



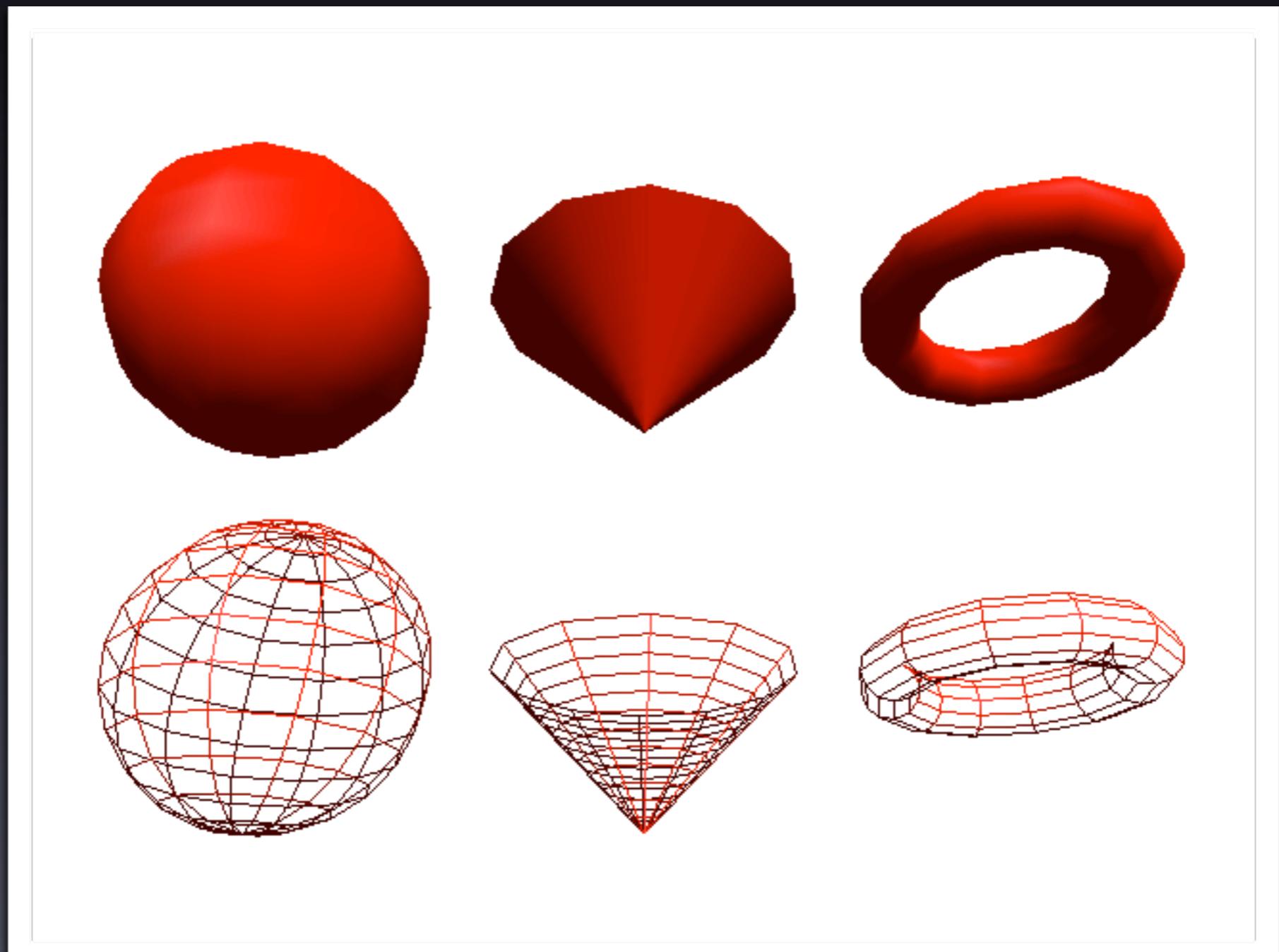
[www.ime.usp.br/dcc](http://www.ime.usp.br/dcc)

# MAC-420: Introdução à Computação Gráfica

Marcel P. Jackowski  
[mjack@ime.usp.br](mailto:mjack@ime.usp.br)

**Aula #2: Aspectos históricos**

# shapes.c



# Tipos de representações

## ■ Vetorial

- Representados por coleções de objetos geométricos
  - Pontos
  - Retas
  - Curvas
  - Planos
  - Polígonos

## ■ Matricial

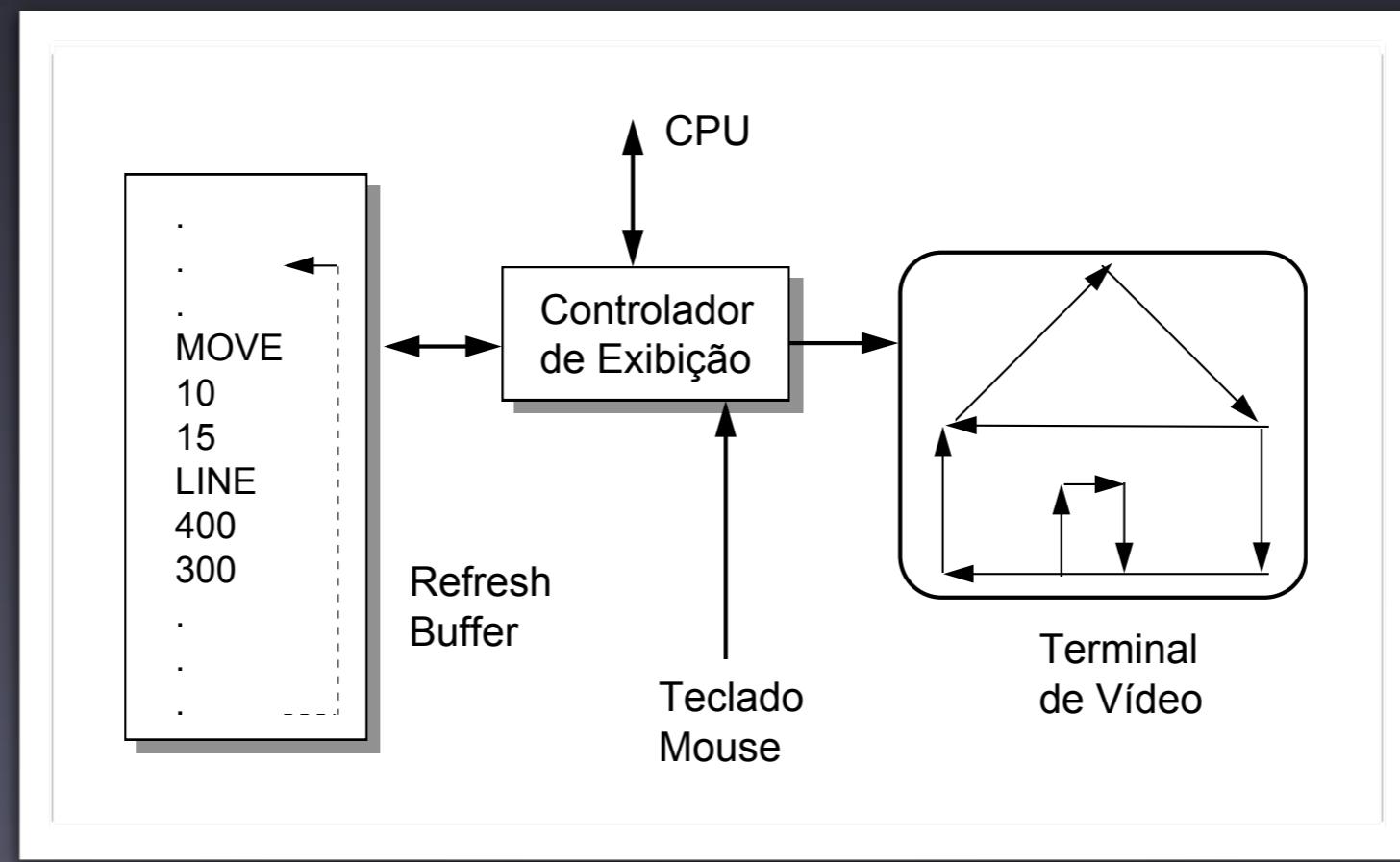
- Amostragem em grades retangulares (ou hexagonais)
  - Matrizes de “pixels”
  - Cada pixel representa uma intensidade de alguma natureza

# Representação vetorial

- **Permitem uma série de operações sem (quase) perda de precisão**
  - Transformações lineares / afins
  - Deformações
- **Por que “quase” ?**
  - É necessário usar aproximações
  - Representação em ponto-flutuante
  - Números racionais
- **Complexidade de processamento = O (no vértices / vetores)**
- **Exibição**
  - Dispositivos vetoriais
  - Dispositivos matriciais (requer amostragem, i.e., rasterização)

# Dispositivo vetorial

- Antigamente, o modo de operação destes dispositivos era semelhante a um osciloscópio
- A exibição no CRT era feita na sequência e posição dos pontos correspondentes às primitivas gráficas a serem exibidas

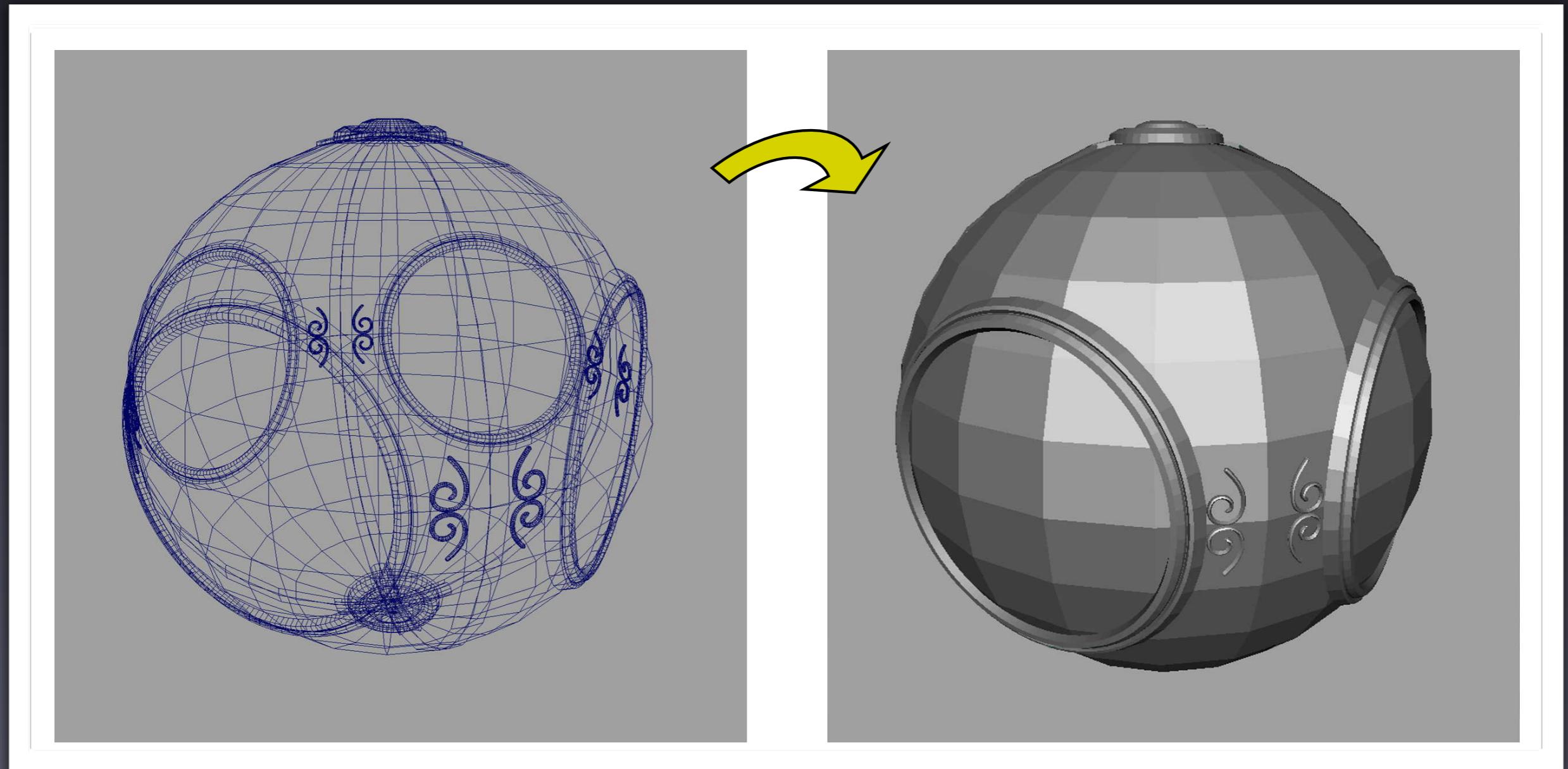


# Representação matricial

- Representação flexível e muito comum
- **Complexidade de processamento =  $O(\text{no. de pixels})$**
- Muitas operações implicam em perda de precisão (reamostragem)
  - Ex.: rotação, escala
- Técnicas para lidar com o problema
  - Ex.: técnicas anti-serrilhado (anti-aliasing)
- Exibição
  - Dispositivos matriciais
  - Dispositivos vetoriais (requer uso de técnicas de reconhecimento de padrões)

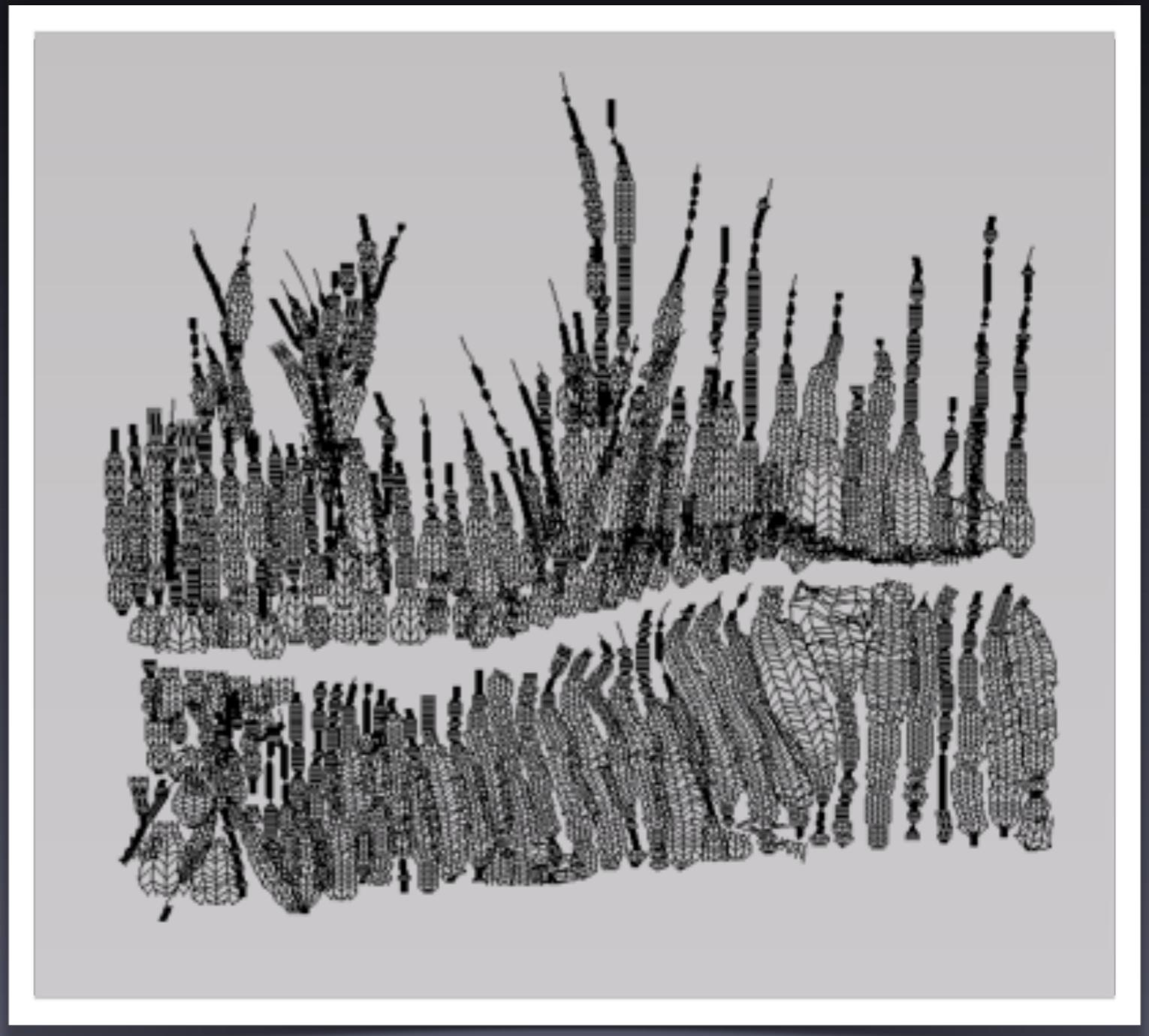
# Representação matricial

- Nos possibilita converter wireframes em polígonos preenchidos



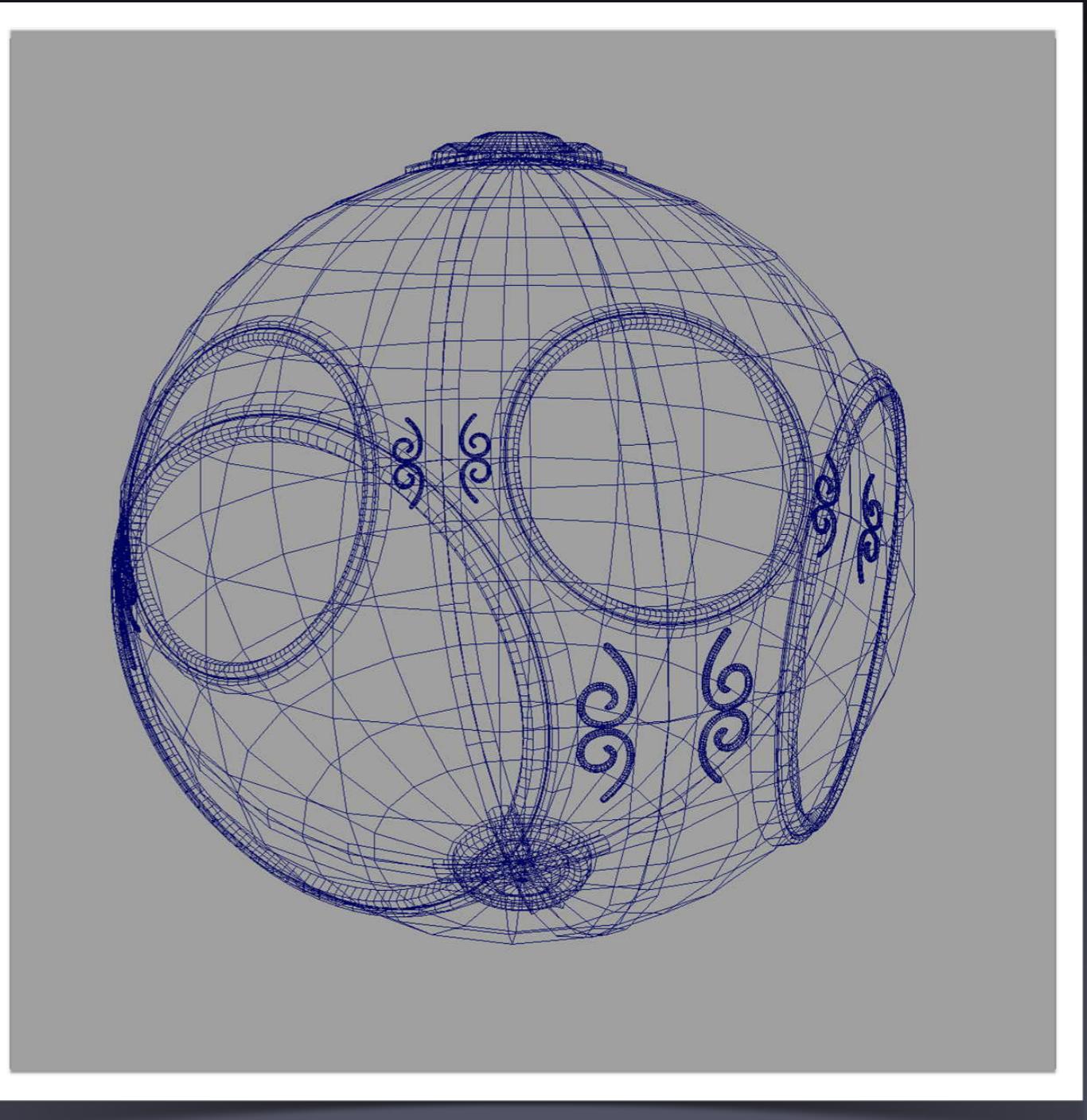
# 1950-1960

- A computação gráfica teve seu início nos primeiros dias da computação
  - Pen plotters
  - Displays simples usando conversores A/D entre computadores e CRTs caligráficas
  - Custo de atualização do CRT era muito alto
  - Computadores eram lentos e caros



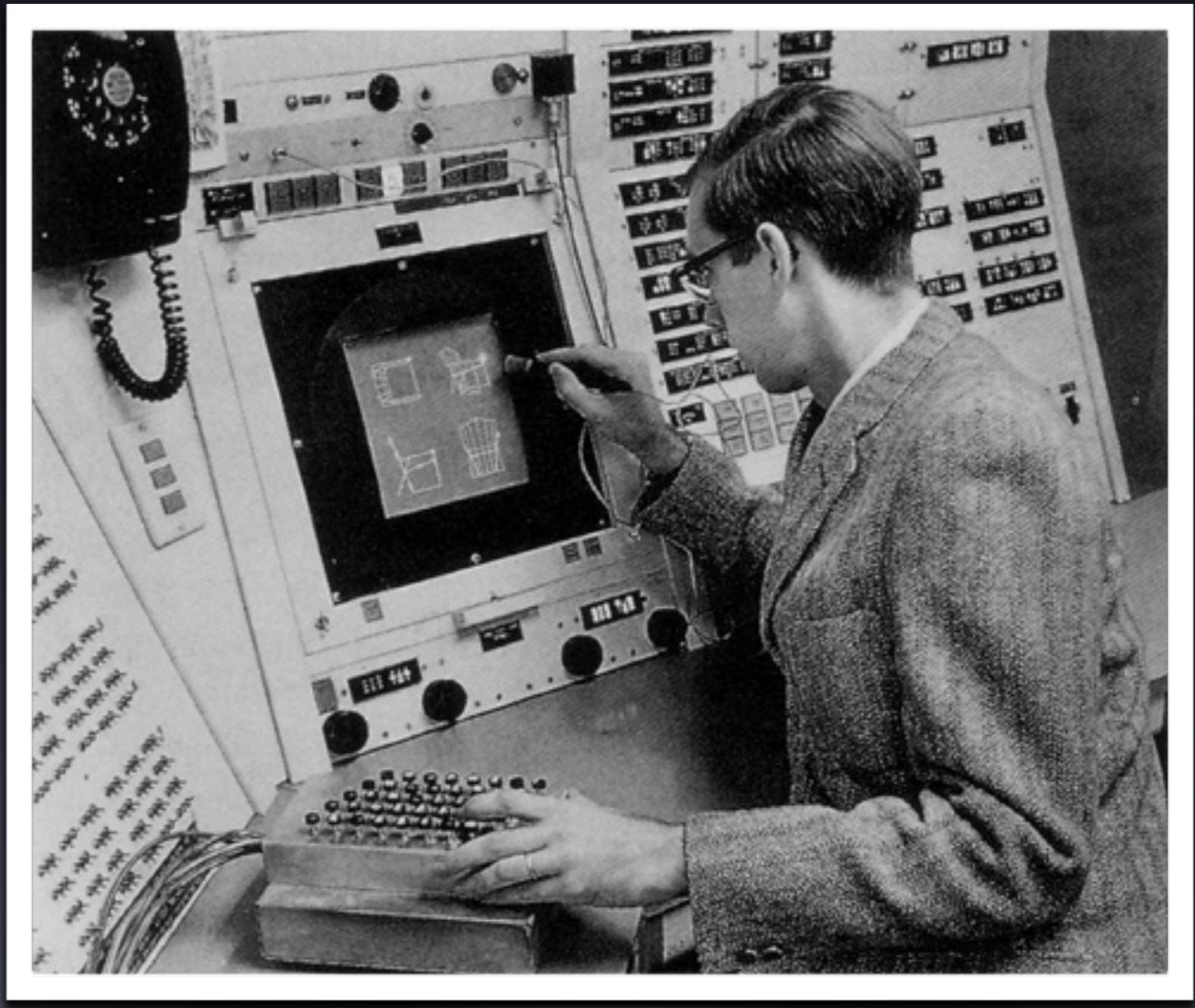
# 1960-1970

- Gráficos “**wireframe**”
  - “Sketchpad”
  - “Display processor”
  - “Storage tube”



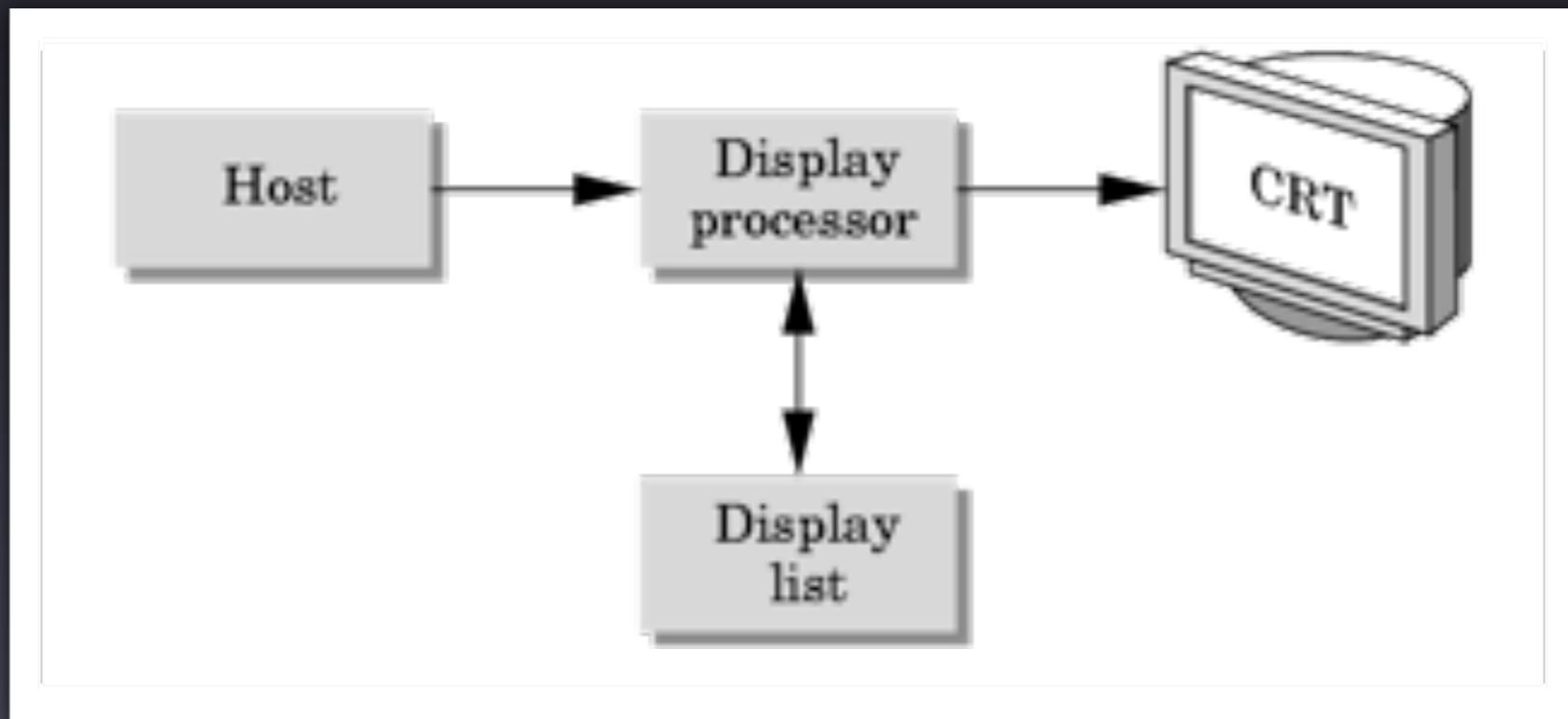
# “Sketchpad”

- Tese de Ph.D. de Ivan Sutherland na MIT (1963)
- Reconhecimento do potencial da interação entre homem e máquina



# “Display processor”

- Ao invés de deixar o computador cuidar da atualização do display, utiliza-se um computador dedicado chamado de Display Processor Unit (DPU)



- Gráficos são armazenados em um “display list” na DPU
- O computador compila a “display list” e manda-a para a DPU

# “Direct view Tube”

- Storage tube criado pela Tektronix
  - Não necessitava de atualizações constantes
  - Padronização de interface
    - Disponibilização de software padronizado
  - Plot3D em Fortran
- Relativamente barato
  - Abriu as portas da computação gráfica para a comunidade de CAD



# 1970-1980

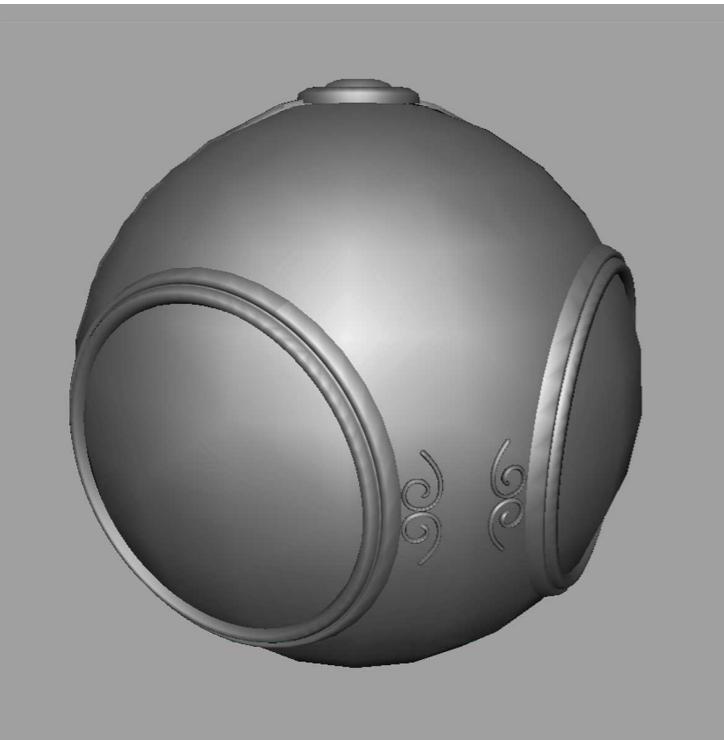
- Início dos padrões gráficos
  - FIPS (“Federal Information Processing Standard”)
    - GKS (“Graphics Kernel System”)
      - Tornou-se padrão ISO 2D - Europa
  - Gráficos “raster”
  - Workstations e PCs

# PCs e “Workstations”

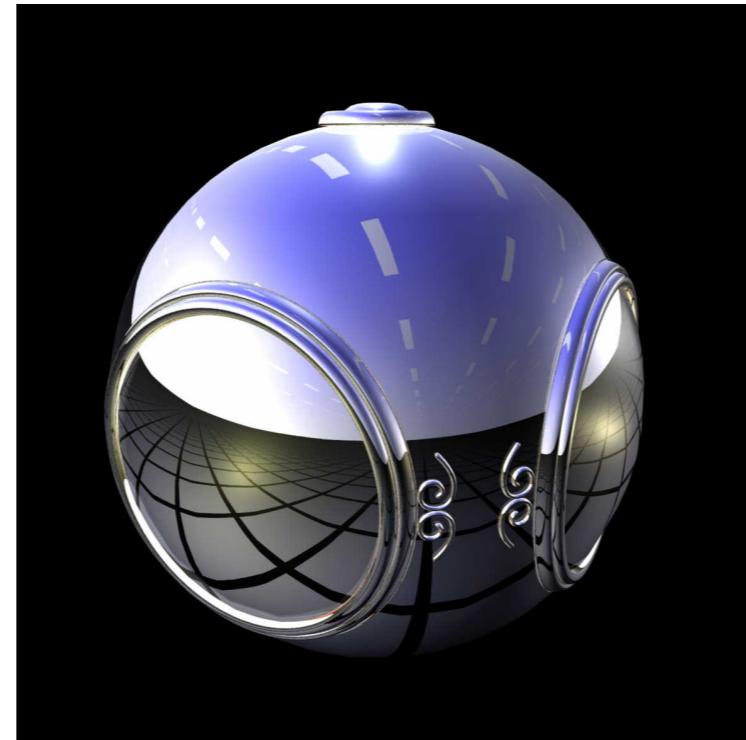
- Embora não façamos mais a distinção entre eles, historicamente PCs e workstations evoluíram de raízes diferentes;
- As primeiras workstations caracterizavam-se por:
  - Conexão em rede: modelo cliente-servidor
  - Alto nível de interatividade
- Primeiros PCs incluiam o frame buffer com parte da memória do computador:
  - Facilidade na troca de conteúdo e imagens

# 1980-1990

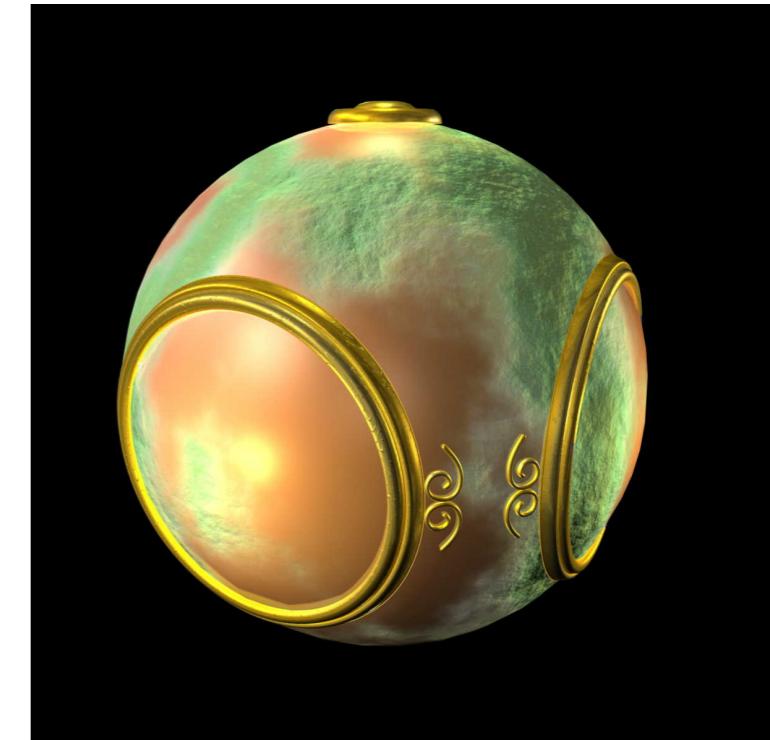
- Realismo



**smooth shading**



**environment  
mapping**



**bump mapping**

# 1980-1990

- Hardware especializado
  - Silicon Graphics “geometry engine”
    - Implementação do pipeline gráfico em VLSI
- Padrões da indústria
  - PHIGS (3D)
  - RenderMan (Pixar)
- Gráficos em rede: X Window System
- Interface Homem-Computador (HCI)

# 1990-2000

- OpenGL API
- Filmes inteiramente gerados por computador (e.g. Toy Story, 1995)
- Novas capacidades em hardware
  - Mapeamento de textura
  - Blending
  - Buffers de acumulação e estêncil

# 2000-atual

- Fotorealismo
- Placas gráficas para PC dominam o mercado:
  - NVidia, AMD (ex ATI), Intel
  - Pipelines programáveis
    - GPU
- Game boxes (videogames) determinam a direção do mercado
- CG se torna rotina em produções cinematográficas: Maya, Lightwave

# Elementos gráficos

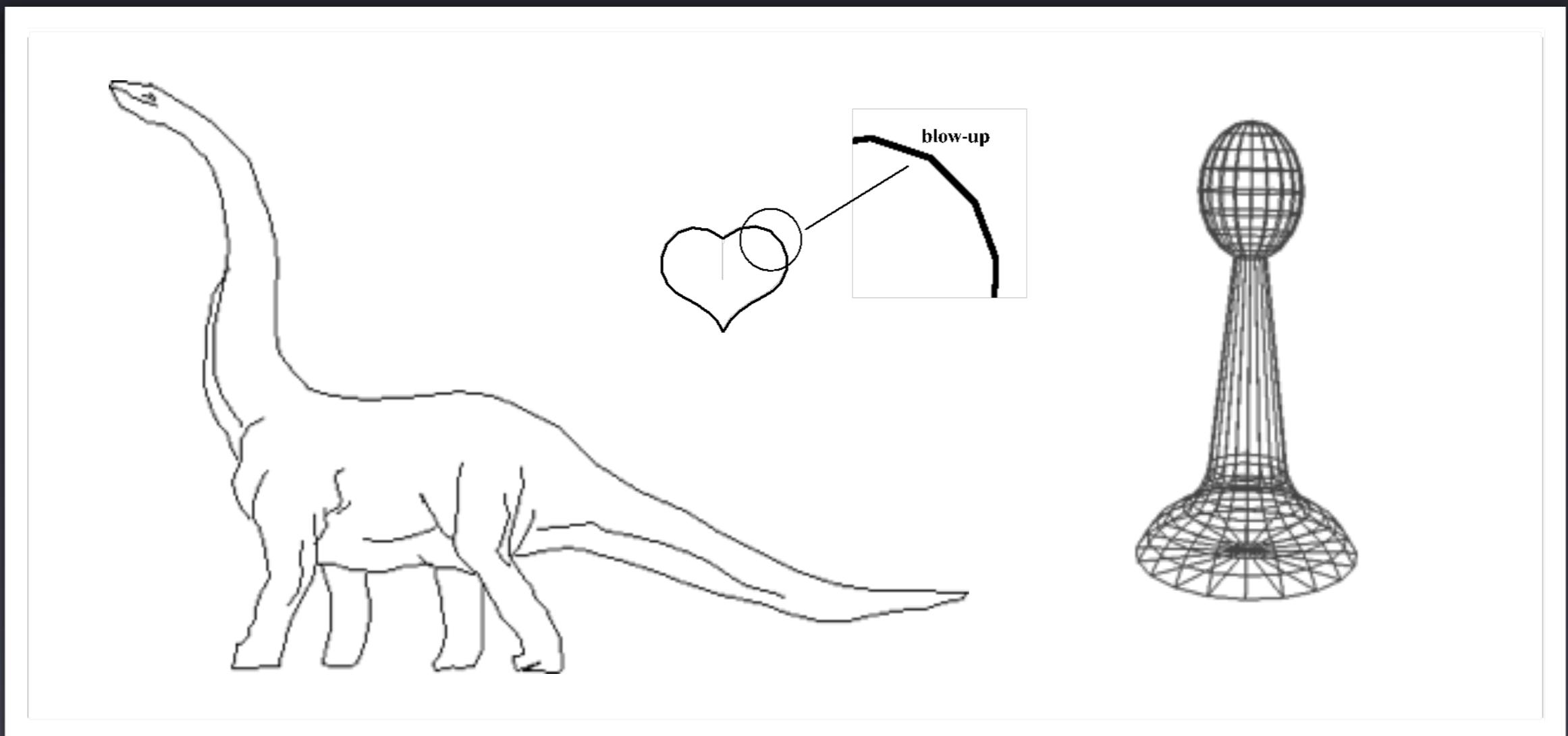
- **Primitivas:**

- pontos
- linhas
  - polilinhas
  - polígonos
- texto
- imagens raster

- **Atributos:** como aparece uma primitiva

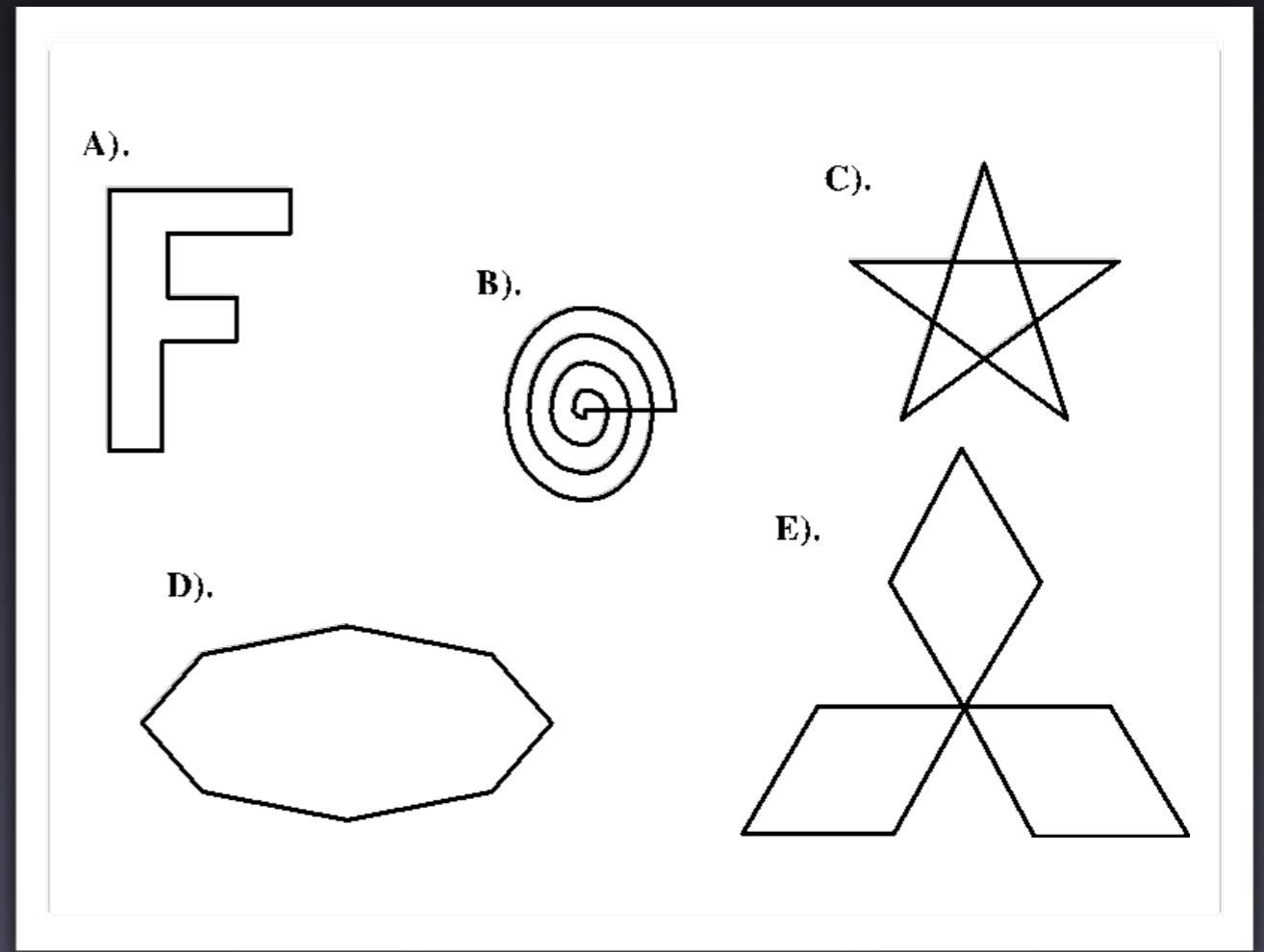
# Polilinhas

- Sequência de linhas retas interconectadas
  - Pode ter o aspecto de uma curva suave



# Polígonos

- Um polígono tem o seu primeiro ponto e último ponto conectado por uma aresta.
- Se nenhuma aresta se cruza, o polígono é chamado de simples. Somente A) e D) são simples.



# Propriedades de textos

- Cor, espessura e “stippling” das arestas, e a maneira estas se combinam nos seus pontos de conexão.



a).



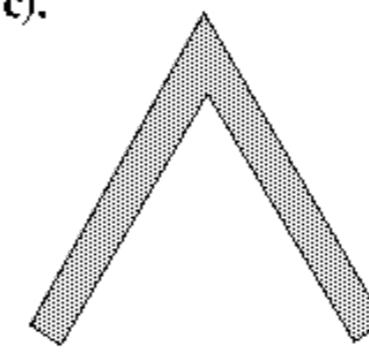
b).



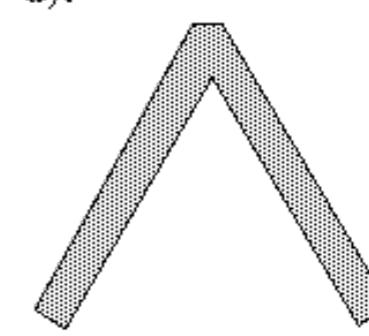
d).



c).



d).



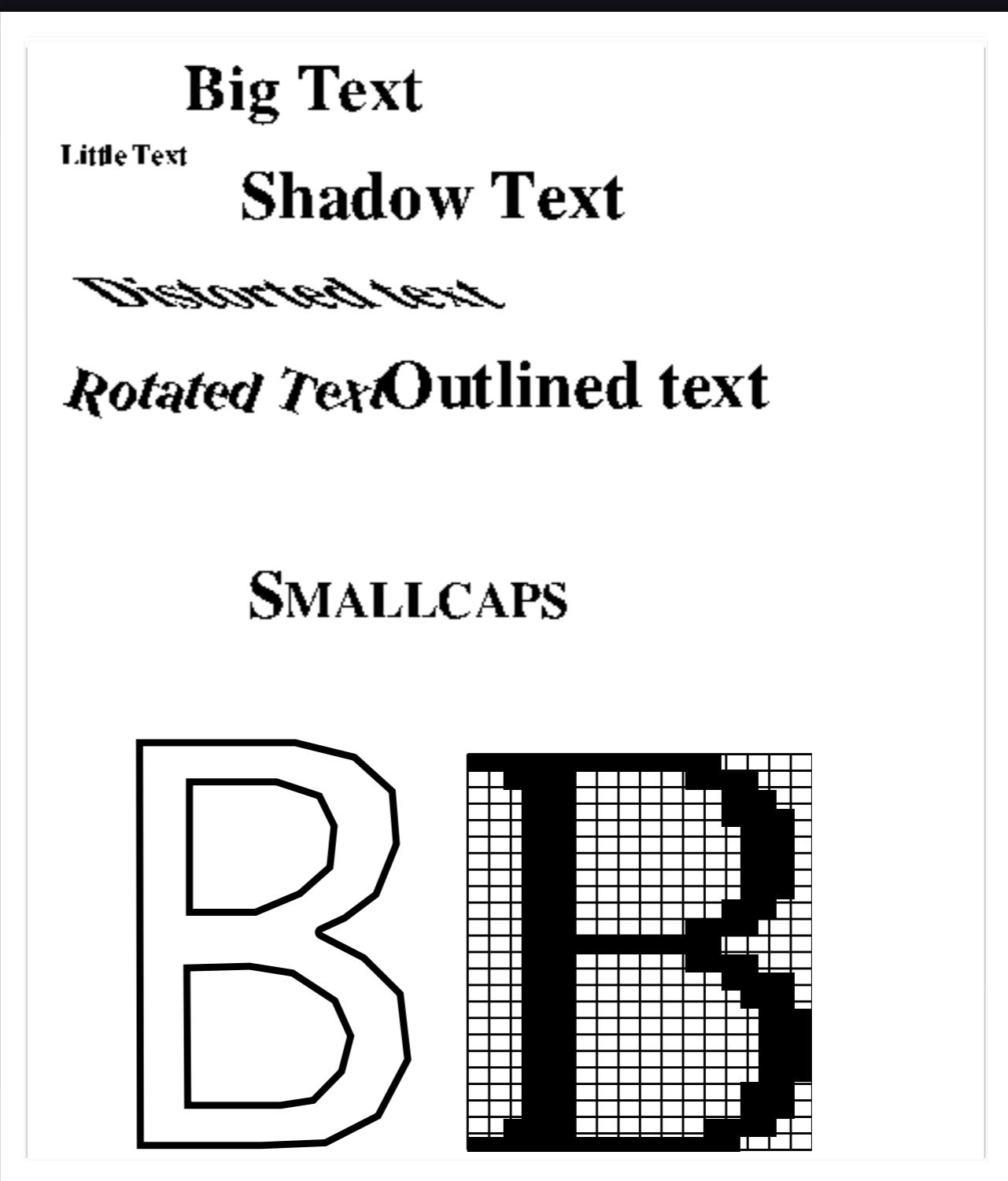
# Texto

- **Atributos:**

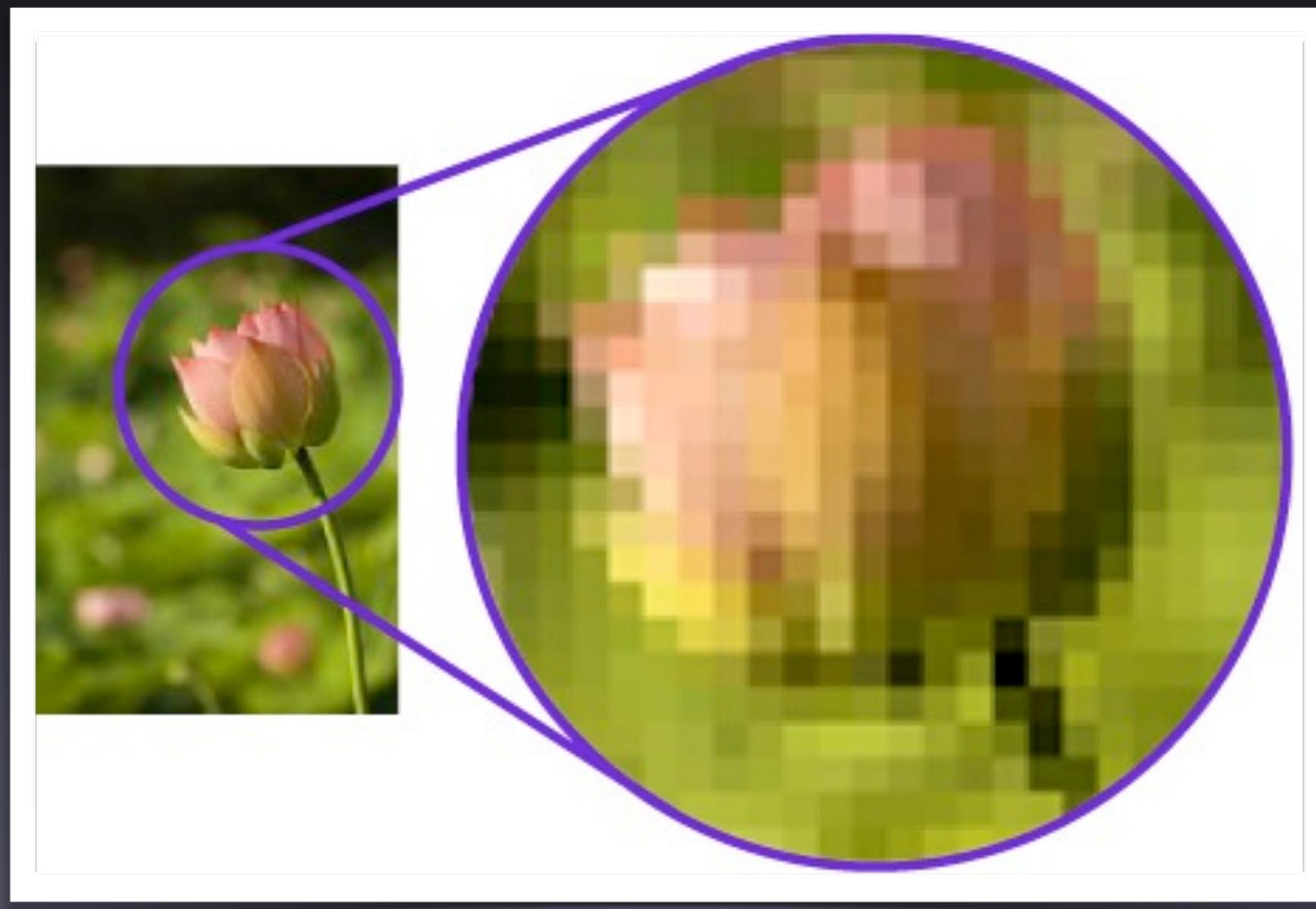
- Fonte
- Cor
- Tamanho
- Espaçamento
- Orientação

- **Representação:**

- Pontos ou linhas



# Imagenes raster

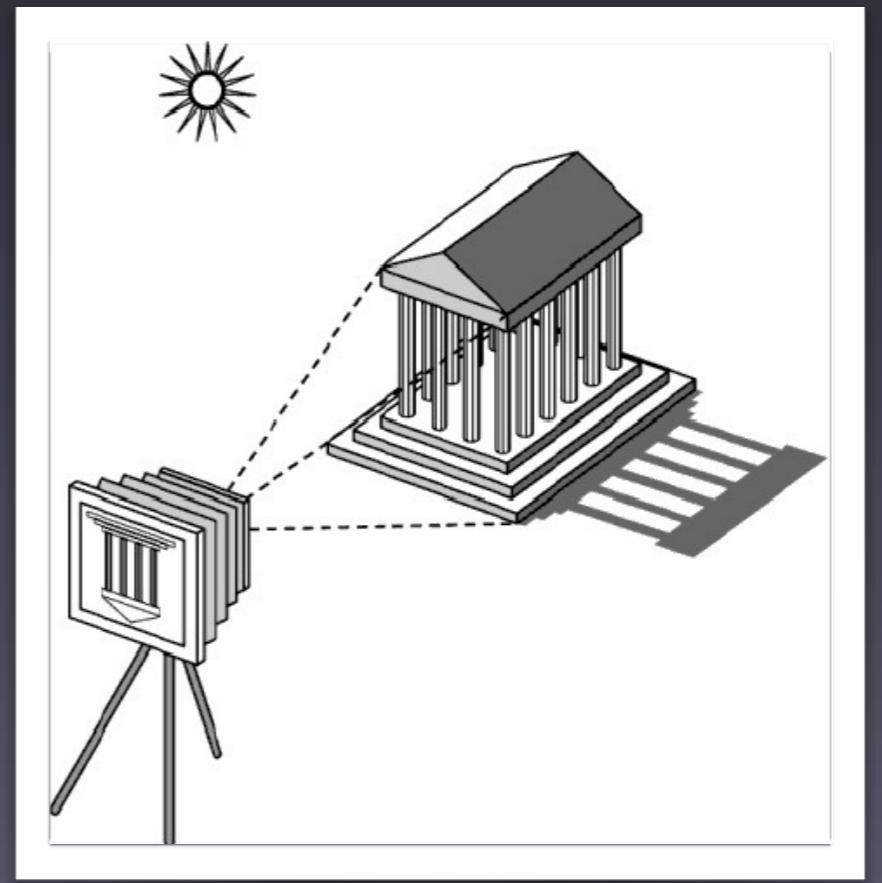


# Programação gráfica

- À primeira vista: basta desenhar
  - Uma subrotina para desenhar cada tipo de objeto
- Porém
  - Como fazer interação?
  - Como estruturar a cena?
  - Como controlar os atributos dos objetos?
  - Como suportar diversos dispositivos gráficos?
  - Como fazer programas independentes dos sistemas operacionais?

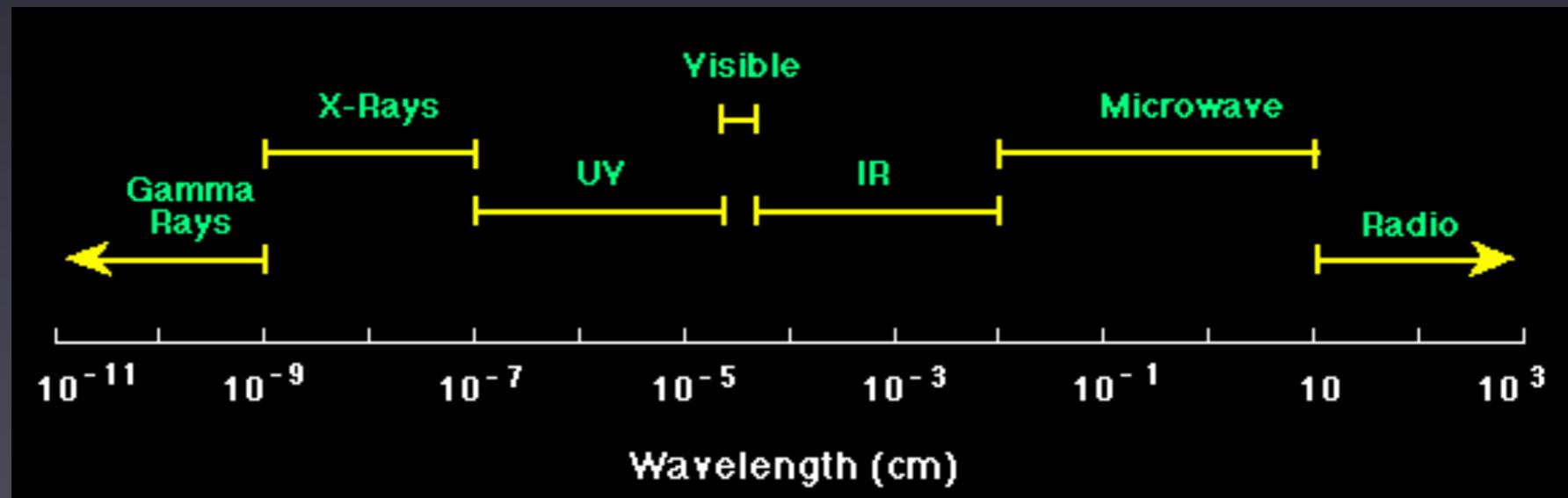
# Formação da imagem

- Usado um processo análogo a das imagens que são formadas por sistemas de imagem físicos (e.g. câmeras, microscópios, sistema visual humano)
- Elementos da formação da imagem
  - Objetos
  - Observador (Viewer)
  - Fonte(s) de iluminação



# LUZ

- A luz é parte do espectro eletromagnético que é sensível ao sistema visual humano
  - Ondas eletromagnéticas que têm tamanho variando entre 350 e 750 nanômetros
  - Ondas maiores aparecem como vermelho e menores como azul (infra-vermelho e ultra-violeta)



# Comprimentos de onda

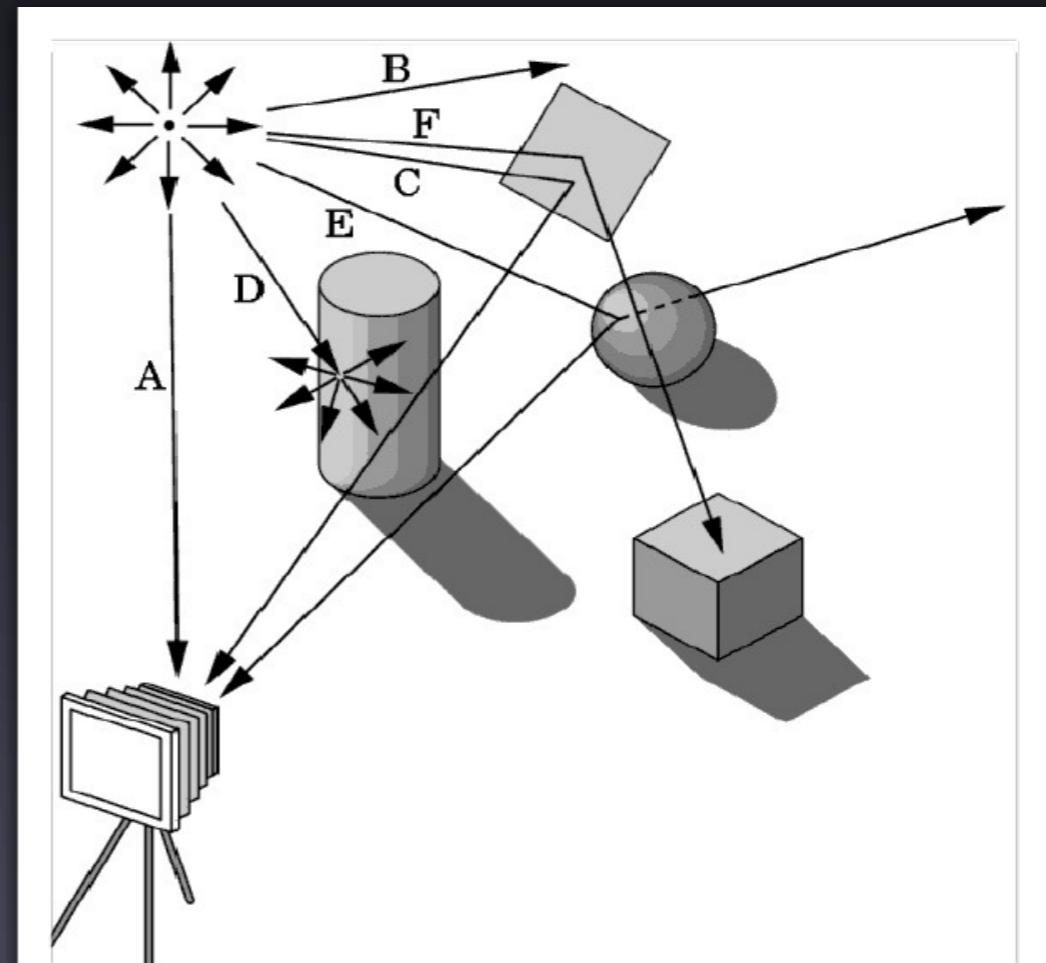
Cores do espectro visível		
Cor	Comprimento de onda	Freqüência
vermelho	~ 625-740 nm	~ 480-405 THz
laranja	~ 590-625 nm	~ 510-480 THz
amarelo	~ 565-590 nm	~ 530-510 THz
verde	~ 500-565 nm	~ 600-530 THz
ciano	~ 485-500 nm	~ 620-600 THz
azul	~ 440-485 nm	~ 680-620 THz
violeta	~ 380-440 nm	~ 790-680 THz

*Espectro Contínuo*

400 500 600 700 800

# Traçado de raios

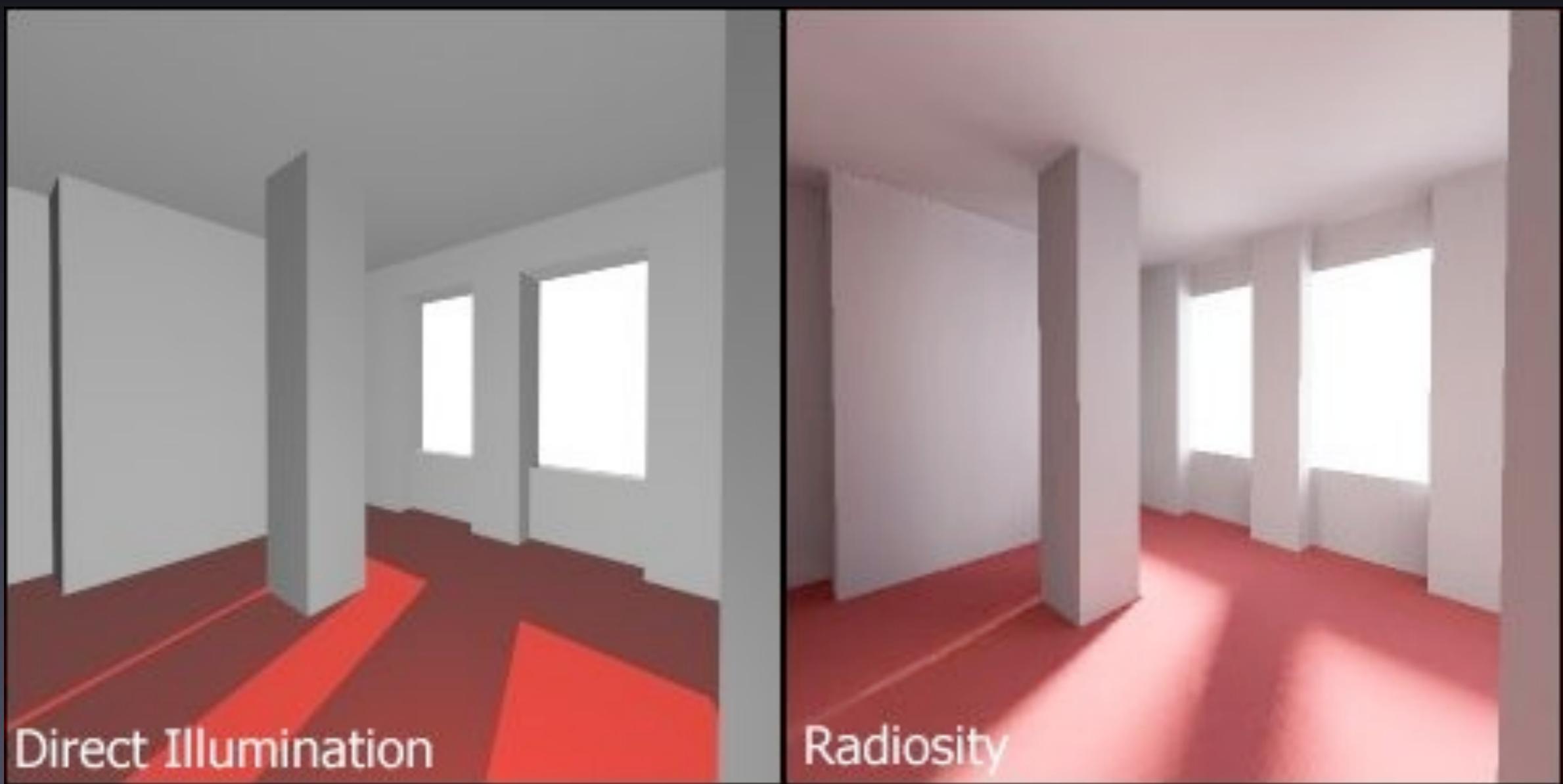
- Uma maneira de formar uma imagem é seguir raios de luz encontrando quais desses raios entram nas lentes da câmera
- Entretanto, cada raio de luz pode ter múltiplas interações com objetos antes de ser absorvido ou ir para o infinito



# Traçado de raios



# Radiosidade

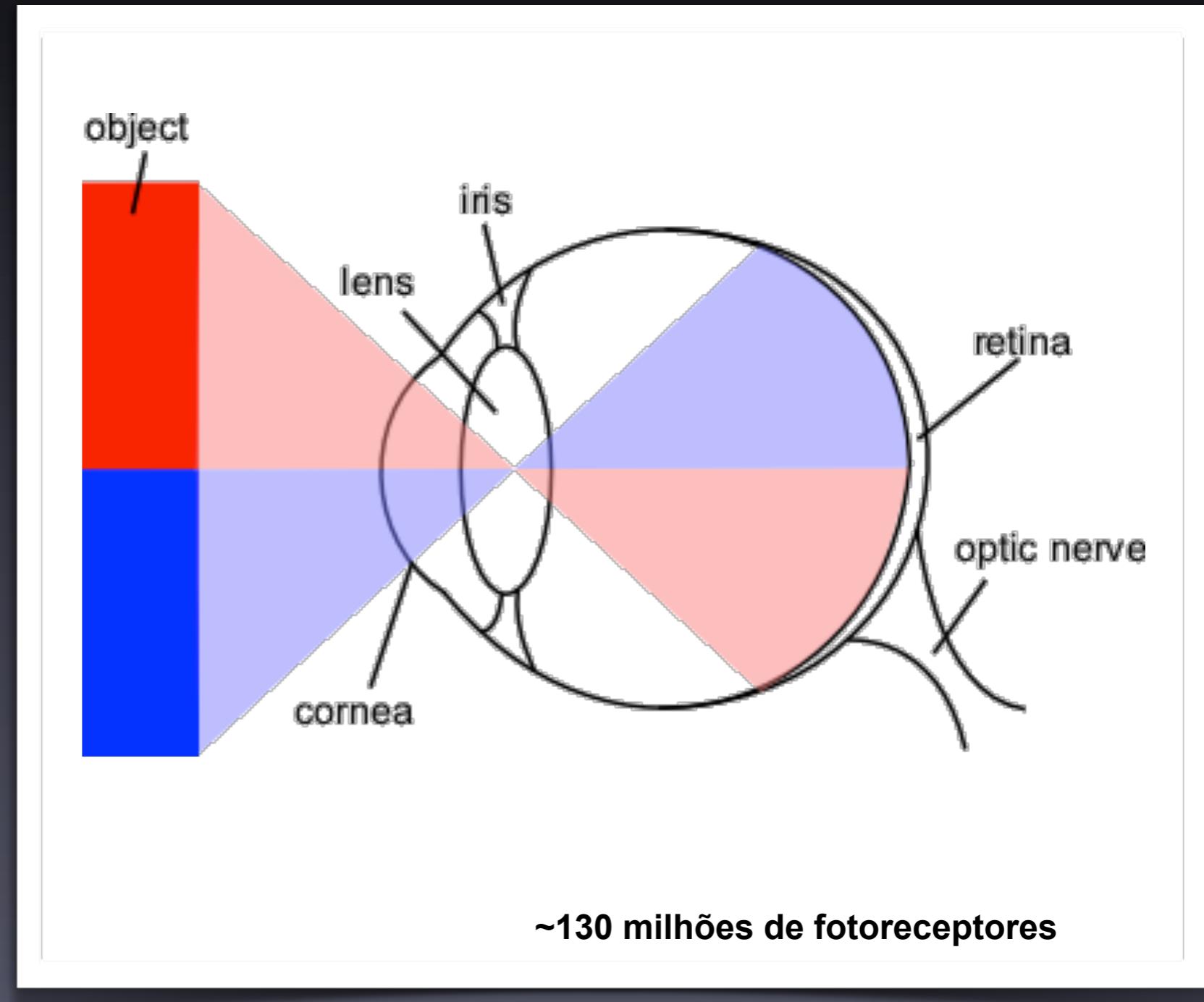


# Desvantagens do traçado de raios

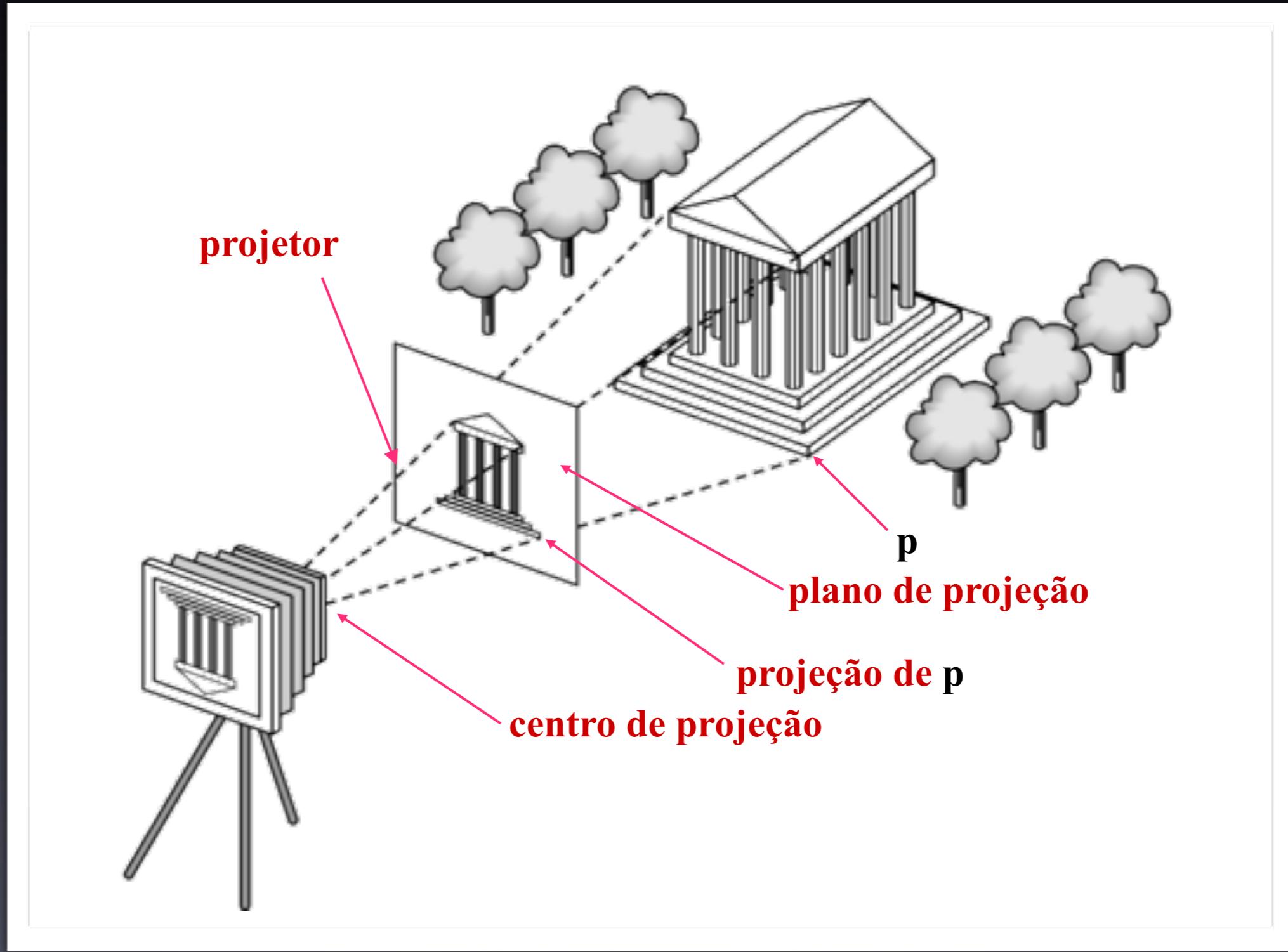
- Ray tracing parece modelar a formação de imagens utilizando conceitos físicos reais, por que então não usá-lo em um sistema gráfico?
  - É possível de ser implementado para objetos geométricos simples constituídos de polígonos e quádricos
  - Em princípio, pode reproduzir efeitos como sombras e reflexões múltiplas porém é um processo lento, o que impede o seu uso em aplicações interativas.

# Sistema visual humano

- Tem dois tipos de sensores
  - **Bastonetes**
    - Monocromáticos, visão noturna
  - **Cones**
    - Sensível a cores
    - Três tipos de cones
    - Somente três valores são enviados para o cérebro
- O processamento inicial da luz em humanos tem os mesmos princípios da maioria dos sistemas ópticos.



# Câmera sintética



# Vantagens

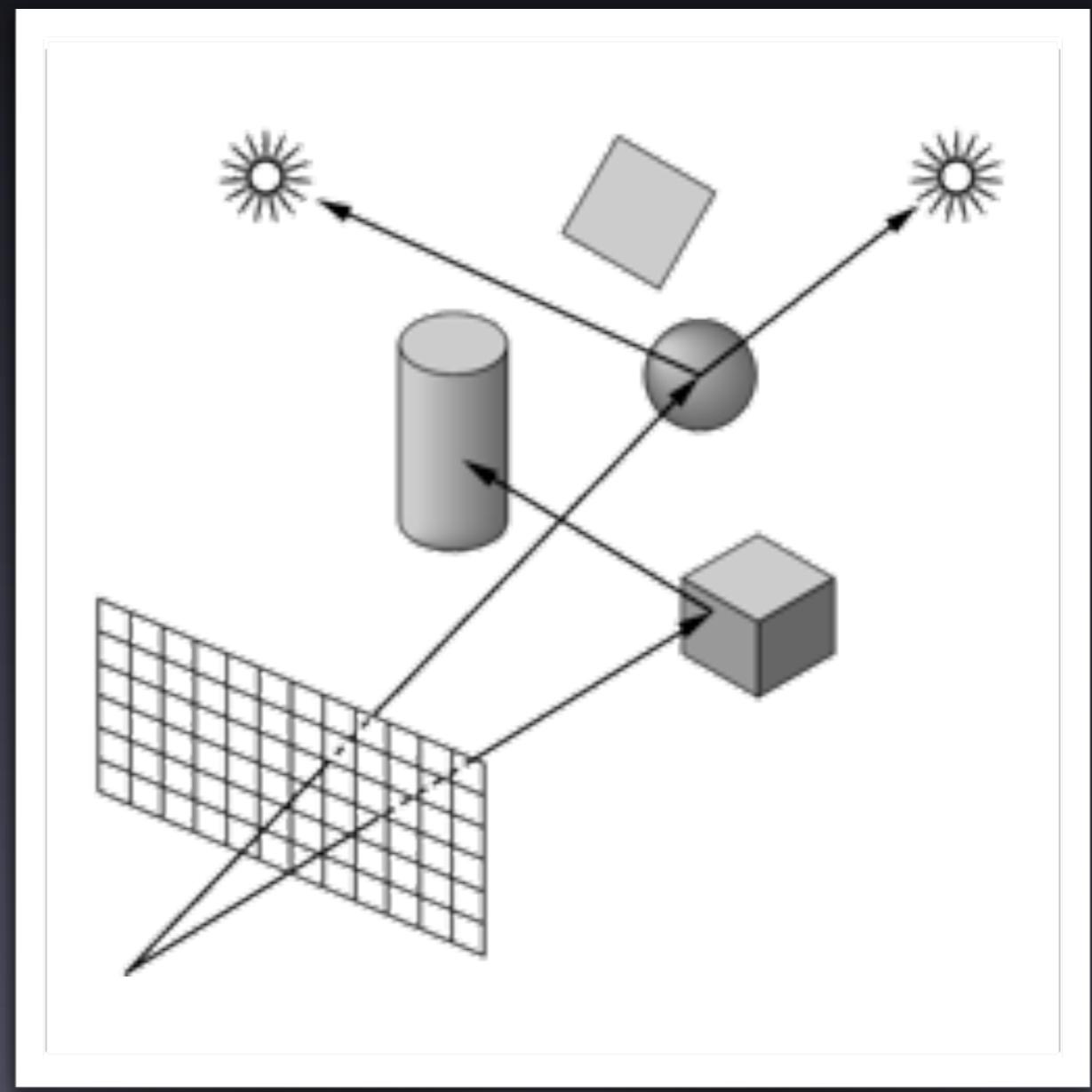
- Separação entre objetos, observador e fontes de iluminação
- Gráficos 2-D tornam-se um caso especial dos gráficos 3-D
- Implica em uma API software mais simples
  - Especificação de objetos, luzes, câmera e atributos
  - A implementação determina a imagem
- Implementação rápida em hardware

# Formação de imagens

- Como podemos utilizar o modelo de câmera sintética para criar aplicações em computação gráfica ?
- Application Programmer Interface (API)
  - Precisamos somente especificar
  - Objetos
  - Materiais
  - Observador
  - Luzes
- Como é sua implementação ?

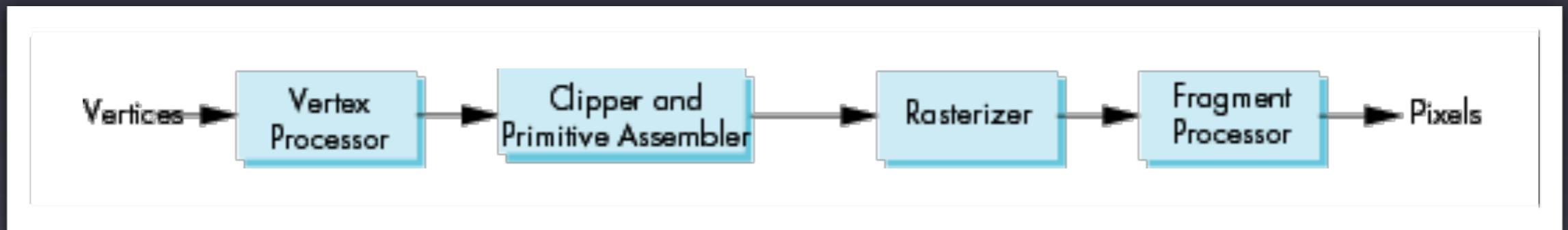
# Aproximações físicas

- **Ray tracing:** seguir raios a partir do centro de projeção até que eles sejam absorvidos pelos objetos ou sigam para o infinito
  - Efeitos globais
    - Reflexões múltiplas
    - Objetos translúcidos
  - Lento
  - Banco de dados deve estar disponível todo o tempo



# Renderização direta

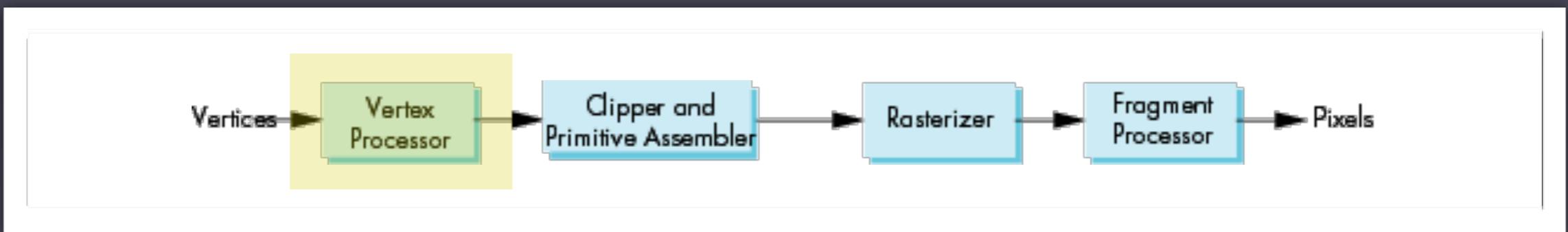
- Processa um objeto de cada vez, na ordem que são gerados pela aplicação
- Considera somente iluminação local
- Arquitetura do pipeline



- Todos estes passos podem ser implementados em hardware (ex: placa gráfica)

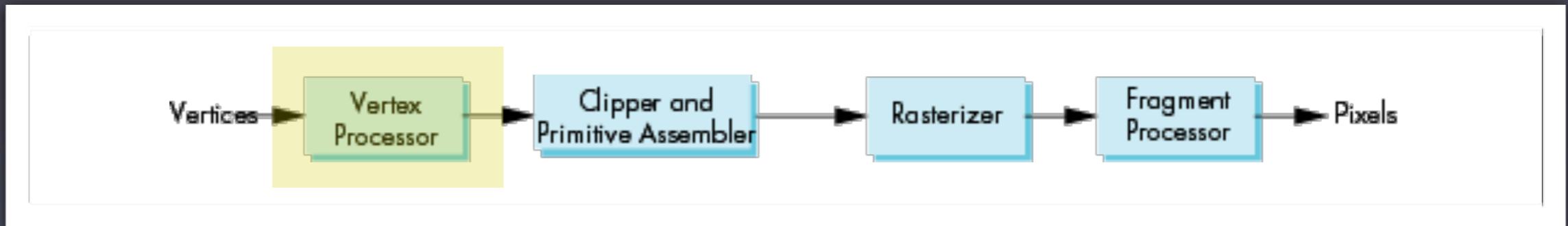
# Processamento de vértices

- A maioria do trabalho executado no pipeline diz respeito à conversão entre sistemas de coordenadas
  - Coordenadas dos objetos
  - Coordenadas da câmera (observador)
  - Coordenadas da tela
- Cada mudança é equivalente à uma matriz de transformação
- Também calcula cores dos vértices



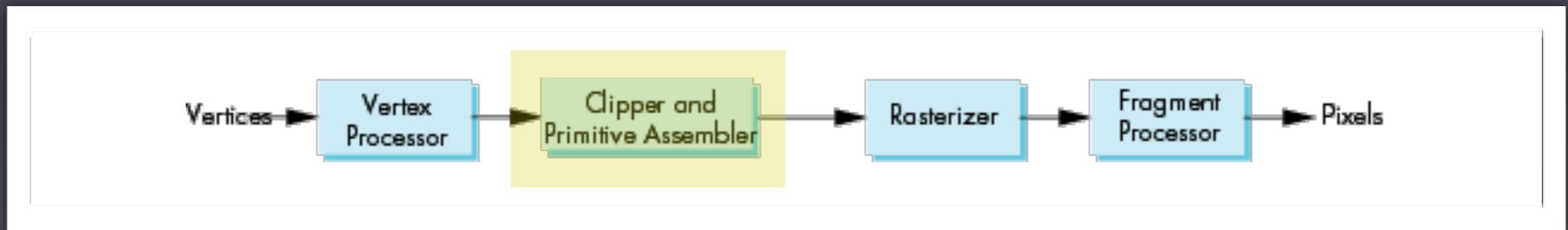
# Projeção

- É o processo que combina o observador em 3D e objetos a fim de produzir uma imagem em 2D
- Projeção perspectiva: todas as linhas de projeção se encontram no centro de projeção
- Projeção paralela: linhas de projeção são paralelas, e o centro de projeção é substituído pela direção de projeção



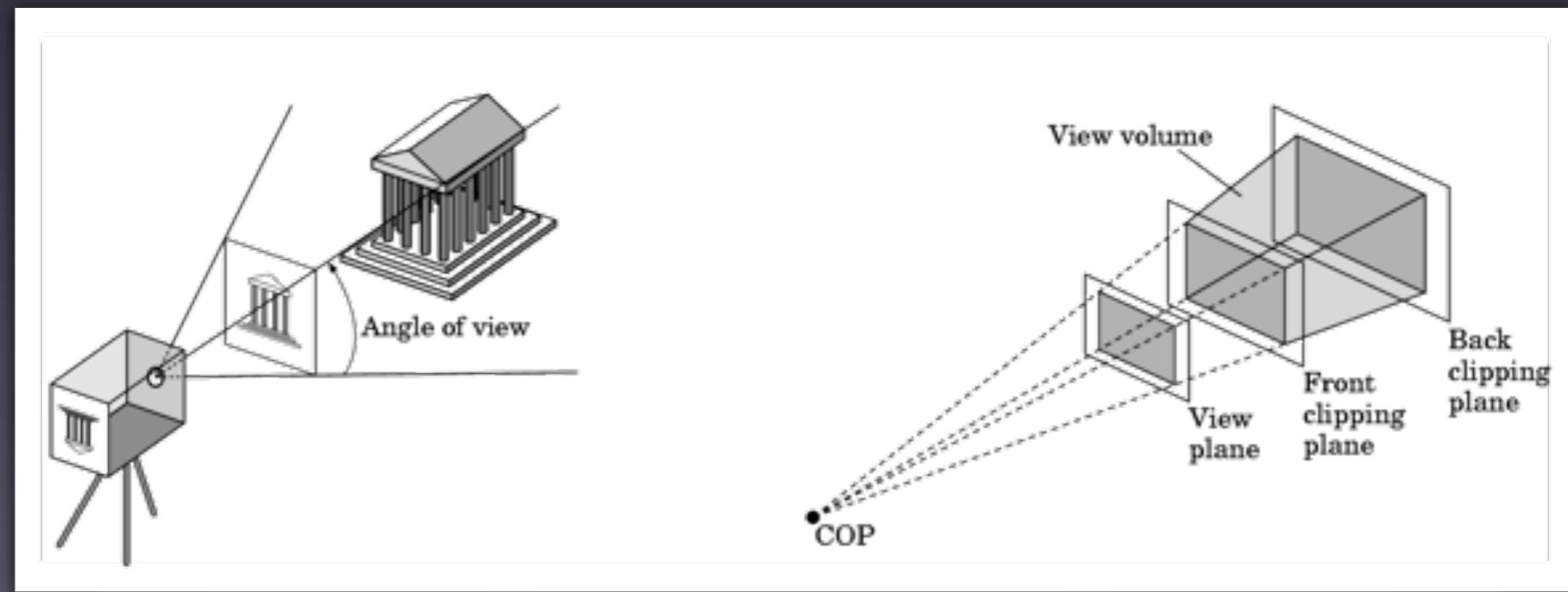
# Agrupamento de vértices

- Vértices devem ser agrupados em objetos geométricos antes das operações de recorte e rasterização
  - Segmentos de linha
  - Polígonos
  - Curvas and superfícies



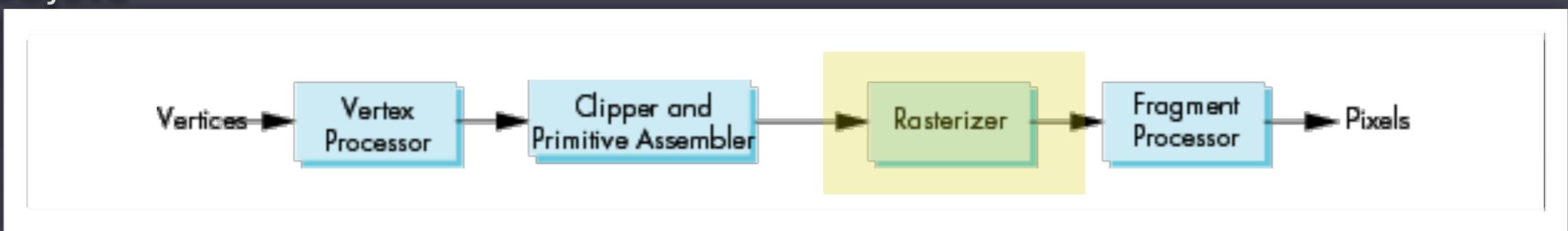
# Recortes

- Assim como uma câmera real não consegue “ver” todo o universo, uma câmera virtual pode somente enxergar parte dele também
- Objetos que não estejam neste volume são “recortados” da cena



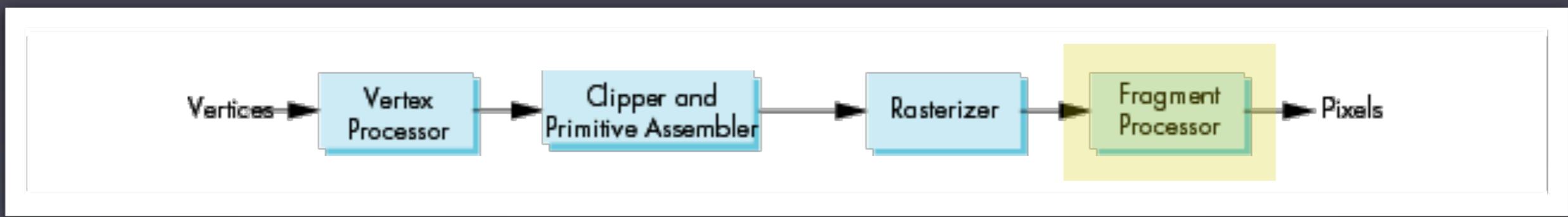
# Rasterização

- Se um objeto não é recortado, cores devem ser atribuídas aos seus pixels
- O processo de rasterização produz uma série de fragmentos para cada objeto
- Fragmentos são ditos “pixels potenciais”
  - Possuem uma posição no frame buffer
  - Possuem atributos cor e profundidade
- O rasterizador interpola atributos dos vértices para toda a região do objeto

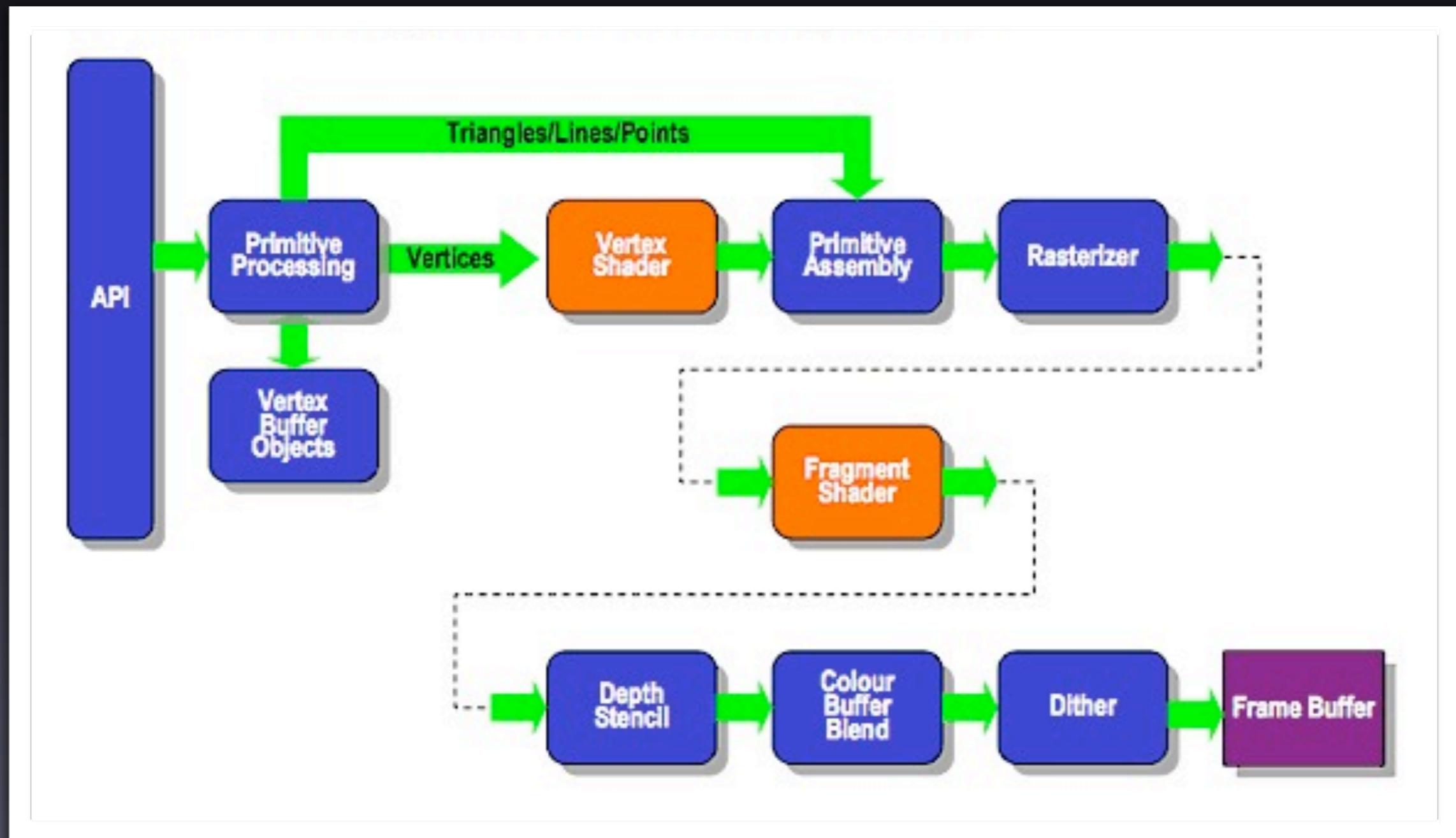


# Fragments

- Fragmentos são produzidos a fim de determinar a cor do pixel correspondente no frame buffer
- Suas cores são determinadas através de mapeamento de textura ou interpolação de cores entre vértices
- Fragmentos podem também estar sendo “bloqueados” por outros fragmentos mais próximos da câmera
  - Remoção de superfícies escondidas

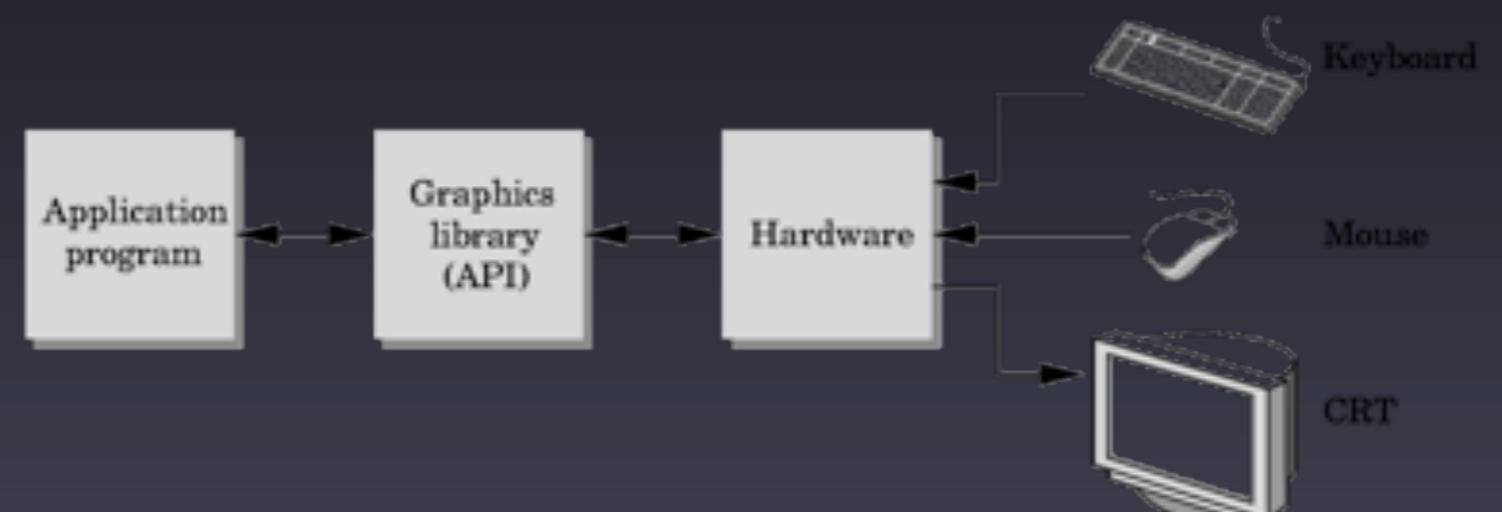


# Pipeline do OpenGL



# API gráfica

- Contém funções que especificam o que precisamos para formar uma imagem
  - Objetos
  - Observador
  - Fonte(s) de luz
  - Materiais
  - Outras informações
  - Entrada de dispositivos como mouse e teclado
  - Capacidades do sistema



# Especificação de objetos

- A maioria das APIs suportam um número limitado de primitivas
  - Pontos (objeto 0D)
  - Segmentos de linha (1D objetos 1D)
  - Polígonos (objetos 2D)
  - Algumas curvas e superfícies
  - Quádricas
  - Polinômios (paramétricos)
- São todos definidos através de coordenadas no espaço (vértices)

# Especificação de objetos

```
glBegin(GL_POLYGON)
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 1.0);
glEnd();
```

tipo do objeto

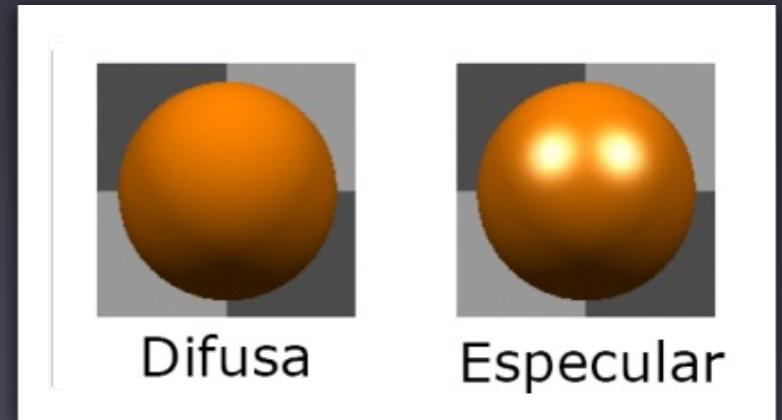
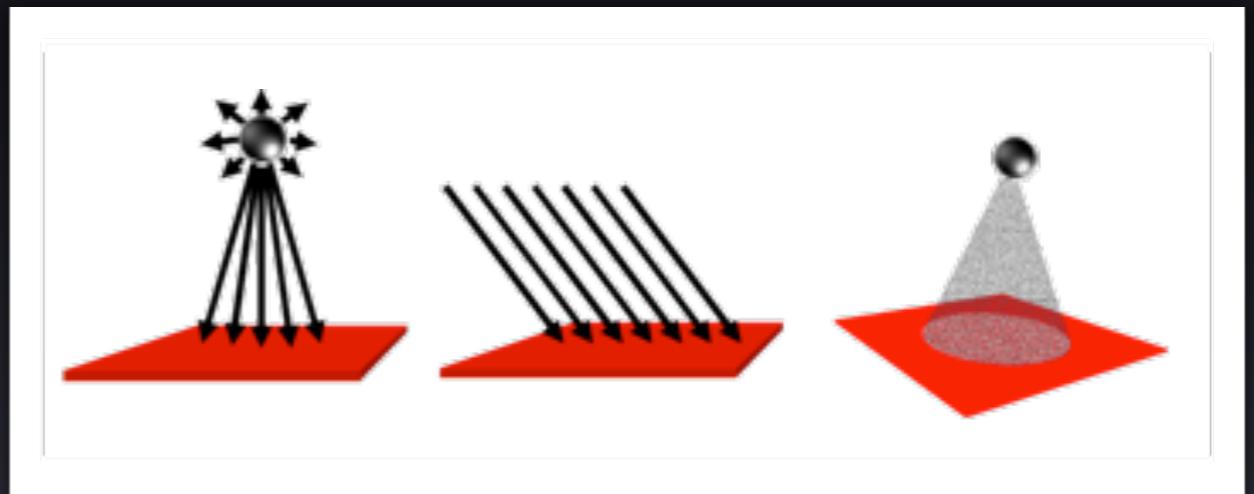
coordenadas do vértice

fim da definição

OpenGL

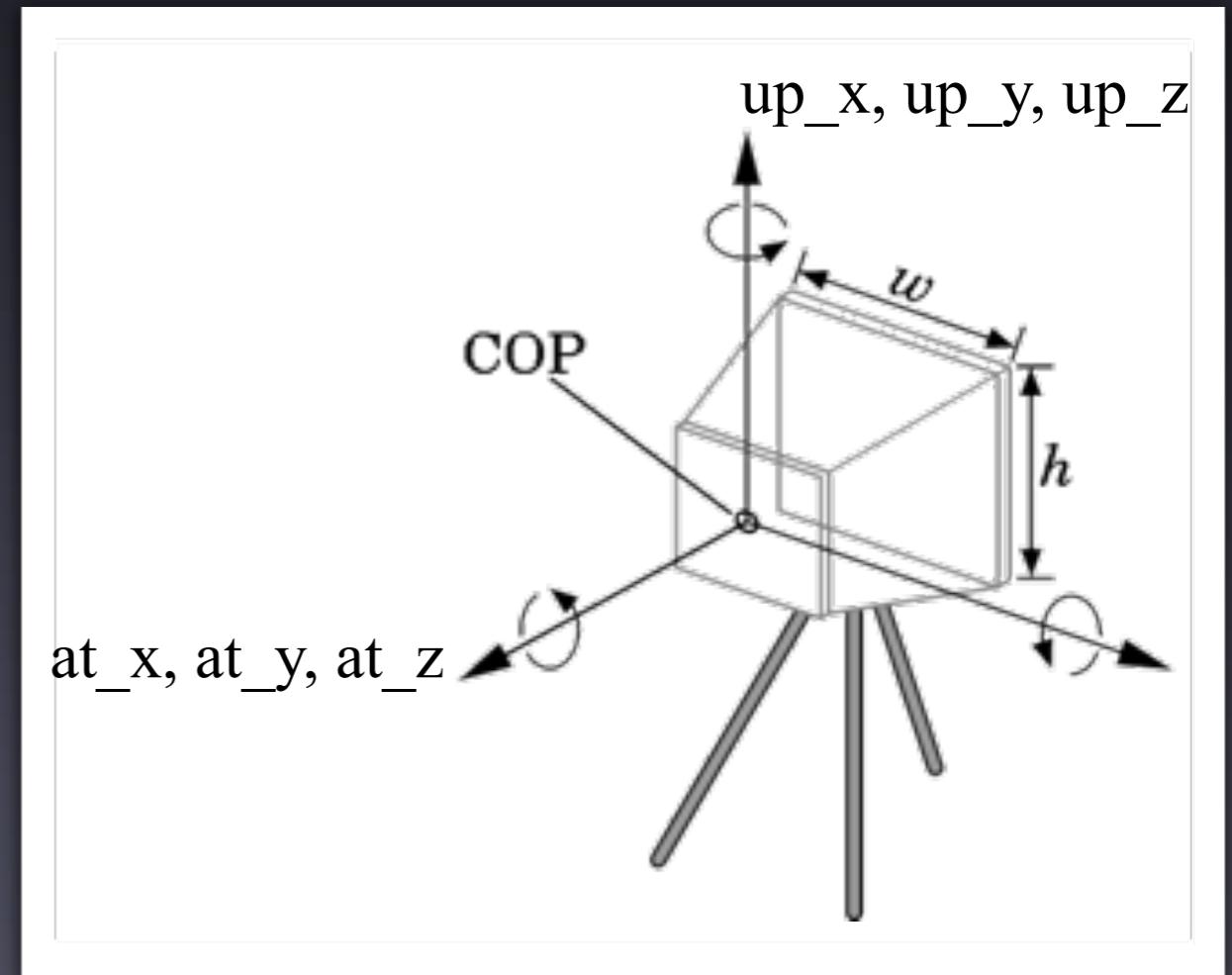
# Luzes e Materiais

- Tipos de luzes
  - Puntual
  - Direcional
  - Tipo “spot”
- Propriedades de materiais
  - Absorção: propriedades de cor
  - Reflexão
    - Difusa (Lambertiana)
    - Especular (“ponto de brilho”)



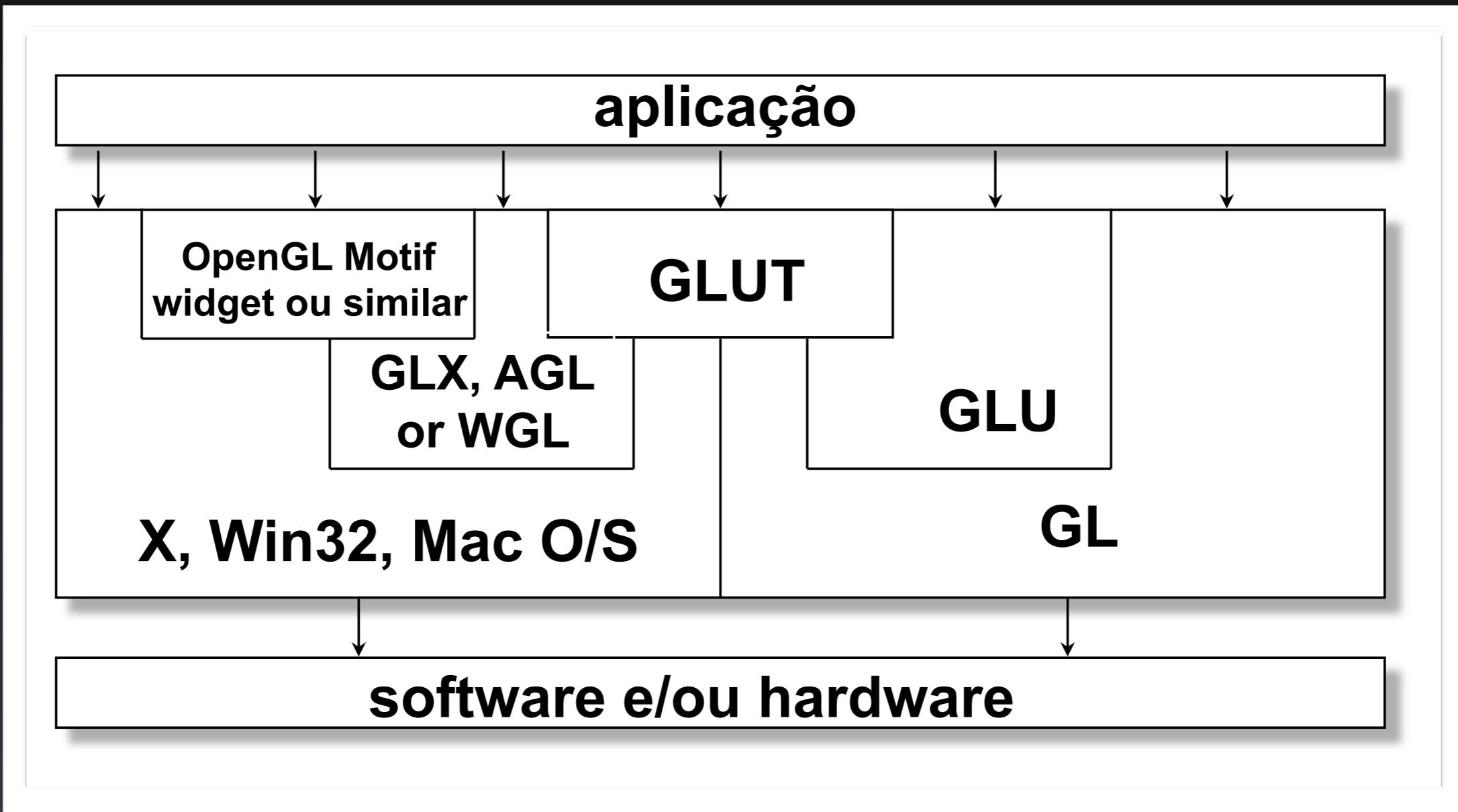
# Especificação da câmera

- Seis graus de liberdade
  - Posição do centro da lente (COP)
  - Orientação
- Lente
- Tamanho do filme



**gluLookAt(cop\_x, cop\_y, cop\_z, at\_x, at\_y, at\_z, up\_x, up\_y, up\_z)**

# Estrutura do OpenGL



# Máquina de estados

- OpenGL é uma máquina de estados
- Existem somente 2 tipos de funções em OpenGL:
  - **Geração de primitivas**
    - Podem gerar saída caso a primitiva seja visível
    - A maneira como os vértices são processados e a sua aparência final são controlados pelo estado
  - **Mudança de estado**
    - Funções de transformação
    - Funções de mudança de atributos (ex: glColor3f)

# Não é O-O

- Em OpenGL existem múltiplas funções para cada operação lógica
  - **glVertex3f**
  - **glVertex2i**
  - **glVertex3dv**
  - **glColor4f**
- Facilidade em criar funções sobre carregadas em C++ mas pode provocar perda de eficiência

# Formato das funções

`glVertex3f(x, y, z)`

nome da função  
dimensões  
`x, y, z` são **floats**

pertencente à biblioteca GL

`glVertex3fv(p)`  
`p` é um ponteiro para um array

# Tipos de dados

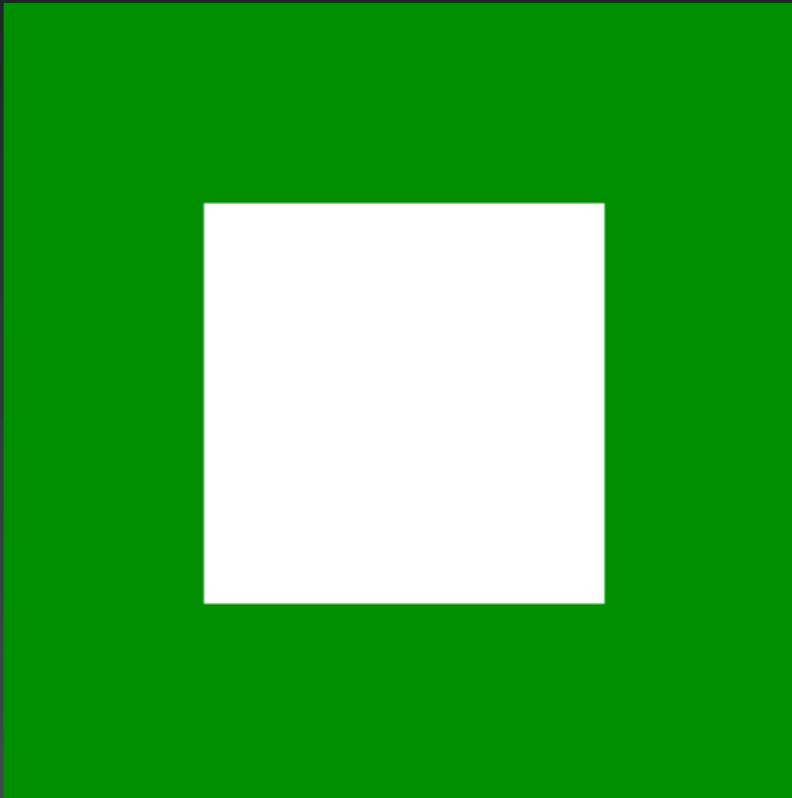
- Funções em OpenGL esperam tipos específicos de dados. Para indicar um destes tipos, OpenGL usa nomes especiais, como:
  - **GLbyte, GLshort, GLint, GLfloat, GLdouble, GLboolean**
- Estes tipos são normalmente mapeados para tipos equivalentes na linguagem C
  - Não existe nenhuma garantia que essa equivalência sempre aconteça.

# Exemplo: simple.c

```
#include <GL/glut.h>

void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glFlush();
}

int main(int argc, char** argv)
{
    glutCreateWindow("simple");
    glutDisplayFunc(mydisplay);
    glutMainLoop();
}
```



# Demo simple.c

# Loop de eventos

- Note que o programa define uma função “callback” de display chamada **mydisplay**
  - Todo programa glut deve ter uma callback de display
  - Esta função é executada toda vez que o OpenGL decide que a tela/janela deve ser redesenhada (ex. quando a janela é aberta)
- A função main termina com o programa entrando no loop de eventos

# Bibliotecas relacionadas

- OpenGL Utility (**GLU**) library
  - Rotinas para selecionar e modificar matrizes de visualização e projeção
  - Descrição de objetos mais complexos via aproximações (linhas e polígonos)
  - Visualização de quádricas e B-splines utilizando aproximações lineares
  - Funções para renderização de superfícies
  - Etc
- Toda implementação OpenGL inclui a biblioteca GLU, e todos os seus nomes de funções iniciam com o prefixo **glu**.

# Estrutura

- A maioria dos programas em OpenGL possui uma estrutura similar que consiste das seguintes funções
- **main()**
  - define as funções de “callback”
  - abre uma ou mais janelas com as devidas propriedades
  - entra no laço de eventos (última instrução)
- **init(): atribui valores às variáveis de estado**
  - Visualização
  - Atributos
- **“Callbacks”**
  - Função de display
  - Entrada, funções de gerenciamento de janelas

# simple1.c

```
#include <GL/glut.h>           ← inclui gl.h

int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0,0);
    glutCreateWindow("simple");   define propriedades da janela
    glutDisplayFunc(mydisplay);

    init();                     ← inicializa estado do OpenGL
    glutMainLoop();             ← callback de display
}                           ← entra laço de eventos
```

# Função: init()

```
void init()
{
    glClearColor (0.0, 0.0, 0.0, 1.0);           ← opaca

    glColor3f(1.0, 1.0, 1.0); ← seleciona cor branca

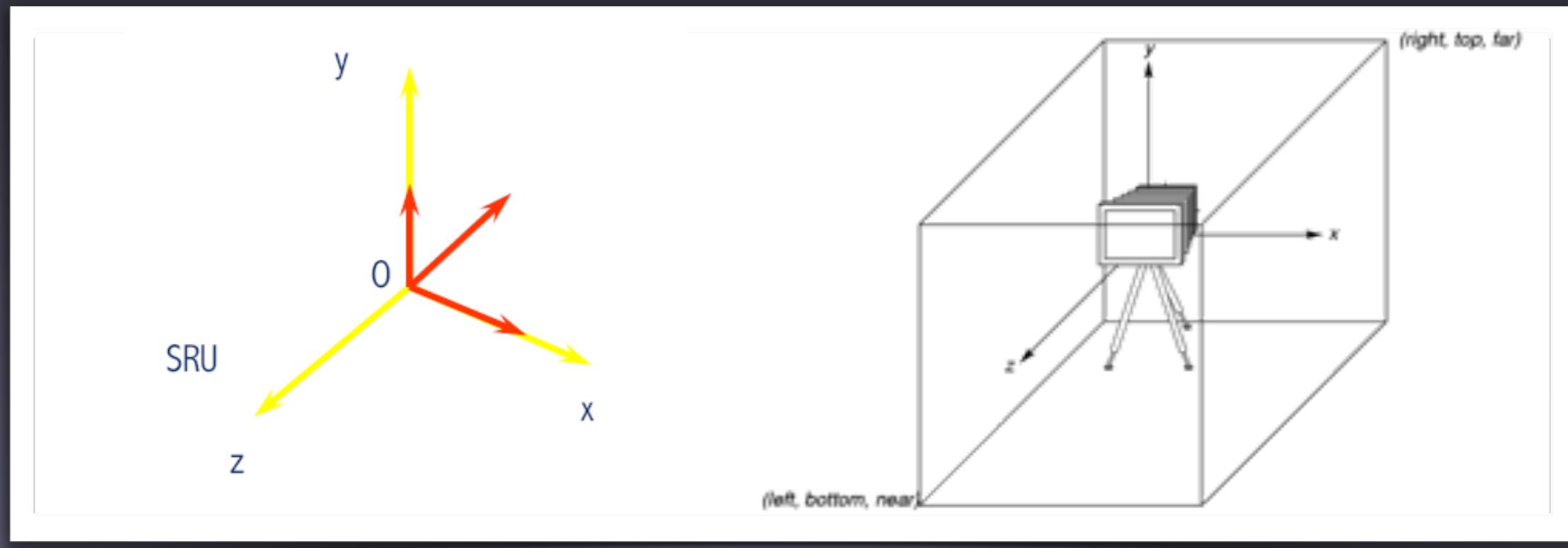
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

← define volume de visualização

# Demo simple1.c

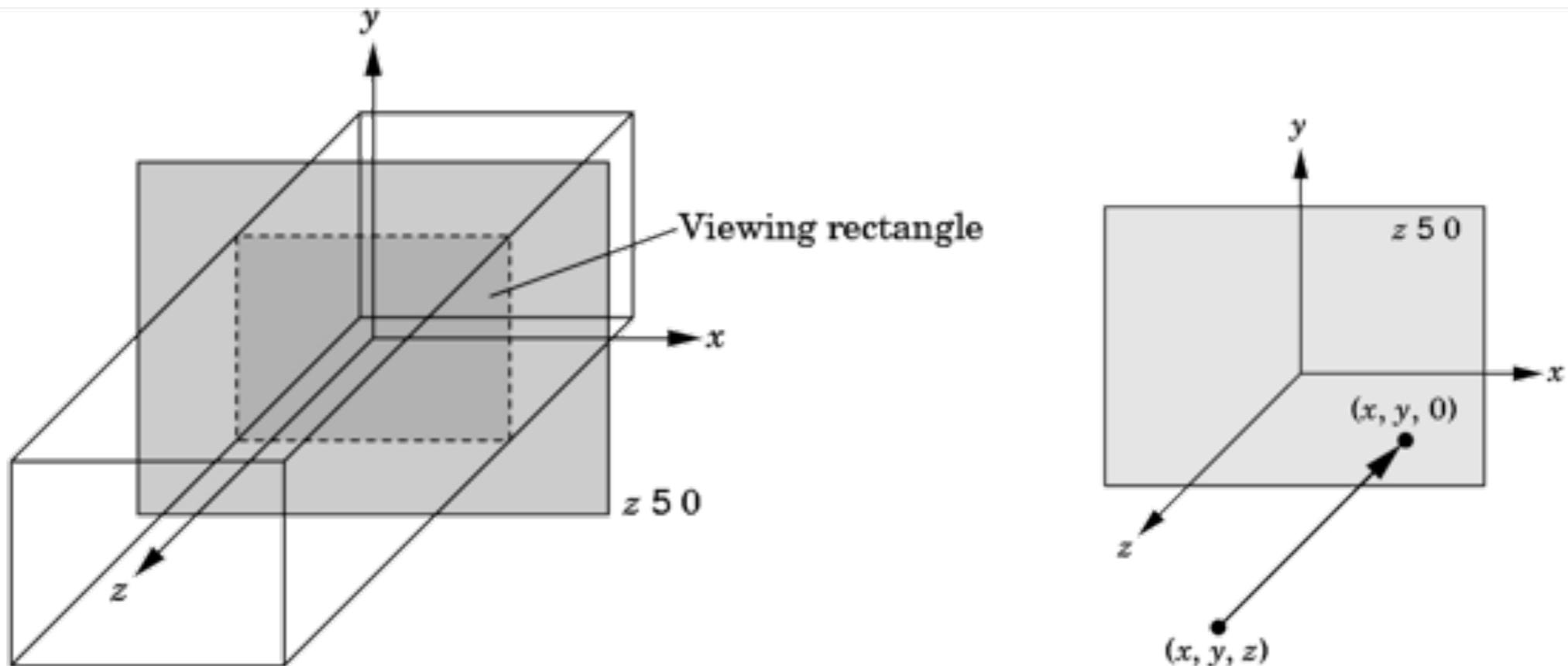
# Câmera em OpenGL

- OpenGL posiciona a câmera na origem do espaço dos objetos apontando na direção negativa de z
- O volume de visualização default é um cubo centrado na origem com lado medindo 2.



# Projeção ortográfica

- Na projeção ortográfica, pontos são projetados ao longo do eixo z no plano definido por  $z=0$



# Transformações de visualização

- Projeções são implementadas por matrizes (transformações)
- Existe somente um único conjunto de funções de transformação

**glMatrixMode(GL\_PROJECTION);**

- Tais funções operam de modo incremental
- Primeiramente, começamos com a matriz identidade
- Alteramos com a matriz de projeção que define o volume de visualização

**glLoadIdentity();**

**glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);**

# Visualização 2D e 3D

- Em **glOrtho(left, right, bottom, top, near, far)** e as distâncias “near” e “far” são medidas a partir da câmera
- Vértices em 2D são posicionados no plano z=0
- Caso uma aplicação opere em 2D, podemos usar a função

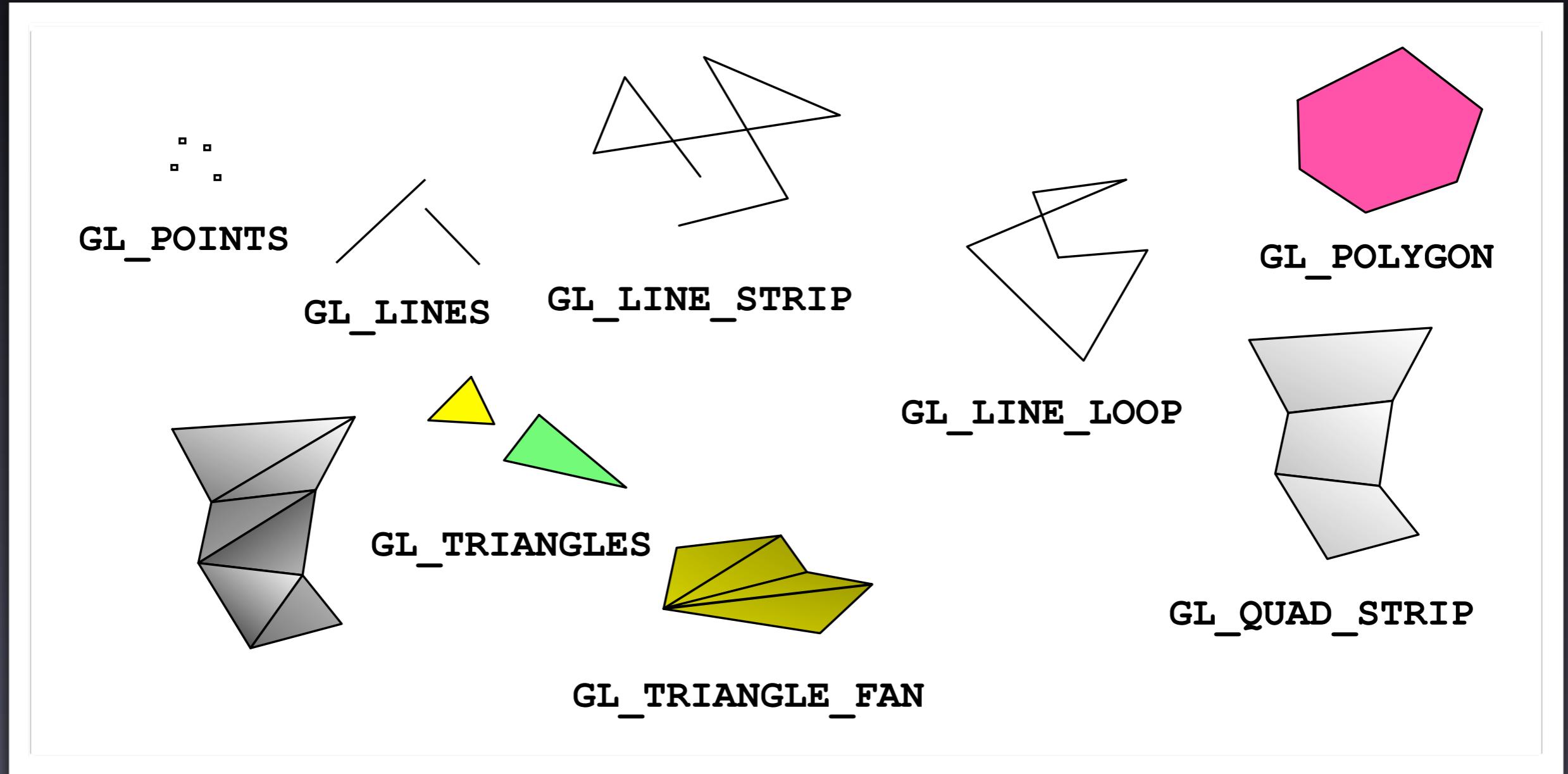
**gluOrtho2D(left,right,bottom,top)**

- Em 2D, o volume de visualização se torna uma “janela” de visualização

# mydisplay

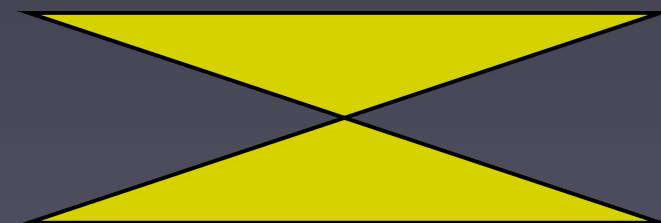
```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT) ;
    glBegin(GL_POLYGON) ;
        glVertex2f(-0.5, -0.5) ;
        glVertex2f(-0.5, 0.5) ;
        glVertex2f(0.5, 0.5) ;
        glVertex2f(0.5, -0.5) ;
    glEnd() ;
    glFlush() ;
}
```

# Primitivas geométricas



# Triângulos

- OpenGL somente desenhará corretamente polígonos que são:
  - Simples:** arestas não se cruzam
  - Convexos:** todos os pontos em um segmento de linha entre dois pontos dentro do polígono também estão dentro do polígono
  - Planares:** todos os vértices estão no mesmo plano
- Programa deverá verificar estas condições
  - OpenGL produzirá uma saída mesmo se tais condições forem violadas
- Triângulos satisfazem todas as condições!



Polígono complexo



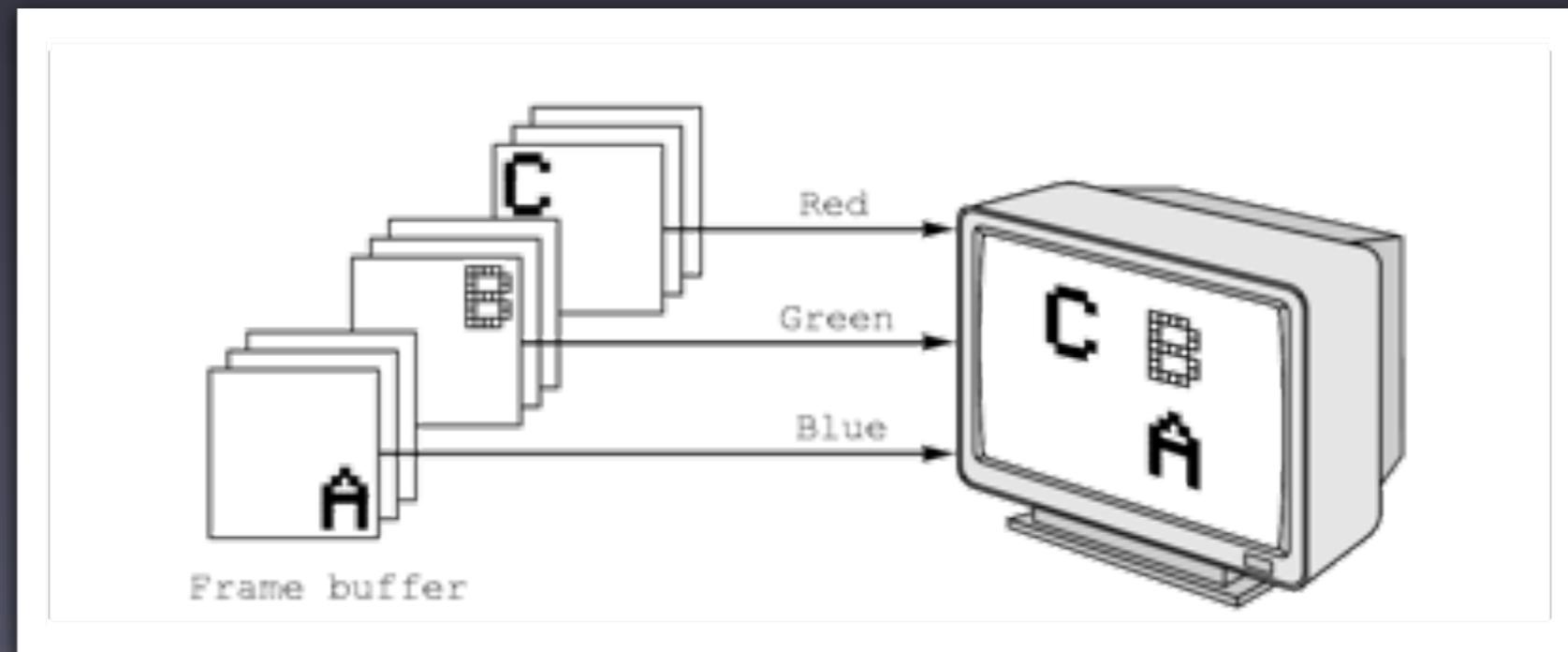
Polígono côncavo

# Atributos

- Atributos são parte do estado OpenGL e determinam a aparência dos objetos
  - **Cor** (pontos, linhas, polígonos)
  - **Tamanho** (pontos, linhas)
  - **Padrão de “stipple”** (linhas, polígonos)
  - Modo de exibição de polígonos
    - **Modo Preenchido** (cor sólida ou padrão stipple)
    - **Modo Arestas** (wireframe)
    - **Modo Vértices** (pontos)

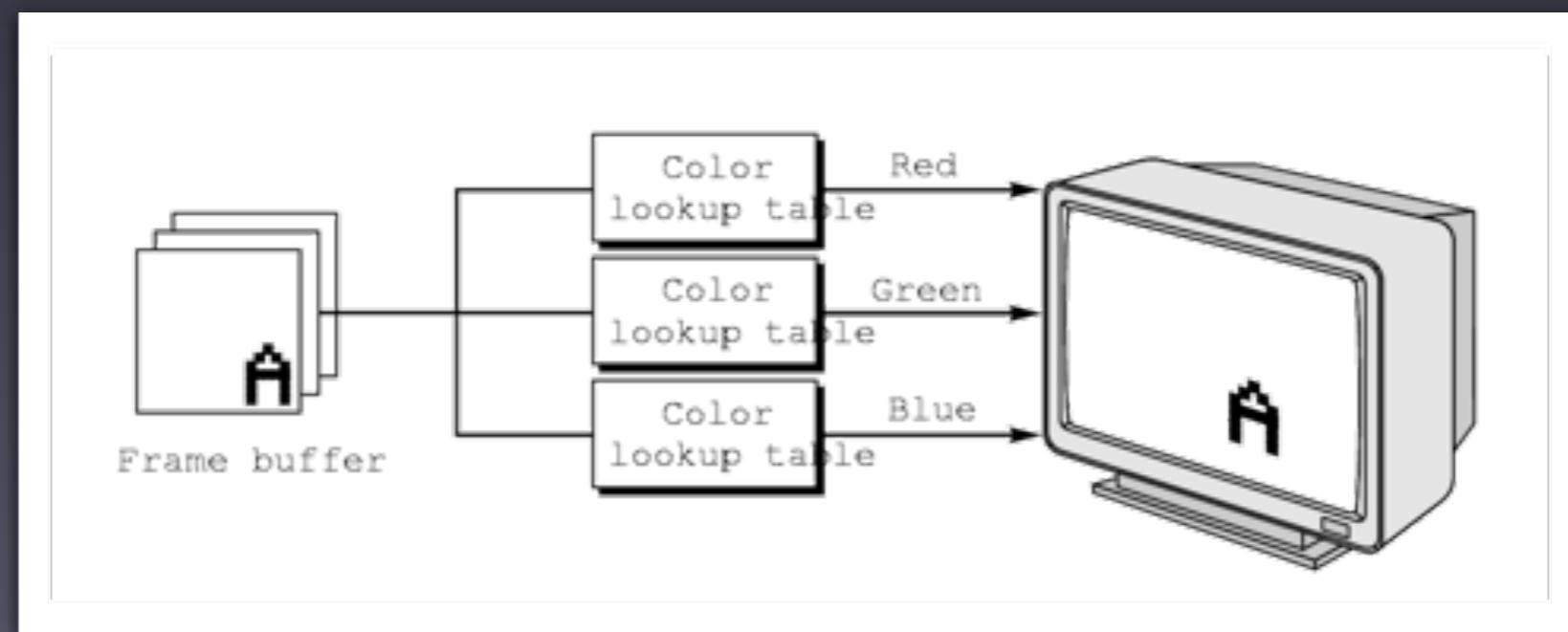
# Cores

- Cada componente de cor é armazenado individualmente no frame buffer
- Normalmente existem 8 bits por componente no buffer
- Note que em **glColor3f** os valores variam de 0.0 a 1.0, e em **glColor3ub** variam de 0 a 255



# Cores indexadas

- Cores se tornam índices para tabelas contendo valores RGB
- Requer menos memória
  - Índices são normalmente 8 bits
  - Hoje em dia, não tão importantes
  - Memória mais barata
  - Necessidade de uma paleta maior para tonalização



# Cores e estados

- A cor selecionada por **glColor** permanece parte do estado e continuará a ser usada até que seja alterada novamente
- Cores e outros atributos não são partes do objeto em si, mas são atribuídos quando o objeto é renderizado
- Podemos atribuir diferentes cores para diferentes vértices da seguinte forma:

```
glColor
```

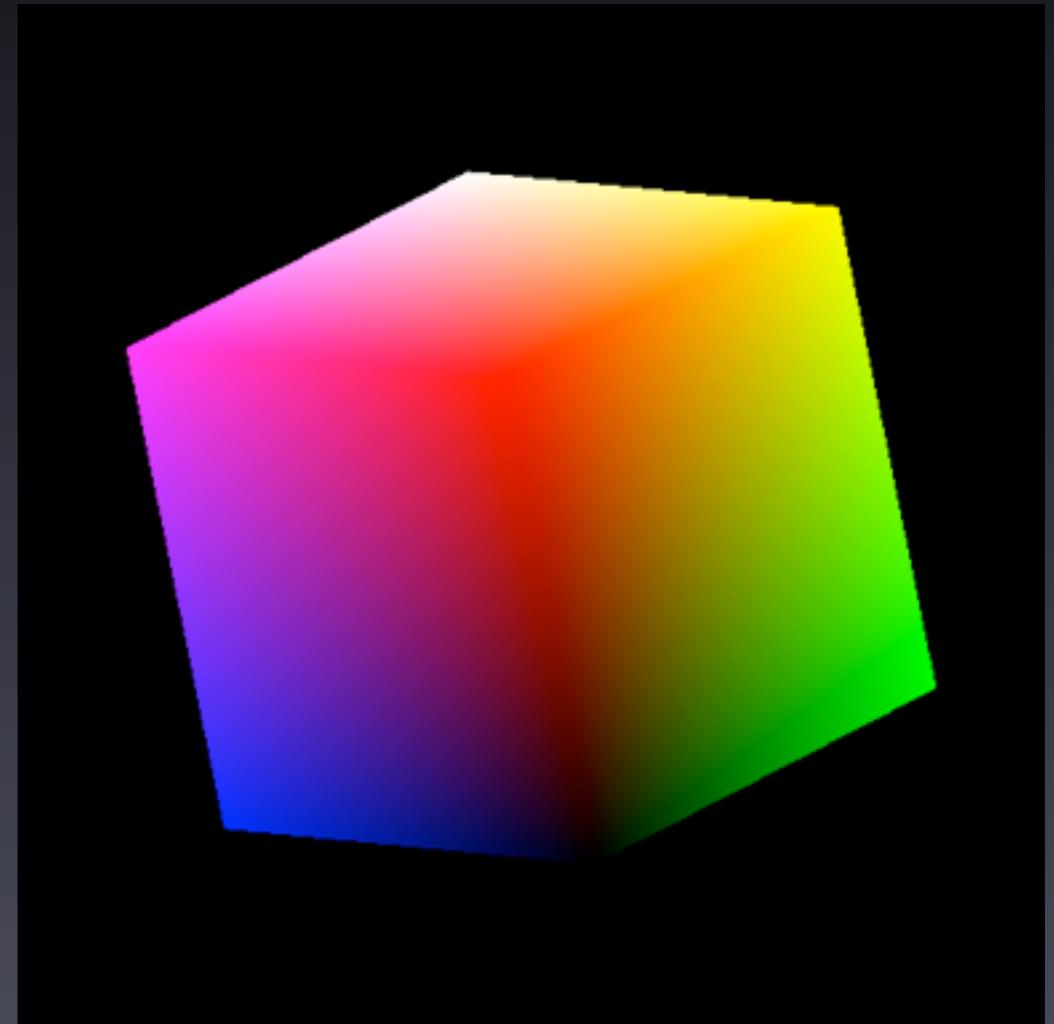
```
glVertex
```

```
glColor
```

```
glVertex
```

# Tonalização

- O default é a tonalização suave (**GL\_SMOOTH**)
- OpenGL interpola valores dos vértices ao longo dos polígonos visíveis
- Alternativa é a tonalização facetada (**GL\_FLAT**)
- Cor do primeiro vértice determina a cor de preenchimento



**colorcube.c**

**glShadeModel(GL\_SMOOTH  
ou GL\_FLAT)**

# Tarefa #2

- Dados  $n$  vértices  $V_i = \{(x_i, y_i), i=1, \dots, n\}$  de um polígono, com arestas ligando os vértices  $\{(x_{i-1}, y_{i-1}), (x_i, y_i), i=\{2, n\}\}$  e  $\{(x_n, y_n), (x_i, y_i), i=1\}$ .
  - Como testar se ele é convexo ou côncavo ?
- Respostas no fórum de discussão da disciplina.