end;

procedure TfrmPrimitivas.TBPointSizeChange(Sender: TObject);
begin
  // Redefine o tamanho dos pontos
  glPointSize(TBPointSize.Position);
  TBPointSize.SelEnd := TBPointSize.Position;
  GLWindow.Invalidate();
end;

procedure TfrmPrimitivas.TBLineWidthChange(Sender: TObject);
begin
  // Redefine a largura das linhas de desenho
  glLineWidth(TBLineWidth.Position);
  TBLineWidth.SelEnd := TBLineWidth.Position;
  GLWindow.Invalidate();
end;

```

```
procedure TfrmPrimitivas.RGLineStippleClick(Sender: TObject);
begin
  // Define o padrão de desenho das linhas
  case RGLineStipple.ItemIndex of
    // Sólida
    0 : glLineStipple(1, $FFFF); // 0x11111111
    // Tracejada
    1 : glLineStipple(1, $F0F0); // 0x11110000
    // Pontilhada
    2 : glLineStipple(1, $AAAA); // 0x10101010
    // Mista - traços e pontos
    3 : glLineStipple(1, $B8B8); // 0x10111000
  end;
  GLWindow.Invalidate;
end;

// ****
procedure TfrmPrimitivas.RGShadeModelClick(Sender: TObject);
begin
  // Define o tipo de preenchimento dos polígonos
  case RGShadeModel.ItemIndex of
    // Preenchimento contínuo
    0 : glShadeModel(GL_FLAT);
    // Preenchimento por interpolação das cores dos vértices
    1 : glShadeModel(GL_SMOOTH);
  end;
  GLWindow.Invalidate;
end;

// ****
procedure TfrmPrimitivas.CBDoubleBufferClick(Sender: TObject);
begin
  CBDoubleBuffer.Checked := ComDoubleBuffer;
end;

// ****
procedure TfrmPrimitivas.CBStereoClick(Sender: TObject);
begin
  CBStereo.Checked := ComStereo;
end;

end.
```

## Referência

**wglGetCurrent (void);**

Método WaiteGL :

**MakeCurrent () ;**

Descrição :

***Rotina disponibilizada pela Graphics Device Interface (GDI) do Windows.***

***Torna o render context (rc) do componente WaiteGL como o rc corrente para a thread chamadora.***

**wglMakeNotCurrent (void);**

Método WaiteGL :

**MakeNotCurrent () ;**

Descrição :

***Rotina disponibilizada pela Graphics Device Interface (GDI) do Windows.***

***Libera o render context (rc) do componente WaiteGL alocado para a thread chamadora.***

***glClearColor (GLclampf R, GLclampf G, GLclampf B, GLclampf A);***

Método WaiteGL :

***ClearColor (float R, float G, float B, float A);***

Valor Default :

***R = G = B = A = 0***

Descrição :

***Define a cor utilizada para limpeza do ColorBuffer. Os valores RGB são definidos no intervalo [0..1].***

***glClear (GLbitfield mascara);***

Método WaiteGL :

***Clear (long mascara);***

Valores :

<b><i>OpenGL</i></b>	<b><i>WaiteGL</i></b>
GL_COLOR_BUFFER_BIT	glColorBuffer();
GL_DEPTH_BUFFER_BIT	glDepthBuffer();
GL_ACCUM_BUFFER_BIT	glAccumBuffer();
GL_STENCIL_BUFFER_BIT	glStencilBuffer();

Descrição :

***Define qual dos 4 buffers será limpo.***

***Por se tratar de um campo binário, a função OpenGL permite que mais de um buffer seja limpo com apenas um comando glClear.***

***glFlush (void);***

Método WaiteGL :

***Flush () ;***

Descrição :

***Força a o início da execução dos comandos previamente emitidos.***

***glVertex {234} {idsf} {v} (Type coords);***

Método WaiteGL :

***Vertex {234} f (float coords);***

Descrição :

***Especifica um vértice utilizado na descrição de um objeto. O número de parâmetros coords pode variar entre 2 e 4 conforme a função, assim como o tipo (no caso do OpenGL) pode variar entre I, d, s, ou f. Chamadas a função glVertex devem ser feita entre um par de funções glBegin e glEnd.***

***glBegin (GLenum modo);***

Método WaiteGL :

***Begin\_***  
***Start***           *(long Modo);*  
***GLBegin***

Valores :

<b><i>OpenGL</i></b>	<b><i>WaiteGL</i></b>
GL_POINTS	glPoints();
GL_LINES	glLines();
GL_POLYGON	glPolygon();
GL_TRIANGLES	glTriangles();
GL_QUADS	glQuads();
GL_LINE_STRIP	glLineStrip();
GL_LINE_LOOP	glLineLoop();
GL_TRIANGLE_STRIP	glTriangleStrip();
GL_TRIANGLE_FAN	glTriangleFan();
GL_QUAD_STRIP	glQuadStrip();

Descrição :

***Marca o início de uma lista de vértices que irá descrever uma primitiva. O tipo da primitiva é definida pelo parâmetro modo.***

**glEnd (void);**

Método WaiteGL :

**End\_**  
**Stop**        0;  
**GLEnd**

Descrição :

**Finaliza a lista de vértices iniciada pela função glBegin.**

**glPointSize (GLfloat tamanho);**

Método WaiteGL :

**PointSize (float tamanho);**

Valor Default :

**tamanho = 1.0.**

Descrição :

**Define o tamanho, em pixels que cada ponto. O valor de tamanho deve ser maior que 0.0.**

**glLineWidth (GLfloat largura);**

Método WaiteGL :

**LineWidth (float largura);**

Valor Default :

**largura = 1.0.**

Descrição :

**Define a largura, em pixels, das linhas. O valor de largura deve ser maior que 0.0.**

***glEdgeFlag (Glboolean flag);***

Método WaiteGL :

***EdgeFlag (Bool flag);***

Valor Default :

***edge flag = GL\_TRUE.***

Descrição :

***Indica se um vértice deve ser considerado como início de uma aresta da fronteira de um polígono. Essa função altera a variável de estado edge flag. Arestras de fronteira de um polígono são as únicas arestras desenhadas.***

***glLineStipple (GLint fator, GLushort padrão);***

Método WaiteGL :

***LineStipple (long fator, short padrão);***

Valor Default :

***fator = 1.0.***

***padrão =  $2^{16} - 1$ .***

Descrição :

***Define uma padrão de traçado para as linhas. Por exemplo :***

***LineStipple ( 2, 21845); define um padrão de linha pontilhada, onde cada grupo de 2 pixels é aceso ou apagado alternadamente. (21845 = 0x5555 = 01010101010101).***

***Essa opção deve ser habilitada por meio da função : glEnable(GL\_LINE\_STTIPLE).***

***glPolygonMode (GLenum face, GLenum modo);***

Método WaiteGL :

***PolygonMode (long face, long modo);***

Valores :

***face***

<b><i>OpenGL</i></b>	<b><i>WaiteGL</i></b>
GL_FRONT	glFront();
GL_BACK	glBack();
GL_FRONT_AND_BACK	glFrontAndBack();

***Mode***

<b><i>OpenGL</i></b>	<b><i>WaiteGL</i></b>
GL_POINT	glPoint();
GL_LINE	glLine();
GL_FILL	glFill();

Valor Default :

***GL\_FRONT\_AND\_BACK, GL\_FILL***

Descrição :

***Controla o modo como a parte frontal e dorsal de cada polígono será desenhada.***

***glViewport (GLint x, GLint y, GLsizei largura, GLsizei altura);***

Método WaiteGL :

***Viewport (long x, long y, long largura, long altura);***

Valor Default :

***x = y = 0;***  
***largura = largura da janela;***  
***altura = altura da janela.***

Descrição :

***Define a janela de visualização onde a imagem final vai ser apresentada.***

***glMatrixMode (GLenum modo);***

Método WaiteGL :

***MatrixMode (long modo);***

Valores :

<b><i>OpenGL</i></b>	<b><i>WaiteGL</i></b>
GL_MODEL_VIEW	glModelView();
GL_PROJECTION	glProjection();
GL_TEXTURE	glTexture();

Valor Default :

***modo = GL\_MODEL\_VIEW.***

Descrição :

***Define qual matriz corrente será modificada a partir desse momento.***

***glLoadIdentity (void);***

Método WaiteGL :

***LoadIdentity () ;***

Descrição :

***Inicializa a matriz corrente com a matriz identidade.***

***GluOrtho2DViewport (GLdouble esquerda, GLdouble direita,  
GLdouble fundo, GLdouble topo);***

Método WaiteGL :

***gluOrtho2D (float esquerda, float direita, float fundo,  
float topo);***

Descrição :

***Define uma matriz para projeção de coordenadas bidimensionais na tela. A área de recorte é definida pela janela retangular com canto inferior esquerdo (esquerda, fundo) e canto superior direito (esquerda, topo).***

***SwapBuffers (void);***

Método WaiteGL :

***SwapBuffers () ;***

Descrição :

***Rotina disponibilizada pela Graphics Device Interface (GDI) do Windows.***

***Promove a troca entre o back buffer e o front buffer, caso a instalação suporte double buffer.***

# CAPÍTULO III

## TRANSFORMAÇÕES GEOMÉTRICAS

---

A representação de um objeto tridimensional, através de uma malha poligonal, define implicitamente a sua forma e posicionamento iniciais. No entanto, é muito comum querermos alterar a forma e/ou posição de um objeto dentro da aplicação. Dessa forma, o mesmo modelo pode ser utilizado mais de uma vez, só que em posições e formas distintas daquela definida por sua malha. Uma aplicação gráfica deve fornecer meios de, uma vez criada a malha poligonal que representa o objeto, poder modificar sua forma e/ou posição. O mecanismo utilizado para tal finalidade são as **transformações geométricas**.

Nesse capítulo estaremos interessados em abordar os aspectos práticos da utilização de transformações dentro da área de Computação Gráfica. Dentro dessa abordagem o formalismo matemático inerente ao assunto será pouco explorado. Apresentaremos no final do capítulo um exemplo de implementação dos conceitos abordados utilizando a biblioteca **OpenGL**.

### Definições

Transformações geométricas são funções matemáticas que alteram pontos no espaço. Do ponto de vista matemático, uma função pode ser vista como um mapeamento entre dois conjuntos : **domínio** e **imagem** da função. As transformações geométricas que iremos estudar se caracterizam por terem domínio e imagem definidos no espaço Euclidiano<sup>2</sup>.

Um ponto qualquer **P** no espaço Euclidiano de dimensão 2 é descrito pela expressão :

$$P = \alpha.u + \beta.v + P_o$$

onde **P<sub>o</sub>** representa a origem do espaço, **u** e **v** os vetores ortonormais. O ponto **P**, portanto, pode ser caracterizado pelos coeficientes  $\alpha$  e  $\beta$  da expressão. Esses coeficientes são ditos **coordenadas** do ponto nesse espaço. A representação desse ponto será dada por um vetor :

---

<sup>2</sup> O espaço Euclidiano é caracterizado por um ponto (origem do espaço) e um conjunto de vetores ortonormais, ou seja, ortogonais entre si e de norma (tamanho) 1.

$$P = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad \text{ou} \quad P = [\alpha \ \beta]$$

Por convenção iremos utilizar a segunda notação.

## Transformações Bidimensionais

Uma transformação geométrica bidimensional  $T$ , que leva um ponto  $P$ , de coordenadas  $(x,y)$ , ao ponto  $P'$ , de coordenadas  $(x',y')$ , pode ser escrita na forma de uma matriz quadrada de dimensão igual a do espaço de trabalho – nesse caso 2. A forma dessa transformação é dada por :

$$P' = P \cdot T_l \quad \text{onde} \quad T_l = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a.x + c.y & b.x + d.y \end{bmatrix}$$

Uma característica importante desse tipo de transformação é que quando aplicada a um ponto localizado na origem do espaço, essa ponto não se altera. Tais transformações são ditas ***lineares***.

Portanto, a matriz  $T_l$  caracteriza uma transformação linear através de 4 coeficientes. De acordo com os seus valores, diferentes resultados geométricos podem ser obtidos. A seguir analisaremos dois tipos de transformações bidimensionais básicas : *Escala* e *Rotação*.

### *Escala*

Uma transformação de escala deve ser capaz de alterar as coordenadas de um ponto por um fator  $E_x$  ou  $E_y$ , ditos fatores de escala da transformação na direção  $x$  e  $y$ , respectivamente. Se o fator de escala for maior que 1 estamos diante de uma transformação de expansão. Caso esse fator seja definido no intervalo  $]0,1[$  temos uma transformação de compressão. A forma da matriz correspondente a essa transformação é dada por :

$$P \cdot E = P'$$

$$\begin{bmatrix} x & y \end{bmatrix} \cdot E = \begin{bmatrix} E_x \cdot x & E_y \cdot x \end{bmatrix}$$

$$E = \begin{bmatrix} E_x & 0 \\ 0 & E_y \end{bmatrix}$$

O efeito geométrico da aplicação de uma escala pode ser visto na figura 55. Uma contração é aplicada ao objeto, com fatores de escala distintos em cada direção.

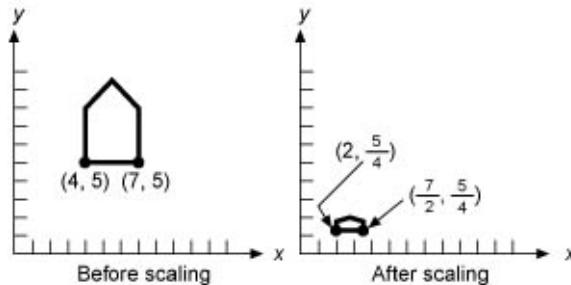


Figura 55 - Transformação de escala, com fatores :  $E_x = \frac{1}{2}$  ;  $E_y = \frac{1}{4}$ .

### Rotação

Uma transformação de rotação se caracteriza por girar um ponto em relação a outro ponto (no caso 2D) denominado centro da rotação. O efeito geométrico pode ser visto na figura 56.

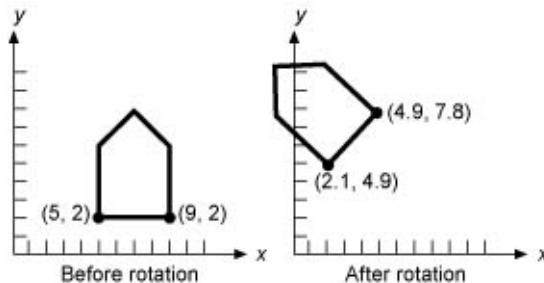


Figura 56 - Transformação de rotação em torno da origem.

Para entender como os coeficientes relacionados a essa transformação serão calculados vamos analisar a figura 57.

As coordenadas do ponto  $P$  podem ser escritas em função do ângulo  $\phi$  e de sua distância a origem :

$$\begin{aligned} x &= r \cos(\phi); \\ y &= r \sin(\phi); \end{aligned} \tag{1}$$

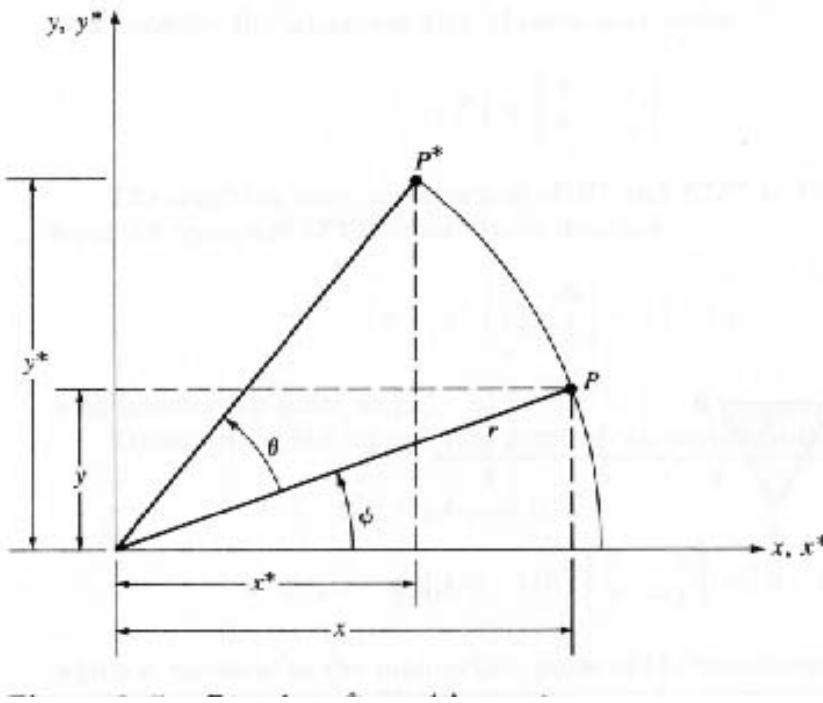


Figura 57 - Rotação de um ponto em torno da origem.

De forma análoga, podemos definir as coordenadas de  $P^*$  como :

$$\begin{aligned} x^* &= r \cos(\phi + \theta); \\ y^* &= r \sin(\phi + \theta); \end{aligned} \tag{2}$$

Aplicando as relações trigonométricas em (2) temos :

$$\begin{aligned} x^* &= r \cos(\phi) \cos(\theta) - r \sin(\phi) \sin(\theta); \\ y^* &= r \cos(\phi) \sin(\theta) + r \sin(\phi) \cos(\theta); \end{aligned} \tag{3}$$

Com base em (1) podemos reescrever (3) como :

$$\begin{aligned} x^* &= x \cos(\theta) - y \sin(\theta); \\ y^* &= x \sin(\theta) + y \cos(\theta); \end{aligned} \tag{3}$$

A partir dessa relação podemos, de forma direta, deduzir a forma geral da matriz de rotação :

$$P \cdot R = P'$$

$$\begin{bmatrix} x & y \end{bmatrix} \cdot R = \begin{bmatrix} x \cdot \cos(\theta) - y \cdot \sin(\theta) & x \cdot \sin(\theta) + y \cdot \cos(\theta) \end{bmatrix}$$

$$R = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

Transformações capazes de alterar a posição de um ponto localizado na origem do espaço são ditas **afins**. Essas transformações tem a forma :

$$P' = P + T_a \quad \text{onde} \quad T_a = [e \quad f]$$

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} + \begin{bmatrix} e & f \end{bmatrix} = \begin{bmatrix} x + e & y + f \end{bmatrix}$$

O efeito geométrico que essa transformação causa no pontos de um objeto pode ser visto na figura 58. Cada ponto é transladado na direção de um do eixo **x** ou **y** de um valor **e** ou **f**, respectivamente.

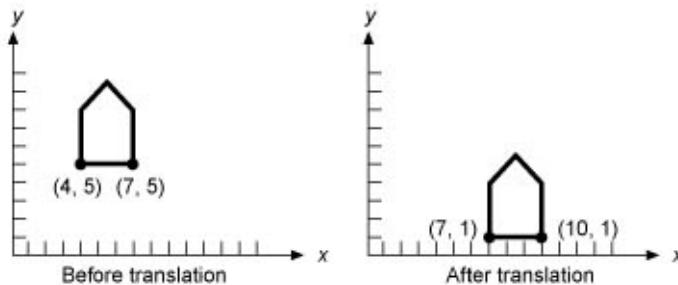


Figura 58 - Transformação de translação de um objeto.

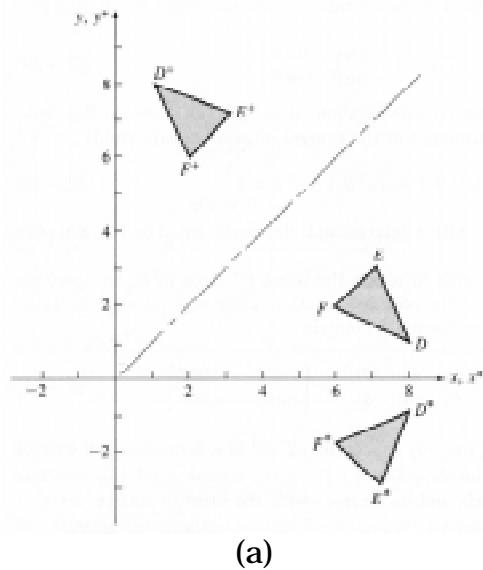
Esse tipo de transformação não pode ser representada por uma matriz, nos moldes das transformações lineares. Dessa forma uma transformação genérica, que possua componentes linear e afim tem a forma :

$$P' = P \cdot T_l + T_a \quad \text{onde} \quad T_l = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \text{e} \quad T_a = [e \quad f]$$

$$[x' \quad y'] = [x \quad y] \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} + [e \quad f] = [a.x + c.y + e \quad b.x + d.y + f]$$

Do ponto de vista prático, seria interessante obtermos uma forma unificada de representar as transformações afins e lineares. Na próxima seção estaremos analisando um artifício matemático que permite representar, em uma mesma matriz, as duas classes de transformações.

Deve ficar claro que descrevemos aqui apenas as transformações mais utilizadas dentro das aplicações gráficas. Várias outras transformações possuem interpretações geométricas de interesse. Algumas delas são mostradas na figura 59, como a reflexão e o cisalhamento. Fica a cargo do leitor, a título de exercício, a construção das matrizes dessas correspondentes a essas transformações.



(a)

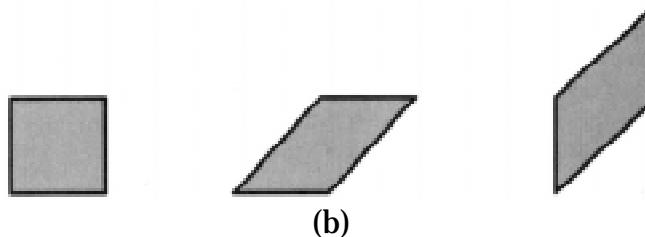


Figura 59 - Transformações de reflexão (a) e cisalhamento (b).

## Sistema de Coordenadas Homogêneas

O artifício utilizado para a unificação da representação das transformações lineares e afins é bem simples. Ao invés de trabalharmos no espaço bidimensional tradicional, utilizaremos o espaço tridimensional restrito a um plano – por exemplo  $z = h$ . Dessa forma um ponto  $P$  qualquer terá coordenadas :

$$P = [x' \ y' \ h]$$

onde  $x = \frac{x'}{h}$  e  $y = \frac{y'}{h}$

Se considerarmos  $h = 1$  temos :

$$P = [x \ y \ 1]$$

A interpretação geométrica desse artifício pode ser analisada na figura 60.

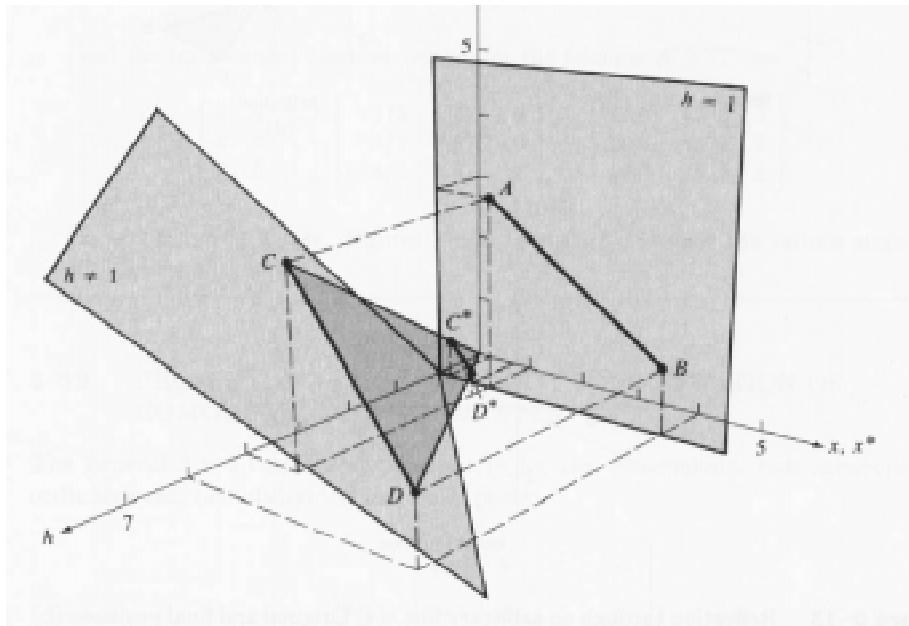


Figura 60 - Interpretação geométrica para o sistema de coordenadas homogêneas : um plano ( $h = 1$ ) imerso no espaço tridimensional.

Nesse espaço as transformações são definidas como matrizes de dimensão 3 :

$$P' = P \cdot M$$

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot M = \begin{bmatrix} a.x + c.y + e & b.x + d.y + f & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Na matriz  $M$  é possível representar as transformações afins e lineares de forma direta : a sub-matriz superior esquerda ( $2 \times 2$ ) armazenada a parte linear da transformação, ao passo que o vetor inferior esquerdo ( $1 \times 2$ ) é responsável pela componente afim da transformação. Em geral o vetor ( $3 \times 1$ ) a direita é constante ( $0,0,1$ ).

Uma justificativa intuitiva para esse artifício é que dentro do plano  $z=k$  a origem do subespaço de dimensão 2 (plano) é o ponto  $(0, 0, k)$ . Esse ponto, no entanto, não é origem do espaço tridimensional  $(0,0,0)$ . Portanto, qualquer transformação restrita ao subespaço definido pelo plano  $z=k$  não é capaz de alterar a origem do espaço 3D.

As transformações apresentadas na seção anterior podem ser re-escritas em coordenadas homogêneas :

### Escala :

$$E = \begin{bmatrix} E_x & 0 & 0 \\ 0 & E_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

### Rotação (em torno da origem) :

$$R = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

### Translação :

$$T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ d_x & d_y & 1 \end{bmatrix}$$

## Transformações Tridimensionais

As transformações geométricas tridimensionais são, em sua maioria, extensões das suas versões bidimensionais. Portanto, temos o mesmo problema de conjugar as transformações lineares e afins. Lançando mão do mesmo artifício de representar os pontos do espaço utilizando coordenadas homogêneas, temos a forma geral de uma transformação tridimensional :

$$P' = P \cdot M = [x \ y \ z \ 1] \cdot \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

Da mesma forma que no caso bidimensional, essa matriz pode ser subdividida em 3 submatrizes, a saber : a submatriz quadrada superior esquerda (3x3) representa a componente linear da transformação; o vetor inferior esquerdo (1x3) representa a componente afim da transformação; e o vetor coluna da direita (4x1) é responsável em garantir que os pontos terão a quarta coordenada igual a 1, portanto tem valores fixos (0,0,0,1).

A seguir apresentaremos as principais transformações tridimensionais descritas no sistema de coordenadas homogêneas.

### Escala :

$$P' = P \cdot E = [E_x \cdot x \ E_y \cdot y \ E_z \cdot z \ 1]$$

$$E = \begin{bmatrix} E_x & 0 & 0 & 0 \\ 0 & E_y & 0 & 0 \\ 0 & 0 & E_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Rotação (em torno de um dos eixos) :

No caso das rotações é importante destacar que no espaço 3D elas são definidas a partir de um eixo de rotação. Portanto temos três variações de rotações básicas, cada uma tendo como base um dos eixos coordenados como eixo de rotação :

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Translação:**

$$P' = P \cdot T = [x + dx \quad y + dy \quad z + dz \quad 1]$$

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

## Composição de Transformações

Até o momento apresentamos apenas transformações geométricas elementares. Apesar de não serem as únicas, elas são as ferramentas básicas para a geração de transformações mais complexas.

A idéia é que essas transformações podem ser combinadas, ou seja, aplicadas em seqüência, de modo que o efeito final seja o de uma transformação mais complicada.

Vejamos um exemplo. As transformações de rotação que apresentamos só se aplicam para rotações ao redor de um dos eixos coordenadas. Suponha que desejamos rodar um cubo ao redor do eixo que passa pelo seu centro de massa e é paralelo ao eixo **z**, tal como na figura 61. Que matriz produz essa transformação?

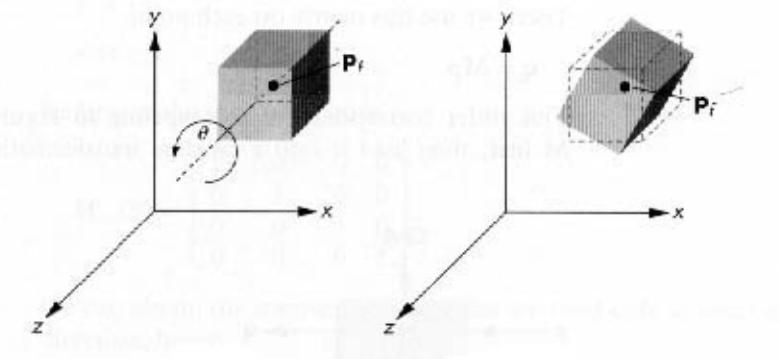


Figura 61 - Rotação de um objeto tridimensional em torno de um eixo paralelo ao eixo z, e que passa pelo seu centro de massa.

Podemos usar duas abordagens para determinar tal transformação. A primeira seria, com base em alguns pontos do objeto montar um sistema de equações, resolver esse sistema e obter os coeficientes da matriz de transformação. A grande desvantagem desse enfoque é ter que montar e resolver um sistema de equações toda a vez que quisermos aplicar uma transformação diferente.

Uma segunda forma, mais simples e genérica, é aplicar uma sucessão de transformações básicas que forneçam o resultado desejado. A partir da figura acima podemos observar que, caso o eixo a partir do qual estamos fazendo a rotação fosse coincidente com um dos eixos coordenados, poderíamos aplicar uma transformação de rotação elementar. Portanto, se aplicarmos ao objeto uma translação apropriada podemos fazer com que o eixo de rotação geral coincida com o eixo z. A partir daí podemos aplicar uma rotação elementar, e em seguida voltar com o objeto a sua posição original, aplicando outra translação simétrica a primeira. A figura 62 mostra essa seqüência de transformações.

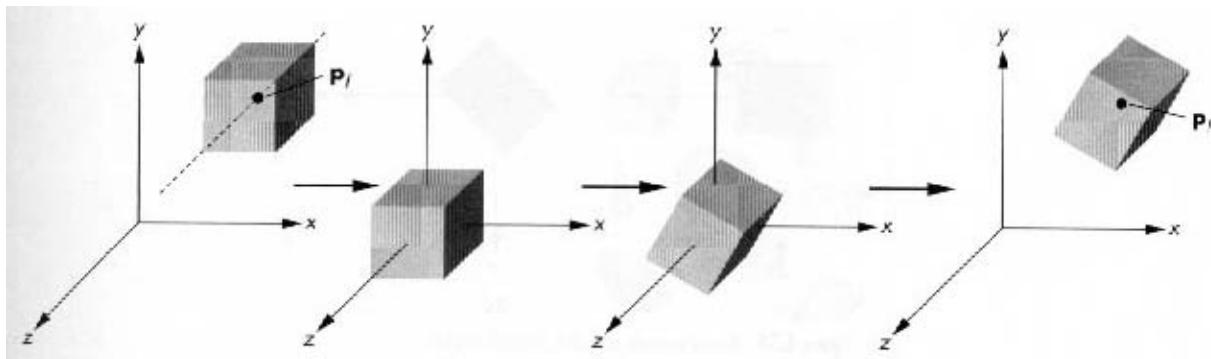


Figura 62 - Composição de transformações elementares equivalentes a transformação da figura 7.

Dessa forma aumentamos significativamente o número de transformações que um sistema gráfico pode suportar, apenas manipulando as transformações básicas de translação, rotação e escala.

Um cuidado que devemos ter é com a ordem com a qual estamos definido as transformações geométrica no processo de composição. Como a figura 63 exemplifica, nem todas as transformações são comutativas. Portanto a ordem em que elas são aplicadas pode alterar o posicionamento/forma final dos objetos submetidos a essa composição.

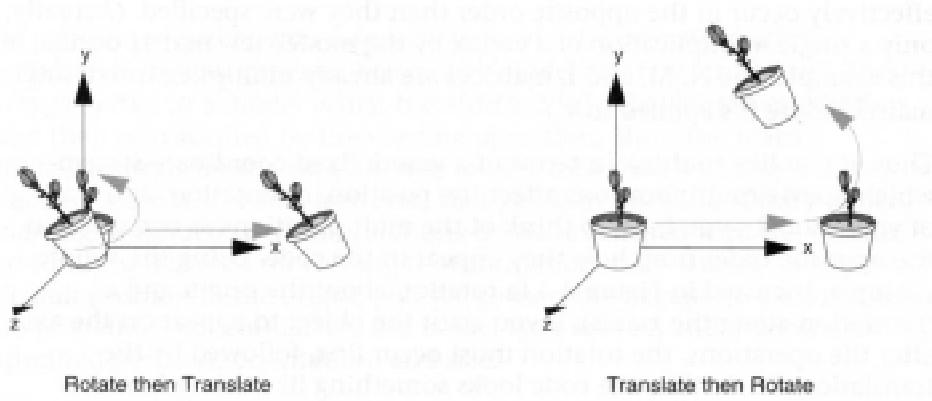


Figura 63 – Exemplo de composição de duas transformações geométricas que não são comutativas. Nesse caso a ordem das operações é relevante para o resultado final.

## Transformações de Deformação

As transformações apresentadas até o momento se caracterizam por transformarem o espaço no qual estão definidas uniformemente. Ou seja, uma escala de fator  $E_x = 2$ , por exemplo, dobra a coordenada  $x$  de **todos** os pontos do espaço de forma **idêntica**.

No entanto, em alguma situações pode ser interessante possuir uma transformação que altere de forma **não uniforme** os pontos do espaço. Nessa seção veremos uma classe de transformações que se possuem essa característica : as **deformações**.

Consideremos a figura 64 onde temos um objeto (tea-pot) e um cubo representando o espaço onde ele está definido - espaço  $E^3$ . Iremos considerar que os eixos coordenados estão dispostos da seguinte forma :  $z$  é o eixo vertical,  $x$  é o eixo horizontal e  $y$  é o eixo que parte da origem na direção do observador.

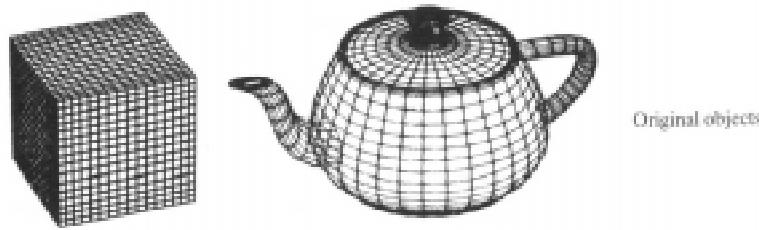


Figura 64 – Representação de um pote de chá (tea pot). Na figura a esquerda vemos a representação do espaço no qual o pote está definido –  $R^3$ .

Queremos definir transformações que, baseadas nas transformações já conhecidas, permitam variar seu “efeito” dentro do espaço. Essa variação, em geral, é conseguida pela aplicação de uma função as coordenadas dos espaço. Ou seja, uma transformação genérica de deformação, por definição tem a forma :

$$\begin{aligned} X &= F_x(x) \\ Y &= F_y(y) \\ Z &= F_z(z) \end{aligned}$$

onde  $\mathbf{X}$ ,  $\mathbf{Y}$  e  $\mathbf{Z}$  são as coordenadas transformadas do objeto e  $x$ ,  $y$  e  $z$  são suas coordenadas originais.

Variando a definição da função  $\mathbf{F}$  de forma adequada podemos conseguir alguns efeitos bastante interessantes.

### *Tapering*

A transformação de **Tapering** é uma deformação baseada na transformação de **Escala**. Essa transformação é dada por :

$$\begin{aligned} X &= x \\ Y &= y \\ Z &= E_z.z \end{aligned}$$

Se considerarmos, no entanto, que o fator de escala não é constante, mas sim uma função da coordenada  $x$ , ou seja :

$$E_z = f(x)$$

temos uma aplicação não uniforme da transformação de escala pelo domínio. Podemos então escrever a matriz de *tapering* como :

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & f(x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O efeito prático dessa transformação pode ser visto na figura 65. Quanto maior o valor da coordenada **x** maior a compressão na direção **z**.

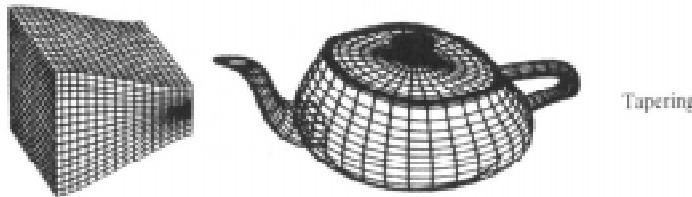


Figura 65 – Transformação de *tapering* aplicada no pote de chá. À direita o efeito produzido no espaço tridimensional.

### *Twisting*

A deformação de ***twisting*** (torção) é gerada com base em uma transformação de rotação. Por exemplo, considerando uma rotação em torno do eixo **z**, temos :

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

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

$$Z = z$$

Se fizermos com que o ângulo de rotação  $\theta$  não seja constante, mas sim uma função da coordenada **z**, ou seja :

$$\theta = f(z)$$

teremos uma transformação semelhante a representada na figura 66.

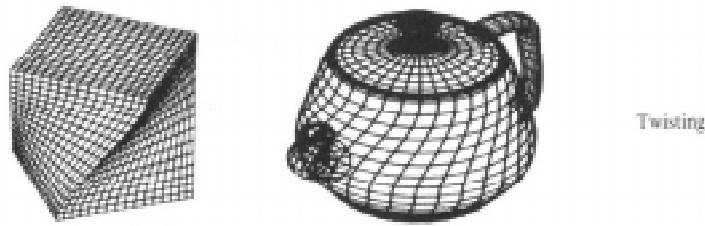


Figura 66 – Transformação de *twisting* aplicada no pote de chá. À direita o efeito produzido no espaço tridimensional.

Nesse caso a matriz de *twisting* pode ser escrita como :

$$T = \begin{bmatrix} \cos(f(z)) & \sin(f(z)) & 0 & 0 \\ -\sin(f(z)) & \cos(f(z)) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A definição de transformações de deformação pode ser de grande valia no processo de modelagem. Compondo adequadamente essas transformações podemos conseguir modelos bastante complexos de forma simples. Um exemplo é a figura 67, gerada com base em um cilindro e deformado através de operações de *twisting* e *tapering*.

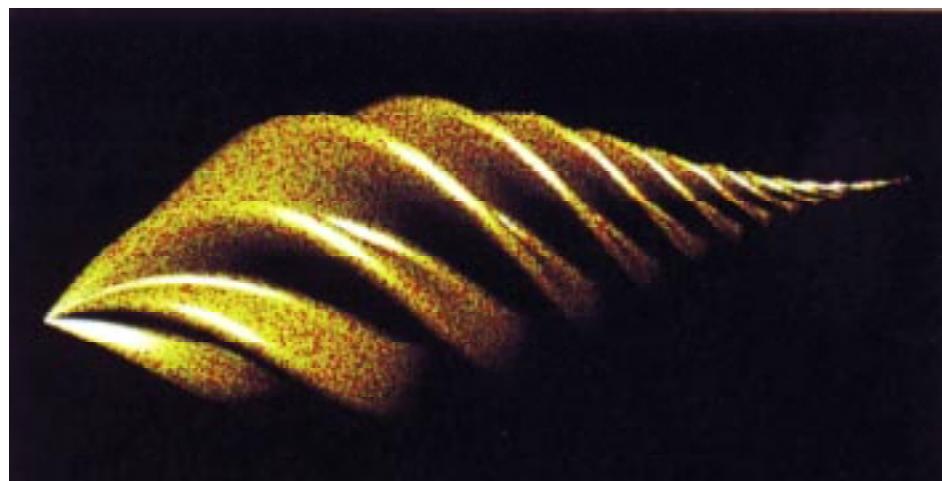


Figura 67 – Objeto construído a partir da aplicação de transformações de deformação de *twisting* e *tapering*.

Vale ressaltar que alguns cuidados devem ser tomados ao se trabalhar com transformações de deformação. Nem todas elas são passíveis de serem aplicadas em modelos poligonais. Alguns problemas devem se levados em consideração, tais como :

Certas operações podem reduzir a resolução da malha poligonal, implicando em problemas de degradação da silhueta dos modelos;

Alguns modelos podem não ter resolução suficiente para representar de forma adequada a deformação (imagine como ficaria a representação do cubo das figuras 64 a 66, caso este fosse representado apenas pelas suas 6 faces quadradas).

## Transformações Geométricas na biblioteca *OpenGL*

Conforme já estudamos, as transformações geométricas ocupam o primeiro estágio no *pipeline* de uma aplicação *OpenGL*.

A biblioteca *OpenGL* trabalha com matrizes de transformação expressas em coordenadas tridimensionais homogêneas, portanto, todas as matrizes tem dimensão 4x4.

Matrizes de transformação são utilizadas em duas situações dentro da *OpenGL* : para alterar forma e posicionamento dos objetos (modelos) ou para definir/alterar características do sistema de visualização. Nesse momento abordaremos apenas o primeiro caso, deixando o segundo para o próximo capítulo.

Para fazer essa indicação a função ***glMatrixMode*** é utilizada. Sua função é definir com qual tipo de matriz iremos trabalhar a partir de um determinado momento<sup>3</sup>. Esses tipos podem ser : ***ModelView***, ***Projection*** e ***Texture***. Por enquanto vamos restringir nosso estudo no primeiro tipo.

### *Manipulação das transformações*

Para cada modo de matrizes existe uma ***matriz corrente***. É com base nessa matriz que os pontos (vértices) dos objetos serão processados, na primeira etapa do *pipeline* gráfico.

<sup>3</sup> Lembre-se que a *OpenGL* funciona como uma máquina de estados. Portanto enquanto um novo comando ***MatrizMode*** não for emitido todas as operações de transformação serão aplicadas aquele modo.

A *OpenGL* permite a manipulação dessa matriz corrente de duas formas :

Via funções específicas

As transformações básicas são geradas pela *OpenGL* através de funções específicas. Essas funções não só montam a matriz de transformação associada, como também aplicam essa matriz, por composição, a matriz corrente.

São três essas transformações :

***glScalef***(Ex, Ey, Ez)

***glRotatef***(θ, x, y, z)

***glTranslatef***(Dx, Dy, Dz)

Via Funções de manipulação direta da matriz corrente

Podemos promover alterações diretamente na matriz de transformação corrente de duas maneiras : carregando uma matriz específica, ou multiplicando a matriz corrente por outra matriz. Isso é possível através das funções :

***glLoadMatrix*** (Matriz)

***glMultiMatrix*** (Matriz)

### *Estrutura de Pilha de Matrizes*

A *OpenGL* utiliza uma estrutura de pilha para armazenar várias matrizes de transformação. No modo *MODELVIEW* essa pilha possui no mínimo 32 matrizes. No modo *PROJECTION* a pilha possui somente 2 matrizes, conforme podemos ver na figura 68.

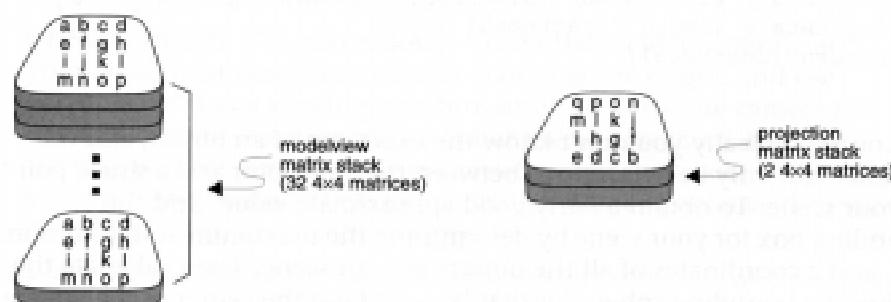


Figura 68 – Estrutura de pilhas utilizada pela biblioteca *OpenGL*. À esquerda temos a pilha associada ao modo *MODELVIEW* e à direita ao modo *PROJECTION*.

A principal motivação para o uso de uma pilha de matrizes é a possibilidade de “salvar” uma determinada configuração de transformações para posterior reutilização. Mas a frente apresentaremos um exemplo de aplicação que faz uso dessa facilidade.

Como em qualquer estrutura de pilha, só é possível, em um determinado instante, visualizar uma única matriz : aquela localizada no topo da pilha – matriz corrente.

A pilha pode ser manipulada por meio de duas funções básicas : ***glPopMatrix*** ou ***glPushMatrix*** (figura 69). Essa última faz com que uma nova matriz seja colocada no topo da pilha, enquanto que a primeira remove a matriz do topo. Fica claro que, em ambos os casos, haverá mudança na matriz corrente.

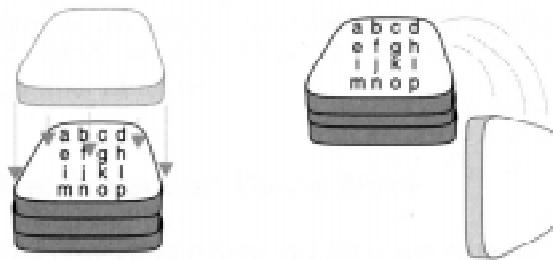


Figura 69 – Primitivas para manipulação da pilha de matrizes : à direita ***glPopMatrix*** e à esquerda ***glPushMatrix***.

### *Exemplo*

Para compreendermos melhor alguns detalhes a respeito do uso de matrizes no OpenGL iremos analisar uma aplicação bastante simples, mas que permite variações interessantes.

Iremos construir uma aplicação que simula o movimento de um planeta. Esse possui um satélite girando ao seu redor (figura 70). Temos portanto dois objetos esféricos, cada um com um movimento independente de rotação em torno de seu próprio eixo. O satélite possui ainda um movimento de rotação em torno do planeta.

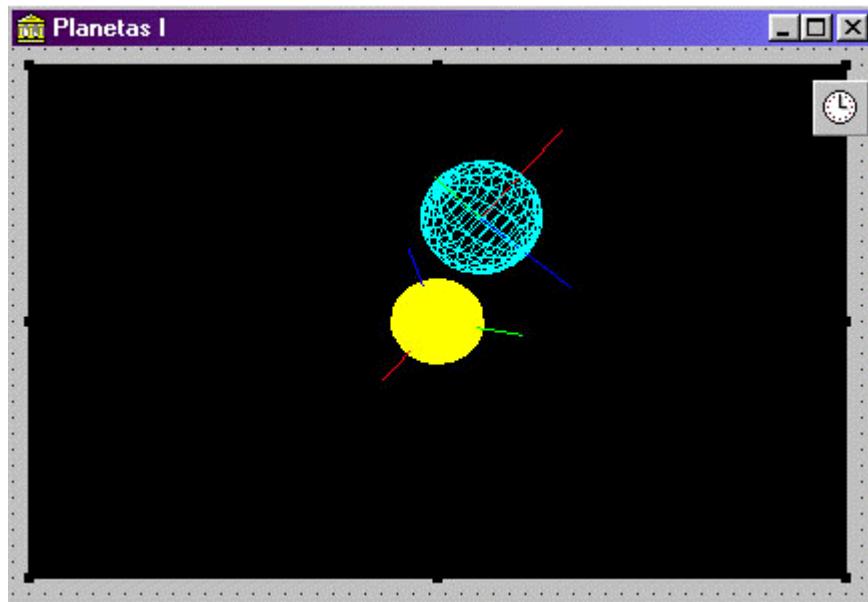


Figura 70 – Tela da aplicação exemplo.

A aplicação deverá, portanto, simular esses dois tipos de movimento através da composição de transformações.

Para um melhor entendimento iremos construir a função de desenho – associada ao evento *OnRender* - passo a passo. No código a seguir temos a primeira versão da aplicação. Ela apenas desenha a primeira esfera, rodando em torno de seu próprio eixo.

```

unit planet4;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs,
  OleCtrls, ExtCtrls, WAITEGLLib_TLB, OpenGL;
type
  TFrmPlanetas = class(TForm)
    Timer1: TTimer;
    GLWindow: TWaiteGL;
    procedure glSetupRC(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Timer1Timer(Sender: TObject);
    procedure glRender(Sender: TObject);
  private
    { Private declarations }
    ano, mes, dia : integer;
  public
    procedure DesenhaEixos;
    { Public declarations }
  end;

var
  FrmPlanetas: TFrmPlanetas;
implementation
{$R *.DFM}

procedure TFrmPlanetas.DesenhaEixos();
begin
  glBegin(GL_LINES);
  glColor4f(1.0, 0, 0, 1.0); // Eixo x
  glVertex3f( 0.0, 0.0, 0.0);
  glVertex3f(2.0, 0.0, 0.0);
  glColor4f(0, 1.0, 0, 1.0); // Eixo Y
  glVertex3f( 0.0, 0.0, 0.0);
  glVertex3f( 0.0, 2.0, 0.0);

```

```

  glEnd();
end;

procedure TFrmPlanetas.glSetupRC(Sender: TObject);
begin
  glClearColor(0.0, 0.0, 0.0, 1.0);

  glEnable(GL_DEPTH_TEST);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(60.0 , width/height , 1.0 , 20.0);
  gluLookAt(6.0, 6.0, 6.0, 0.0, 0.0, 0.0, 1.0 , 0.0,
  0.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glFlush();
  GLWindow.SwapBuffers();
end;

procedure TFrmPlanetas.FormActivate(Sender: TObject);
begin
  dia := 0;
  mes := 0;
  ano := 0;
  GLWindow.MakeCurrent();
end;

procedure TFrmPlanetas.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
  GLWindow.MakeNotCurrent();
end;

procedure TFrmPlanetas.Timer1Timer(Sender: TObject);
begin
  dia := dia + 10;

```

```
if dia >= 360 then
  dia := 0;

mes := mes + 5;
if mes >= 360 then
  mes := 0;

ano := ano + 1;
if ano >= 360 then
  ano := 0;

GLWindow.Invalidate(); // Força a ocorrência
de um evento de redesenho do componente.
end;

procedure TfrmPlanetas.glRender(Sender: TObject);
begin
  glClear(GL_COLOR_BUFFER_BIT);
  glClear(GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glRotatef(ano, 1.0, 0.0, 0.0);
  DesenhaEixos;
  glColor4f(1.0, 1.0, 1.0, 1.0);
  GLWindow.auxWireSphere(1.0);
  glPopMatrix();
  glFlush;
  GLWindow.SwapBuffers;
end;

end.
```

Para definir os ângulos de rotação para os três movimentos (rotação do planeta, rotação do satélite e rotação do satélite ao redor do planeta) foram criadas três variáveis globais : *dia*, *mês* e *ano*.

Para gerar uma seqüência animada utilizamos o componente **TTIMER**, que aparece na figura no canto superior direito. Esse componente gera, em intervalos de tempo constantes, eventos *OnTimer*. Na função que responde a esse evento atualizamos os ângulos de rotação e forçamos o redesenho da janela *OpenGL* – através do método *invalidate* do componente *WaiteGL*.

O procedimento associado ao evento *OnRender* é, portanto acionado a cada instante para redesenhar as esferas nos novos ângulos.

Nessa primeira versão apenas uma esfera é desenhada. Para isso o método *WaiteGL auxWireSphere* é utilizado.

Antes de desenhar a esfera, uma rotação de *ano* graus em torno do eixo *x* é aplicada. Como o ângulo da rotação é sempre acrescido de 1 grau no procedimento *OnTimer*, é necessário reinicializar a matriz corrente antes de cada transformação (o que acontece caso isso não seja feito ?). Ao invés de reinicializar explicitamente a matriz corrente com a função *glLoadIdentity*, foram utilizadas as funções *glPushMatrix* / *glPopMatrix*. Dessa forma a matriz corrente (que é inicializada com a identidade pelo procedimento *OnSetupRC*) é “guardada” na pilha de matrizes (Push) e depois recuperada (Pop). Cada comando *glPushMatrix* faz com que a matriz corrente (topo da pilha) seja replicada. O procedimento *glPopMatrix* remove a matriz do topo da pilha (corrente) e a substitui pela matriz imediatamente abaixo, na pilha.

Para acrescentar a segunda esfera – o satélite – basta compor uma translação e outra rotação a transformação corrente. A translação irá deslocar a nova esfera, enquanto que a rotação (de angulo dia) irá fazer a nova esfera girar sobre o seu próprio eixo.

O código a seguir implementa essa idéia.

```
procedure TFrmPlanetas.glRender(Sender: TObject);
begin
  glClear(GL_COLOR_BUFFER_BIT);
  glClear(GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glRotatef(ano, 0.0, 0.0, 1.0);
  DesenhaEixos;
  glColor4f(1.0, 1.0, 1.0, 1.0);
  GLWindow.auxWireSphere(1.0);

  glTranslatef(3.0, 0.0, 0.0);
  glRotatef(dia, 0.0, 1.0, 0.0);
  DesenhaEixos;
  glColor4f(0.0, 1.0, 1.0, 1.0);
```

```

        GLWindow.auxWireSphere(1.0);
        glPopMatrix();
        glFlush;
        GLWindow.SwapBuffers;
    end;

```

Repare que a rotação de *ano* graus é comum aos dois objetos. Portanto a rotação da segunda esfera acompanha a rotação em torno do próprio eixo da primeira esfera. Para que esse vínculo seja desfeito devemos isolar as transformações de cada esfera. Isso pode ser feito através do tratamento da pilha de matrizes adequado, tal como no código abaixo.

```

procedure TFrmPlanetas.glRender(Sender: TObject);
begin
    glClear(GL_COLOR_BUFFER_BIT);
    glClear(GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(ano, 1.0, 1.0, 1.0);
    DesenhaEixos;
    glColor4f(1.0, 1.0, 0, 1.0);
    GLWindow.auxSolidSphere(1.0);
    glPopMatrix();

    glPushMatrix();
    glRotatef(mes, 0.0, 1.0, 0.0);
    glTranslatef(3.0, 0.0, 0.0);
    glRotatef(dia, 0.0, 0.0, 1.0);
    DesenhaEixos;
    glColor4f(0.0, 1.0, 1.0, 1.0);
    GLWindow.auxWireSphere(1.0);
    glPopMatrix();
    glFlush;
    GLWindow.SwapBuffers;
end;

```

Nesse caso não temos a rotação da primeira esfera não exerce nenhuma influência na rotação da segunda. A chamada a função `glPopMatrix` depois do desenho da primeira esfera faz com que a matriz corrente seja reinicializada com matriz identidade. A chamada `glPushMatrix` subsequente guarda novamente a identidade na pilha e aplica uma rotação de *dia* graus (equivalente a rotação em torno de seu próprio eixo), uma translação (para deslocar a esfera em relação a primeira) e uma nova rotação (diferente daquela aplicada a primeira esfera) equivalente ao movimento em torno da primeira esfera.

## Referência

***glMatrixMode (GLenum modo);***

Método WaiteGL :

***MatrixMode (long modo);***

Valores :

<b><i>OpenGL</i></b>	<b><i>WaiteGL</i></b>
GL_MODELVIEW	glModelView();
GL_PROJECTION	glProjection();
GL_TEXTURE	glTexture();

Descrição :

***Especifica qual das 3 matrizes correntes será afetada pelas transformações geométricas especificadas a seguir.***

***glLoadIdentity (void);***

Método WaiteGL :

***LoadIdentity 0;***

Descrição :

***Inicializa a matriz do modo corrente com uma matriz identidade.***

***glLoadMatrix {fd} (const TYPE \*m);***

Método WaiteGL :

***LoadMatrix (float FAR\* m);***

Descrição :

***Carrega a matriz especificada pelo parâmetro m na matriz do modo corrente.***

***glMultiMatrix {fd} (const TYPE \*m);***

Método WaiteGL :

***MultiMatrix (float FAR\* m);***

Descrição :

***Multiplica a matriz especificada no parâmetro m pela matriz do modo corrente. O resultado é armazenado na matriz corrente.***

***glTranslate {fd} (TYPE x, TYPE y, TYPE z);***

Método WaiteGL :

***Translate (float x, float y, float z);***

Descrição :

***Multiplica a matriz do modo corrente pela matriz de translação gerada a partir dos parâmetros x, y e z - valores dos deslocamento nos respectivos eixos.***

***glRotate {fd} (TYPE ângulo, TYPE x, TYPE y, TYPE z);***

Método WaiteGL :

***Rotate (float ângulo, float x, float y, float z);***

Descrição :

***Multiplica a matriz do modo corrente pela matriz de rotação em torno de um eixo, definido pelo vetor que liga a origem ao ponto (x,y,z). A rotação é feita no sentido anti-horário de um valor - expresso em graus - pelo parâmetro ângulo.***

***glScale {fd} (TYPE x, TYPE y, TYPE z);***

Método WaiteGL :

***Scale (float x, float y, float z);***

Descrição :

***Multiplica a matriz do modo corrente pela matriz de escala gerada a partir dos fatores de escala definidos por x, y e z, em cada um dos respectivos eixos.***

***glPushMatrix (void);***

Método WaiteGL :

***PushMatrix ();***

Descrição :

***Empilha uma cópia da matriz corrente na estrutura de pilha associada ao modo corrente.***

***glPopMatrix (void);***

Método WaiteGL :

***PopMatrix ();***

Descrição :

***Desempilha a matriz corrente da estrutura de pilha associada ao modo corrente. A matriz imediatamente abaixo passa a ser a matriz corrente.***

# CAPÍTULO IV

## SISTEMA DE VISUALIZAÇÃO

---

Dando seqüência a análise das etapas do pipeline de uma aplicação gráfica, temos as fases de recorte (*clipping*) e projeção. Essas duas etapas serão avaliadas em conjunto nesse capítulo, já que ambas dependem basicamente dos parâmetros definidos pelo ***Sistema de Visualização*** associado.

Em outras palavras : para definir o que será visto da cena (*recorte*) e como será visto (*projeção*) é necessário conhecer dados sobre quem observa a cena.

Apresentaremos um paradigma para construção do sistema de visualização baseado em uma ***camera virtual***. Iremos avaliar quais são e como devem ser definidos os seus controles, bem como o processo de geração da imagem bidimensional associada. Nesse último tópico estaremos interessados em estudar as transformações de projeção, em particular as projeções paralela e perspectiva.

Fechando o capítulo apresentaremos um exemplo de implementação utilizando a biblioteca gráfica ***OpenGL***.

### Construção do Sistema de Visualização

O processo de definição de um sistema de visualização pode ser comparado a preparação de um fotografo para tirar uma fotografia (figura 71). As etapas relacionadas com o processo da fotografia são :

Definir o posicionamento da camera,

Definir o posicionamento dos objetos

Definição da lente utilizada e

Geração (revelação) da fotografia.

A contrapartida no mundo virtual possui as mesmas etapas. No entanto, elas devem estar sempre associadas a uma representação abstrata que consiga reproduzir o efeito do mundo real. Por exemplo, a definição da lente utilizada por um fotografo define implicitamente parâmetros tais como : campo de visão e distância focal. No caso da camera virtual é necessário que esses parâmetros sejam definidos explicitamente, para compormos um modelo matemático que possa transformar objetos 3D em uma imagem 2D.

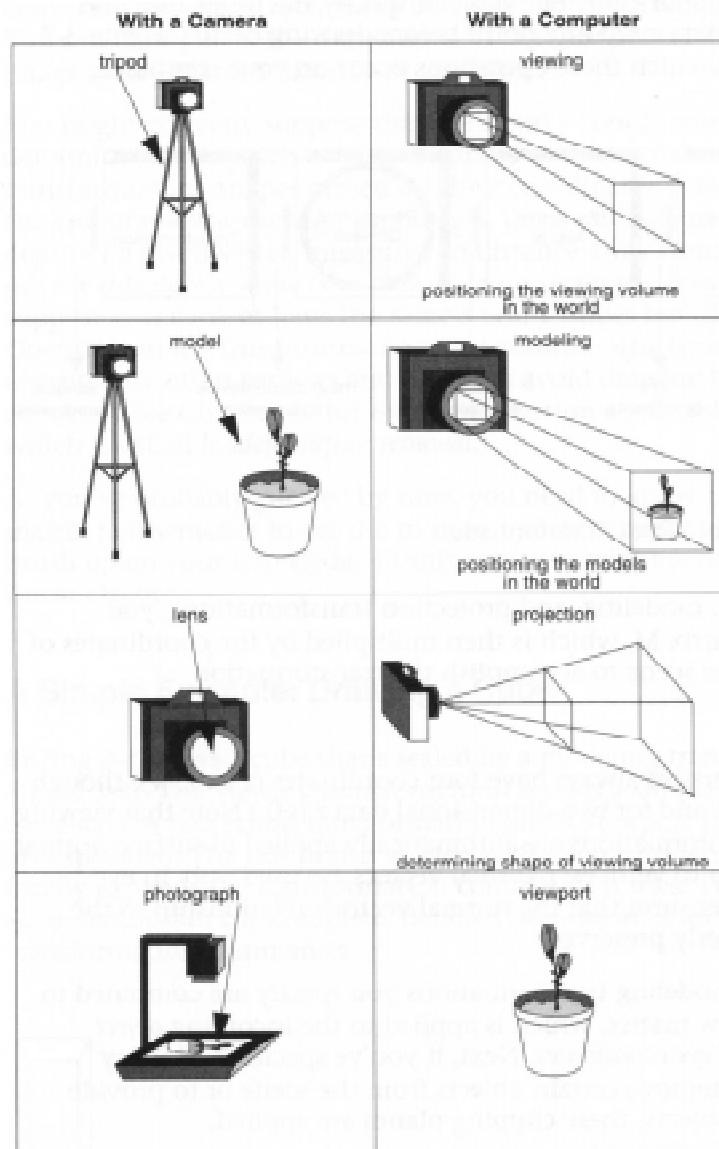


Figura 71 - Comparação entre os processos de geração de uma imagem real (fotográfica) e sintética (com um computador).

Vamos iniciar a análise da construção de um sistema de visualização estudando os tópicos relacionados com o posicionamento de uma camera no espaço. Para nossa facilidade iremos considerar que as “fotos” que queremos tirar são de um ambiente natural, onde a única fonte de luz é a solar. Ou seja, não temos que controlar fontes de luz artificiais.

## *Posição da Camera no Espaço*

O primeiro parâmetro relacionado a uma camera virtual é a sua posição no espaço. Essa posição pode ser caracterizada por um ponto no espaço 3D. Nesse ponto será posicionada a camera virtual. Na figura 72 podemos ver esse ponto, denominado ***view point***.

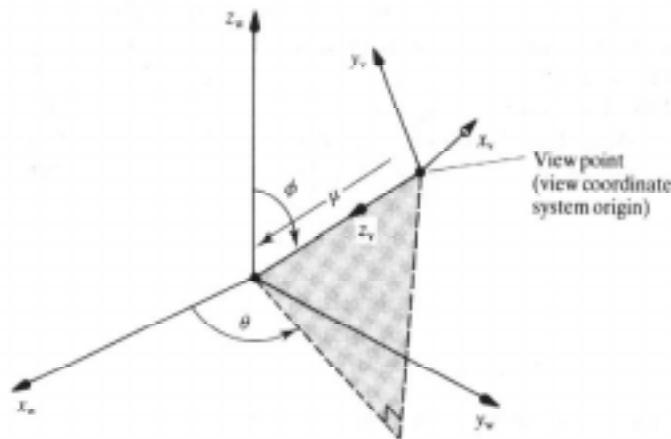


Figura 72 - Definição do posicionamento de uma camera virtual.

O *view point* pode ser definido explicitamente por suas coordenadas cartesianas ( $x, y, z$ ). Uma outra forma de defini-lo é a partir de suas coordenadas esféricas (o equivalente as coordenadas polares bidimensionais). Nesse sistema de coordenadas o ponto *view point* é caracterizado por dois ângulos ( $\phi$  e  $\theta$ , valores de rotação em torno dos eixos  $x$  e  $z$ , respectivamente) e sua distância ( $\rho$ ) à origem. A figura 72 mostra esses três parâmetros.

## *Direção de Observação*

Uma vez fixado o *view point* em um ponto  $C$  no espaço, temos que definir a direção para a qual iremos apontar a camera. Essa direção será dada a partir de um vetor, cuja origem é o ponto  $C$  e por um outro ponto. Esse ponto é denominado ***foco*** e está associado ao ponto  $F$  na figura 73. A definição de um ponto no espaço associado a direção de observação é simples, porém pouco intuitiva.

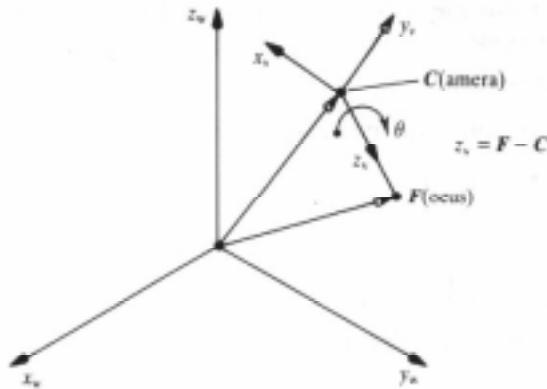


Figura 73 – Definição da direção de observação.

Uma forma alternativa, e bem mais intuitiva, pode ser definida com base em dois ângulos, relativos a rotações em torno do ponto **C**. Essas rotações são definidas da seguinte forma : suponha que o vetor que define a direção de observação é unitário. Nesse caso o lugar geométrico de todos os possíveis pontos extremos desse vetor é uma esfera de raio 1, centrada no ponto **C**. Portanto, para definirmos o ponto extremo do vetor direção sobre essa esfera, basta especificar um angulo relativo a elevação do ponto e outro relativo ao seu deslocamento horizontal.

Esses movimentos são denominados **tilt** e **pan** (ou **pitch**). A figura 74 ilustra esse dois movimentos.

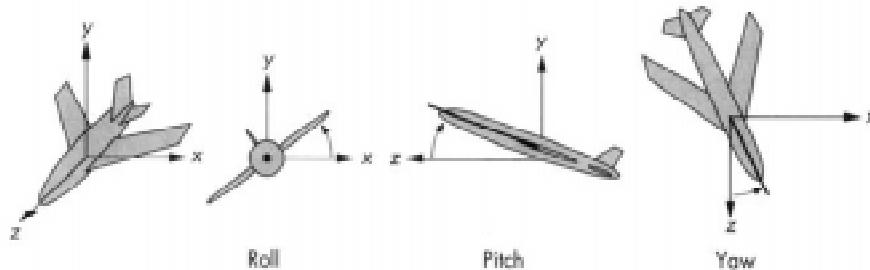


Figura 74 – Movimentos associados a uma objeto qualquer centrado em um ponto no espaço 3D.

### Orientação da Camera

Uma vez definida a localização da camera e sua direção de observação ainda temos um grau de liberdade : a orientação da camera relativa a direção de observação. Em outras palavras, dada uma direção de observação, é possível ainda rotacionar a camera em relação a esse vetor de um angulo  $\theta$ , como mostra

a figura 73. Na prática, essa rotação é equivalente ao posicionamento da camera : horizontal, vertical, ou “cabeça para baixo”, por exemplo. Esse movimento de rotação é denominado **roll** e pode ser visto também na figura 74.

### *Tipo de Lente da Camera*

Dados o ponto e a direção de observação ainda não podemos caracterizar de forma precisa quais objetos serão visíveis. Tão pouco como será a imagem gerada.

Em uma camera real, podemos variar o tipo de lente utilizada para obter diferentes efeitos na imagem final. A definição de uma lente está associada a diversos parâmetros relativos a imagem. O primeiro é o **campo de visão (FOV – Field Of Vision)**. O *campo de visão* define um ângulo dentro do qual os objetos podem ser vistos, como na figura 75. Logo, dada uma lente podemos definir a região da cena que será visível, e caracterizar a fase de recorte. Essa região tem a forma de um cone de pirâmide de base retangular.

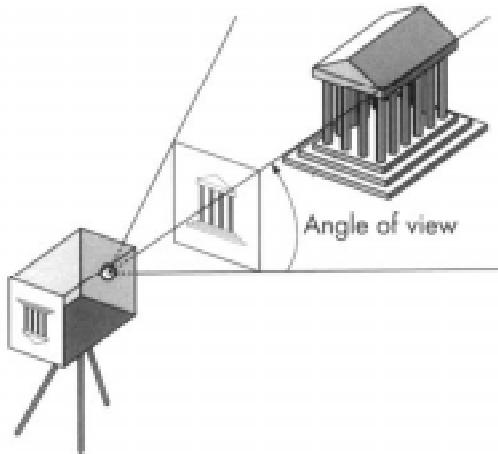


Figura 75 – Angulo de visualização ou campo de visão (FOV) associado a uma camera.

Em uma máquina fotográfica real, objetos muito próximos ou muito distantes tendem a aparecer “deformados” na fotografia. Dizemos que esses objetos estão “fora de foco”, pois se encontram em uma região onde a imagem definida através da lente não fica nítida.

Da mesma forma, iremos associar uma região do campo de visão onde os objetos aparecerão sem distorções na imagem final. Essa faixa é definida por dois planos : o **plano de recorte frontal** e o **plano de recorte dorsal** (respectivamente *front clipping plane* e *back clipping plane* na figura 76). Esse

planos também podem ser chamados plano mais próximo (*near plane*) e plano mais distante (*far plane*). Nesse caso a região visível passa a ser definida pelo tronco de pirâmide (figura 6). Essa porção do espaço visível é denominada **Volume de Visão** (*view volume*).

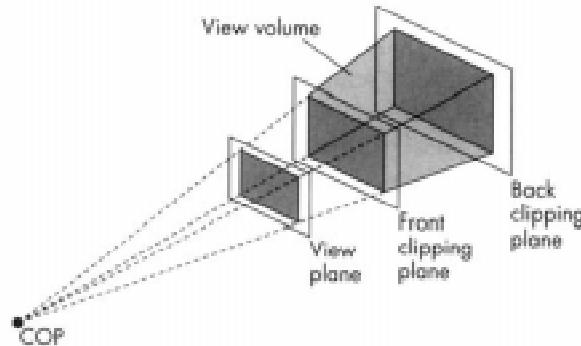


Figura 76 - Definição de um volume de visão.

Em resumo, a fase de recorte elimina os objetos que não se encontram dentro do campo de visão da camera virtual, além dos objetos que se encontrariam fora de foco.

### *Sistema de Coordenadas da Camera*

A definição de uma camera virtual em um sistema de visualização está diretamente associada a existência de um segundo sistema de coordenadas dentro da aplicação – relacionado com a camera virtual – tal como na figura 77.

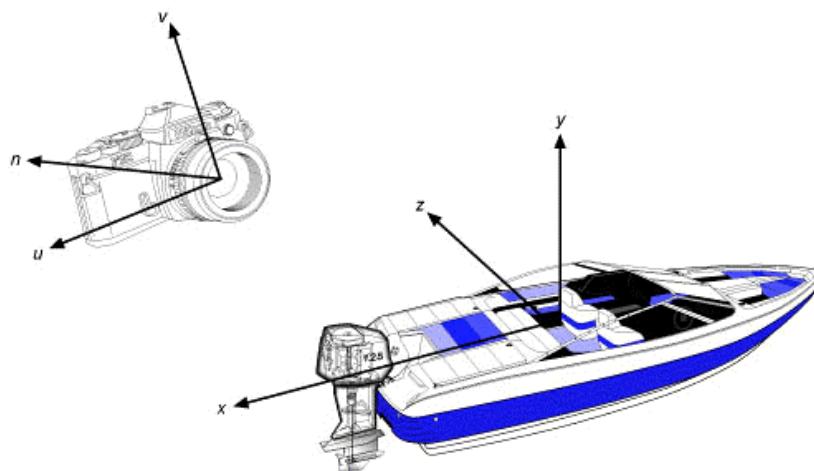


Figura 77 - Caracterização dos dois sistemas de coordenadas definidos por uma camera virtual dentro de um sistema de visualização.

Dessa forma temos :

### Sistema de Coordenadas Global

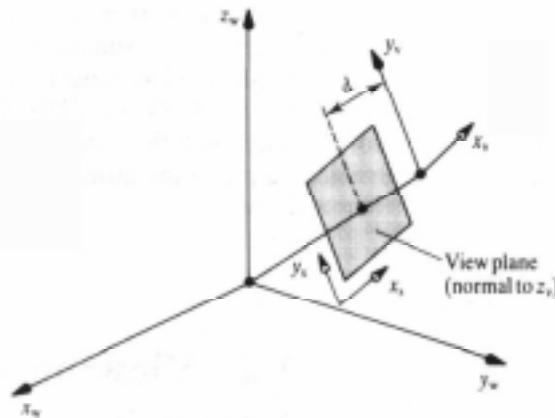
É nesse sistema que a cena (objetos e camera) é descrita. Esse sistema é definido pelos eixos coordenados  $x_w$ ,  $y_w$  e  $z_w$  (figura 8).

### Sistema de Coordenadas da Camera

É definido a partir do posicionamento da camera virtual dentro do sistema de coordenadas global. Será com base nesse sistema de coordenadas que o recorte e a projeção serão definidos. Esse sistema é definido pelos eixos coordenados  $x_c$ ,  $y_c$  e  $z_c$  (figura 78).

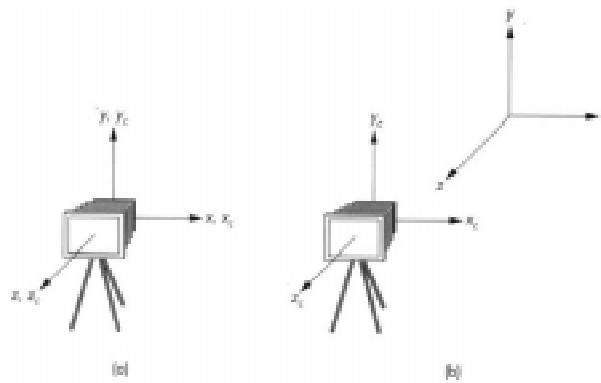
### Sistema de Coordenadas da Imagem

É o sistema onde estão definidos os pixels que irão compor a imagem. É definido pelos eixos  $x_s$  e  $y_s$  (figura 78).



**Figura 78 - Definição dos sistemas de coordenadas global da camera e da imagem.**

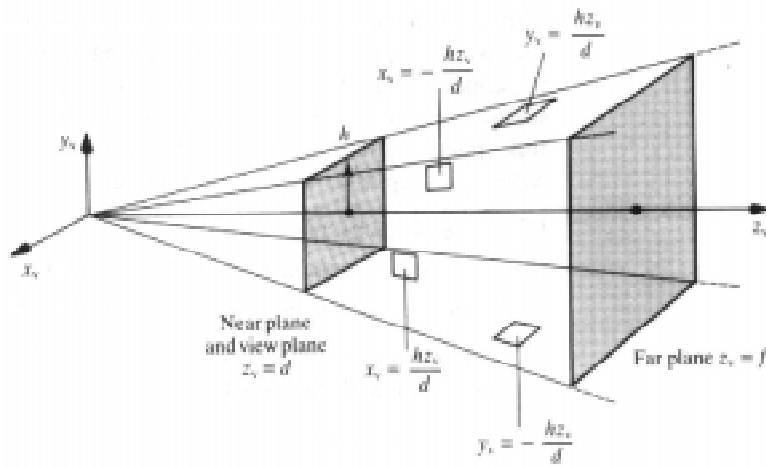
Na figura 79 podemos observar a grande vantagem de se definir, em função da camera, um outro sistema de coordenadas. Ao movimentarmos a camera pelo espaço, o seu sistema de coordenadas sofre o mesmo movimento, ao passo que o sistema de coordenadas global permanece estático.



**Figura 79 – A movimentação da camera virtual define uma movimentação equivalente do seu sistema de coordenadas, enquanto que o sistema global permanece estático.**

Portanto, no sistema de coordenadas da camera os cálculos das fases de recorte e projeção podem ser muito mais simples que no sistema global. Conforme podemos observar na figura 80, se considerarmos um sistema de coordenadas onde a camera esteja centrada na origem, e cuja direção de observação seja coincidente com o eixo  $z$  temos as equações dos planos de recorte muito simples.

Além disso, uma vez que esses planos de recorte são definidos em relação a camera, mesmo que essa se movimente eles não terão de ser recalculados. Caso a sua definição fosse feita no sistema de coordenadas global toda vez que a camera mudasse de posição, novos planos de recorte teriam que ser calculados.



**Figura 80 - Definição dos planos de recorte para uma camera centrada na origem com direção de observação coincidente com eixo  $z$ .**

Esses mesmos argumentos podem ser aplicados a fase de projeção, conforme veremos mais adiante.

Nesse ponto deve estar claro que para podermos utilizar as facilidades de cálculo associadas ao sistema de coordenadas da camera, temos que ter os objetos descritos nesse sistema. Originalmente os objetos são definidos no sistema de coordenadas global, ou seja, possuem coordenadas descritas nesse sistema. Portanto, é preciso definir uma transformação que permita que, dada a coordenada de um ponto no sistema global, a coordenada relativa a camera desse ponto possa ser calculada.

Essa transformação nós denominamos de ***mudança de sistema de coordenadas***. Sua construção é bastante simples : podemos gera-la através da composição das transformações necessárias para levar o sistema de coordenadas da camera a coincidir com o sistema de coordenadas global.

Considere o ponto  $P$  descrito no sistema de coordenadas definido pelos vetores  $X$  e  $Y$ , como na figura 81. É possível representar esse ponto pela expressão :

$$P = x_P \cdot X + y_P \cdot Y + O \quad (1)$$

onde  $x_p$  e  $y_p$  são ditas coordenadas de  $P$  no sistema de coordenadas definido por  $X$  e  $Y$ . O ponto  $O$  representa a origem do sistema de coordenadas.

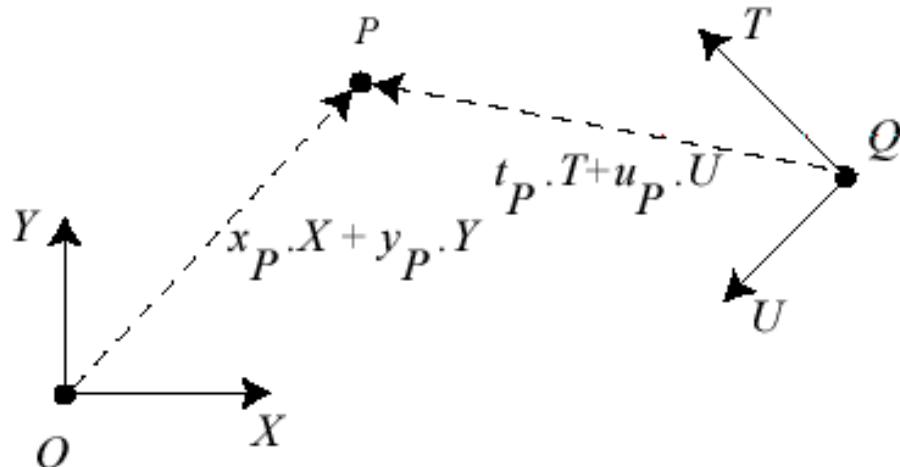


Figura 81 - O ponto  $P$  descrito em dois sistemas de coordenadas distintos.

Agora considere um novo par de vetores  $T$  e  $U$ . Esses vetores descrevem um outro sistema de coordenadas. O ponto  $P$  nesse outro sistema será descrito pela expressão :

$$P = t_p \cdot T + u_p \cdot U + Q$$

Portanto, nesse novo sistema de coordenadas o ponto  $\mathbf{P}$  é descrito pelas coordenadas  $\mathbf{t}_p$  e  $\mathbf{u}_p$ .

Uma vez conhecendo as coordenadas do ponto  $\mathbf{P}$  em uma sistema de coordenadas, digamos  $\mathbf{XYO}$ , e conhecendo também os vetores que descrevem nosso segundo sistema de coordenadas, podemos construir uma transformação que, aplicada as coordenadas do ponto  $\mathbf{P}$  em  $\mathbf{XYO}$ , fornece as suas coordenadas no sistema  $\mathbf{TUQ}$ .

Se decretarmos  $\mathbf{T}$ ,  $\mathbf{U}$  e  $\mathbf{Q}$  em relação ao sistema  $\mathbf{XYO}$  chegamos a expressão :

$$\begin{aligned} \mathbf{P} &= t_P \cdot \mathbf{T} + u_P \cdot \mathbf{U} + \mathbf{Q} \\ &= t_P \cdot (x_T \cdot X + y_T \cdot Y) + u_P \cdot (x_U \cdot X + y_U \cdot Y) + (x_Q \cdot X + y_Q \cdot Y + O) \\ &= (t_P \cdot x_T + u_P \cdot x_U + x_Q) \cdot X + (t_P \cdot y_T + u_P \cdot y_U + y_Q) \cdot Y + O \end{aligned}$$

Comparando com a expressão (1) podemos concluir que :

$$\begin{aligned} x_P &= t_P \cdot x_T + u_P \cdot x_U + x_Q \\ y_P &= t_P \cdot y_T + u_P \cdot y_U + y_Q \end{aligned}$$

Ou seja :

$$\begin{bmatrix} x_P \\ y_P \end{bmatrix} = \begin{bmatrix} x_T & x_U \\ y_T & y_U \end{bmatrix} \times \begin{bmatrix} t_P \\ u_P \end{bmatrix} + \begin{bmatrix} x_Q \\ y_Q \end{bmatrix}$$

Em coordenadas homogêneas a mesma expressão é representada por :

$$\begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} = \begin{bmatrix} x_T & x_U & x_Q \\ y_T & y_U & y_Q \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} t_P \\ u_P \\ 1 \end{bmatrix}$$

Portanto, a operação de mudança de sistema de coordenadas pode ser descrita por uma transformação geométrica afim.

Na expressão acima, a partir de um ponto descrito no sistema  $\mathbf{TUQ}$  geramos as coordenadas no sistema  $\mathbf{XYO}$ . Caso nosso interesse seja o contrário basta reescrevermos a expressão como :

$$\begin{bmatrix} t_P \\ u_P \\ 1 \end{bmatrix} = \begin{bmatrix} x_T & x_U & x_Q \\ y_T & y_U & y_Q \\ 0 & 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix}$$

## *Utilização de um Sistema de Visualização*

Uma vez definidos os parâmetros necessários a construção de um sistema de visualização uma pergunta aparece : como interagir com esse sistema ? Duas abordagens podem ser utilizadas :

Baseada na camera

Camera é estática e os objetos se movimentam no ambiente.

Baseada nos objetos

Os objetos são estáticos e a camera se movimenta dentro da cena.

A figura 82 ilustra a diferença entre as duas abordagens.

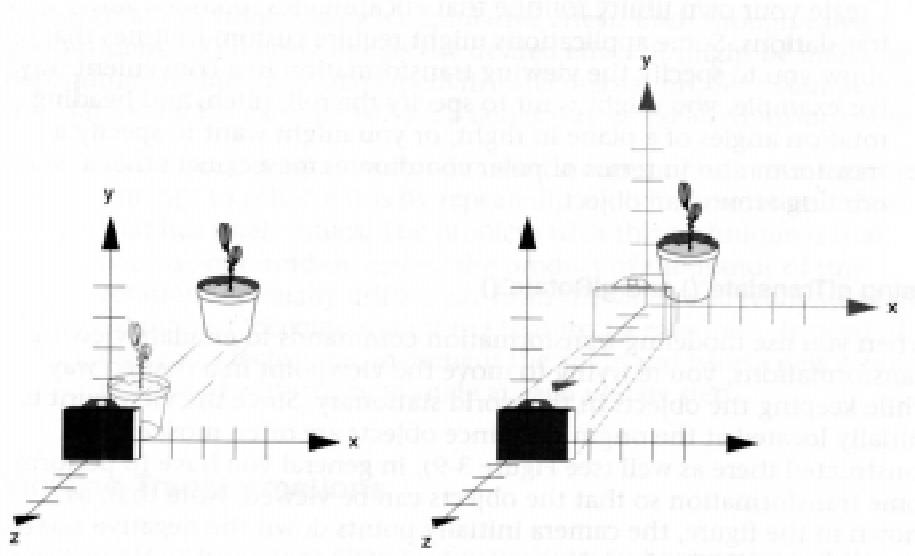


Figura 82 – Duas abordagens para movimentação em um sistema de visualização : objetos se movem e a camera fica estática (à esquerda) ou objetos ficam estáticos enquanto que a camera se movimenta (à direita).

No primeiro caso podemos fazer uma analogia com um fotógrafo de estúdio. Nesse caso pode ser mais simples posicionar os objetos em relação a camera, que fica fixa em um tripé.

No entanto, essa abordagem não se mostra adequada para fotografias da natureza ou mesmo de objetos arquitetônicos, como edificações por exemplo. Nesses casos o único modelo possível é movimentar a camera para obter o melhor posicionamento, já que os objetos estão “fixos” na cena.

Do ponto de vista matemático, as duas abordagens são equivalentes. No entanto, a interface fornecida por uma aplicação que trabalhe com uma ou outra abordagem pode ser bastante diferente.

Uma vez definido o posicionamento e as características da camera virtual – o que determina que objetos serão visíveis – a fase subsequente é a da geração da imagem. As ferramentas para essa tarefa são as **Transformações Projetivas**, que estaremos estudando na seção seguinte.

## Projeções Geométricas Planares

O processo de geração de uma imagem bidimensional, obtida a partir de um objeto tridimensional, é bastante conhecido. Seus princípios básicos são os mesmos associados ao processo de formação de uma imagem pelo sistema óptico humano.

Como premissa temos a propagação retilínea da luz. Portanto, para que um objeto qualquer possa ser visualizado é necessário que os raios de luz que partem dele (por um processo de reflexão ou emissão de luz) atinjam o sistema óptico do observador. Esses raios de luz podem ser considerados como retas que partem de cada ponto localizado na superfície visível do objeto.

Imagine um plano se interpondo entre o objeto e o observador (figura 83). Todas as retas (raios de luz) que partem do objeto e chegam no observador interceptam esse plano. Se calcularmos essas interseções e registrarmos a cor (intensidade de luz) do ponto correspondente no objeto, teremos a sua imagem bidimensional.

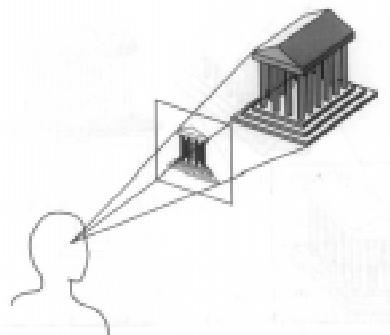


Figura 83 – Processo de projeção : formação de uma imagem bidimensional a partir de um objeto tridimensional.

## *Componentes Básicos de uma Projeção*

No processo descrito anteriormente, temos uma série de elementos envolvidos. Os elementos necessários para a construção de uma projeção são :

### ***Objetos***

### ***Projetores***

São os raios de luz que partem da superfície do objeto na direção do observador.

### ***Plano de Projeção***

É o lugar geométrico onde a imagem será gerada.

### ***Centro de Projeção (COP - Center of Projection)***

Nada mais é que o ponto no espaço onde o observador está localizado.

Na figura 84 podemos observar cada um desses componentes.

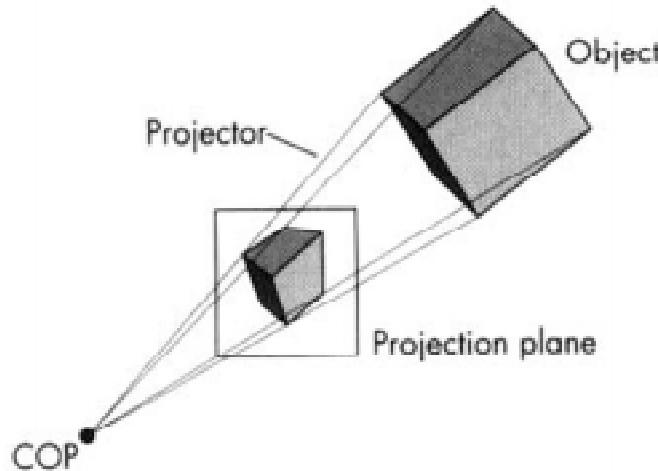


Figura 84 - Componentes básicos envolvidos no processo de projeção.

Imaginando o processo de geração da projeção (imagem) como um algoritmo temos :

Algoritmo GerarProjeçãoPara cada ponto da superfície do **objeto** faça

*Traçar um reta (**projetor**) que liga esse ponto ao **COP**;  
 Calcular a interseção do **projetor** como o **plano de projeção**;  
 Atribuir a esse ponto a cor (intensidade de luz) a partir do ponto correspondente da superfície do objeto*

fim-para

fim.

O conjunto de pontos gerados por esse algoritmo define a **projeção do objeto**, ou seja, sua a imagem bidimensional.

Um caso especial ocorre quando o ponto utilizado como centro de projeção está a uma distância muito grande dos objetos e do plano de projeção. Imagine, a partir da figura 84, que o **COP** comece a se distanciar. A tendência dos projetores é de se afastarem entre si. No entanto esse afastamento tem um limite. No caso do **COP** ser levado para uma distância infinita os projetores se tornam retas paralelas. Nesse caso temos a situação mostrada na figura 15. Portanto, não temos caracterizado um ponto para o qual os projetores convergem (já que ele está no infinito) mas sim uma **direção de projeção (DOP - Direction Of Projection)**.

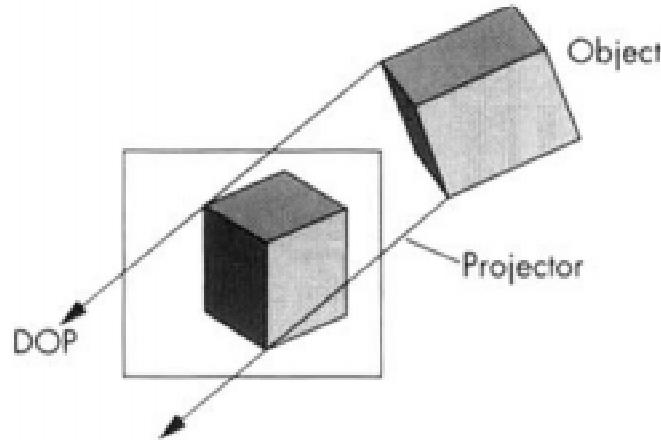


Figura 85 - Quando a distância do centro de projeção ao plano de projeção tende ao infinito temos projetores paralelos segundo uma **direção de projeção (DOP)**.

Conforme veremos a seguir essas duas situações caracterizam dois tipos de projeção : **perspectiva** e **paralela**.

## *Tipos de Projeção*

Com base no processo de geração de projeções descritos na seção anterior, podemos caracterizar dois tipos de projeção, com base no posicionamento do centro de projeção.

### **Projeção Perspectiva**

O centro de projeção está localizado a uma distância finita dos objetos e do plano de projeção.

### **Projeção Paralela**

Quando o centro de projeção está localizado a uma distância infinita em relação ao plano de projeção e aos objetos.

Na figura 86 temos um gráfico apresentando as possíveis variações de projeções dentro de cada tipo básico.

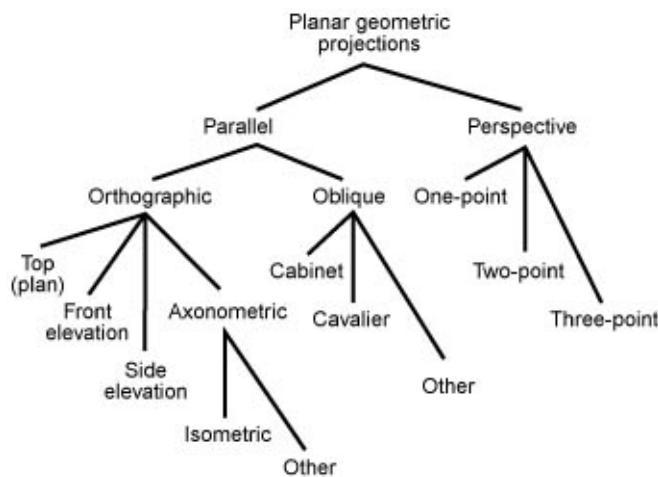
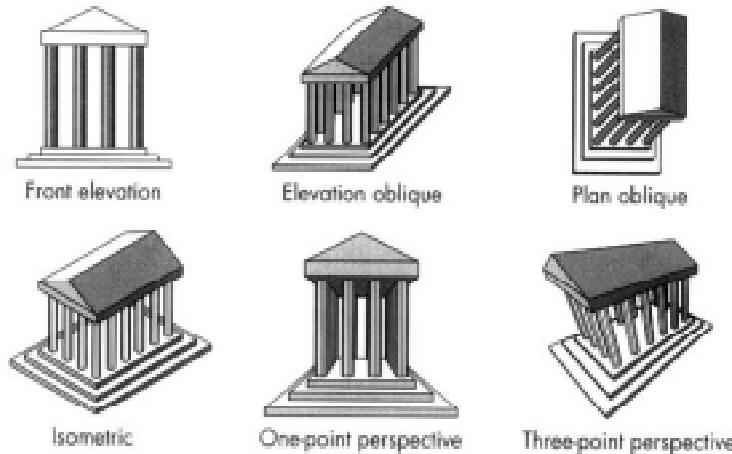


Figura 86 - Classificação das transformações de projeção planares.

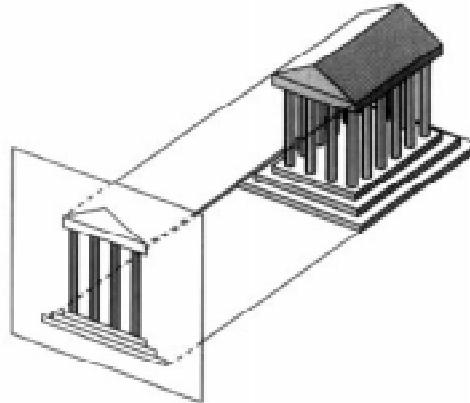
Na figura 87 podemos observar alguns exemplos de projeções da figura 86.



**Figura 87 - Diferentes projeções de uma mesmo objeto.**

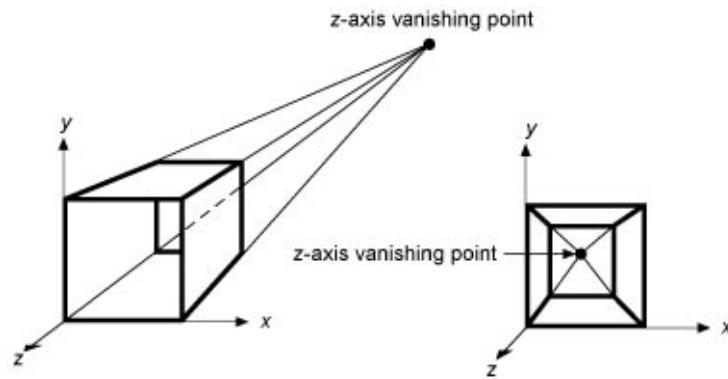
Dentro do curso estaremos interessados nas **projeções paralelas ortogonais** e nas **projeções perspectivas com um ponto de fuga**.

Uma projeção paralela é dita **ortogonal** sempre que os projetores forem perpendiculares ao plano de projeção, como na figura 88.



**Figura 88 – Projeção paralela ortogonal.**

No caso das projeções perspectivas com **um ponto de fuga**, a convergência dos projetores se dá em apenas um ponto, como na figura 89.

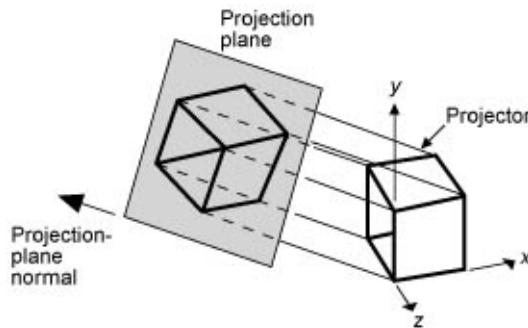


**Figura 89 - Projeção em perspectiva com um ponto de fuga (*vanishing point*).**

A seguir iremos analisar com mais precisão esses dois tipos de projeção, bem como definir as matrizes de transformação associadas.

### *Projeção Paralela Ortogonal*

Uma projeção paralela ortogonal apresenta como principal características os projetores serem paralelos entre si e perpendiculares ao plano de projeção (figura 90).



**Figura 90 - Formação da imagem em uma projeção paralela ortogonal.**

#### Vantagem:

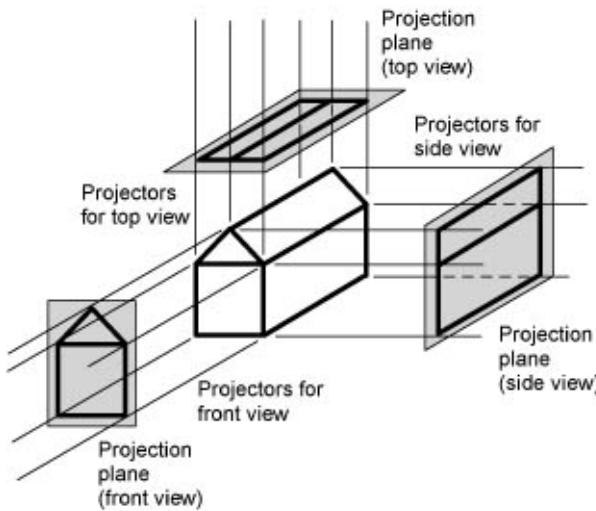
A projeção pode preservar ângulos e distâncias. Essa característica só se mostra válida para as partes do objeto (faces) paralelas ao plano de projeção.

#### Desvantagem:

As projeções dos objetos não sofrem alterações quando se movem segundo a direção de observação. Ou seja, dois objetos de mesma

dimensão, porém a distâncias diferentes do plano de projeção (segundo a direção de projeção) terão projeções idênticas.

Analizando as vantagens e desvantagens apresentadas, fica claro que as projeções paralelas ortogonais são recomendadas para aplicações que necessitam recuperar medidas a partir da projeção. Nesse tipo de aplicação é comum termos várias projeções, cada uma relacionada a uma face do objeto, tal como na figura 91.



**Figura 91 - Varias visões obtidas por projeções paralelas ortogonais de um mesmo objeto, uma para cada face principal.**

No entanto, não são adequadas para aplicações que necessitam de realismo.

### Matriz de Projeção Paralela Ortogonal

Para que possamos construir a matriz da projeção ortogonal, temos que definir um plano de projeção. Para nossa facilidade iremos considerar um plano paralelo àquele definido pelos eixos **xy**, ou seja, o plano de equação :

$$z_p = -d$$

A direção de projeção será perpendicular a esse plano (pela definição da projeção paralela ortogonal), ou seja, é paralela ao eixo **z**.

Analizando a forma da projeção podemos facilmente concluir que para um ponto qualquer **P** de coordenadas  $(x, y, z)$ , as coordenadas do ponto **P<sub>p</sub>** após a projeção serão :

$$x_p = x$$

$$\begin{aligned}y_p &= y \\z_p &= -d\end{aligned}$$

Todos os pontos projetados terão coordenada  $z_p = -d$ , pois estarão contidos no plano  $z = -d$ . Como o valor de  $\mathbf{d}$  é qualquer podemos considerar que  $\mathbf{d}=\mathbf{0}$ . Nesse caso a matriz  $\mathbf{M}_p$  associada a essa transformação (em coordenadas homogêneas) é :

$$\mathbf{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As principais características da matriz de transformação de projeção ortogonal são:

As equações são lineares.

A matriz é não inversível.

Linhas são preservadas.

### *Projeção Perspectiva*

Uma projeção perspectiva apresenta como principal características os projetores convergirem em um ponto, o **centro de projeção** (figura 92).

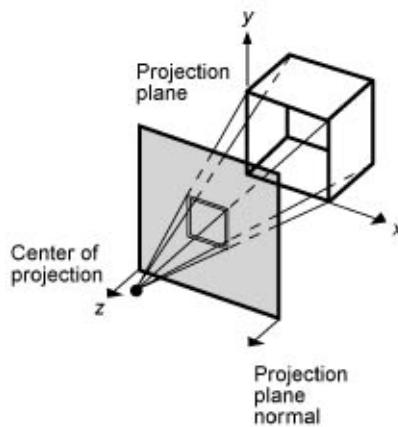


Figura 92 – Formação da imagem em uma projeção perspectiva.

A principal característica da projeção perspectiva é considerar a relação entre o tamanho da projeção de um objeto proporcional a distância objeto / observador.

Vantagem:

Aparência realista da projeção.

Desvantagem:

Não preserva distâncias.

### Matriz de Projeção Perspectiva

Para determinarmos a forma da matriz de projeção vamos analisar a figura 93.

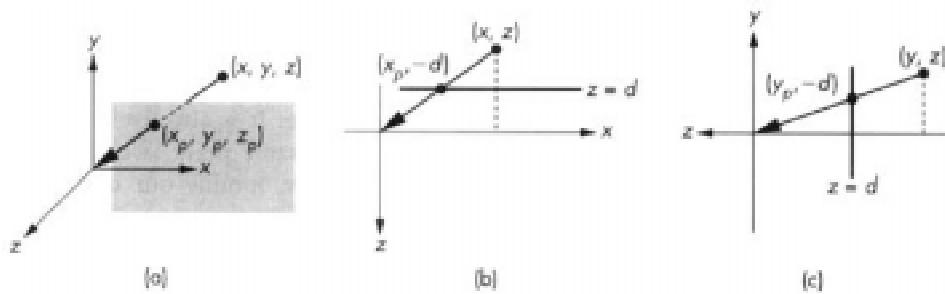


Figura 93 - Cálculo das coordenadas projetadas do ponto  $P$ .

Consideremos um ponto  $P$  de coordenadas  $(x, y, z)$  descritas no sistema cartesiano. Queremos calcular a sua projeção em um plano paralelo ao plano definido pelos eixos  $x$  e  $y$ , a partir de um centro de projeção localizado na origem do sistema de coordenadas.

A equação que descreve o plano de projeção é :

$$z = d$$

O ponto  $P_p$ , projeção de  $P$ , terá coordenadas  $(x_p, y_p, d)$ . Analisando as vistas ao longo dos eixos  $y$  e  $x$ , respectivamente, calcular os valores de  $x_p$  e  $y_p$  com base nas relações obtidas por semelhança de triângulos. Temos :

$$x_p = \frac{x}{\sqrt{z/d}}$$

$$y_p = \frac{y}{\sqrt{z/d}}$$

Em coordenadas homogêneas temos :

$$P_p = \begin{bmatrix} x \\ \cancel{z/d} \\ y \\ \cancel{z/d} \\ -d \\ 1 \end{bmatrix}$$

Matriz de transformação projetiva perspectiva tem a forma :

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

Uma vez aplicada essa transformação a um ponto  $P(x, y, z)$  temos :

$$Q = M.P = \begin{bmatrix} x \\ y \\ -z \\ \cancel{z/d} \end{bmatrix}$$

Dividindo todas as coordenadas por  $\cancel{z/d}$  temos :

$$Q = \begin{bmatrix} \frac{x}{\cancel{z/d}} \\ \frac{y}{\cancel{z/d}} \\ -d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

As principais características da matriz de transformação de projeção ortogonal são:

As equações não são lineares.

A matriz é não inversível.

Linhas são preservadas.

É interessante salientar que as transformações de projeção são formadas por *equações não lineares*. Essa característica é que permite que a projeção

gerada seja proporcional (inversamente) a distância do objeto ao plano de projeção.

### *Projeção Perspectiva x Projeção Paralela*

A partir da definição das transformações de projeção ortogonais e perspectiva podemos traçar um comparativo entre as duas.

#### Perspectiva

Projetores se encontram no *Centro de Projeção*;  
Preserva linhas;  
Não preserva ângulos;  
Não preserva medidas;  
Equações não lineares;  
Não inversível;  
Mais realista.

#### Paralela

Projetores se encontram no infinito  $\Rightarrow$  *Direção de Projeção*;  
Preserva linhas;  
Pode preservar angulos;  
Pode preservas medidas.  
Equações lineares;  
Não inversível;  
Resultado pouco realista.

Cabe nesse momento ressaltar novamente a importância de se utilizar um sistema de coordenadas próprio para a camera. Ao definirmos as matrizes de transformações consideramos o posicionamento da camera e do plano de projeção em condições bastante “confortáveis” do ponto de vista matemático. Ou seja, a matriz gerada é bastante simples e seu cálculo é facilmente processado.

Dessa forma voltamos a caracterizar as vantagens de se definir um sistema de coordenadas próprio associado a camera virtual.

## Aplicação no OpenGL

Como nos capítulos anteriores, apresentaremos agora uma aplicação mostrando como a **OpenGL** implementa os conceitos de sistemas de visualização e projeções.

O exemplo é bastante simples e apresenta a visualização de um cubo. Através da interface é possível não só alterar todos os parâmetros de

posicionamento da camera, como também definir o tipo de projeção que será utilizado (perspectiva ou paralela). A interface dessa aplicação pode ser vista na figura 94.

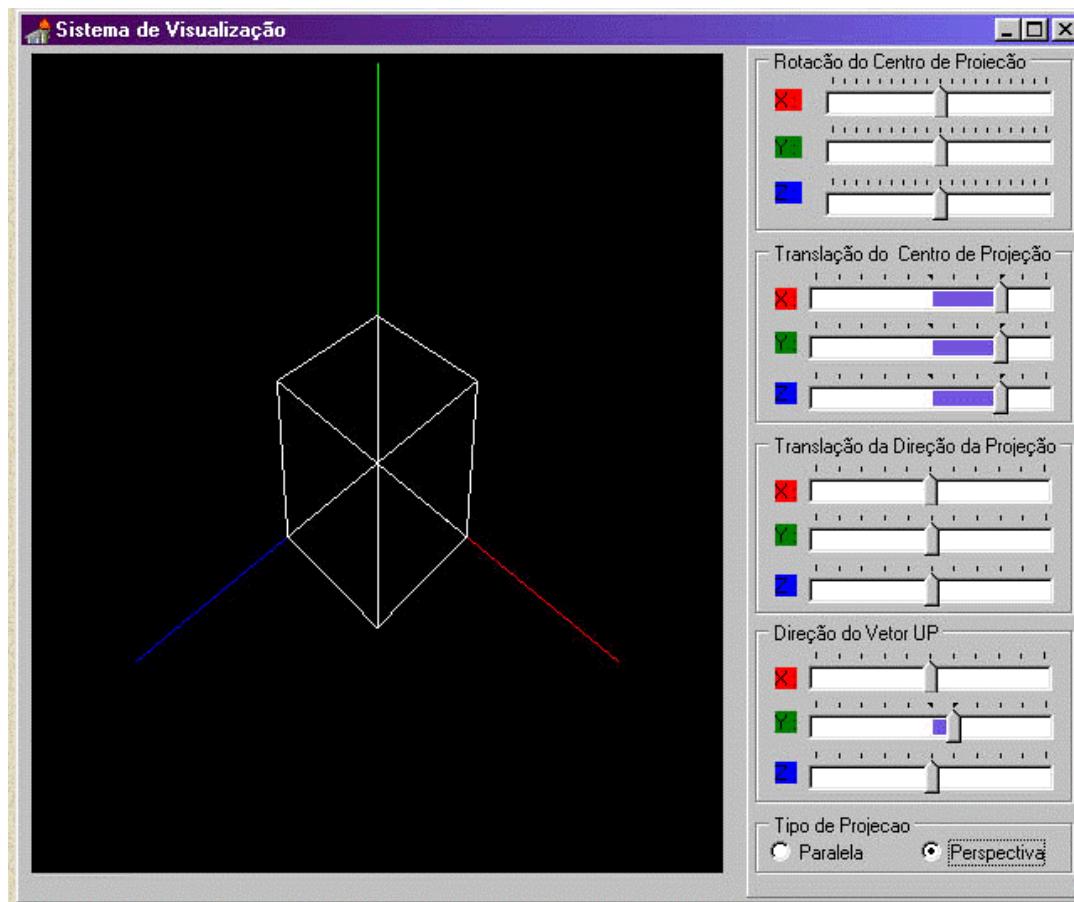


Figura 94 - Tela da aplicação exemplo. À esquerda a janela de visualização e à direita aos componentes de interface.

A camera virtual da ***OpenGL*** é posicionada, por *default*, na origem dos sistemas de coordenadas global, com a direção de observação coincidente com o eixo ***z***, na direção negativa. A orientação da camera é paralela ao eixo ***Y*** positivo (como na figura 95).

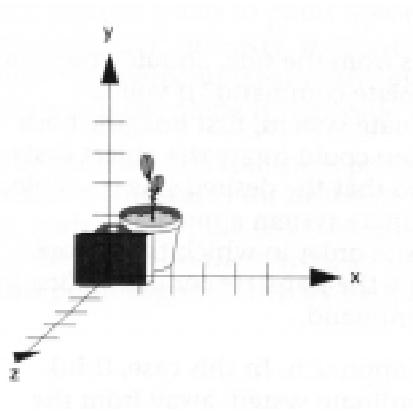


Figura 95 - Posicionamento *default* da camera virtual na *OpenGL*.

Essa posição pode ser alterada de forma direta ou via transformações geométricas. Na primeira hipótese – utilizada parcialmente dentro do programa – utilizamos a função ***gluLookAt***. Ela recebe como parâmetros dois pontos no espaço 3D (respectivamente o centro de projeção (*eye*) e o ponto focal (*at*), que juntos definem o vetor *direção de observação*) e um vetor (*up*) (que determina a orientação da camera, associado ao ângulo de torção da camera). A figura 95 mostra esses parâmetros.

Na aplicação, a camera é inicialmente posicionada com centro de projeção no ponto (3,3,3) e com foco na origem (0,0,0). O vetor *up* é posicionado paralelo ao eixo ***Y*** positivo (0,1,0). Isso é feito através da inicialização das variáveis *PosCPCameraX*, *PosCPCameraY*, *PosCPCameraZ*, *PosDPCameraX*, *PosDPCameraY*, *PosDPCameraZ*, *VetorUpX*, *VetorUpY*, *VetorUpZ*, respectivamente, no evento *FormCreate*.

Outra possibilidade é aplicar transformações geométrica a camera. Utilizamos essa abordagem para rodar a camera em torno do ponto focal. Os ângulos de rotação em cada direção, ***x***, ***y***, e ***z***, são definidos, respectivamente, pelas variáveis *RotacaoX*, *RotacaoY* e *RotacaoZ*.

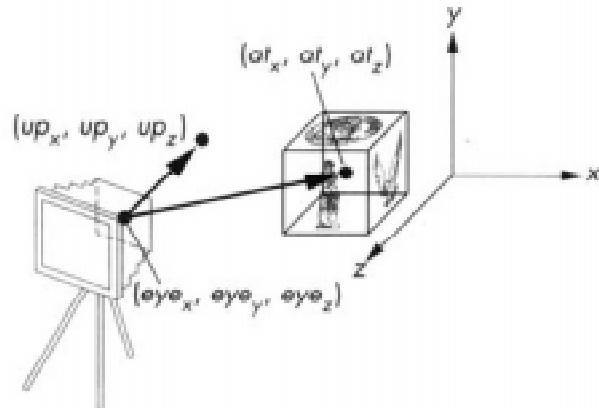


Figura 96 – Interpretação geométrica dos parâmetros da função *gluLookAt*.

Relembrando : a ***OpenGL*** trabalha com três pilhas de transformações. Uma delas está vinculada ao sistema de coordenadas global (*GL\_MODELVIEW*), conforme vimos no capítulo anterior. Outra pilha está vinculado ao sistema de coordenadas da camera virtual – definido pela constante *GL\_PROJECTION*. Dessa forma quando queremos aplicar transformações ao sistema de coordenadas da camera devemos direcionar essas transformações para a pilha correta. Isso é feito pela função ***glMatrixMode***.

O tipo de projeção a ser utilizada é definido a partir de duas funções : ***glOrtho*** para projeções ortográficas e ***glFrustum*** para projeções perspectivas.

No primeiro caso, a função ***glOrtho*** define uma projeção ortogonal e o volume de visão associado. Para tanto os parâmetros da função são valores limites para os planos *topo*, *fundo*, *esquerda*, *direita*, *perto* e *longe*, tal como na figura 97.

Para as projeções perspectivas a função ***glFrustum*** é utilizada. Ela define, da mesma forma que ***glOrtho***, o volume de visualização associado, através dos parâmetros : *topo*, *fundo*, *esquerda*, *direita*, *perto* e *longe*, como na figura 98.

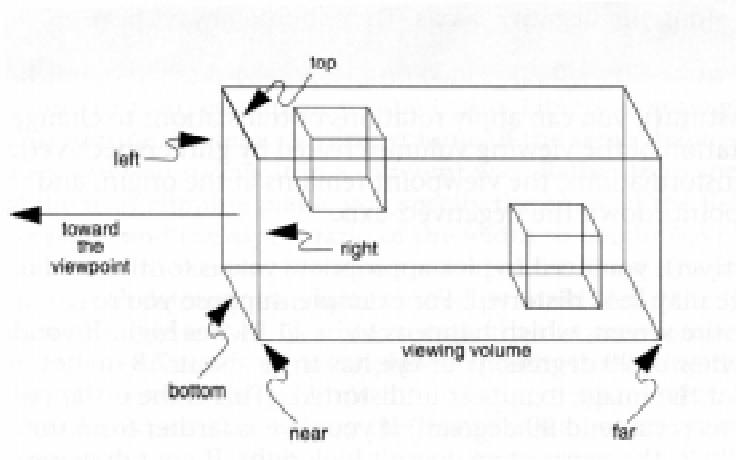


Figura 97 - Volume de visão associado a projeção ortogonal definida pela função `glOrtho`.

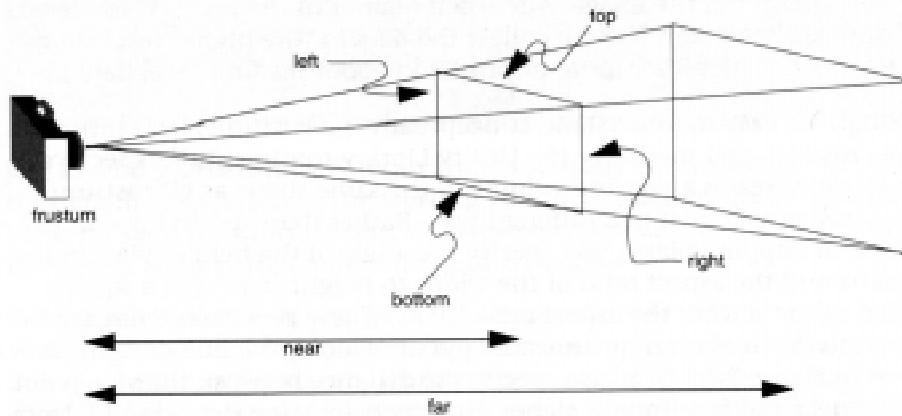


Figura 98 - Volume de visão associado a projeção perspectiva definida pela função `glFrustum`.

Essa construção para a definição do volume de visualização se mostra muito pouco intuitiva. Em geral utilizamos uma função da biblioteca `glu` : **`gluPerspective`**. Seu efeito final é equivalente a **`glFrustum`**, no entanto seus parâmetros são mais intuitivos : *fovy*, *aspecto*, *perto* e *longe*, como na figura 99. O valor do parâmetro *aspecto* é igual a a razão entre a *altura* e a *largura* da janela definida sobre o plano mais próximo. Com base no ângulo *fovy*, o *aspecto* e a distância desse plano ao observador é possível determinar os valores *topo*, *fundo*, *esquerda* e *direta* de **`glFrustum`**.

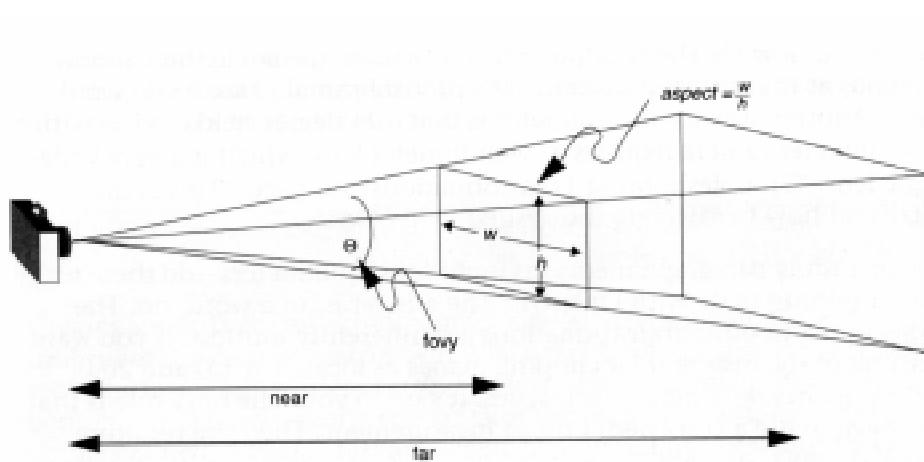


Figura 99 - Volume de visão associado a projeção perspectiva definida pela função *gluPerspective*.

A seguir temos o código fonte da aplicação.

```

unit SistVisual;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, OleCtrls, ComCtrls, StdCtrls, WAITEGLLib_TLB, OpenGL;

type
  TForm1 = class(TForm)
    Panel2: TPanel;
    GroupBox3: TGroupBox;
    labelx: TLabel;
    labely: TLabel;
    labelz: TLabel;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    TBRotacaoX: TTrackBar;
    TBRotacaoZ: TTrackBar;
    TBRotacaoY: TTrackBar;
    RGProjecao: TRadioGroup;
    TBPosCPCameraX: TTrackBar;
    TBPosCPCameraY: TTrackBar;
    TBPosCPCameraZ: TTrackBar;
    GLWindow: TWaiteGL;
    GroupBox2: TGroupBox;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    TBPosDPCameraX: TTrackBar;
    TBPosDPCameraY: TTrackBar;
    TBPosDPCameraZ: TTrackBar;
    GroupBox4: TGroupBox;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    TBVetorUpX: TTrackBar;
    TBVetorUpY: TTrackBar;
    TBVetorUpZ: TTrackBar;
    procedure GLWindowSetupRC(Sender: TObject);
    procedure GLWindowRender(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure TBRotacaoXChange(Sender: TObject);
    procedure TBRotacaoYChange(Sender: TObject);
    procedure TBRotacaoZChange(Sender: TObject);
    procedure BtnFimClick(Sender: TObject);
    procedure RGProjecaoClick(Sender: TObject);
  end;

```

```

procedure TBPosCPCameraXChange(Sender: TObject);
procedure TBPosCPCameraYChange(Sender: TObject);
procedure TBPosCPCameraZChange(Sender: TObject);
procedure TBPosDPCameraXChange(Sender: TObject);
procedure TBPosDPCameraYChange(Sender: TObject);
procedure TBPosDPCameraZChange(Sender: TObject);
procedure TBVetorUpXChange(Sender: TObject);
procedure TBVetorUpYChange(Sender: TObject);
procedure TBVetorUpZChange(Sender: TObject);
private
  { Private declarations }
public
  RotacaoX,
  RotacaoY,
  RotacaoZ : integer;
  PosCPCameraX,
  PosCPCameraY,
  PosCPCameraZ : real;
  PosDPCameraX,
  PosDPCameraY,
  PosDPCameraZ : real;
  VetorUpX,
  VetorUpY,
  VetorUpZ : real;
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//*****
//*****procedure TForm1.GLWindowSetupRC(Sender: TObject);
begin
  glClearColor(0.0, 0.0, 0.0, 1.0);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(45.0 , width/height , 1.0 , 10.0);
  gluLookAt(PosCPCameraX, PosCPCameraY, PosCPCameraZ,
            PosDPCameraX, PosDPCameraY, PosDPCameraZ,
            VetorUpX , VetorUpY, VetorUpZ);

  glMatrixMode(GL_MODELVIEW);

```

```

glLoadIdentity();
glFlush();
end;

//*****
//*****

procedure TForm1.GLWindowRender(Sender: TObject);
begin
  glClear(GL_COLOR_BUFFER_BIT);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  // Decide se faz projeção perspectiva ou paralela de acordo com o
  // componente da interface

  if RGProjecao.ItemIndex = 0 then
    glOrtho(-2.0, 2.0, -2.0, 2.0, -10.0, 10.0)
  else
    gluPerspective(45.0, width/height, 0.01, 15.0);

  // Posiciona a camera no espaço com base nas variaveis de controle,
  // cujos valores são gerados a partir da interface

  gluLookAt(PosCPCameraX, PosCPCameraY, PosCPCameraZ,
            PosDPCameraX, PosDPCameraY, PosDPCameraZ,
            VetorUpX, VetorUpY, VetorUpZ);

  // Rotaciona a camera na direção de cada eixo coordenado

  glRotatef(-RotacaoX, 1.0, 0.0, 0.0);
  glRotatef(-RotacaoY, 0.0, 1.0, 0.0);
  glRotatef(-RotacaoZ, 0.0, 0.0, 1.0);

  // desenha os eixos coordenados e o cubo.

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glBegin(GL_LINES);
  glColor4f(1.0, 0.0, 0.0, 1.0);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(2.0, 0.0, 0.0);
  glColor4f(0.0, 1.0, 0.0, 1.0);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(0.0, 2.0, 0.0);
  glColor4f(0.0, 0.0, 1.0, 1.0);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(0.0, 0.0, 2.0);

```

```

glEnd();
	glColor4f(1.0, 1.0, 1.0, 1.0);
	GLWindow.auxWireCube(1.0);
	glFlush();
	GLWindow.SwapBuffers();
end;

//*****
//*****



procedure TForm1.FormActivate(Sender: TObject);
begin
  GLWindow.MakeCurrent();
end;

//*****
//*****



procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  GLWindow.MakeNotCurrent();
end;

//*****
//** inicializa as variaveis de controle da camera e os componentes da *
//** interface associados.
//*****



procedure TForm1.FormCreate(Sender: TObject);
begin
  RotacaoX := 0;
  RotacaoY := 0;
  RotacaoZ := 0;

  PosCPCameraX := 3.0;
  PosCPCameraY := 3.0;
  PosCPCameraZ := 3.0;

  TBPosCPCameraX.Position := round(PosCPCameraX * 10);
  TBPosCPCameraX.SelEnd := TBPosCPCameraX.Position;
  TBPosCPCameraY.Position := round(PosCPCameraY * 10);
  TBPosCPCameraY.SelEnd := TBPosCPCameraY.Position;
  TBPosCPCameraZ.Position := round(PosCPCameraZ * 10);
  TBPosCPCameraZ.SelEnd := TBPosCPCameraZ.Position;

  PosDPCameraX := 0.0;
  PosDPCameraY := 0.0;
  PosDPCameraZ := 0.0;

```

```

VetorUpX := 0.0;
VetorUpY := 1.0;
VetorUpZ := 0.0;

TBVetorUpX.Position := round(VetorUpX * 10);
TBVetorUpX.SelEnd := TBVetorUpX.Position;
TBVetorUpY.Position := round(VetorUpY * 10);
TBVetorUpY.SelEnd := TBVetorUpY.Position;
TBVetorUpZ.Position := round(VetorUpZ * 10);
TBVetorUpZ.SelEnd := TBVetorUpZ.Position;
end;

//*****
//*****
procedure TForm1.TBRotacaoXChange(Sender: TObject);
begin
  RotacaoX := TBRotacaoX.Position;
  TBRotacaoX.SelEnd := RotacaoX;
  GLWindow.Invalidate();
end;
//*****
//*****
procedure TForm1.TBRotacaoYChange(Sender: TObject);
begin
  RotacaoY := TBRotacaoY.Position;
  TBRotacaoY.SelEnd := RotacaoY;
  GLWindow.Invalidate();
end;
//*****
//*****
procedure TForm1.TBRotacaoZChange(Sender: TObject);
begin
  RotacaoZ := TBRotacaoZ.Position;
  TBRotacaoZ.SelEnd := RotacaoZ;
  GLWindow.Invalidate();
end;

//*****
//*****
procedure TForm1.BtnFimClick(Sender: TObject);
begin
  Form1.Close();
end;

//*****
//*****
procedure TForm1.RGProjecaoClick(Sender: TObject);
begin

```

```

GLWindow.Invalidate();
end;
//*****
//*****
procedure TForm1.TBPosCPCameraXChange(Sender: TObject);
begin
  PosCPCameraX := TBPosCPCameraX.Position / 10;
  TBPosCPCameraX.SelEnd := TBPosCPCameraX.Position;
  GLWindow.Invalidate();
end;

//*****
//*****
procedure TForm1.TBPosCPCameraYChange(Sender: TObject);
begin
  PosCPCameraY := TBPosCPCameraY.Position / 10;
  TBPosCPCameraY.SelEnd := TBPosCPCameraY.Position;
  GLWindow.Invalidate();
end;

//*****
//*****
procedure TForm1.TBPosCPCameraZChange(Sender: TObject);
begin
  PosCPCameraZ := TBPosCPCameraZ.Position / 10;
  TBPosCPCameraZ.SelEnd := TBPosCPCameraZ.Position;
  GLWindow.Invalidate();
end;

//*****
//*****
procedure TForm1.TBPosDPCameraXChange(Sender: TObject);
begin
  PosDPCameraX := TBPosDPCameraX.Position / 10;
  TBPosDPCameraX.SelEnd := TBPosDPCameraX.Position;
  GLWindow.Invalidate();
end;

//*****
//*****
procedure TForm1.TBPosDPCameraYChange(Sender: TObject);
begin
  PosDPCameraY := TBPosDPCameraY.Position / 10;

```

```
TBPosDPCameraY.SelEnd := TBPosDPCameraY.Position;
GLWindow.Invalidate();
end;

//*****
//*****

procedure TForm1.TBPosDPCameraZChange(Sender: TObject);
begin
  PosDPCameraZ      := TBPosDPCameraZ.Position / 10;
  TBPosDPCameraZ.SelEnd := TBPosDPCameraZ.Position;
  GLWindow.Invalidate();
end;

//*****
//*****



procedure TForm1.TBVetorUpXChange(Sender: TObject);
begin
  VetorUpX      := TBVetorUpX.Position / 10;
  TBVetorUpX.SelEnd := TBVetorUpX.Position;
  GLWindow.Invalidate();
end;

//*****
//*****



procedure TForm1.TBVetorUpYChange(Sender: TObject);
begin
  VetorUpY      := TBVetorUpY.Position / 10;
  TBVetorUpY.SelEnd := TBVetorUpY.Position;
  GLWindow.Invalidate();
end;

//*****
//*****



procedure TForm1.TBVetorUpZChange(Sender: TObject);
begin
  VetorUpZ      := TBVetorUpZ.Position / 10;
  TBVetorUpZ.SelEnd := TBVetorUpZ.Position;
  GLWindow.Invalidate();
end;

end.

end.
```

## Referência

***glOrtho (Gldouble esquerda, Gldouble direita, Gldouble fundo,  
Gldouble topo, Gldouble perto, Gldouble longe);***

Método WaiteGL :

***Ortho (float esquerda, float direita, float fundo, float topo,  
float perto, float longe);***

Descrição :

*Constrói a matriz de projeção paralela ortográfica e a multiplica pela matriz de projeção corrente. O volume de visão é definido a partir dos pontos (esquerda, fundo, -perto) e (direita, fundo, -perto). Veja a figura ???.*

---

***glOrtho2D (Gldouble esquerda, Gldouble direita, Gldouble fundo,  
Gldouble topo);***

Método WaiteGL :

***Ortho2D (float esquerda, float direita, float fundo, float topo);***

Descrição :

*Constrói a matriz de projeção paralela ortográfica e a multiplica pela matriz de projeção corrente. Nesse caso as coordenadas z relativas ao plano mais perto e longe são fixas em 1.0 e -1.0, respectivamente. Em geral essa função é utilizada para definir a projeção do objetos bidimensionais (onde z=0). Como exemplo observe o programa exemplo do capítulo 2.*

---

***glFrustum (Gldouble esquerda, Gldouble direita, Gldouble fundo, Gldouble topo, Gldouble perto, Gldouble longe);***

Método WaiteGL :

***Frustum (float esquerda, float direita, float fundo, float topo, float perto, float longe);***

Descrição :

*Constrói a matriz de projeção perspectiva e a multiplica pela matriz de projeção corrente. O volume de visão é definido a partir dos pontos (esquerda, fundo, -perto) e (direita, fundo, -perto). As distâncias perto e longe devem ser dadas em relação ao observador, e portanto são sempre positivas. Veja a figura ???.*

***gluPerspective (Gldouble fovy, Gldouble aspecto, Gldouble perto, Gldouble longe);***

Método WaiteGL :

***gluLoadIdentity (double fovy, double aspecto, double perto, double longe);***

Descrição :

*Constrói a matriz de uma projeção perspectiva simétrica e a multiplica pela matriz de projeção corrente. O parâmetro fovy define o angulo do campo de visão na direção do plano xz (deve ter valor entre 0 à 180). Os valores perto e longe definem a distância do centro de projeção aos planos mais próximo e mais distante, respectivamente, na direção negativa do eixo z. Veja a figura ???.*

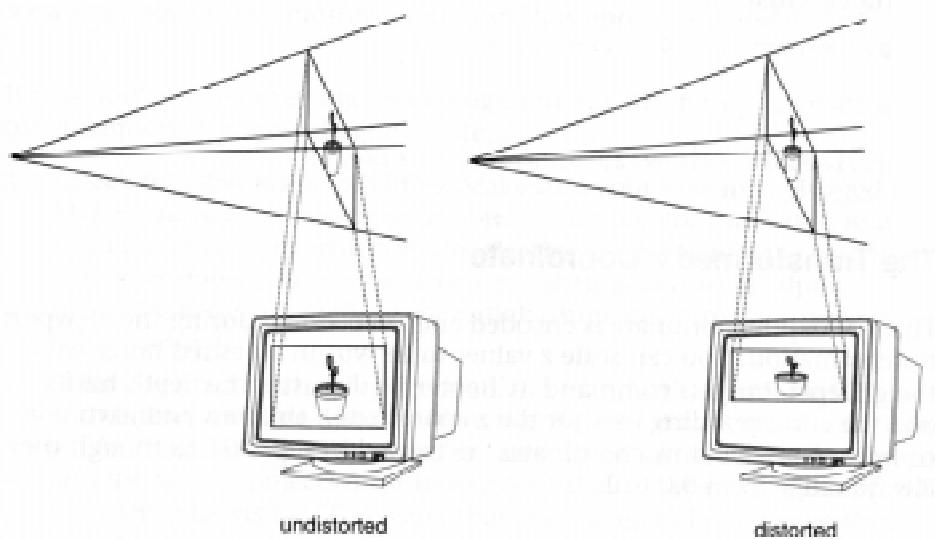
***glViewport (Glint x, Glint y, Glsizeli largura, Glsizeli altura);***

Método WaiteGL :

***Viewport (long x, long y, long largura, long altura);***

Descrição :

***Define a área retangular da janela onde a imagem final irá ser gerada. O ponto (x,y) define o canto inferior esquerdo da área retangular de largura x altura pontos. Por default a janela de viewport é definida a partir do ponto (0,0) com altura e largura iguais a da janela. O efeito obtido pode ser observado na figura 30.***



# CAPÍTULO V

## MODELO DE ILUMINAÇÃO

---

### Introdução

Até o momento estudamos como representar objetos tridimensionais, alterar sua forma e posicionamento e como gerar sua imagem bidimensional. Em nenhum momento, porém, tivemos preocupação de fazer com que esses objetos parecessem realistas. Eles eram sempre “desenhados” em **wireframe**, ou seja, suas faces eram representadas pelas retas definidas por suas arestas da malha poligonal que define sua superfície.

Agora, imagine que queremos dar ao objeto uma aparência sólida. Para isso vamos pintar a sua superfície com a mesma cor do objeto. O algoritmo a seguir apresenta como essa idéia seria implementada – considerando sempre que o objeto é representado como uma malha poligonal.

Algoritmo PintaObjeto3D;

para cada face da aproximação da superfície do objeto faça  
          pintar toda a face da cor do objeto;

Fim-para;

Fim.

A aplicação desse algoritmo simples em um objeto esférico branco sobre um fundo preto pode ser vista na figura 100b. Na verdade o objeto que estamos vendo na figura em nada se difere de um círculo branco sobre um fundo preto. Tal fato está relacionado ao funcionamento da nossa visão. Para que possamos perceber a sensação de que uma imagem representa algo tridimensional, é indispensável que essa imagem apresente variações de intensidade de luz que ocorrem na superfície do objeto<sup>4</sup>.

---

<sup>4</sup> Sendo mais rigoroso, a noção de volume que vemos se deve ao fato de possuirmos dois olhos. Isso permite que nosso cérebro aplique uma técnica de reconstrução que com base nas duas imagens bidimensionais geradas por nossos olhos, a terceira dimensão dos objetos seja obtida. É o que chamamos de **Visão Estéreo**.

Esse é o caso da figura 100a. Nela temos a mesma esfera, porém a variação de intensidade luminosa em sua superfície está evidenciada, permitindo que a “sensação” de volume da esfera se torne evidente.

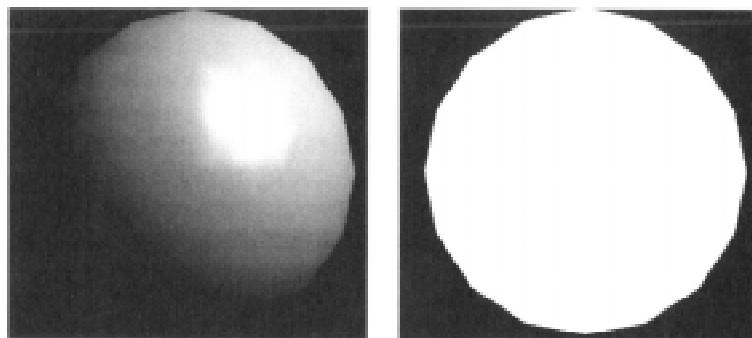


Figura 100- Duas esferas brancas sobre um fundo preto. (a) À esquerda com variação de intensidade da luz refletida. (b) À direita sem levar em conta variações de intensidade de luz.

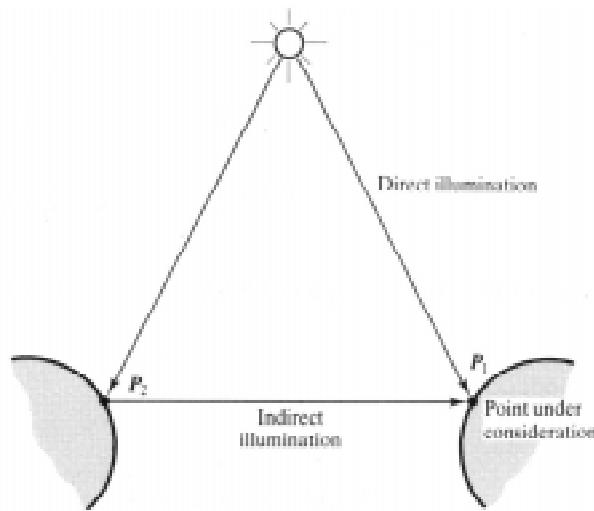
O objetivo desse capítulo é discutir um modelo de iluminação que permita que, dado um objeto descrito por uma malha poligonal, seja possível calcular a variação de intensidade de luz em sua superfície. Dessa forma essa intensidade pode ser levada em consideração quando do preenchimento de suas faces.

Para tanto precisamos compreender como o processo físico de interação luz/objeto/sistema visual acontece. Esse é o objetivo básico desse capítulo, apresentar o modelo no qual iremos nos basear para o cálculo da iluminação de um objeto.

## Modelo de Iluminação

A base para a construção de imagens realistas (no sentido de representarem de forma adequada a noção de volume de um objeto) é a análise a interação fonte de luz / objeto. Podemos caracterizar essa interação de duas formas. A primeira se dá quando um objeto recebe luz diretamente de uma fonte luminosa. Tomando por base a figura 101 é o que ocorre nos pontos  $P_1$  e  $P_2$ . Os raios de luz que emanam da fonte, percorrem o espaço e atingem de forma direta o ponto  $P_1$  e  $P_2$ . Nesses casos estamos diante do que chamamos de **iluminação direta**.

Ao atingir o ponto  $P_2$  parte da energia luminosa pode ser refletida e atingir novamente o ponto  $P_1$ . Esse raio de luz que não parte de uma fonte luminosa mais sim de outro objeto da cena faz parte do que denominamos **iluminação indireta**. Em um mundo real os objetos estão sujeitos sempre aos dois tipos de iluminação.



**Figura 101 - Exemplo de iluminação direta e indireta.**

A iluminação direta é relativamente simples de ser calculada, visto que depende unicamente da avaliação da fonte de luz e do objeto em questão. Já a luz indireta é muito mais complexa, pois é função do número de objetos da cena. Afinal todos os objetos podem contribuir indiretamente para a iluminação de um único ponto da superfície de um objeto.

O modelo de iluminação que apresentaremos a seguir irá levar em conta apenas a iluminação direta de uma cena. Modelos que pertencem a essa classe são denominados **modelos de iluminação local**. No capítulo VII apresentaremos um **modelo de iluminação global** (que irá levar em conta a iluminação indireta).

### *Modelo Físico*

Um modelo de iluminação geral deve considerar todos os tipos de fenômenos que podem ocorrer com um raio de luz ao entrar em contato com a superfície de um objeto. A figura 102 representa essa situação. As possibilidades são :

**Reflexão;**

Em última análise a reflexão de um raio de luz na superfície de um objeto é o que permite que um observador enxergue esse objeto.

**Transmissão;**

Alguns materiais – tais como vidro, cristal, acrílico, água, etc – permitem que um raio de luz os atravesse. Esse tipo de interação está sujeito ao efeito da refração – passagem da luz de um meio para outro.

### Absorção;

Cada material possui entre suas características a capacidade de reter determinados comprimentos de onda luminosa, convertendo essas freqüências em outro tipo de energia (tipicamente térmica, e em alguns casos elétrica). A percepção da cor de um material se dá em função da sua capacidade de absorção. Por exemplo, se um material tem cor azul significa que é capaz de absorver todas as freqüências da luz exceto aquela correspondente ao azul<sup>5</sup>. Portanto, todos os raios refletidos a partir desse material contém apenas ondas luminosas de freqüência equivalente a cor azul.

### Emissão.

Certos corpos quando aquecidos ou sujeitos a algum tipo de processo químico são capazes de emitir ondas luminosas.

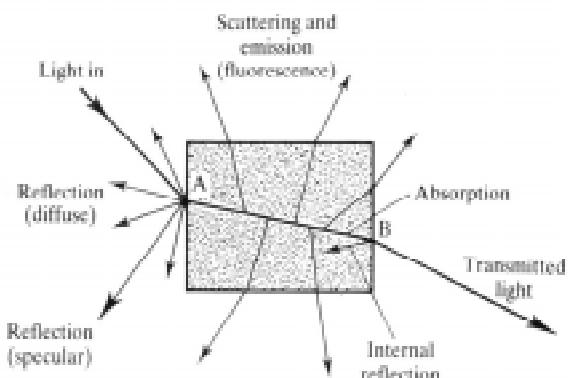


Figura 102 – Possíveis formas de interações entre um raio de luz e um objeto.

Um modelo que se proponha a representar todos esses tipos de interação torna-se bastante complexo, tanto do ponto de vista físico/matemático quanto do ponto de vista computacional. Como nosso objetivo principal é dar uma impressão “realista” as imagens de modelos sintéticos, podemos fazer algumas

---

<sup>5</sup> Considerando que estamos usando um azul “puro”, ou seja, que possui apenas uma determinada freqüência de onda correspondente ao azul.

suposições a respeito desses tipos de interação da luz e simplificar os requisitos do nosso modelo.

A primeira delas é quanto a emissão. Em geral a emissão, quando ocorre, é o tipo de interação predominante. Da mesma, poucos são os materiais capazes de, em condições normais, emitir ondas de luz. Resumindo, podemos simplificar nosso modelo considerando que os objetos ou são emissores, e nesse caso podem ser encarados apenas como tal (fontes luminosas) ou não irão possuir componente emissora (considerada desprezível).

Podemos considerar também que todos os nossos modelos serão constituídos de materiais opacos. Dessa forma não precisaremos, nesse momento, nos preocupar com a transmissão do material. Estaremos perdendo em poder de representação, mas ganhando em simplicidade / velocidade de processamento do modelo.

Por fim podemos associar a componente de absorção como uma função do material do objeto. Dessa forma, uma vez conhecido o material a componente absorção é constante para esse material.

A forma de interação restante é a reflexão. É com base nessa componente que somos capazes de enxergar os objetos, já que a imagem que se forma em nossa retina é resultado dos raios de luz refletidos pelos objetos.

Em resumo, iremos transformar o **modelo físico** em um **modelo de reflexão**. Historicamente essas simplificações fazem sentido, uma vez que os recursos computacionais necessários para se implementar modelos completos de iluminação eram proibitivos a maioria das aplicações a alguns anos atrás. Atualmente, com a popularização de placas aceleradoras 2D e 3D, diminuição do custo de memória e aumento na capacidade de processamento, muitas dessas barreiras foram quebradas. No entanto, esses modelos ainda se justificam quando queremos ter visualização interativa (em tempo real) de modelos complexos e com efeitos de iluminação.

### *Modelo de Reflexão Local*

Nosso objetivo é formular um modelo que permita calcular a intensidade da luz refletida em um ponto da superfície de um objeto. Para isso vamos considerar uma fonte de luz pontual, que emite raios de luz uniformemente em todas as direções. Cada raio de luz tem intensidade  $I_r$ .

O motivo pelo qual dizemos que esse é um *modelo de reflexão local* é que iremos considerar para efeitos de sua formulação, apenas contribuições resultantes de iluminação direta. Interações entre objetos não serão levadas em consideração – de forma precisa – pelo nosso modelo.

Porém, não deixaremos de representar a componente de iluminação indireta. Utilizaremos uma aproximação que irá considerar que essa contribuição é uniforme em toda a cena.

A seguir vamos apresentar os três componentes nos quais o nosso modelo de reflexão local estará baseado.

### Componente Ambiente

Conforme foi dito anteriormente um objeto pode ser iluminado direta ou indiretamente. Modelar a contribuição indireta de cada objeto da cena pode ser muito custoso. No entanto, uma forma simples de aproximar essa contribuição é considerá-la uma constante dentro da cena. Dessa forma iremos calcular uma valor de intensidade luminosa que chega a todos os pontos de todos os objetos da cena de forma idêntica. A essa componente damos o nome de **componente de luz ambiente**, já que está representando a luz dispersa no ambiente, resultado das reflexões entre objetos.

O valor dessa componente é dado pela equação :

$$\mathbf{I}_g = \mathbf{I}_a \cdot \mathbf{k}_a$$

onde  $\mathbf{I}_g$  é a *intensidade de luz “global” ou indireta* que chega em um dado ponto da superfície do objeto,  $\mathbf{I}_a$  é a *intensidade da luz ambiente* da cena e  $\mathbf{k}_a$  é um valor definido no intervalo  $[0,1]$  denominado **coeficiente de reflexão ambiente**. Essa constante visa atenuar a contribuição da luz ambiente em função do material do objeto. Sua determinação é empírica (visual) e não tem qualquer correspondência com propriedades físicas reais dos materiais.

A componente ambiente sozinha não tem muita utilidade, já que o seu efeito equivale a aplicação de uma cor constante em todas faces do objeto (como na figura 1b). Sua principal função é, em composição com as demais componentes da luz refletida, permitir que haja alguma intensidade luminosa em locais onde não há incidência direta de luz.

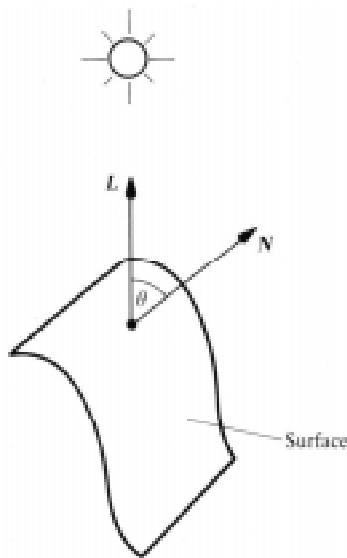
### Componente Difusa

Materiais foscos, sem brilho, como papel, giz, etc são denominados **refletores Lambertianos perfeitos**. A principal características desse tipo de material é que ele reflete a luz incidente em todas as direções. Dessa forma não há noção de reflexo.

Nesse caso, a intensidade de luz refletida em uma direção particular é função apenas da orientação da superfície no ponto a ser analisado.

Para definirmos a orientação da superfície em um dado ponto  $P$  utilizamos um vetor ortogonal ao plano tangente a superfície em  $P$ . Esse vetor denominamos **vetor normal  $N$** , como na figura 103.

Nessa mesma figura, temos um raio de luz proveniente de uma fonte pontual, que incide em na superfície de um objeto em um ponto. O vetor  $L$ , definido com base no ponto da superfície e na posição da fonte de luz, representa a direção do raio de luz incidente.



**Figura 103 – Parâmetros relacionados com o cálculo da componente difusa da luz refletida em um ponto de uma superfície.**

Um resultado básico da óptica geométrica é que a intensidade de luz refletida de forma difusa é proporcional ao angulo que o raio de luz incidente forma com a normal à superfície em um dado ponto. Esse resultado é expresso pela **lei de Lambert**:

$$I_d = I_i k_d \cos(\theta) \quad (1)$$

Onde :

$I_d$  é a intensidade da luz refletida,

$I_i$  é a intensidade da luz incidente,

$\theta$  é o ângulo que o vetor normal faz com a direção do raio de luz incidente e

$k_d$  é uma constante com valores no intervalo [0,1].

O valor de  $k_d$  é uma constante definida em função de cada tipo de material. A influência da constante  $k_d$  em uma imagem pode ser vista na figura 104.

Uma forma de aumentar a velocidade do cálculo da componente difusa é substituir o termo  $\cos(\theta)$  por um produto escalar dos vetores  $L$  e  $N$ . Como sabemos o produto escalar é dado pela expressão :

$$L \cdot N = \|L\| \cdot \|N\| \cdot \cos(\theta)$$

Como os dois vetores  $L$  e  $N$  podem ser considerados unitários<sup>6</sup> temos que :

$$\cos(\theta) = L \cdot N$$

Dessa forma podemos reescrever (1) como :

$$I_d = I_i k_d (L \cdot N) \quad (2)$$

O produto escalar pode ser feito através de operações de soma e produto, enquanto que a função  $\cos$  em geral é implementada através de uma expansão de uma série, que é bem mais custoso que o produto escalar.



Figura 104 – Esferas iluminadas a partir da aplicação da equação (2). Para todas as esferas o valor de  $I_i=1.0$ . Da esquerda para a direita os valores de  $k_d$  são respectivamente 0.4, 0.55, 0.7, 0.85 e 1.0. (Foley et al.)

Na figura 105 podemos ver o efeito da composição das duas componentes definidas até o momento : ambiente e difusa. Na esfera mais à esquerda, onde temos somente a componente difusa, podemos observar que a região posterior da esfera, que não recebe iluminação direta, fica completamente escura. Conforme a componente ambiente é aumentada essa região passa a receber iluminação.

Vale lembrar novamente que o nosso objetivo (por enquanto) não é construir um modelo rigoroso do ponto de vista da física envolvida na iluminação de um objeto ou cena. Queremos obter um modelo o mais simples possível, mas que seja visualmente aceitável, ou seja, o resultado final visual seja capaz de

<sup>6</sup> o que importa para o nosso modelo é a direção dos vetores e não seus módulos.

“enganar” o olho humano e dar a este a sensação de estar vendo algo próximo da realidade. E nesse sentido nosso modelo para a representação da componente difusa da luz refletida ainda apresenta uma falha.



**Figura 105 – Esferas iluminadas com as componentes difusa e ambiente.**  
Para todas as esferas temos  $I_g=I_d=1.0$  e  $k_d=0.4$ . Da esquerda para a direita temos os valores de  $k_a$  assumindo os valores 0.0, 0.15, 0.30, 0.45 e 0.60 respectivamente. (Foley et al.)

Suponha que dois objetos estejam posicionados de forma idêntica em relação a uma fonte de luz (suas orientações sejam as mesmas), mas suas distâncias a essa fonte sejam diferentes. Nesse caso a aplicação da equação (2) irá gerar a mesma intensidade de luz para os dois objetos. Por quê? Ora, em momento algum levamos em consideração o fato de que quanto mais distante um objeto da fonte de luz, menor será a intensidade da luz que ele recebe.

Para que essa característica possa ser levada em conta, basta acrescentarmos uma fator de atenuação que diminua a intensidade da luz incidente em função da distância da fonte ao objeto. Mas precisamente a atenuação da fonte é inversamente proporcional ao quadrado da distância entre a fonte e o objeto. Levando-se em conta esse fator temos que :

$$I_d = f_{att} \cdot I_i \cdot k_d \cdot (L \cdot N) \quad (3)$$

O fator de atenuação  $f_{att}$  é usualmente expresso por :

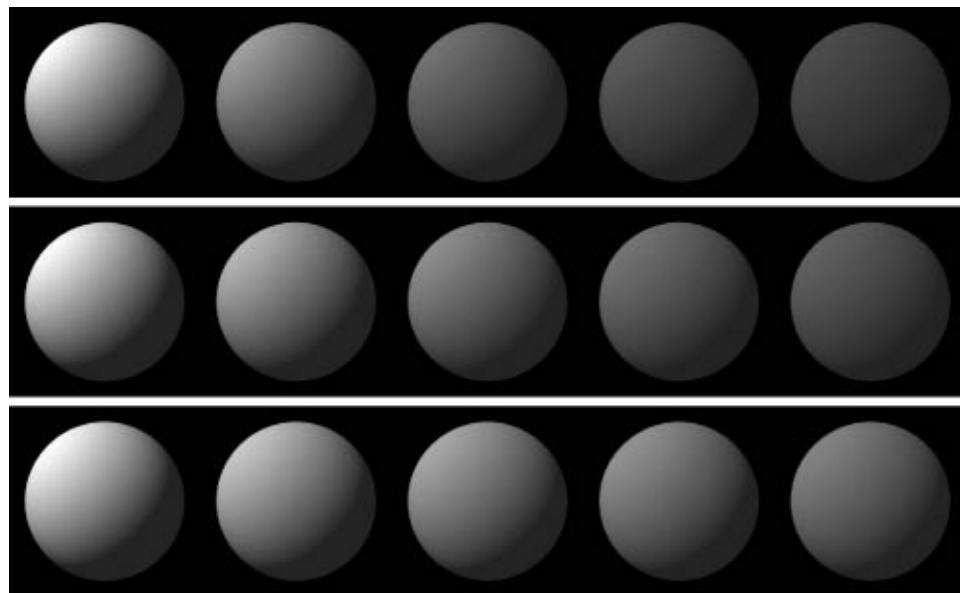
$$fatt = 1 / d^2$$

onde  $d$  é a distância da fonte de luz ao objeto. Essa forma de cálculo, apesar de ser simples e intuitiva, não apresenta bons resultados na prática. Para objetos próximos a fonte de luz a variação de  $f_{att}$  é muito rápida, enquanto que para objetos a grandes distâncias da fonte a variação de  $f_{att}$  é muito pequena. Uma versão alternativa para o fator de atenuação  $f_{att}$  é :

$$fatt = 1 / (c_1 + c_2 \cdot d + c_3 \cdot d^2)$$

onde  $c_1$ ,  $c_2$  e  $c_3$  são constantes definidas pelo usuário em função das características de atenuação que este quer associar a uma dada fonte de luz.

O efeito obtido em um objeto quando esse tipo de fator de atenuação é aplicado pode ser observado na figura 106. Para cada esfera em uma dada linha temos a distância  $d$  variando. Em cada linha temos constantes os valores de  $c_1$ ,  $c_2$  e  $c_3$ . Na primeira linha o fator de atenuação equivale a expressão  $1 / d^2$ , enquanto que na última linha esse mesmo fator pode ser reduzido a expressão  $1 / d^4$ , em função das escolhas das constantes  $c_1$ ,  $c_2$  e  $c_3$ .



**Figura 106** – Esferas iluminadas com as componentes ambiente e difusa da luz refletida. Para todas as esferas temos  $I_g=I_d=1.0$  e  $k_d=0.9$  e  $k_a=0.1$ . De cima para baixo temos os valores de  $(c_1, c_2, c_3)$ , que definem o fator de atenuação da componente difusa, assumindo os valores (em cada linha) :  $(0.0, 0.0, 1.0)$ ,  $(0.25, 0.25, 0.5)$  e  $(0.0, 1.0, 0.0)$  respectivamente. Em cada coluna temos os valores de  $d$  assumindo os valores :  $1.0, 1.375, 1.75, 2.125$  e  $2.5$  respectivamente. (Foley et al.)

### Componente Especular

Certos materiais ao invés de refletir a luz incidente em todas as direções – como as superfícies Lambertianas – refletem a luz em uma única direção. Esse tipo de superfície é chamada de “**espelho perfeito**”. A componente de luz refletida nesse caso é dita **especular**.

Superfícies que possuem componente especular, mas não são espelhos perfeitos refletem a luz segundo uma direção preferencial. Nessa direção temos a intensidade de luz refletida (de forma especular) máxima. Conforme nos afastamos dessa direção, a intensidade de luz refletida especular diminui.

Como essa componente define uma direção na qual os raios de luz refletidos seguem, devemos levar em consideração a posição do observador no cálculo da sua intensidade.

Considere a figura 107. Da óptica geométrica temos que o **angulo de incidência**  $\theta$  (formado pelos vetores  $L$  e  $N$ ), deve ser igual ao **angulo de reflexão** (formado pelos vetores  $R$  e  $N$ ). Portanto com esse resultado podemos calcular qual será a direção em que a luz refletida espelhada será máxima.

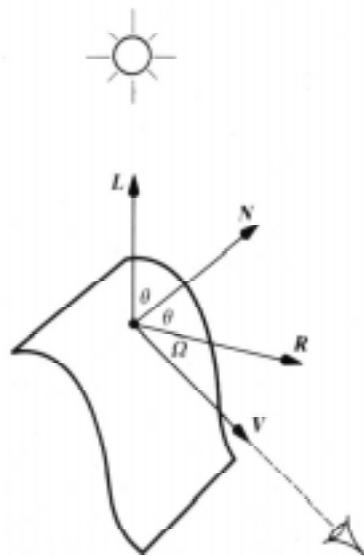


Figura 107 – Parâmetros relacionados com o cálculo da componente espelhada da luz refletida em um ponto de uma superfície.

No entanto o que precisamos saber é, em relação a essa direção máxima, onde se localiza o observador. Na figura 107 podemos ver que essa medida será dada pelo angulo  $\Omega$  (definido pelos vetores  $R$  e  $V$ ). Quanto maior o valor de  $\Omega$  menor será a intensidade da componente espelhada na direção  $V$  do observador.

A expressão matemática que representa esse modelo é dada por :

$$I_e = I_r k_e \cos^n(\Omega) \quad (4)$$

O valor de  $k_e$ , a semelhança de  $k_d$  no modelo de reflexão difusa, é uma constante definida em função de cada tipo de material. Podemos observar o efeito de sua variação na figura 108 – em cada linha o valor de  $k_e$  é constante.

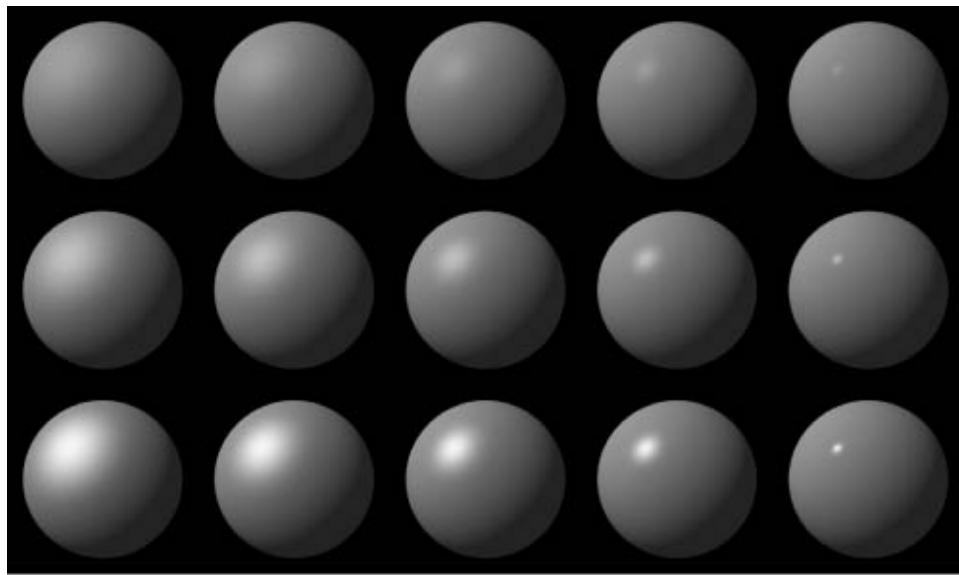


Figura 108 - Esferas iluminadas com as componentes ambiente, difusa e especular da luz refletida. Para todas as esferas temos  $I_g=I_d=I_e=1.0$  e  $k_d=0.45$  e  $k_a=0.1$ . Da esquerda para a direita temos os valores do expoente  $n$  da equação 4 assumindo os valores (em cada coluna) : 3.0, 5.0, 10.0 27.0 e 200.0 respectivamente. Em cada linha temos os valores de  $k_e$  assumindo os valores : 0.1, 0.25 e 0.5 respectivamente. (Foley et al.)

Tal como foi feito para a componente difusa, podemos substituir o termo  $\cos^n(\Omega)$  pelo produto escalar dos vetores  $\mathbf{R}$  e  $\mathbf{V}$ . Dessa forma a expressão da luz refletida pode ser escrita como :

$$\mathbf{I}_e = \mathbf{I}_r \mathbf{k}_e \cdot (\mathbf{R} \cdot \mathbf{V})^n \quad (5)$$

A função do expoente  $n$  associado ao  $\cos(\Omega)$  é produzir o efeito de atenuação da intensidade da luz refletida. Analisando o comportamento da função  $\cos$  em relação ao expoente  $n$  temos os gráficos da figura 109. Podemos observar que quanto maior o expoente mais rapidamente o valor da função tende a zero.

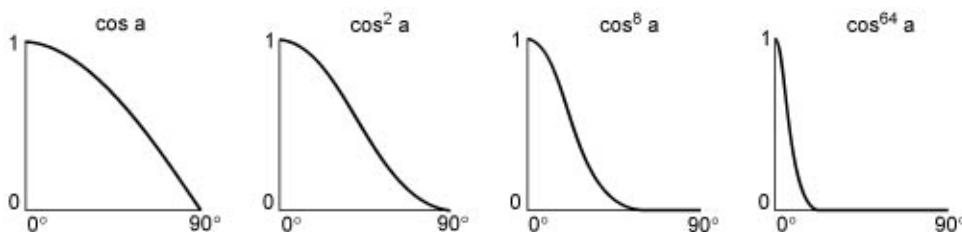


Figura 109 – Variação da função  $\cos^n(a)$ , no intervalo  $[0^\circ, 90^\circ]$ , em relação ao valor do expoente  $n$ .

O efeito que esse expoente produz na avaliação da componente refletida especular pode ser visto na figura 110. Quanto maior o expoente mais rápido a intensidade da luz refletida especular diminui, conforme o observador se afasta da direção de reflexão especular máxima. Na figura 110a temos um valor de  $n$  maior que na figura 110b.

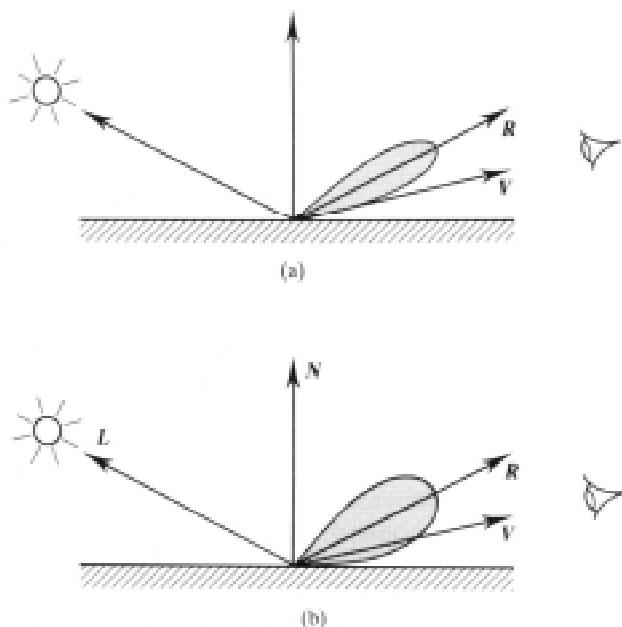


Figura 110 – Efeito da variação do expoente  $n$  no termo  $\cos(\Omega)$  da expressão (3).

### Composição da Componentes

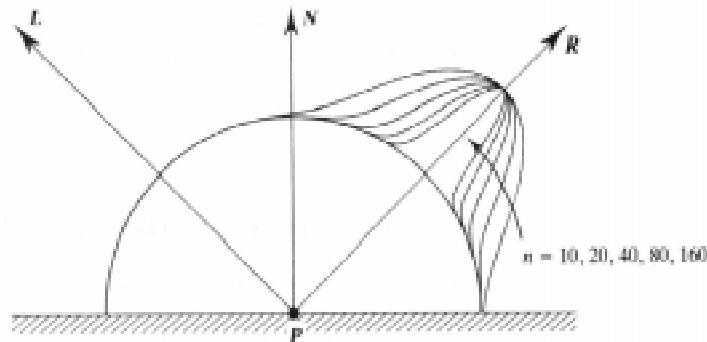
Uma vez definidos os componentes do nosso modelo de reflexão, temos agora que compo-los. Dessa forma teremos um modelo geral para o cálculo da intensidade da luz refletida. Esse modelo composto foi primeiro proposto por *Phong* e por isso é usualmente denominado **modelo de reflexão de Phong**.

Dada uma fonte de luz de intensidade  $I_i$  o valor da intensidade refletida  $I_r$  será a dado pela composição das componentes difusa, especular e ambiente, ou seja :

$$I_r = I_g + I_d + I_e = I_a \cdot k_a + I_i (k_d \cos(\theta) + k_e \cos^n(\Omega)) \quad (6)$$

O gráfico da função que representa a distribuição da intensidade de luz refletida a partir de um ponto  $P$ , dado um raio de luz incidente na direção  $L$ , pode ser visto na figura 111. Nesse gráfico temos de forma clara a composição da

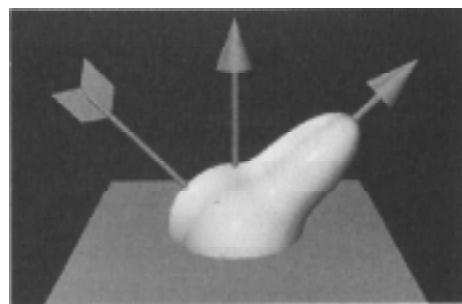
reflexão difusa com a especular. Quanto mais afastado um observador estiver da direção de reflexão especular máxima maior a tendência a componente difusa ser predominante. Nessa figura podemos ver novamente o efeito do expoente  $n$  na variação dessa função.



**Figura 111 –** Função de distribuição da intensidade da luz refletida visão de um corte bidimensional.

A participação da luz ambiente só faz com que o gráfico seja “expandido”, já que sua contribuição é constante em todas as direções em todos os pontos.

Na figura 112 podemos observar essa mesma função vista em sua forma tridimensional.



**Figura 112 –** Visualização da função de distribuição luminosa em 3D.

# CAPÍTULO VI

## ALGORITMOS DE COLORAÇÃO

---

A partir do modelo de reflexão apresentado no capítulo anterior temos meios de calcular a intensidade luminosa em um ponto qualquer da superfície de um objeto. No entanto, para que esse cálculo possa ser processado em um computador, precisamos especificá-los sob a forma de algoritmos.

A primeira consideração a ser feita sobre os algoritmos diz respeito aos seus dados de entrada e saída. Como entrada os algoritmos devem conhecer :

Dados da fonte de luz – posição e intensidade;

Dados do observador – posição;

Dados do objeto – forma e material.

No item (iii), os dados do objeto devem levar em consideração a forma com que ele é representado, ou seja, como o esquema de representação utilizado representa a geometria do objeto. No nosso caso convencionamos que os objetos são descritos por malhas poligonais. Nesse esquema é comum termos associado a cada vértice de cada face, além de suas coordenadas, o valor do vetor normal a superfície nesse ponto. É óbvio que essa informação é fundamental para o modelo matemático utilizado no cálculo da intensidade da luz refletida. A figura 113 mostra um modelo tradicionalmente utilizado em computação gráfica, o pote de chá (*teapot*) representado como uma malha poligonal. Nos vértices que definem suas faces podemos observar os vetores normais a superfície nesses pontos.

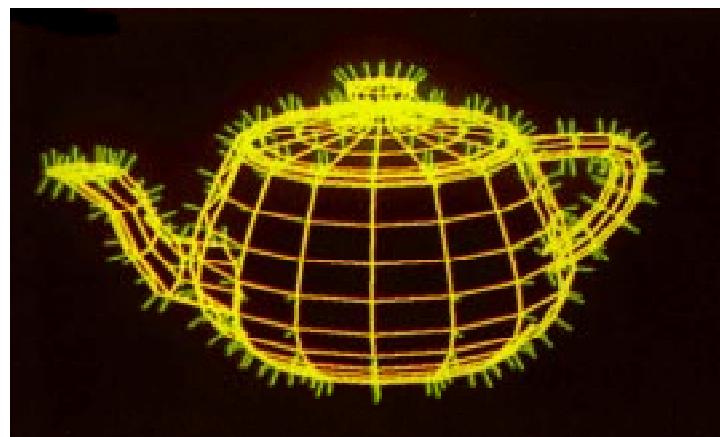


Figura 113 - Pote de chá em sua representação no esquema de malha poligonal. Em detalhe as normais definidas em cada um dos vértices da aproximação linear por partes da sua superfície.

Podemos construir um algoritmo bastante genérico que permita calcular a coloração (*rendering*) de um objeto. Esse algoritmo tem a seguinte especificação :

**Algoritmo ColoreObjeto:**

**Dados : Fonte de Luz, Observador, Objeto.**

para cada face do objeto faça

para cada ponto da face faça

calcular a normal no ponto;

calcular os ângulos  $\theta$  e  $\Omega$  formado pelos vetores  $L$ ,  $N$  e  $N$ ,  $R$ , respectivamente;

aplicar a equação de Phong calculando as intensidades refletidas no ponto em cada uma das componentes RGB;

pintar esse ponto com a cor correspondente;

fim-para;

fim-para:

fim.

Podemos ver que esse algoritmo pode ser bastante caro, do ponto de vista computacional, uma vez que para cada ponto da superfície do objeto todos os cálculos inerentes a equação de Phong devem ser calculados. Dessa forma temos um algoritmo que depende do número de faces do objeto e da resolução da imagem (quantos pontos – pixels – serão associados ao preenchimento da face).

O primeiro algoritmo que iremos abordar tem como principal objetivo reduzir ao máximo esses cálculos.

## Algoritmo Flat

Conforme vimos na seção anterior, o algoritmo geral tem um custo computacional relativamente alto. Se levarmos em consideração que em determinadas situações queremos interagir com objetos iluminados em tempo real, esse algoritmo se torna pouco prático. A geração da imagem pode demorar um tempo maior que o intervalo de tempo mínimo de espera requerido pela aplicação.

Para esses casos precisamos de um algoritmo rápido e que nos forneça uma aproximação razoável do resultado obtido pelo algoritmo geral.

A forma mais “radical” de reduzirmos o número de cálculos da equação de Phong é tornar a intensidade refletida uma constante por face. Em outras

palavras, considerar que todos os pontos de uma face possuem mesma intensidade refletida. Dessa forma se calcularmos esse valor em um ponto podemos propagá-lo para todos os demais pontos.

Dessa forma temos o seguinte algoritmo :

**Algoritmo ColoraçãoFlat;**

**Dados : Fonte de Luz, Observador, Objeto.**

para cada face do objeto faça

    calcular a média das normais em cada vértice da face;

    calcular os ângulos  $\theta$  e  $\Omega$  formado pelos vetores  $L$ ,  $N$  e  $N$ ,  $R$ , respectivamente;

    aplicar a equação de Phong calculando a intensidade refletida em um ponto (em cada uma das componentes RGB);

    pintar todo a face com a cor correspondente;

fim-para:

fim.

O nome desse algoritmo – **Flat** ou “**Plano**” – vem do fato das imagens dos objetos geradas todas as faces terem a aparência plana, como na figura 114. Tal fato é decorrente da nossa hipótese : todos os pontos da face possuem a mesma intensidade refletida.



Figura 114 – Pote de chá colorido com o algoritmo *Flat*.

Esse algoritmo é muito mais rápido que o algoritmo geral, não só pelo fato de fazer menos cálculos, mas também por prescindir de um passo de

discretização da face. Em outras palavras, não precisamos calcular a posição (pixel) de cada ponto da face. Ao invés disso podemos chamar uma primitiva do próprio sistema operacional (presumindo-se se tratar de um sistema baseado em interface gráfica tipo Windows) que desenha um polígono preenchido com uma cor sólida (constante em todos os seus pontos).

Dessa forma temos um algoritmo rápido o bastante para poder ser implementado como algoritmo básico de coloração na fase de edição/construção de objetos/cenas. O tempo gasto em seu cálculo é muito pequeno. Além disso várias otimizações podem ser feitas, de modo a acelerar o algoritmo ou tirar partido de recursos implementados em hardware (placas aceleradas com suporte 2D e/ou 3D).

Seu inconveniente é deixar bem a mostra a malha de polígonos que aproxima a superfície. Todos os modelos tem aparência “talha” ou facetada. Esse problema pode ser amenizado aumentando-se a resolução do modelo, ou seja, aumentando o número de polígonos que compõe a aproximação linear. Com isso a aparência facetada tenderá a diminuir. No entanto, o número de cálculos e os gastos de armazenamento – tanto da estrutura de dados em memória necessária para representar a malha como espaço em memória secundária para armazenamento do modelo – tenderão a aumentar proporcionalmente.

Para eliminar essa aparência facetada não temos outra opção senão diminuirmos nossa economia de cálculos, gastando um pouco mais para obter um resultado visualmente mais aceitável. É o que veremos no próximo algoritmo.

## Algoritmo de Gouraud

O principal problema do algoritmo *Flat* é tornar aparente a malha poligonal utilizada. Tal fato se deve a transição entre a cor (intensidade da luz refletida) ser feita de forma abrupta. Como cada face adjacente pode possuir intensidade constante a fronteira entre uma face e outra fica bem caracterizada.

A proposta do algoritmo de *Gouraud* é justamente suavizar a transição entre a coloração de faces adjacentes. A cor de uma face, portanto, não pode ser constante. Ela deve variar de modo que as fronteiras entre faces as cores possam ser “combinadas”. Dessa forma a aresta que marca a passagem de uma face para outra ficará “escondida” pela variação das cores.

***Algoritmo ColoreGouraud;******Dados : Fonte de Luz, Observador, Objeto.****para* cada face do objeto *face*

calcular a normal em cada vértice da face;

    calcular os ângulos  $\theta$  e  $\Omega$  formado pelos vetores  $L$ ,  $N$  e  $N$ ,  $R$ , respectivamente;    aplicar a equação de *Phong* calculando as intensidades refletidas em cada vértice para cada uma das componentes RGB;*para* cada ponto da face *face*

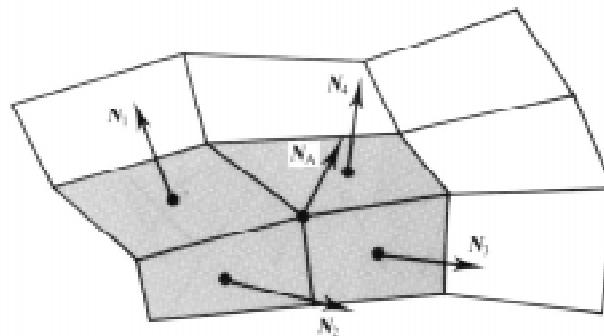
calcular a intensidade no ponto com base na interpolação linear das intensidades nos vértices;

pintar esse ponto com a cor correspondente;

*fim-para;**fim-para;**fim.*

A primeira etapa do algoritmo tem por finalidade calcular a intensidade da luz refletida em cada vértice da face. Esse cálculo é feito pela aplicação direta da equação (6). Para tanto é necessário o cálculo do vetor normal em cada vértice. Caso esse já faça parte do modelo poligonal, tanto melhor. Porém, caso esse vetor precise ser calculado pelo algoritmo explicitamente, é preciso tomar certos cuidados.

Considere, por exemplo, a malha poligonal da figura 115. O vértice **A**, ao está associado o vetor normal  $N_A$ , pertence às 4 faces sombreadas. A cada face podemos associar um único vetor normal, já que todas são planas. Ora, o vetor normal nada mais é do que uma forma de se representar a orientação de uma dada superfície em um ponto. No vértice em questão nenhuma das normais das faces a ele associadas conseguem representar a direção da superfície aproximada nesse ponto. A normal em **A** deve levar em consideração todas as orientações a sua volta. Isso é conseguido tomando-se a média das normais dessas faces como o valor do vetor normal nesse vértice.



**Figura 115 -Cálculo da normal em um dado vértice, através da média das normais em cada face adjacente.**

Uma vez que temos um valor de intensidade (cor) em cada vértice, aplicamos um processo de interpolação linear (“média ponderada”) para obter valores em pontos intermediários da face.

Esse processo nos garante, naturalmente, que, dadas duas faces adjacentes, os valores de cor próximos da aresta comum serão “harmoniosos” já que serão gerados a partir de vértices que possuem os mesmos valores de intensidade.

O resultado obtido aplicando-se esse algoritmo ao modelo do pote de chá pode ser visto na figura 116. Podemos notar que os “vincos” que apareciam nas arestas no algoritmo *Flat* desaparecem, dando lugar a uma superfície de aparência suave e contínua.



**Figura 116 -Pote de chá colorizado pelo algoritmo *Gouraud*.**

É claro que esse algoritmo é mais custoso, do ponto de vista computacional, que o algoritmo *Flat*. No entanto seu resultado visual é muito superior.

Ainda que a melhora na aparência do modelo tenha sido grande, ainda assim esse algoritmo apresenta problemas. Por tentar suavizar as transições entre as faces da intensidade de luz refletida, efeitos associados a variações abruptas dessa intensidade são eliminados. Um exemplo típico é o efeito de espelhamento que superfícies especulares possuem. Em materiais desse tipo, é comum a fonte de luz aparecer refletida na superfície do objeto como uma área de brilho mais intenso. No processo apresentado por *Gouraud* esse brilho também seria suavizado, eliminando assim a característica especular do objeto.

Portanto, em geral, as imagens geradas pelo algoritmo de *Gouraud* tem como “assinatura” a aparência fosca, típica de superfícies onde a componente difusa da luz refletida é predominante.

Um problema mais grave ainda surge quando as faces que aproximam o objeto tem uma configuração semelhante a da figura 117. Nesse caso a média das normais das superfícies em cada vértice produz normais com direções idênticas. Nesses casos, apesar da aproximação não ser plana, a aparência da imagem geradas para esse conjunto de faces o será. Esse efeito é o que chamamos de “planificação” da superfície.

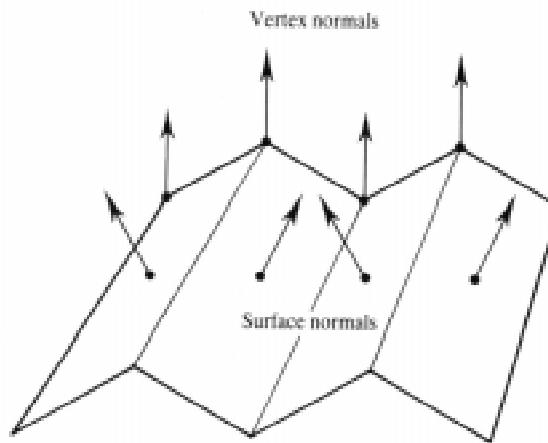


Figura 117 -Composição de faces / normais que geram o efeito de “planificação” quando da aplicação do algoritmo de *Gouraud*.

A seguir veremos mais um algoritmo que elimina esses problemas apresentados pelo algoritmo *Gouraud*.

## Algoritmo de Phong

O algoritmo de Phong é o que mais se aproxima do algoritmo geral apresentado no início dessa seção. Seu resultado é, portanto, melhor que o algoritmo Gouraud. Com esse algoritmo podemos representar objetos cujo material tem uma forte característica especular, como podemos observar na figura 118. A seguir temos a especificação do algoritmo.

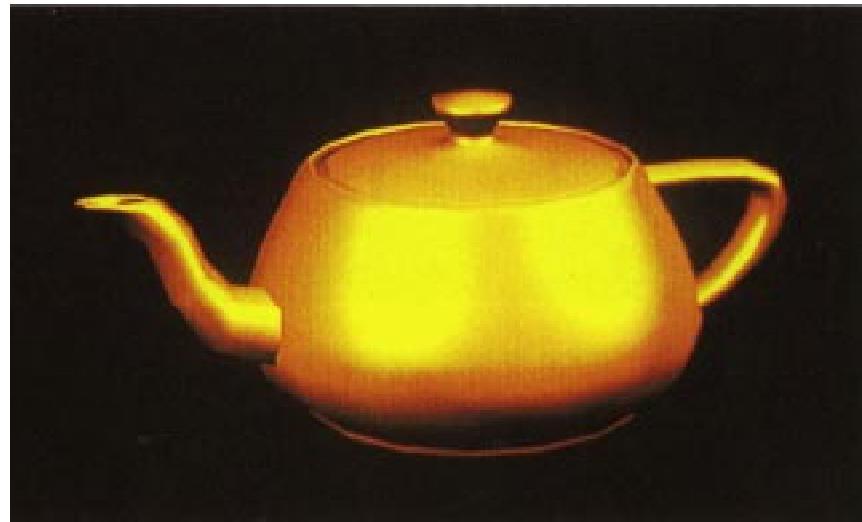


Figura 118 -Pote de chá colorizado pelo algoritmo *Phong*.

*Algoritmo ColorePhong:*

*Dados : Fonte de Luz, Observador, Objeto.*

*para* cada face do objeto *faca*

calcular a normal em cada vértice da face;

*para* cada ponto da face *faca*

calcular a normal no ponto com base na interpolação linear das normais nos vértices;

calcular os ângulos  $\theta$  e  $\Omega$  formado pelos vetores  $L$ ,  $N$  e  $N$ ,  $R$ , respectivamente;

aplicar a equação de *Phong* calculando a intensidade refletida no ponto para cada uma das componentes RGB;

pintar esse ponto com a cor correspondente;

*fim-para;*

*fim-para;*

*fim.*

A interpolação dos vetores normais permite que o comportamento da superfície original possa ser melhor representado pelo algoritmo de coloração. Na figura 119 podemos ver como a interpolação das normais produz um resultado mais real da variação de orientação da superfície original.

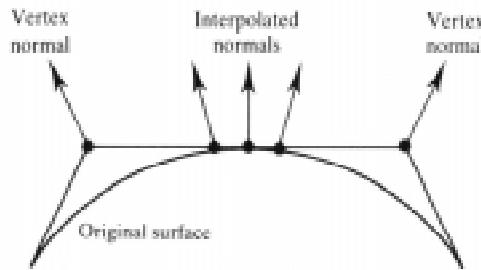


Figura 119 -Interpolação das normais dos vértices. As normais interpoladas representam, de forma mais precisa, a variação da orientação da superfície original.

O processo de interpolação das normais pode ser conseguido de maneira semelhante ao utilizado no algoritmo de Gouraud. A figura 120 mostra como esse processo é feito : para obter a normal em um ponto da face interpolamos as normais em duas aresta. Com isso temos duas normais nas arestas. Interpolando essas normais ao longo da reta que passa pelo ponto de interesse temos a normal no ponto.

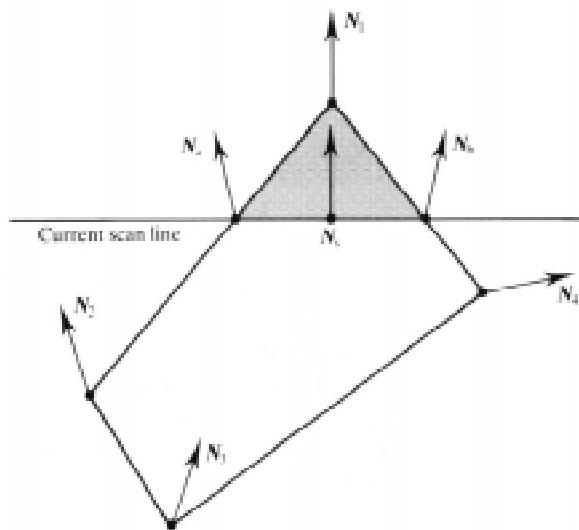


Figura 120 -Cálculo do vetor normal em um ponto de uma face, utilizando a interpolação dos vetores normais dos vértices.

## Aplicação OpenGL

Uma vez que já conhecemos o modelo básico de reflexão e algumas de suas implementações, passaremos a discutir como a biblioteca gráfica OpenGL trabalha com fontes de luz.

Para começar, temos que indicar ao ambiente da ***OpenGL*** que o cálculo da iluminação deve ser feito durante o processo de renderização. Para isso usamos o comando :

***glEnable(GL\_LIGHT);***

Para desabilitar esse cálculo usamos a função :

***glDisable(GL\_LIGHT);***

A partir daí podemos começar a definir nossas fontes de luz. A ***OpenGL*** suporta no mínimo 8 fontes de luz<sup>7</sup>. Essas fontes são identificadas por meio de constantes **GL\_LIGHT0**, ..., **GL\_LIGHT7**. As mesmas funções glEnable/glDisable serão utilizadas para ativar ou desativar, respectivamente, qualquer uma das 8 fontes de luz. Inicialmente, a única fonte cujos parâmetros *default* produzem efeitos na renderização é a **GL\_LIGHT0**. Todas as outras fontes, por *default* não produzem efeito algum mesmo quando habilitadas.

Feito isso todas as imagens geradas irão levar em conta a(s) fonte(s) de luz habilitadas, compondo a participação de cada uma na iluminação final de cada ponto.

Associado a uma fonte de luz temos uma série de parâmetros que definem como a fonte de luz irá se comportar. Uma única função da ***OpenGL***:

***glLight{if}{v} (GLenum luz, GLenum param, Type parm);***

Como em outras funções da ***OpenGL***, *i* ou *f* indicam, respectivamente, que a função vai receber um parâmetro inteiro ou ponto flutuante, enquanto que *v* é indicativo que um vetor contendo os parâmetros será passado. Essa última versão em geral é a mais utilizada.

<sup>7</sup> A constante **GL\_MAX\_LIGHTS** indica quantas fontes de luz o ambiente suporta. Para ter acesso a essa informação basta usar a função ***glGetIntegerv (GLenum pname, GLint \*params)***. O valor de **pname** deve ser **GL\_MAX\_LIGHTS**. O valor retornado em **params** indicará quantas fontes o seu ambiente suporta.

**Luz** indica para qual das fontes de luz estaremos definindo parâmetros. **Param** sinaliza qual dos parâmetros da fonte **Luz** estaremos alterando. Por fim **parm** é o parâmetro propriamente dito.

Todos os parâmetros de uma fonte de luz (assim como seus respectivos valores *default*) estão listados na seção de referência no final desse capítulo. A seguir comentaremos alguns.

O primeiro parâmetro é a posição da fonte de luz (GL\_POSITION). Essa posição é dada por um ponto no espaço homogêneo (4D). A quarta coordenada tem uma utilização especial. Caso seu valor seja 0 a fonte de luz é considerada uma **frente de luz direcional**, ou seja, está distante o suficiente para que os raios de luz sejam considerados paralelos. Portanto, nesse caso a posição da fonte tem por função definir um vetor direção, a partir do qual os raios de luz paralelos serão computados. Esse tipo de fonte é útil para simular fontes como o sol por exemplo. Caso contrário temos uma **frente de luz posicional**. Nesse caso os raios de luz são gerados radialmente a partir do ponto definido.

A intensidade da luz emitida pela fonte é definida em suas três componentes – **GL\_AMBIENT**, **GL\_DIFFUSE** e **GL\_SPECULAR** – através da intensidade dos quatro canais de cores RGBA.

No caso de fontes de luz posicionais, é possível criar o efeito de um spot de luz, ou seja, a luz não é dispersa em todas as direções de maneira radial em torno do ponto onde se localiza, mas sim dentro de um cone. O primeiro parâmetro que regula esse efeito define a direção para onde o spot será apontado – **GL\_SPOT\_DIRECTION** – expresso por um ponto no espaço homogêneo.

Outro parâmetro é o **GL\_SPOT\_CUTOFF**, que define o ângulo do cone definido pela fonte spot. Esse angulo deve ser definido dentro do intervalo  $[0^\circ, 90^\circ]$ . Caso esse valor seja  $180^\circ$ , temos um caso especial, indicando que esse efeito ficará desabilitado.

Como vimos no capítulo V, para aumentarmos o realismo do modelo de reflexão é usual introduzirmos um fator de atenuação no cálculo das componentes difusa e especular. Sua função é fazer com que a intensidade luminosa emitida por uma fonte sofra um decaimento em função de sua distância ao objeto. A **OpenGL** permite que termo seja regulado, provendo para isso três parâmetros : **GL\_CONSTANT\_ATTENUATION**, **GL\_LINEAR\_ATTENUATION** e **GL\_QUADRATIC\_ATTENUATION**, respectivamente os termos constante ( $c_1$ ), linear ( $c_2$ ) e quadrático ( $c_3$ ) da fórmula apresentada no capítulo V.

Uma vez que temos as fontes de luz criadas e ajustadas, temos que definir como os objetos irão “reagir” aos raios de luz emitidos por essas fontes. Ou seja, definir quais serão as características do material de cada objeto. Devemos lembrar que uma das simplificações feitas no modelo físico de iluminação foi

considerar que a componente absorção do modelo era constante em função do material do objeto.

Para isso a OpenGL possui a função :

***glMaterial {if} (GLenum face, GLenum pname, GLfloat param)***

Sua sintaxe é bastante semelhante a ***glLight***. Porém seu primeiro parâmetro indica não mais qual fonte de luz será utilizada mas sim como esse material será aplicado nas faces do modelo poligonal utilizado. Cada face (plano) pode ser vista no espaço 3D por dois lados : o “frontal” e o “dorsal”. Esses dois lados podem ser vistos, respectivamente, como a parte externa e interna da casca do objeto. É possível, portanto, definir materiais diferentes para o interior e exterior do objeto. Os valores desse parâmetro podem ser **GL\_FRONT**, **GL\_BACK** e **GL\_FRONT\_AND\_BACK**, sendo que nesse último o mesmo material será associado aos dois lados da face.

O material é caracterizado pelas suas componentes ambiente, difusa e especular (respectivamente **GL\_AMBIENT**, **GL\_DIFFUSE** e **GL\_SPECULAR** para valores de **pname**) indicando como a luz será refletida em termos de sua componente RGBA.

Conforme já vimos a componente especular da luz refletida pode produzir um “reflexo” da fonte de luz mais ou menos nítido em função do material do objeto. Materiais metálicos refletem com maior nitidez a fonte de luz que materiais plásticos, por exemplo. No modelo de reflexão especular definido o controle desse característica era feito através do expoente **n** aplicado ao termo  **$\cos(\Omega)$** . Esse termo pode ser controlado dentro da **OpenGL** através do parâmetro **GL\_SHININESS**. Seu valor varia na faixa de 0 à 128.

É possível ainda definir materiais emissores de luz (**GL\_EMISSION**) também especificando a sua componente RGBA<sup>8</sup>.

A aplicação a seguir tem por objetivo mostrar como essas funções podem ser utilizadas para gerar uma imagem com efeitos de iluminação. Nela são definidos um objeto (que pode ser alterado via interface) e uma única fonte de luz. A localização dessa última é fixa, acima do objeto.

Através da interface podemos alterar todos os parâmetros referentes a fonte de luz (relativos a função ***glLight***) e ao material do objeto (função ***glMaterial***).

<sup>8</sup> Vale ressaltar que materiais emissores não são considerados dentro do modelos da **OpenGL** como novas fontes de luz.

Para a construção do objeto nos valemos de funções pre-definidas dentro do componente WaiteGL, associadas a biblioteca aux. Essas funções geram automaticamente os vértices do objeto em questão (cilindro, cone, esfera, etc).

No entanto, conforme vimos nos algoritmos apresentados nesse capítulo é fundamental o conhecimento dos vetores normais a superfície do objeto em cada vértice. A biblioteca ***OpenGL*** ***não*** faz esse cálculo automaticamente. Cabe ao programador que está construindo o modelo fazer esse cálculo e informar a ***OpenGL*** o valor a ser utilizado. Para isso, temos a função :

***glNormal{bdfis}*** ( Type  $n_x$ ,  $n_y$ ,  $n_z$  )

Nela o vetor normal é definido em termos das suas componentes  $n_x$ ,  $n_y$  e  $n_z$ , para cada vértice.

No caso de nossa aplicação esses vetores foram calculados automaticamente, assim como os vértices.

## Referência

***glLight{if}{v} (GLenum luz, GLenum parmnome, Type parm);***

Descrição :

**Define as características da fonte de luz especificada por luz. O tipo de característica é definido por parmnome, e seu valor é especificado em parm. Os tipos de parmnome podem ser:**

<b>Nome</b>	<b>Valor Default</b>	<b>Descrição</b>
GL_POSITION	(0.0,0.0,1.0,1.0)	Define a posição da fonte de luz. O 4º parâmetro define se a fonte é direcional ou posicional.
GL_DIFFUSE	(1.0,1.0,1.0,1.0)	Define os valores RGBA da componente difusa da fonte de luz.
GL_SPECULAR	(1.0,1.0,1.0,1.0)	Define os valores RGBA da componente especular da fonte de luz
GL_AMBIENT	(0.0,0.0,0.0,1.0)	Define os valores RGBA da componente ambiente da fonte de luz
GL_SPOT_DIRECTION	(0.0,0.0,-1.0)	Define a direção da fonte de luz do tipo spot.
GL_SPOT_EXPONENT	0.0	Define o grau de decrescimento da intensidade de luz da fonte spot a medida que os raios de luz se distanciam do direção definido por GL_SPOT_DIRECTION.
GL_SPOT_CUTOFF	180.0	Defino o angulo de abertura da fonte de luz spot. Varia entre [0°, 90°], sendo que o valor 180° indica que a fonte não será spot.
GL_CONSTANT_ATTEN UATION	1.0	Define o termo constante da função de atenuação.
GL_LINEAR_ATTENUAT ION	0.0	Define o termo linear da função de atenuação.

***GLightModel{if}(GLenum parmname, Type parm);***

Descrição :

**Define as características do modelo global de iluminação. O tipo de característica é definido por parmname, e seu valor é especificado em parm. Os tipos de parmname podem ser:**

<b>Nome</b>	<b>Valor Default</b>	<b>Descrição</b>
GL_LIGHT_MODEL_AMBIENT	(0.2,0.2,0.2,1.0)	Define o valor da luz ambiente para toda a cena.
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0 ou GL_FALSE	Define como o angulo da componente especular será calculado.
GL_LIGHT_MODEL_TWO_SIDE	0.0 ou GL_FALSE	Define se a iluminação das faces do modelo será aplicada aos dois lados de cada face (GL_TRUE) ou só para o lado frontal (GL_FALSE)

***glMaterial{if} (GLenum face, GLenum parmname, GLenum parm);***

Descrição :

**Define as características do material associado a um objeto. O parâmetro face determina em que lado da face esse material será aplicado. Seus valores podem ser :**

<b>Nome</b>	<b>Descrição</b>
GL_FRONT	Aplica o material apenas no lado frontal das faces do objeto.
GL_BACK	Aplica o material apenas no lado dorsal das faces do objeto.
GL_FRONT_AND_BACK	Aplica o material nos dois lados das faces do objeto.

**O tipo de característica do material é definido por parmname, e seu valor é especificado em parm. Os tipos de parmname podem ser:**

<b>Nome</b>	<b>Valor Default</b>	<b>Descrição</b>
GL_AMBIENT	(0.2,0.2,0.2,1.0)	Define os valores RGBA da componente ambiente do material do objeto.
GL_DIFFUSE	(0.8,0.8,0.8,1.0)	Define os valores RGBA da componente difusa do material do objeto.
GL_SPECULAR	(0.0,0.0,0.0,1.0)	Define os valores RGBA da componente especular do material do objeto.
GL_AMBIENT_AND_DIFFUSE		Define os valores RGBA das componentes ambiente e difusa do material do objeto.
GL_SHININESS	0.0	Define o valor do expoente da componente especular. Varia na faixa entre [0, 128]
GL_EMISSION	(0.0,0.0,0.0,1.0)	Define os valores RGBA da componente emissora do material do objeto.

***glColorMaterial(Glenum face, Glenum modo);***

Descrição :

**Promove a associação entre a cor utilizada correntemente para definir uma das características do material do objeto. O parâmetro face determina em que lado da face esse material será aplicado. Seus valores podem ser :**

<b>Nome</b>	<b>Descrição</b>
GL_FRONT	Aplica a cor dos vértices apenas no lado frontal das faces do objeto.
GL_BACK	Aplica a cor dos vértices apenas no lado dorsal das faces do objeto.
GL_FRONT_AND_BACK	Aplica a cor dos vértices nos dois lados das faces do objeto.

**O parâmetro modo define qual a característica do material será vinculada a cor corrente. Seus valores podem ser :**

<b>Nome</b>	<b>Descrição</b>
GL_AMBIENT	Associa a cor corrente a componente ambiente do material do objeto.
GL_DIFFUSE	Associa a cor corrente a componente difusa do material do objeto.
GL_AMBIENT_AND_DIFFUSE	Associa a cor corrente a componente ambiente e difusa do material do objeto.
GL_SPECULAR	Associa a cor corrente a componente especular do material do objeto.
GL_EMISSION	Associa a cor corrente a componente emissão do material do objeto.

# CAPÍTULO VII

## REALISMO E MODELOS GLOBAIS

---

Os modelos locais de iluminação são ótimos no sentido de serem capazes de gerar imagens de qualidade razoável com um baixo custo computacional. No entanto, o grau de realismo desses modelos deixa muito a desejar. Pela sua própria definição, uma série de efeitos da luz não podem ser modelados. Entre eles :

- Sombra e penumbra;
- Iluminação indireta;
- Superfícies espelhadas;
- Transparência;
- Materiais heterogêneos.

A seguir veremos algumas técnicas que permitem “simular” alguns desses efeitos sem ter que incorporá-los efetivamente dentro do modelo.

### Modelos Locais

#### *Mapeamento de Texturas*

Os modelos locais permitem *renderizar* (aplicar cor aos pontos de um objeto em função da intensidade de luz) um objeto cujo material é homogêneo. Ou seja, materiais como mármore, granito, madeira, etc – heterogêneos em sua composição – não podem ser modelados de forma precisa.

A técnica de mapeamento de textura nos permite simular esses materiais. A idéia básica é “embrulhar” o objeto com uma imagem do material que se deseja simular. A figura 121 mostra uma imagem gerada com essa técnica.

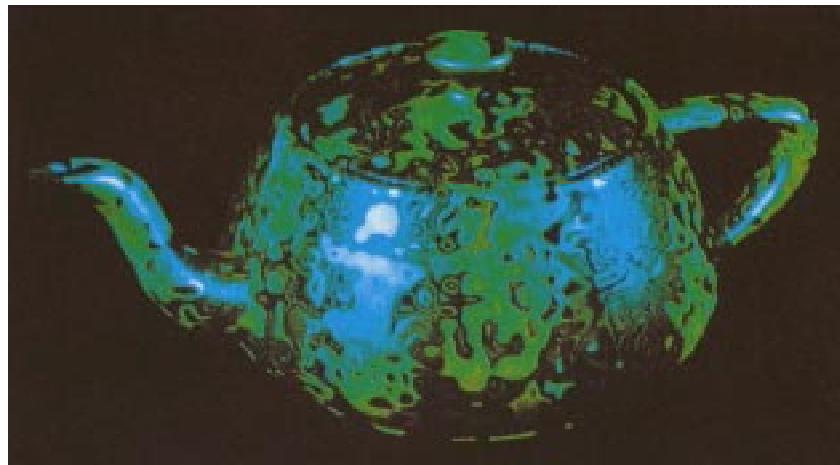


Figura 121 – Pote de chá com mapeamento de texturas.

Outra aplicação da técnica de mapeamento de textura é representar detalhes em objetos de uma cena, sem que esses precisem ser modelados e incorporados ao objeto. A figura 122 dá um exemplo desse caso. Para conseguir o efeito da visão das construções do terreno, não foi necessário modelar cada casa, prédio ou estrada. A partir de uma imagem aérea da cidade foi possível, através da técnica de mapeamento de texturas, gerar uma imagem com alto grau de realismo.



Figura 122 – Aplicação de mapeamento de textura na modelagem de um terreno.

A grande dificuldade dessa técnica é como construir formalmente esse “embrulho”. Em termos práticos, esse “embrulho” é uma função matemática que define uma relação entre pontos de uma imagem (textura) em pontos sobre a superfície do objeto. Fica claro que estamos tratando de uma transformação entre domínios distintos. Na figura 123 mostramos os três domínios envolvidos nesse processo, também denominados espaços. São eles :

**Espaço de textura** – em geral bidimensional, é o espaço onde a imagem é definida;

**Espaço do objeto** – tridimensional por natureza, é o espaço onde o objeto está definido;

**Espaço da imagem** – onde efetivamente a imagem da cena será gerada.

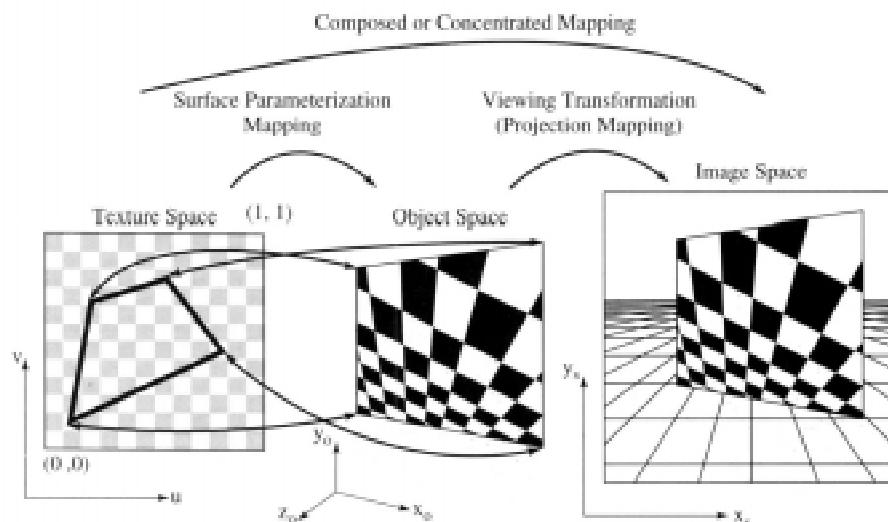


Figura 123 – Relacionamento entre os diversos espaços envolvidos no processo de mapeamento de texturas.

Uma vez definidas as funções de mapeamento, elas serão utilizadas dentro dos algoritmos de reflexão como base para decidir sobre que cor a composição das componentes da luz refletida será aplicada. Em outras palavras, para cada ponto será calculada uma intensidade de luz refletida, baseada na cor da imagem associada a textura, indicada pela função de mapeamento<sup>9</sup>.

---

<sup>9</sup> Lembrando que nos algoritmos de reflexão apresentados anteriormente a cor de um ponto sobre a superfície de um objeto era função do seu material.

Os algoritmos que incorporam essa técnica produzem objetos muito mais realistas que os modelos locais simples. No entanto, seu custo computacional aumenta significativamente, já que a cada ponto da superfície do objeto, além do cálculo da equação de Phong para o modelo de reflexão, a função de mapeamento também deverá ser aplicada.

A utilização de imagens para representar materiais possui suas limitações. Uma delas é o gasto de memória adicional, associado ao armazenamento da imagem associada ao material do objeto. Além disso, materiais naturais, como madeira, mármore, etc, possuem padrões que não se repetem de forma idêntica. Além disso esses materiais possuem uma característica de disposição espacial (3D) que nem sempre é reproduzida com sucesso pelo simples mapeamento da imagem do material na superfície do objeto.

Uma alternativa nesses casos é substituir a imagem da textura por um algoritmo que forneça, dado um ponto do espaço de textura, um valor de cor associado. Texturas definidas dessa forma são chamadas ***texturas procedurais***. A grande vantagem é que um só algoritmo, por exemplo, pode ser capaz de simular vários tipos de texturas, apenas variando alguns parâmetros de controle. Nesses casos ao invés de se ter que armazenar diversas imagens, um único algoritmo pode fazer todo o serviço.

Em contrapartida, o processamento desse tipo de textura, em geral, é mais caro e lento, já que o valor da textura em um ponto é definido algorítmicamente e não por uma função do tipo índice, como nas imagens (dados os valores  $(u, v)$ , coordenadas do espaço de textura, temos imediatamente o valor do ponto na imagem). Seus resultados, no entanto, tendem a ser mais realistas que a aplicação de texturas baseadas em imagens. Na figura 124 podemos observar os detalhes dos nós da madeira se propagando de forma consistente por em todas as faces.



Figura 124 – Aplicação de uma textura procedural para simulação de um bloco de madeira. Podemos observar que os nós circulares da face lateral esquerda podem ser acompanhados nas faces frontal e superior.

## Mapeamento de Ambiente (Environment Mapping)

Um caso particular de mapeamento de textura é aplicado quando queremos simular objetos espelhados, sem ter necessariamente que calcular as reflexões que ocorrem na superfície dos objetos.

Ao invés disso construímos uma imagem do ambiente que circunda o objeto, segundo um dado ponto de vista – tal como na figura 125.

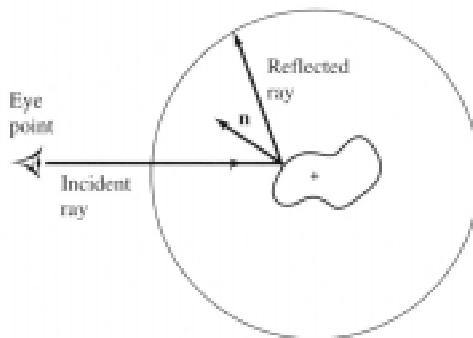


Figura 125 – Construção da imagem do ambiente objeto.

Uma vez de posse dessa imagem, aplicamos a técnica de mapeamento de texturas para conseguir o efeito de um objeto refletindo todo o ambiente.

Com o objetivo de simplificar os cálculos, ao invés de utilizar uma esfera como na figura 125, podemos imaginar que em volta do objeto existe uma caixa que o envolve. Portanto, construir a imagem do ambiente onde ele se localiza significa gerar 6 imagens que serão aplicadas em cada face desse cubo. Na figura 126 temos uma exemplo dessas imagens, já organizadas em função do seu mapeamento nas faces de um cubo.

Uma vez construído o mapeamento do ambiente, a técnica de mapeamento de textura pode ser aplicada para correlacionar os pontos na superfície dessa caixa envolvente na superfície do objeto. O resultado final pode ser visto na figura 127. Podemos observar que o grau de realismo obtido é bastante alto. Apesar do custo computacional não ser tão barato assim, ainda é mais barato que a aplicação de um algoritmo que calcule as reflexões do modelo de forma efetiva.

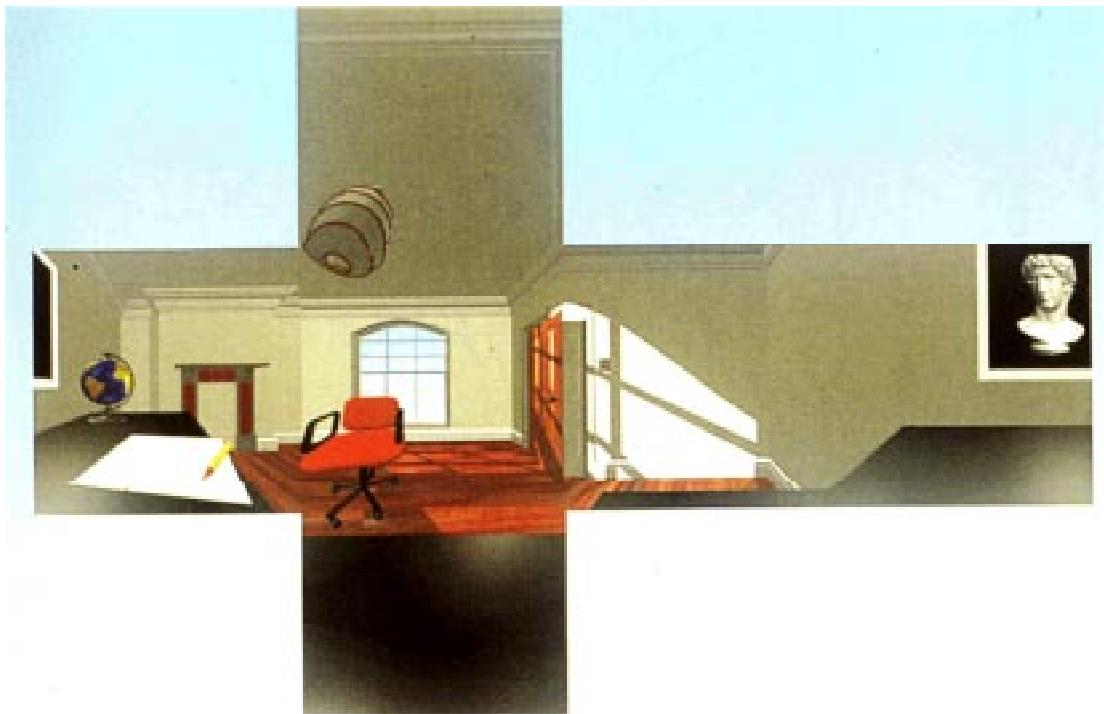


Figura 126 – construção de um mapeamento de ambiente. Em cada face do cubo (visto “aberto” para facilitar a visualização) temos a imagem dos objetos vistos sob aquele ponto de vista.

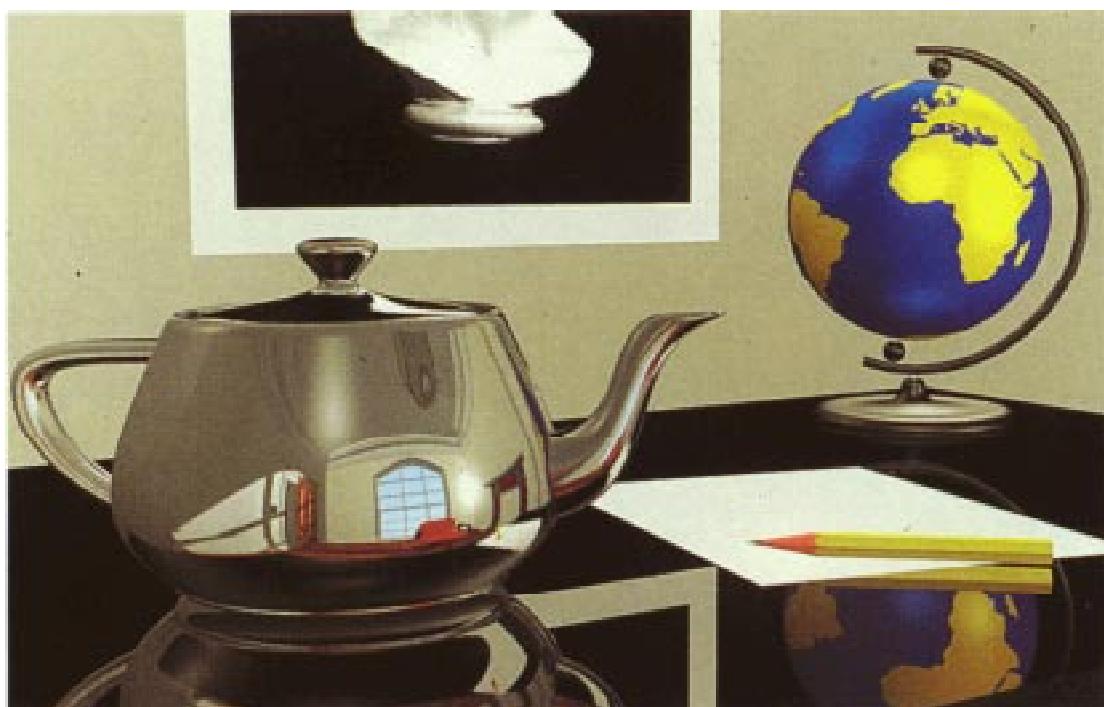


Figura 127 – Aplicação da técnica de mapeamento de ambiente. Os reflexos no pote de chá e na mesa não são calculados através de um modelo computacional, mas sim gerados a partir de imagens da cena.

## *Bump Mapping*

Com o mapeamento de texturas adicionamos aos modelos locais a capacidade de simular materiais heterogêneos, bem como objetos espelhados. No entanto, propriedades de materiais como rugosidade e porosidade, não podem ser representados com realismo suficiente por essa técnica.

Um artifício bastante simples que pode ser acoplado ao modelo local de reflexão apresentado anteriormente é a técnica denominada ***Bump Mapping ou Mapeamento de “Solavancos”***.

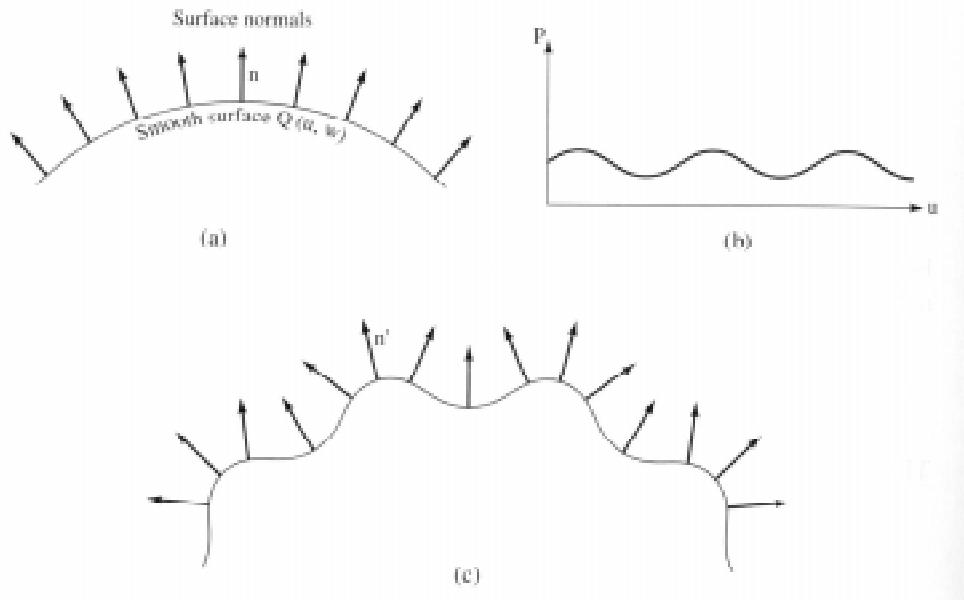
O que desejamos é tornar a superfície dos objetos rugosas ou porosas. Uma possibilidade seria tentar modelar essas a rugosidade ou porosidade do material do objeto de forma explícita. Esse solução mostra-se pouco indicada, já que o grau de refinamento do modelo teria de ser muito grande. Como consequência o número de faces do objetos cresceria, tornando a seu armazenamento e processamento inviáveis. Nossa objetivo é ter sempre modelos tão pequenos quanto possíveis, mantendo uma certa qualidade visual.

A solução para esse problema, portanto, não deve implicar em aumento da resolução da malha de polígonos que representa o objeto.

Ora, se pensarmos um pouco podemos concluir que o que caracteriza uma superfície irregular é a sua orientação varia de forma pouco suave, em contraste as superfícies lisa ou polidas, onde a sua orientação varia de forma muito suave.

A orientação de uma superfície é dada pelo vetor normal em cada um de seus pontos. Portanto se queremos representar uma superfície com irregularidades em sua superfície temos que fazer com que a normal em cada ponto não varie de forma suave (como é presumido nos algoritmos apresentados até o momento) mas sim de forma aleatória e abrupta.

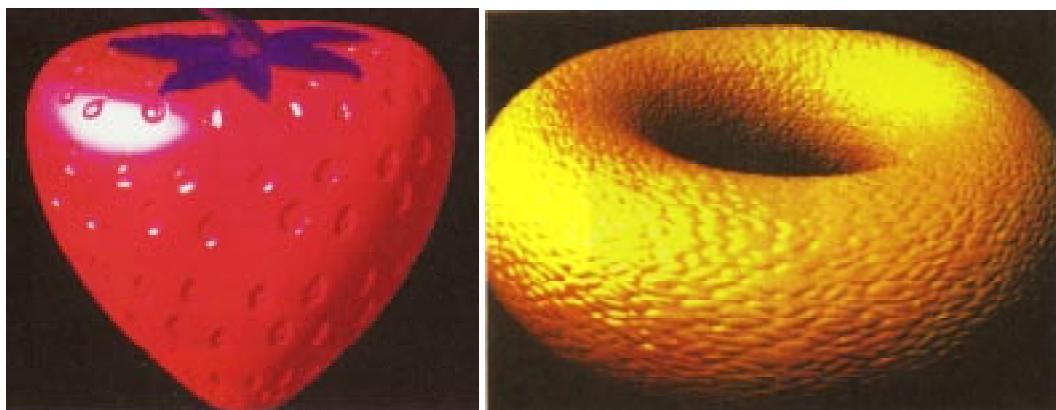
Portanto, a técnica de ***Bump Mapping*** propõe a construção de uma função que promova “solavancos” (**bumps**) na orientação da superfície. Essa função é aplicada quando do cálculo do vetor normal a superfície em um dado ponto. Na figura 128(b) podemos ver o efeito que a função de **bump** promove nas normais de uma superfície bem comportada - figura 128(a).



**Figura 128 – Aplicação da técnica de *Bump Mapping*.** (a) superfície original com seus vetores associados; (b) função de “saltos” ou deslocamentos; (c) vetores normais alterados pela aplicação da função de deslocamento.

Portanto com a inclusão dessa função de “solavancos” podemos simular de forma relativamente simples materiais cuja superfície possui irregularidades. A sua utilização torna os algoritmos de *renderização* mais custosos em função da complexidade da função de deslocamentos. No entanto, essa solução ainda se mostra mais vantajosa que o aumento da resolução do modelo para representar as irregularidades da sua superfície.

Na figura 129 temos duas imagens geradas utilizando-se essa técnica.



**Figura 129 – Aplicação da técnica de *Bump Mapping* para simulação da irregularidade da superfície do modelo.**

## Modelo de Iluminação Global

Como vimos na seção anterior, alguns efeitos que os modelos locais não suportam diretamente, podem ser incorporados para garantir uma melhor qualidade da imagem final. No entanto, todas essas técnicas se resumem a imprimir mais realismo aos objetos. Os efeitos que dependem da modelagem das interações entre objetos não são resolvidos, como iluminação indireta, transmissão e sombras<sup>10</sup>.

Os modelos de iluminação ditos globais, por outro lado, são capazes de reproduzir tais efeitos de forma natural. A seguir, veremos, de forma bastante resumida, as principais idéias por de trás dos dois principais algoritmos que implementam modelos globais : ***Ray Tracing*** e ***Radiosidade***.

### *Ray Tracing*

A idéia fundamental do algoritmo de ***Ray Tracing (Acompanhamento de Raios)*** é, como o próprio nome já sugere, fazer um acompanhamento dos raios luminosos que chegam a um observador. O caminho que cada raio de luz fez, da fonte de luz até atingir o observador, é reconstruindo. Nesse caminho serão computadas “todas” as interações do raio de luz com os objetos da cena. Na figura 130 podemos observar um par de raios de luz provenientes de uma fonte de luz localizada no infinito (por isso suas direções são paralelas). Ao atingirem o cubo os dois raios são refletidos e tomam nova direção, indo de encontro a esfera. Na superfície da esfera, nova reflexão acontece, uma nova direção é determinada e os dois novos raios de luz por fim alcançam o observador.

Portanto, a base do algoritmo de *Ray Tracing* é acompanhar os raios de luz até que alcancem um objeto. Nesse ponto o modelo local de Phong pode ser aplicado, um raio de luz refletido é calculado e um novo acompanhamento se inicia.

O acompanhamento dos raios a partir da fonte de luz é o modelo mais próximo da realidade, afinal os raios de luz emanam das fontes de luz. No entanto, para a implementação, essa abordagem não se mostra conveniente. Primeiro nem todos os raios de luz que partem da fonte irão alcançar o observador. Portanto, podemos gastar tempo avaliando um raio para depois descobrir que esta não alcança o observador. Além disso um raio pode “demorar” muito até alcançar o observador. Se lembarmos que a cada interação a

<sup>10</sup> É possível através de tratamento específico (pós-processamento) adicionar os efeitos de sombra e transmissão. No entanto, esse tratamento não pode ser “adicionado” ao modelo básico de equações proposto por Phong, tal como fizemos com os outros efeitos.

intensidade luminosa decresce, após um certo número de interações, o raio de luz pode ter sua intensidade tão diminuída que sua contribuição para a formação da imagem pode ser praticamente nula.

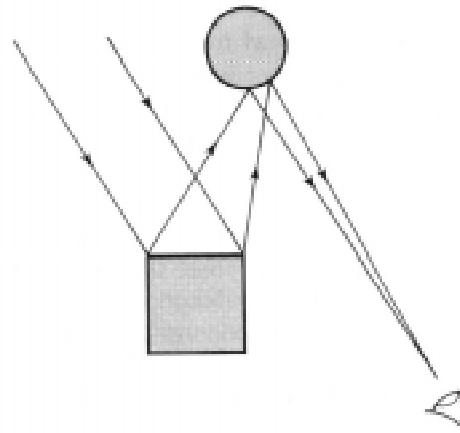
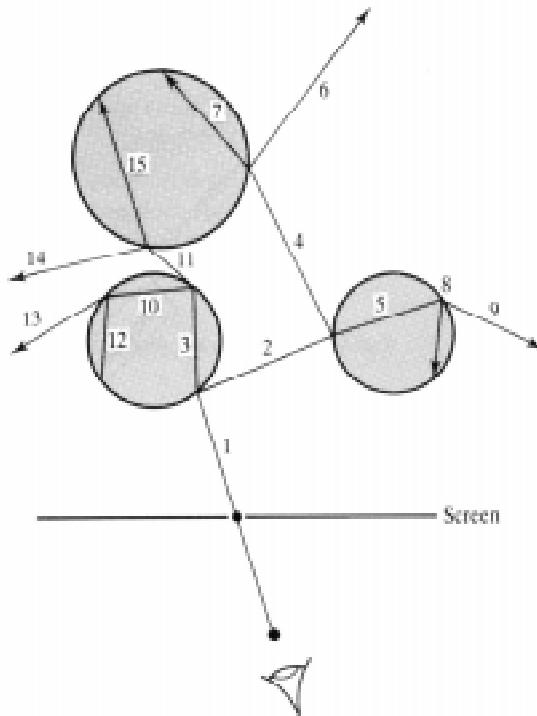


Figura 130 – Acompanhamento de dois raios de luz, provenientes de uma fonte de luz localizada no infinito.

Por esse motivo fazemos o acompanhamento dos raios no sentido oposto ao convencional, ou seja, os raios são acompanhados a partir do observador.

Essa abordagem tem várias vantagens. A primeira é que garantimos que apenas raios de luz que efetivamente alcançam o observador serão considerados. Outra vantagem é que o processo de discretização pode ser gerado de forma natural. Ou seja, o número de raios que precisamos lançar a partir do observador depende diretamente do número de pixels da imagem que queremos gerar.

Na figura 131 temos um exemplo de como esse acompanhamento de um raio é feito. Para um dado pixel da tela, um raio (1) é lançado a partir do observador. Ao alcançar a esfera o raio (1) é subdividido em dois outros raios, (2) e (3), em função, respectivamente, da reflexão e refração do raio (1). O mesmo algoritmo de acompanhamento é aplicado para cada um dos dois novos raios. Após um número  $n$  de interações esse processo é abortado, já que a intensidade da luz diminui a cada interação. O somatório de todas as contribuições encontradas em cada interação a partir do raio inicial define o valor da intensidade de luz do pixel associado ao raio.



**Figura 131 – Processo de acompanhamento de um raio partindo do observador e passando por um pixel da tela. Durante o percurso do raio ocorrem reflexões e refrações.**

### Algoritmo de Ray Tracing

O procedimento **RayTracing** é chamado a partir do algoritmo :

*Algoritmo GeraImagemRayTracing;*

*Dados : resolução da imagem, observador.*

*para* cada pixel  $p_y$  da imagem a ser gerada *faca*

*Calcular direção  $d_y$  do raio que parte do observador e passa pelo pixel  $p_y$ ;*

*RayTracing ( $p_y, d_y, 0$ , cor);*

*Pinta pixel  $p_y$  com cor;*

*fim-para*

*fim.*

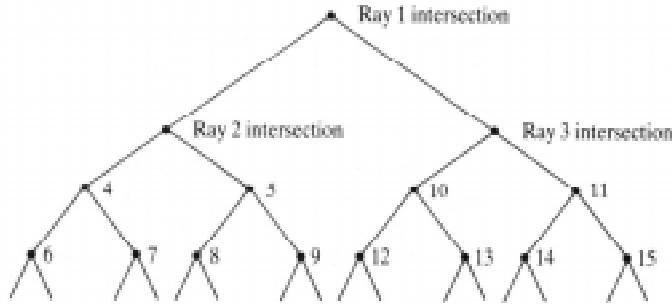
O algoritmo de que gera a imagem de uma cena através do modelo de **Ray Tracing** está baseado em um procedimento recursivo como descrito a seguir.

**Procedimento RayTracing** (*inicio, direção* : vetor; *profundidade* : inteiro; *cor* : RGB);  
*Dados* : fontes de luz, observador, objetos

```

CorLocal := RGB(preto);
se profundidade > NívelMáximo então
    cor := RGB(preto);
senão
    Calcular os pontos de interseção do raio de luz definido por inicio e direção com todos os
    objetos da cena;
    se não há pontos de interseção então
        cor := RGB(cor de fundo);
    senão
        Obter o ponto de interseção P, mais próximo do inicio do raio;
        para cada fonte de luz faz
            Calcular a intensidade de luz refletida local utilizando a equação de Phong;
            CorLocal := CorLocal + intensidade de luz refletida local;
        fim-para
        Calcular a direção dr do novo raio de luz refletido no ponto P;
        RayTracing (P, dr, profundidade + 1, CorReflexão);
        se material do objeto é transmissor então
            Calcular a direção dt do novo raio de luz refratado no ponto P;
            RayTracing (P, dt, profundidade + 1, CorTransmissão);
        fim-se;
        Cor := compõe(CorReflexão, CorTransmissão, CorLocal);
    fim-se;
fim-se;
fim.
```

Na figura 132 temos a representação gráfica do processo recursivo do algoritmo **GeraImagemRayTracing**, quando aplicado ao raio (1) da figura 131. Em cada nó da árvore temos um ponto de interseção de um raio com um objeto da cena.



**Figura 132 – Árvore associada ao processo recursivo do algoritmo de Ray Tracing. Essa árvore é equivalente ao acompanhamento do raio (1) apresentado na figura 131.**

A grande vantagem desse algoritmo é permitir que a componente refletida / refratada da luz que chega ao observador seja calculada de maneira **global**. Ou seja, ao aplicar o algoritmo recursivamente em cada ponto onde há reflexão / refração estamos levando em conta as interações que não são resultado da iluminação direta. Esse fato é importante já que será uma forma de “assinatura” das imagens geradas por esse algoritmo : superfícies espelhadas ou transparentes. No entanto, a componente difusa / ambiente continua sendo simulada quando da aplicação da equação de *Phong*. Portanto, continuam sendo avaliadas de forma local.

O parâmetro **profundidade** tem como função controlar o nível de recursão em que processo se encontra em determinado momento. Essa profundidade está relacionada com o número de reflexões / transmissões que serão levadas em conta na geração da imagem. Na figura 133 podemos ver o efeito que o aprofundamento do processo recursivo gera na imagem final. No detalhe da imagem podemos ver como o aumento do nível máximo de recursão gera um número cada vez maior de reflexões entre as esferas.

Outro cálculo que pode ser facilitado dentro desse algoritmo é o cálculo da sombra de um objeto. Apesar do algoritmo em si não prever esse cálculo, podemos a cada interação testar se aquele ponto está em uma região de sombra ou não. Aproveitando a idéia do acompanhamento de raios, podemos lançar um raio do ponto em questão até a fonte de luz e ver se existe algum objeto nesse caminho. Caso exista sabemos que essa fonte não pode contribuir para a iluminação desse ponto, ou seja, ele está em uma região de sombra para essa fonte de luz.

Apesar de apresentar resultados bem mais realistas que os algoritmos locais (como já era de se esperar), o algoritmo de **Ray Tracing** possui algumas dificuldades. O seu custo computacional é relativamente alto, e é função do

número de objetos na cena (e sua complexidade) assim como da resolução da imagem que se pretende gerar (vinculado ao número de raios a serem lançados).

Outra característica é que o algoritmo é dependente do ponto de vista do usuário. Ou seja, em uma animação, toda vez que o observador se desloca é necessário recalcular a imagem.

Apesar de representar com bastante realismo as componentes refletida especular e refratada da luz, esse algoritmo ainda se mostra deficiente na representação da componente difusa da luz refletida. Ela continua a ser levada em conta apenas quando da avaliação local. A seguir veremos um algoritmo que consegue representar de maneira global essa componente.

### *Radiosidade*

O modelo de iluminação denominado ***Radiosidade*** está baseado em um modelo de conservação de energia ou equilíbrio de energia. Esse modelo foi originalmente proposto por Siegel e Howell, em 1984, para a análise de transferência de calor entre modelos de engenharia. No nosso contexto a energia em questão é a luminosa.

O que o modelo propõe é que a quantidade de energia que uma determinada região da cena emite é função da energia de emissão mais a energia refletida pelas outras regiões da cena. Essas regiões são denominadas ***patches*** ou ***retalhos***, e a sua área também tem influência na quantidade de energia que aquele *patch* emite. Na figura 134 temos uma imagem gerada por esse modelo e sua respectiva discretização em *patches*.

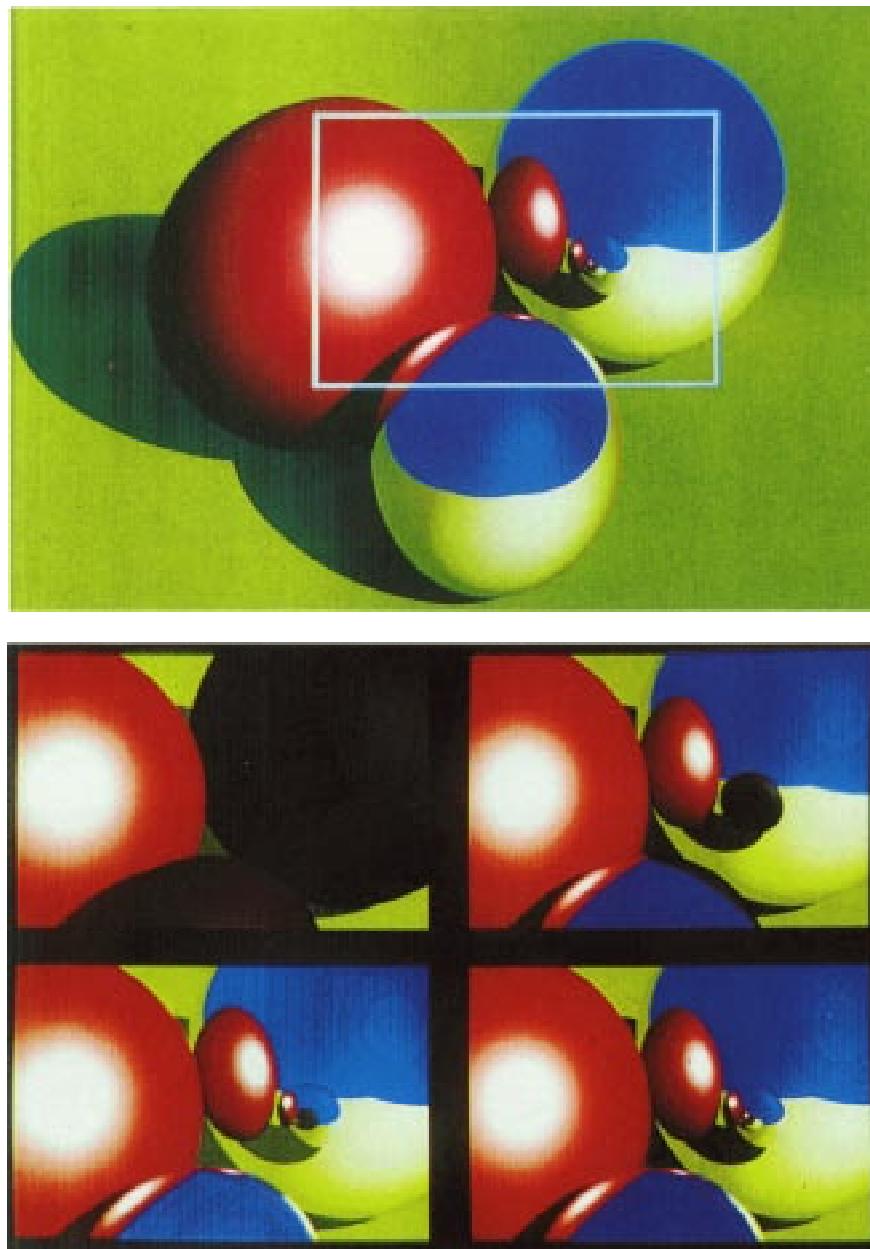


Figura 133 – Exemplo de imagem gerada pelo algoritmo de *Ray Tracing*.

Na figura do topo podemos observar três esferas, sendo duas delas espelhadas. Em detalhe na figura abaixo o resultado da variação no nível máximo de profundidade associado ao processo recursivo.

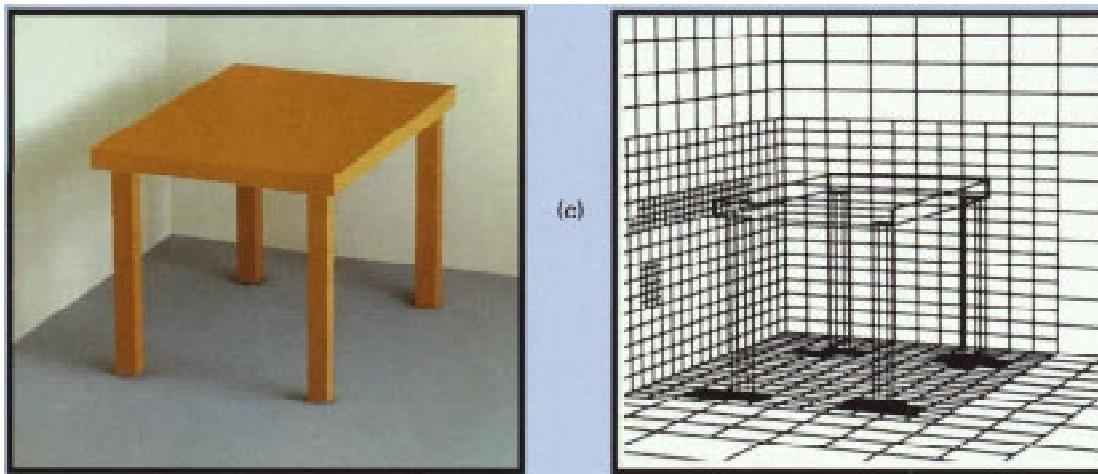


Figura 134 – Imagem gerada pelo algoritmo de radiosidade. À direita os *patches* definidos para a cena.

Podemos resumir essa relação pela equação :

$$\mathbf{B}_i = \mathbf{E}_i + \mathbf{R}_i \int_j \mathbf{B}_j \cdot \mathbf{F}_{ij} d\mathbf{A}_j$$

Onde  $\mathbf{B}_i$  é a **radiosidade** de um **patch**;  $\mathbf{E}_i$  é a energia emitida pelo **patch**;  $\mathbf{R}_i$  a energia refletida ponderada pela **radiosidade** de cada patch  $j \neq i$ . A integral é calculada em  $d\mathbf{A}_j$  onde  $\mathbf{A}_j$  é a área do **patch**  $j$ .

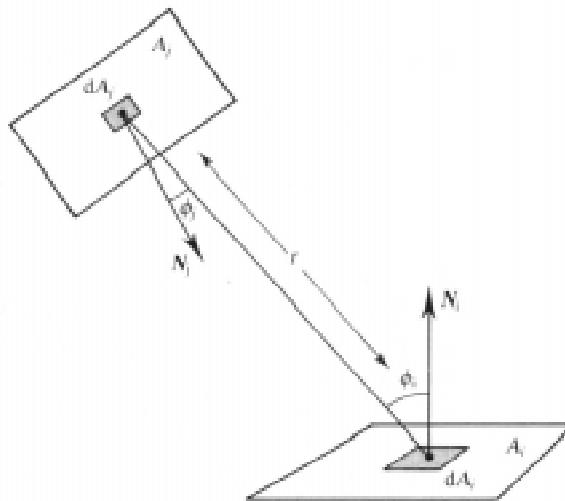
A formulação discreta dessa mesma equação é :

$$\mathbf{B}_i = \mathbf{E}_i + \mathbf{R}_i \sum_j \mathbf{B}_j \cdot \mathbf{F}_{ij}$$

Os fatores  $\mathbf{F}_{ij}$  são chamados **fatores de forma** e são utilizados para se levar em consideração o posicionamento relativo entre dois *patches*  $i$  e  $j$ . Seu cálculo é dado pela formula :

$$\mathbf{F}_{ij} = 1/A_i \int_{A_i} \int_{A_j} \cos(\phi_i) \cdot \cos(\phi_j) / \pi r^2 d\mathbf{A}_j d\mathbf{A}_i$$

Podemos ver a interpretação de cada um dos termos da equação acima na figura 135.



**Figura 135 – Parâmetros utilizados no cálculo do fator de forma associados a dois patches.**

Dessa forma, o modelo apresentado leva em conta a participação global de todos os **patches** no cálculo da energia luminosa da refletida.

Uma análise mais precisa desse modelo vai além do escopo desse trabalho. No entanto, podemos ressaltar algumas das suas principais características.

A primeira vantagem desse modelo é ser independente da ponto de vista do observador. Na sua formulação em momento algum a posição de observação é levada em conta, o que é razoável para um modelo baseado em conservação de energia.

Portanto esse método é muito interessante para aplicação em animações onde os objetos são estáticos e a camera “passeia” pela cena – denominados **walk-throught**. Os cálculos associados a determinação da intensidade luminosa de um **patch** podem ser feitos uma única vez (desde que os objetos não mudem de posição).

No entanto, como podemos observar, o modelo matemático é muito mais pesado que os modelos anteriormente estudados. Portanto, seu custo computacional é uma das grandes barreiras a sua utilização.

O grau de realismo das imagens geradas é muito alto. No entanto, pela sua própria formulação, esse modelo não abrange as componentes de reflexão especular e transmissão da luz. Por isso esse método é indicado para cenas de ambientes fechados onde os objetos sejam predominantemente refletores difusos – como os exemplos da figura 136.



**Figura 136 – Imagens geradas pelo modelo de radiosidade. Podemos observar a ausência de objetos espelhados ou transparentes. Em contrapartida as regiões de sombra e penumbra são muito realistas.**