5. Viewing

Lecture Overview

- Classical Viewing (ANG Ch. 5.1-5.9)
  - Skipping 5.10

Classical Viewing

Objectives

- Introduce the classical views
- Compare and contrast image formation by computer with how images have been formed by architects, artists, and engineers
- Learn the benefits and drawbacks of each type of view

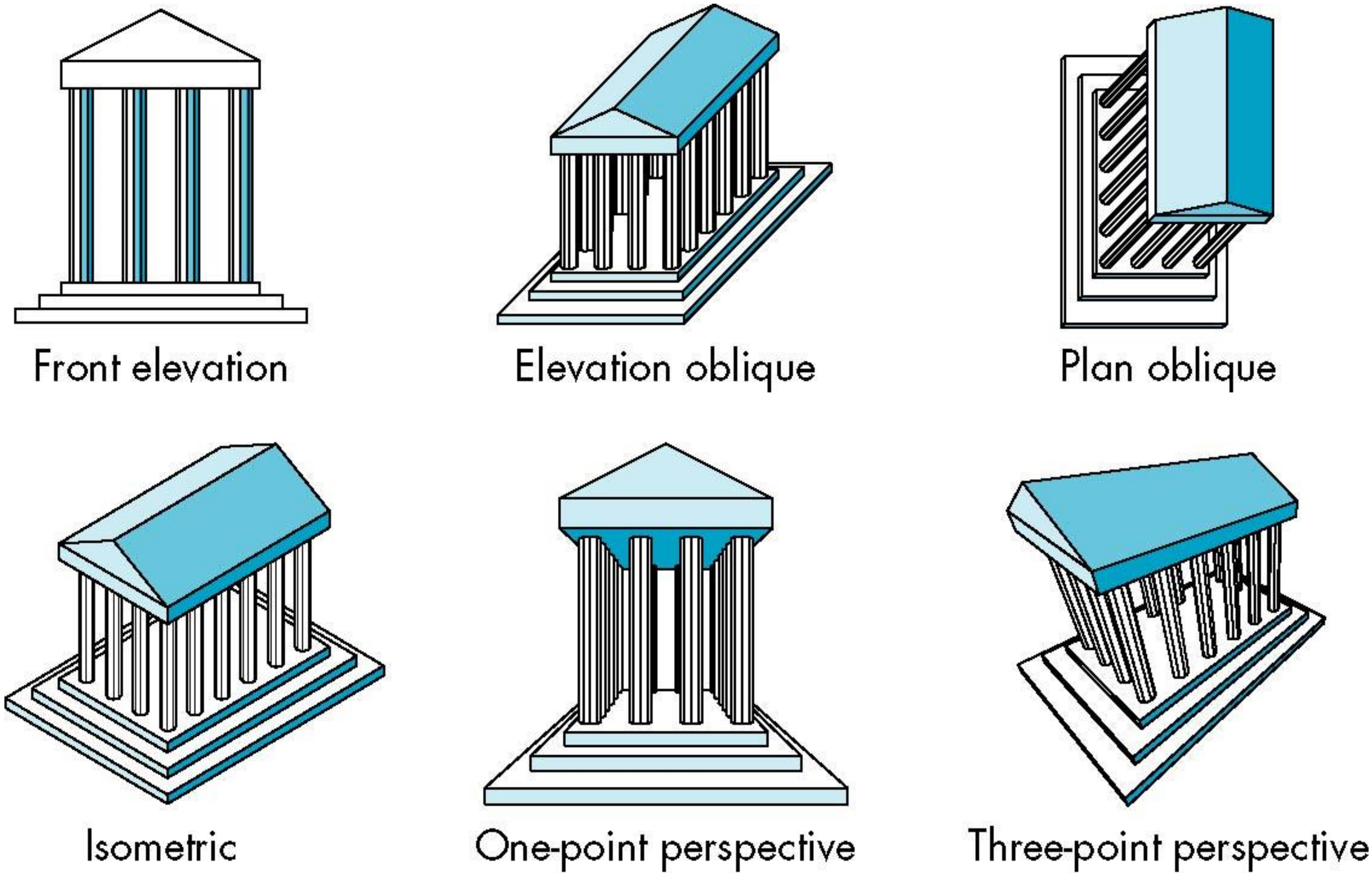
Classical Viewing

- Viewing requires three basic elements
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface
- Classical views are based on the relationship among these elements
  - The viewer picks up the object and orients it how she would like to see it
- Each object is assumed to constructed from flat *principal faces*
  - Buildings, polyhedra, manufactured objects

Planar Geometric Projections

- Standard projections project onto a plane
- Projectors are lines that either
  - converge at a center of projection
  - are parallel
- Such projections preserve lines
  - but not necessarily angles
- Nonplanar projections are needed for applications such as map construction

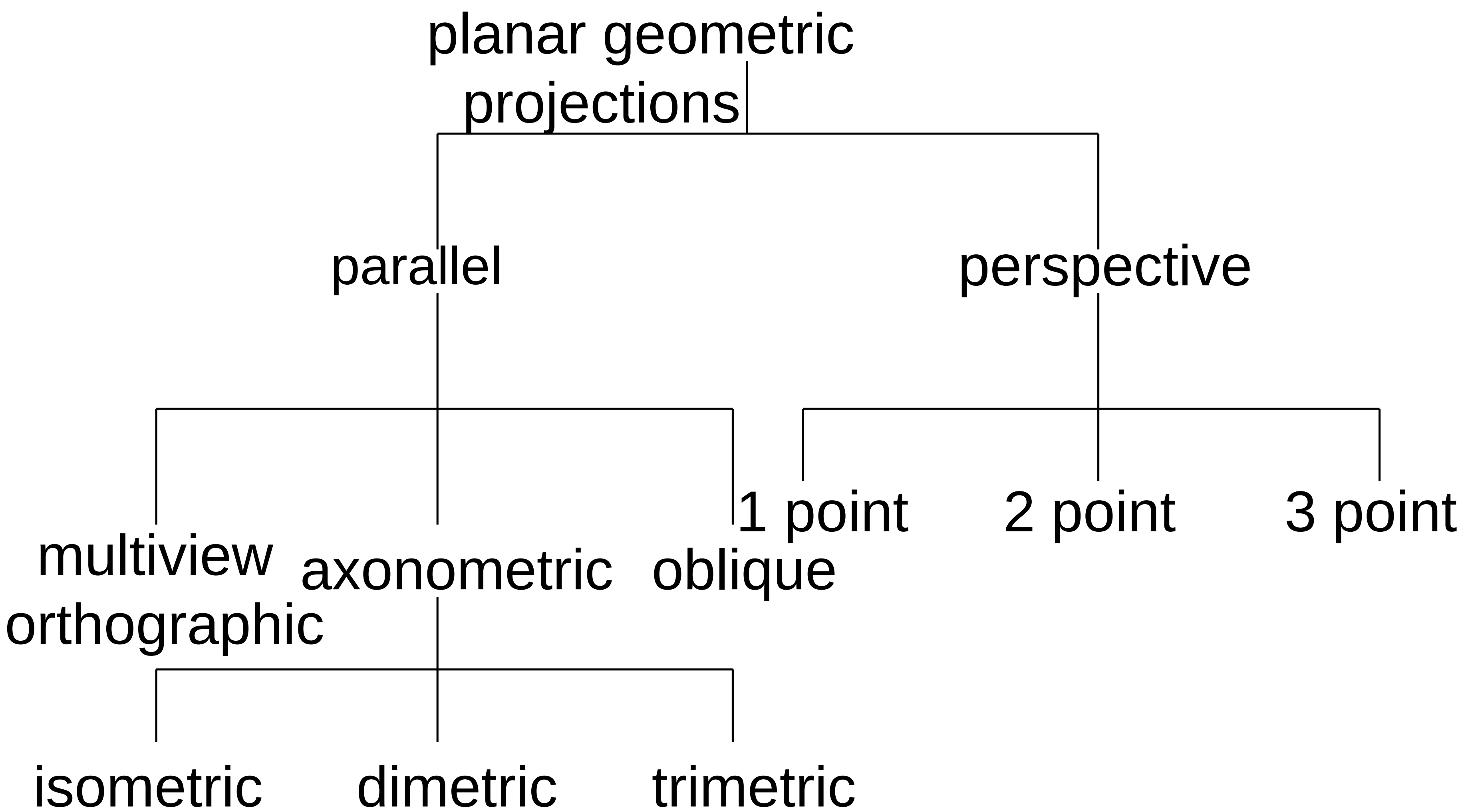
Classical Projections



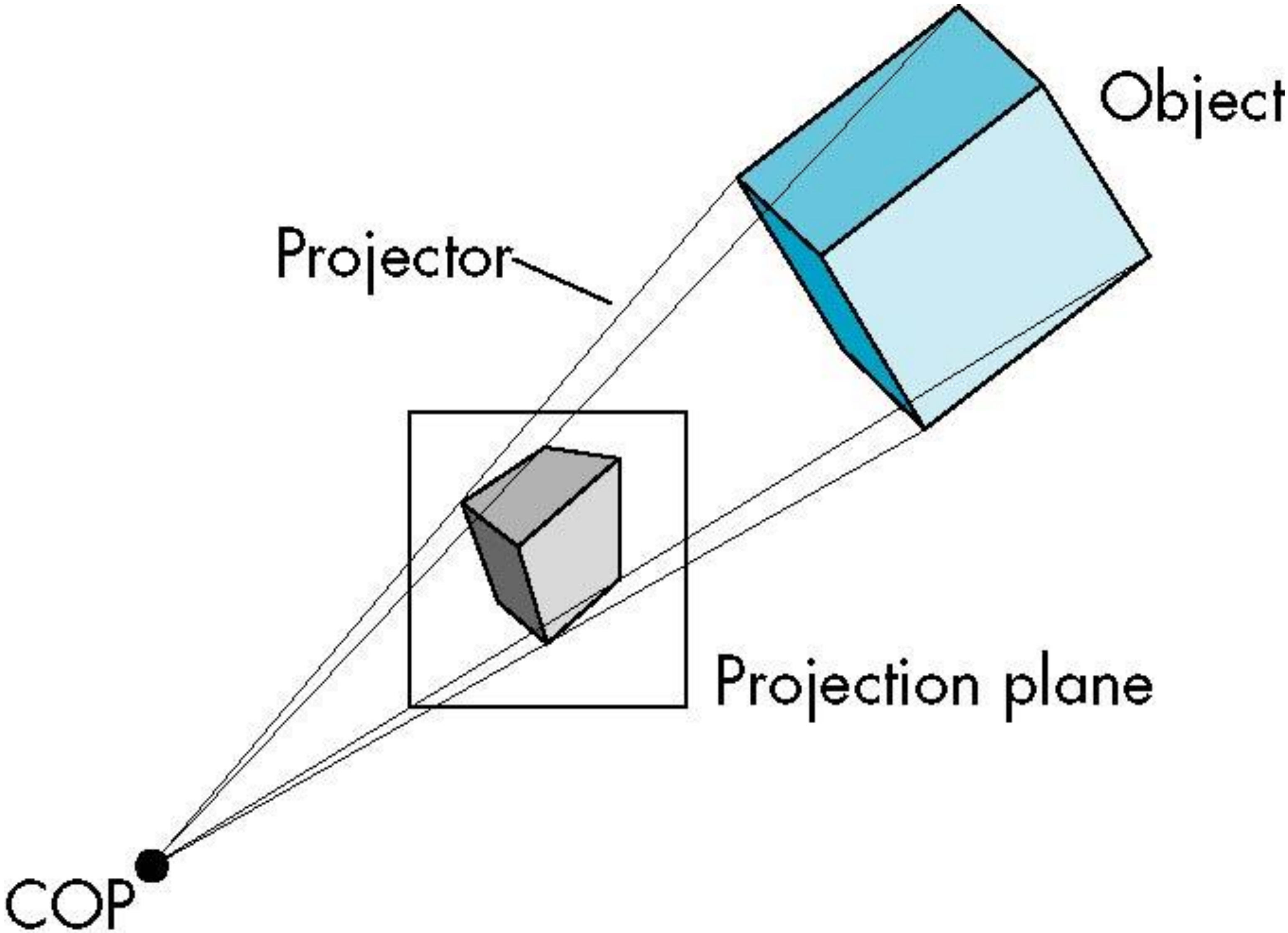
Perspective vs Parallel

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

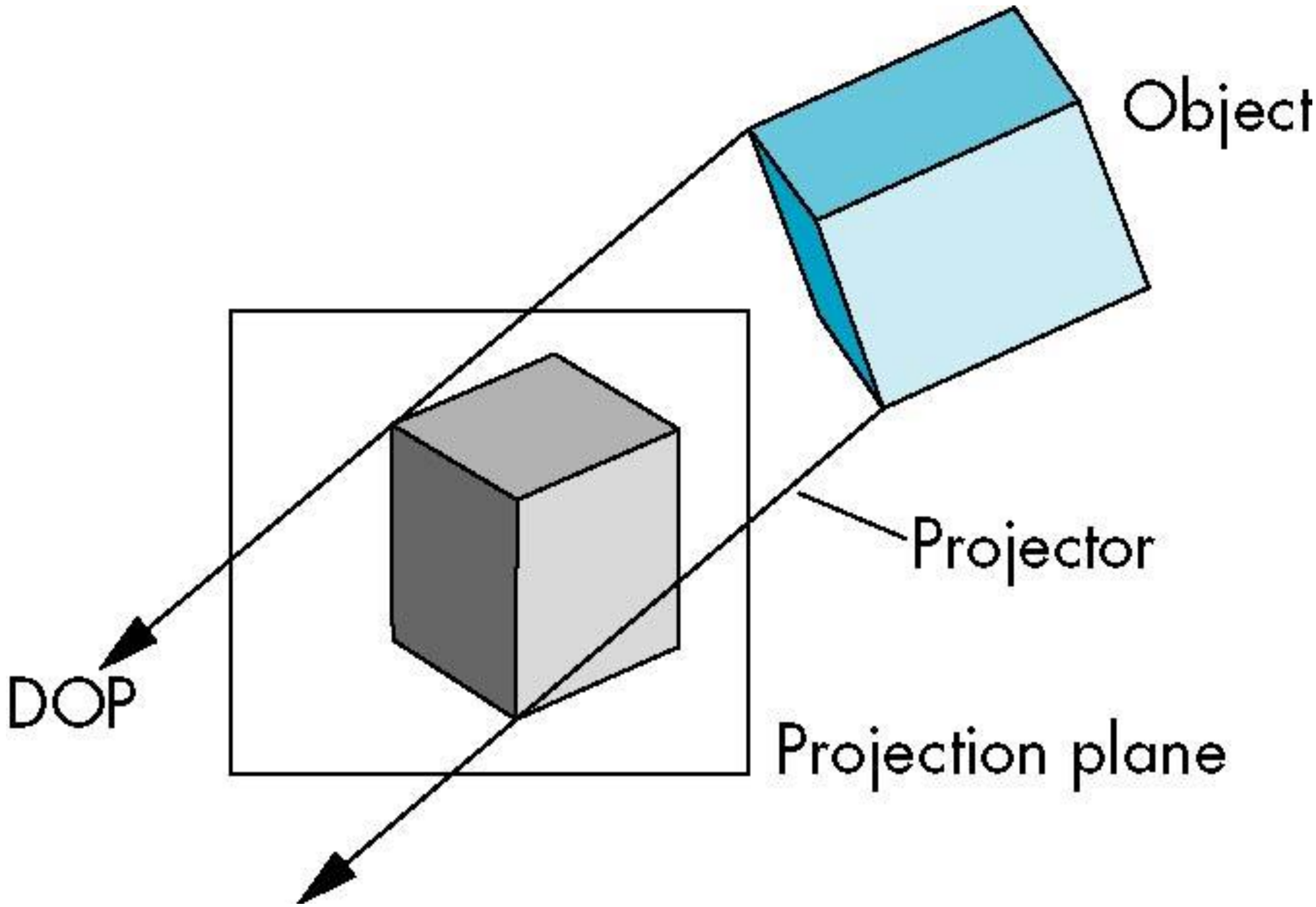
Taxonomy of Planar Geometric Projections



Perspective Projection

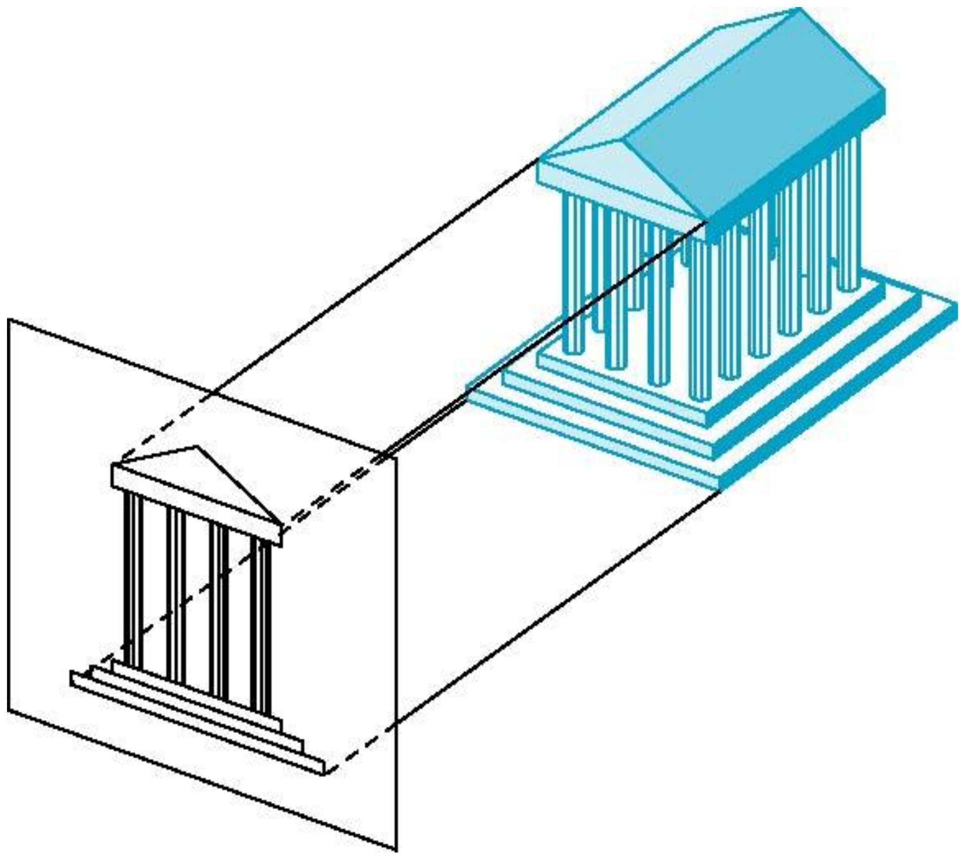


Parallel Projection



Orthographic Projection

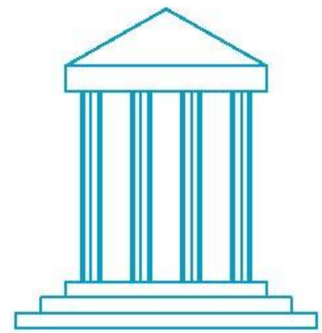
Projectors are orthogonal to projection surface



Multiview Orthographic Projection

- Projection plane parallel to principal face
- Usually form front, top, side views

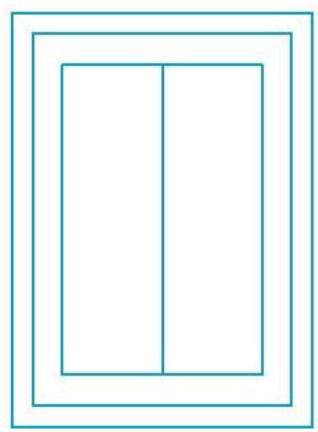
isometric (not multiview orthographic view)

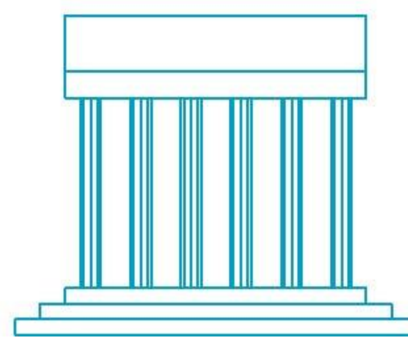


front

in CAD and architecture, we often display three multiviews plus isometric

top





side

Advantages and Disadvantages

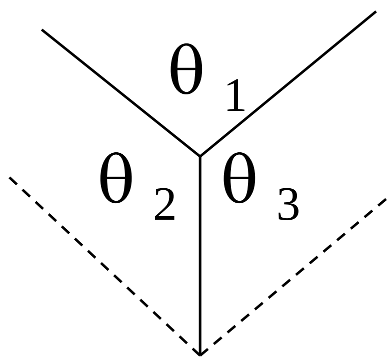
- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
  - Often we add the isometric

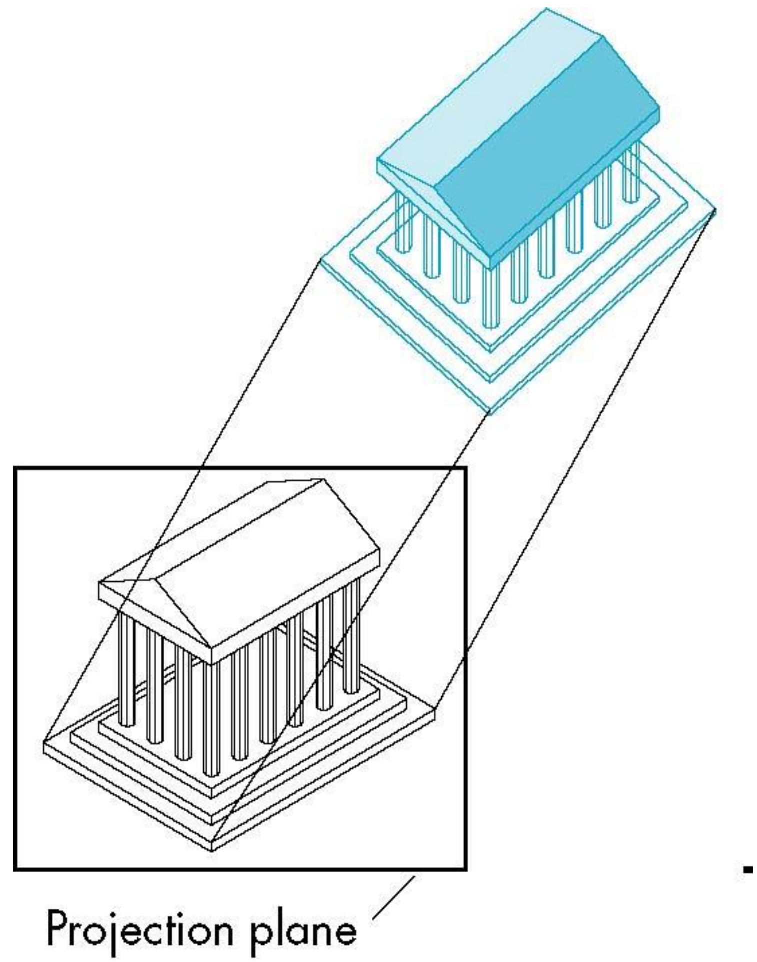
Axonometric Projections

Allow projection plane to move relative to object

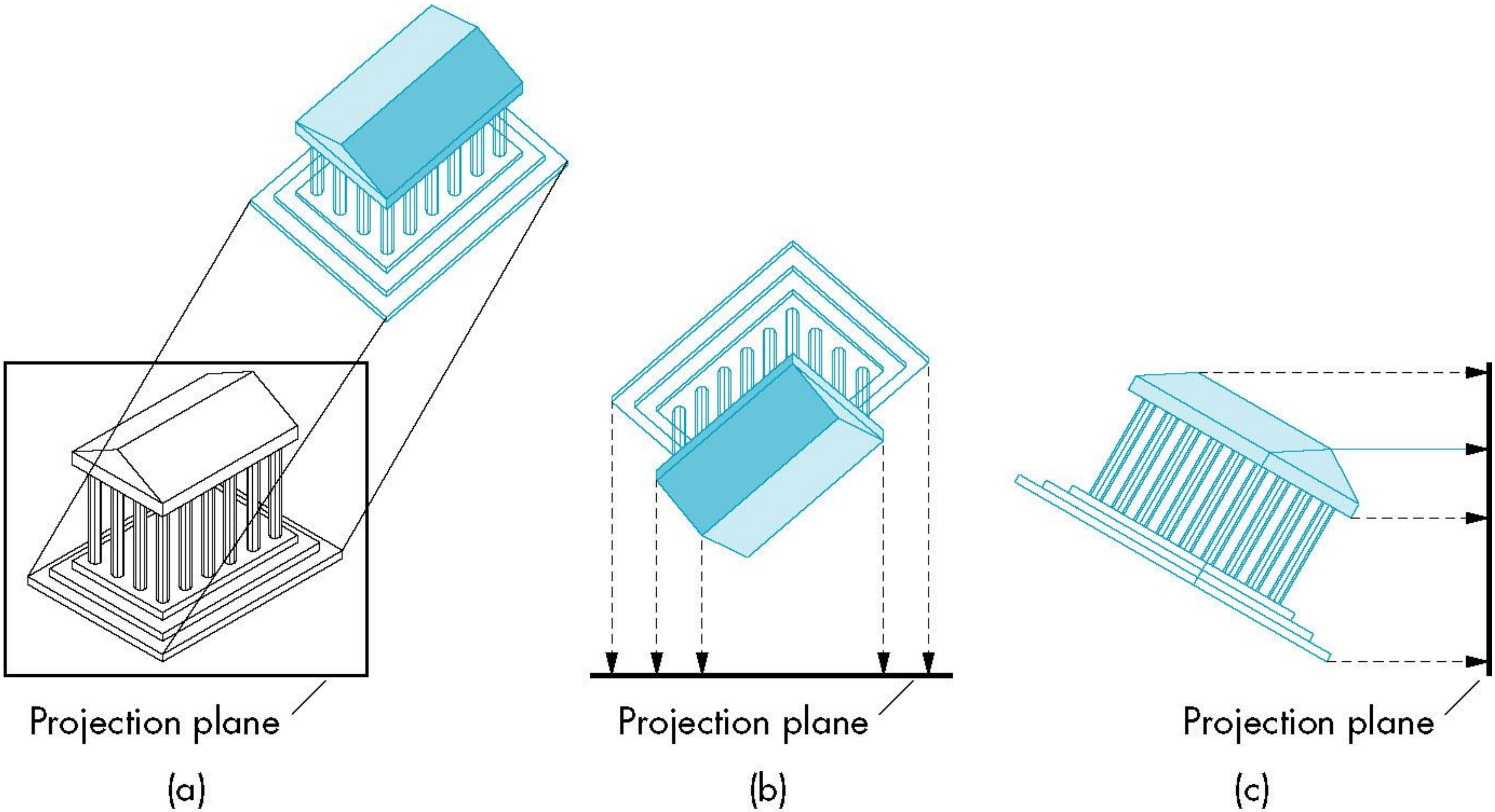
classify by how many angles of a corner of a projected cube are the same

none: trimetric
two: dimetric
three: isometric





Axonometric Projections

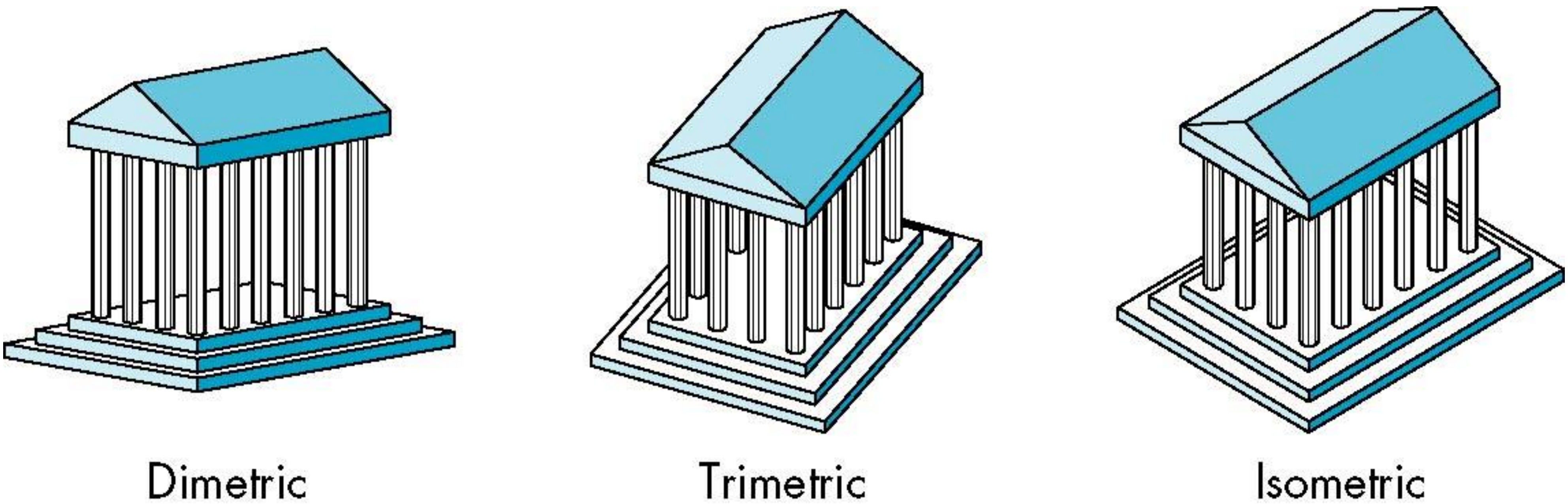


Construction of trimetric-view projection

Top view

Side view

Types of Axonometric Projections

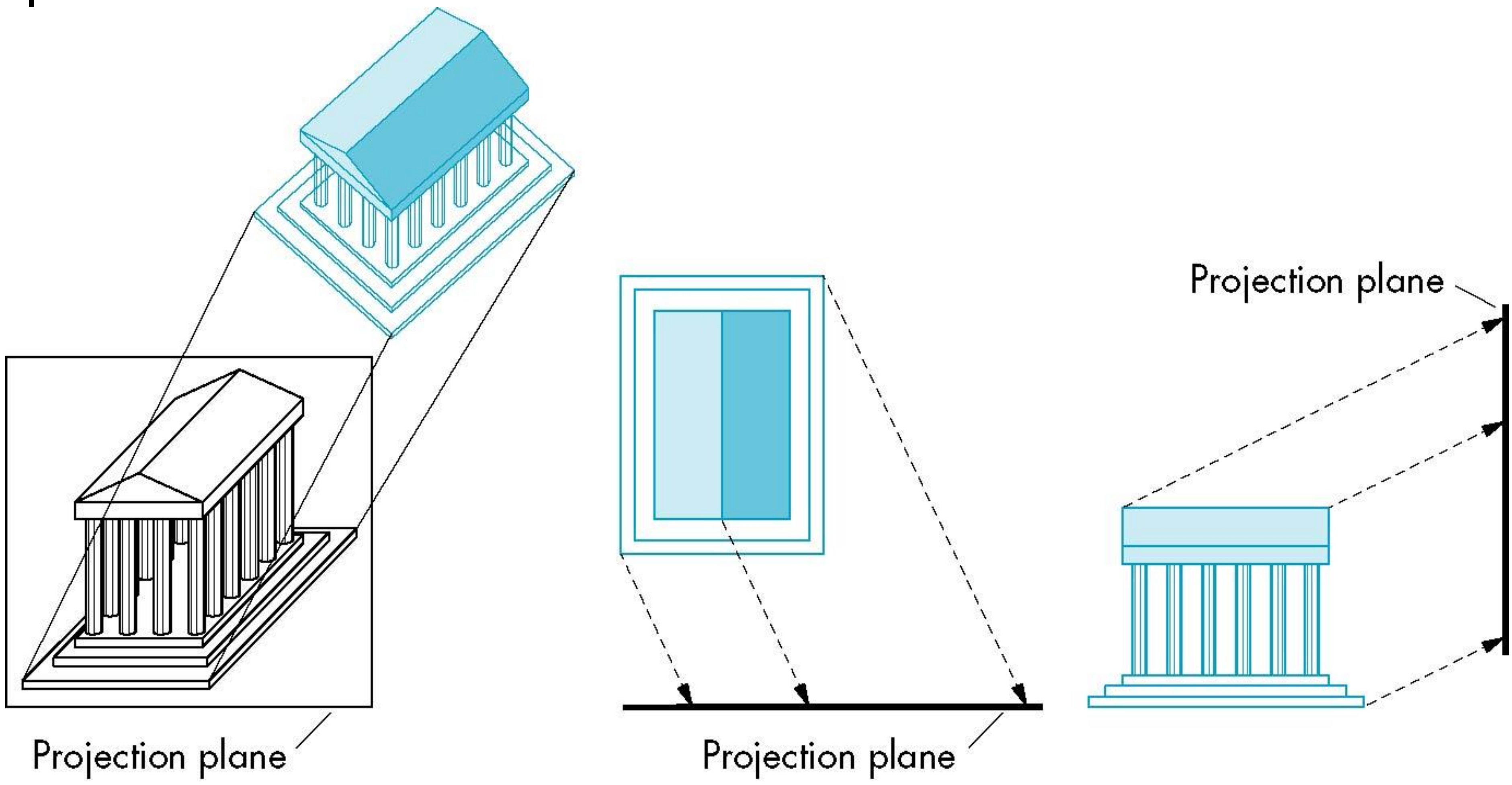


Advantages and Disadvantages

- Lines are scaled (*foreshortened*) but can find scaling factors
- Lines preserved but angles are not
  - Projection of a circle in a plane not parallel to the projection plane is an ellipse
- Can see three principal faces of a box-like object
- Some optical illusions possible
  - Parallel lines appear to diverge
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications

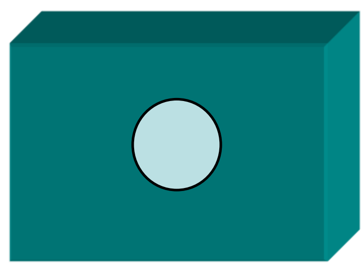
Oblique Projection

- Arbitrary relationship between projectors and projection plane



Advantages and Disadvantages

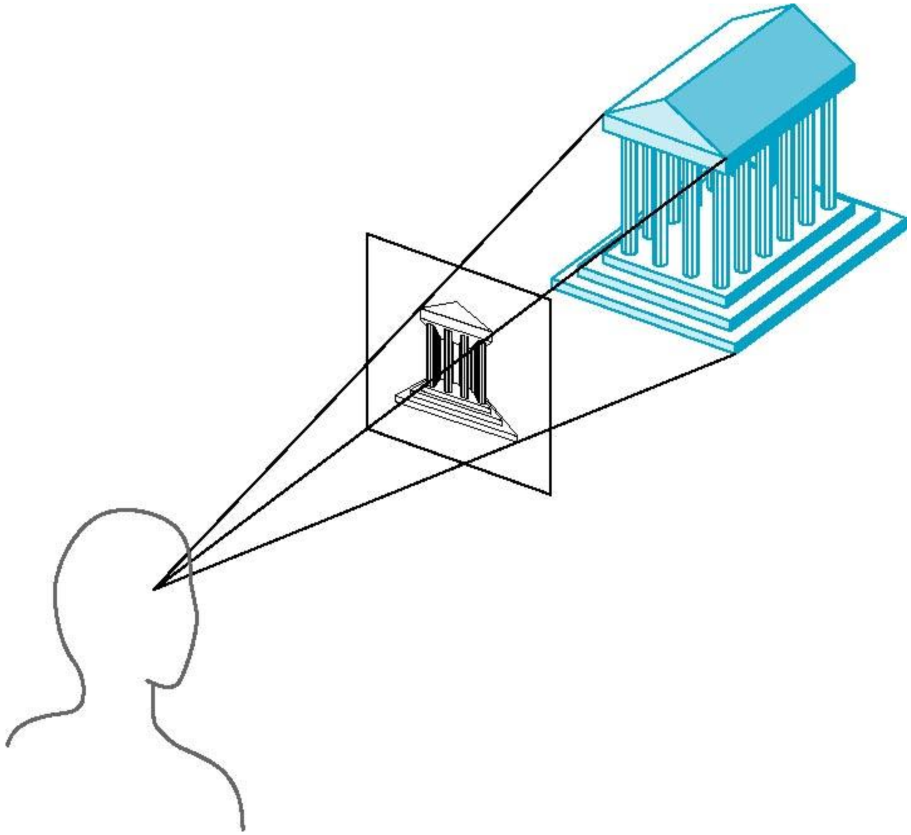
- Can pick the angles to emphasize a particular face
  - Architecture: plan oblique, elevation oblique
- Angles in faces parallel to projection plane are preserved while we can still see “around” side



- In physical world, cannot create with simple camera; possible with bellows camera or special lens (architectural)

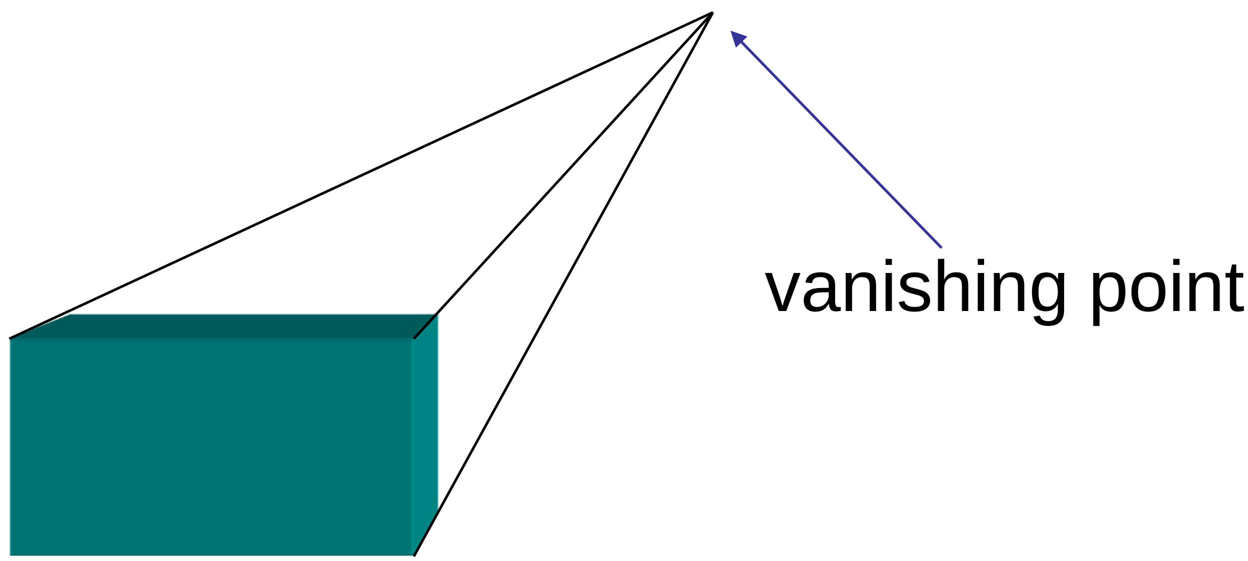
Perspective Projection

Projectors coverage at center of projection



Vanishing Points

- Parallel lines (not parallel to the projection plan) on the object converge at a single point in the projection (the *vanishing point*)
- Drawing simple perspectives by hand uses these vanishing point(s)



Three-Point Perspective

- No principal face parallel to projection plane
- Three vanishing points for cube

Two-Point Perspective

- On principal direction parallel to projection plane
- Two vanishing points for cube

One-Point Perspective

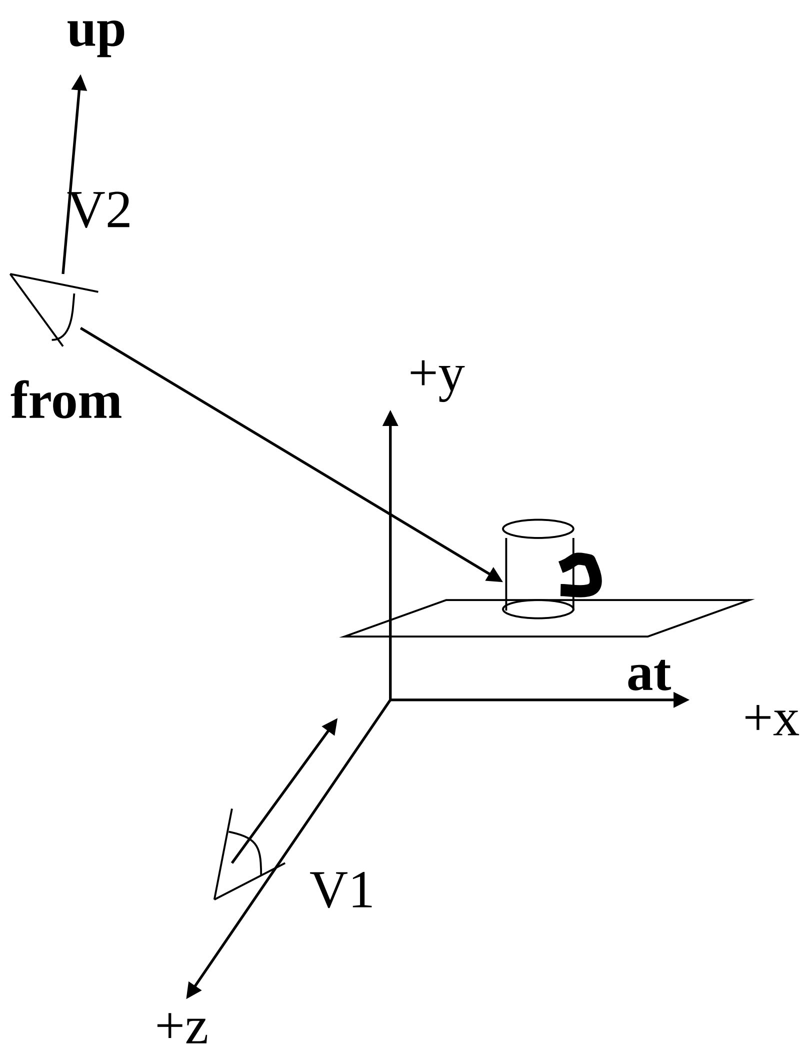
- One principal face parallel to projection plane
- One vanishing point for cube

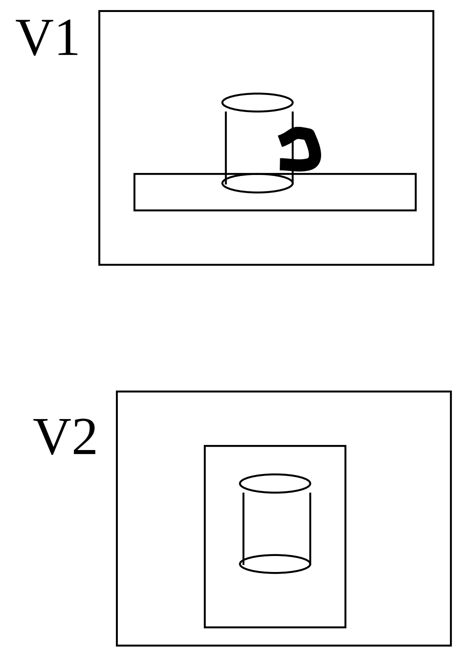
Advantages and Disadvantages

- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

Computer Viewing

Viewing Transformation





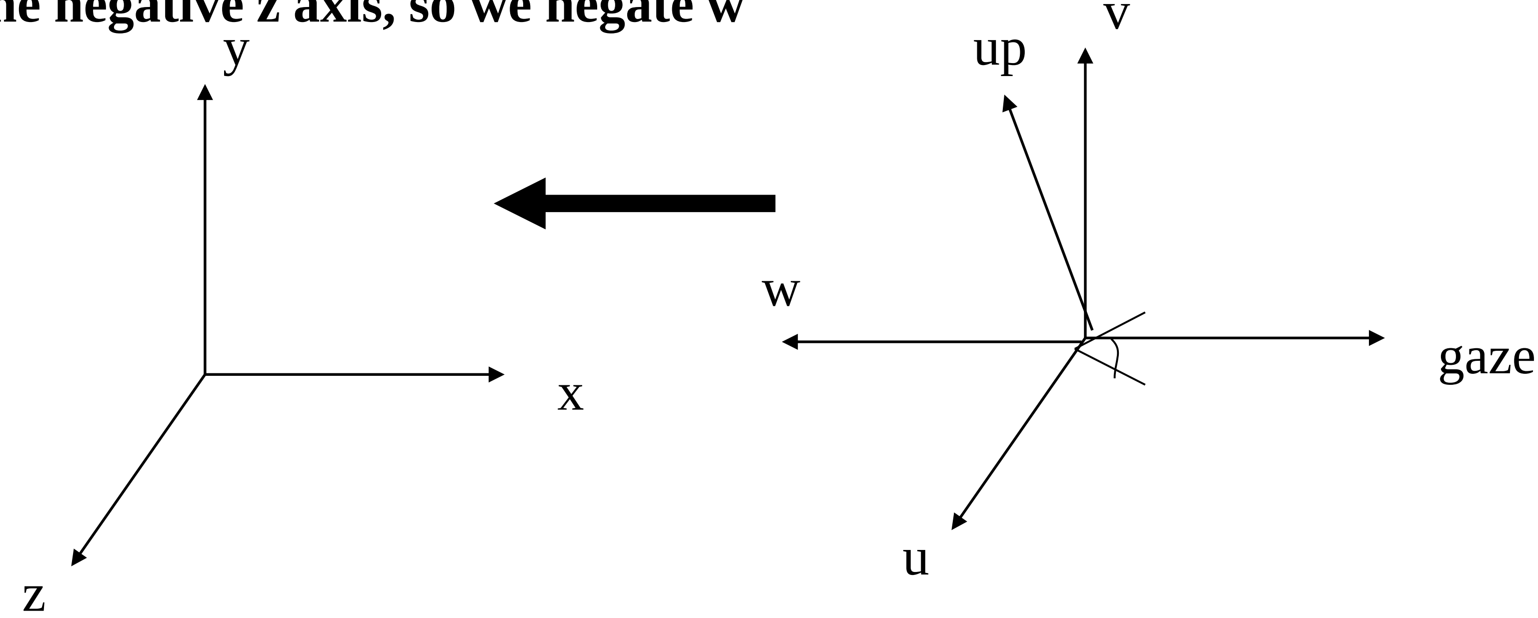
How do we get from V2 to V1?

Viewing Transformation

Fill the rows of the rotation matrix: R

$$w = -\frac{at - from}{\|at - from\|} \qquad u = \frac{up \times w}{\|up \times w\|} \qquad v = w \times u$$

Note: This will orient the eye looking down the z axis!!! We want it looking down the negative z axis, so we negate w



Graphics Pipeline

Model Space

Model Transformations

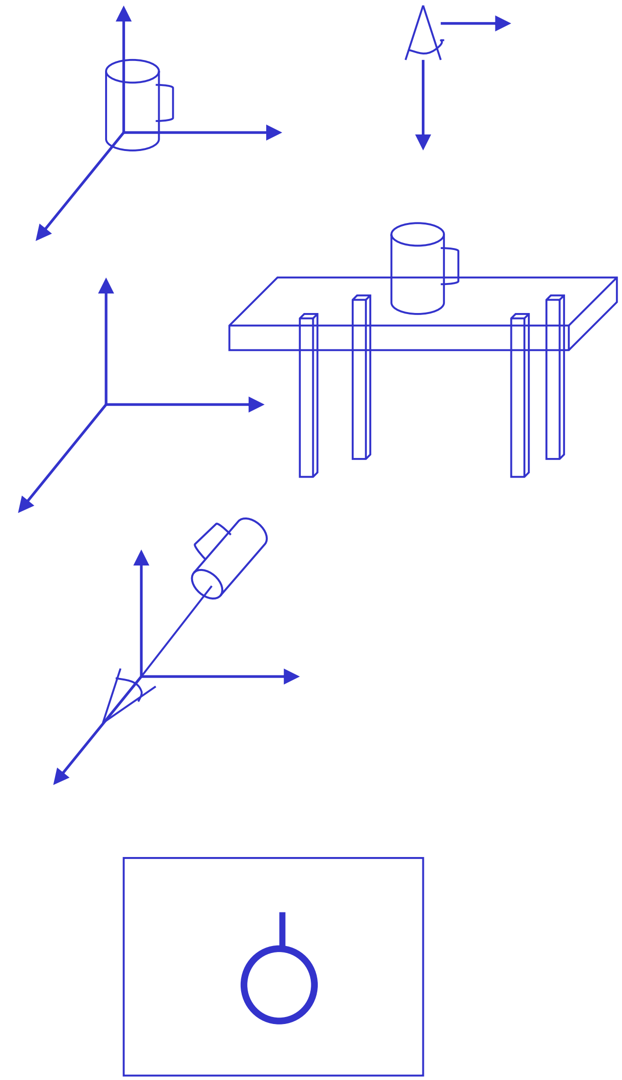
World Space

Viewing Transformation

Eye/Camera Space

Projection & Window Transformation

Screen Space



Graphics Pipeline

(ortho projection only)

Model Space

M_{model} **Model Transformations**

World Space

$M_{viewing}$ **Viewing Transformation**

Eye/Camera Space

$M_{orthographic}$ **Projection & Window Transformation**

(ortho projection includes the windowing transform)

Screen Space

$$P' = M_{ortho} M_{viewing} M_{model} P$$

Viewing

- Orthographic views
- Perspective views

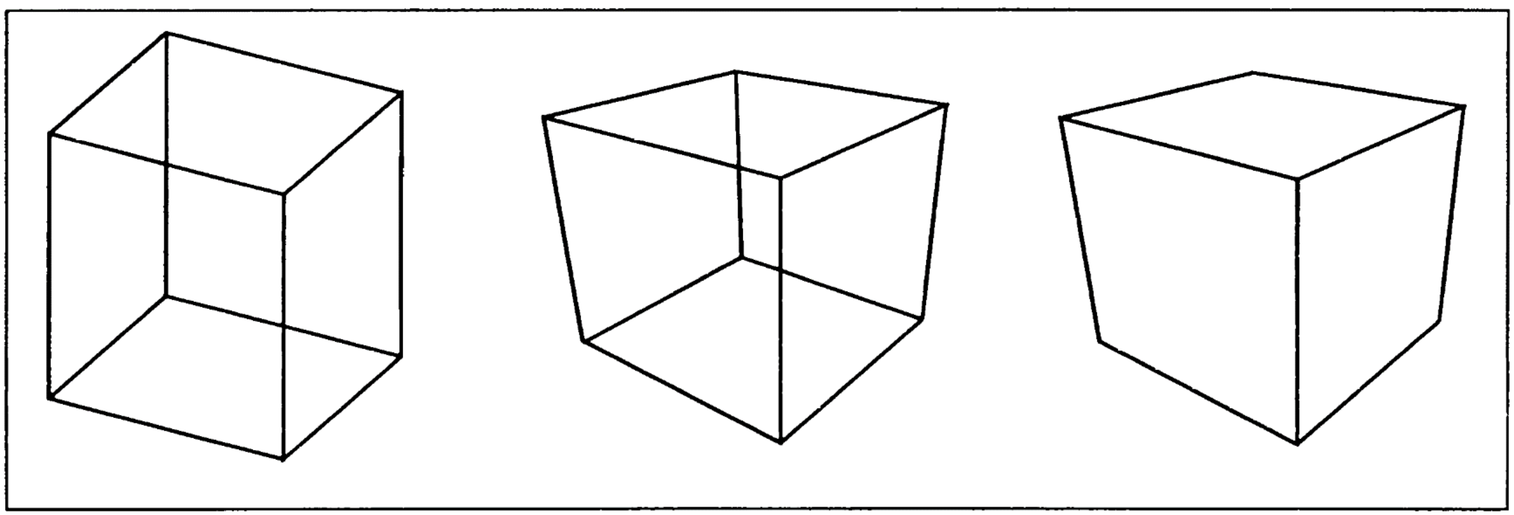
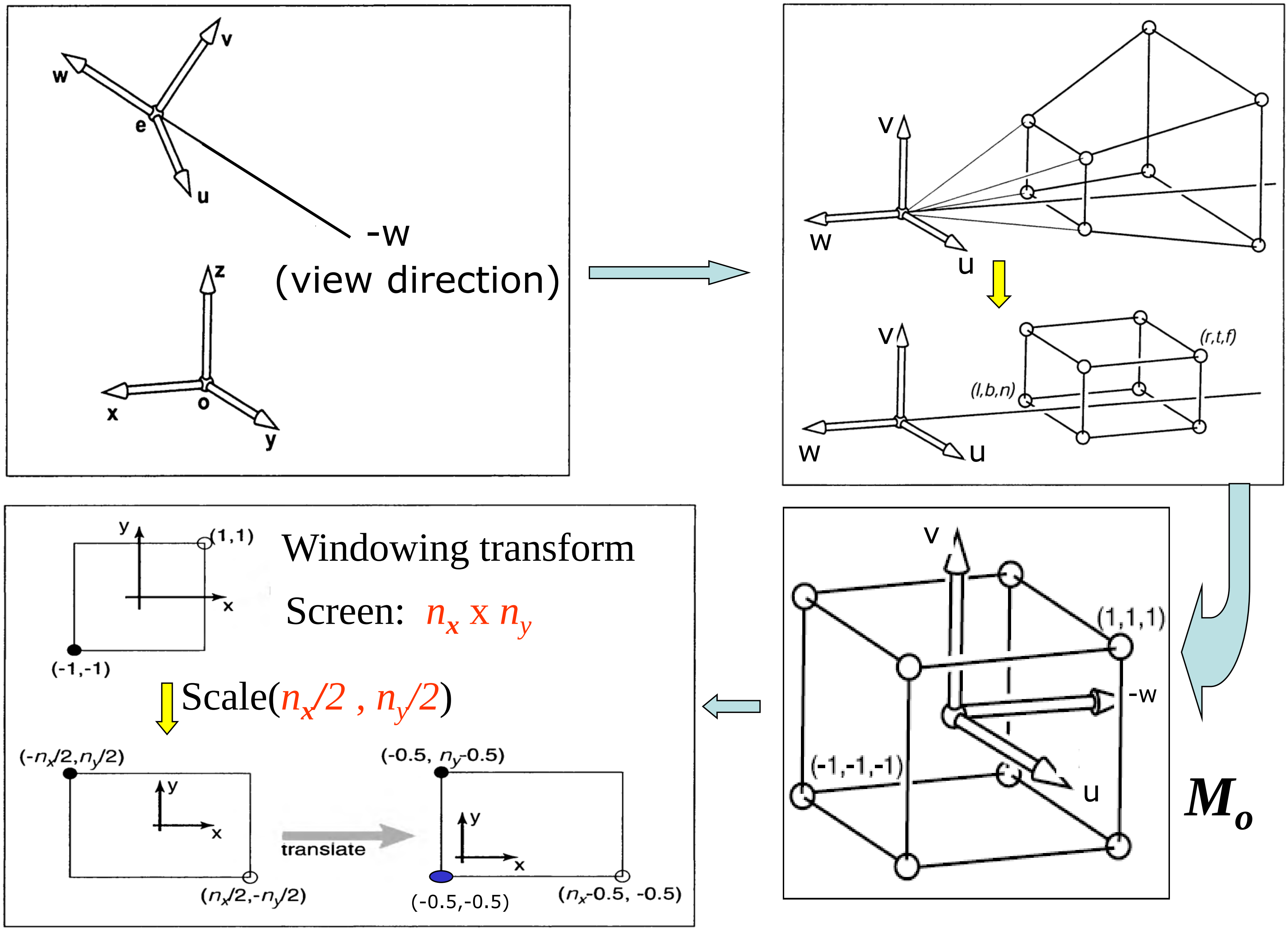


Figure 7.1. Left: orthographic projection. Middle: perspective projection. Right: perspective projection with hidden lines removed.

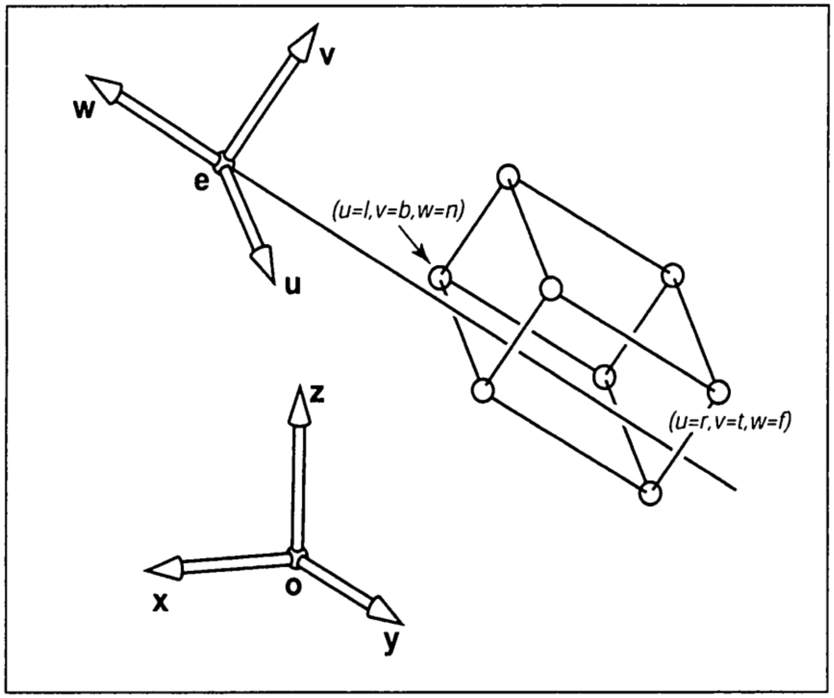
Perspective Viewing Transformation

M_v M_p

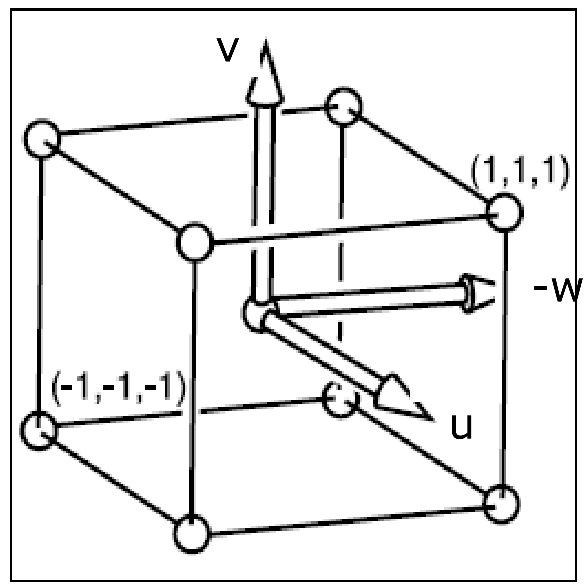


Orthographic Viewing Transformation

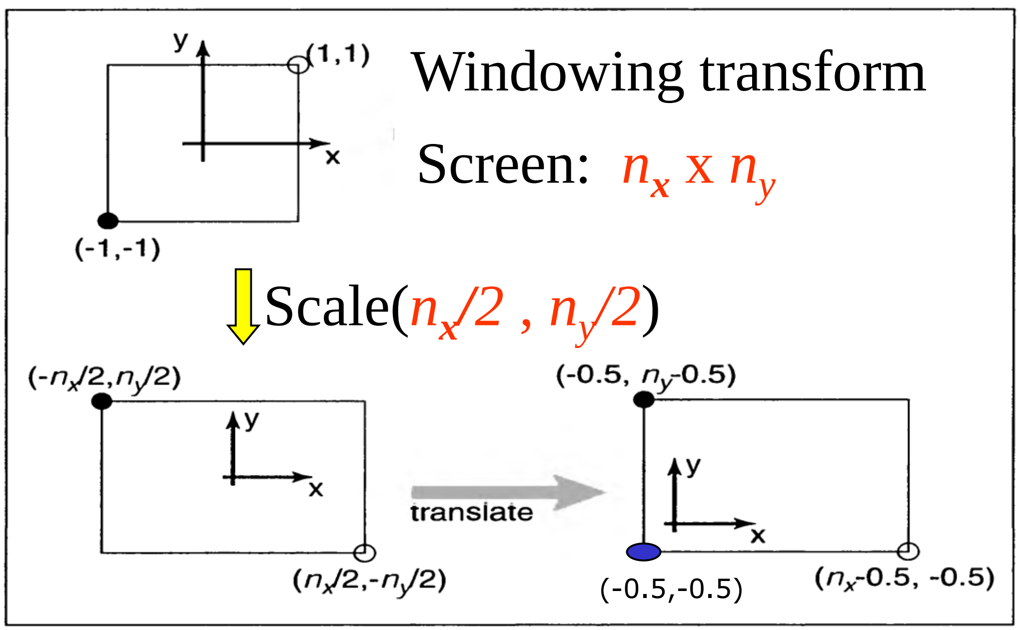
M_v



$$M = M_o M_v$$



M_o



Objectives

- Introduce the mathematics of projection
- Introduce OpenGL viewing functions
- Look at alternate viewing APIs

Computer Viewing

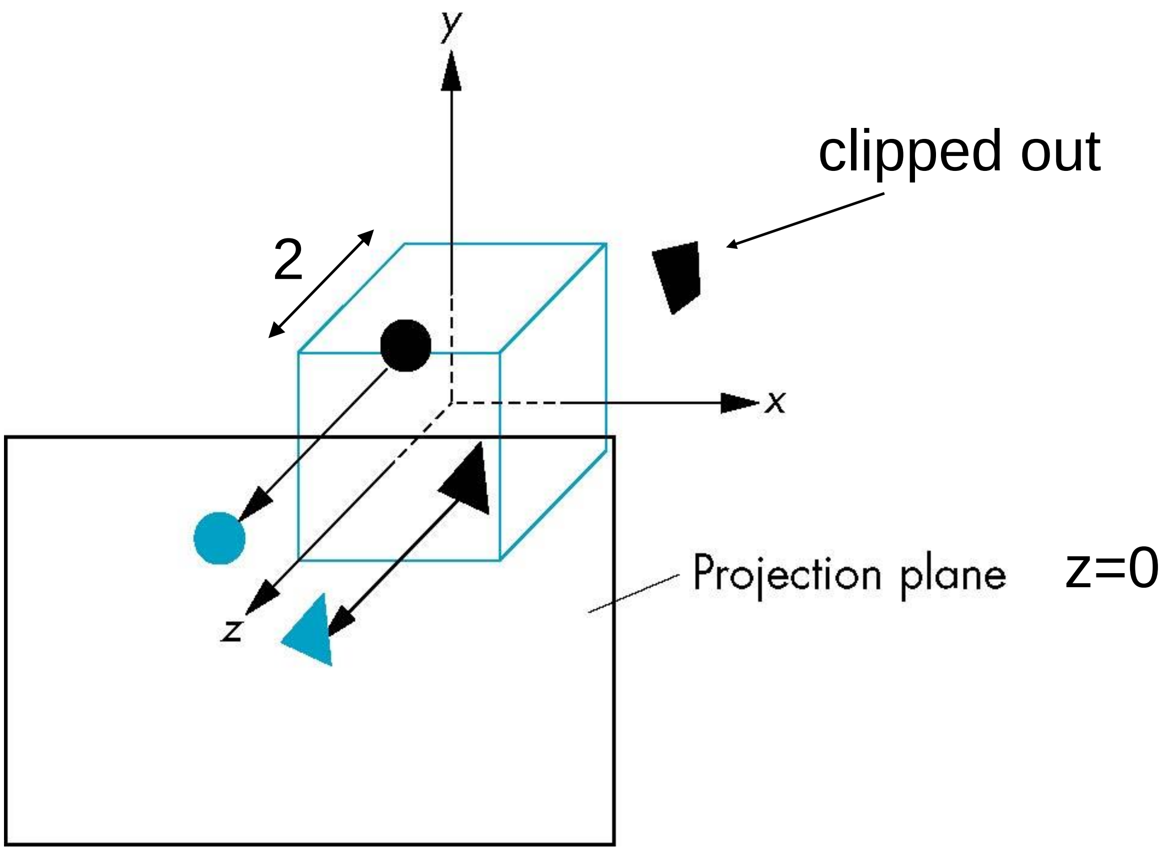
- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume

The OpenGL Camera

- In OpenGL, initially the object and camera frames are the same
  - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a *default view volume* that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity

Default Projection

Default projection is orthogonal



Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation (`glTranslatef(0.0,0.0,-d);`)

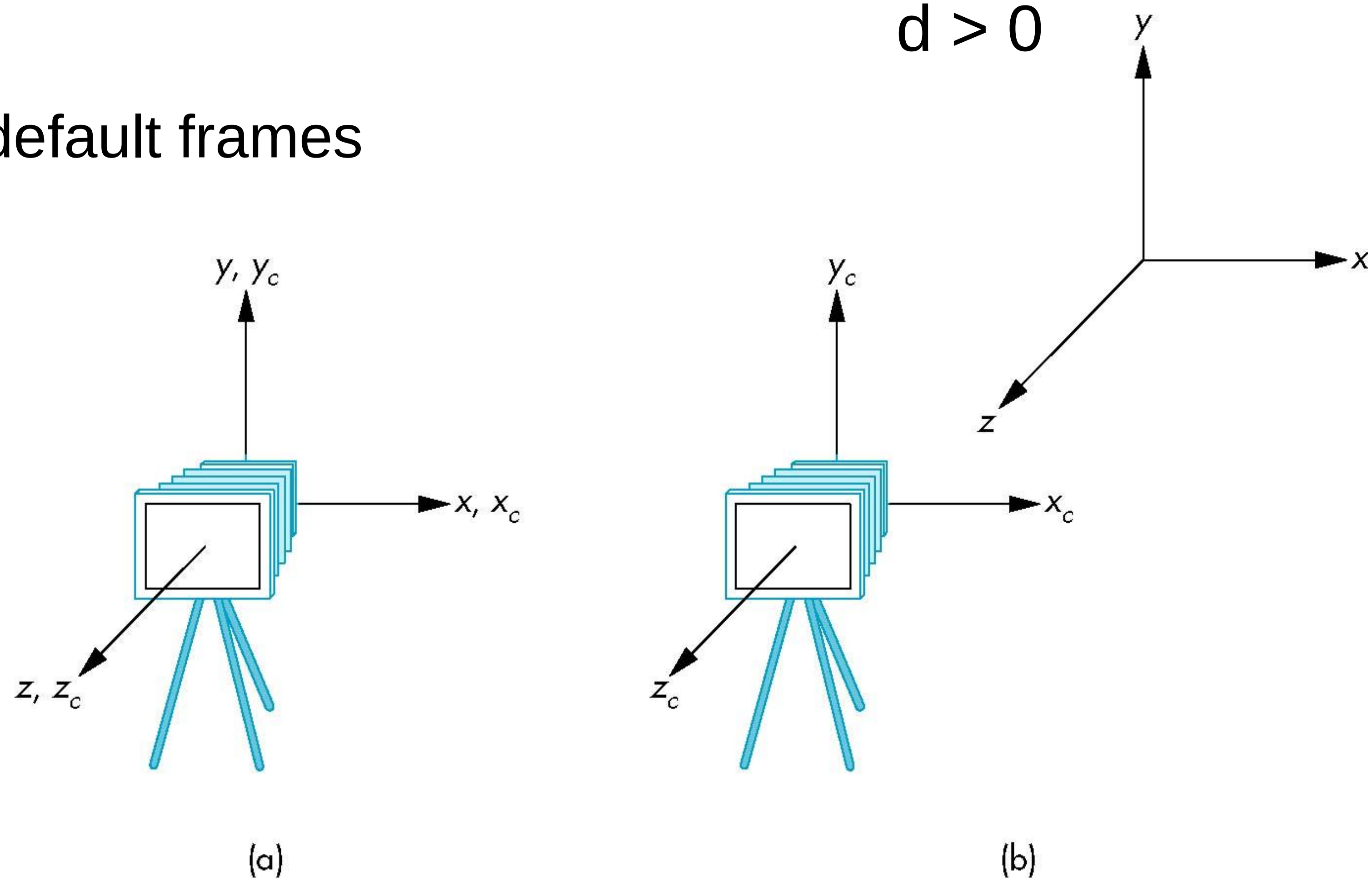
  $-d > 0$

Moving Camera back from Origin

frames after translation by –d

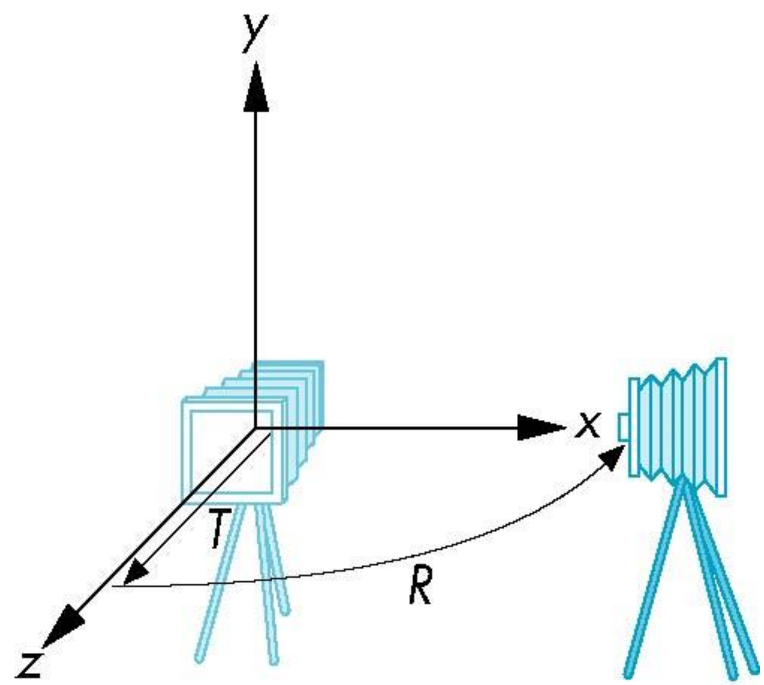
d > 0

default frames



Moving the Camera

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



OpenGL code

- Remember that last transformation specified is first to be applied

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

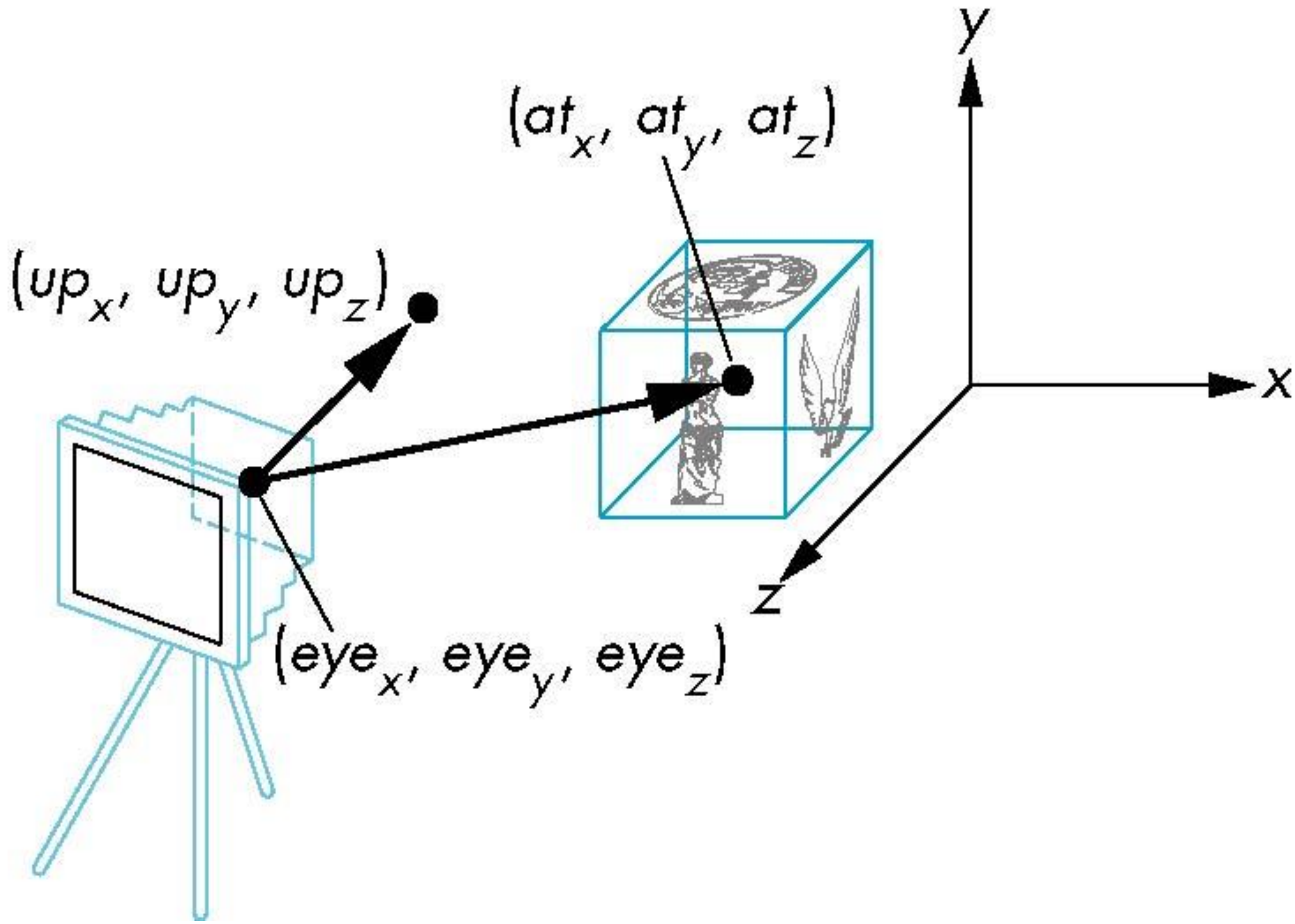
The LookAt Function

- The GLU library contains the function gluLookAt to form the required modelview matrix through a simple interface
- Note the need for setting an up direction
- Still need to initialize
  - Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```
glMatrixMode(GL_MODELVIEW):
glLoadIdentity();
gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0., 1.0. 0.0);
```

gluLookAt

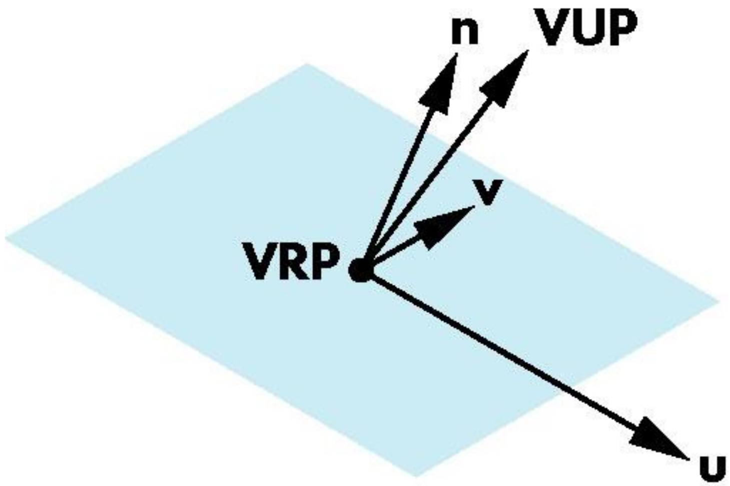
`gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)`



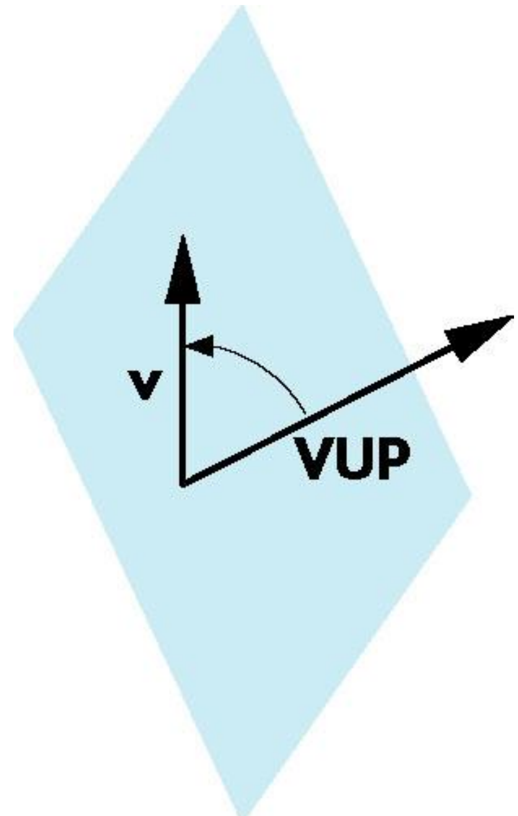
Other Viewing APIs

- 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

View reference point, view plane normal, view up (PHIGS, GKS-3D)

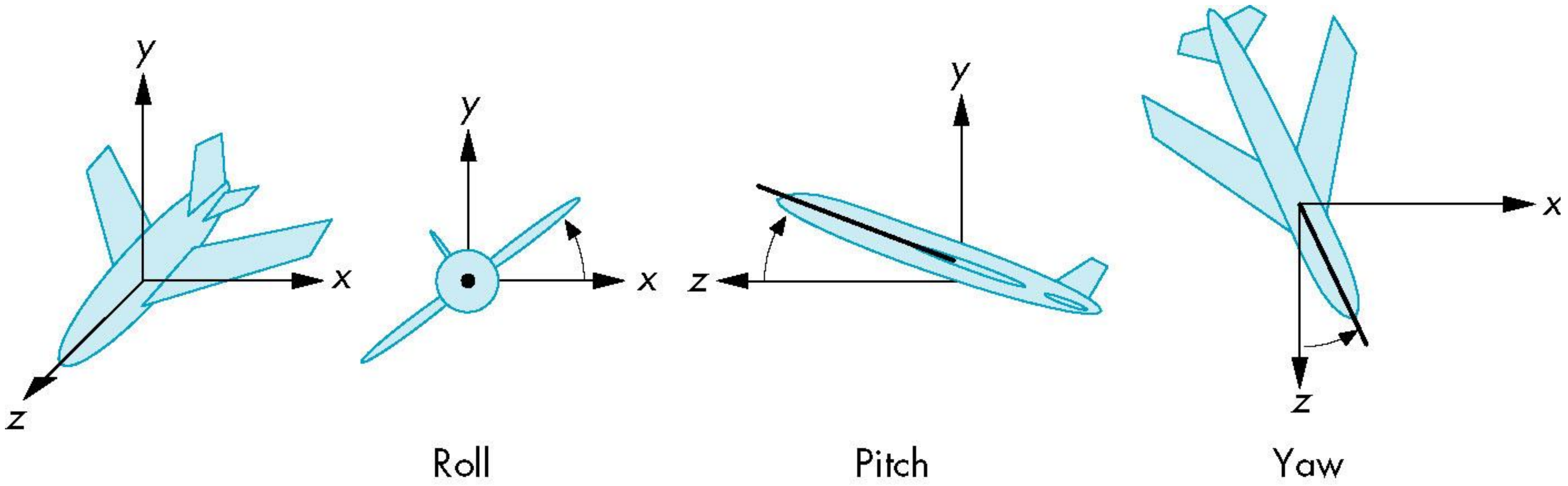


Camera frame

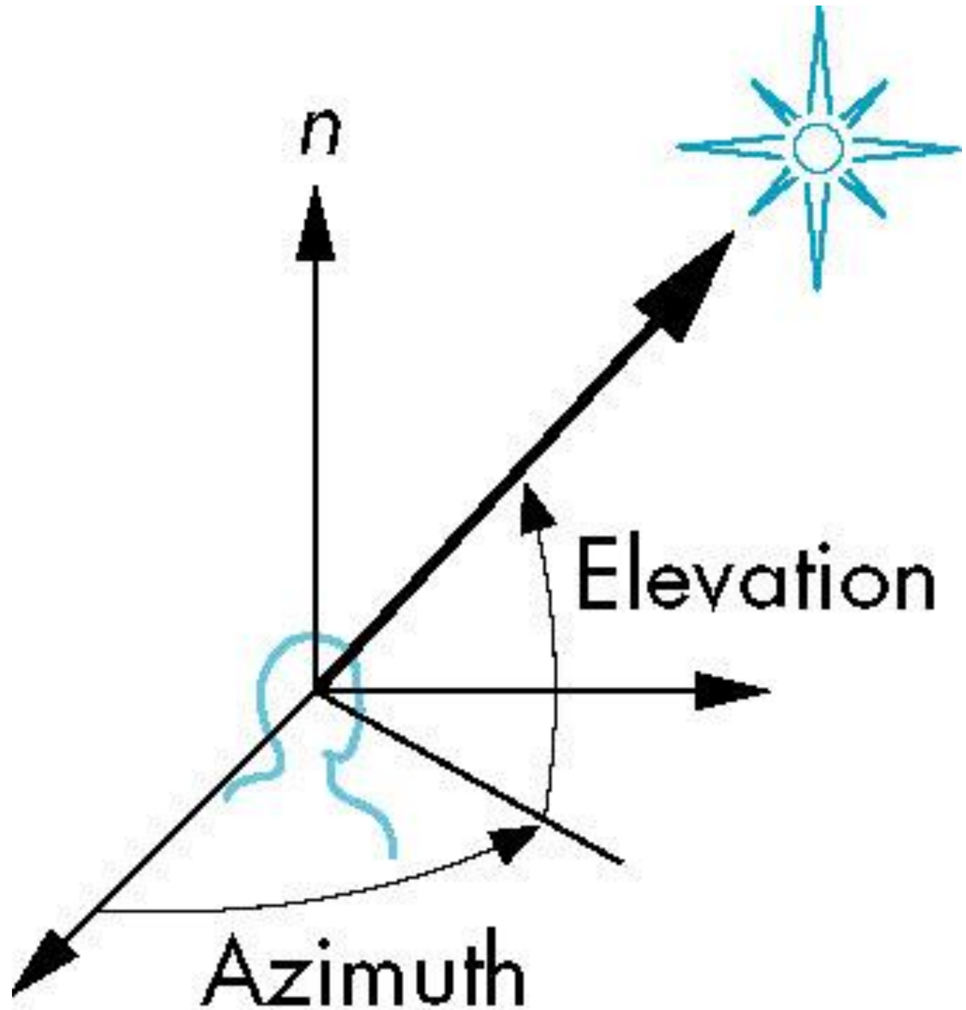


Determination of the view-up vector

Yaw, Pitch, and Roll



Elevation, Azimuth, and Twist



Projections 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

Homogeneous Coordinate Representation

default orthographic projection

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

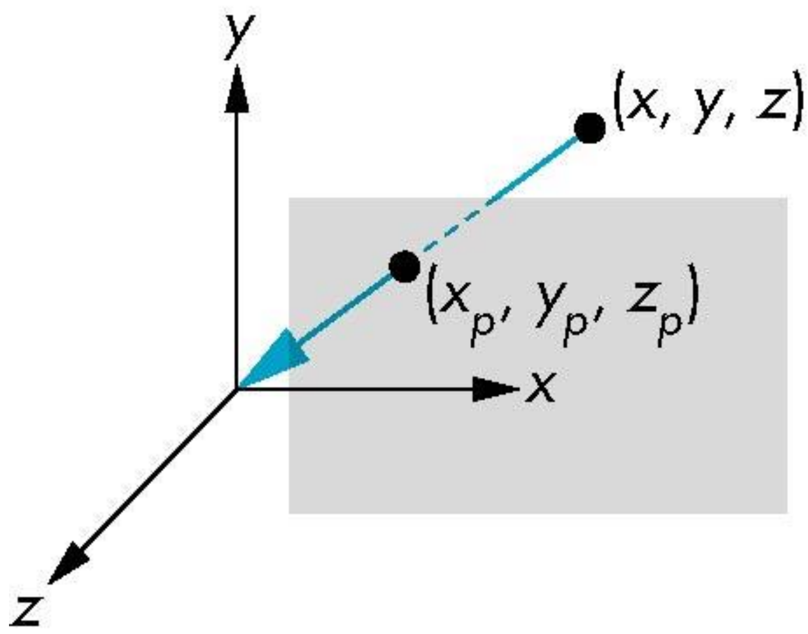
$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

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

In practice, we can let $\mathbf{M} = \mathbf{I}$ and set the z term to zero later

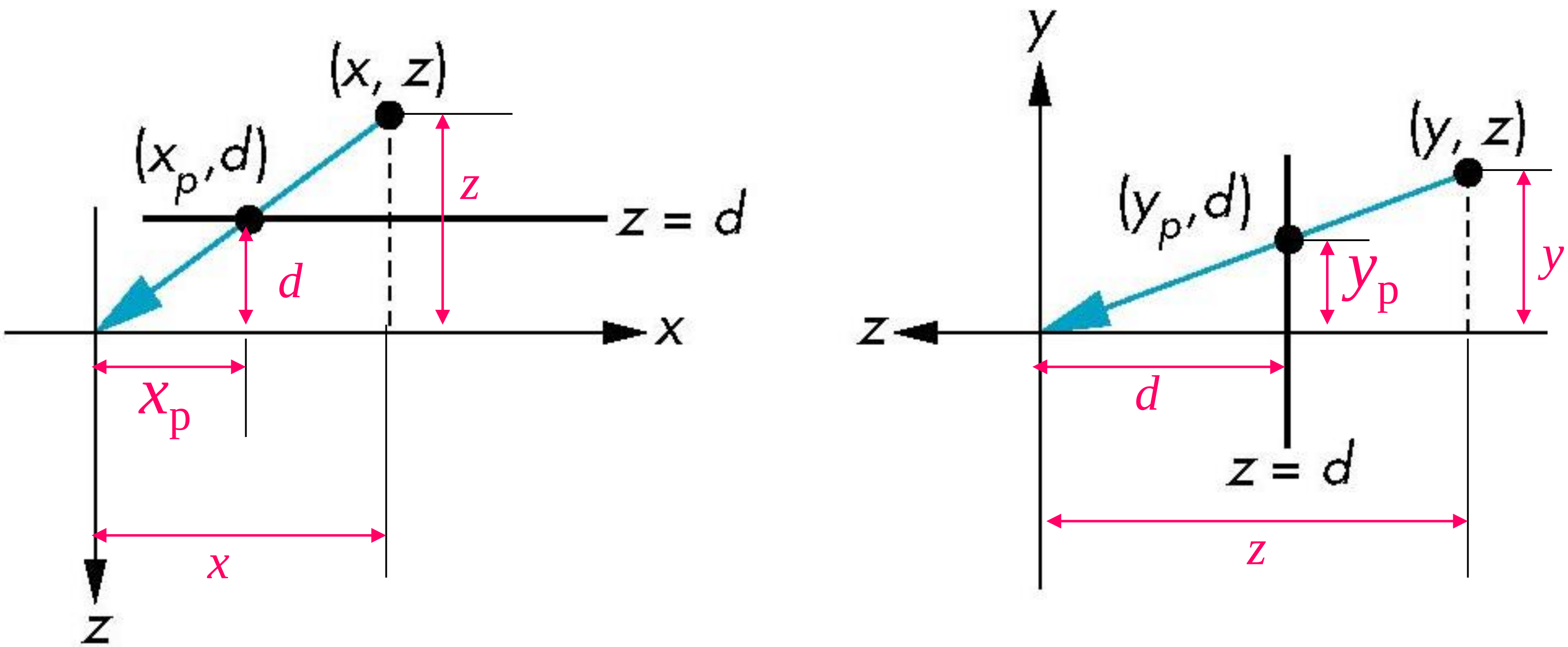
Simple Perspective

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



Perspective Equations

Consider top and side views



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

Homogeneous Coordinate Form

consider $\mathbf{q} = \mathbf{Mp}$ where

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

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

Perspective Division

- However $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This *perspective division* yields

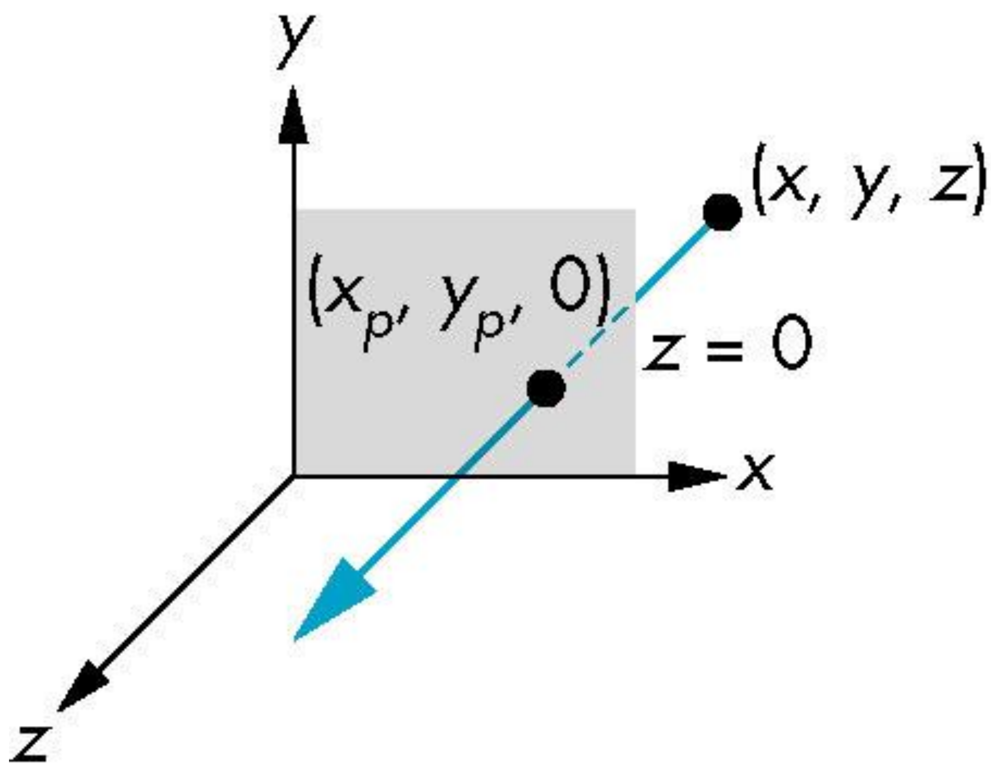
$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

the desired perspective equations

- We will consider the corresponding clipping volume with the OpenGL functions

Projection Pipeline

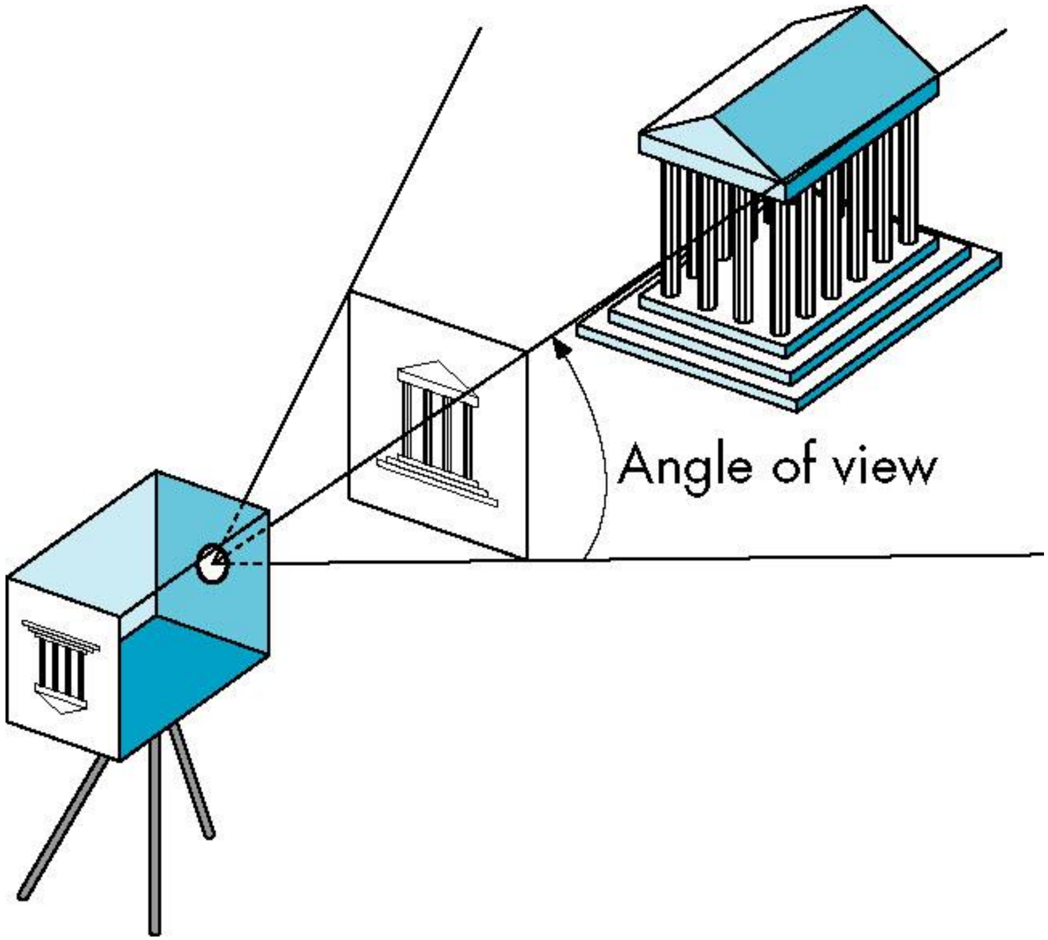
Orthogonal Projections



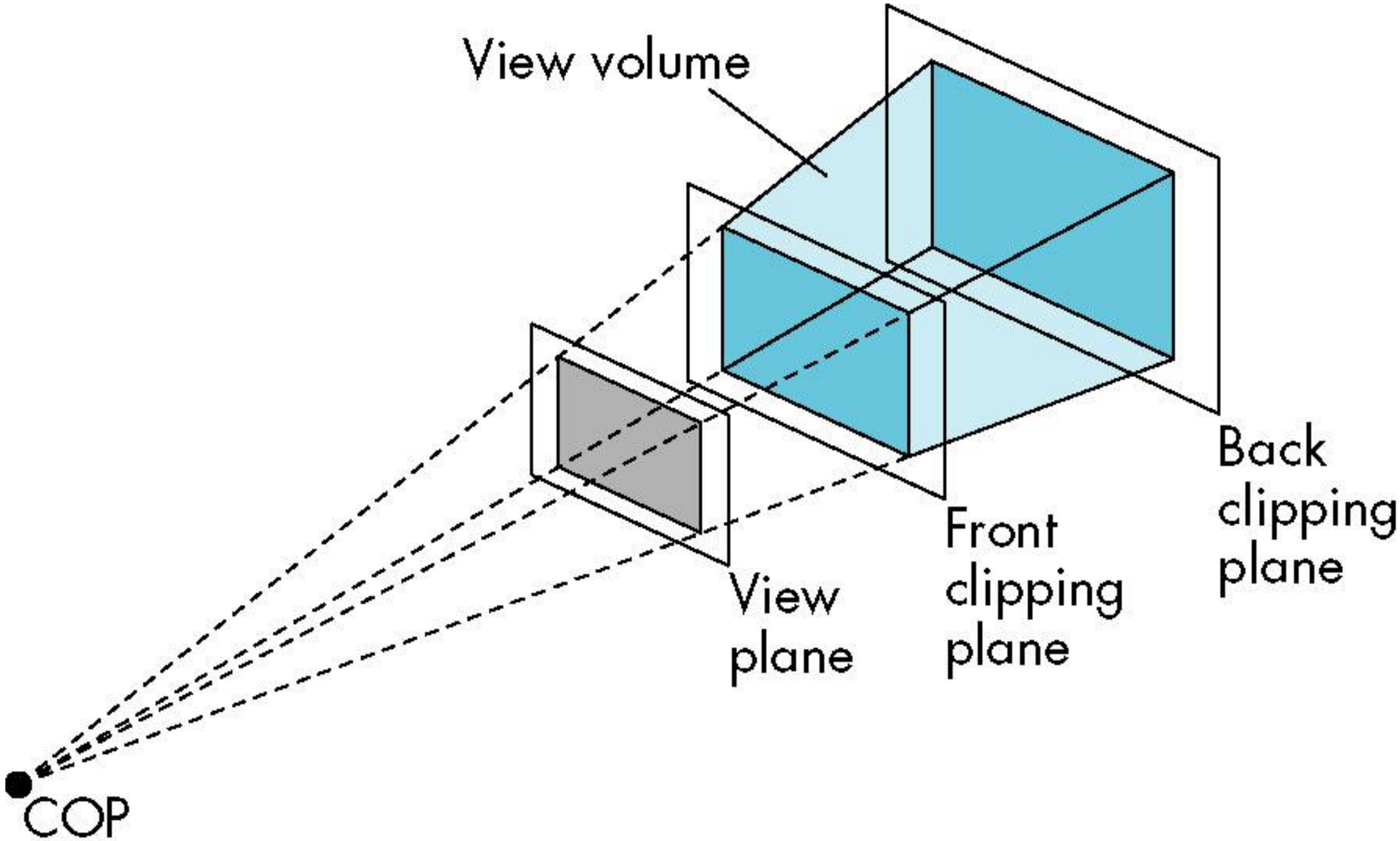
$$x_p = x \qquad y_p = y \qquad z_p = 0$$

Projection in OpenGL

- Definition of a view volume

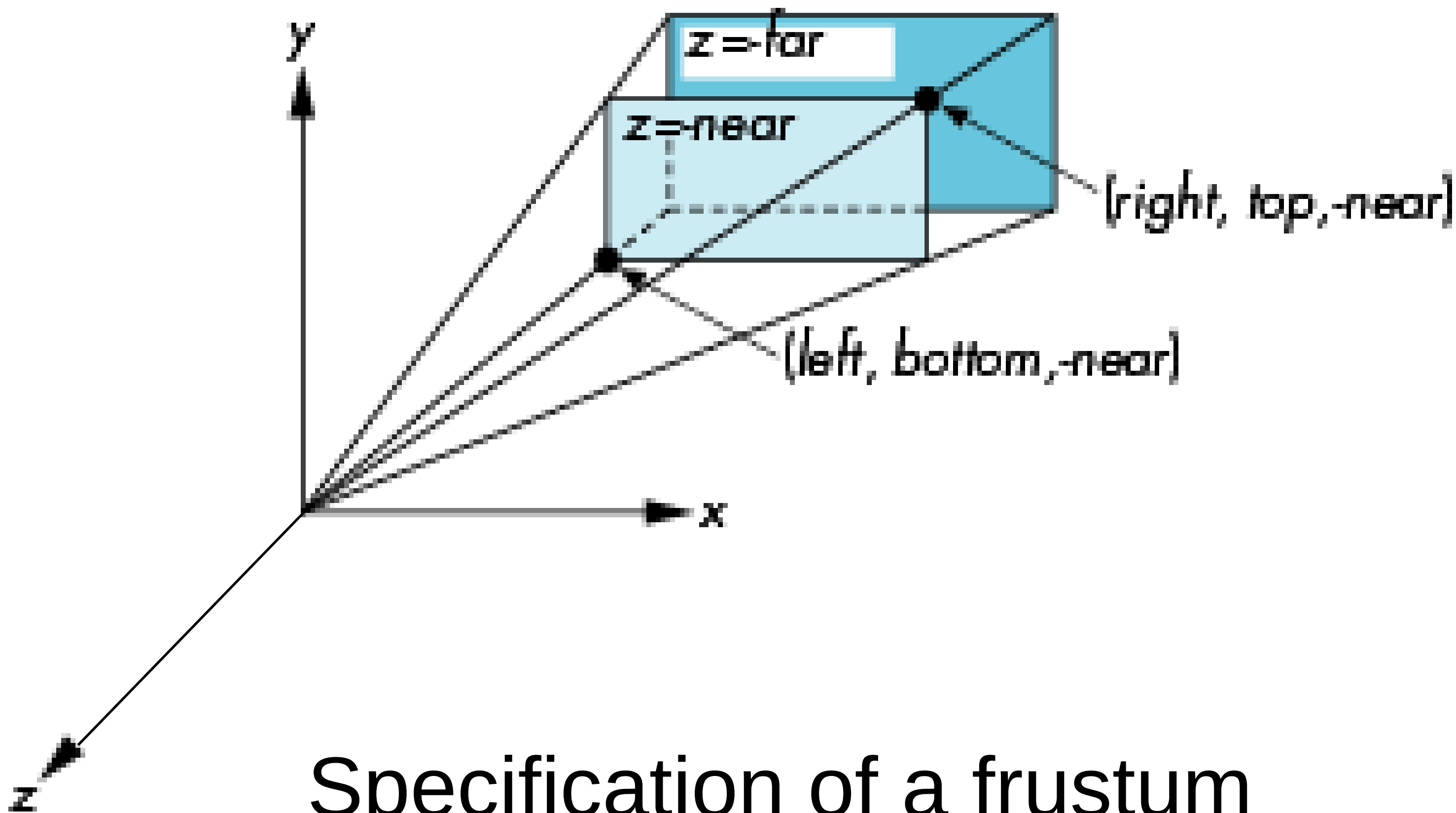


Front and Back Clipping Planes



OpenGL Perspective

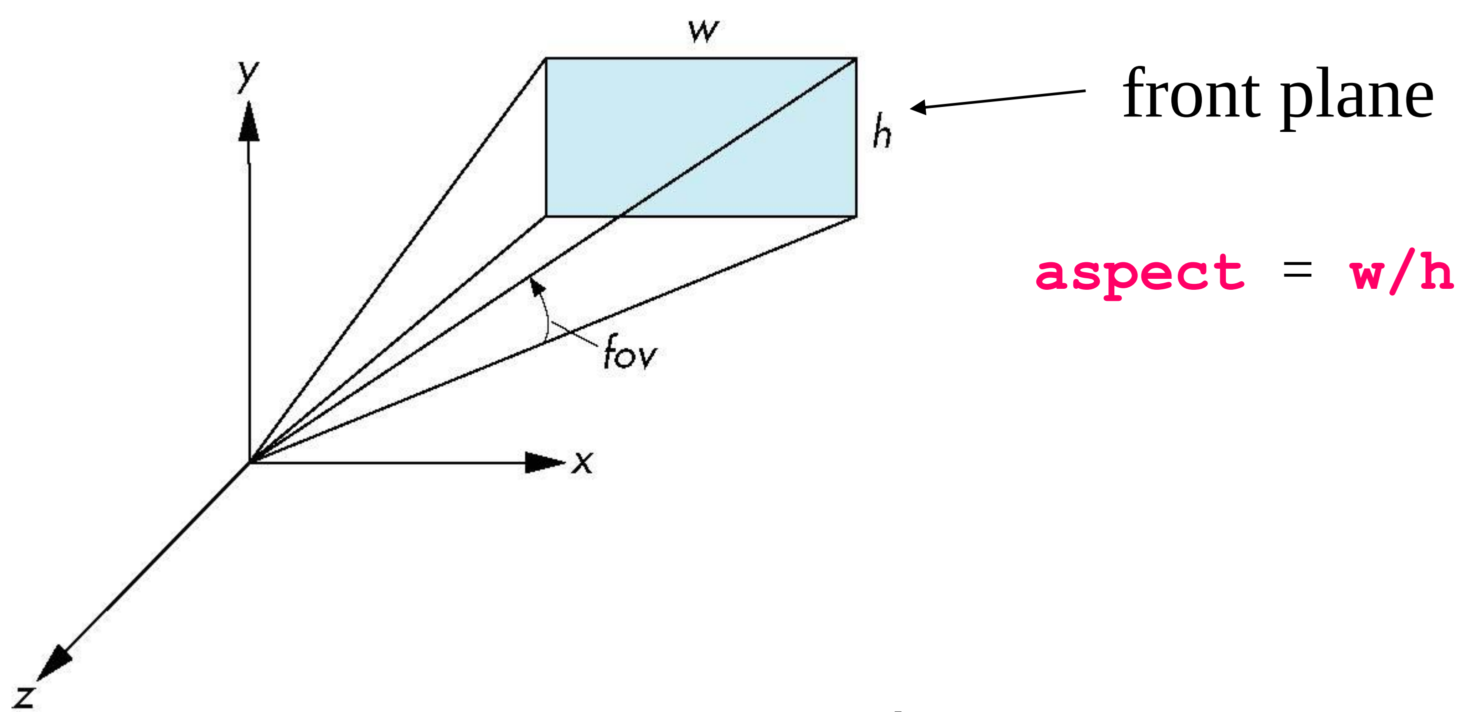
`glFrustum(left,right,bottom,top,near,far)`



Specification of a frustum

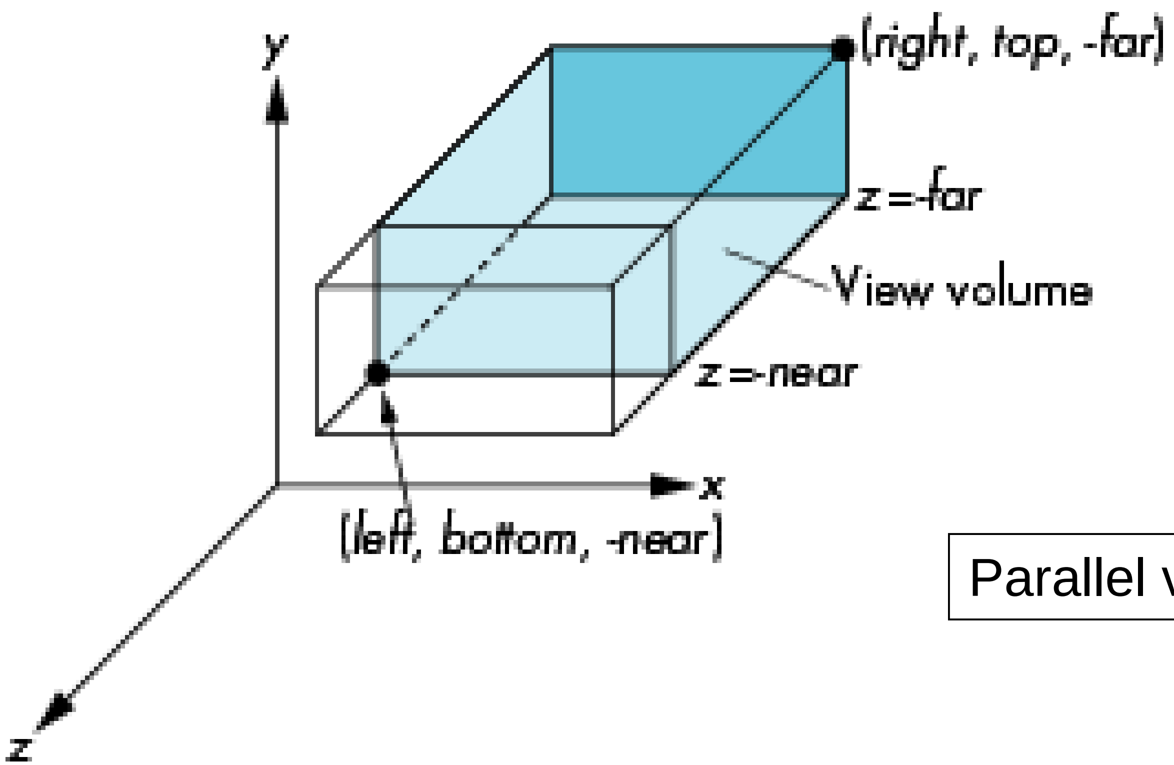
Using Field of View

- With **`glFrustum`** it is often difficult to get the desired view
- **`gluPerpective(fovy, aspect, near, far)`** often provides a better interface



OpenGL Orthogonal Viewing

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



Parallel viewing in OpenGL

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

Projection Matrices

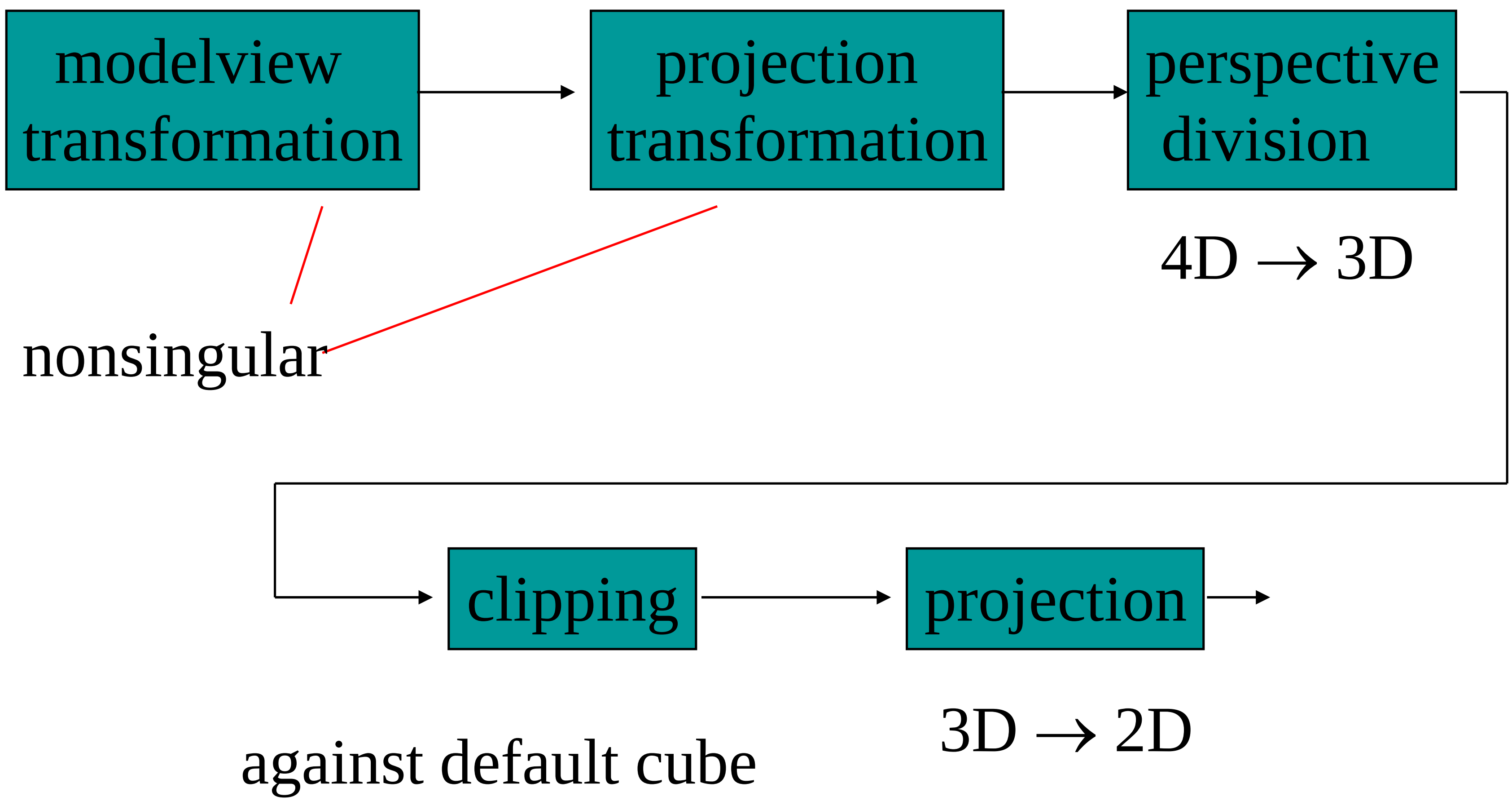
Objectives

- Derive the projection matrices used for standard OpenGL projections
- Introduce oblique projections
- Introduce projection normalization

Normalization

- 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



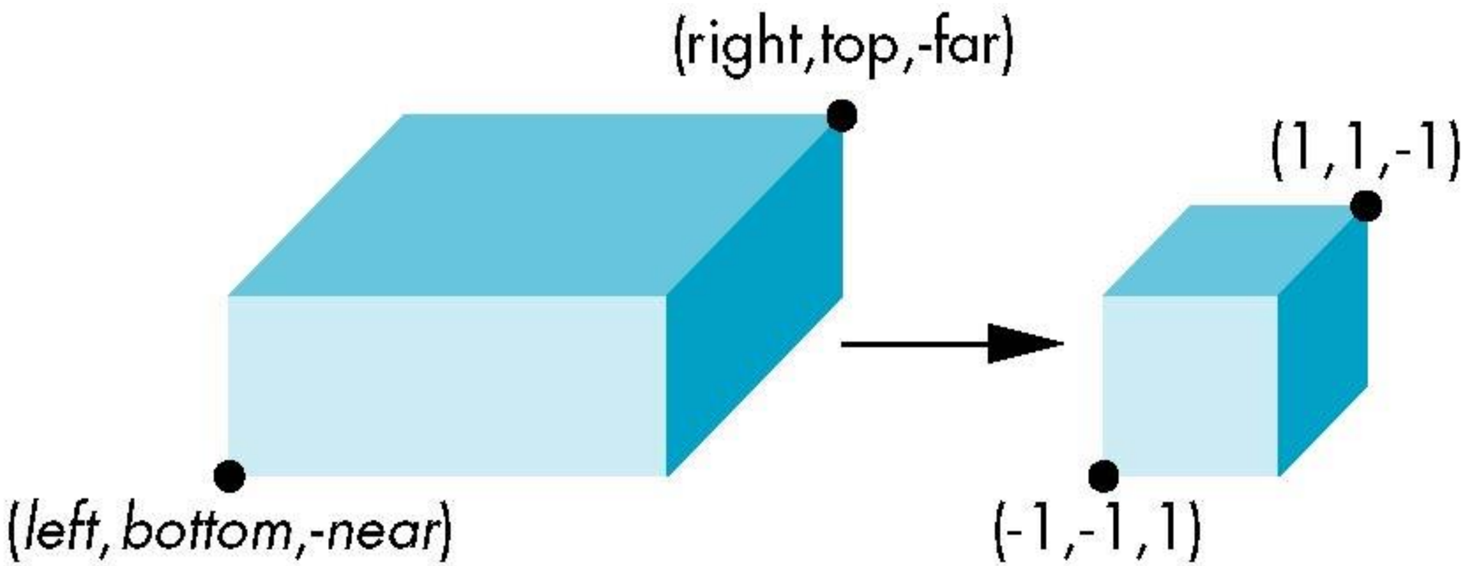
Notes

- 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

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

normalization ⇒ find transformation to convert specified clipping volume to default



Orthogonal 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))

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

Final Projection

- 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

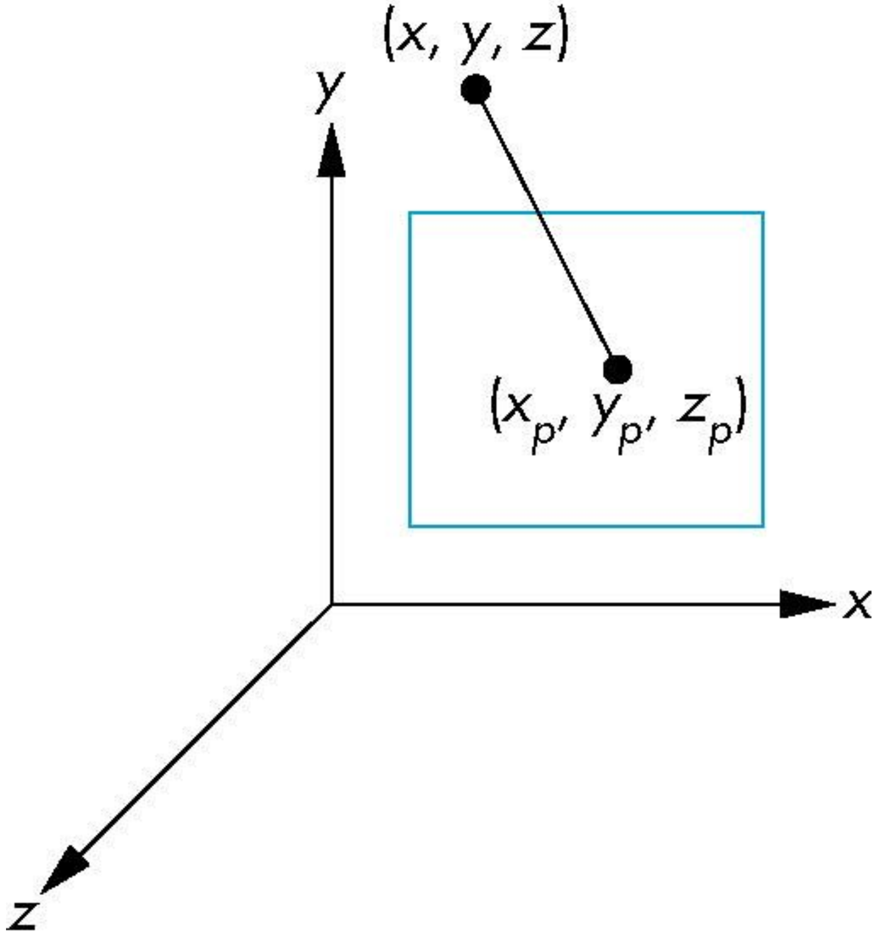
$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$$

Oblique Projections

- The OpenGL projection functions cannot produce general parallel projections such as

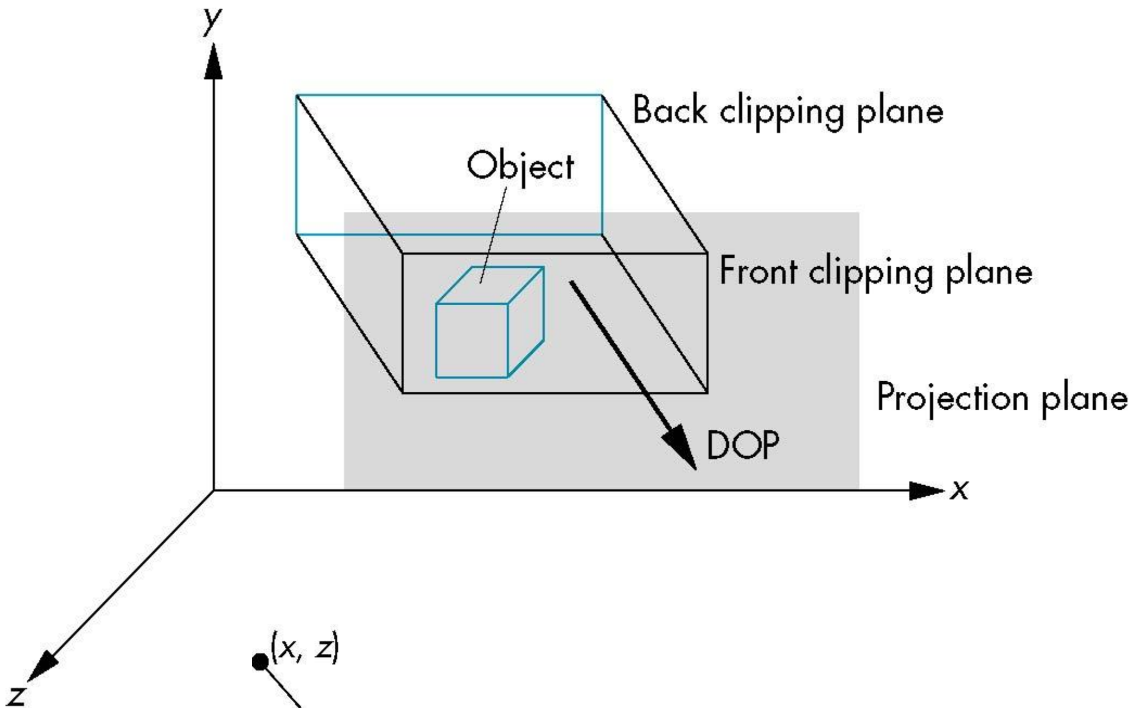
- However if we look at the example of the cube it appears that the cube has been sheared
- Oblique Projection = Shear + Orthogonal Projection

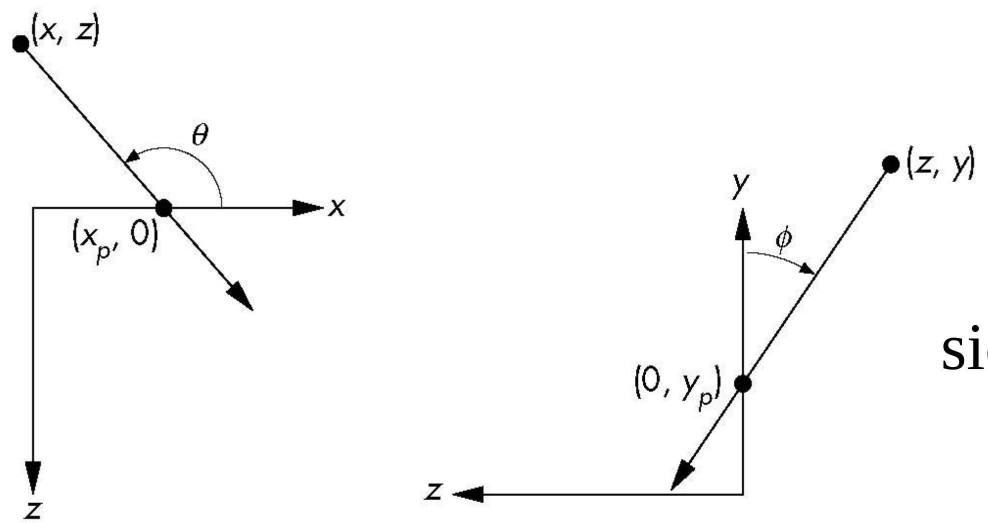
Oblique Projections



General Shear

Oblique Projection





top view

side view

Shear Matrix

xy shear (*z* values unchanged)

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

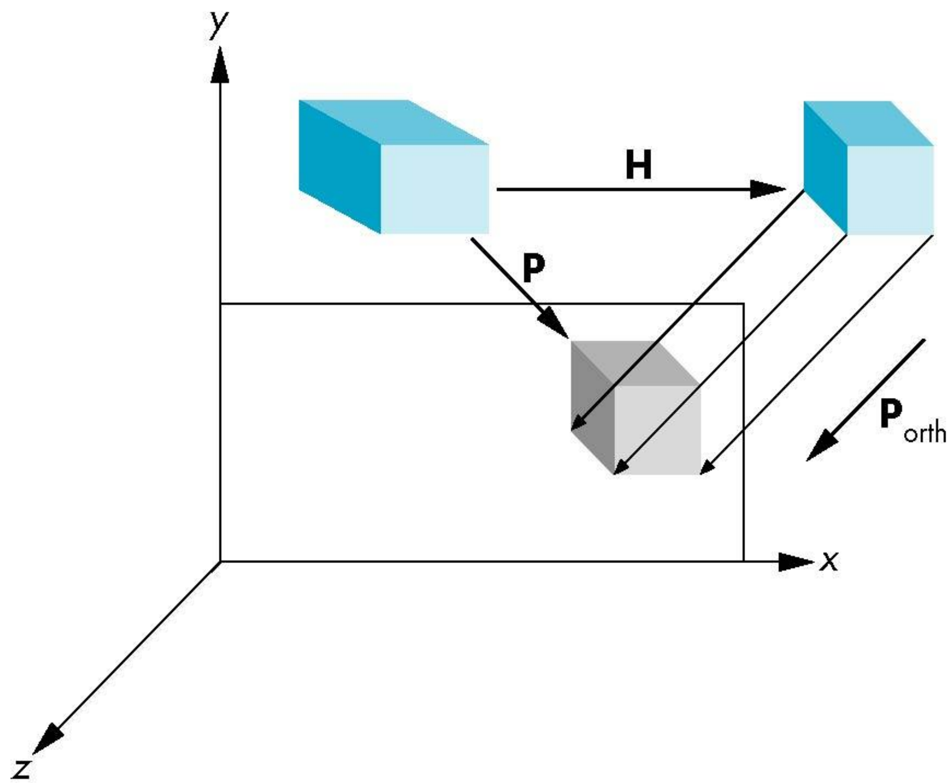
Projection matrix

$$\mathbf{P} = \mathbf{M}_{\text{orth}}\, \mathbf{H}(\theta,\phi)$$

General case:

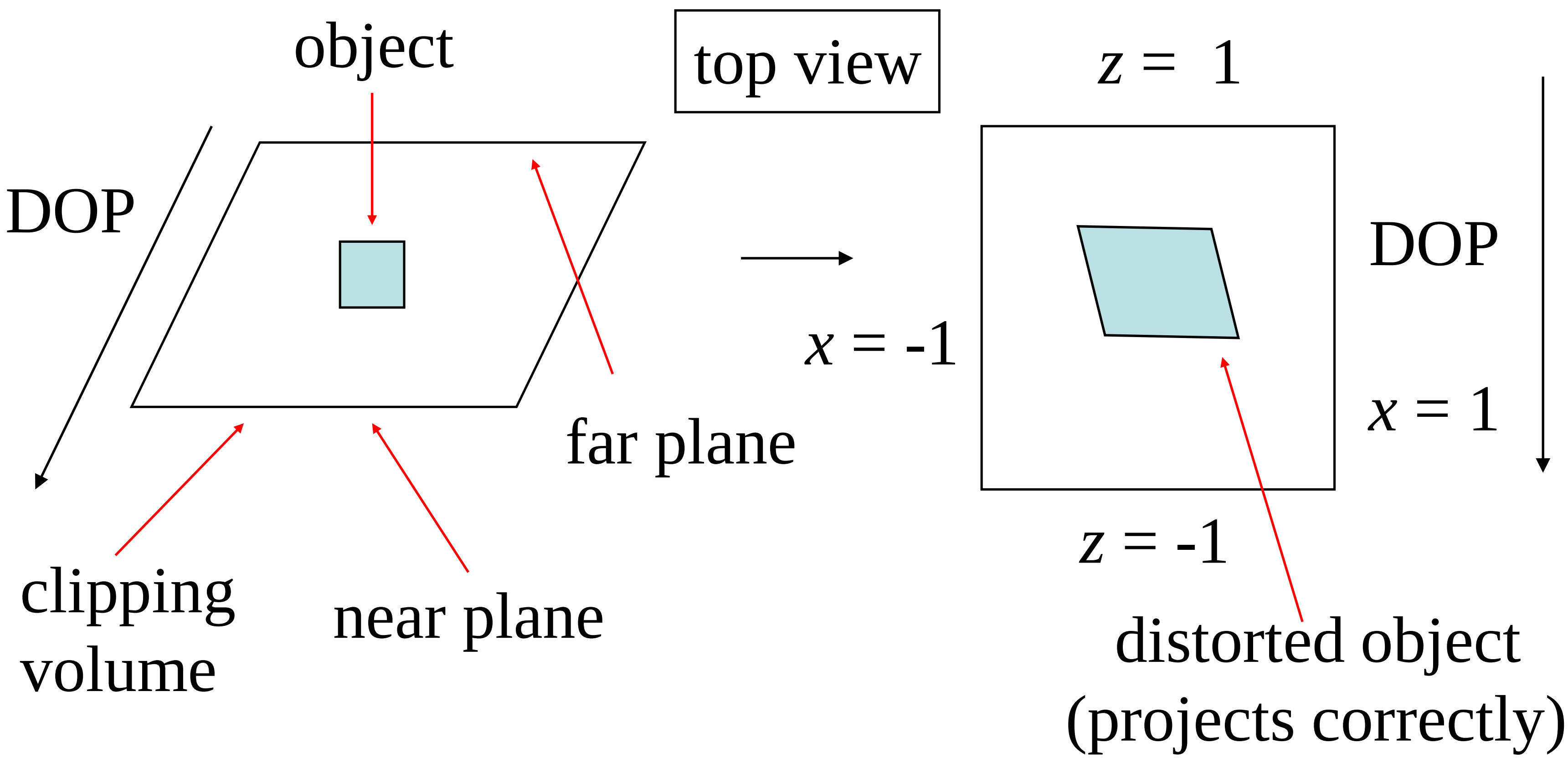
$$\mathbf{P} = \mathbf{M}_{\text{orth}}\, \mathbf{STH}(\theta,\phi)$$

Equivalency



Effect on Clipping

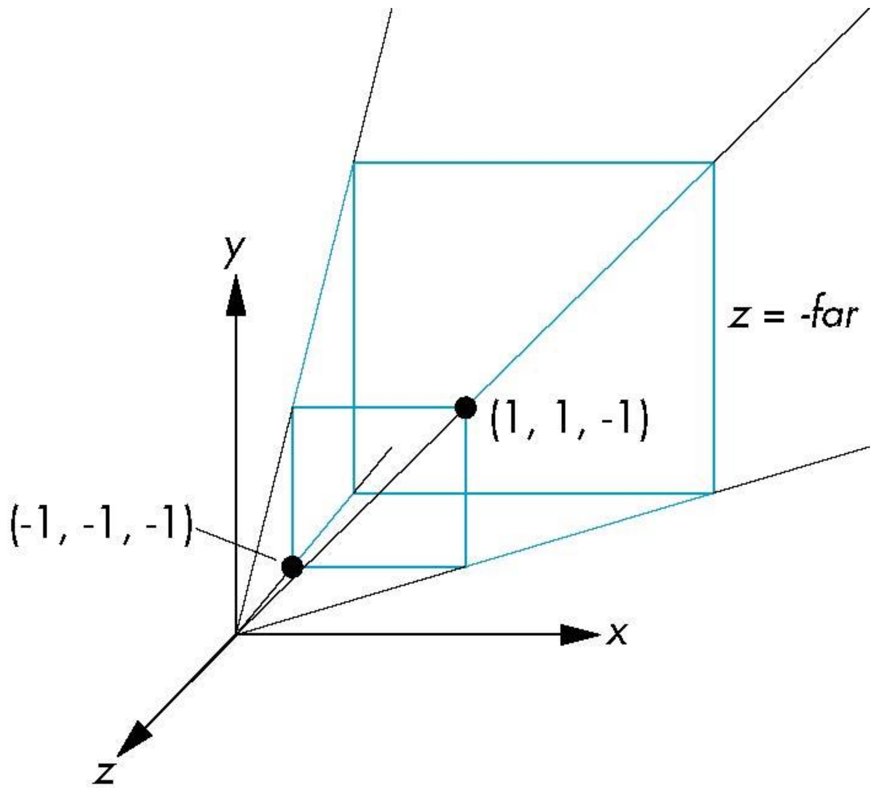
- The projection matrix $\mathbf{P} = \mathbf{STH}$ transforms the original clipping volume to the default clipping volume



Simple Perspective

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

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



Perspective Matrices

Simple projection matrix in homogeneous coordinates

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

Note that this matrix is independent of the far clipping plane

Generalization

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

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

Picking α and β

If we pick

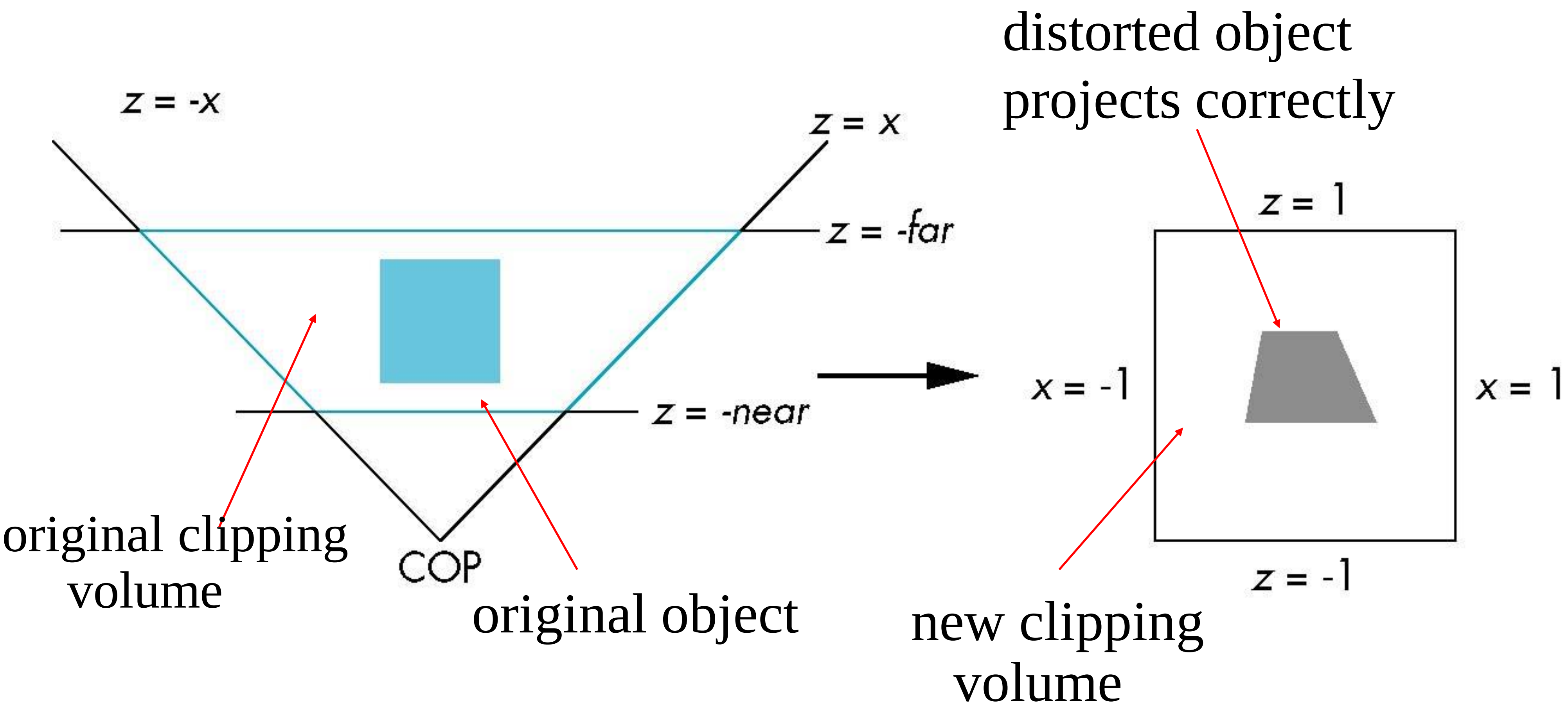
$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

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

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

Hence the new clipping volume is the default clipping volume

Normalization Transformation

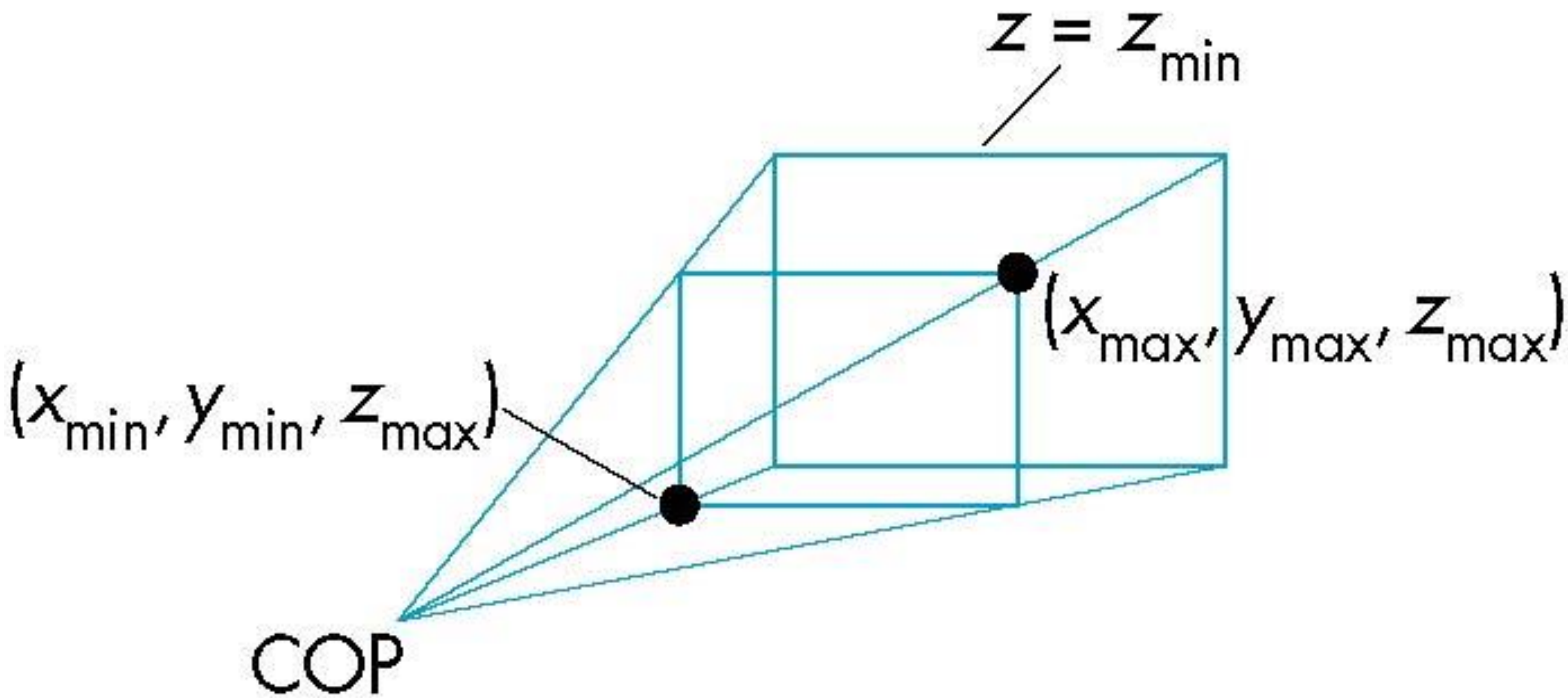


Normalization and Hidden-Surface Removal

- Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the for the transformed points $z_1' > z_2'$
- Thus hidden surface removal works if we first apply the normalization transformation
- However, the formula $z'' = -(\alpha+\beta/z)$ implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small

OpenGL Perspective

- **`glFrustum`** allows for an unsymmetric viewing frustum (although **`gluPerspective`** does not)



OpenGL Perspective Matrix

- The normalization in `glFrustum` requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation

$$\mathbf{P} = \mathbf{NSH}$$

our previously defined perspective matrix

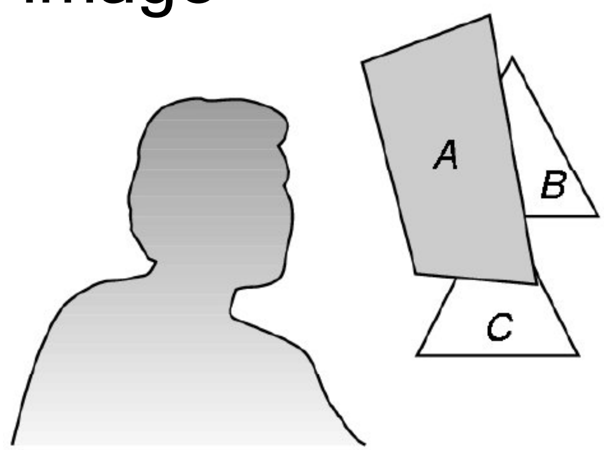
shear and scale

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

Hidden-Surface Removal

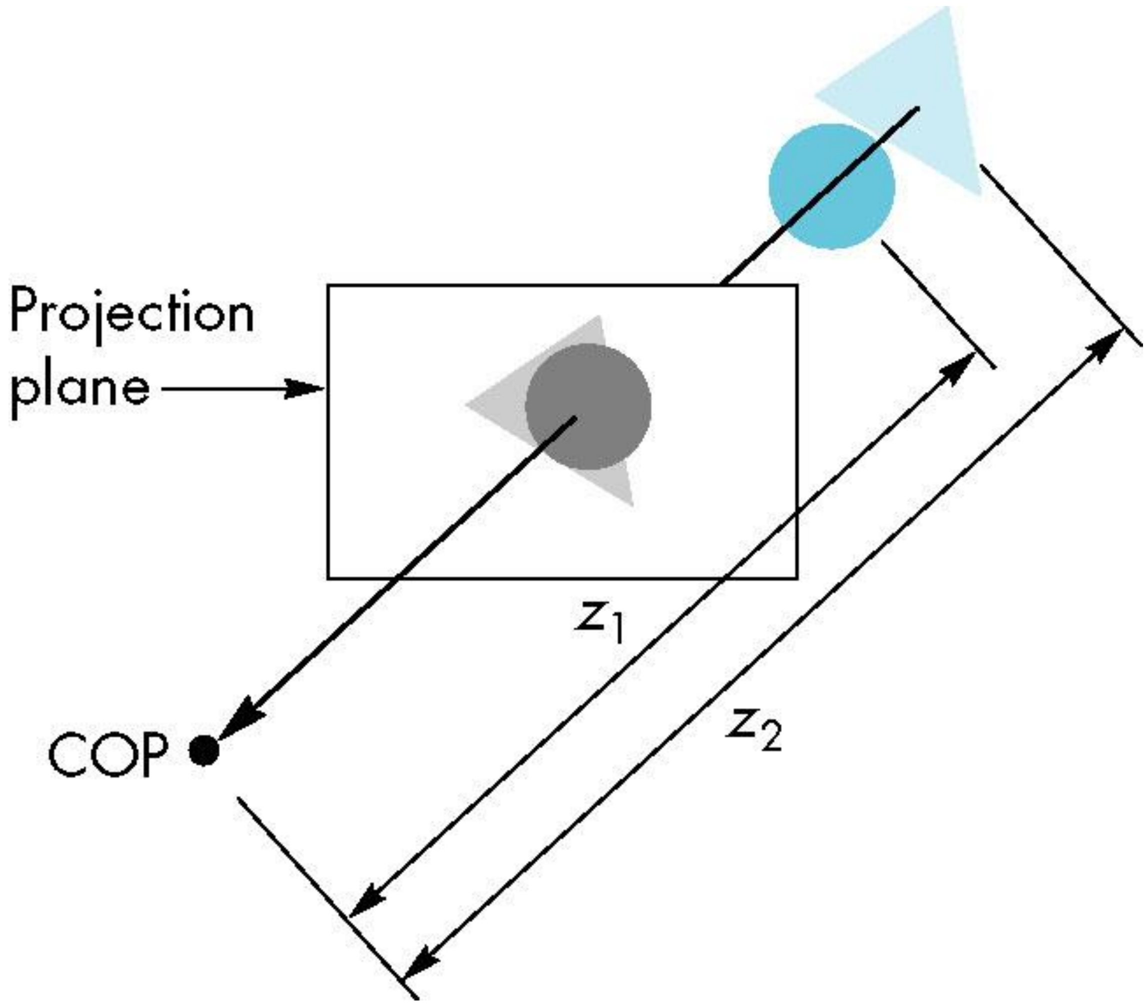
- We want to see only those surfaces in front of other surfaces
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



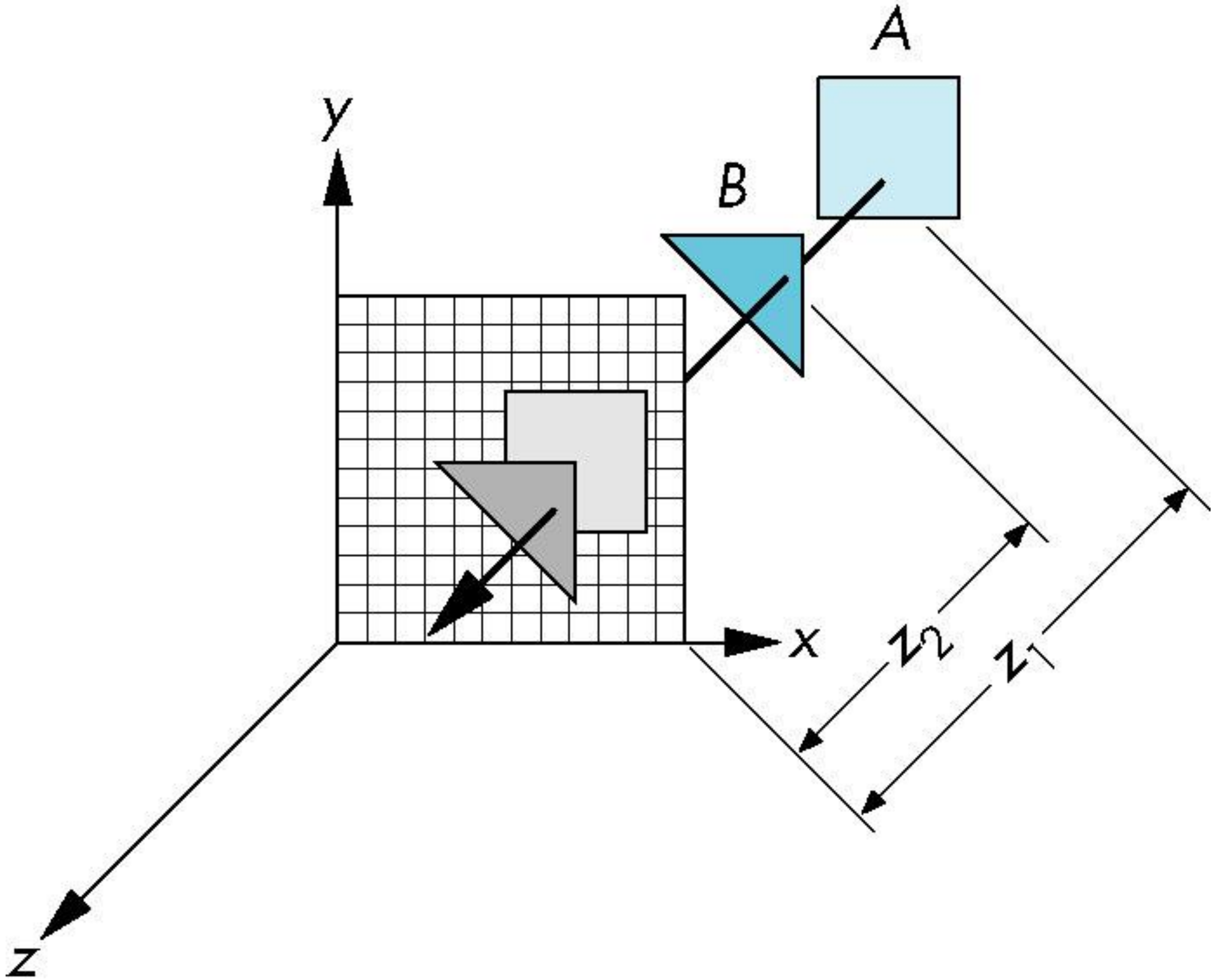
Using the z-buffer algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- It must be
  - Requested in **main.c**
    - **glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)**
  - Enabled in **init.c**
    - **glEnable(GL_DEPTH_TEST)**
  - Cleared in the display callback
    - **glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)**

The Z-buffer Algorithm

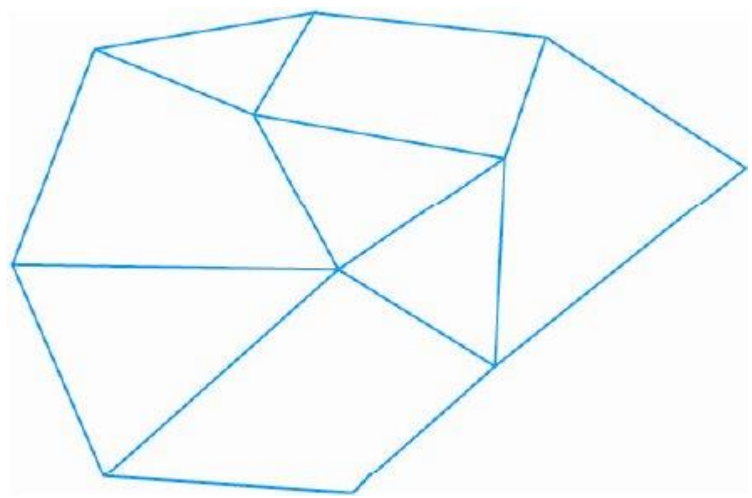


The Z-buffer Algorithm



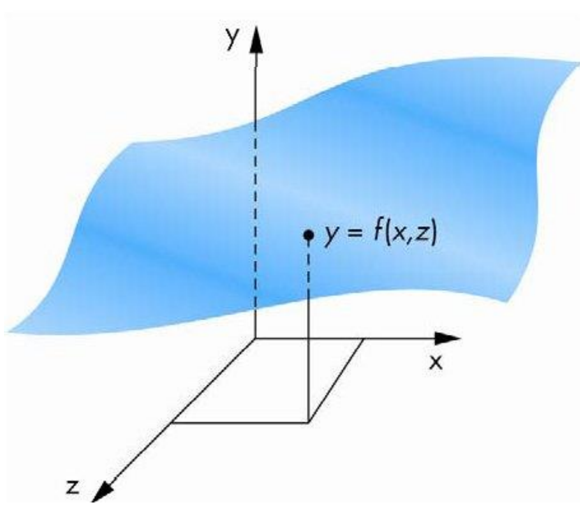
Interactive Mesh Displays

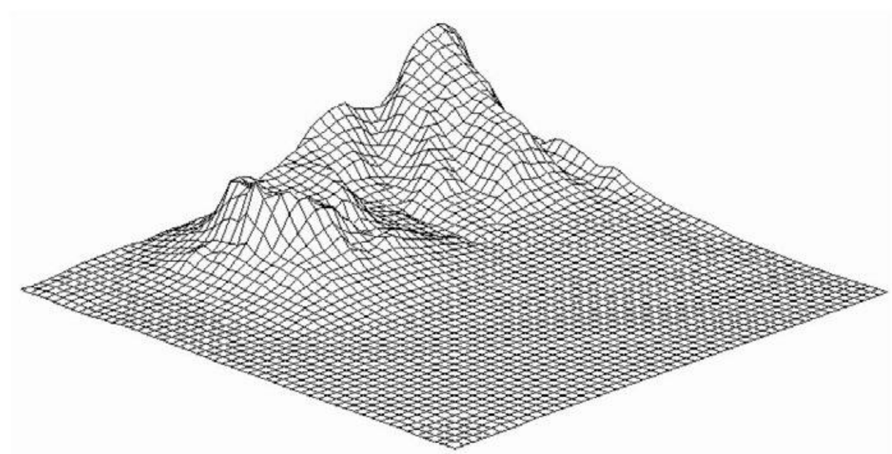
- Surfaces are represented by meshes
  - Usually triangular
- Approximate objects by polyhedral surfaces
  - Vertex and edge lists
  - Vertex arrays
- Use transformations to move camera and generate views



Height Fields

- Height fields or Digital Elevation Models (DEMs) are 2-1/2 D surfaces
- Used to represent terrain data





Display callback

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glLookAt(viewer[0], viewer[1], viewer[2],
        0.0, 0.0, 0.0,
        0.0, 1.0, 0.0);

    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);
    /* Draw surfaces*/
    mesh();
    glutSwapBuffers();
}
```

Reshape callback

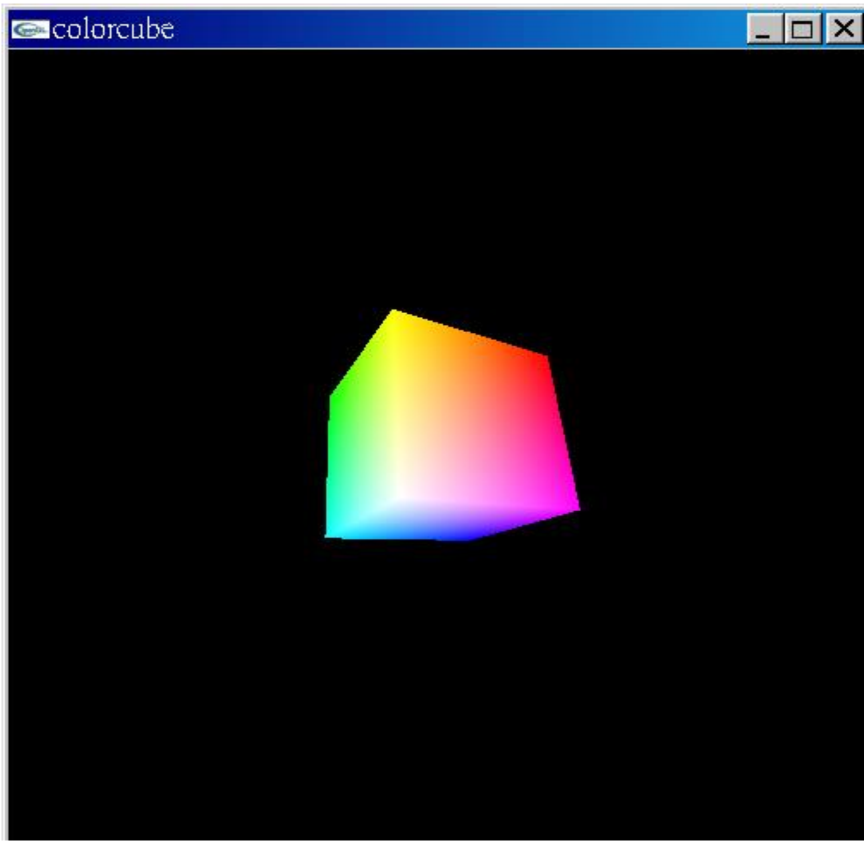
```
void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION); /* switch matrix mode */
   glLoadIdentity();
   if (w <= h)
        glFrustum (-2.0, 2.0, -2.0 * (GLfloat) h /
         (GLfloat) w, 2.0 * (GLfloat) h / (GLfloat) w,
        2.0, 20.0);
   else
        glFrustum (-2.0, 2.0, -2.0 * (GLfloat) w /
         (GLfloat) h, 2.0 * (GLfloat) w / (GLfloat) h,
        2.0, 20.0);

   glMatrixMode(GL_MODELVIEW);
   /* return to modelview mode */
}
```

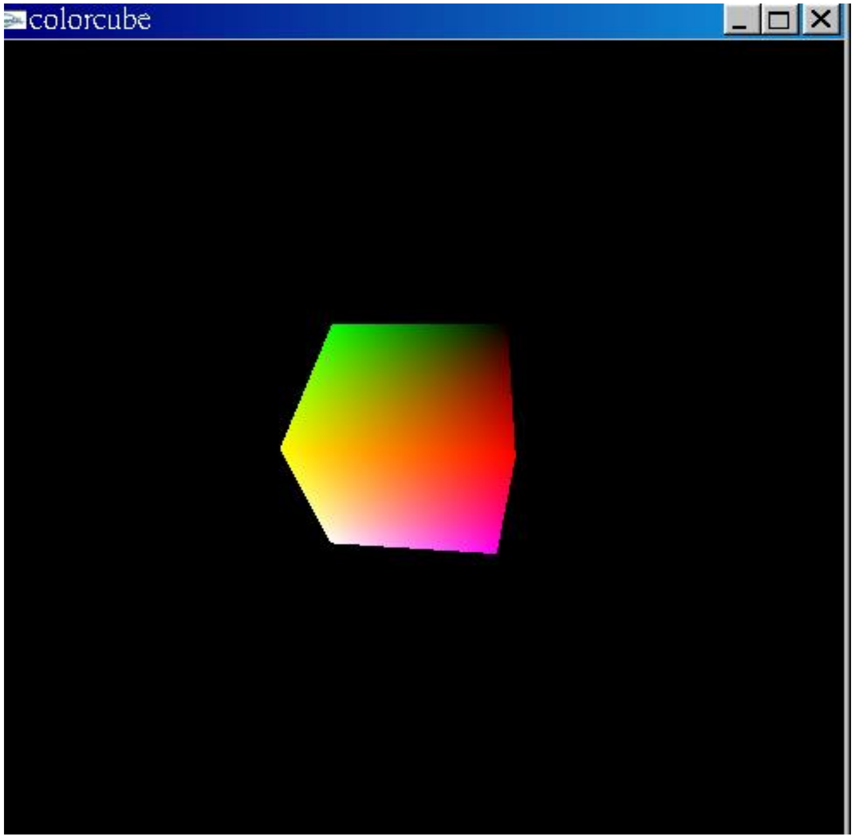
glFrustum(left,right,bottom,top,near,far)

Sample Programs

- Moving viewer
  – A.11 cubeview.c

A.11 cubeview.c (1/7)





A.11 cubeview.c (2/7)

```
#include <stdlib.h>
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif


    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 normals[][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[][3] = {{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}};
```

A.11 cubeview.c (3/7)

```
void polygon(int a, int b, int c , int d)
{
        glBegin(GL_POLYGON);
                glColor3fv(colors[a]);
                glNormal3fv(normals[a]);
                glVertex3fv(vertices[a]);
                glColor3fv(colors[b]);
                glNormal3fv(normals[b]);
                glVertex3fv(vertices[b]);
                glColor3fv(colors[c]);
                glNormal3fv(normals[c]);
                glVertex3fv(vertices[c]);
                glColor3fv(colors[d]);
                glNormal3fv(normals[d]);
                glVertex3fv(vertices[d]);
        glEnd();
}
void colorcube()
{
        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);
}
```

A.11 cubeview.c (4/7)

```
static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;
static GLdouble viewer[]= {0.0, 0.0, 5.0}; /* initial viewer location */

void display(void)
{

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/* Update viewer position in modelview matrix */

        glLoadIdentity();
        gluLookAt(viewer[0],viewer[1],viewer[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

/* rotate cube */

        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();

 glFlush();
 glutSwapBuffers();
}
```

A.11 cubeview.c (5/7)

```
void mouse(int btn, int state, int x, int y)
{
        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
        theta[axis] += 2.0;
        if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        display();
}

void keys(unsigned char key, int x, int y)
{

/* Use x, X, y, Y, z, and Z keys to move viewer */

   if(key == 'x') viewer[0]-= 1.0;
   if(key == 'X') viewer[0]+= 1.0;
   if(key == 'y') viewer[1]-= 1.0;
   if(key == 'Y') viewer[1]+= 1.0;
   if(key == 'z') viewer[2]-= 1.0;
   if(key == 'Z') viewer[2]+= 1.0;
   display();
}
```

A.11 cubeview.c (6/7)

```
void myReshape(int w, int h)
{
 glViewport(0, 0, w, h);

/* Use a perspective view */

 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 if (w<=h) glFrustum(-2.0, 2.0, -2.0 * (GLfloat) h/ (GLfloat) w,
                      2.0* (GLfloat) h / (GLfloat) w, 2.0, 20.0);
   else      glFrustum(-2.0, 2.0, -2.0 * (GLfloat) w/ (GLfloat) h,
                      2.0* (GLfloat) w / (GLfloat) h, 2.0, 20.0);

/* Or we can use gluPerspective */

 /* gluPerspective(45.0, w/h, -10.0, 10.0); */

 glMatrixMode(GL_MODELVIEW);
}
```

A.11 cubeview.c (7/7)

```
void
main(int argc, char **argv)
{
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
 glutInitWindowSize(500, 500);
 glutCreateWindow("colorcube");
 glutReshapeFunc(myReshape);
 glutDisplayFunc(display);
 glutMouseFunc(mouse);
 glutKeyboardFunc(keys);
 glEnable(GL_DEPTH_TEST);
 glutMainLoop();
}
```