Introdução à Computação Gráfica

Marcel P. Jackowski
mjack@ime.usp.br

Aula #5: Transformações (Parte 2)

Objetivos

- Subdivisão do círculo/esfera
- Mudança entre sistemas de coordenadas
- Especificação da câmera em OpenGL
- Matriz de modelo (GL_MODELVIEW)
- Projeções (ortográfica e perspectiva)
  - Matriz de projeção (GL_PROJECTION)
- Sólidos Platônicos

Sistemas de coordenadas

- Um sistema de coordenadas para $\mathbf{R}^n$ é definido por um ponto (origem) e *n* vetores
- Ex. Seja um sistema de coordenadas para $\mathbf{R}^2$ definido pelo ponto *O* e os vetores *X* e *Y*, e denomina-se OXY
- Um ponto P é dado por coordenadas x_P e y_P tais que

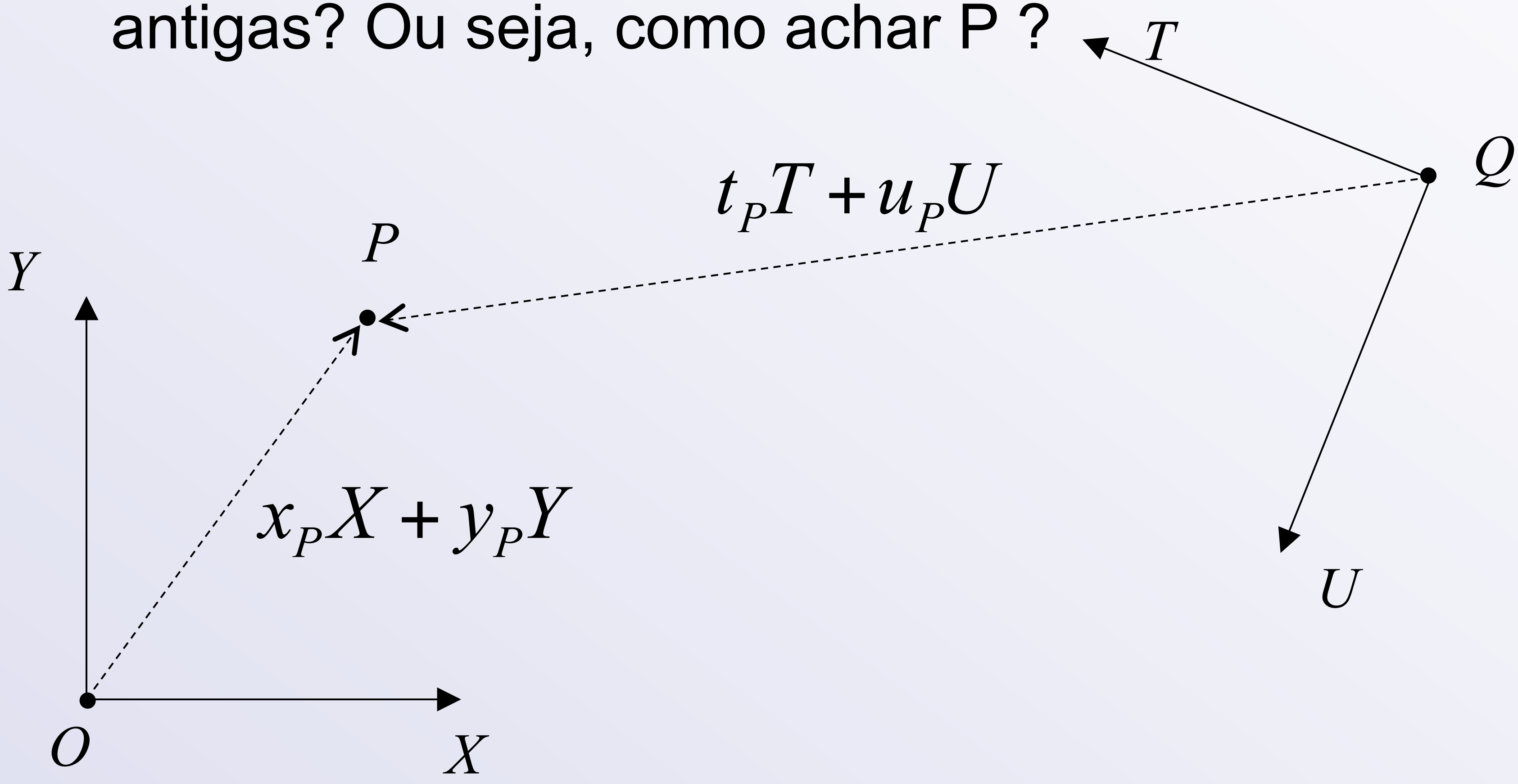
$$P = x_P . X + y_P . Y + O$$

- Um vetor *V* é dado por coordenadas x_V e y_V tais que

$$V = x_V . X + y_V . Y$$

Mudança entre sistemas de coordenadas

- Se estabelecermos um outro sistema (ex.: QTU), como calcular as novas coordenadas dadas as antigas? Ou seja, como achar P ?



Mudança de sistemas de coordenadas

- Como calcular as coordenadas de um ponto P = (x_P, y_P) em OXY dadas as coordenadas de P em QTU, isto é, (t_P, u_P) ?

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

Logo,

$$x_P = t_P.x_T + u_P.x_U + x_Q$$

$$y_P = t_P.y_T + u_P.y_U + y_Q$$

Mudança de sistemas de coordenadas

- Matricialmente:

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

- Usando coordenadas homogêneas:

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

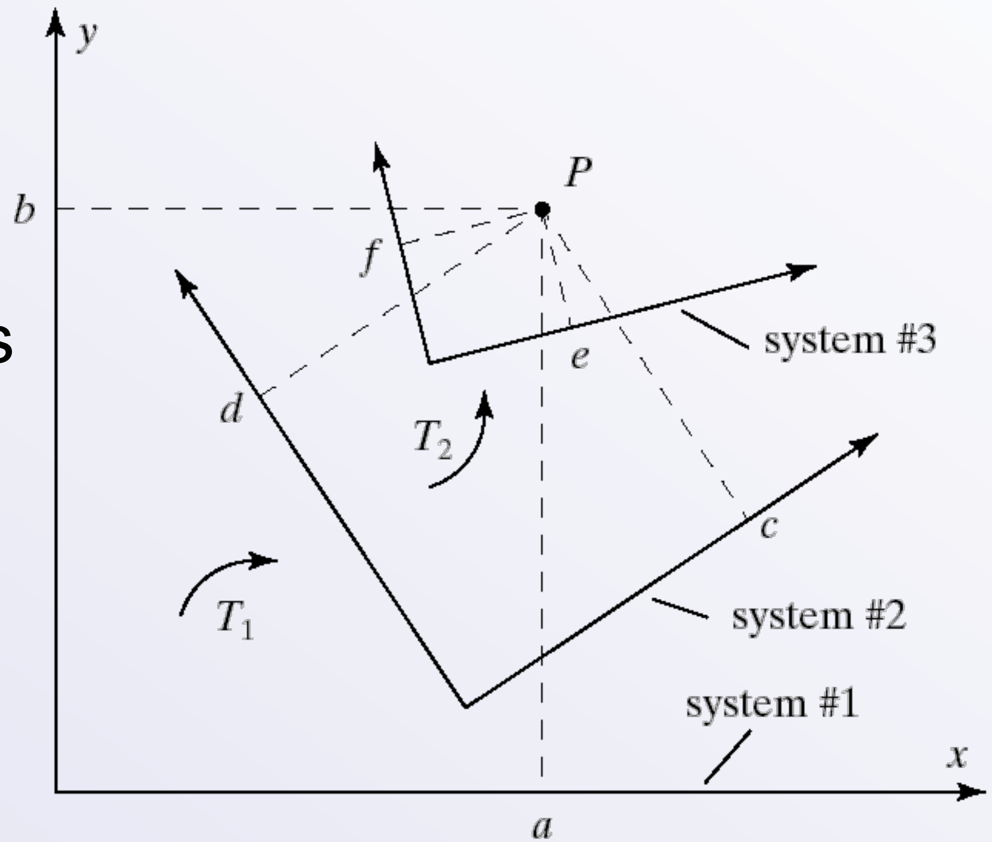
- Para resolver o problema inverso:

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

Mudança de sistemas de coordenadas

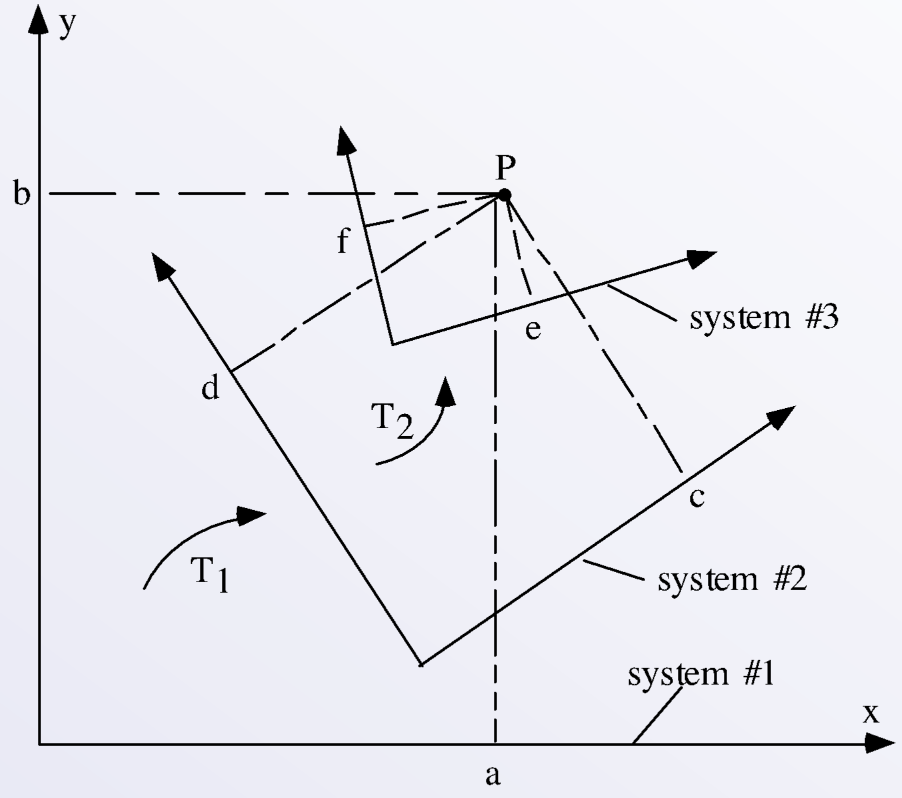
- Seja *P* um ponto com representação $(c, d, 1)^T$ no novo sistema #2.
- Quais são os valores de *a* e *b* na sua representação original $(a, b, 1)^T$ usando o sistema #1?
- Resposta:

$$(a, b, 1)^T = T_1 (c, d, 1)^T$$



Transformações sucessivas

- Ponto *P* possui representação $(e, f, 1)^T$ em relação ao sistema #3. Quais são as suas coordenadas $(a, b, 1)^T$ em relação ao sistema #1?



Transformações sucessivas

- No sistema #2, o ponto P possui coordenadas $(c, d, 1)^T = T_2(e, f, 1)^T$.
- No sistema #1 o ponto P possui coordenadas $(a, b, 1)^T = T_1(c, d, 1)^T$.
- Desta maneira:
  - $(a, b, 1)^T = T_1(c, d, 1)^T = T_1T_2(e, f, 1)^T$
- É importante observar que para determinar as coordenadas $(a, b, 1)^T$ a partir de $(e, f, 1)^T$, *primeiro* aplicamos T_2 e *depois* T_1.

Transformações sucessivas

- **Para transformar pontos.** Uma sequência de transformações $T_1()$, $T_2()$, $T_3()$ (nesta ordem) aplicadas em um ponto P, resultará na matriz M = M_3 x M_2 x M_1.
  - Então P é transformado por MP;
- **Para transformar sistemas de coordenadas.** Uma sequência de transformações $T_1()$, $T_2()$, $T_3()$ (nesta ordem) aplicadas em um sistema de coordenadas, resultará na matriz M = M_1 x M_2 x M_3.
  - Então P no sistema transformado possui coordenadas MP no sistema original.

Geometria Afim

- Composta dos elementos básicos
  - Valores escalares
  - Pontos: denotam posição
  - Vetores: denotam deslocamento (direção e magnitude)
- Operações
  - escalar · vetor = vetor
  - vetor + vetor ou vetor – vetor = vetor
  - ponto – ponto = vetor
  - ponto + vetor ou ponto – vetor = ponto

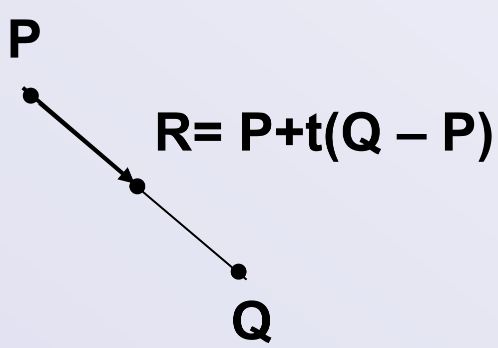
Combinações Afim

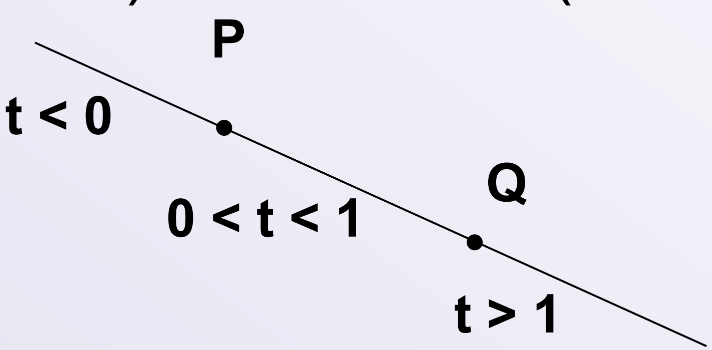
- Maneira especial de combinar pontos

$$\alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$$

$$\text{onde } \sum_{i=1}^{n} \alpha_i = 1$$

- Para 2 pontos *P* e *Q* poderíamos ter uma combinação afim *R* = (1 – t)*P* +t*Q* = *P* +t(*Q* – *P*)





Geometria Euclidiana

- Extensão da geometria afim pela adição de um operador chamado *produto interno*
- Produto interno é um operador que mapeia um par de vetores em um escalar. Tem as seguintes propriedades:
  - Positividade : $<u,u> > 0$ e $<u,u> = 0$ sse $u=0$
  - Simetria: $<u,v> = <v,u>$
  - Bilinearidade: $<u,v+w>= <u,v>+<u,w>$ e

$$<u,\alpha v>= \alpha<u,v>$$

Geometria Euclidiana

- Normalmente usamos o produto escalar como operador de produto interno:

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{d} u_i v_i$$

- Comprimento de um vetor é definido como:

$$\left|\vec{v}\right| = \sqrt{\vec{v} \cdot \vec{v}}$$

- Vetor unitário (normalizado):

$$\hat{v} = \frac{\vec{v}}{\left|\vec{v}\right|}$$

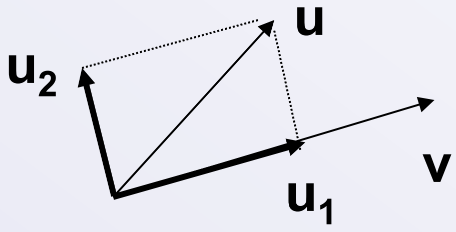
Geometria Euclidiana

- Distância entre dois pontos *P* e *Q* = |*P* – *Q*|
- O ângulo entre dois vetores pode ser determinado por

$$\hat{a}ngulo(\vec{u},\vec{v}) = \cos^{-1}\left(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}\right) = \cos^{-1}(\hat{u}\cdot\hat{v})$$

- Projeção ortogonal: dados dois vetores *u* e *v*, deseja-se decompor *u* na soma de dois vetores u_1 e u_2 tais que u_1 é paralelo a *v* e u_2 é perpendicular a *v*

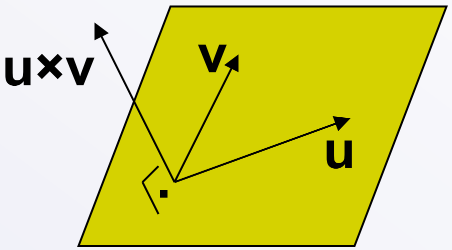
$$\vec{u}_1 = \frac{\vec{u}\cdot\vec{v}}{\vec{v}\cdot\vec{v}}\vec{v} \qquad \vec{u}_2 = \vec{u} - \vec{u}_1$$



Produto Vetorial (3D)

- Permite achar um vetor perpendicular a outros dois dados
- Útil na construção de sistemas de coordenadas

$$\vec{u} \times \vec{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$



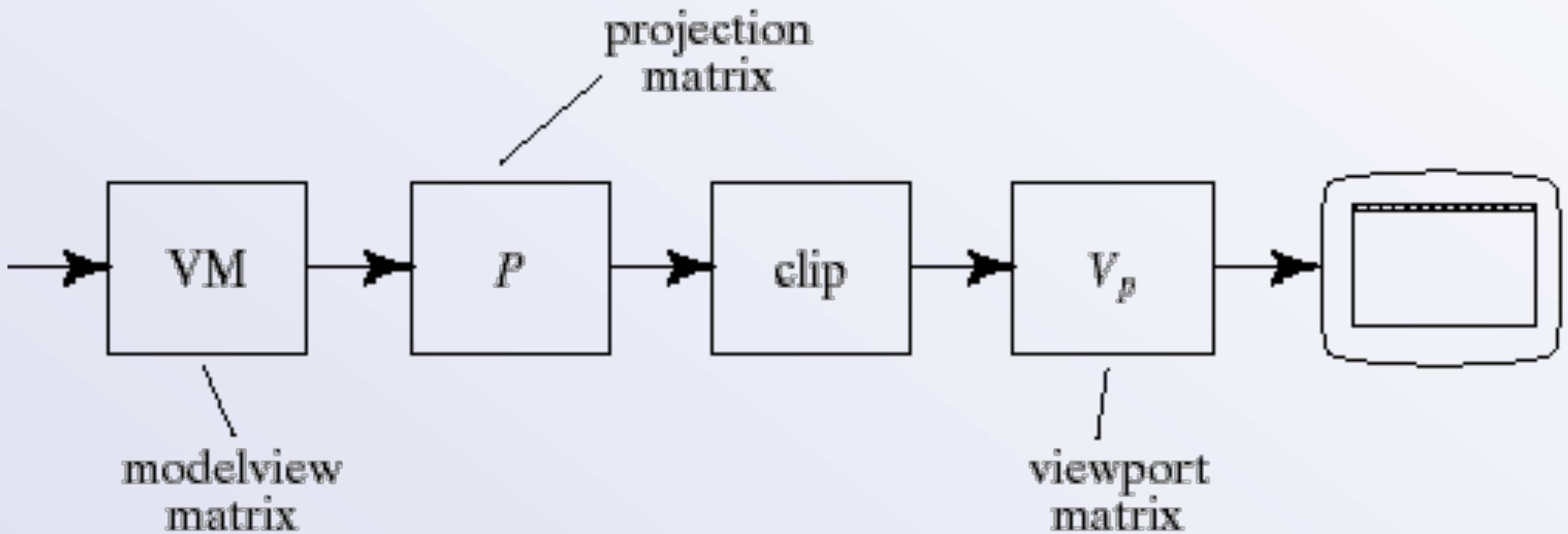
- **Propriedades (assume-se u, v linearmente independentes):**
  - Antisimetria: u × v = – v × u
  - Bilinearidade: u × (αv) = α (u × v) e u × (v + w) = (u × v) + (u × w)
  - u × v é perpendicular tanto a u quanto a v
  - O comprimento de u × v é igual a área do paralelogramo definido por u e v, isto é, | u × v | = | u | | v | sen θ

Visualização em 3D

- Existem três aspectos a serem considerados durante o processo de visualização, todos os quais já implementados no pipeline.
  - Posicionamento da câmera
    - Matriz de modelo
  - Selecionando a lente da câmera
    - Matriz de projeção
  - Recorte
    - Volume de visualização

Pipeline de visualização

- O OpenGL oferece funções para definir o volume de visualização e a câmera utilizando matrizes dentro do pipeline gráfico.



Configurando a cena

```
glMatrixMode(GL_MODELVIEW);
   // initializa matriz de modelo
glLoadIdentity();
   // configura matriz de visualização
   // configura transformações do modelo
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
   // configura tipo de projeção
   // glOrtho(), glFrustum() ou glPerspective()
```

Matrizes de visualização

- Cada vértice de um objeto é passado por este pipeline usando `glVertex3d(x,y,z).`
- O vértice é multiplicado por várias matrizes, recortado da cena caso necessário, e caso elesobreviva, é finalmente mapeado no viewport.
- Cada vértice encontra pela frente três matrizes de transformação:
  - A matriz de **modelo;**
  - A matriz de **projeção;**
  - A matriz de **viewport;**

A matriz de modelo

- A matriz de **modelo** (GL_MODELVIEW) é também chamada de matriz de transformação corrente T_C.
- Ela combina transformações de objetos e transformações que orientam e posicionam a câmera no espaço (por isso o nome *modelview*).
- Representada através de uma única matriz 4x4 dentro do pipeline.

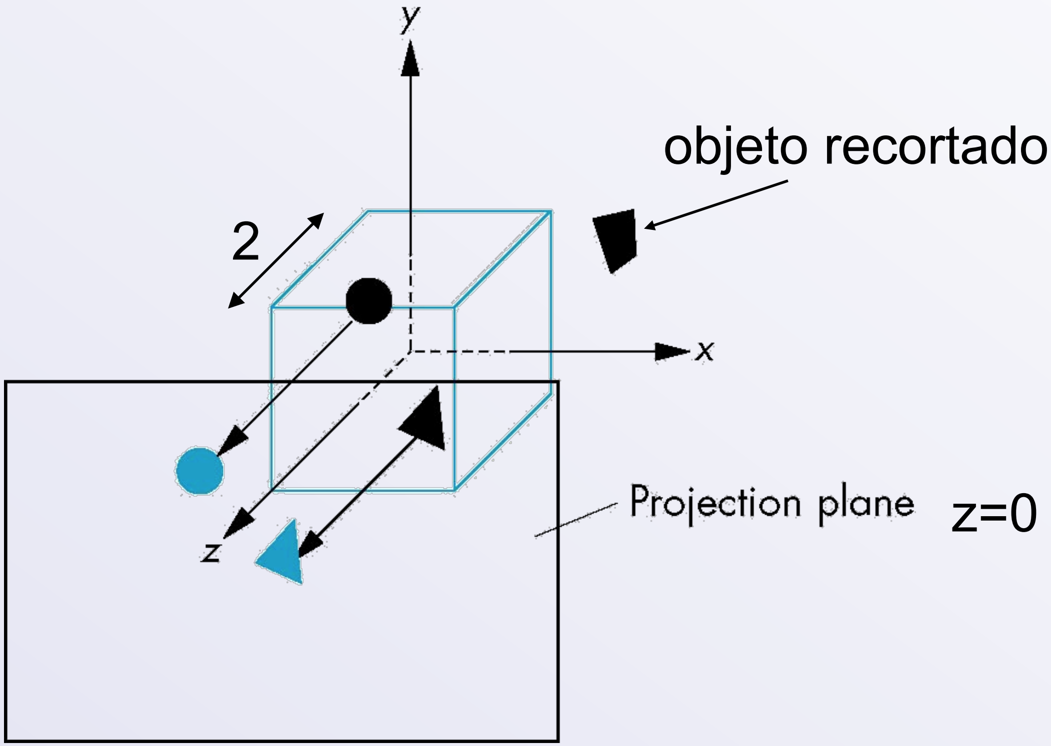
Podemos pensar que T_C é formada pelo produto de duas matrizes: uma matriz de modelo-objeto M, e uma matriz de visualização V. A matriz M é aplicada antes da matriz V, e desta forma a matriz ***modelo*** *se torna o produto VM.*

A câmera em OpenGL

- Em OpenGL, inicialmente o sistema de coordenadas da cena (SRU) e o sistema de coordenadas da câmera (SRC) são os mesmos.
  - A matriz de modelo default é a identidade.
- A câmera é posicionada na origem e aponta na direção negativa do eixo z
- OpenGL também especifica o volume default de visualização que é um cubo com lados de comprimento 2, centralizado na origem
  - A matriz de projeção default também é a matriz identidade.

Projeção default

A projeção default é a ortogonal.



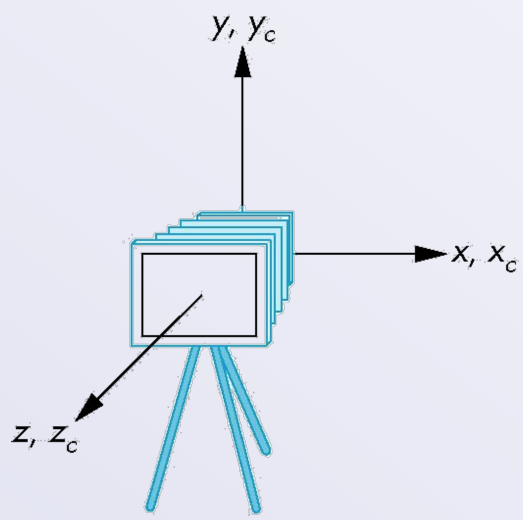
Movendo a câmera

- Caso queiramos visualizar um objeto com valores positivos e negativos de z, podemos:
  - (i) Mover a câmera na direção positiva de z
    - Translação da SRC
  - (ii) Mover o objeto na direção negativa de z
    - Translação da SRU
- Essas duas visões são equivalentes e determinadas pela matriz de modelo
  - Translação (`glTranslatef(0.0,0.0,-d);`)
  - `d > 0`

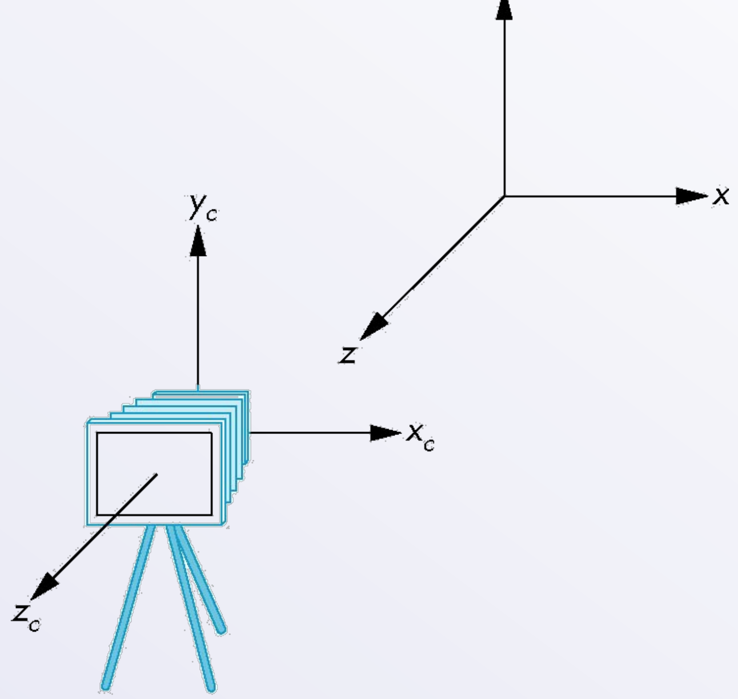
Tirando a câmera da origem

sistema após a translação por –d
d > 0

sistema default de
coordenadas
da câmera



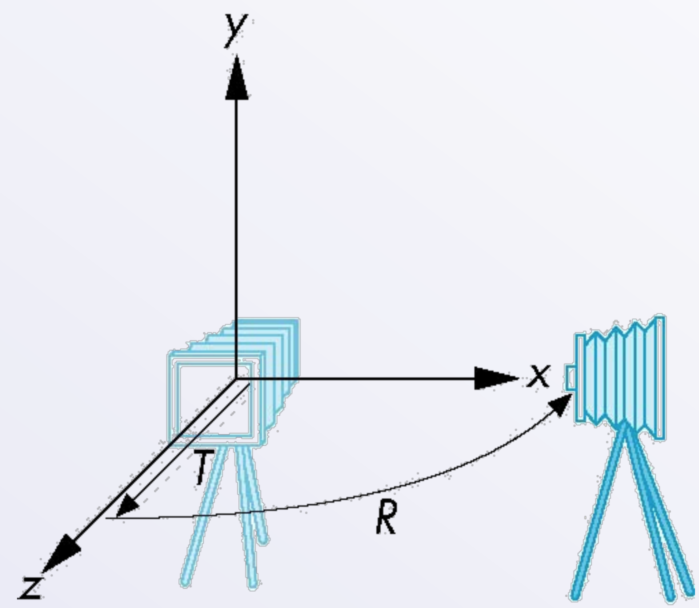
(a)



(b)

Movendo a câmera

- A posição da câmera pode ser movida através de rotações e translações
- Exemplo: visão lateral
  - Rotacione a câmera
  - Tire da origem
  - Matriz de modelo $C = T.R_y$



Em OpenGL

- Lembre-se que a última transformação especificada é a *primeira* a ser aplicada

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
glTranslatef(0.0, 0.0, -d);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

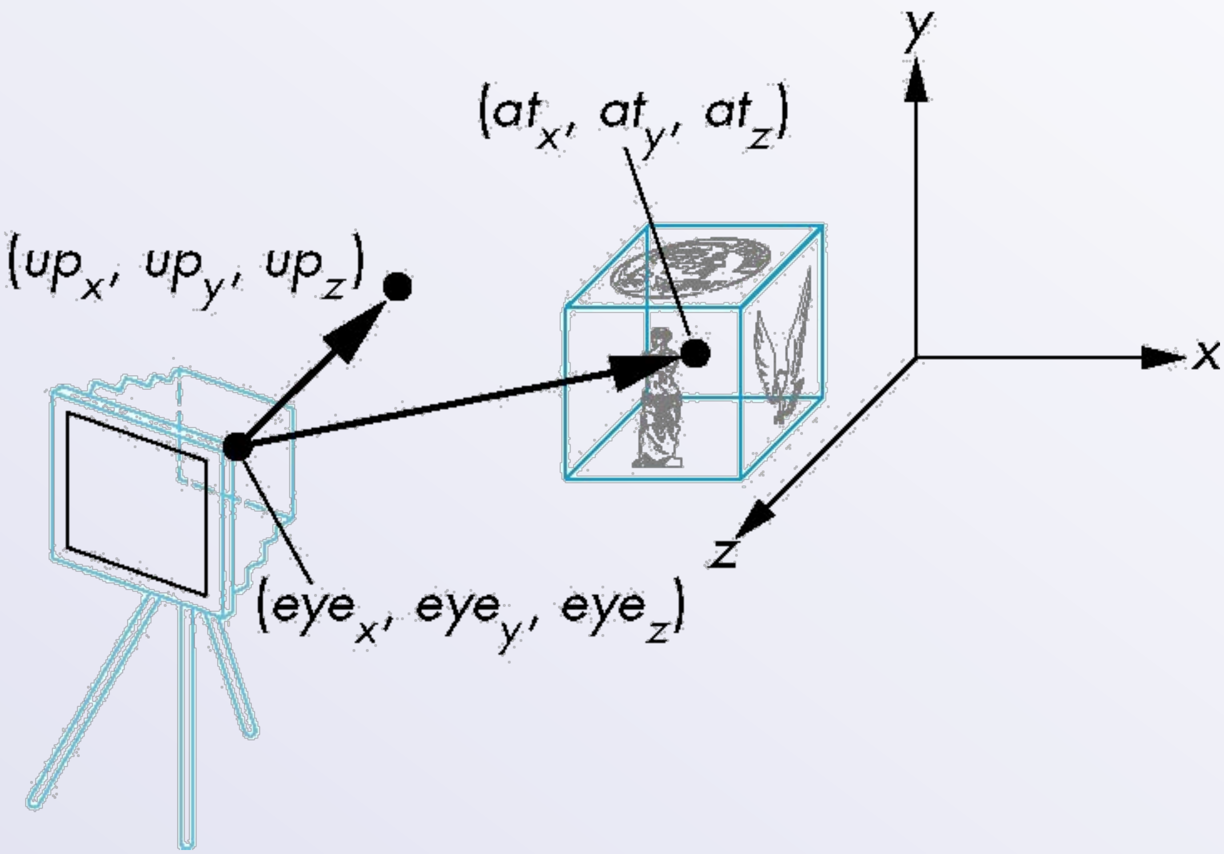
A função gluLookAt

- A biblioteca GLU oferece a função gluLookAt para formar a matriz de modelo através de parâmetros simples.
- Mas precisamos definir o vetor "up" (cima).
- Podemos concatenar com transformações de modelo.
- Exemplo:

```
glMatrixMode(GL_MODELVIEW):
glLoadIdentity();
gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0,
1.0. 0.0);
```

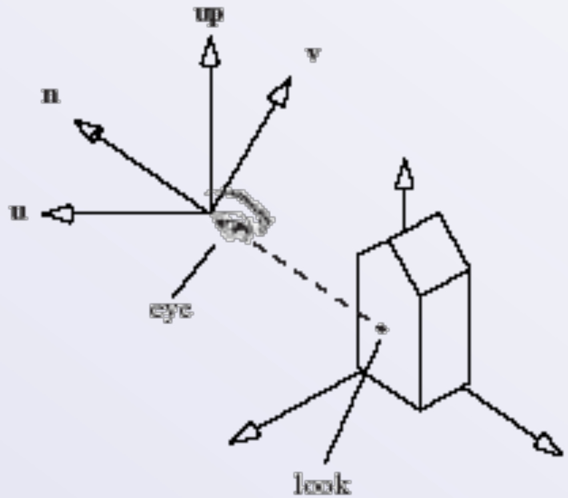
gluLookAt

```
gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)
```



gluLookAt

- `gluLookAt` cria um sistema de coordenadas da câmera (SRC) que consiste em de três vetores ortogonais: **u**, **v**, and **n.**
- **n** = eye – at; **u = up** x **n**; **v = n** x **u**
- Normaliza **n**, **u**, **v** (no sistema da câmera)



Matriz de visualização

- Então `gluLookAt()` monta a seguinte matriz de visualização:

$$M = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`gluLookAt()` é equivalente:

```
glMultMatrix(M);
glTranslated(-eye[0],-eye[1],-eye[2]);
```

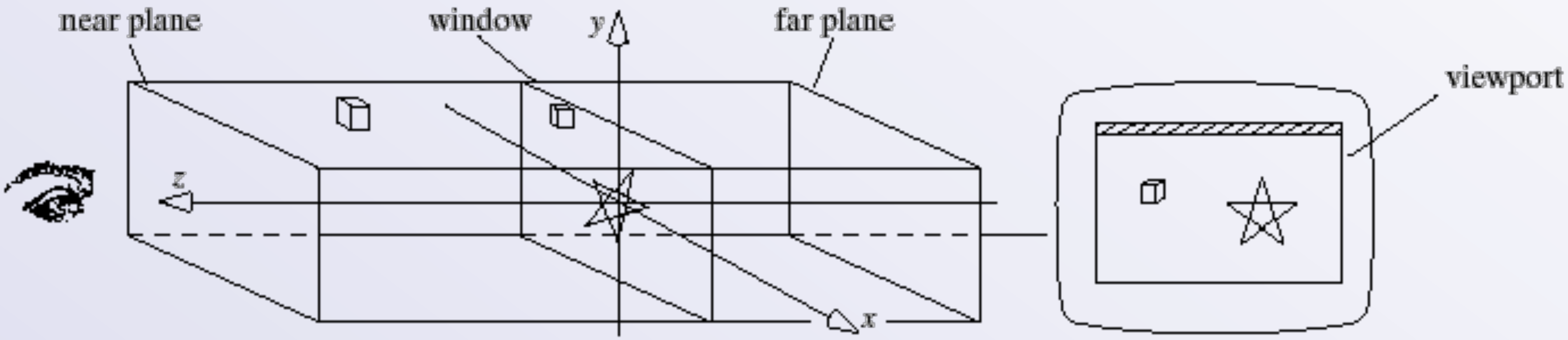
Exemplo

```
glMatrixMode(GL_PROJECTION);
   // seleciona volume de visualização (SRU)
glLoadIdentity();
glOrtho(-3.2, 3.2, -2.4, 2.4, 1, 50);
glMatrixMode(GL_MODELVIEW);
   // posiciona e direciona a câmera
glLoadIdentity ();
gluLookAt (4, 4, 4, 0, 1, 0, 0, 1, 0);
   // monta transformações do modelo
```

Projeções e normalização

- A projeção default da câmera é a ortogonal.
- Para pontos no volume de visualização default

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$



Matriz de projeção ortogonal

projeção ortogonal default

$x_p = x$
$y_p = y$
$z_p = 0$
$w_p = 1$

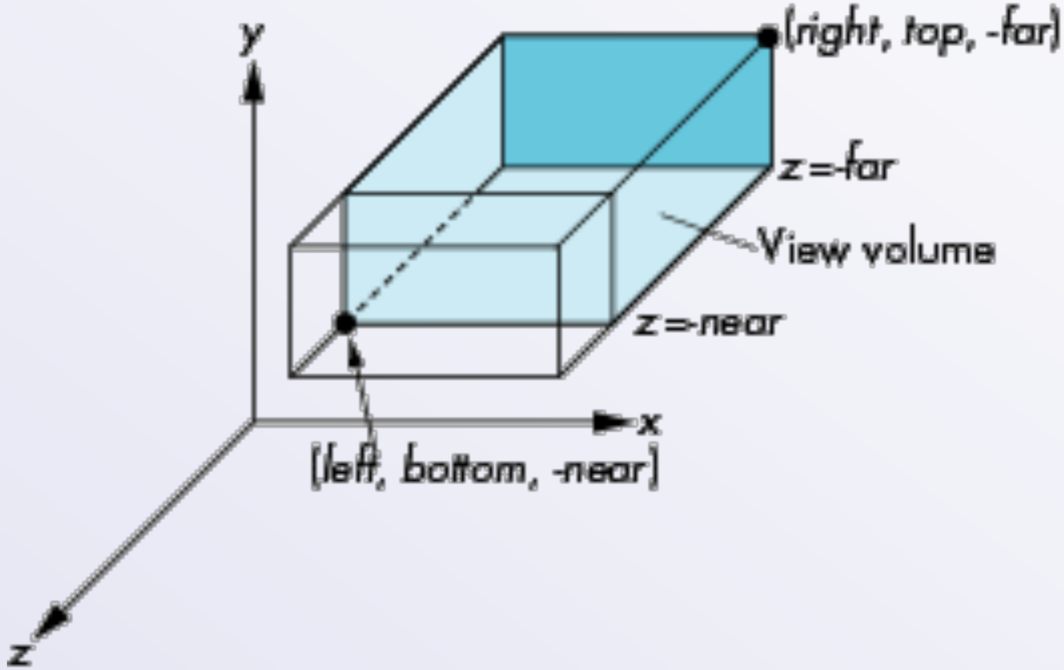
$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

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

Na prática, podemos fazer $\mathbf{M} = \mathbf{I}$ e zerar o termo z depois

Projeção ortogonal

`glOrtho(left,right,bottom,top,near,far)`



`near` e **`far`** medidos a partir da posição da câmera

Projeção ortogonal

```
glMatrixMode(GL_PROJECTION);
```

// faz com que a matriz corrente seja a matriz de projeção

```
glLoadIdentity();
```

// carrega a matriz identidade

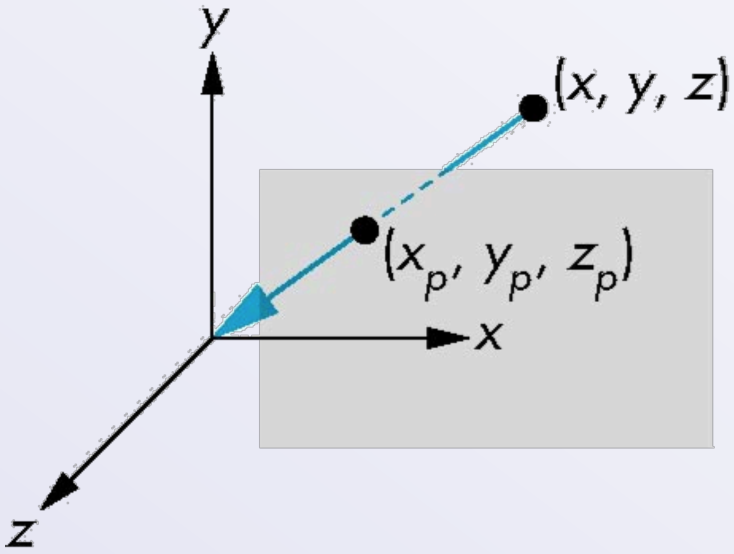
```
glOrtho(left, right, bottom, top, near, far);
```

// multiplica pela nova matriz

- Usando o valor 2 para *near* posiciona o plano de frente em *z* = -2, isto é, 2 unidades na frente do observador (eye).
- Usando 20 para *far* posiciona o plano de fundo em -20, 20 unidades na frente do observador (eye).

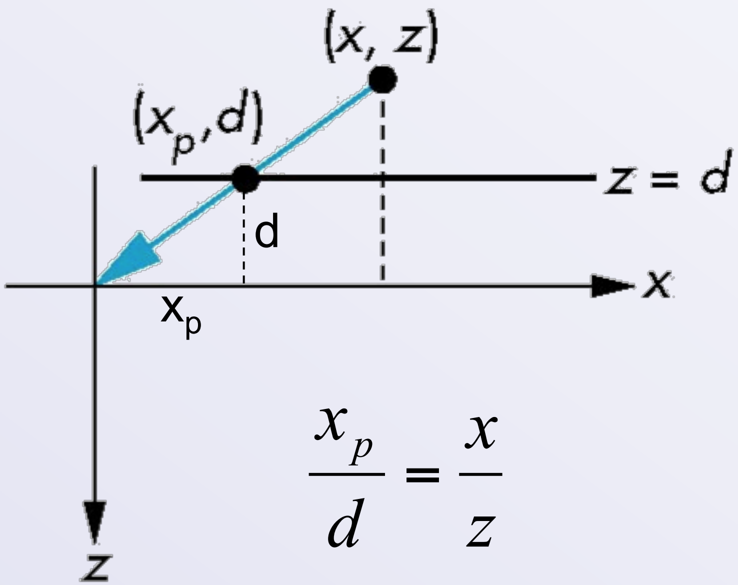
Projeção perspectiva

- Centro da projeção na origem
- Plano de projeção $z = d, d < 0$

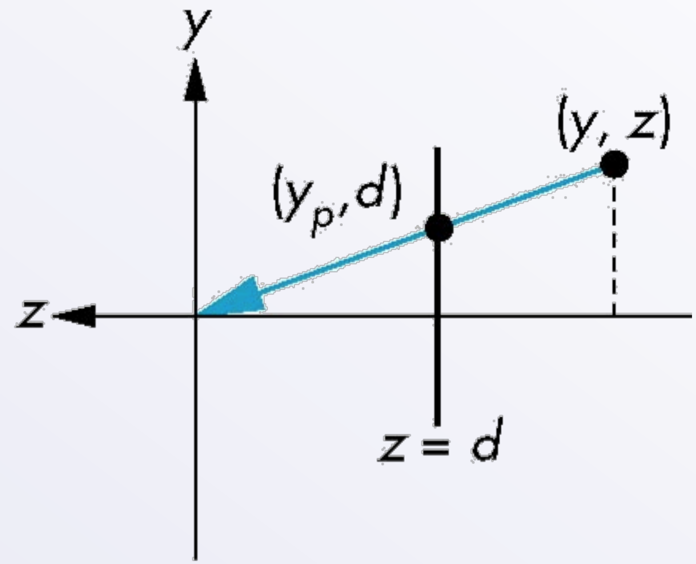


Equações de perspectiva

Visões superior e lateral



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



$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

Matriz de projeção perspectiva

considere $\mathbf{q} = \mathbf{M}\mathbf{p}$ onde

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

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Normalização da perspectiva

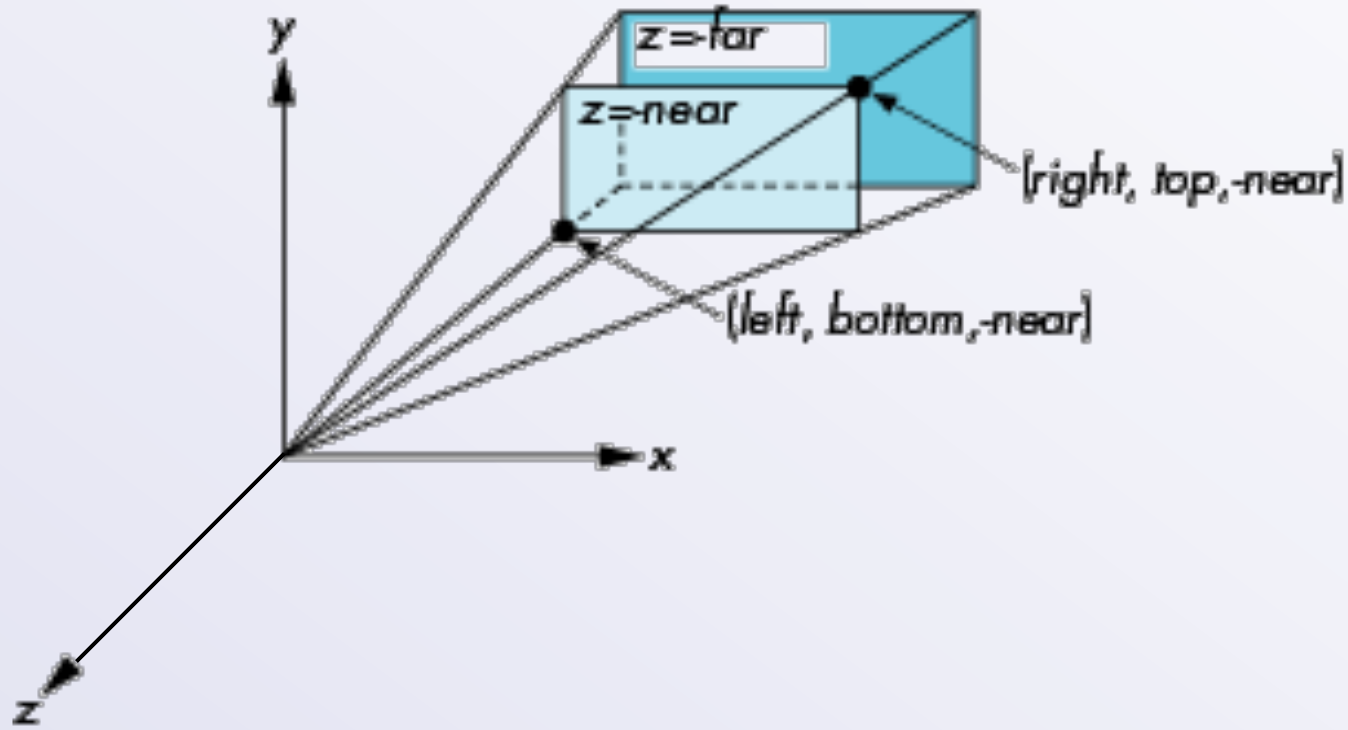
- Note que w = 1, então precisamos dividir por w para retornar às coordenadas homogêneas
- Essa divisão gera:

$$x_{\mathrm{p}} = \frac{x}{z/d} \qquad y_{\mathrm{p}} = \frac{y}{z/d} \qquad z_{\mathrm{p}} = d$$

que são as equações desejadas.

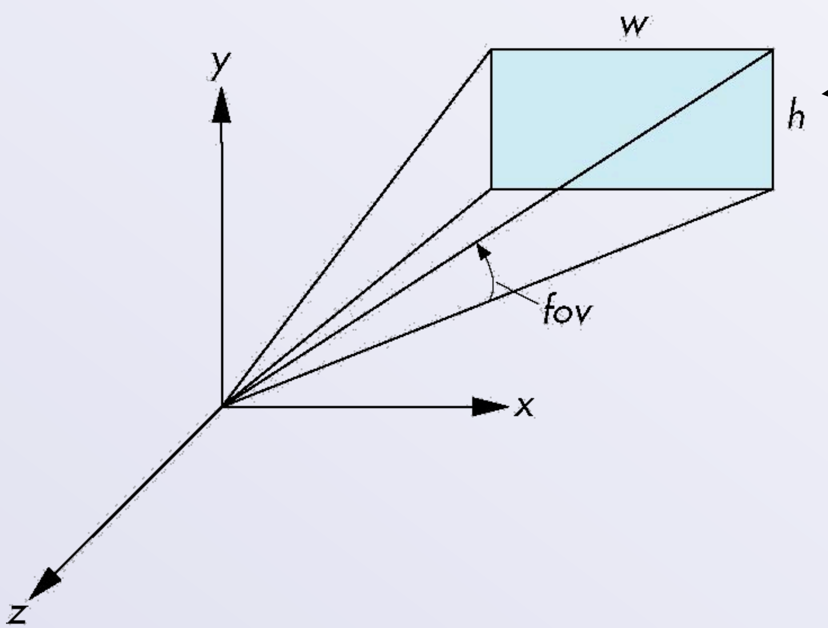
Projeção perspectiva

```
glFrustum(left,right,bottom,top,near,far)
```



Usando campo de visão (fov)

- Com **`glFrustum`** é um pouco complicado obter a visualização desejada
- **`gluPerpective(fovy, aspect, near, far)`** já oferece uma opção mais simples.



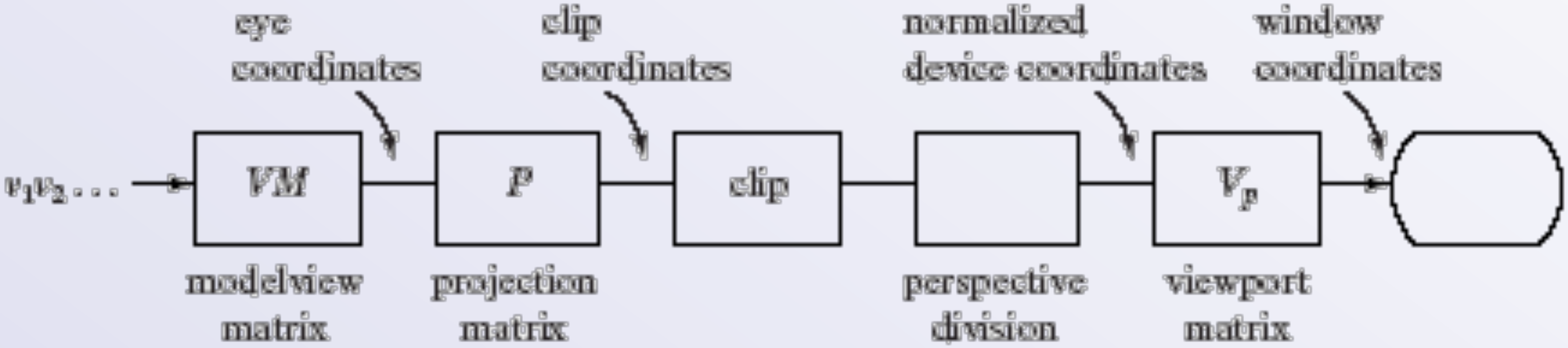
plano de projeção

`aspect = w/h`

Demo Nate Robins

Pipeline de visualização

- Vértices iniciam em coordenadas do universo;
- Após MV, em coordenadas do observador,
- Após P, em coordenadas de recorte;
- Após a normalização de perspectiva, em coordenadas normalizadas do dispositivo;
- E, finalmente após V_p, em coordenadas da tela.



$$W = V_p * D_{div} * C_{clip} * P * V * M$$

Construindo cenas em 3D

- Desejamos transformar objetos para orientá-los e posicioná-los em uma cena.
- A biblioteca OpenGL provém as funções necessárias para construir e aplicar as matrizes de transformações necessárias.
- As pilhas de matrizes mantidas pelo OpenGL tornam mais fácil a especificação de transformações para diferentes objetos:

```
glMatrixMode(GL_MODELVIEW)
glPushMatrix()
    Define transformação para objeto #1
    Desenha objeto #1
glPopMatrix()
glPushMatrix()
    Define transformação para objeto #2
    Desenha objeto #2
glPopMatrix()
```

Desenhando objetos com GLU

- A GLU disponibiliza uma série de objetos 3D: esfera, cone, toro, 5 sólidos platônicos, e o bule de chá (“teapot”).
- Cada um deles é disponível em modelo wireframe e modelo sólido.
- Todos eles são desenhados por default na origem.
- Para usar a versão sólida de cada, troque Wire por Solid nas funções.

Sólidos Platônicos

- Todo poliedro convexo onde:
  - Todas as suas faces são polígonos congruentes.
  - Em cada vértice encontram-se o mesmo número de faces.

Nome	Imagem	Faces	Arestas	Vértices	Vértices por face	Encontros de faces em cada vértice	Configuração vértices
tetraedro		4	6	4	3	3	3.3.3
cubo (hexaedro)		6	12	8	4	3	4.4.4
octaedro		8	12	6	3	4	3.3.3.3
dodecaedro		12	30	20	5	3	5.5.5
icosaedro		20	30	12	3	5	3.3.3.3.3

Desenho de objetos em 3D

- **cubo:** glutWireCube (GLdouble size);
  - Cada lado tem comprimento size.
- **esfera:** glutWireSphere (GLdouble radius, GLint nSlices, GLint nStacks);
  - nSlices é o número de cortes;
  - nStacks é o número de discos;
  - De outra forma, nSlices representam número de linhas longitudinais e nStacks representam o número de linhas latitudinais.

Desenho de objetos em 3D

- **toro:** glutWireTorus (GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks);
- **teapot:** glutWireTeapot (GLdouble size);

Desenho de objetos em 3D

- **tetraedro:** glutWireTetrahedron ();
- **octaedro:** glutWireOctahedron ();
- **dodecaedro:** glutWireDodecahedron ();
- **icosaedron:** glutWireIcosahedron ();
- **cone:** glutWireCone (GLdouble baseRad, GLdouble height, GLint nSlices, GLint nStacks);

Desenho de objetos em 3D

- **cilindro:** gluCylinder (GLUquadricObj *qobj, GLdouble baseRad, GLdouble topRad, GLdouble height, GLint nSlices, GLint nStacks);
- A função **gluCylinder** desenha uma *família* de objetos, dependendo do valor de topRad.
  - Quando topRad é 1, ela desenha um cilindro.
  - Quando topRad é 0, ela desenha um cone.

Desenho de objetos em 3D

```
// cria um objeto quádrico
GLUquadricObj *qobj =
  gluNewQuadric();
// muda estilo para wireframe
gluQuadricDrawStyle(qobj,GLU_LINE |
  GLU_FILL);
// desenha cilindro
gluCylinder(qobj, baseRad, topRad,
  nSlices, nStacks);
```

Demo teapot.c

Exercício

- Lei da reflexão: ângulo de incidência = ângulo de reflexão.
- Dados somente a normal à superfície e o vetor incidente, deduzir o vetor de reflexão, assumindo um espelho perfeito.
- Pode-se assumir que os vetores tem norma 1.