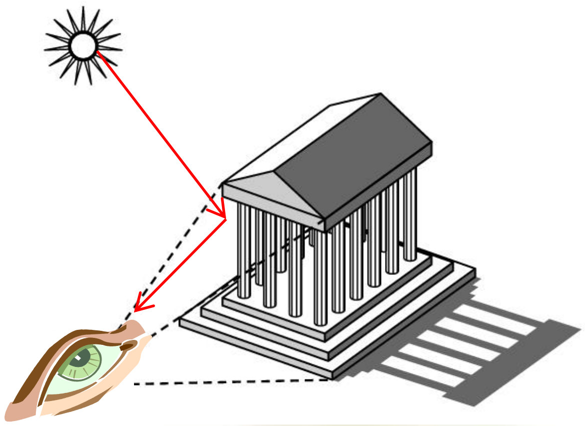
第3课
计算机图形变换

Key Contents

1. Viewing
2. Computer Viewing Pipeline
3. Modeling and Viewing Transformations
4. Viewing Matrix by LookAt
5. Projection: Orthogonal and Perspective Matrices
6. Normalization of View Volume
7. Viewport Transformation
8. Applications of Transformation: Moving Light Sources

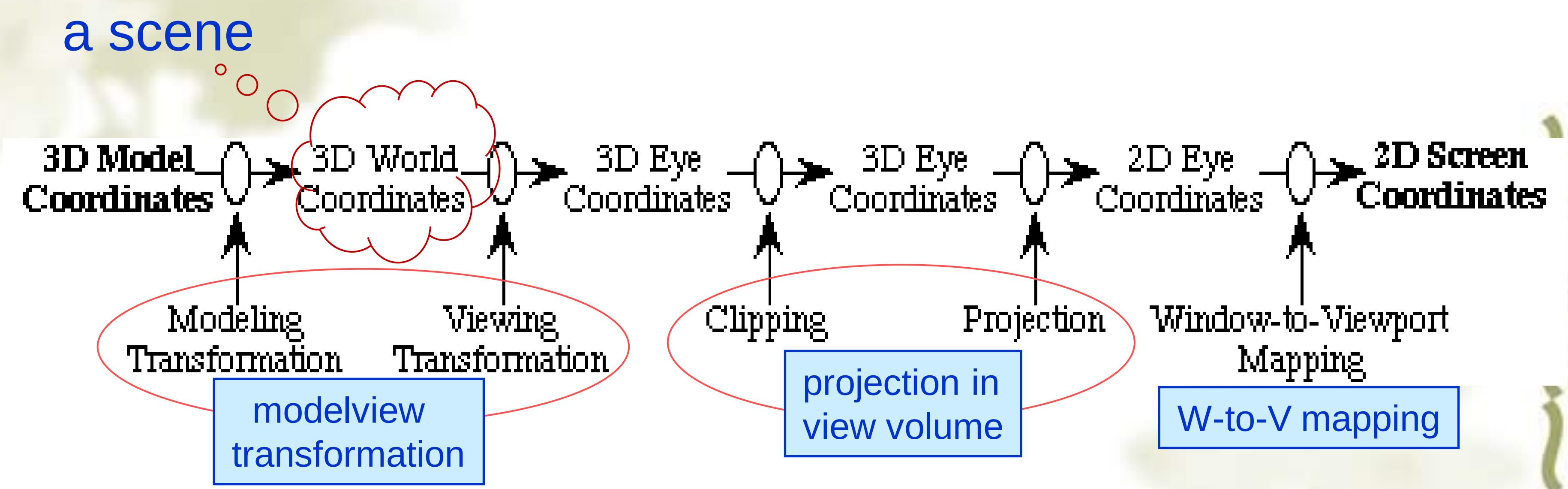
Viewing

- physical imaging systems
  - Eye(viewer) viewing
  - Camera(video) viewing



- Three basic elements:
  - Objects
  - Viewer with a projection surface
  - Light
- Note independence of objects, viewer, and light
- The light **reflects off** the objects（materials）to the viewer
- CG imaging system: Screen is an emission display, not reflection
  - Objects
  - Viewer with a projection surface
  - Viewer direction

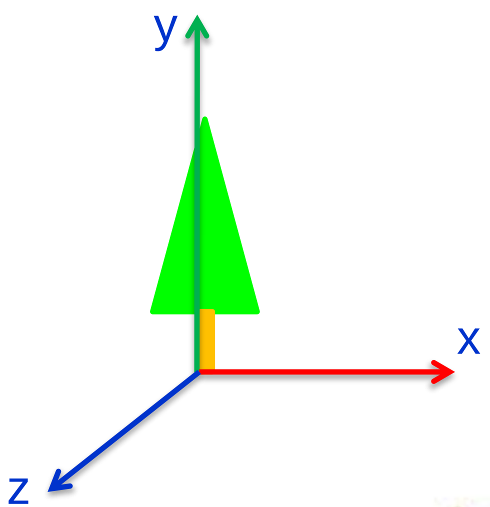
Computer Viewing Pipeline

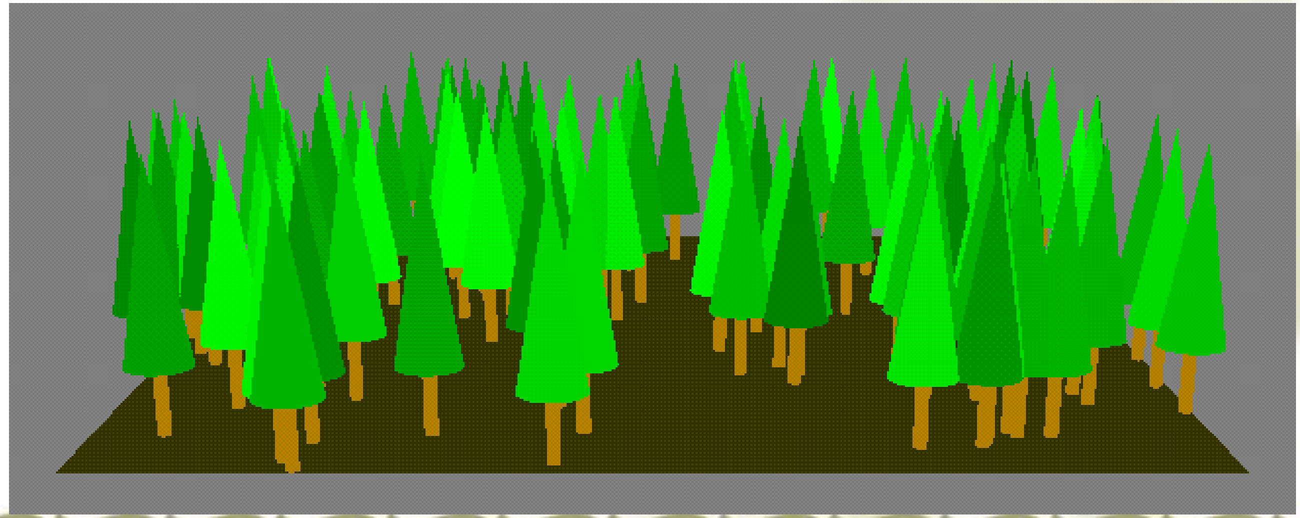


- The pipeline involves several **spaces** and **transformations** between them
- A graphics scene starts with geometry
- The scene is independent of the viewer

Modeling Transformations

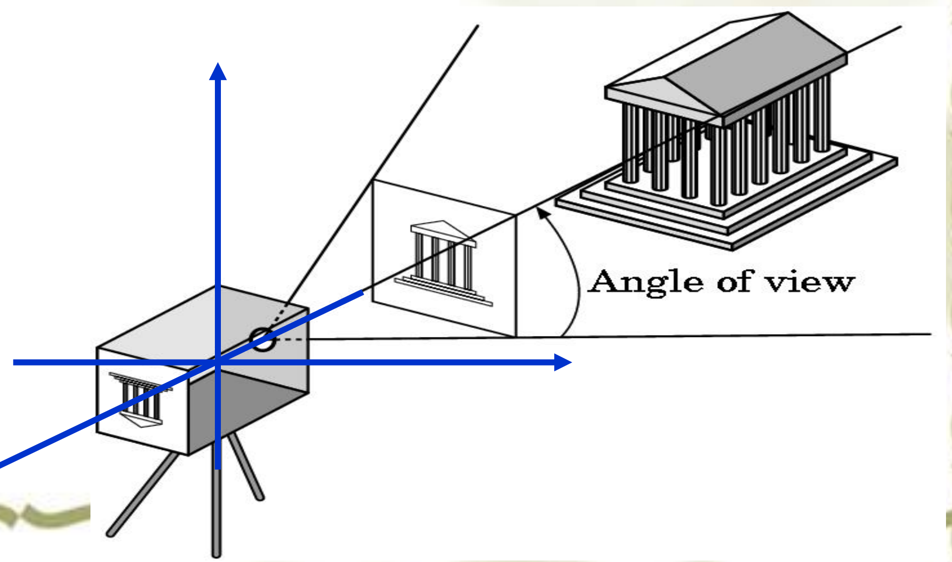
- Your models in their coordinates are natural to be defined
- Modeling transformations put the parts of your models together properly into a world space by translations, scales and rotations
- All the parts of a scene are placed in a single 3D world coordinate system





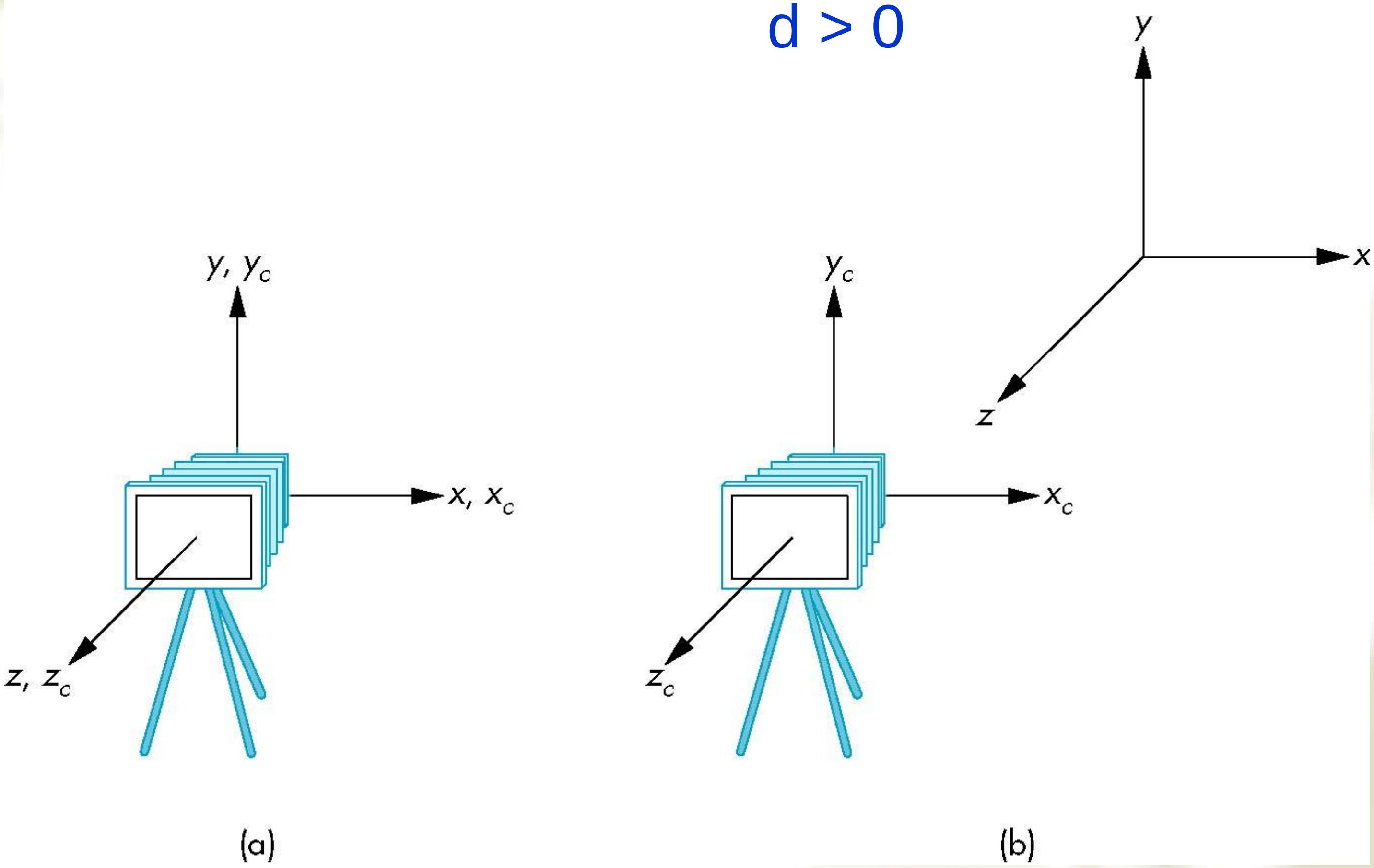
Viewing Transformations

- A scene becomes an image when there is a viewer and a viewing context
- A viewer (or camera) is placed in the world space with a position and orientation
- The camera is located at origin and points in the negative z direction by default in OpenGL
- Camera can view the scene in anywhere



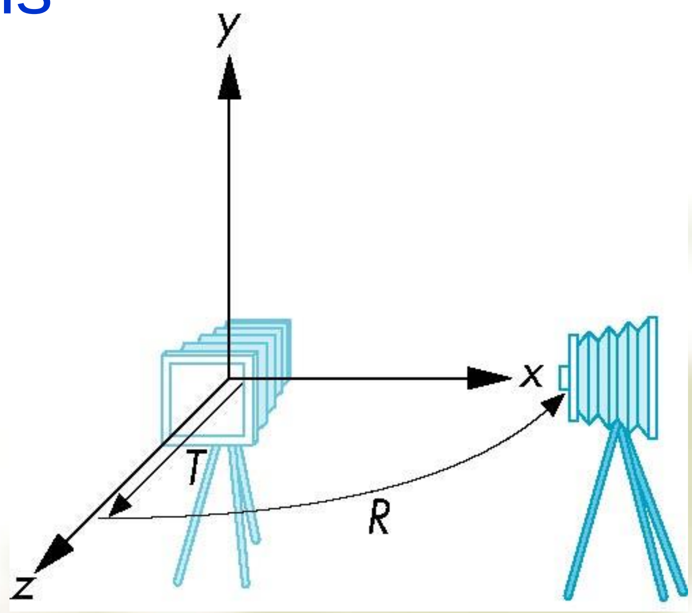
Moving Camera back from Origin

coordinates after translation by –d
d > 0



Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view from x-axis
  - Rotate the camera
  - Move it away from origin
  - Viewing matrix C = TR

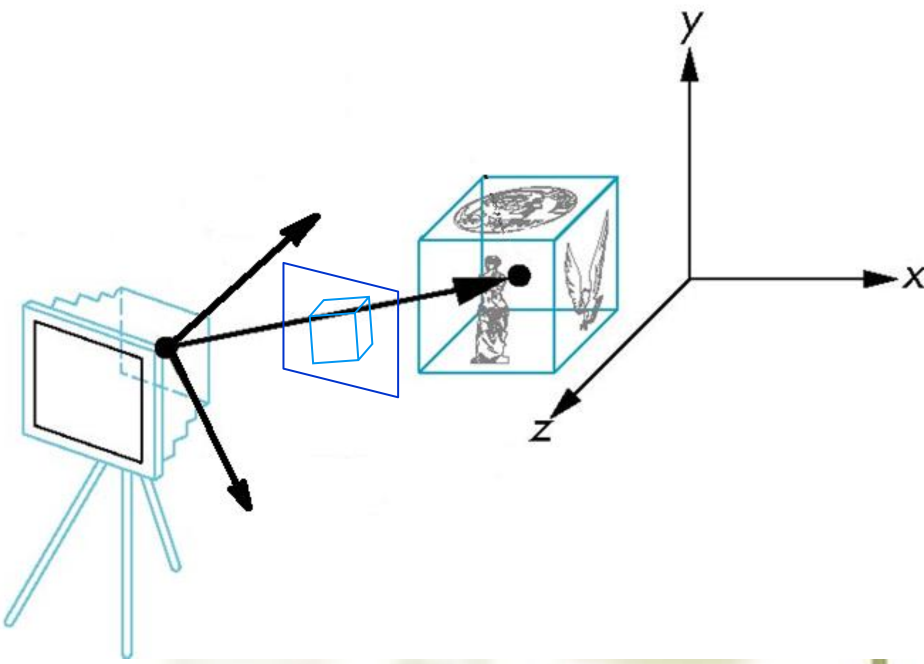


Camera in Anywhere

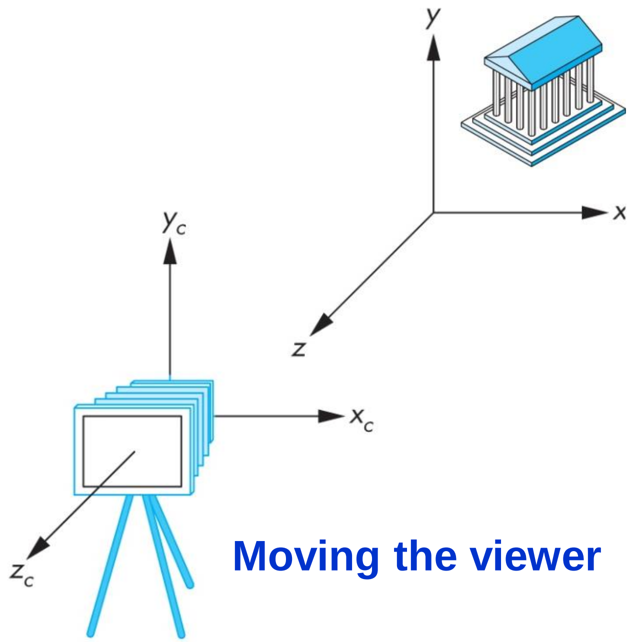
- Where is the projection plane?
- Transform the objects of the world space into eye space because projection is transformed in eye space
- Viewing Matrix

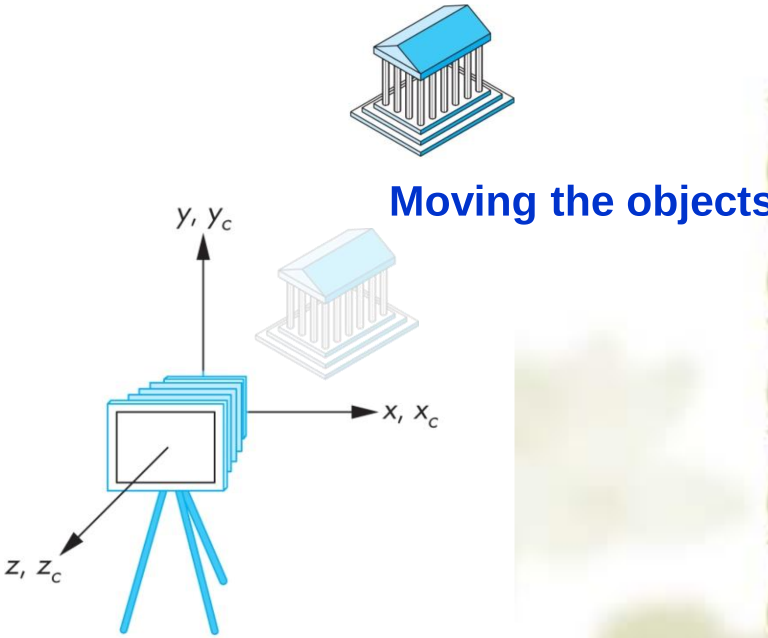
  $M = R_2 * R_1 * T$

- Can be done by rotations and translations or easier to use a LookAt function



Viewing and Modeling Transformations





Viewing transformation:
LookAt(`eyex, eyey, eyez,`
`atx, aty, atz,`
`upx,upy,upz`);

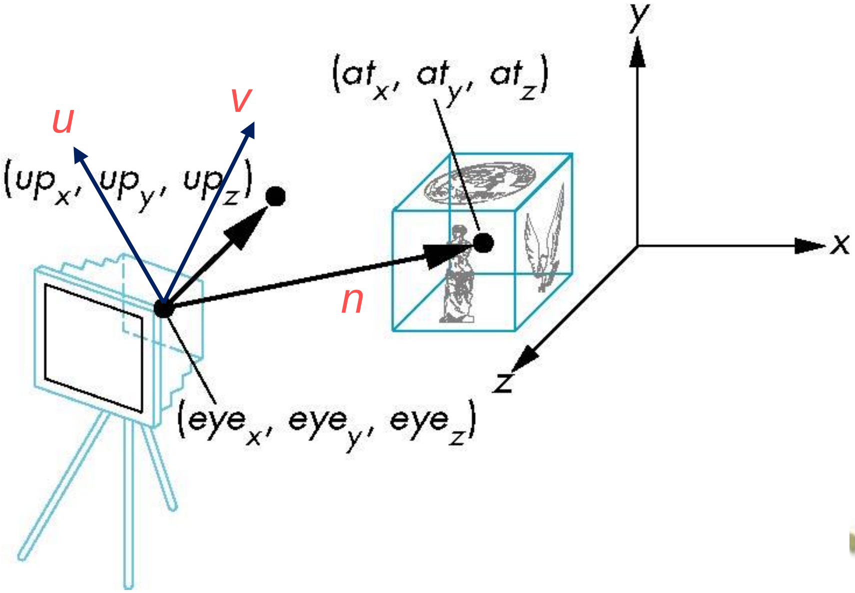
Modeling transformation:
Translate(`tx, ty, tz`);

Viewing Transformation by LookAt

`LookAt(eye_x, eye_y, eye_z, at_x, at_y, at_z, up_x, up_y, up_z)`

For eye coordinate system:

1. ***eye (View Reference Point,VRP)*** as an origin, (eye_x, eye_y, eye_z)
2. **View-Plane Normal** *n* as a *z*-axis, VPN=at - eye, *n* = VPN / |VPN|
3. The x-axis (the third vector *u*) as a vector perpendicular to *n* and *up* by cross product: $u = up \times n \,/\, |\, up \times n \,|$
4. **View-*UP vector(VUP)*** *v* as a *y*-axis, $v = n \times u \,/\, |\, n \times u \,|$



If translate, and rotate twice along z-axis and y-axis:

$$M = R_y(\beta)\, R_z(\alpha)\, T$$

The eye space and the world space overlap

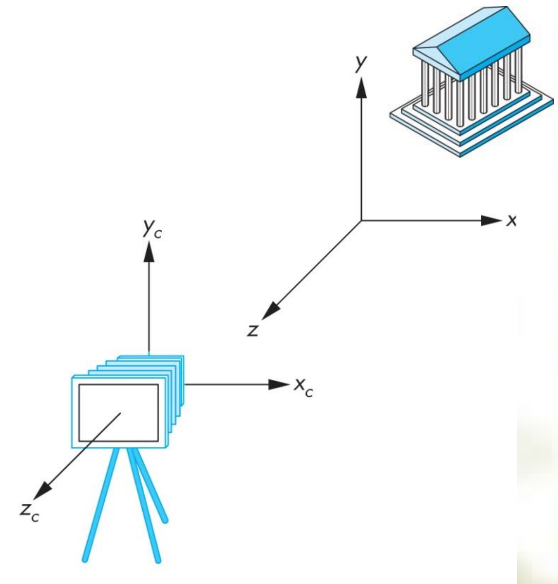
Viewing Matrix

- Here we use vectors *u*,*v* and *n* to represent the viewing matrix
- VRP: $E_0=(e_x,e_y,e_z)$

$$T=\begin{bmatrix}1 & 0 & 0 & -e_x\\ 0 & 1 & 0 & -e_y\\ 0 & 0 & 1 & -e_z\\ 0 & 0 & 0 & 1\end{bmatrix}$$

- The rotation matrix:

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



- The Viewing matrix:

$$M=RT=\begin{bmatrix}u_x & u_y & u_z & 0\\ v_x & v_y & v_z & 0\\ n_x & n_y & n_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & -e_x\\ 0 & 1 & 0 & -e_y\\ 0 & 0 & 1 & -e_z\\ 0 & 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}u_x & u_y & u_z & -u\cdot E_0\\ v_x & v_y & v_z & -v\cdot E_0\\ n_x & n_y & n_z & -n\cdot E_0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

p'=M*p* the point *p* in the world space is transformed into *p*' in eye space

Other Viewing APIs

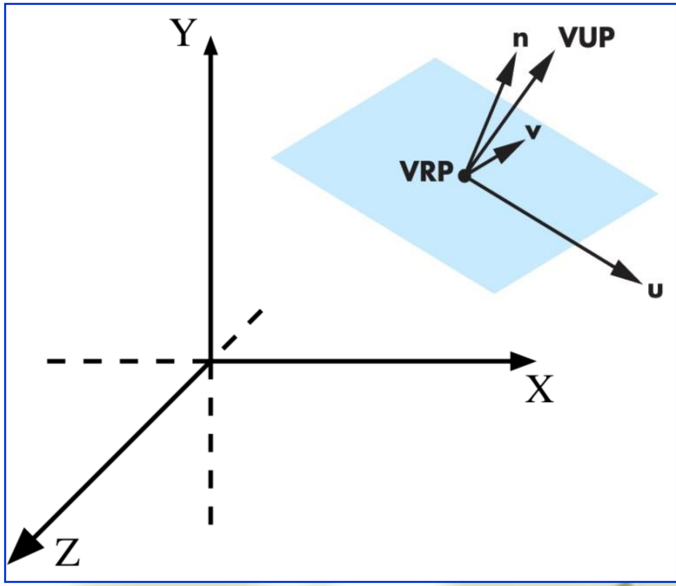
- The LookAt function is only one possible API for positioning the camera
- Others include
  - View Reference Point, View Plane Normal, View UP (PHIGS, GKS-3D)
  - Yaw, Pitch, Roll
  - Elevation, Azimuth, Twist
  - Direction angles

VRP, VPN and VUP Matrix

Look(eye_x, eye_y, eye_z, n_x, n_y, n_z, vup_x, vup_y, vup_z);

For eye coordinate system in PHIGS, GKS-3D:

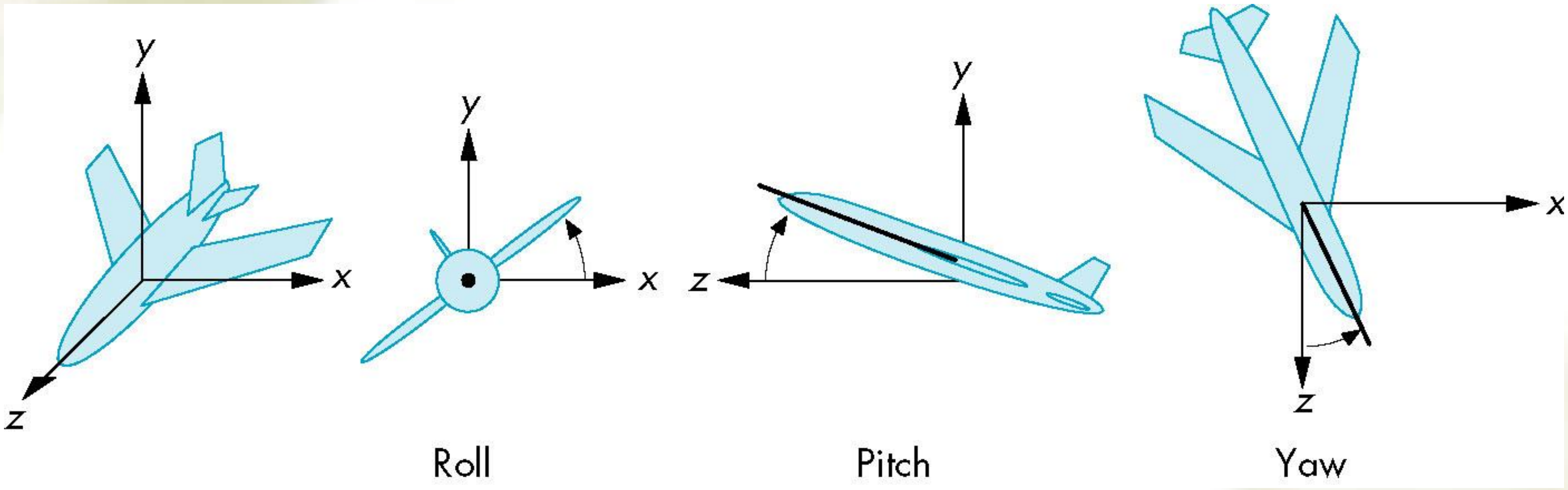
1. ***VRP: View Reference Point*** as an origin
2. ***VPN: View-Plane Normal*** *n* as a *z*-axis, normal of projection face
3. ***VUP: View-UP*** *v* as a *y*-axis, can not be parallel with projection face
4. The x-axis (the third vector *u*) can be computed with a cross product: $u = v \times n$



Viewing matrix：

$$M = \begin{bmatrix} u_x & u_y & u_z & -xu_x - yu_y - zu_z \\ v_x & v_y & v_z & -xv_x - yv_y - zv_z \\ n_x & n_y & n_z & -xn_x - yn_y - zn_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Yaw, Pitch, Roll

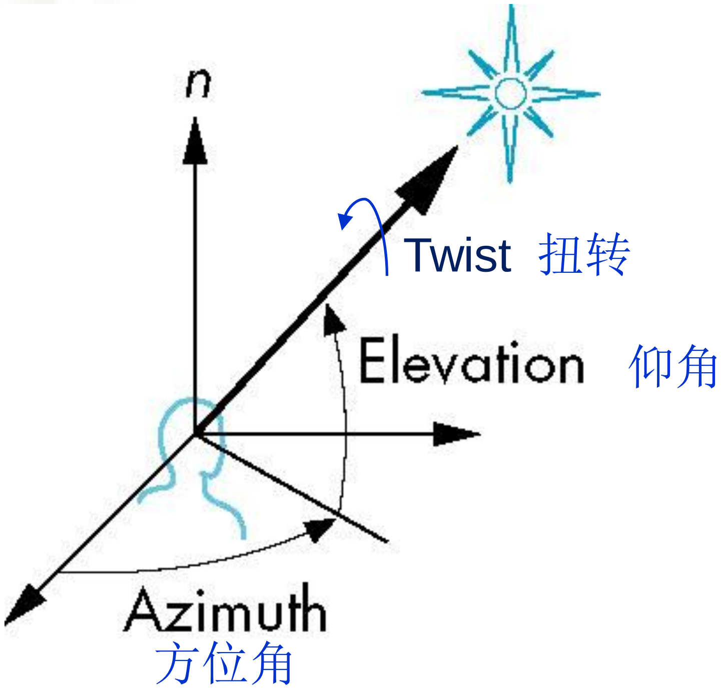


侧滚　　倾斜　　偏转

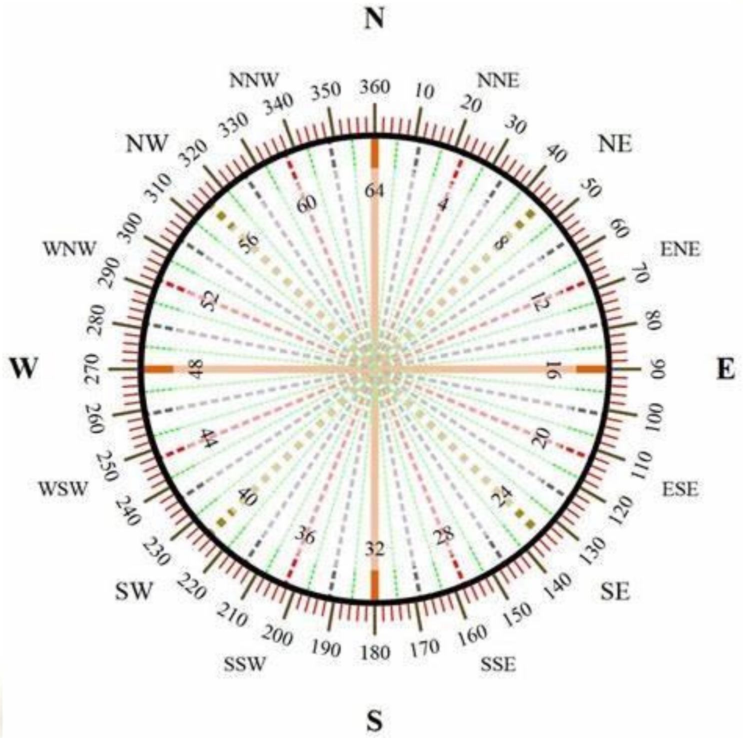
How to represent a viewing matrix in terms of these parameters?

Elevation, Azimuth, Twist

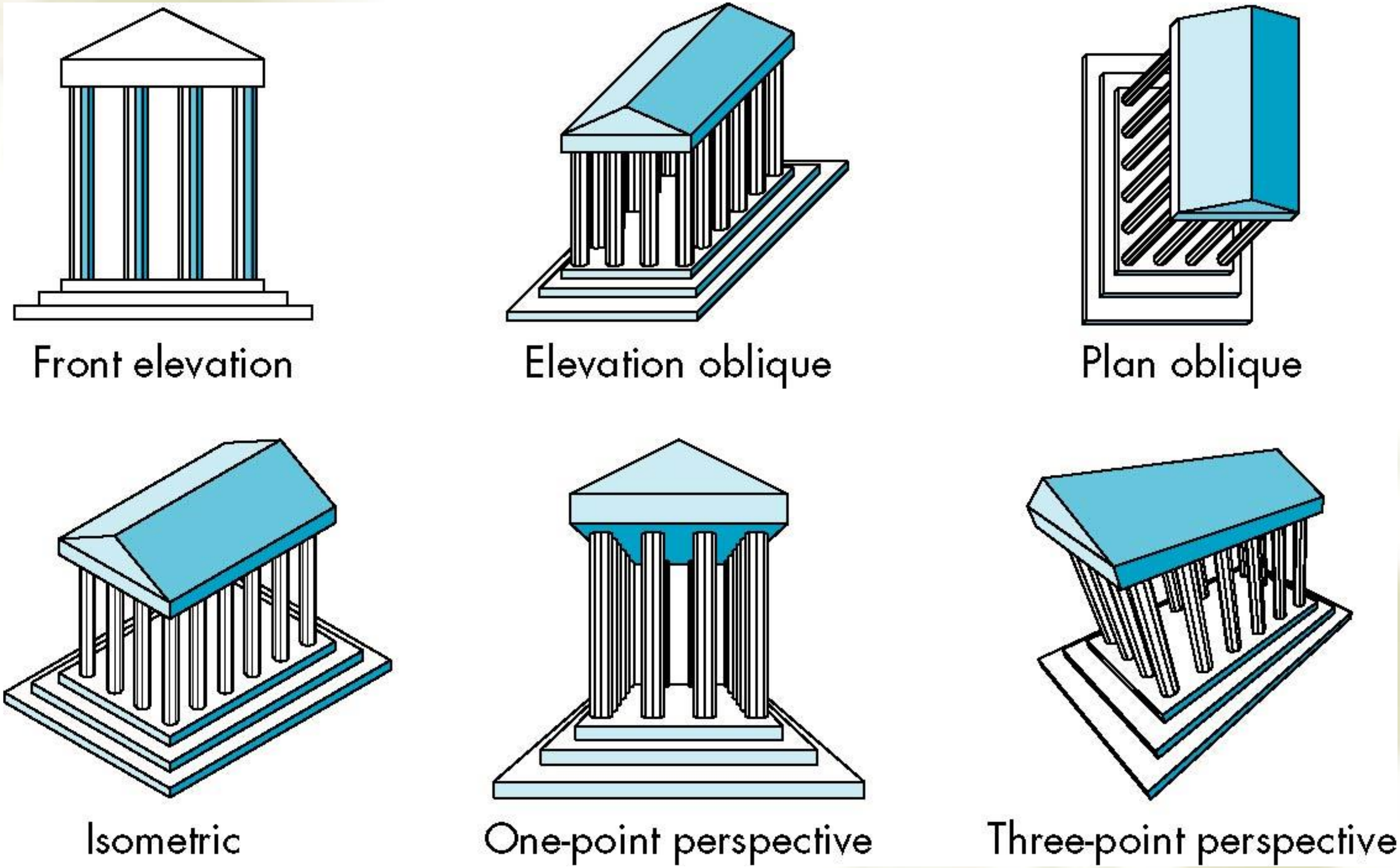
Elevation, Azimuth, Twist



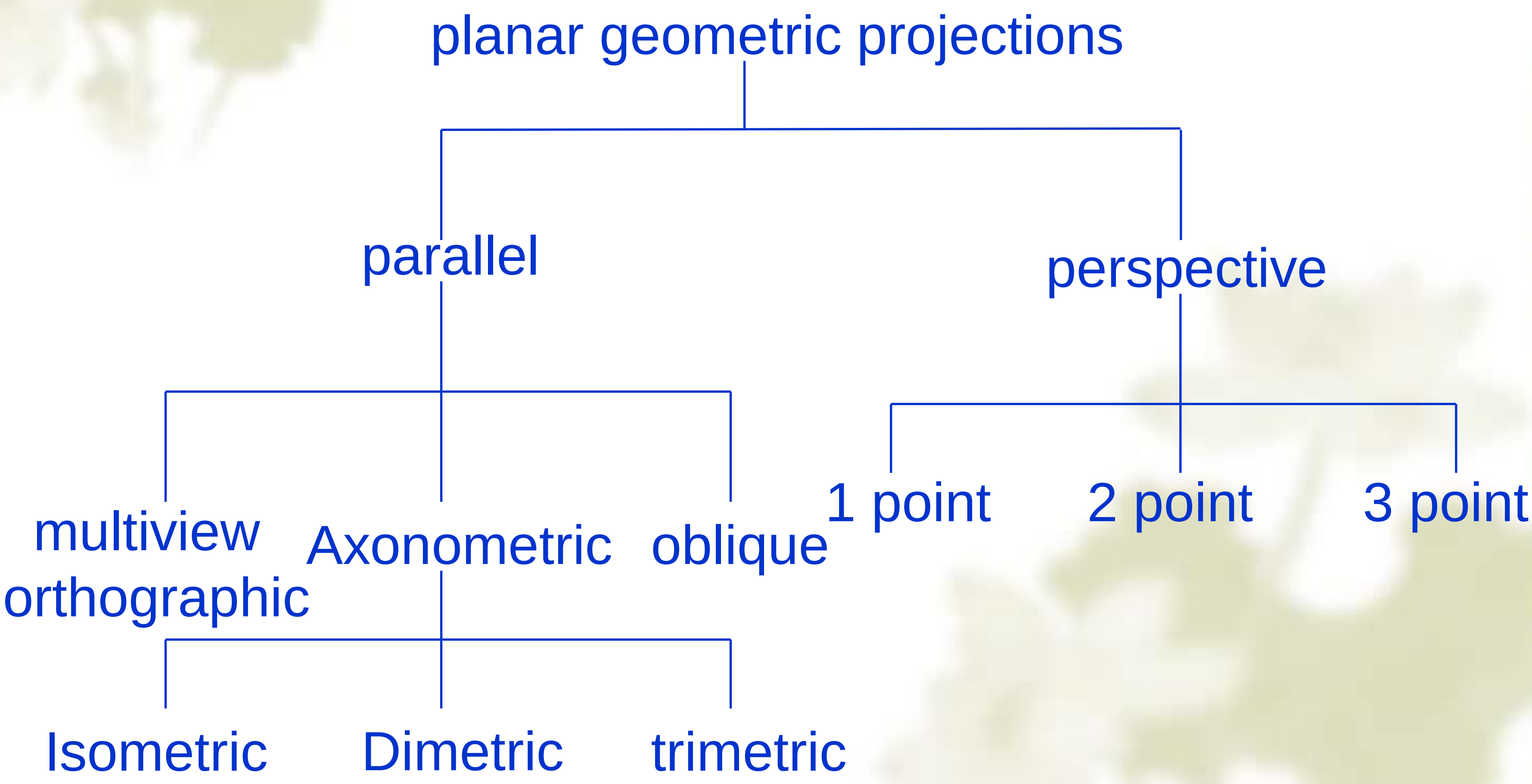
Direction angles



Classical Projections



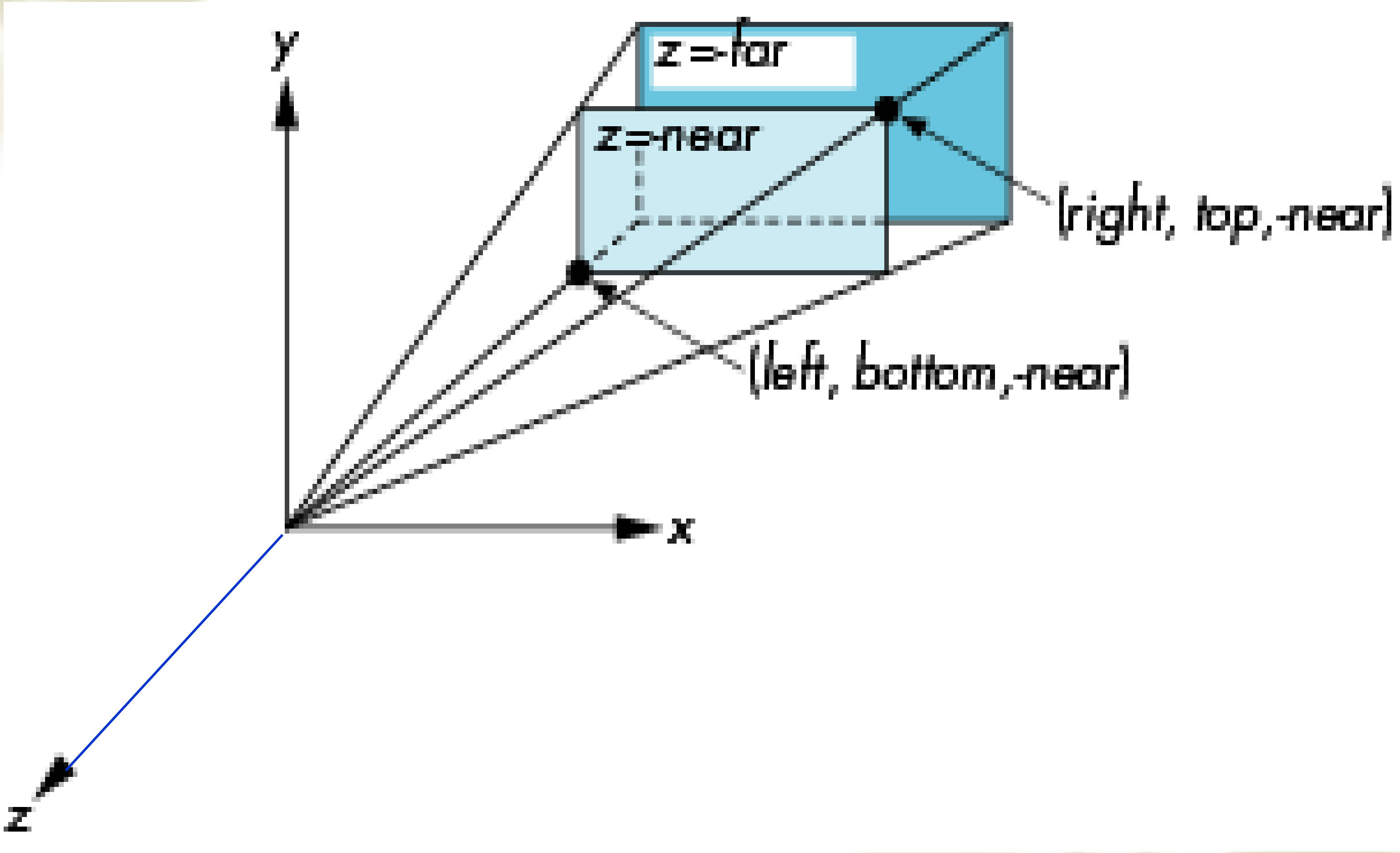
Taxonomy of Planar Geometric Projections



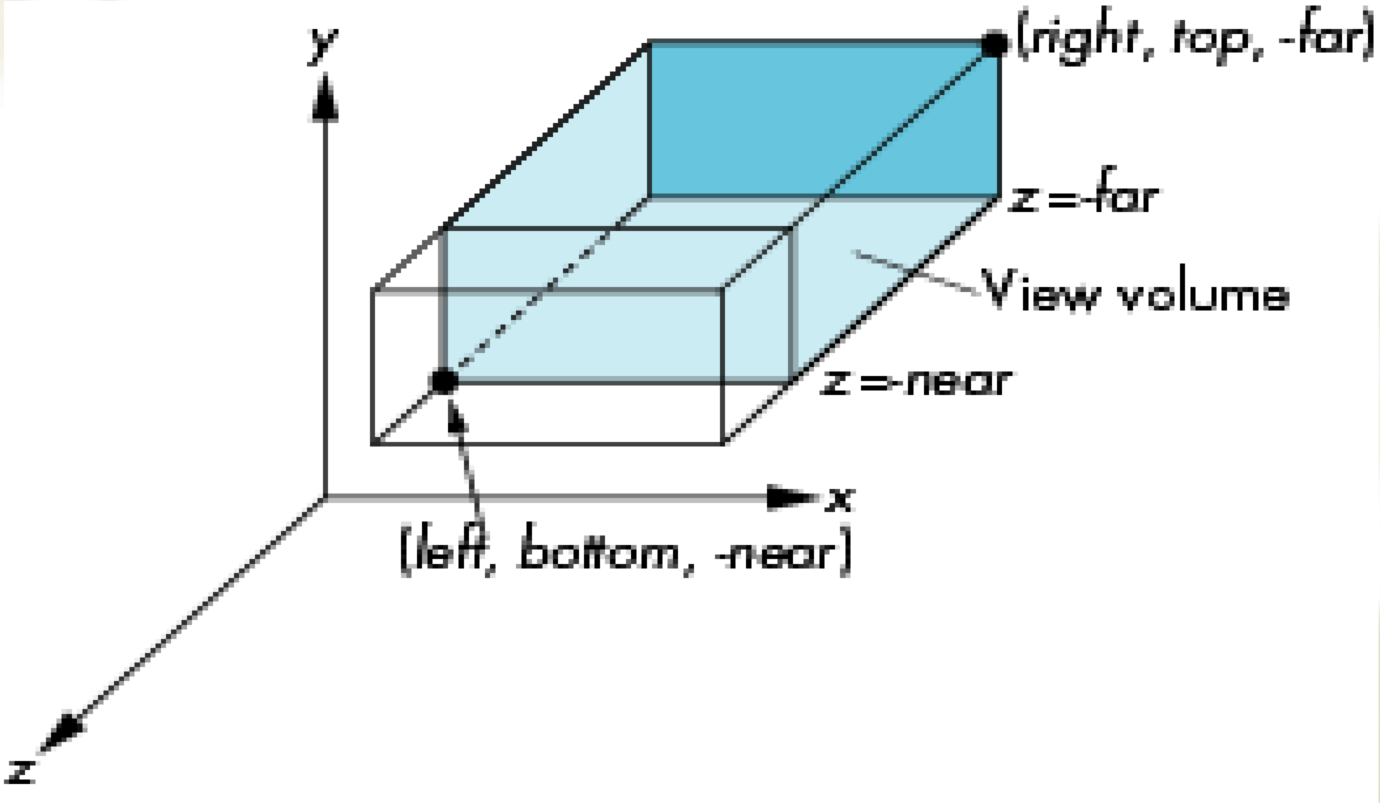
Perspective vs Parallel

- Classical viewing developed different techniques for drawing each type of projection
- Computer graphics treats all projections the same and implements them with a single pipeline
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing

Perspective View Volume

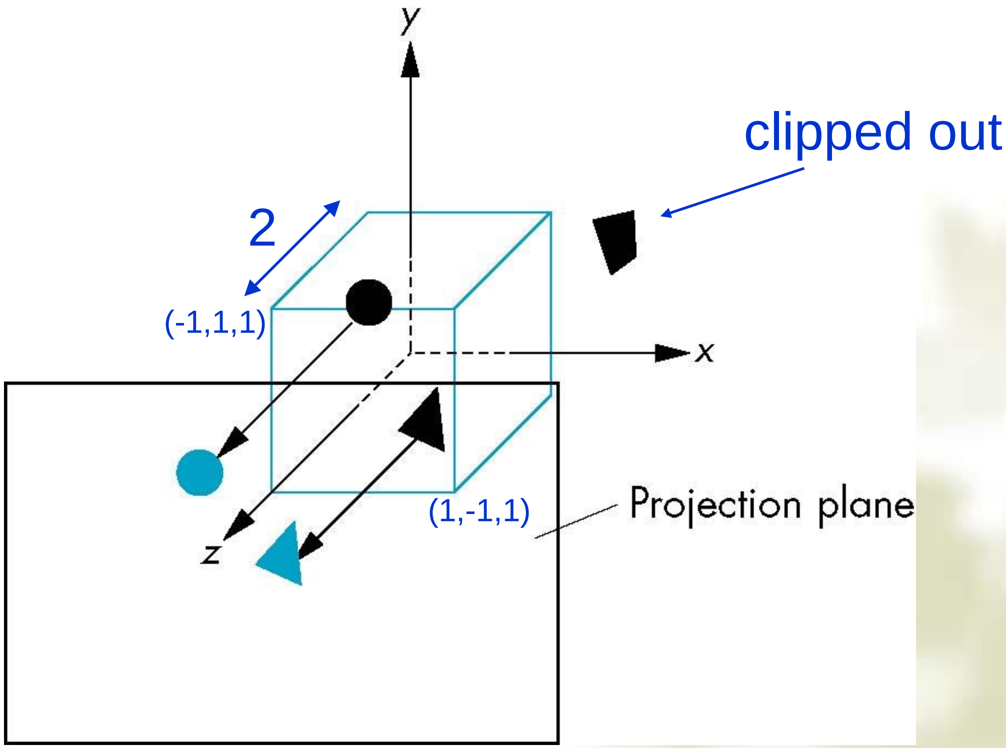


Orthographic View Volume



Default Projection

Default projection is normalized and orthogonal



OpenGL Orthogonal Viewing

To define the view volume for clipping when projection

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

Six planes of view volume:

$$x = right$$

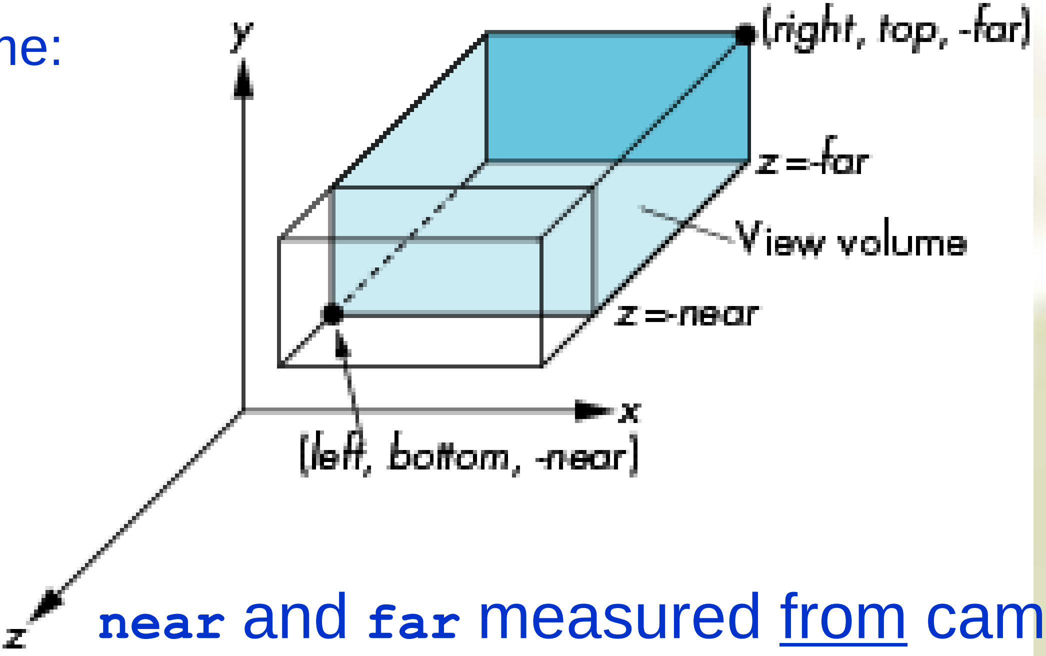
$$x = left$$

$$y = top$$

$$y = bottom$$

$$z_{\mathrm{min}} = -near$$

$$z_{\mathrm{max}} = -far$$

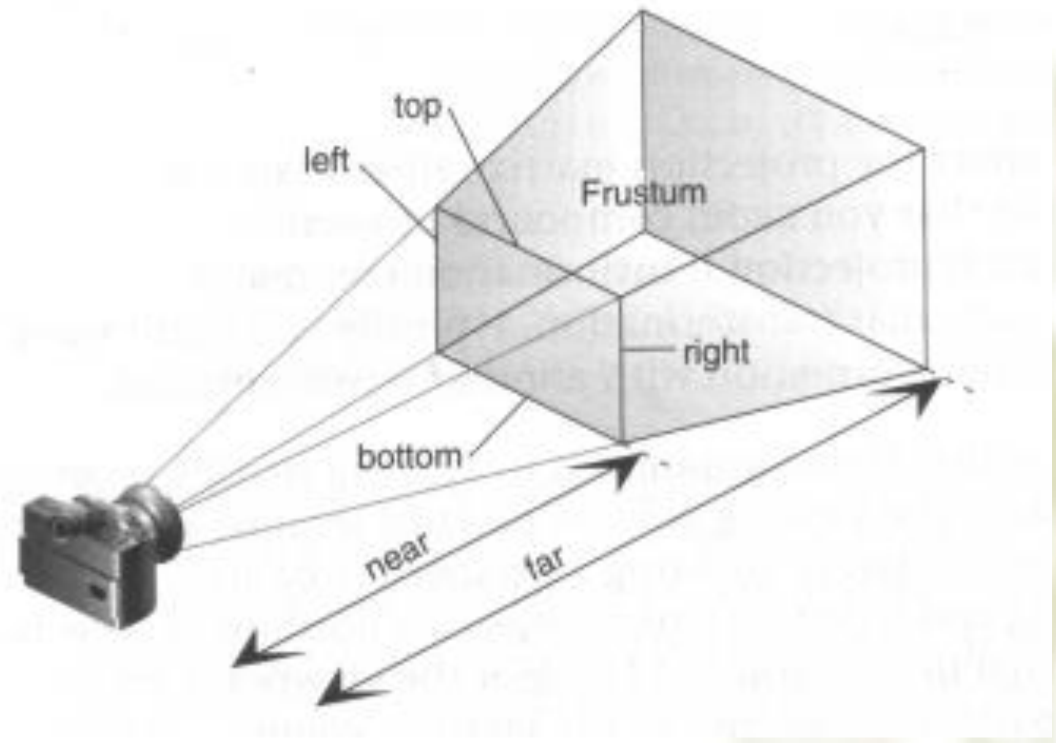


`near` and **`far`** measured <u>from</u> camera

OpenGL Perspective Viewing

To define the view volume for clipping when projection

```
void glFrustum(GLdouble left,GLdouble Right,GLdouble
 bottom,GLdouble top,GLdouble near,GLdouble far);
```

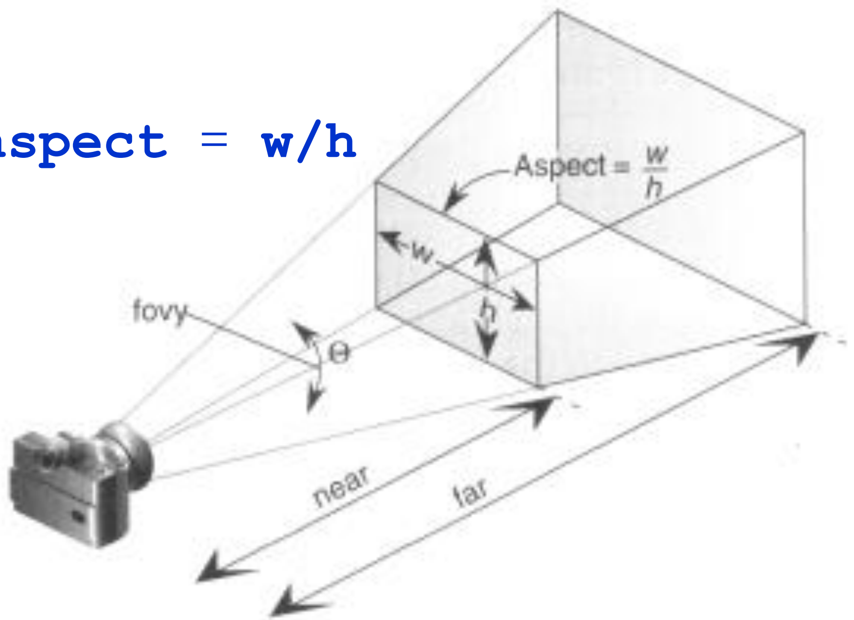


Using Field of View for Perspective

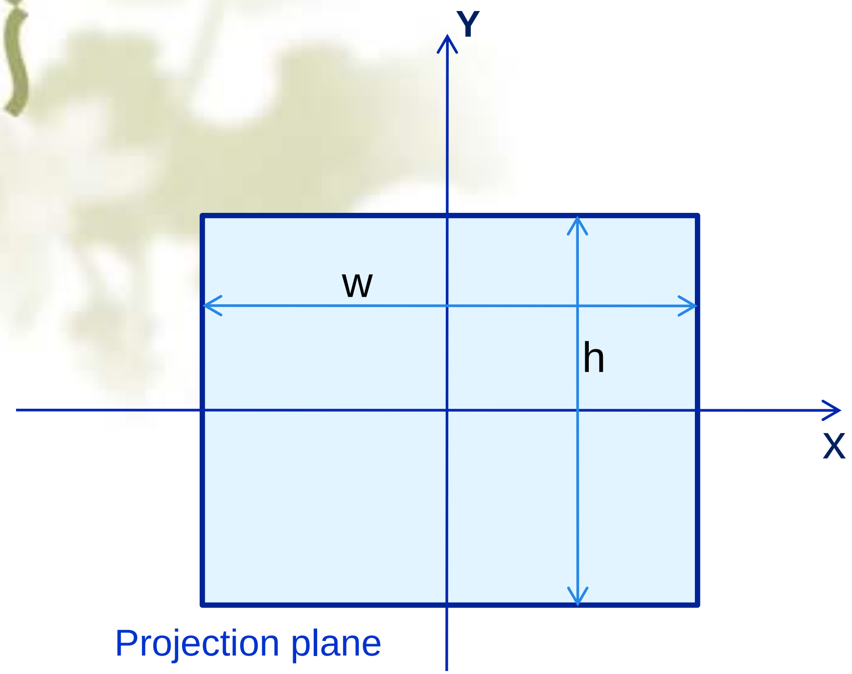
- With **`glFrustum`** it is often difficult to get the desired view
- Field of view often provides a better interface

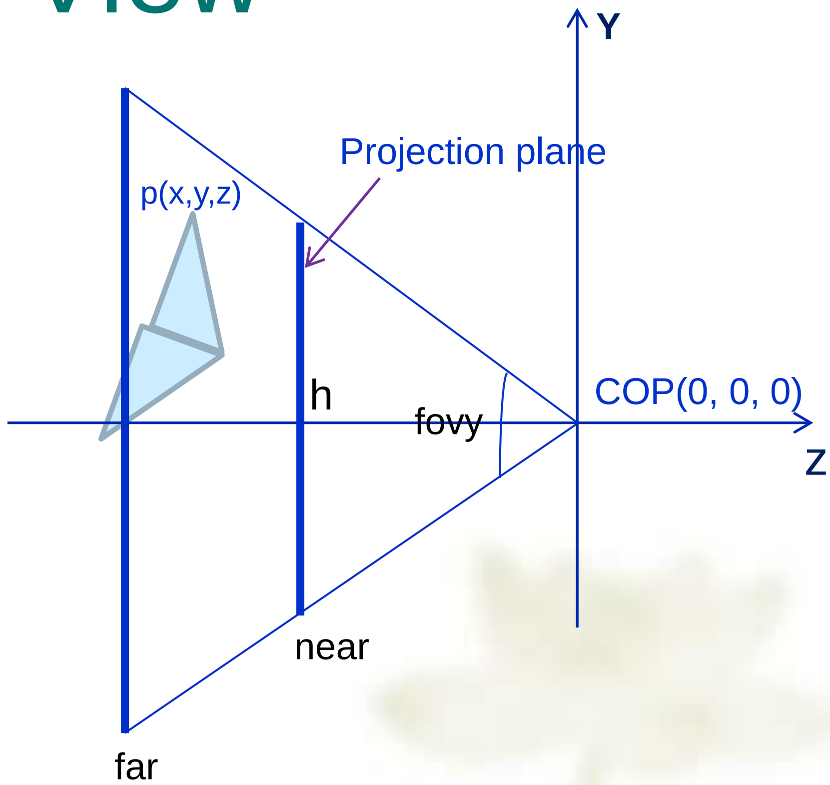
```
void gluPerspective(GLdouble fovy, GLdouble aspect,
GLdouble zNear, GLdouble zFar);
```

`aspect = w/h`



Field of View





perspective(fovy, aspect, near, far)

where, aspect=w/h, and 0<near<far

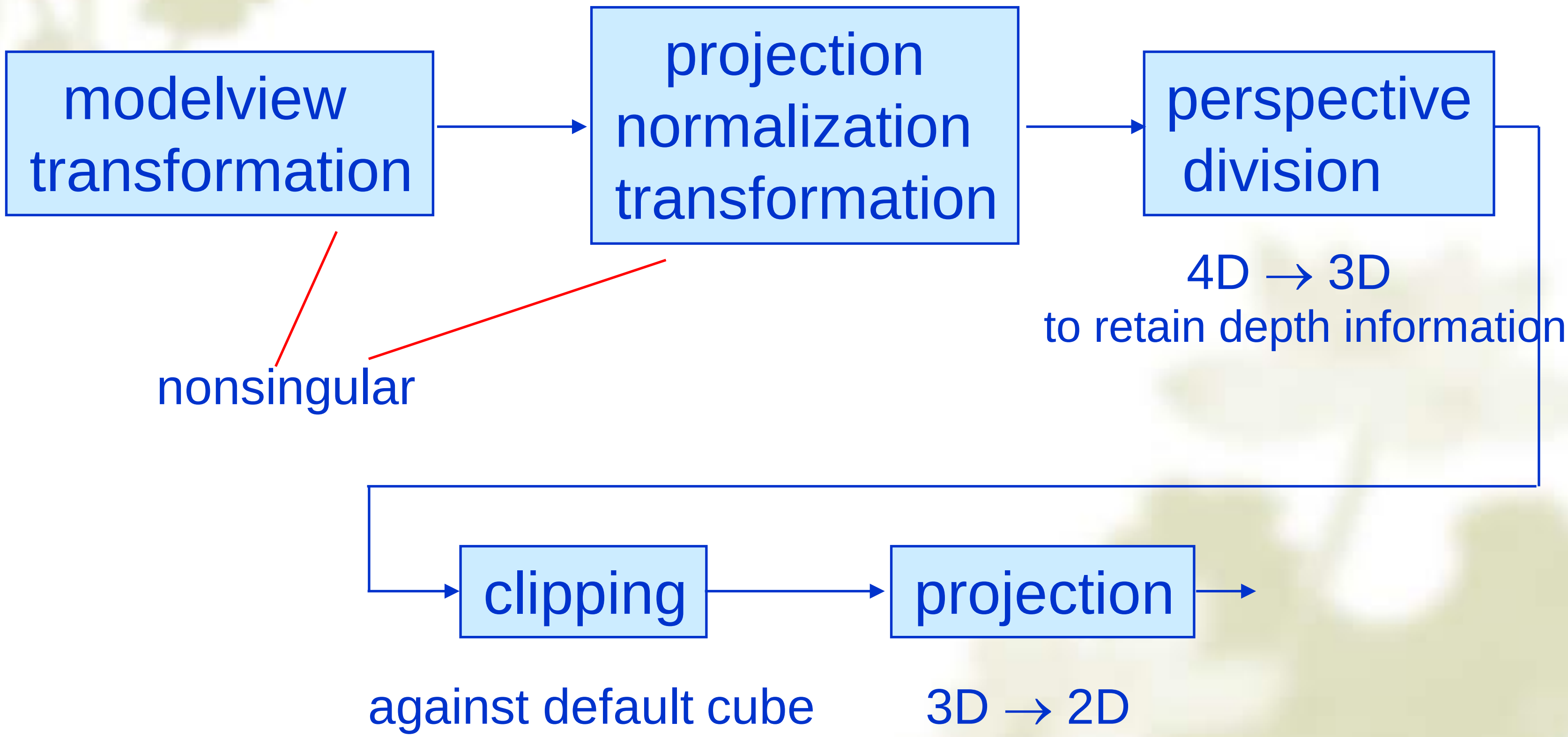
So, top=h/2=near*tg(fovy/2)

right=w/2=(aspect*h)/2

Normalization of View Volume

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to **orthogonal projections** with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Pipeline View



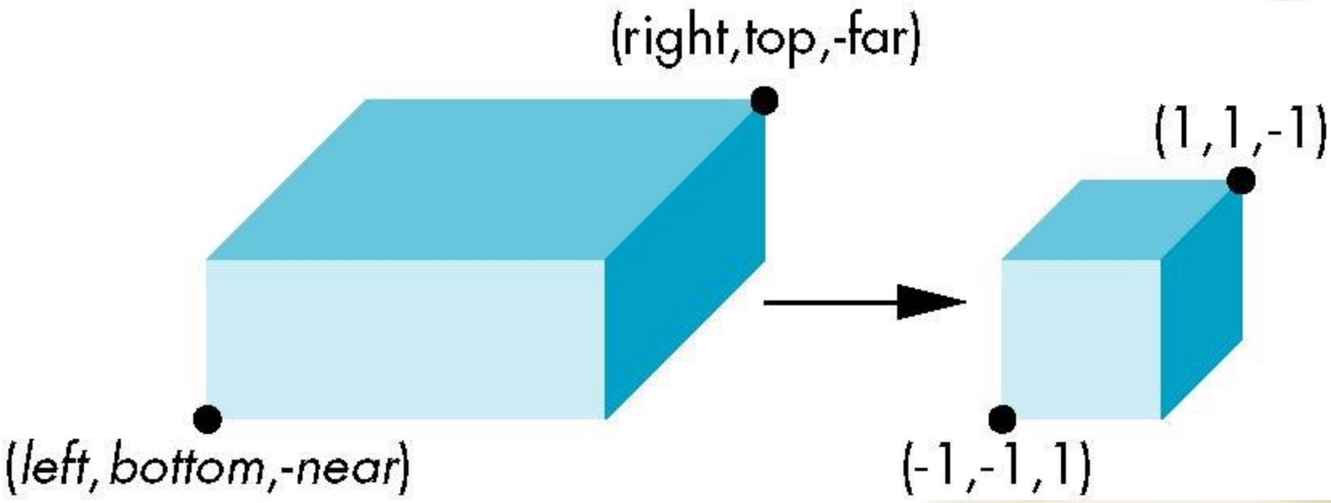
Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible

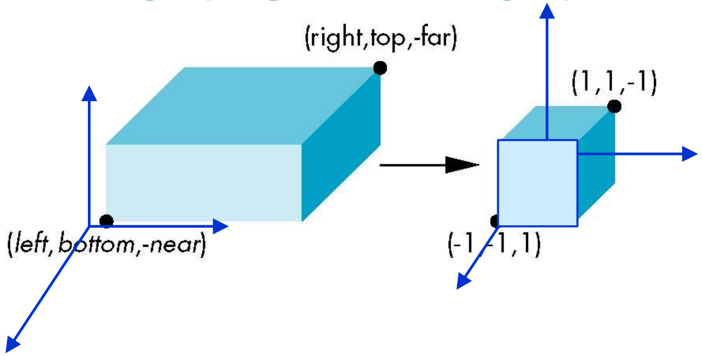
Orthogonal Normalization

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

normalization ⇒ find transformation to convert specified clipping volume to default



Orthogonal 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))

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

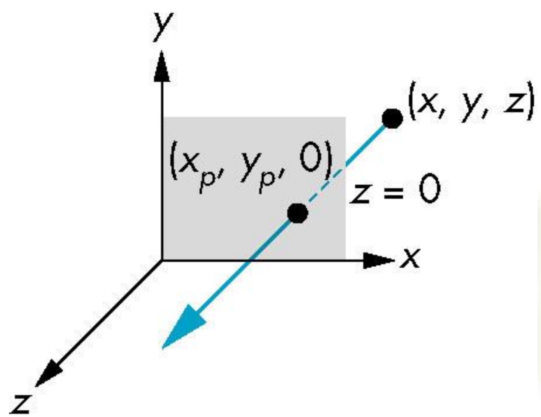
Ortho(left,right,bottom,top,Near,Far)--code

```
mat4 Ortho( const GLfloat left, const GLfloat right, const
GLfloat bottom, const GLfloat top,    const GLfloat zNear,
const GLfloat zFar )
{
    mat4 c;
    c[0][0] = 2.0/(right - left);
    c[1][1] = 2.0/(top - bottom);
    c[2][2] = 2.0/(zNear - zFar);
    c[3][3] = 1.0;
    c[0][3] = -(right + left)/(right - left);
    c[1][3] = -(top + bottom)/(top - bottom);
    c[2][3] = -(zFar + zNear)/(zFar - zNear);
    return c;
}
```

Final Normalization Orthogonal

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

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



- Hence, general orthogonal projection in 4D is

$$M_p = M_{orth}ST$$

We can let $\mathbf{M}_{orth} = \mathbf{I}$ and set the z term to zero later

Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at z = -1(d=-1), and a 90 degree field of view determined by the planes

$$x = \pm z, y = \pm z$$

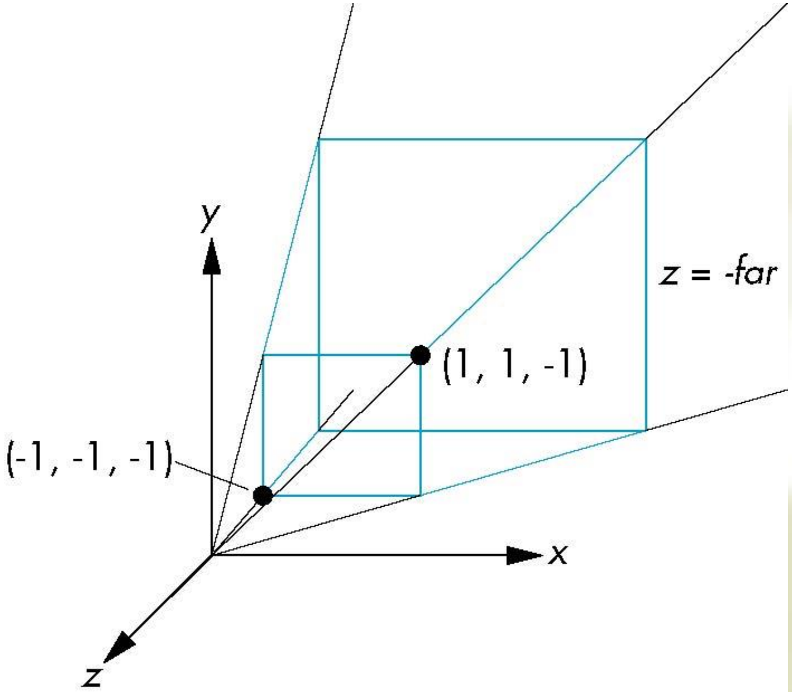
six planes of view volume:

$x = \pm z$,

$y = \pm z$

$z = -1$

$z = -far$



Perspective Matrices

Simple projection matrix in homogeneous coordinates, z= -1

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

Note that this matrix is independent of the far clipping plane

Perspective Normalization Matrix

N matrix transforms the frustum into parallelepiped

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$p' = \mathrm{N}p$

Then

$x' = x$

$y' = y$

$z' = \alpha z + \beta$

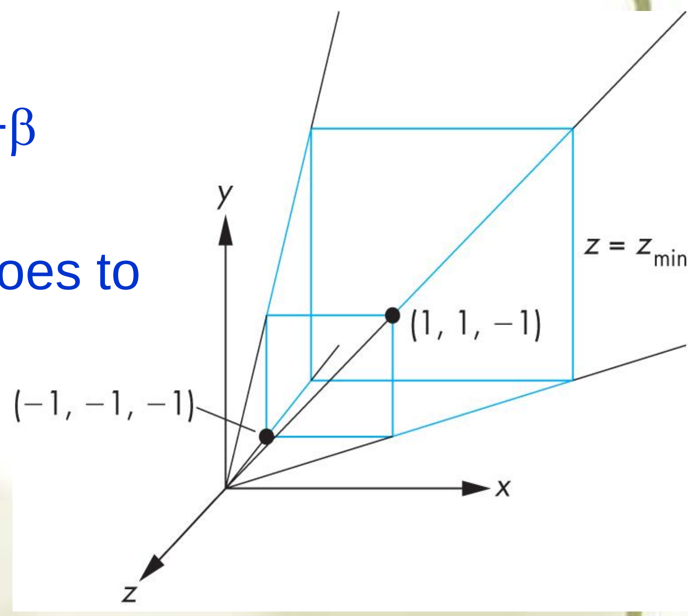
$w' = -z$

after perspective division, the point (x, y, z, 1) goes to

$x'' = -x/z$

$y'' = -y/z$

$z'' = -(\alpha+\beta/z)$



which projects orthogonally to the desired point regardless of α and β

$z'' = -(\alpha+\beta/z)$ is nonlinear but preserves the ordering of depths, if $z_1 > z_2$, then $z_1'' > z_2''$

Picking α and β

If we pick

$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

the near plane is mapped to z = -1
the far plane is mapped to z =1
and the sides are mapped to $x = \pm 1$, $y = \pm 1$

Perspective Normalization Matrix

- If we apply an orthographic projection along the z-axis to N, the matrix is

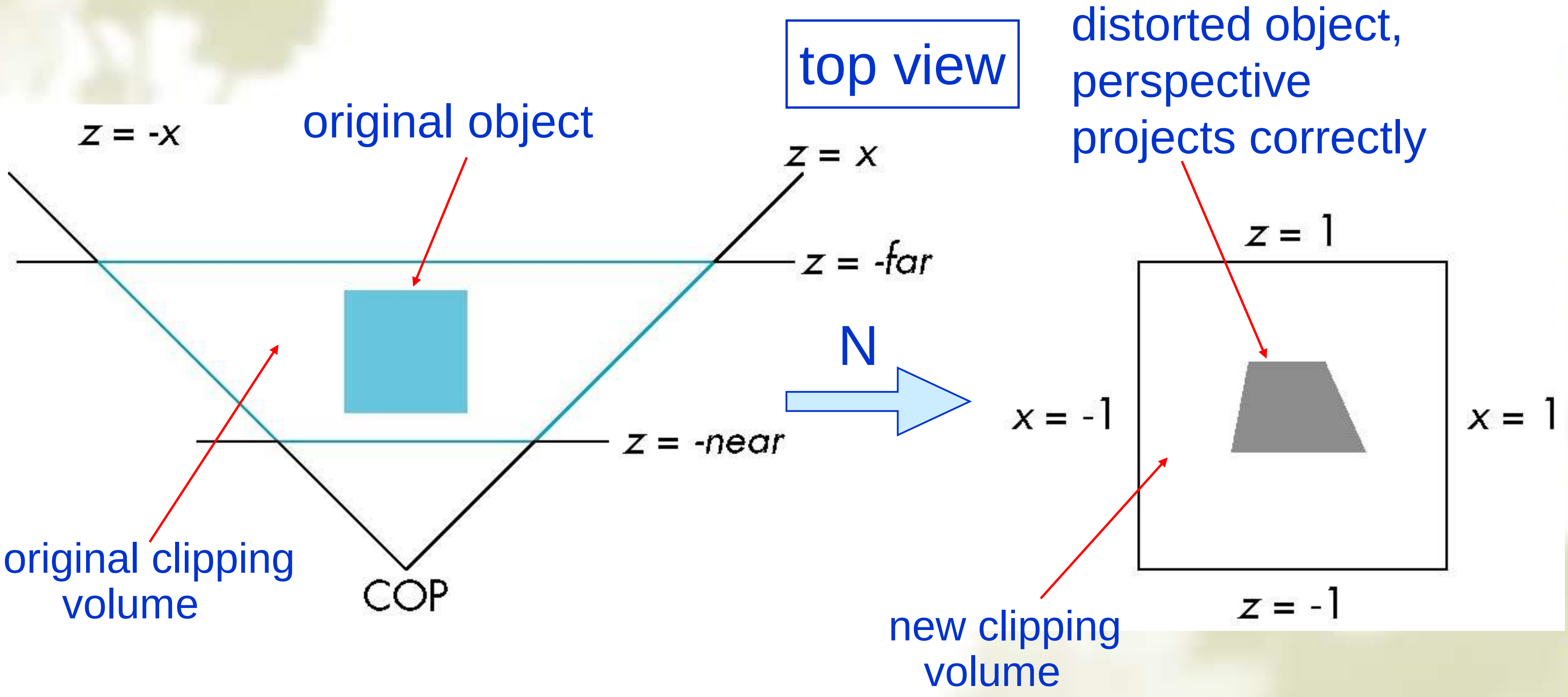
$$M_p = M_{orth} N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$p' = M_{orth} N p = \begin{bmatrix} x \\ y \\ 0 \\ -z \end{bmatrix}$$

- After doing the perspective division, we obtain the perspective point xp and yp

$$x_p = -\frac{x}{z} \qquad y_p = -\frac{y}{z}$$

Normalization Transformation



Hence the new clipping volume is the default clipping volume

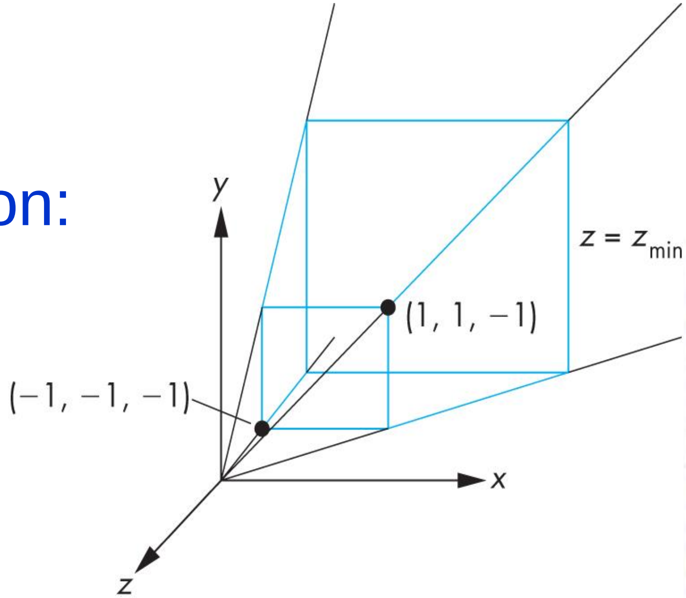
Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping

Perspective Normalization Matrix

Scaling from view volume to normalization:

$$s_x = 1/x = -2 * near/(right - left)$$
$$s_y = 1/y = -2 * near/(top - bottom)$$
$$s_z = 1$$



So,

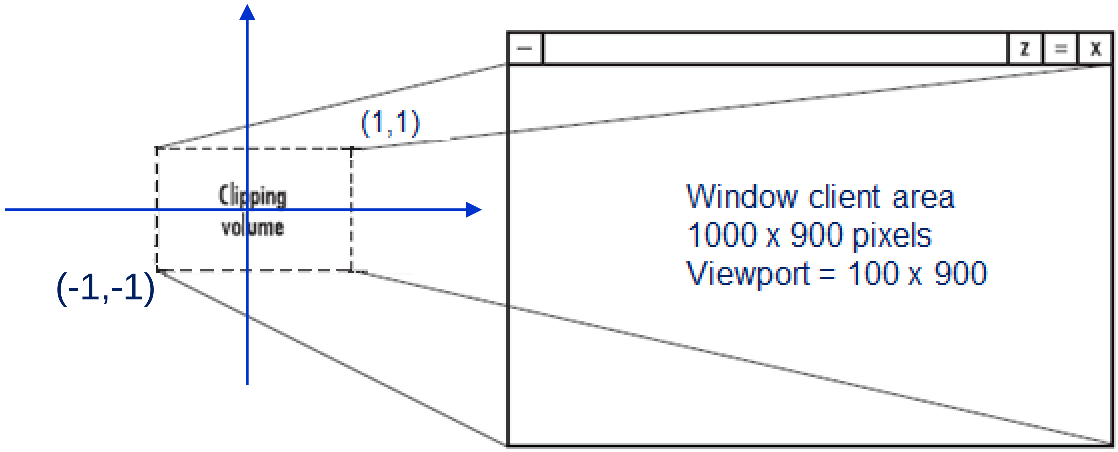
$$M_{persp} = \begin{bmatrix} \frac{2 * near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2 * near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & \frac{-2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

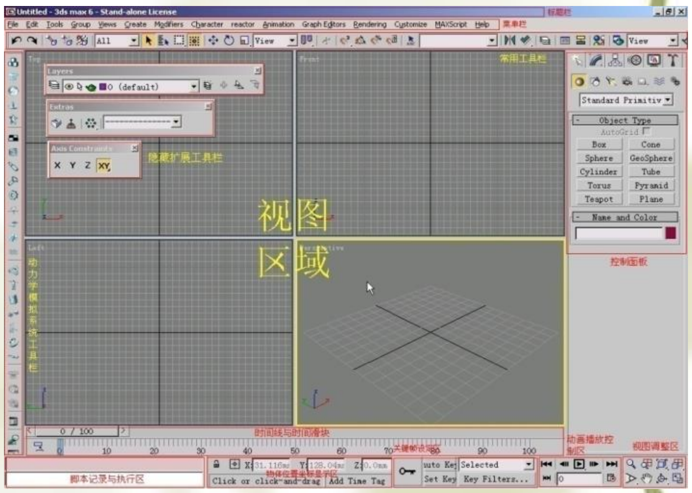
perspective(fovy, aspect, near, far)--code

```
mat4 Perspective( const GLfloat fovy, const GLfloat aspect,
                  const GLfloat zNear, const GLfloat zFar)
{
    GLfloat top   = zNear * tan(fovy*DegreesToRadians/2);
    GLfloat right = top * aspect;

    mat4 c;
    c[0][0] = zNear/right;
    c[1][1] = zNear/top;
    c[2][2] = -(zFar + zNear)/(zFar - zNear);
    c[2][3] = -2.0*zFar*zNear/(zFar - zNear);
    c[3][2] = -1.0;
    return c;
}
```

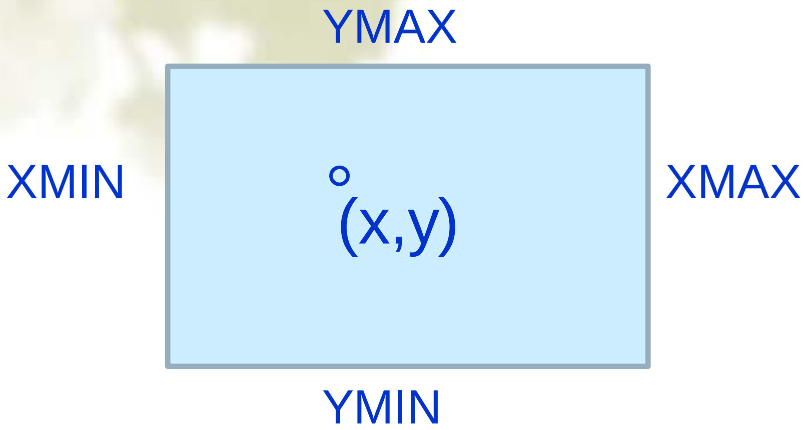
Definition of Viewport

- A viewport is a rectangular region in the window to which you can draw
- Default viewport is the entire window
- You can define a smaller viewport so all drawing is restricted to that region
- You can use separate modeling for each viewport

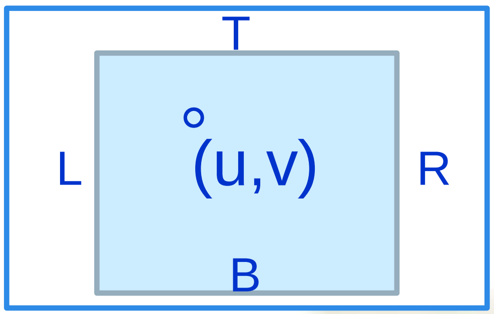




2D Window Mapping into Viewport



window



viewport

WW=XMAX – XMIN
WH=YMAX – YMIN

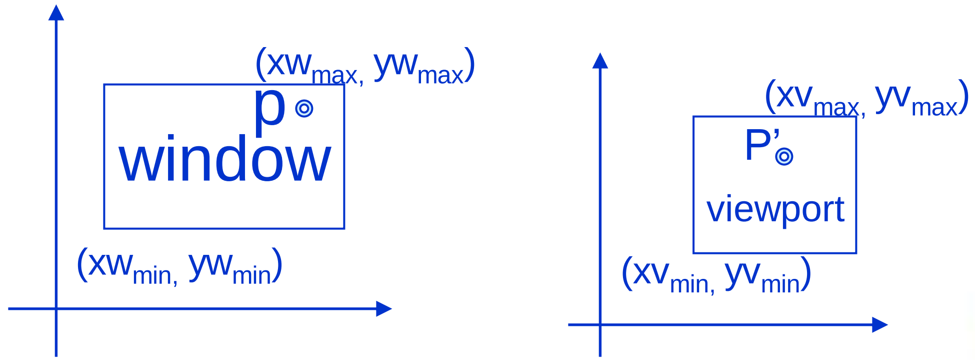
VW=R – L
VH=T – B

(x – XMIN) / WW = (u – L) / VW
(y – YMIN) / WH = (v – B) / VH

$\Longrightarrow$

u = L + (x – XMIN) * VW / WW
v = B + (y – YMIN) * VH / WH

Viewport Matrix



$$p' = T_2 S T_1 p$$

$$T_1 = \begin{pmatrix} 1 & 0 & -xw_{min} \\ 0 & 1 & -yw_{min} \\ 0 & 0 & 1 \end{pmatrix} \qquad T_2 = \begin{pmatrix} 1 & 0 & xv_{min} \\ 0 & 1 & yv_{min} \\ 0 & 0 & 1 \end{pmatrix}$$

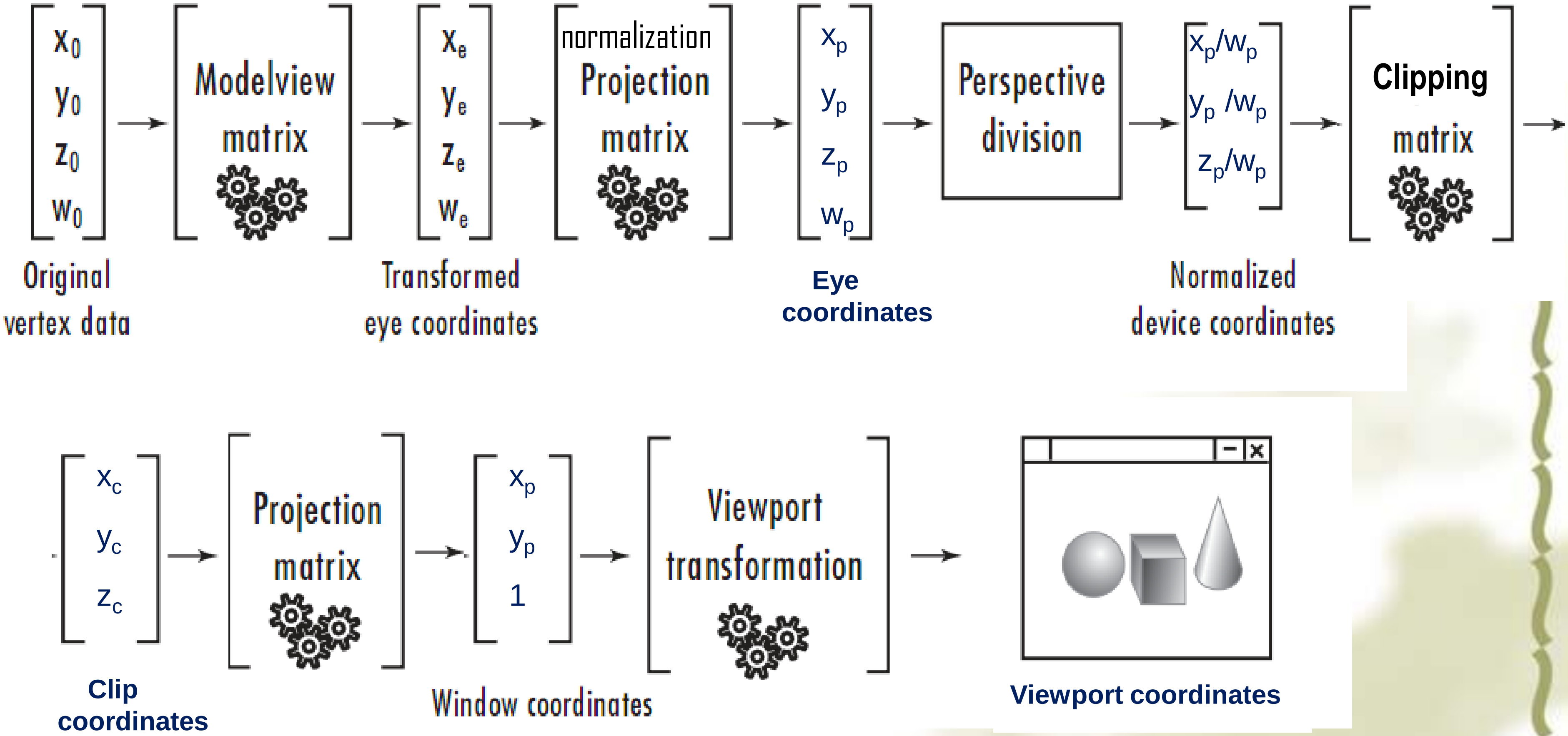
$$S = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

其中: $S_X = \dfrac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$

$$S_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

Question：if Sx≠Sy，how to make the transformation distortionless?

Sequence of OpenGL Transformation



Applications of Transformation: Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- Depending on where we place the position (direction) setting function, such as we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently
  - ……

Positioning and Moving Lights

Lights are affected by all transformations:

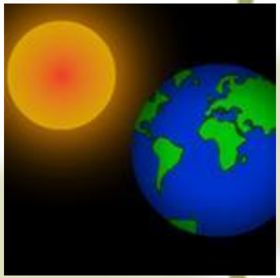
Sun light

1. The light is at a fixed place in the scene
2. The light is at a fixed place relative to the eyepoint, light's geometry is modified

   by the viewing transformation

3. The light is at a fixed place relative to an object in the scene, light's geometry is defined

   in a branch of the group node



4. The light moves around in the scene on its own
5. Move the light source(s) with the object(s)

6. Move the light source(s) and object(s) independently

Summary of Viewing Transformation

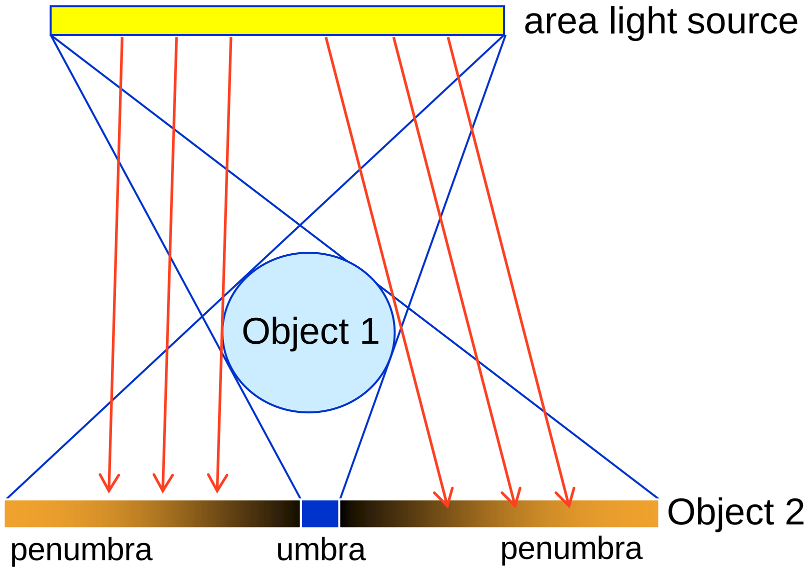
Transformation	Use
Viewing	Specifies the location of the viewer or camera
Modeling	Moves objects around the scene
Modelview	Describes the duality of viewing and modeling transformations
Projection	Normalization projection in the view volume; Perspective division; Transform 4D model into 3D window; Clipping
Viewport	Specifies the size of viewport; Transforms (scales) the geometry on the normalized window to the screen

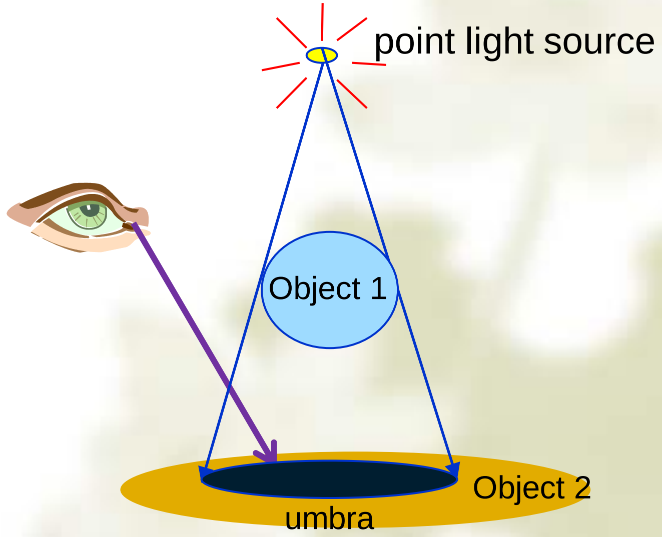
Shadows

The shadow areas are

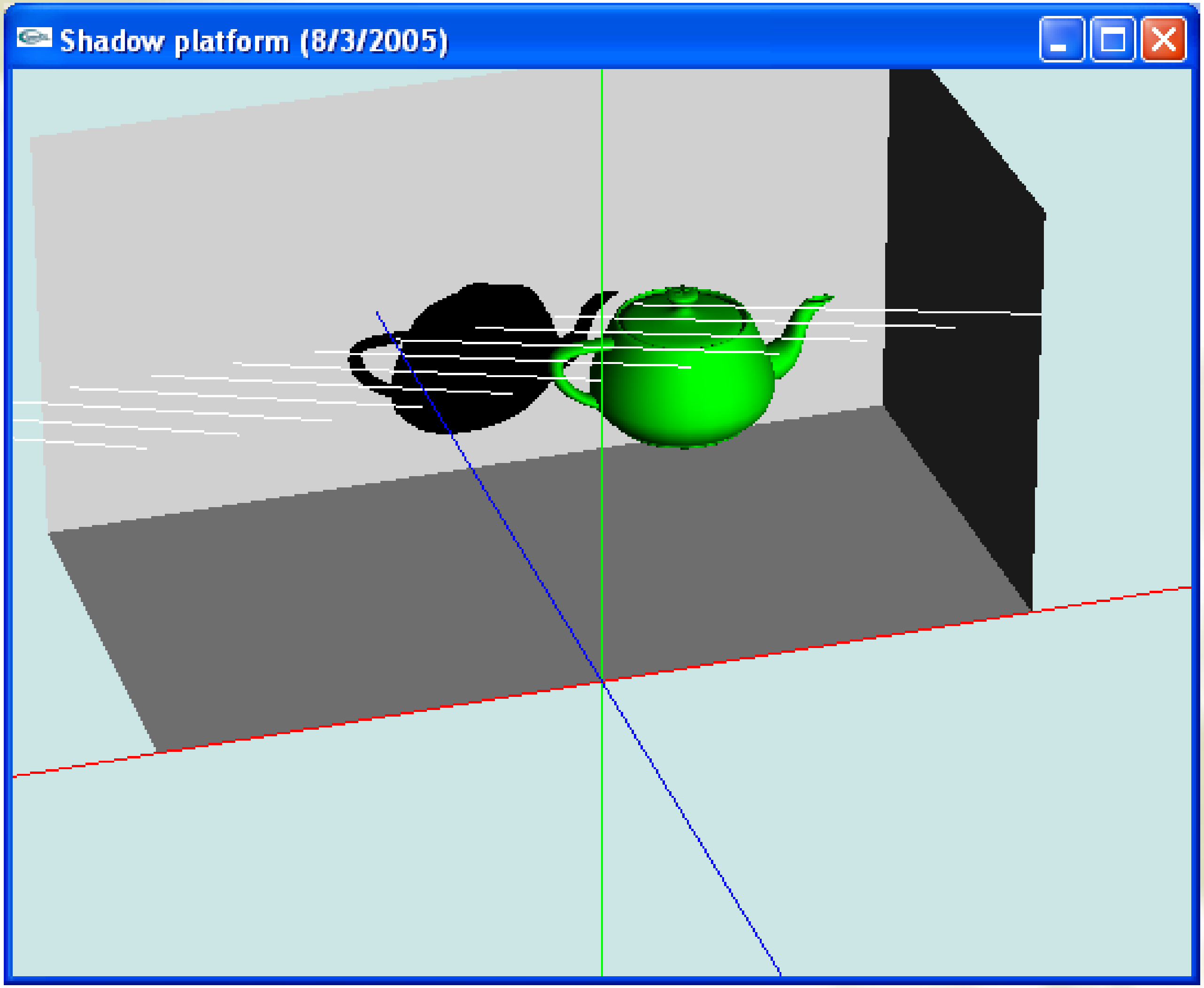
- ✓ **to be seen from the view position**
- ✓ **not to be seen from light source position**

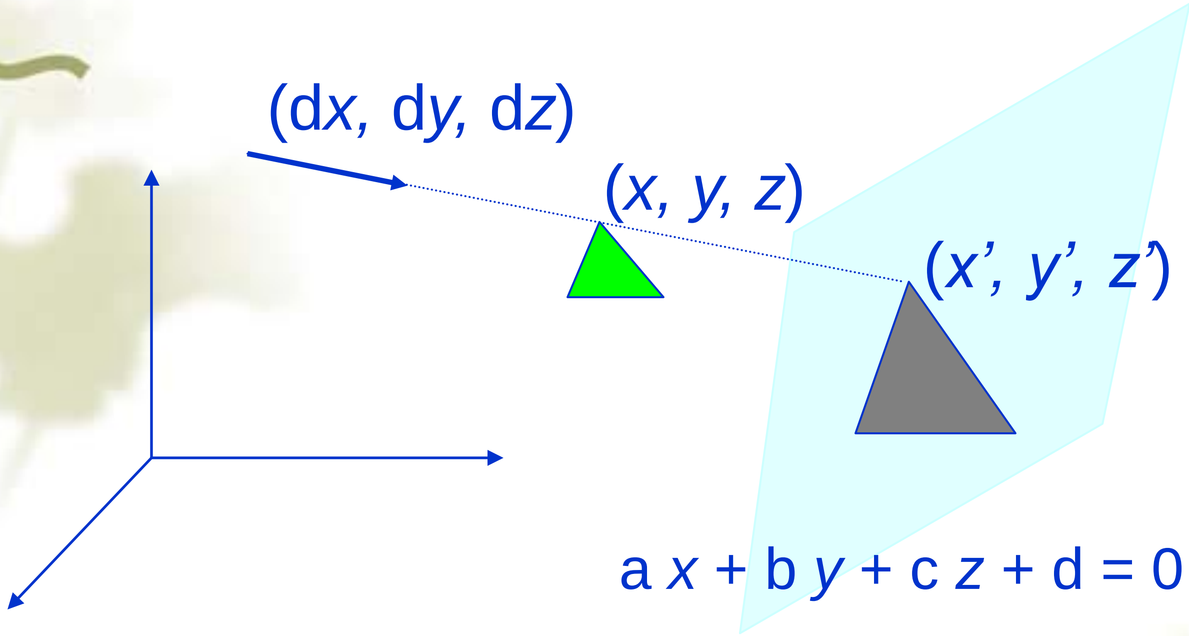
soft shadow





5.1 Shadows from a directional light





The direction of light is $(\mathrm{d}x, \mathrm{d}y, \mathrm{d}z)$.

The shadow of a vertex (x, y, z) on the plane is at (x', y', z').

Hence, we have $x' = x + \alpha\, \mathrm{d}x$, $y' = y + \alpha\, \mathrm{d}y$, $z' = z + \alpha\, \mathrm{d}z$.

Moreover, $a\,(x + \alpha\, \mathrm{d}x) + b\,(y + \alpha\, \mathrm{d}y) + c\,(z + \alpha\, \mathrm{d}z) + d = 0$.

$\alpha = -\,(a\,x + b\,y + c\,z + d) \,/\, (a\, \mathrm{d}x + b\, \mathrm{d}y + c\, \mathrm{d}z)$

$\alpha = -(a\,x + b\,y + c\,z + d) / (a\,dx + b\,dy + c\,dz)$

$$x' = x - \frac{a \times x + b \times y + c \times z + d}{a \times dx + b \times dy + c \times dz} dx$$

$$x' = \frac{(b \times dy + c \times dz)x - (b \times dx)y - (c \times dx)z - d \times dx}{a \times dx + b \times dy + c \times dz}$$

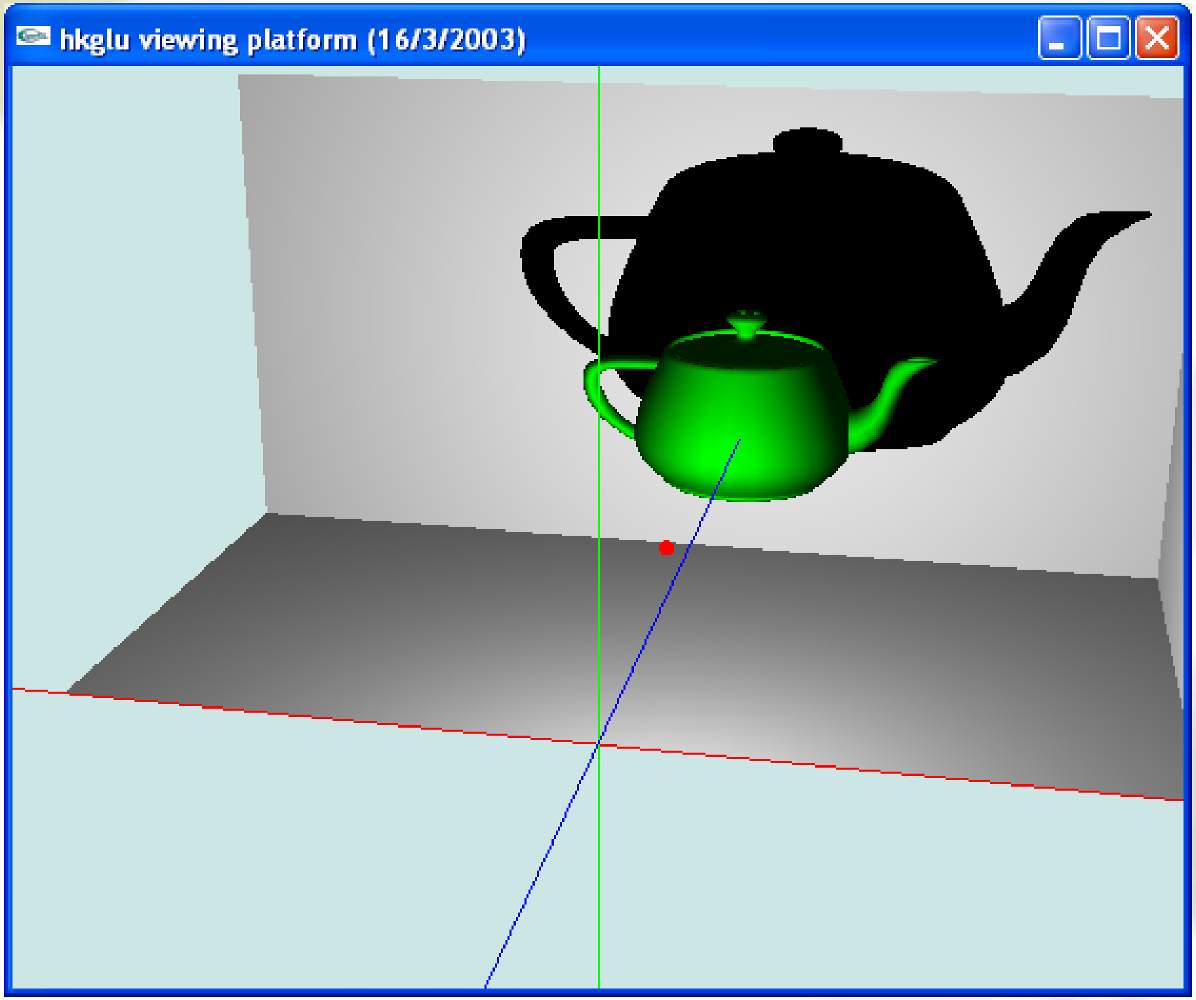
$$y' = \ldots$$

$$z' = \ldots$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} b \times dy + c \times dz & -b \times dx & -c \times dx & -d \times dx \\ -a \times dy & a \times dx + c \times dz & -c \times dy & -d \times dy \\ -a \times dz & -b \times dz & a \times dx + b \times dy & -d \times dz \\ 0 & 0 & 0 & a \times dx + b \times dy + c \times dz \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

```
//Set up the matrix M such that it projects a vertex on the plane
//      ax + by + cz + d = 0.
//      The direction of light is (dx, dy, dz).

void directionalLightShadow( double dx, double dy, double dz,
                 double a, double b, double c, double d, GLfloat M[16])
{
    M[0] = b*dy+c*dz;  M[4] = -b*dx;  M[8] = -c*dx;  M[12] = -d*dx;
    M[1] = -a*dy;  M[5] = a*dx+c*dz;  M[9] = -c*dy;  M[13] = -d*dy;
    M[2] = -a*dz;  M[6] = -b*dz;  M[10] = a*dx+b*dy;  M[14] = -d*dz;
    M[3] = 0;      M[7] = 0;      M[11] = 0;  M[15] = a*dx+b*dy+c*dz;
}
```

5.2 Shadows from a point light



Assume that the point light source is at the origin.

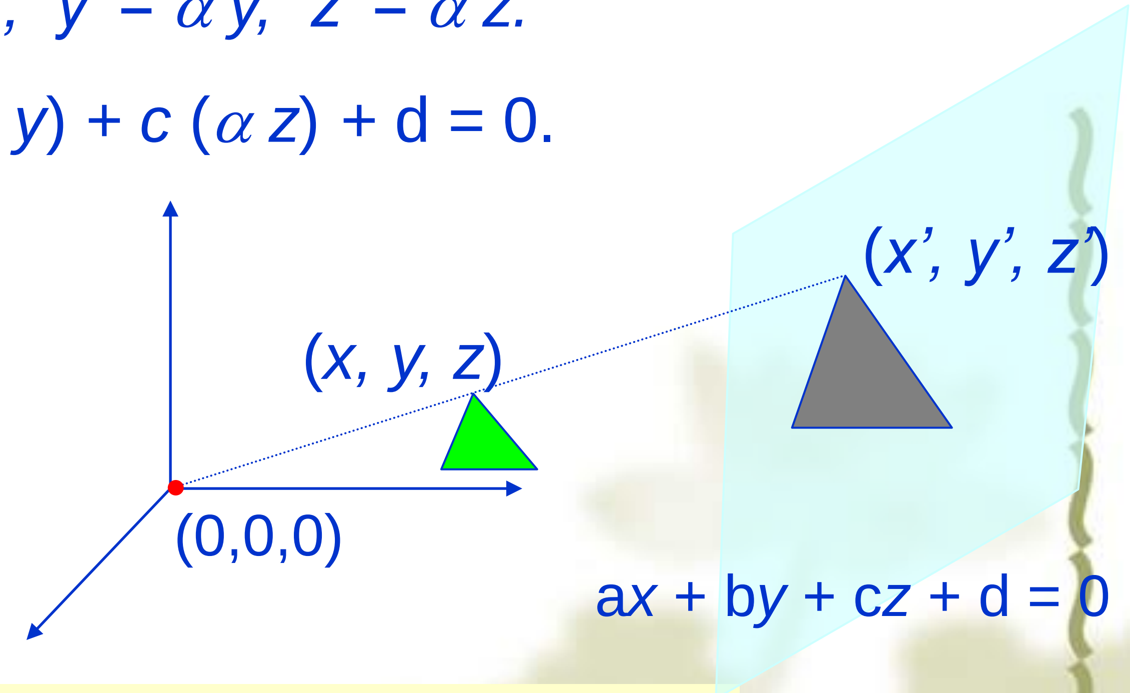
The shadow of a vertex (x, y, z) on the plane is at (x', y', z').

Hence, we have $x' = \alpha x$, $y' = \alpha y$, $z' = \alpha z$.

Moreover, $a(\alpha x) + b(\alpha y) + c(\alpha z) + d = 0$.

$\alpha = -d / (ax + by + cz)$

$x' = -d \cdot x / (ax + by + cz)$



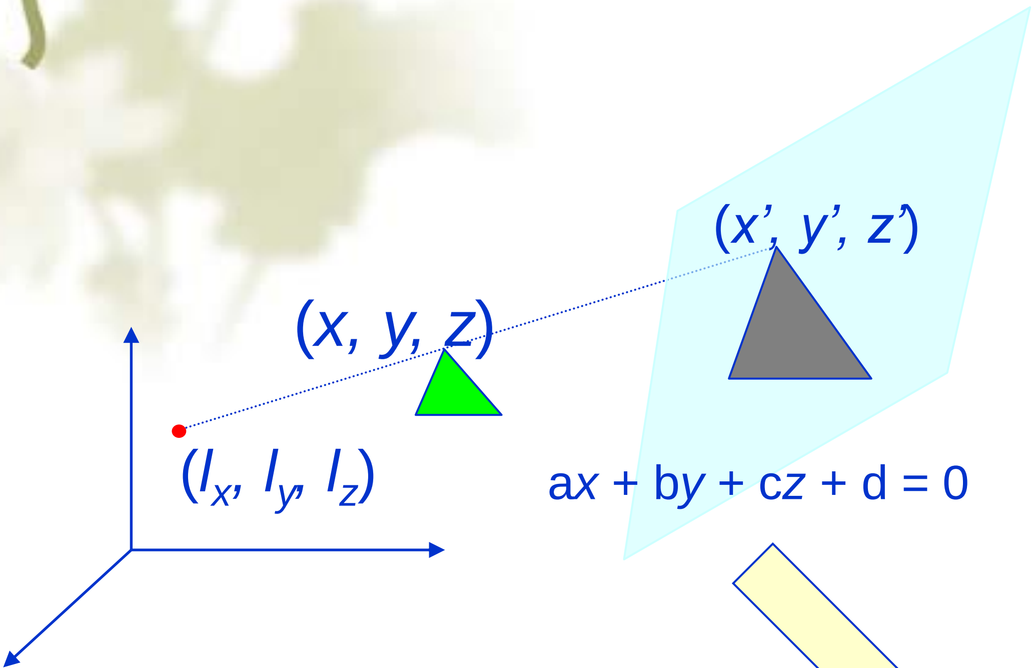
$$x' = \frac{x}{\frac{a}{-d}x + \frac{b}{-d}y + \frac{c}{-d}z}$$

$$y' = \ldots$$

$$z' = \ldots$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{a}{-d} & \frac{b}{-d} & \frac{c}{-d} & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

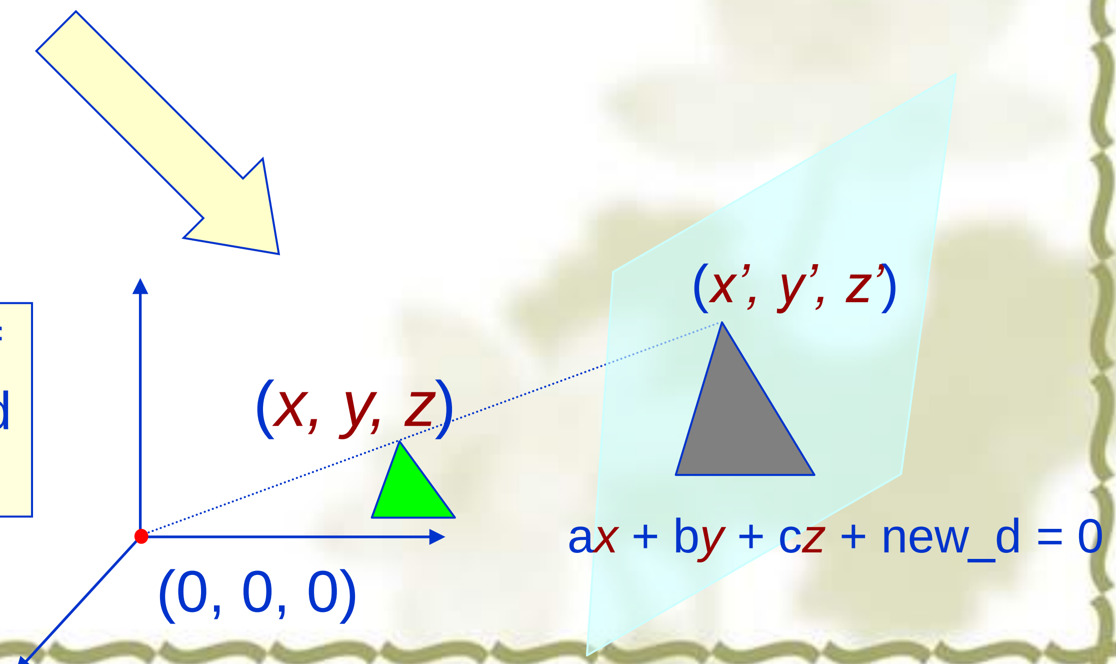
If the light source is at (*lx, ly, lz*) instead of at the origin, first translate the origin to the light source.



$x = x - l_x$
$y = y - l_y$
$z = z - l_z$

$a(x - l_x) + b(y - l_y) +$
$c(z - l_z) + new\_d = 0$

$new\_d = d + a\, l_x + b\, l_y + c\, l_z$



Note that the constant term of the plane equation is changed after the translation.

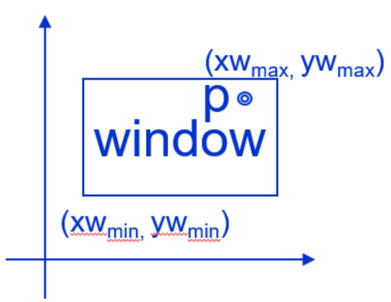
```
//Set up the matrix M such that it projects a vertex on the plane
//      ax + by + cz + new_d = 0        that new_d=d+a lx+b ly +c lz
//      The light source is at (lx, ly, lz).

void pointSourceShadow( double lx, double ly, double lz,
        double a, double b, double c, double new_d, GLfloat M[16]) {

    M[0] = 1;          M[4] = 0;          M[8] = 0;          M[12] = 0;
    M[1] = 0;          M[5] = 1;          M[9] = 0;          M[13] = 0;
    M[2] = 0;          M[6] = 0;          M[10] = 1;         M[14] = 0;
    M[3]= -a/d;  M[7]= -b/d;  M[11]= -c/d;  M[15] = 0;

}
```

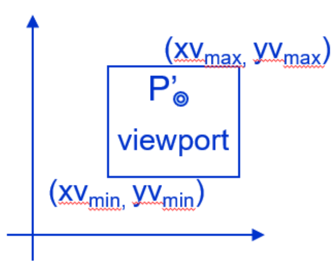
Questions

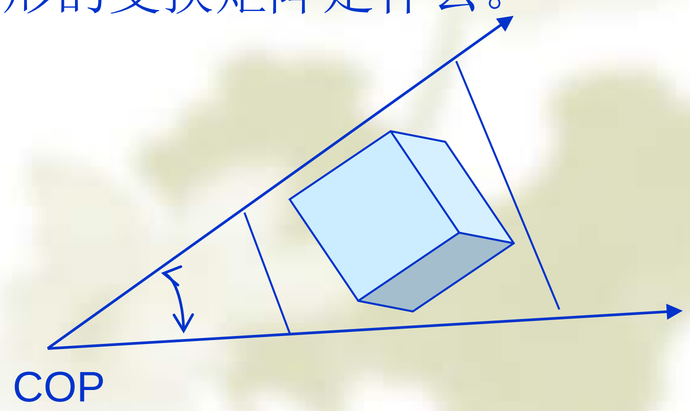
- In interactive computer graphics, when modeling, how to use two-dimensional devices such as a mouse to interface with three dimensional objects?
- In the future, volume holographic imaging will be applied to 3D display 全息成像

作业5

1. 飞机移动的位置由滚转角、俯视角和偏航角以及与物体的距离确定。根据这些参数给出一个观察矩阵？
2. 在进行视见变换时，眼睛空间的作用是什么？
3. 当我们从远处观察一个封闭房间的内部时，给出透视视见体的定义。
4. 我们有一个变换，是从场景2D窗口映射到视见窗，其中缩放系数为(Sx，Sy), 讨论当Sx≠Sy时，视见窗的无变形的变换矩阵是什么。



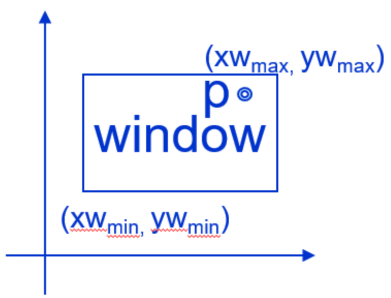


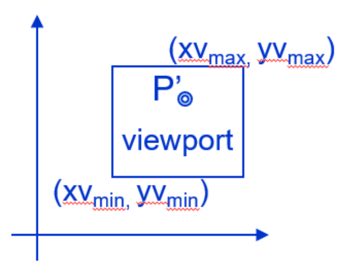


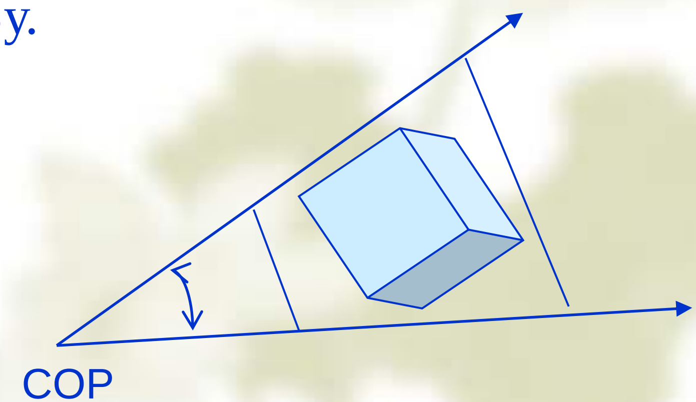
Perspective(fovy, aspect, zNear, zFar);

exercises

1. Consider an airplane whose position is specified by the roll, pitch, and yaw and by the distance from the object. Give a viewing matrix in terms of these parameters?
2. When viewing transformations, what is the eye space function?
3. When we view the interior of a closed room from a distance, give a define for view volume of perspective.
4. We have the transformation matrix from scene 2D window into viewport, here scaling by (S_x, S_y). Discuss transformation matrix of distortionless in the viewport when $S_x \neq S_y$.







Perspective(fovy, aspect, zNear, zFar);