3
Transformations

Points and vectors

Consider a point P and a vector v:

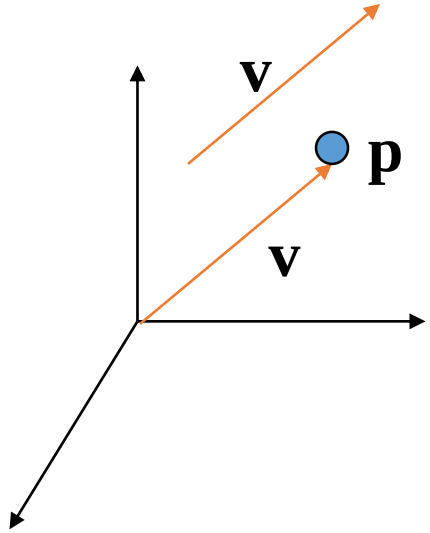
$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

They appear to have similar representations:

$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3]$ $\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3]$

But: contrary to a point, a vector has no position



Homogeneous coordinate representation

If we define $0.\mathrm{P} = \mathbf{0}$ and $1.\mathrm{P} = \mathrm{P}$ then:

$$\mathrm{P} = \mathrm{P}_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1\ \beta_2\ \beta_3\ 1\ ]\ [v_1\ v_2\ v_3\ \mathrm{P}_0]^{\mathrm{T}}$$

$$\mathrm{v} = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0\ ]\ [v_1\ v_2\ v_3\ \mathrm{P}_0]^{\mathrm{T}}$$

And we obtain the four-dimensional representation in *homogeneous coordinates*:

$$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3\ 1\ ]^{\mathrm{T}}$$

$$\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3\ 0\ ]^{\mathrm{T}}$$

Homogeneous coordinates

The homogeneous coordinates form for a three dimensional point [x y z] is given as

$\mathbf{p} = [x' \; y' \; z' \; w]^T = [wx \; wy \; wz \; w]^T$

We return to a three dimensional point (for $w \neq 0$) by

$x \leftarrow x'/w$

$y \leftarrow y'/w$

$z \leftarrow z'/w$

If $w=0$, the representation is that of a vector

Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

For w=1, the representation of a point is [x y z 1]

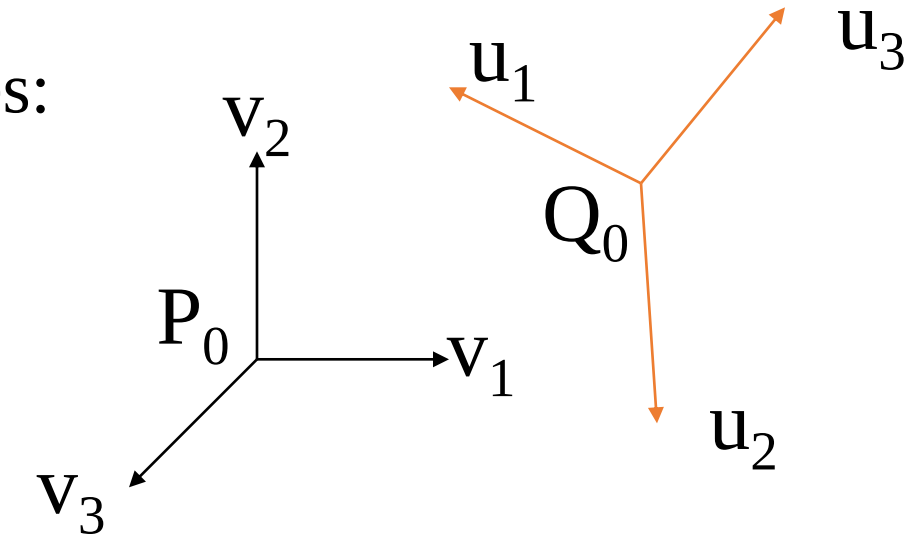
Homogeneous coordinates in graphics system

- Homogeneous coordinates are widely used in the implementation of graphics systems
- All standard, i.e. affine, transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
- Modern hardware implements homogeneous coordinate operations directly using parallelism
- For perspective projection we need a *perspective division*

Change of frame

- Any point or vector can be represented in either frame
- We can represent Q_0, u_1, u_2, u_3 in terms of P_0, v_1, v_2, v_3

Consider two frames:
(P_0, v_1, v_2, v_3)
(Q_0, u_1, u_2, u_3)



Change of frame

- Expressing the second frame (vectors and origin) in the first frame:

$$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$$
$$Q_0 = \gamma_{41}v_1+\gamma_{42}v_2+\gamma_{43}v_3+P_0$$

- Giving the following transformation matrix:

$$M = [u_1 \quad u_2 \quad u_3 \quad Q_0]$$

Change of frame

- Given a point (or vector) $\mathbf{b}$ in the second frame, M b gives the point in the first frame
- To express a point (or vector) in the second frame, given its coordinates in the first frame, use the inverse of M
- Changing frames correspond to what we do when going from object space coordinates to world space coordinates to camera space coordinates for example
- Matrix $\mathbf{M}$ is 4 x 4 and specifies an affine transformation in homogeneous coordinates

Affine transformations

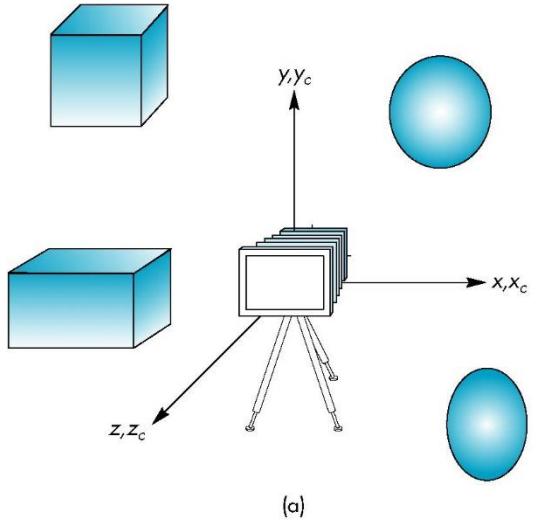
- Every linear transformation is equivalent to a change in frames
- An affine transformation is a transformation that preserves: collinearity and ratios of distances between points lying on a straight line
- Examples of affine transformation: contraction, expansion, dilation, reflection, rotation, translation, … and their combinations
- An affine transformation is expressed as the composition of a linear transformation and a translation
- So it has only 12 *degrees of freedom*

World and Camera frames

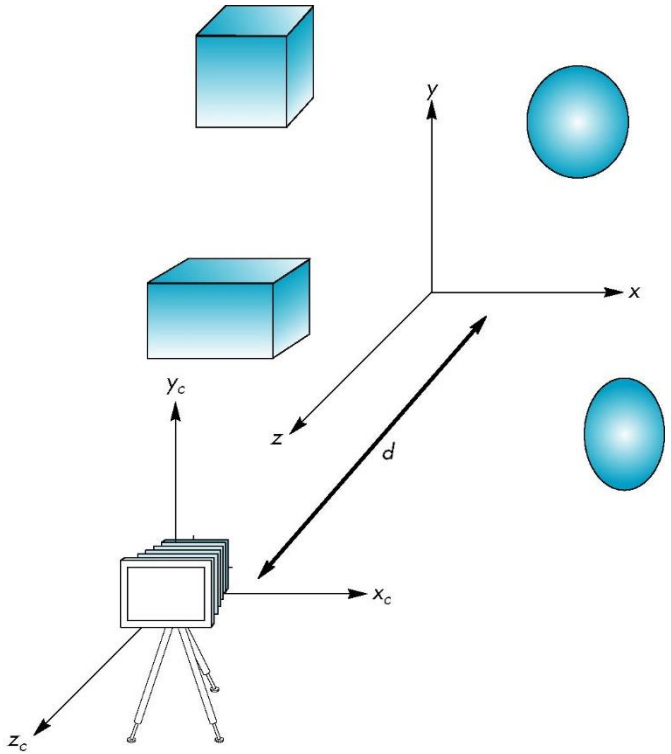
- Changes in frame are defined by 4 x 4 matrices
- In OpenGL, the base frame that we start with is the world frame
- Objects need to be represented in the camera frame. Changing from the world frame to the camera frame is done with the model-view matrix
- Initially these frames are the same ($\mathbf{M}=\mathbf{I}$)

Example: moving the camera

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



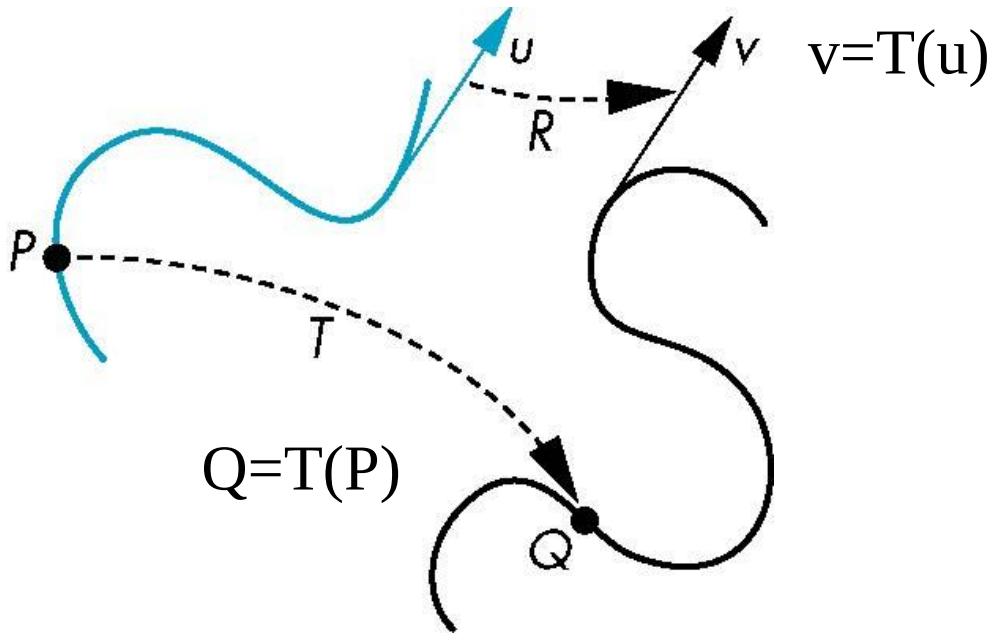
(a)



(b)

General transformations

- Maps points to points (or vectors to vectors)



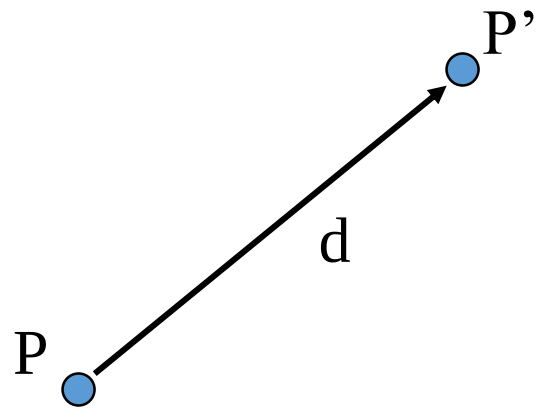
v=T(u)

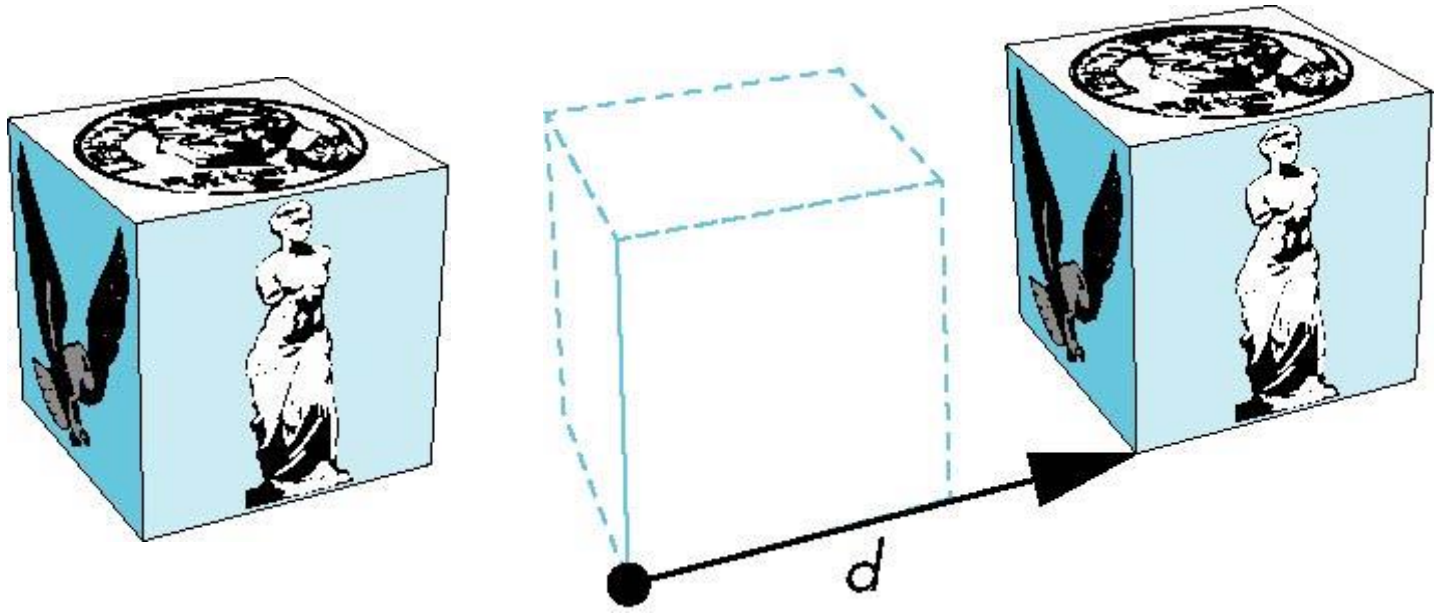
Affine transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints

Translation

- Translate a point to a new point
- Three degree of freedoms





Translation

Translation is expressed as:

$\mathbf{p}' = \mathbf{p} + \mathbf{d}$

Where:

$\mathbf{p} = [\, x \; y \; z \; 1]^T$

$\mathbf{p}' = [x' \; y' \; z' \; 1]^T$

$\mathbf{d} = [dx \; dy \; dz \; 0]^T$

Translation in matrix form

In matrix form, we can express the translation with the 4 x 4 matrix $\mathbf{T}$ in homogeneous coordinates, such that: $\mathbf{p'=Tp}$ where

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

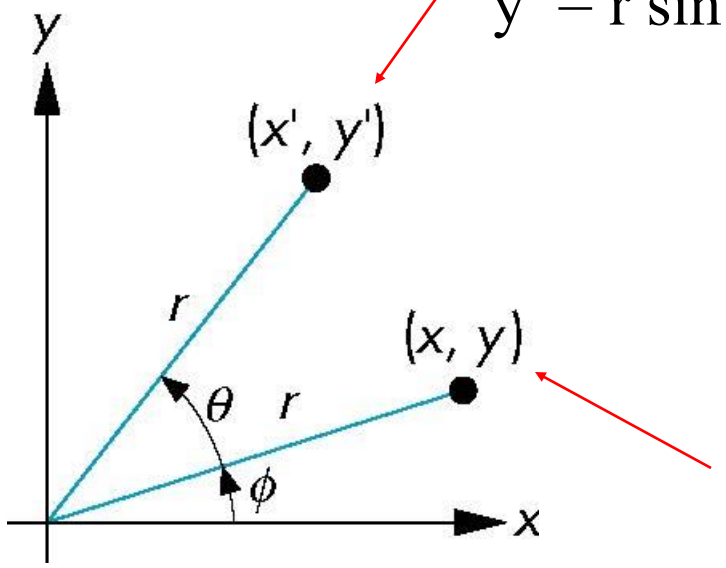
This form is better for implementation because all affine transformations can be expressed by a matrix. Composition of transformations is therefore equivalent to a product of matrices.

Rotation in 2D

Rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

$$x' = r \cos(\phi + \theta)$$
$$y' = r \sin(\phi + \theta)$$



$$x' = x \cos\theta - y \sin\theta$$
$$y' = x \sin\theta + y \cos\theta$$

$$x = r \cos\phi$$
$$y = r \sin\phi$$

Rotation in 3D about the z-axis

- Rotation about z-axis in three-dimension leaves all points along the z-axis unchanged (invariant)
  - Equivalent to a rotation in two-dimension for constant z

    $x' = x \cos\theta - y \sin\theta$

    $y' = x \sin\theta + y \cos\theta$

    $z' = z$
  - or in homogeneous coordinates

    $$\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$$

Rotation in 3D about the z-axis

- In matrix form:

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

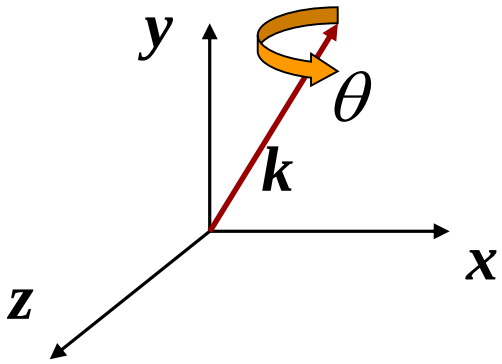
Rotation about the x and y-axis

- Similarly we get the matrices for the rotations about the x and y-axis:

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

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

Rotation about a general vector



- Assuming k is a general, unit, vector

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} k_x k_x(1-c)+c & k_x k_y(1-c)-k_z s & k_x k_z(1-c)+k_y s & 0 \\ k_y k_x(1-c)+k_z s & k_y k_y(1-c)+c & k_y k_z(1-c)-k_x s & 0 \\ k_z k_x(1-c)-k_y s & k_z k_y(1-c)+k_x s & k_z k_z(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$c = \cos\theta \qquad s = \sin\theta$$

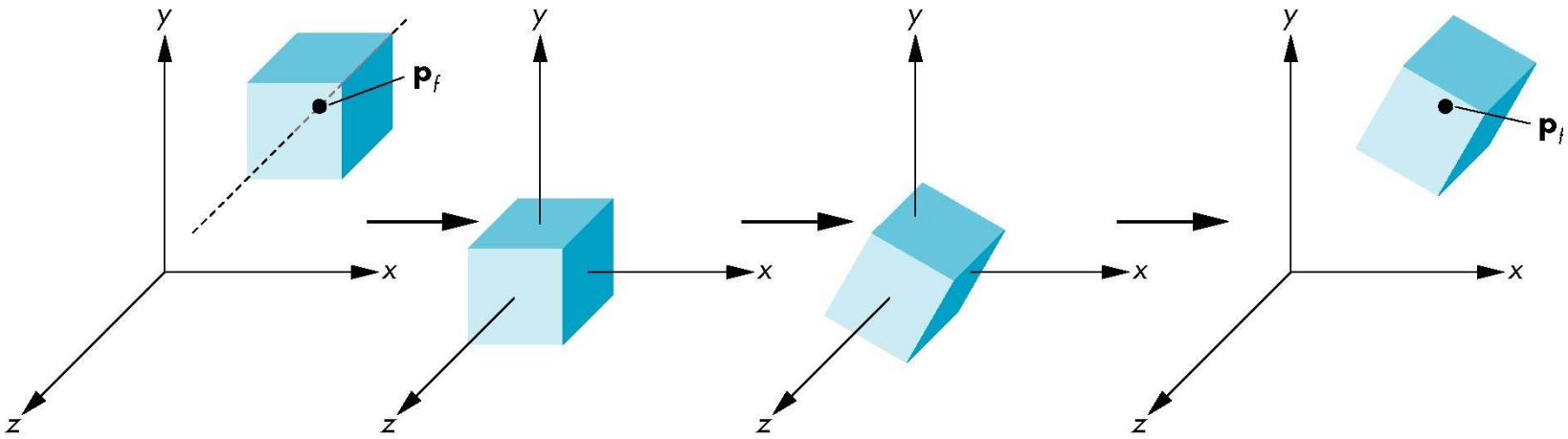
Rotation about a point other than origin

Move fixed point to origin

Rotate

Move fixed point back

$$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$$



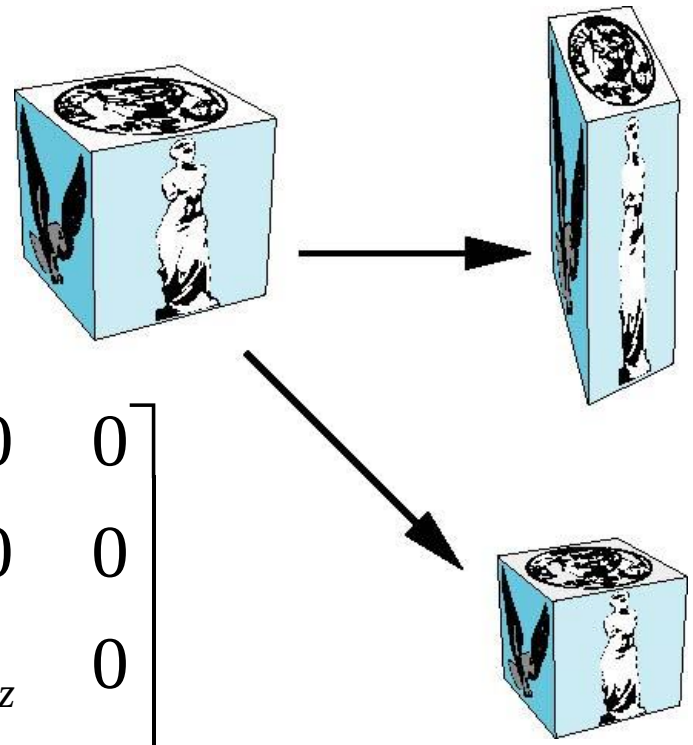
Scaling

- Expand or contract along each axis (fixed point of origin)

$$x' = s_x x$$
$$y' = s_y y$$
$$z' = s_z z$$

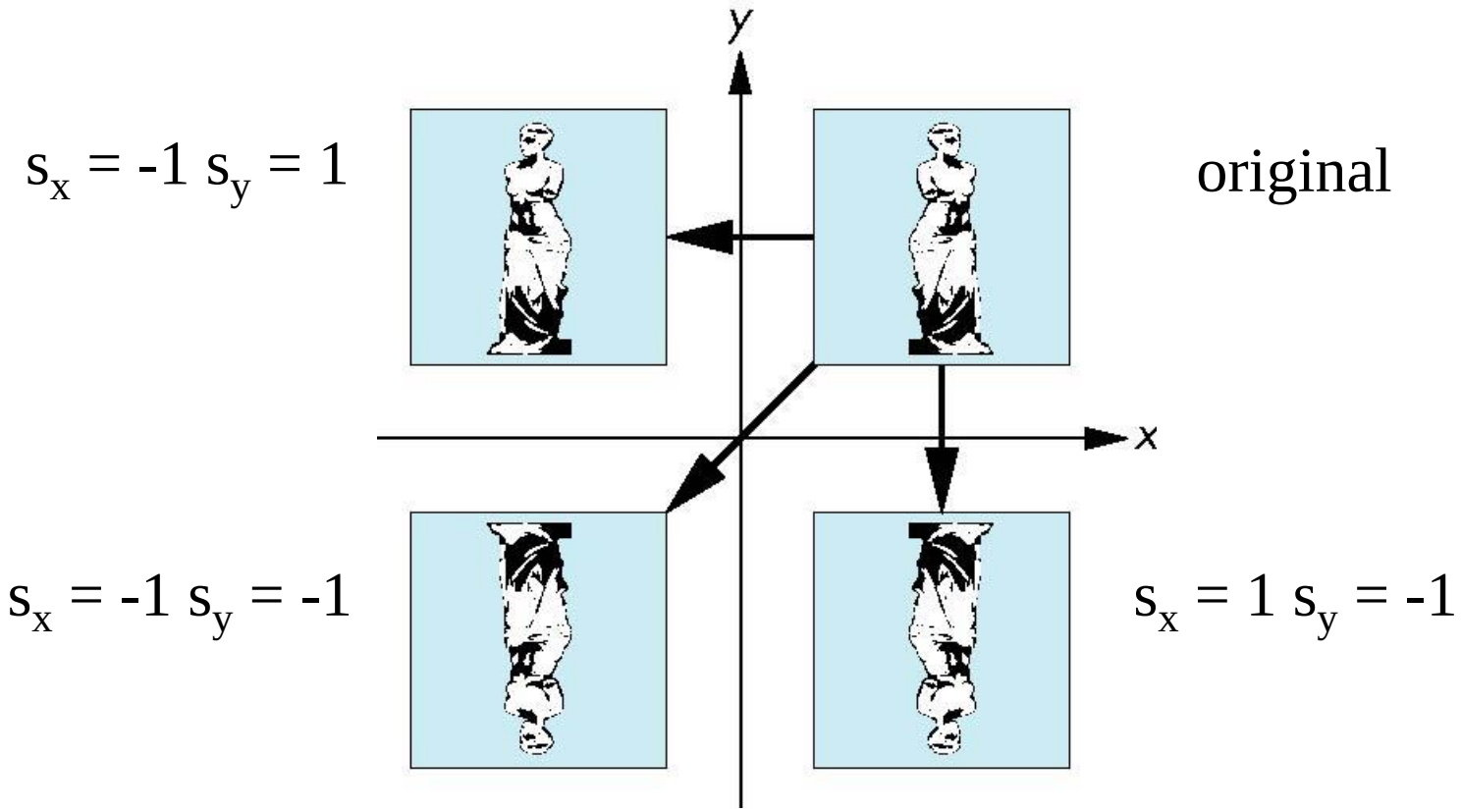
$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



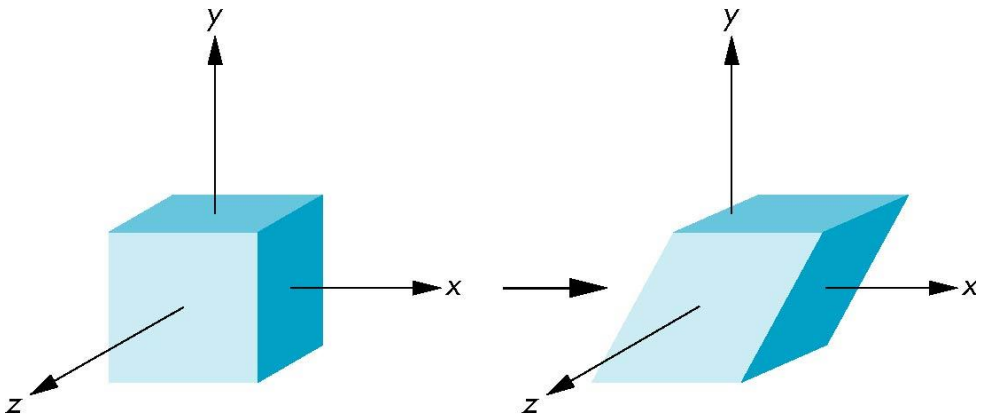
Reflection

- corresponds to negative scale factors



Shear

- Equivalent to pulling faces in opposite directions

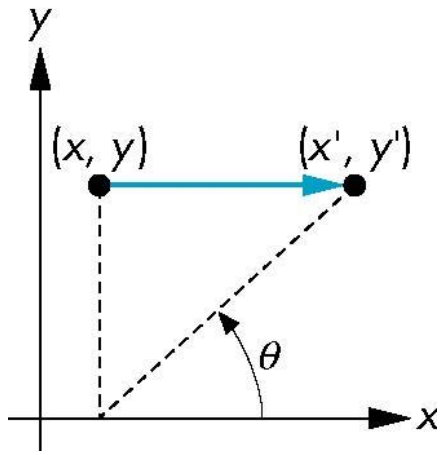


Shear

- Consider simple shear along x axis

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$

$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Inverses

- In the general case, one can use a formula to invert a 4x4 matrix
- However, for particular cases, one can use simple geometric observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$

    $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

- Composition of transformations is equivalent to matrix products
- Transformation matrix can be obtained by multiplying rotation, translation and scaling matrices
- Because the same transformation M is applied to many vertices, the cost of forming M by matrix multiplication of elementary transformations is negligible compared to the cost of computing M.p for multiple p

Order of transformation

- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

$$\mathbf{p'} = \mathbf{ABCp} = \mathbf{A(B(Cp))}$$

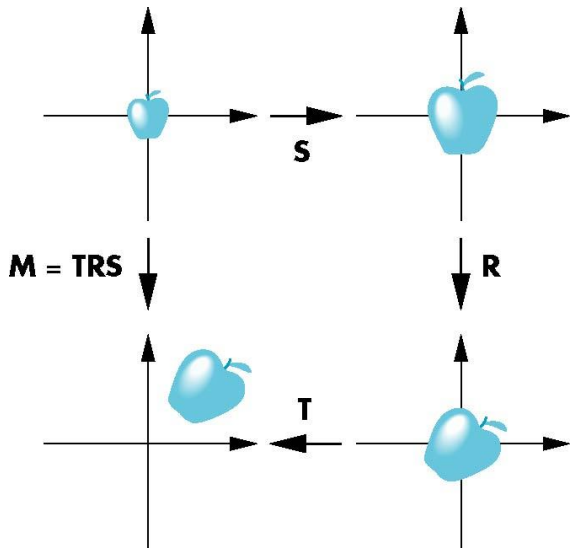
Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to

  Scale

  Orient

  Locate



OpenGL transformations

- In OpenGL matrices are part of the state
- Multiple types
  - Model-View (**`GL_MODELVIEW`**)
  - Projection (**`GL_PROJECTION`**)
  - Texture (**`GL_TEXTURE`**)
  - Color(**`GL_COLOR`**)
- Single set of functions for manipulation
- Select which to manipulate by
  - **`glMatrixMode(GL_MODELVIEW);`**
  - **`glMatrixMode(GL_PROJECTION);`**

OpenGL transformations

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

Right multiply the current matrix by the translation matrix constructed from (x, y, z).

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

Right multiply the current matrix by the rotation matrix constructed from (angle, x, y, z), corresponding to the counterclockwise rotation about the vector from the origin through the point (*x*, *y, z*). The *angle* is in degrees.

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

Right multiply the current matrix by the scaling matrix constructed from (*x*, *y, z).*

OpenGL transformations

glLoadMatrixf(m): Replace the current matrix with m

```
float m[]={m0,m1,m2,m3,
    m4,m5,m6,m7,
    m8,m9,m10,m11,
    m12,m13,m14,m15};
```

$$m = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

OpenGL transformations

glMultMatrixf(m): right multiply the current matrix by the matrix m and update the current matrix with the result

Example:

If the current (OpenGL) matrix is A, then after a call to glMultMatrixf(B), the current matrix becomes A B

Projection and normalization

- The default projection in the eye (camera) frame is orthogonal
- For points within the default view volume

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

- Most graphics systems use *view normalization*
  - All other views are converted to the default view by transformations that determine the projection matrix
  - Allows use of the same pipeline for all views

Default orthographic projection

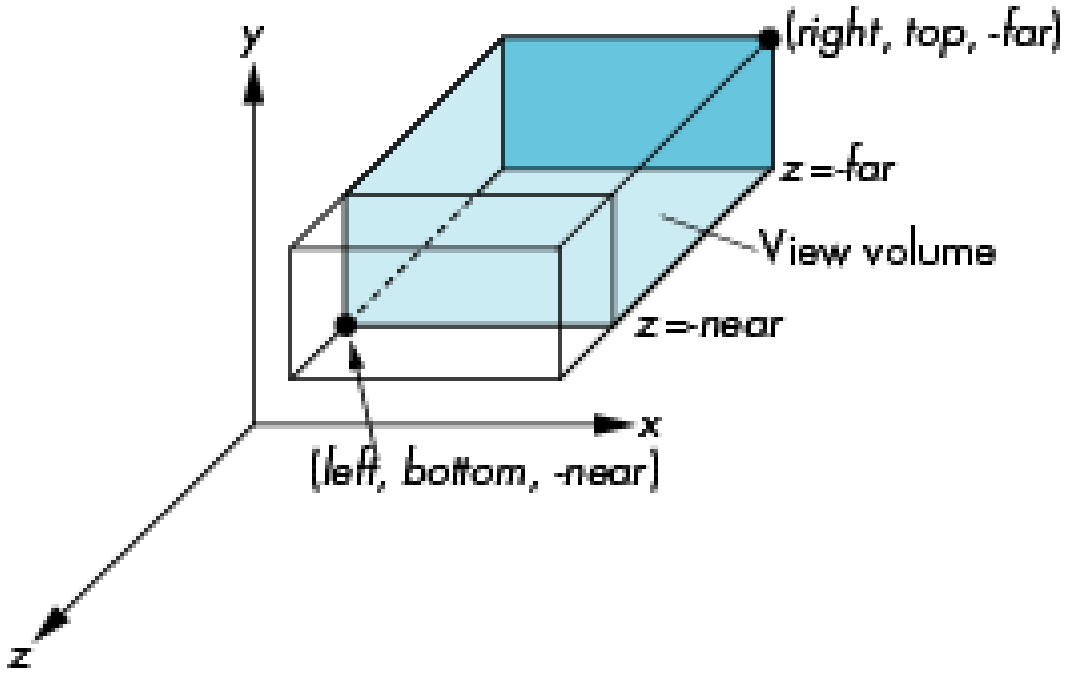
$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$
$$w_p = 1$$

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

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

OpenGL orthogonal projection

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

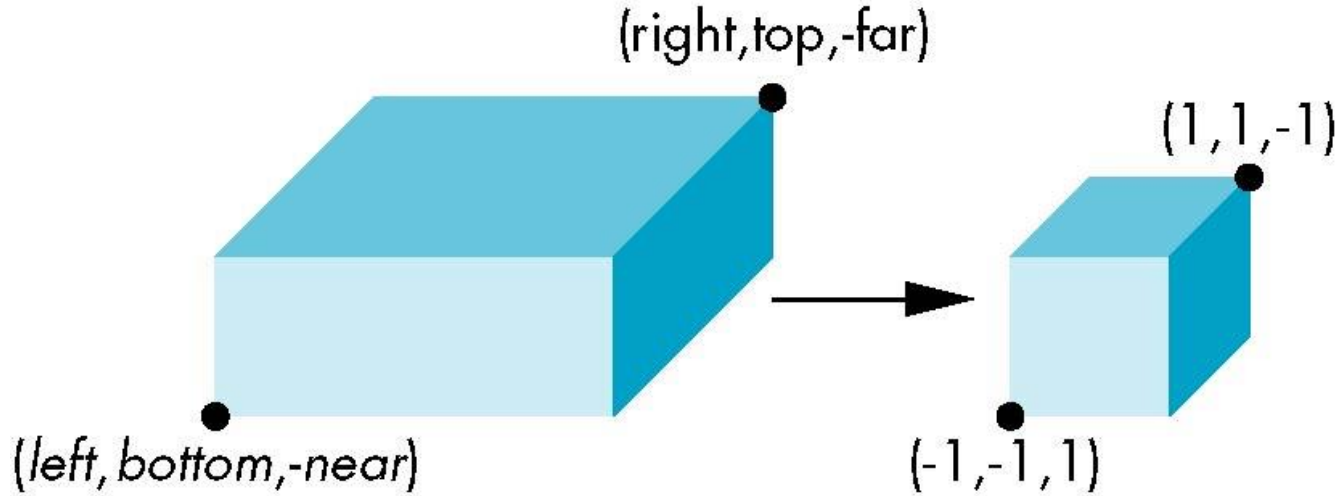


`near` and **`far`** measured from camera

Orthogonal projection and normalization

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

normalization ⇒ find transformation to convert specified clipping volume to default



OpenGL orthogonal projection and normalization matrix

- Two steps
  - Move center to origin

    T(-(left+right)/2, -(bottom+top)/2,(near+far)/2))
  - Scale to have sides of length 2

    S(2/(left-right),2/(top-bottom),2/(near-far))

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{near-far} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OpenGL orthogonal projection and normalization matrix

- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

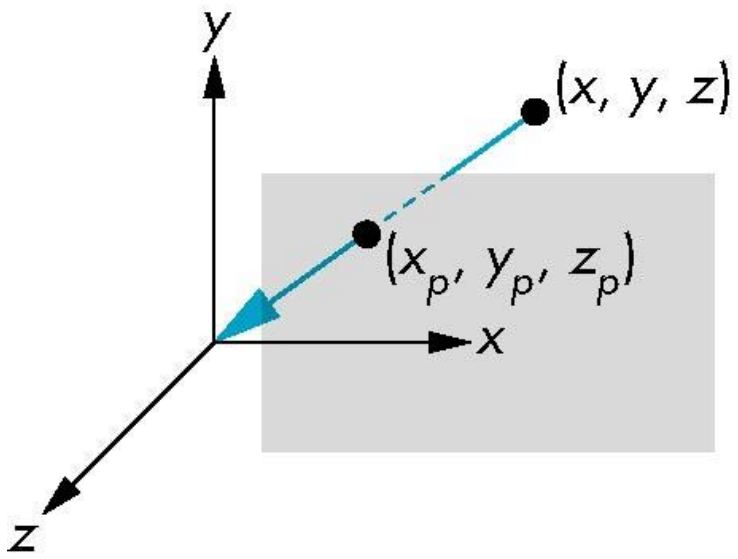
$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Hence, general orthogonal projection in 4D is

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$$

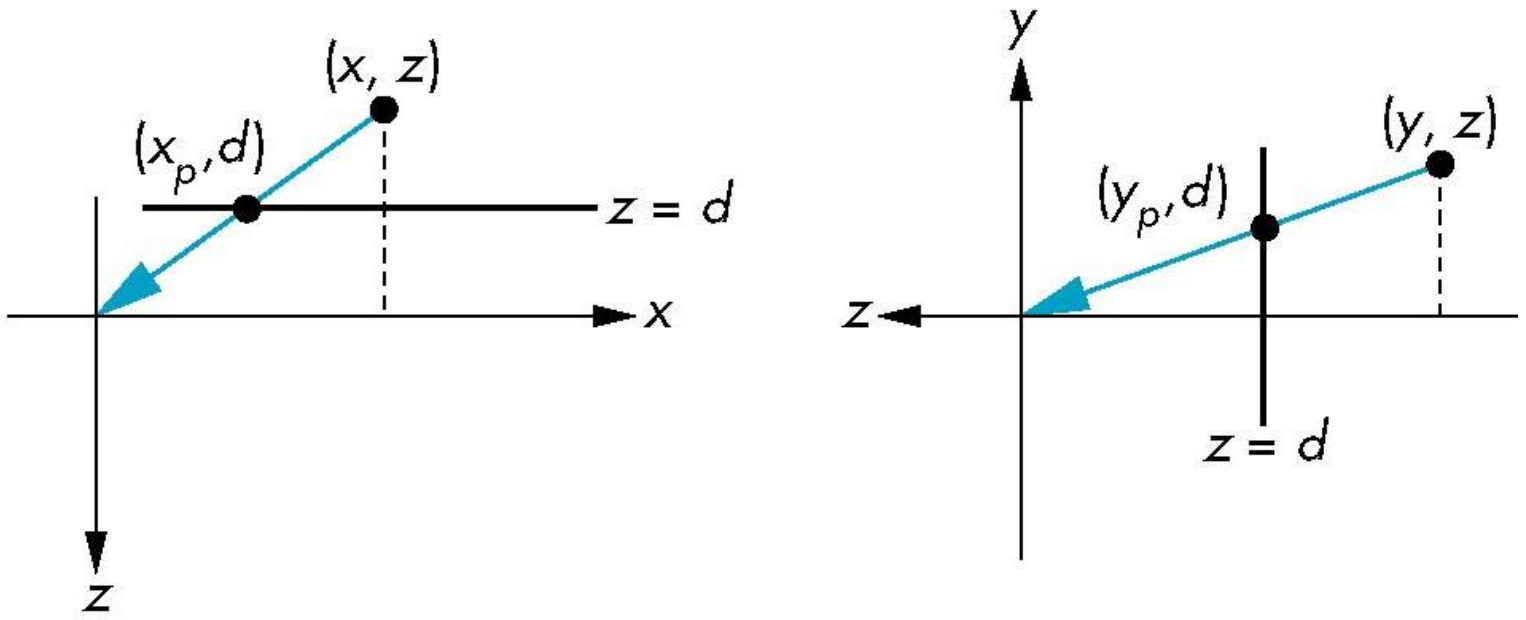
Perspective projection

- Center of projection at the origin
- Projection plane $z = d$, $d < 0$



Perspective projection

- Consider top and side views



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

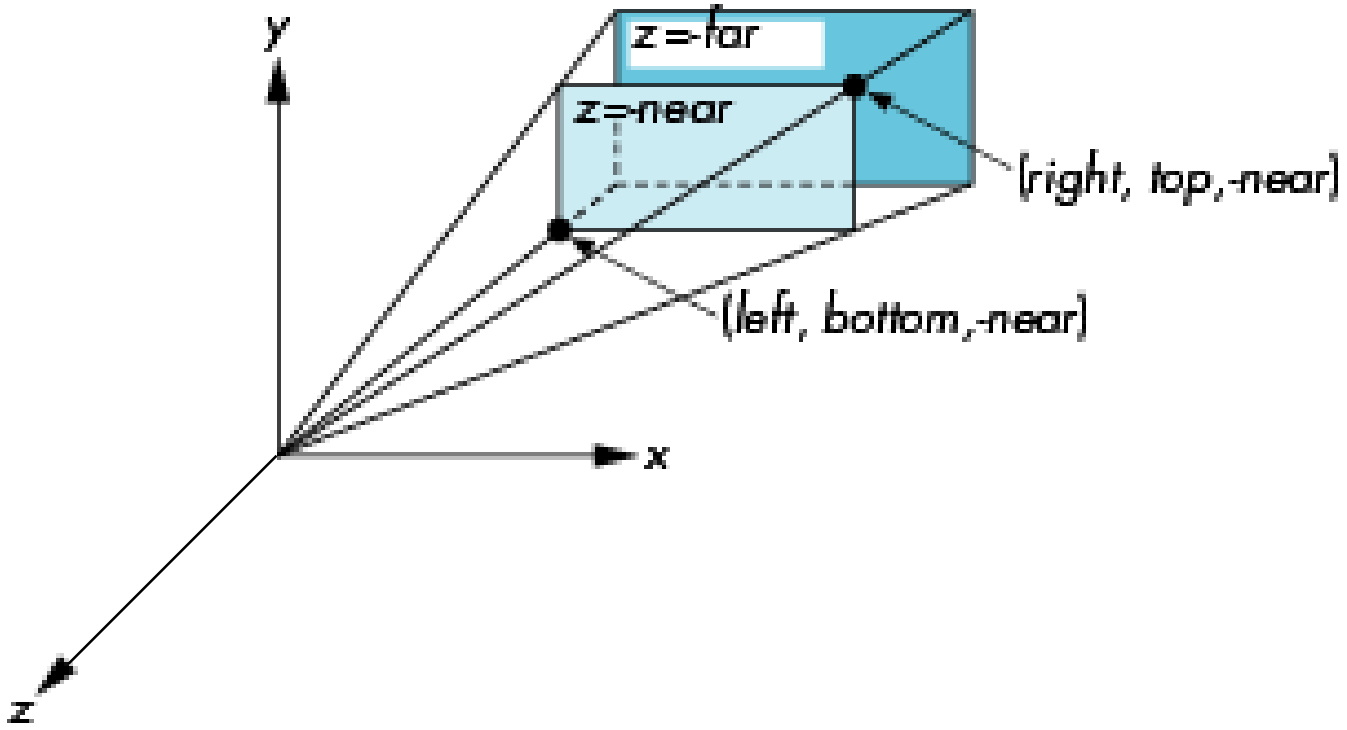
Perspective projection matrix

- consider $\mathbf{p} = \mathbf{M}\mathbf{q}$ where $\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$

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

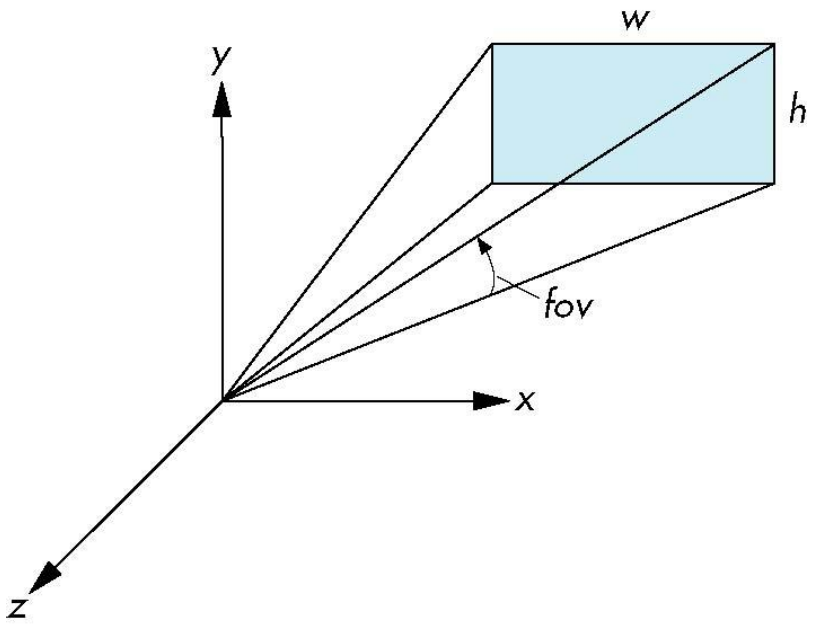
OpenGL perspective 1

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



OpenGL perspective 2

- **`gluPerpective(fovy, aspect, near, far)`** often provides a better interface



front plane

`aspect = w/h`

OpenGL perspective projection matrix

$$\begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} & \frac{-2\text{far}*\text{near}}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Where: left, right, bottom, top, near and far are the parameters of glFrustum

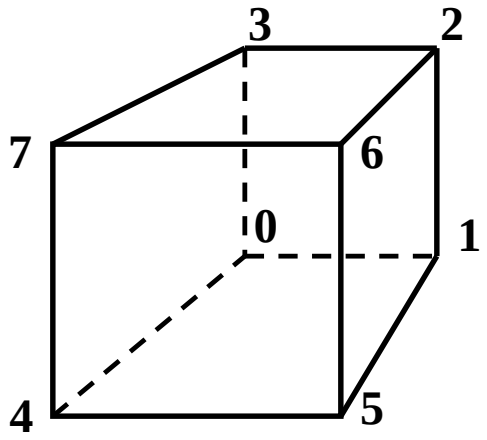
Code: rotating cube

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

GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[][4] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};

static GLfloat theta[] = {0.0,0.0,0.0};
```



Geometry

```
void polygon(int a, int b,
             int c, int d)
{
  glBegin(GL_POLYGON);
    glColor3fv(colors[a]);
    glVertex3fv(vertices[a]);
    glColor3fv(colors[b]);
    glVertex3fv(vertices[b]);
    glColor3fv(colors[c]);
    glVertex3fv(vertices[c]);
    glColor3fv(colors[d]);
    glVertex3fv(vertices[d]);
  glEnd();
}
```

```
void colorcube(void)
{
  polygon(0,3,2,1);
  polygon(2,3,7,6);
  polygon(0,4,7,3);
  polygon(1,2,6,5);
  polygon(4,5,6,7);
  polygon(0,1,5,4);
}
```

Rotation by pressing keys

```
void key(unsigned char k, int x, int y)
{
  if      (k == '1') rotateCube(0);
  else if(k == '2') rotateCube(1);
  else if(k == '3') rotateCube(2);
  else if(k == 'q') exit(0);
}

void rotateCube(int axis)
{
  theta[axis] += 5.0;
  if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
  glutPostRedisplay();
}
```

Reshape and display

```
void reshape(int w, int h)
{
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  glTranslatef(0.0,0.0,-5.0);
  glRotatef(theta[0], 1.0, 0.0, 0.0);
  glRotatef(theta[1], 0.0, 1.0, 0.0);
  glRotatef(theta[2], 0.0, 0.0, 1.0);
  colorcube();
  glutSwapBuffers();
}
```

Main

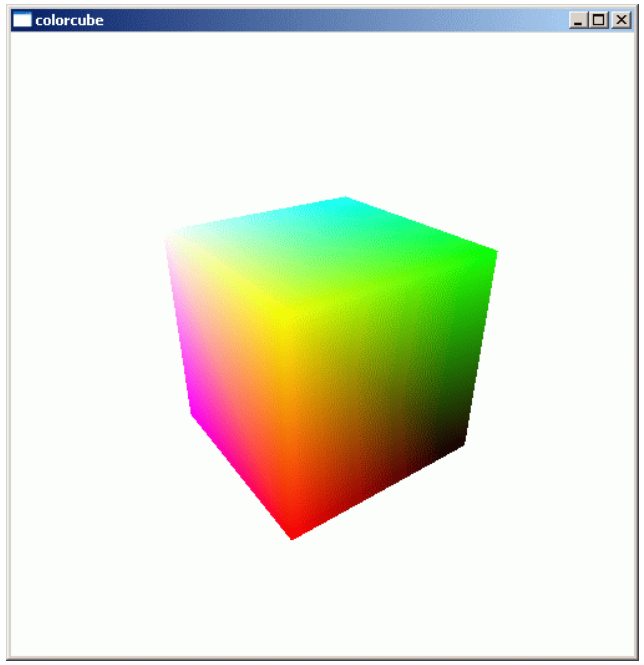
Use double buffering

Use depth-buffer

```
void main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutCreateWindow("colorcube");

  glEnable(GL_DEPTH_TEST);
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutKeyboardFunc(key);
  glClearColor(1.0,1.0,1.0,1.0);
  glutMainLoop();
}
```

Use depth-buffer (z-buffer)



Double buffering

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

Requests double buffering and RGB colors

```
glutSwapBuffers();
```

Swaps the front buffer (displayed on the screen) and the back buffer (used for drawing). Useful in codes with moving objects.

```
glutPostRedisplay();
```

Requests to execute the display callback