CLASSICAL VIEWING

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

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
  1. One or more **objects**
  2. A **viewer** with a projection surface
  3. **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 he would like to see it

- Each object is assumed to be constructed from flat *principal faces*
  - Buildings, polyhedra, manufactured objects

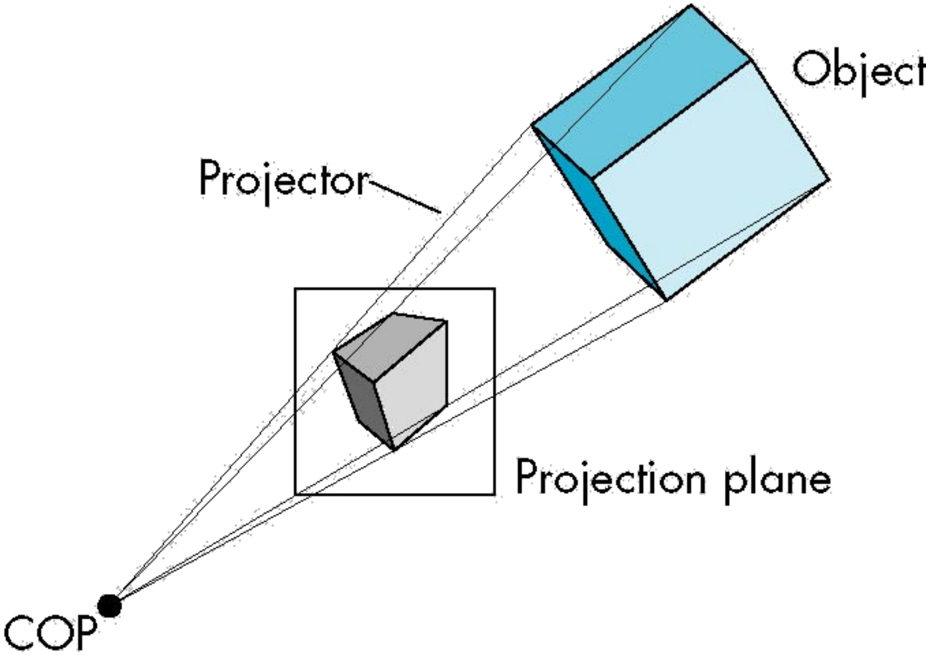
Planar Geometric Projections

- Standard projections project onto a plane

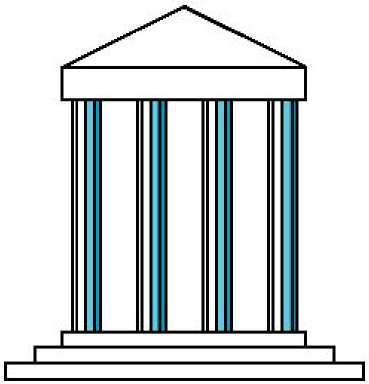
- **Planar projectors** are lines that either
    1. converge at a center of projection
    2. are parallel

  - Planar projections preserve lines
  - but not necessarily angles

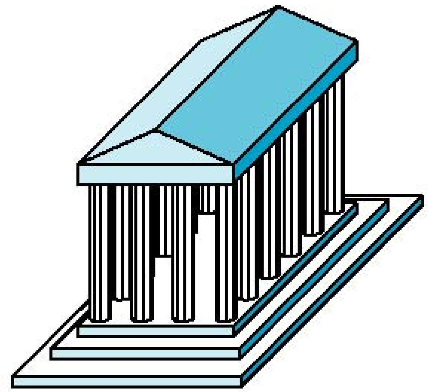
- **Nonplanar projections** are needed for applications such as **map construction**



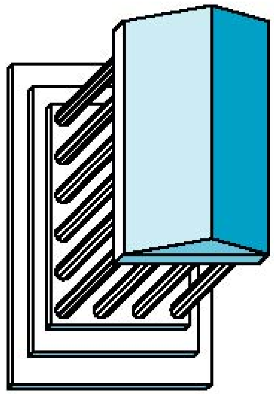
Classical Projections



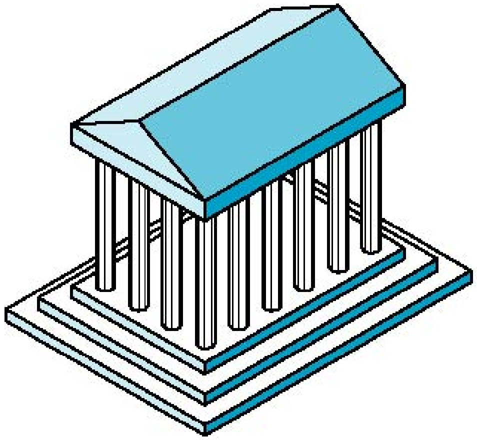
Front elevation



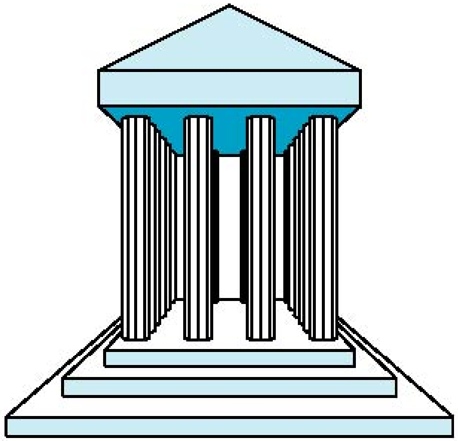
Elevation oblique



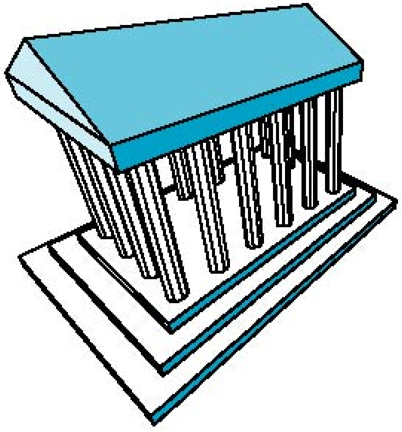
Plan oblique



Isometric



One-point perspective

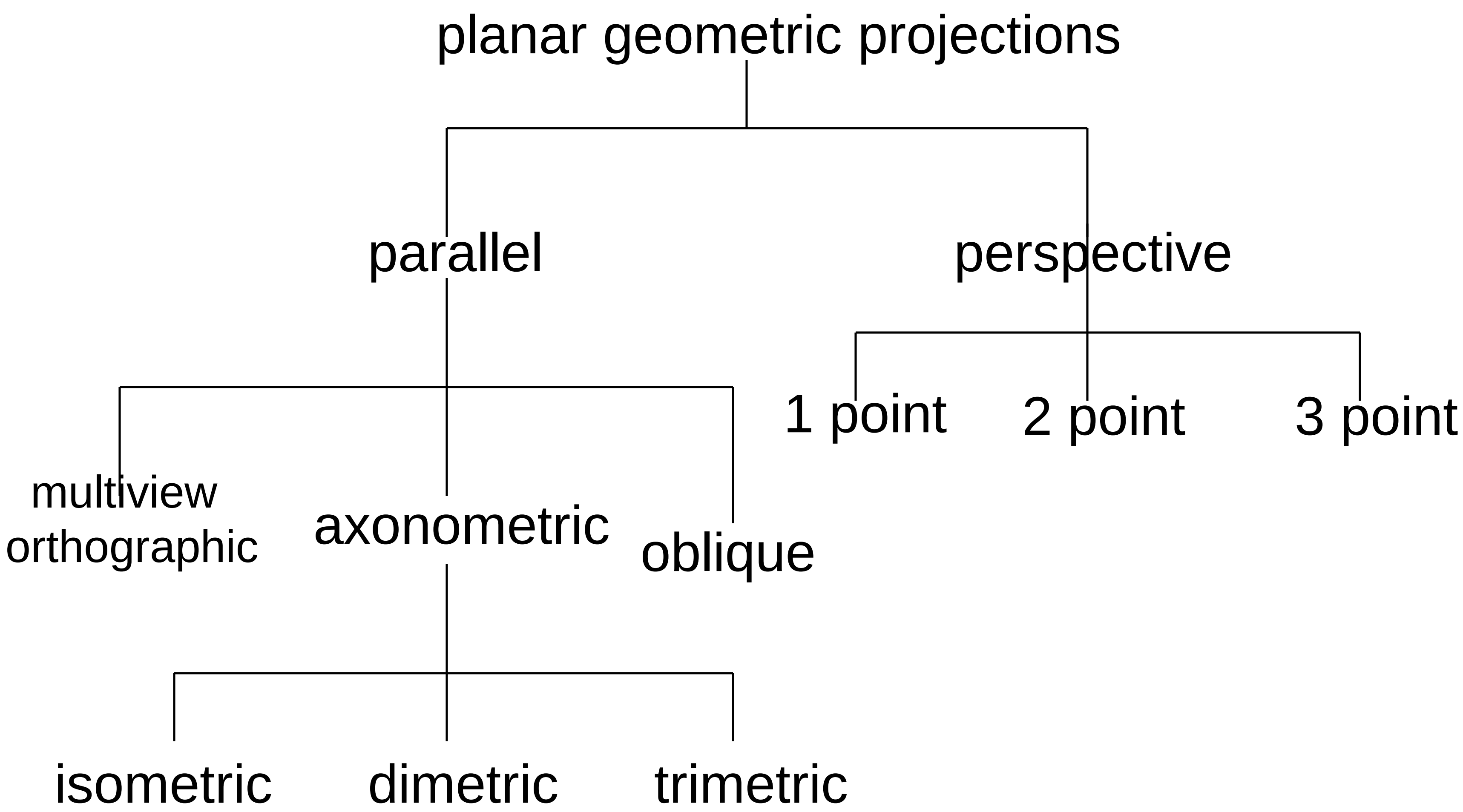


Three-point perspective

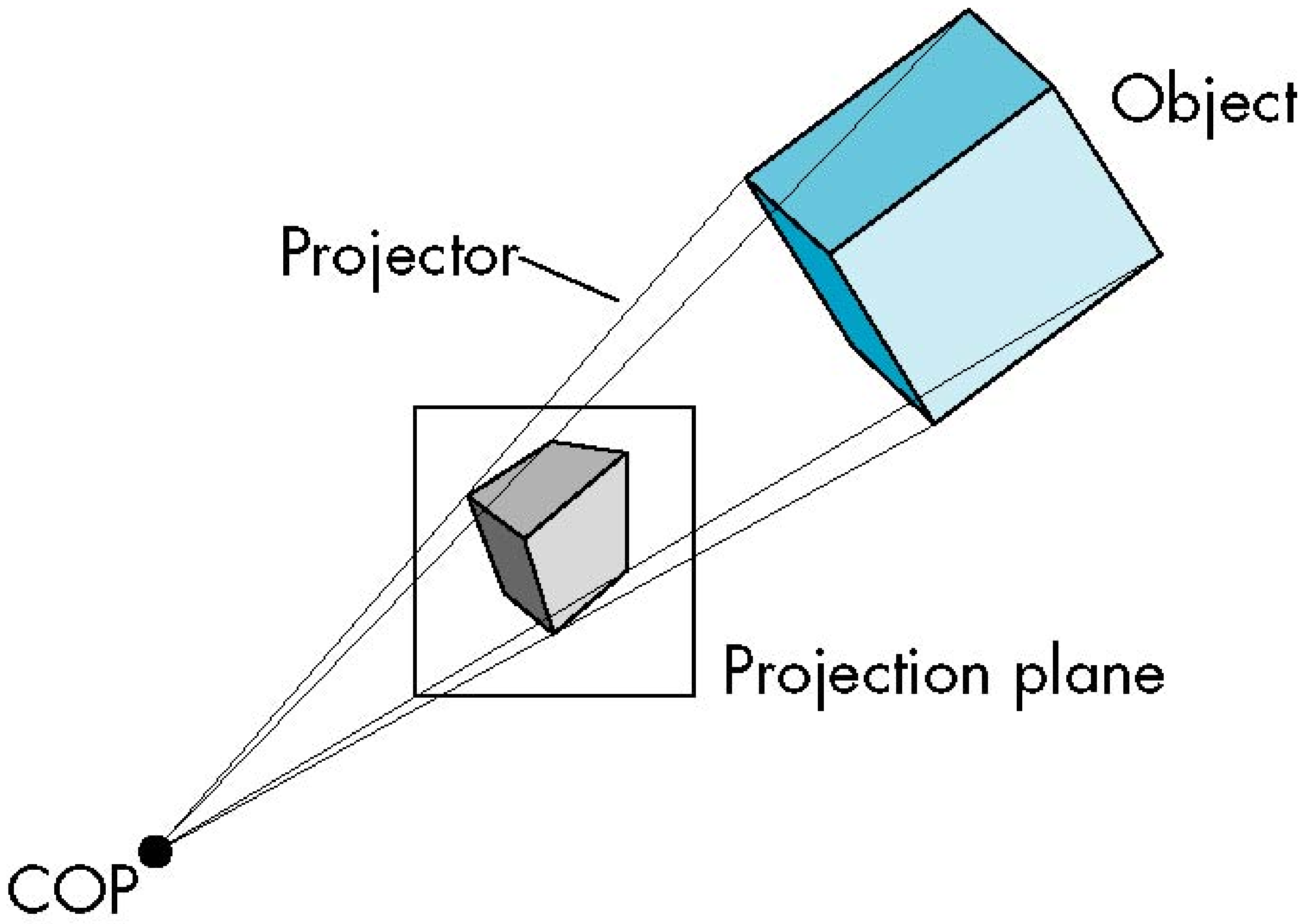
Perspective vs Parallel

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

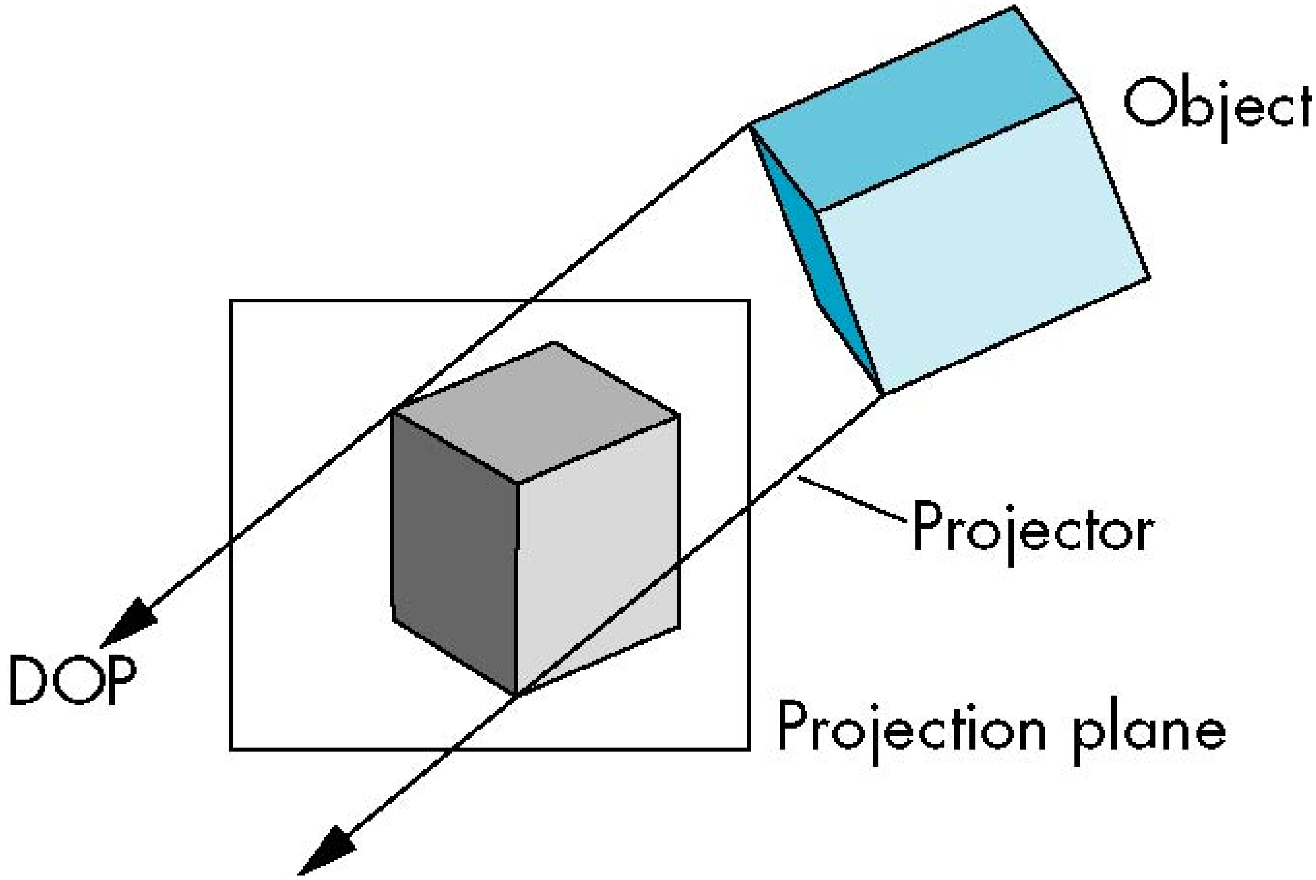
Taxonomy of Planar Geometric Projections



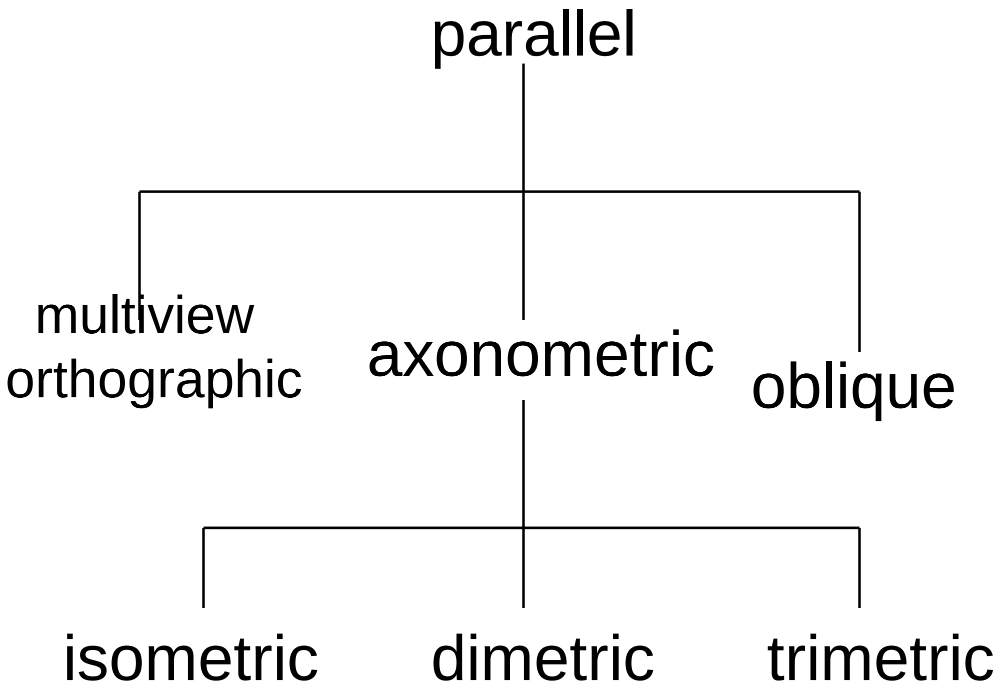
Perspective Projection



Parallel Projection

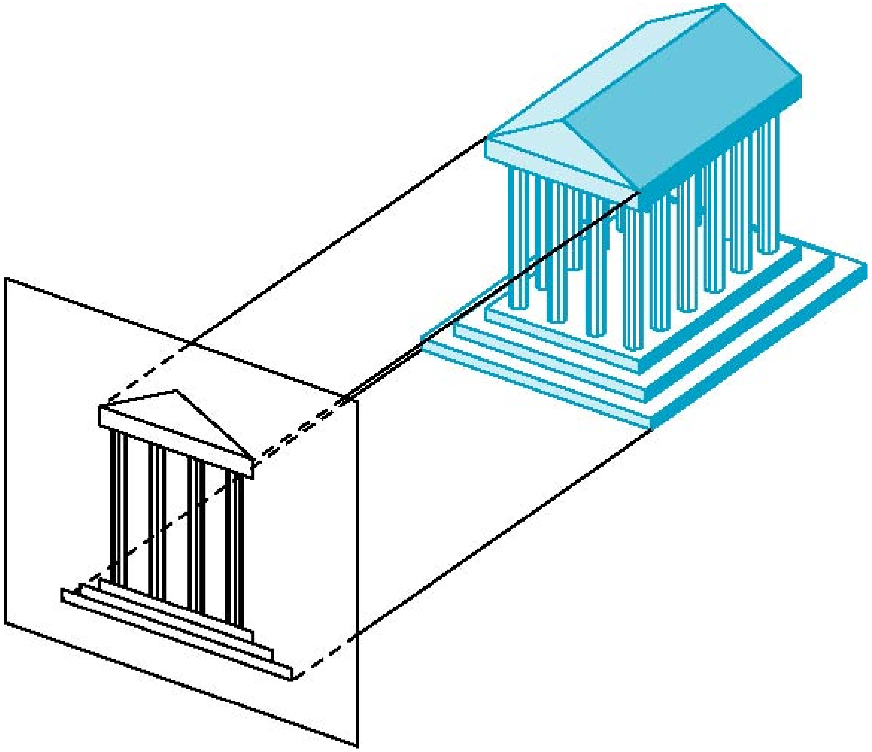


Parallel Projections



Orthographic Projection

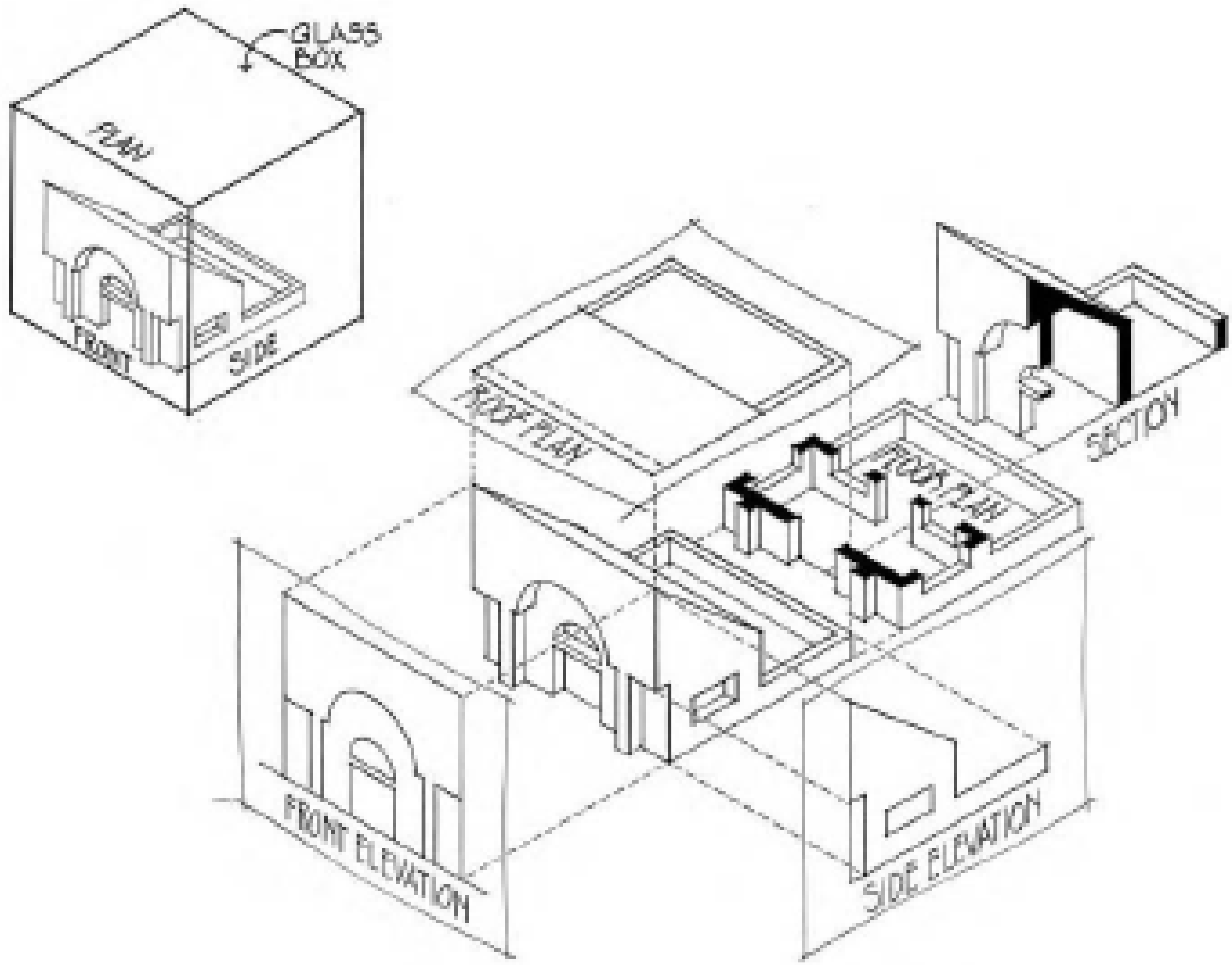
- Projectors are orthogonal to projection surface



Orthographic Projection

- The word *orthographic* refers to the projection system that is used to derive multiview drawings based on the glass box model.
- Drawings that appear on a surface are the view a person sees on the transparent viewing plane that is positioned perpendicular to the viewer’s line of sight and the object.
- In the orthographic system, the object is placed in a series of positions (plan or elevation) relative to the viewing plane.

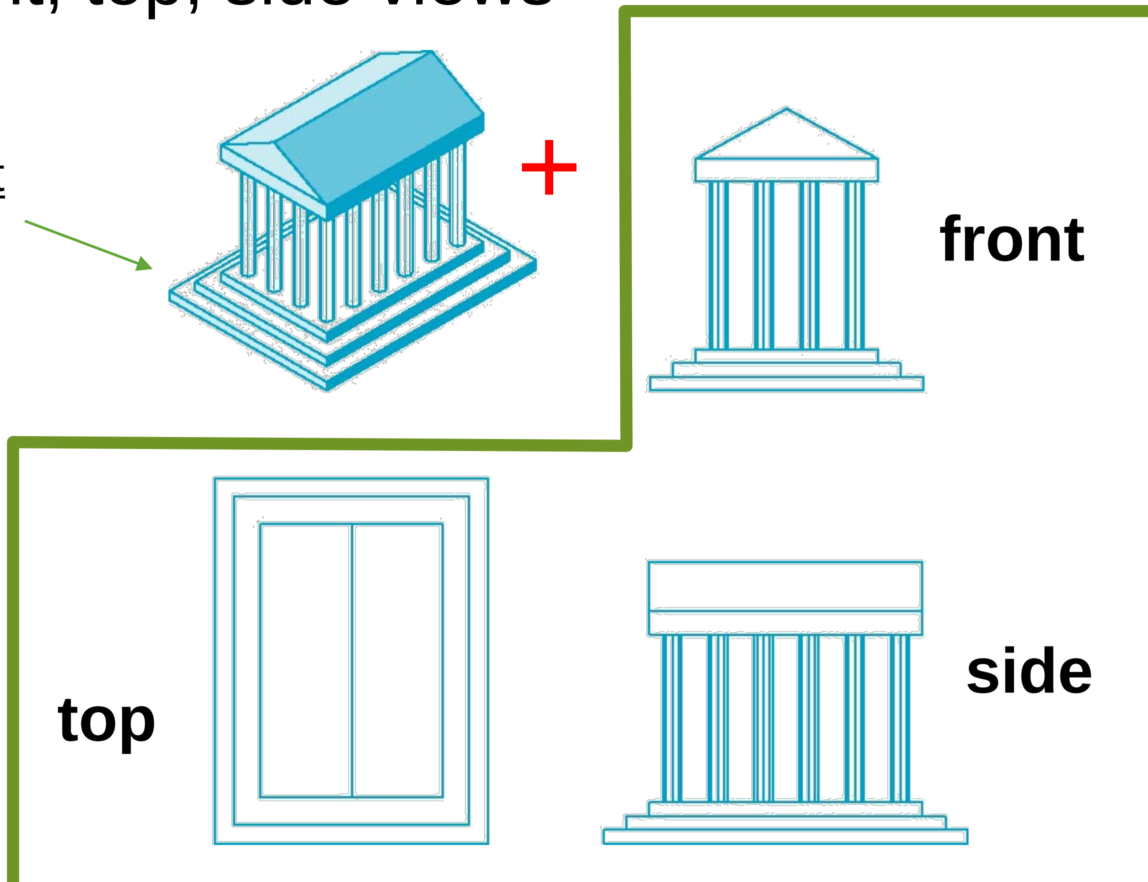
Multiview: Orthographic Projections



Multiview Orthographic Projection

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

Multiview Orth. is usually presented with isometric (not one of orthographic views)



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

Advantages and Disadvantages of Orthographic Projection

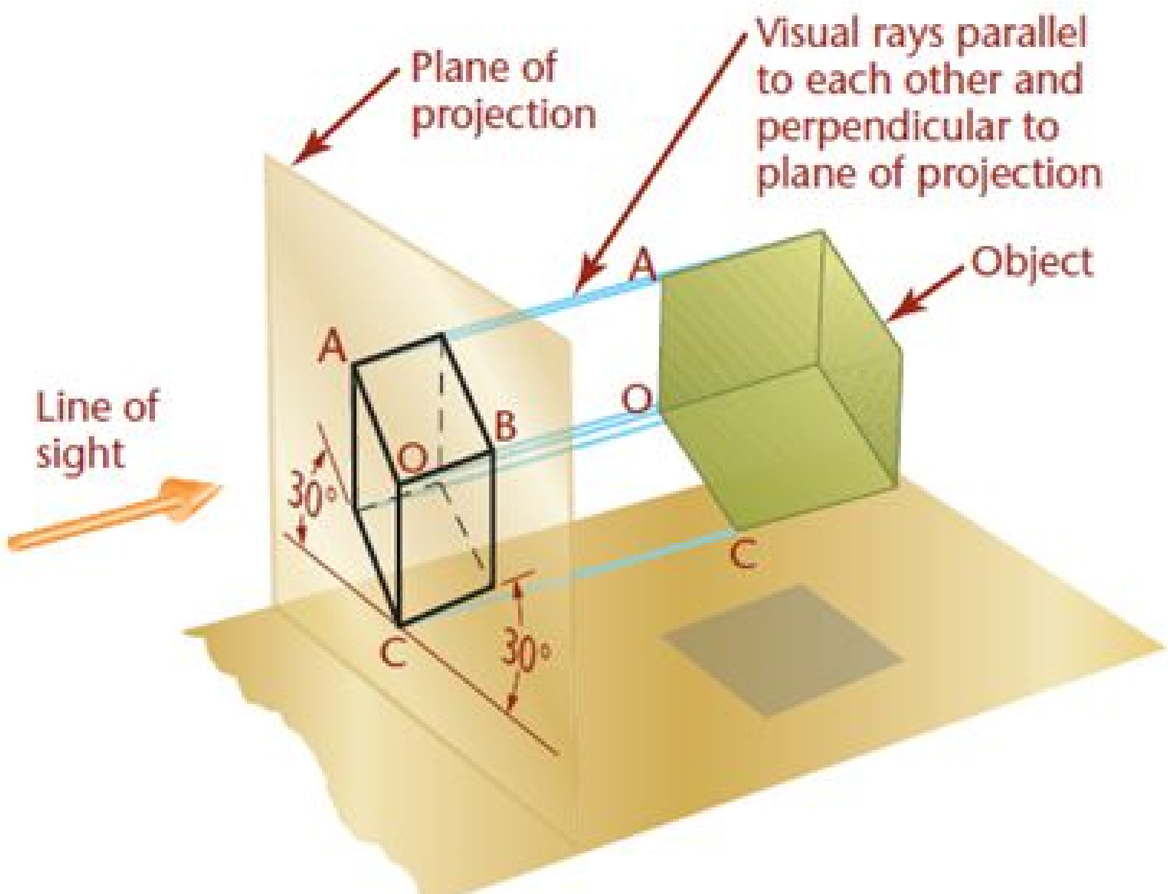
- Preserves **both distances and angles**
  1. Shapes preserved
  2. Relative sizes preserved
  3. Can be used for measurements
     - Building plans
     - Manuals

- Cannot see what object really looks like because many surfaces are hidden from view
  - >> Therefore we often add the isometric view too

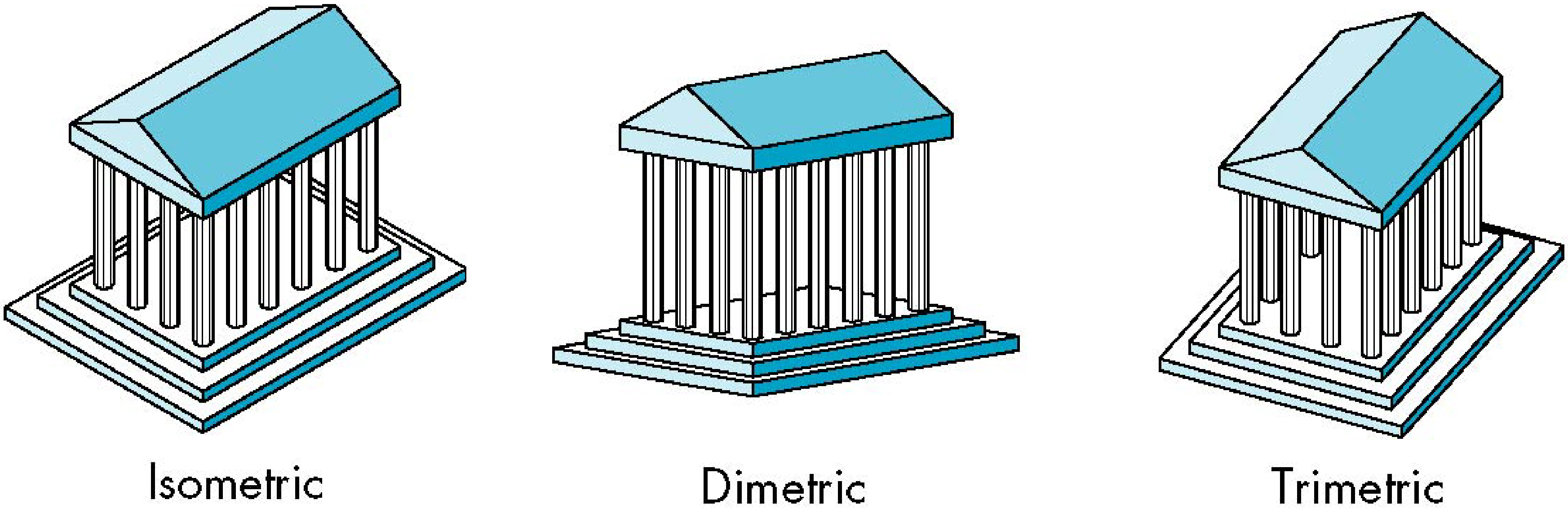
Axonometric Projections

Allow projection plane to move relative to object

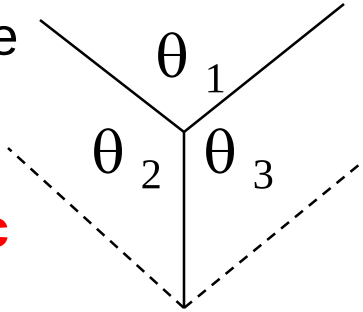
- **projectors are still orthogonal** to projection plane
- but projection plane can have any orientation with respect to the object



Types of Axonometric Projections



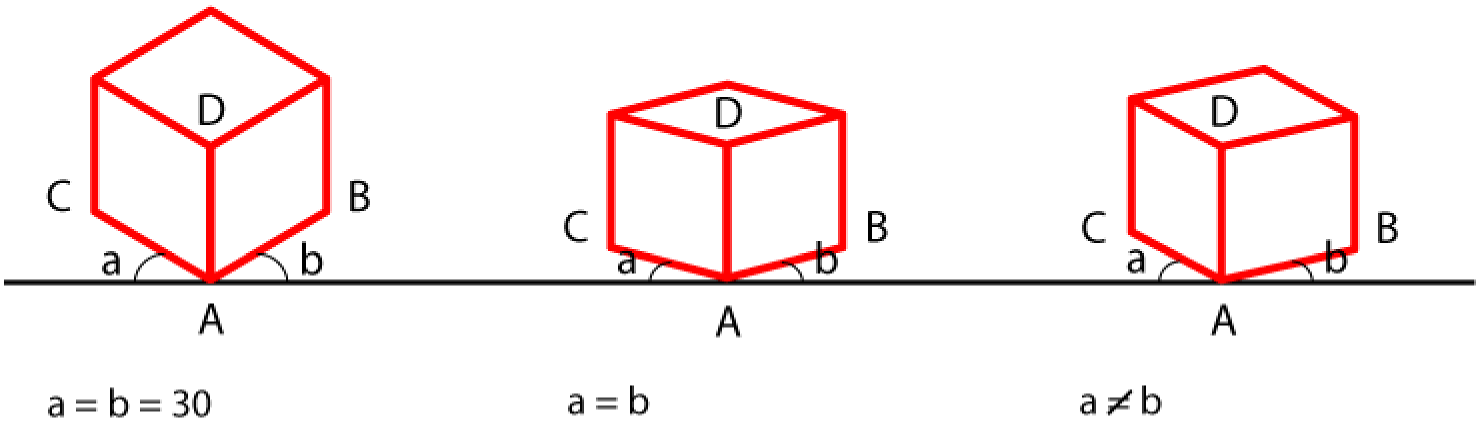
classified by how many angles of a corner of a projected cube are the same



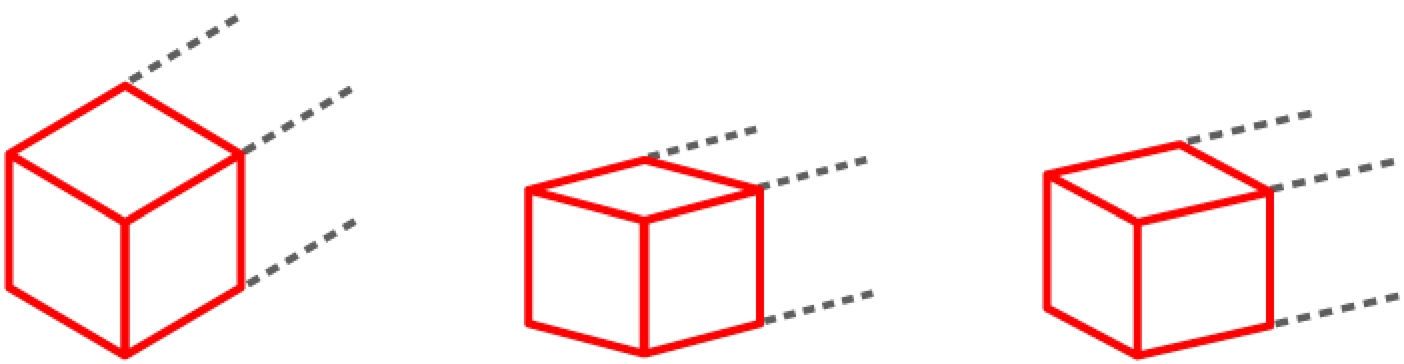
- all 3 same = 120 : **isometric**
- 2 of them same : **dimetric**
- all different : **trimetric**

If the projection plane is placed symmetrically
-wrt to all 3 principal faces that meet at a corner of the rectangular object: isometric.
-wrt to 2 principal faces: dimetric.
-The general case is a trimetric view.

Types of Axonometric Projections



Isometric Projection Dimetric Projection Trimetric Projection



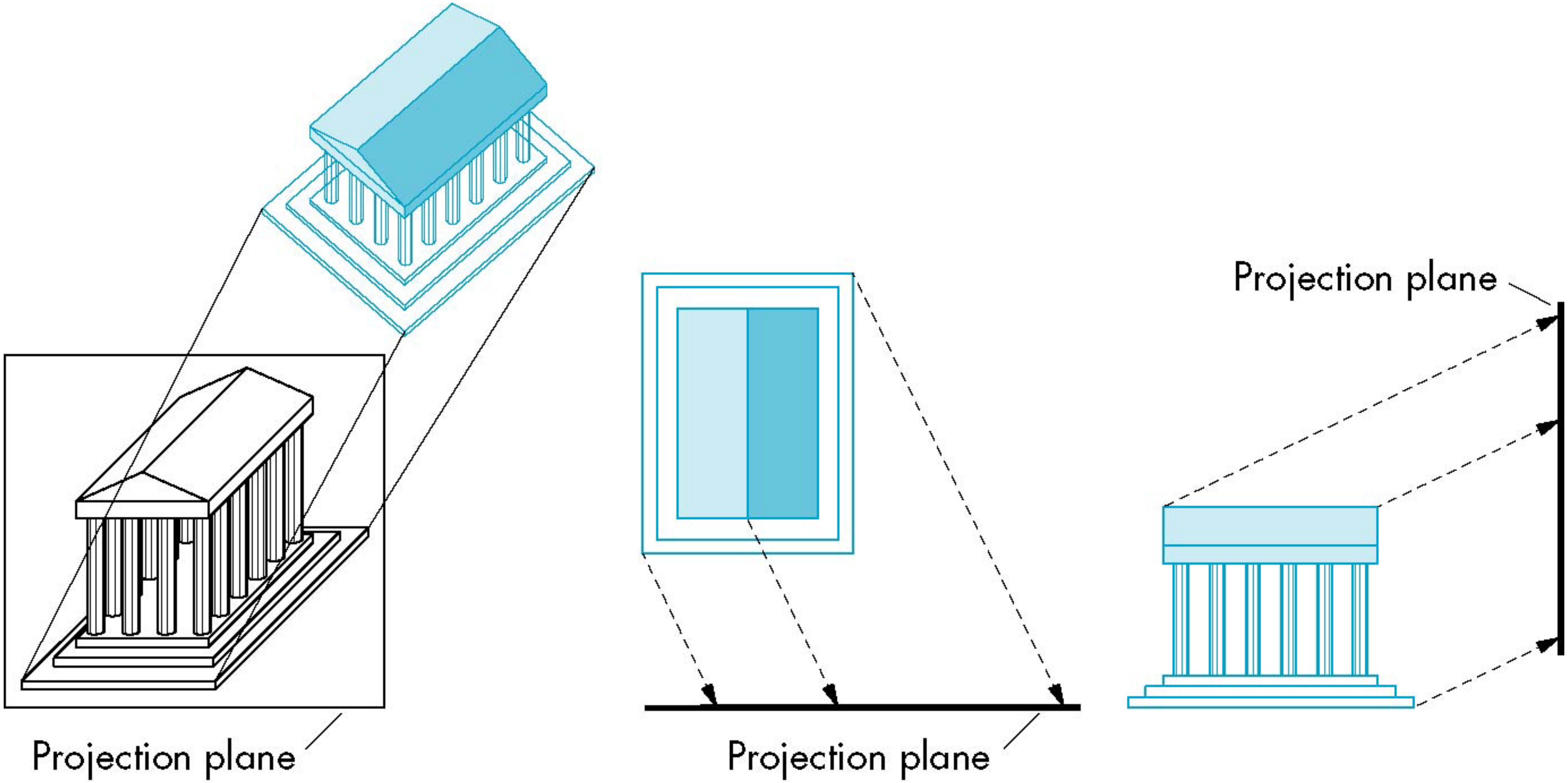
Isometric Projection Dimetric Projection Trimetric Projection

Advantages and Disadvantages of Axonometric Projections

- Lines are scaled (*foreshortened*)
  - but can find scaling factors since the factors do not change with distance
- Angles are not preserved
  - 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

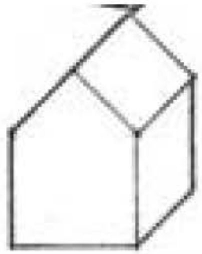
Oblique Projection

Arbitrary relationship between projectors and projection plane



Elevation Oblique

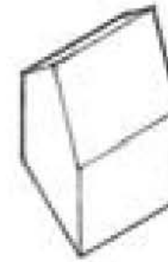
Principal vertical face of rectangular form is parallel with the picture plane.

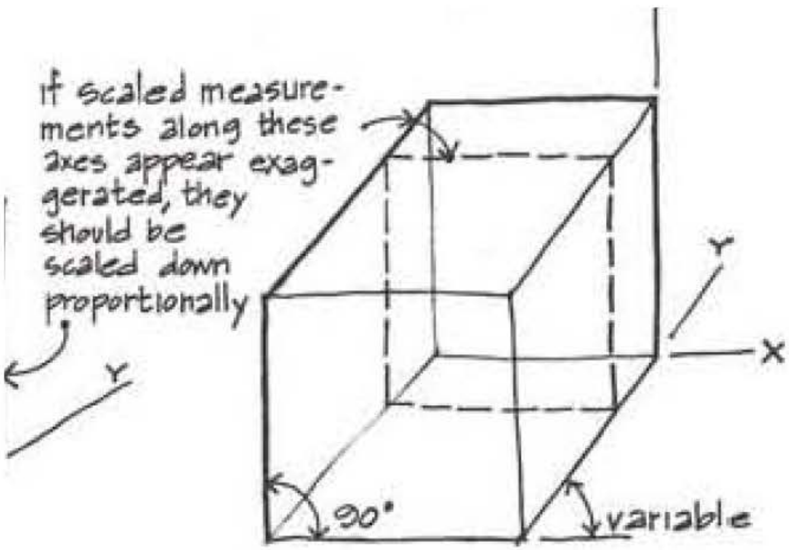


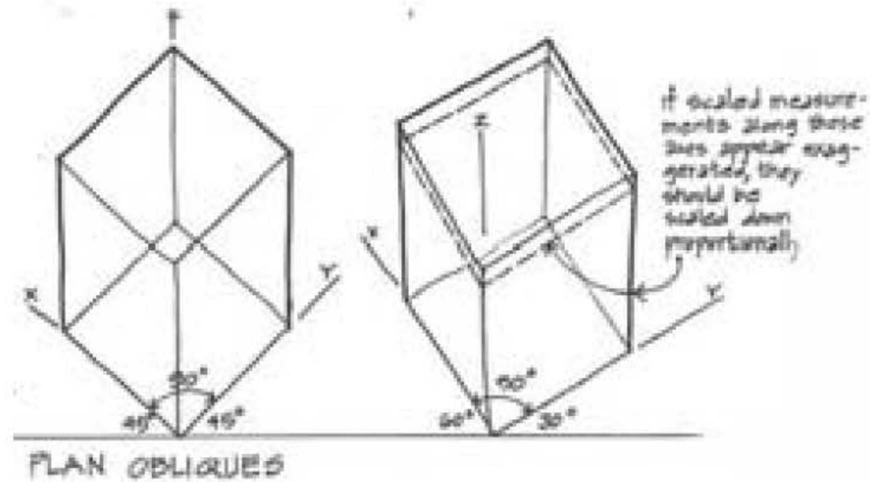
Source: Ching. Architectural graphics. 2003 page 115

Plan Oblique

Principal horizontal face of rectangular form is parallel with the picture plane.

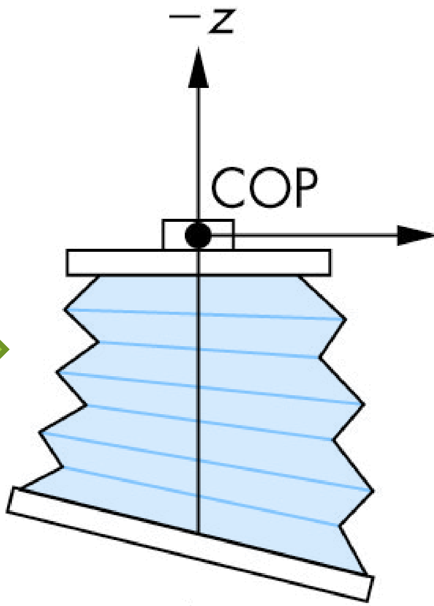






- “Unnatural” as physical viewing devices (e.g., eye, most cameras) have lens that is in a fixed relationship with the image plane (parallel)
  - Except with Bellows Cameras

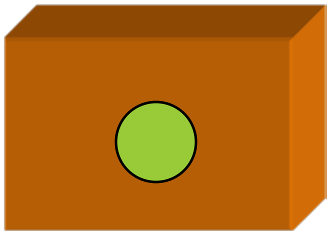
  - No problem to program it, though!



Advantages and Disadvantages of Oblique Projections

- Can pick the angles to emphasize a particular face
  - Architecture: plan oblique, elevation oblique

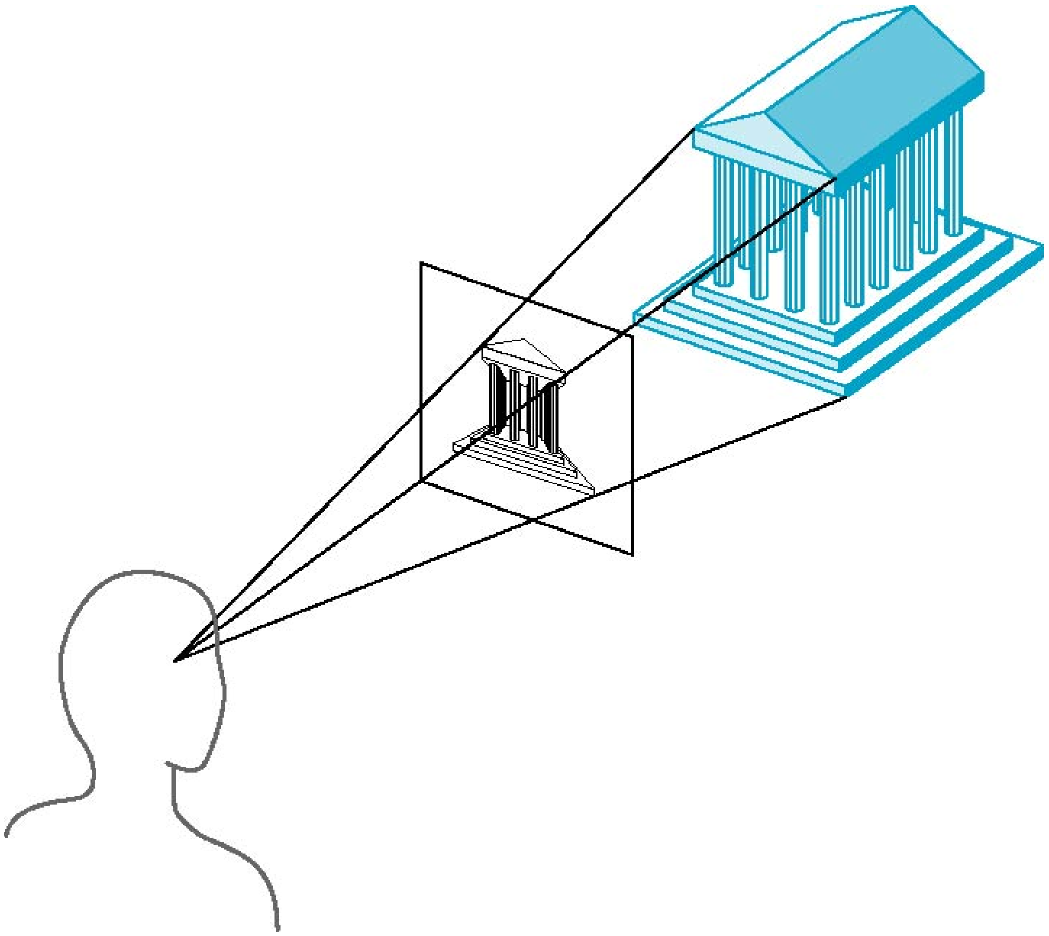
- Angles in faces parallel to projection plane are preserved while we can still see “around” side



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

Perspective Projection

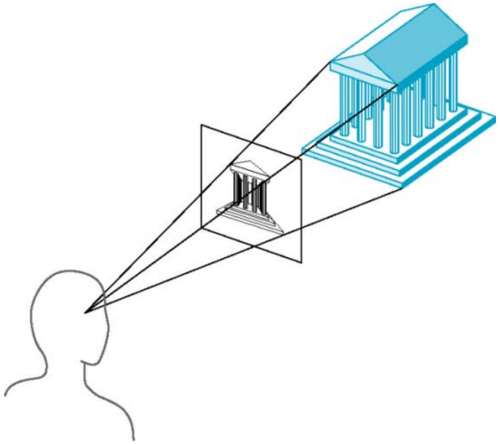
Projectors converge at center of projection



Perspective Viewing

- characterized by diminution of size
  - Projectors are not parallel
  - Leads to natural appearance
  - Amount by which line is foreshortened depends on how far from user

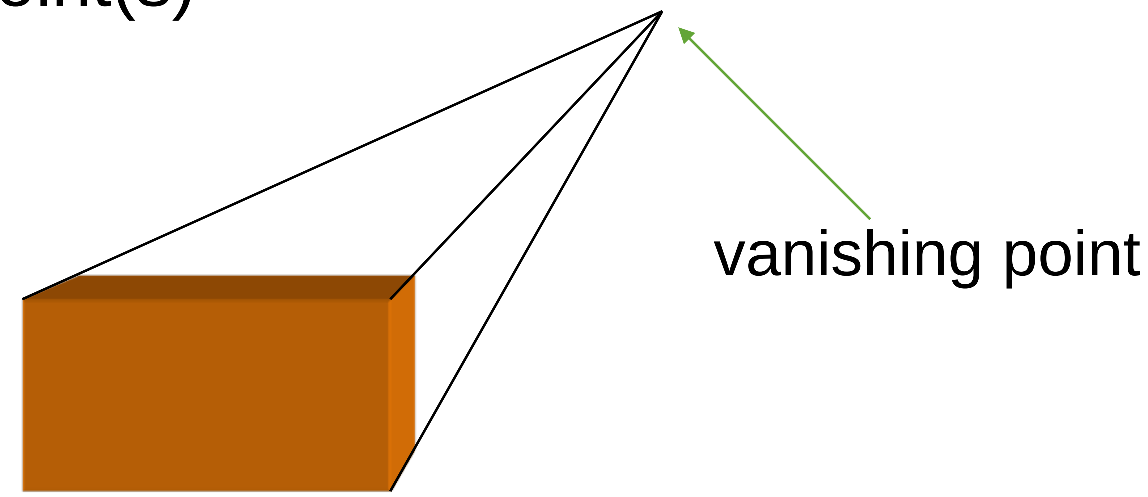
- In classical perspective views, viewer is located symmetrically wrt projection plane
  - Viewing pyramid (frustum) determined by window in projection plane and COP is a symmetric or right pyramid



- One-, two-, or three-point perspectives
  - How many of the three principal directions in object are parallel to the projection plane
  - How many vanishing points for projections

Vanishing Points

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

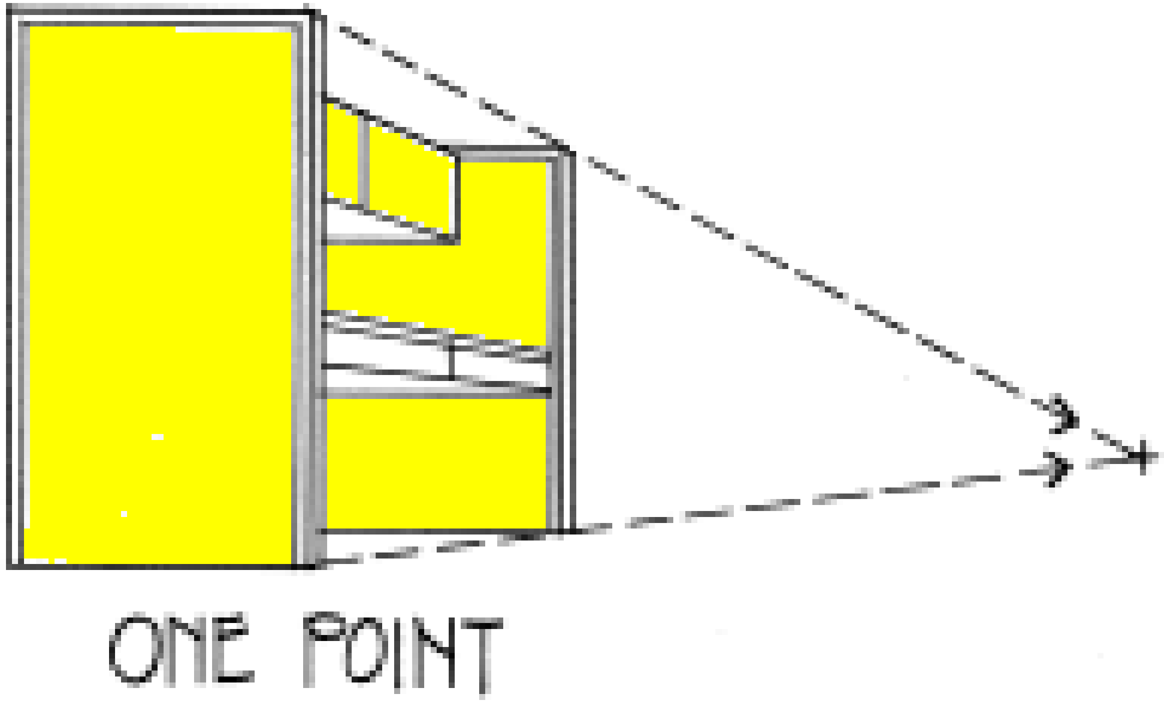


One-Point Perspective

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

One-Point Perspective

- receding lines or sides of an object appear to vanish to a single point on the horizon

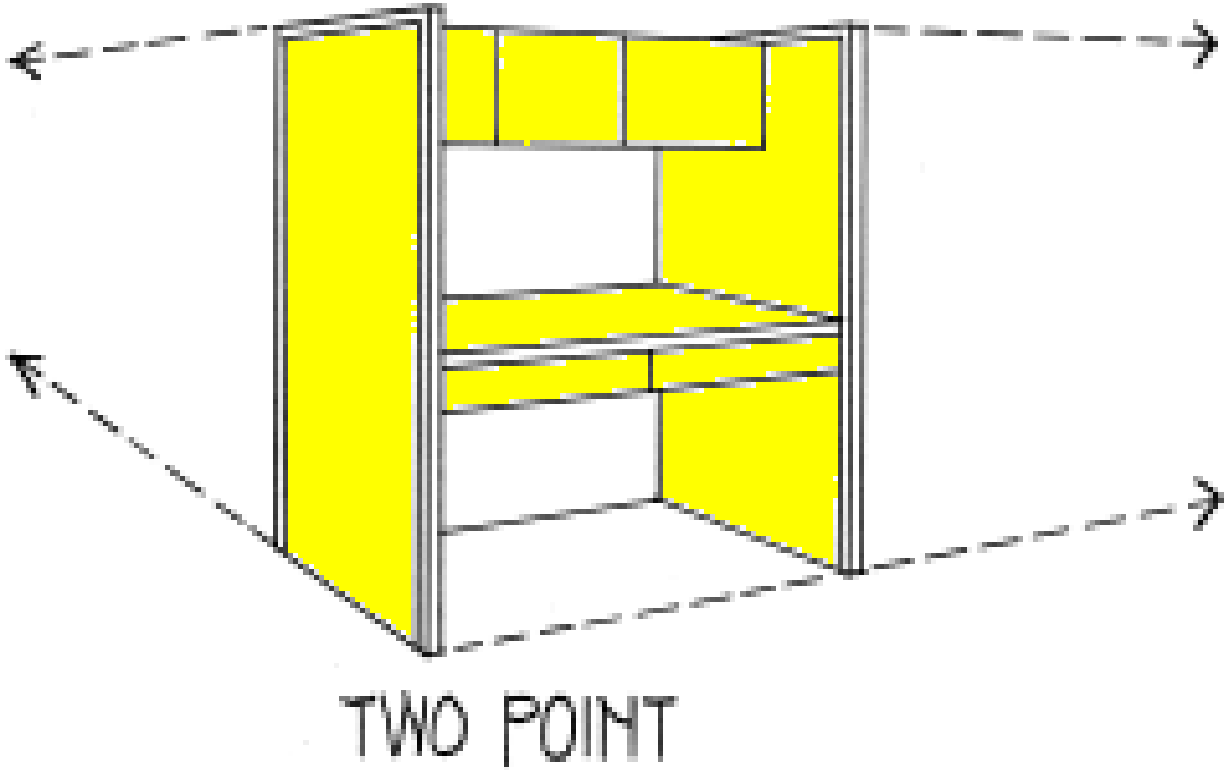


Two-Point Perspective

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

Two-Point Perspective

- The two-point perspective is one of the most widely used of the three types, as it **portrays the most realistic view for the observer**

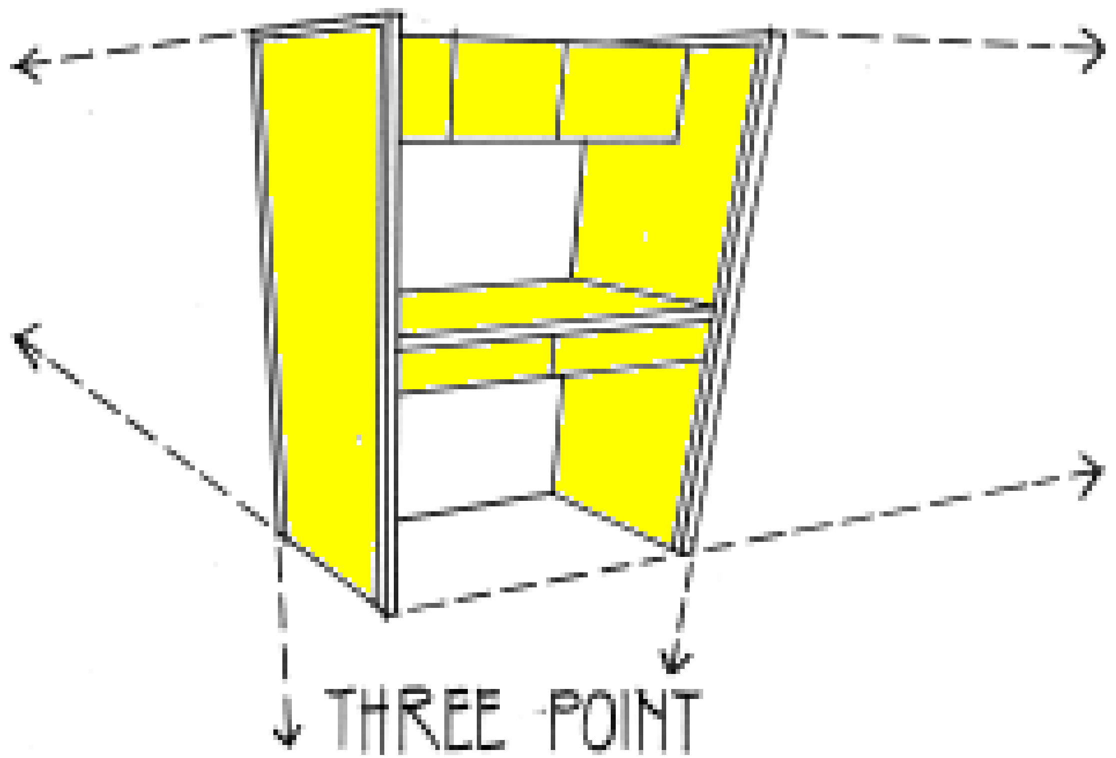


Three-Point Perspective

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

Three-Point Perspective

- Generally drawn with the viewer at a distance above the horizon (**bird's-eye view**) or below the horizon (**worm's-eye view**).
- Used mostly for very tall buildings and is rarely used in interior spaces, unless they are multistoried.
- More complicated than the former two types, as a third vanishing point is introduced, which precludes all parallel lines.



Advantages and Disadvantages of Perspective Projection

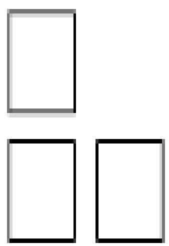
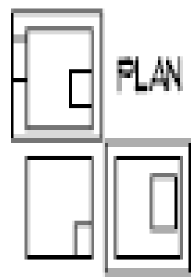
- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (***diminution***)
  - Looks realistic

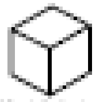
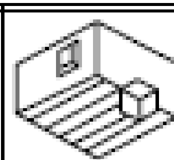
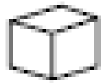
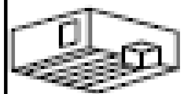
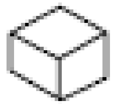
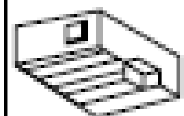
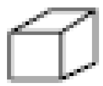
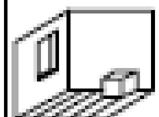
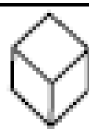
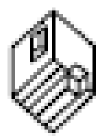
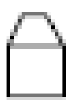
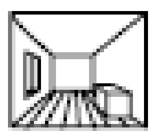
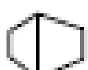
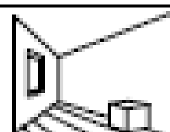
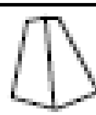
- Equal distances along a line are not projected into equal distances (***nonuniform foreshortening***)

- 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)

CLASSIFICATION OF DRAWING SYSTEMS

	TYPE	APPLICATION		RELATIONSHIP OF OBJECTS TO PICTURE PLANE
		OBJECT	INTERIORS	
MULTIVIEW	ORTHOGRAPHIC	 PLAN/ELEVATION	 PLAN ELEVAT'N SECT'N	AN OBJECT'S RECTANGULAR FACES ARE PARALLEL TO THE PICTURE PLANE.

SINGLEVIEW	PARALINE PARALLEL LINES REMAIN PARALLEL TO EACH OTHER	AXONOMETRIC	 ISOMETRIC		THE THREE PRINCIPAL AXES MAKE EQUAL ANGLES (30°) WITH THE PICTURE PLANE. ALL LENGTHS ARE EQUAL.
			 DIMETRIC		THE TWO PRINCIPAL AXES MAKE EQUAL ANGLES WITH THE PICTURE PLANE, AND TWO LENGTHS ARE EQUAL. OBJECTS CAN BE ROTATED AT VARIOUS ANGLES.
			 TRIMETRIC		EACH OF THE TWO PRINCIPAL AXES MAKES A DIFFERENT ANGLE WITH THE PICTURE PLANE. HEIGHT IS REDUCED, SIMILAR TO A DIAMETRIC.
		OBLIQUE	 ELEVATION		THE FACE (ELEVATION) OF THE OBJECT IS PARALLEL TO THE PICTURE PLANE. DEPTHS ARE USUALLY REDUCED IN RATIO.
			 PLAN		THE TOP VIEW (OR PLAN) OF THE OBJECT IS PARALLEL TO THE PICTURE PLANE. HEIGHTS ARE USUALLY REDUCED.
	PERSPECTIVE PARALLEL LINES APPEAR TO CONVERGE TO VANISHING POINTS		 ONE-POINT		ONE FACE IS PARALLEL TO THE PICTURE PLANE. PROJECTOR LINES CONVERGE TO ONE POINT.
			 TWO-POINT		VERTICAL FACES ARE AT AN ANGLE TO THE PICTURE PLANE. PROJECTOR LINES CONVERGE TO TWO POINTS.
			 THREE-POINT		VERTICAL FACES ARE AT AN ANGLE TO THE PICTURE PLANE. PROJECTOR LINES CONVERGE TO THREE POINTS.

COMPUTER VIEWING

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Objectives

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

Computer Viewing

- There are 3 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

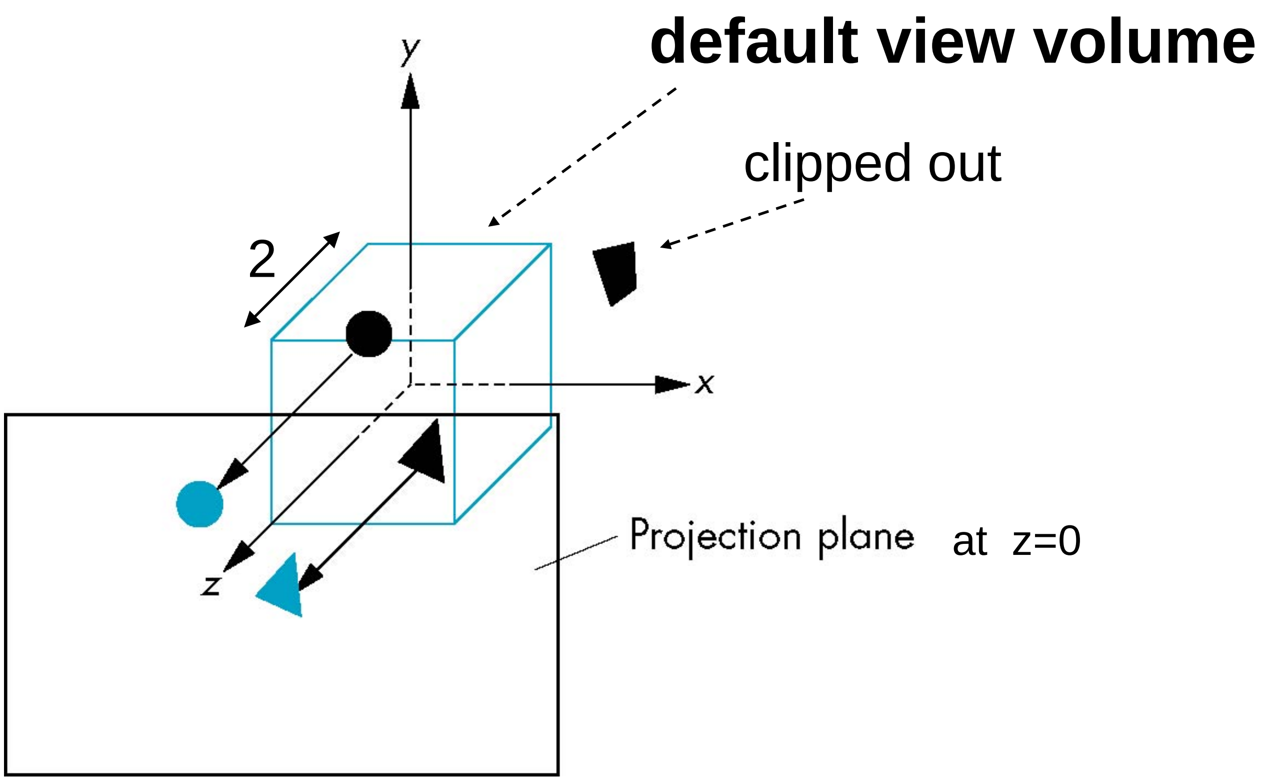
- In OpenGL, initially the object and camera frames are the same
  - **Default model-view matrix is an identity "I" matrix**

- The camera is **located at origin** and **points in the <u>negative</u> z direction**

- OpenGL also specifies **a <u>default view volume</u> that is a cube with sides of length 2 centered at the origin**
  - **Default projection matrix is also an identity "I" matrix**

Default Projection

Default projection is <u>orthogonal</u>



Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either

  - Make a **view matrix**: to move the camera in the positive z direction
    - Translate the camera frame

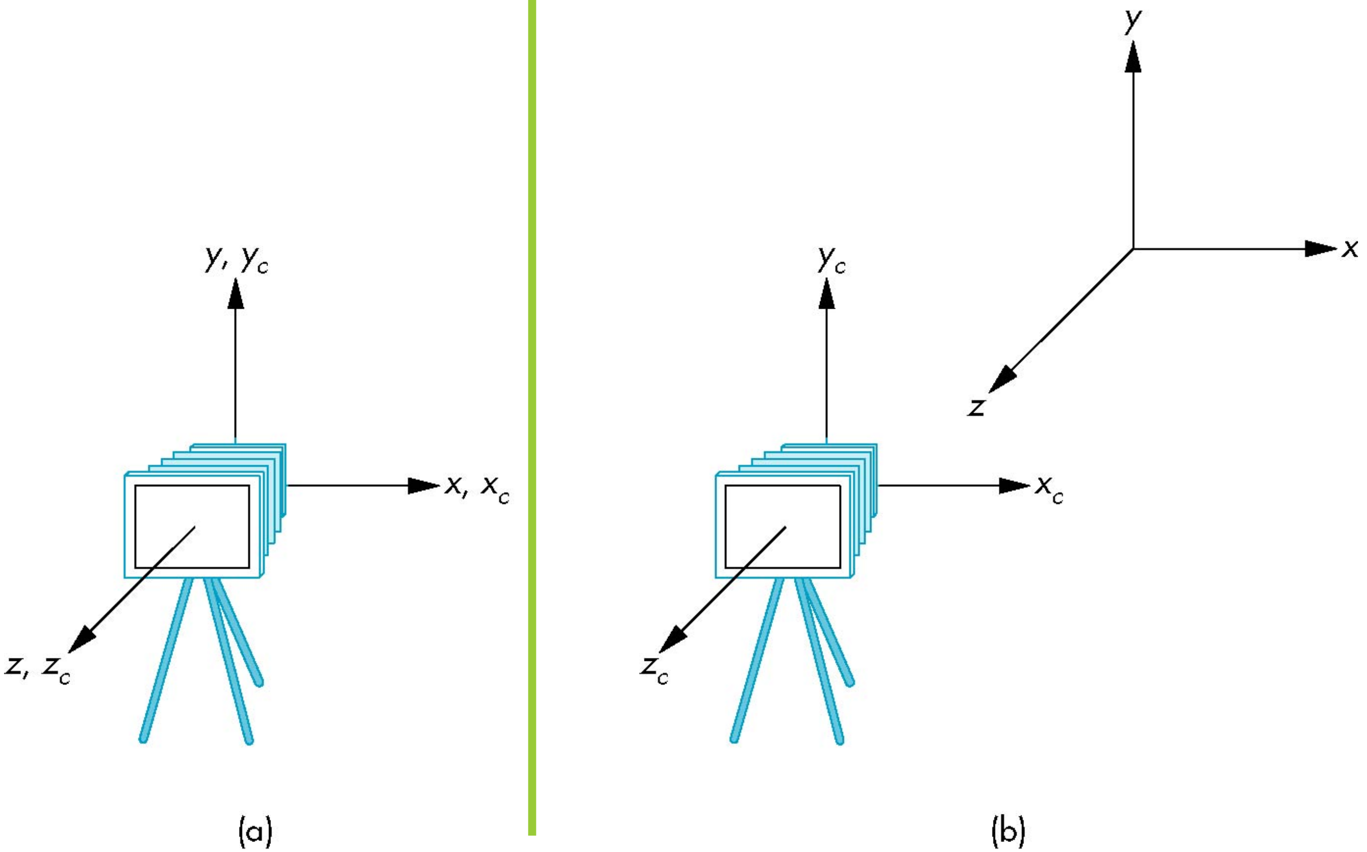
  - Make a **world matrix**: to move the objects in the negative z direction
    - Translate the world frame

- **Both of these views are equivalent and are determined by the same model-view matrix**
  - Want a translation: `Translate(0.0,0.0,-d)`
    - Where `d > 0`

Moving Camera back from Origin

frames after translation by –d

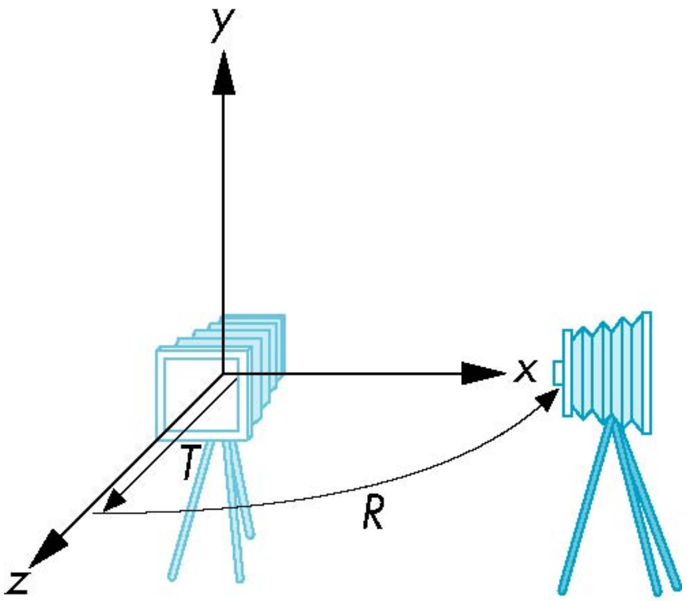
d > 0



Moving the Camera

- We can move the camera to any desired position by a <u>sequence of rotations and translations</u>
- Example: side view
  - Move it away from origin: T
  - Rotate the camera about origin: R
  - Model-view matrix M = RT

  so that Mv = RTv



The LookAt Function

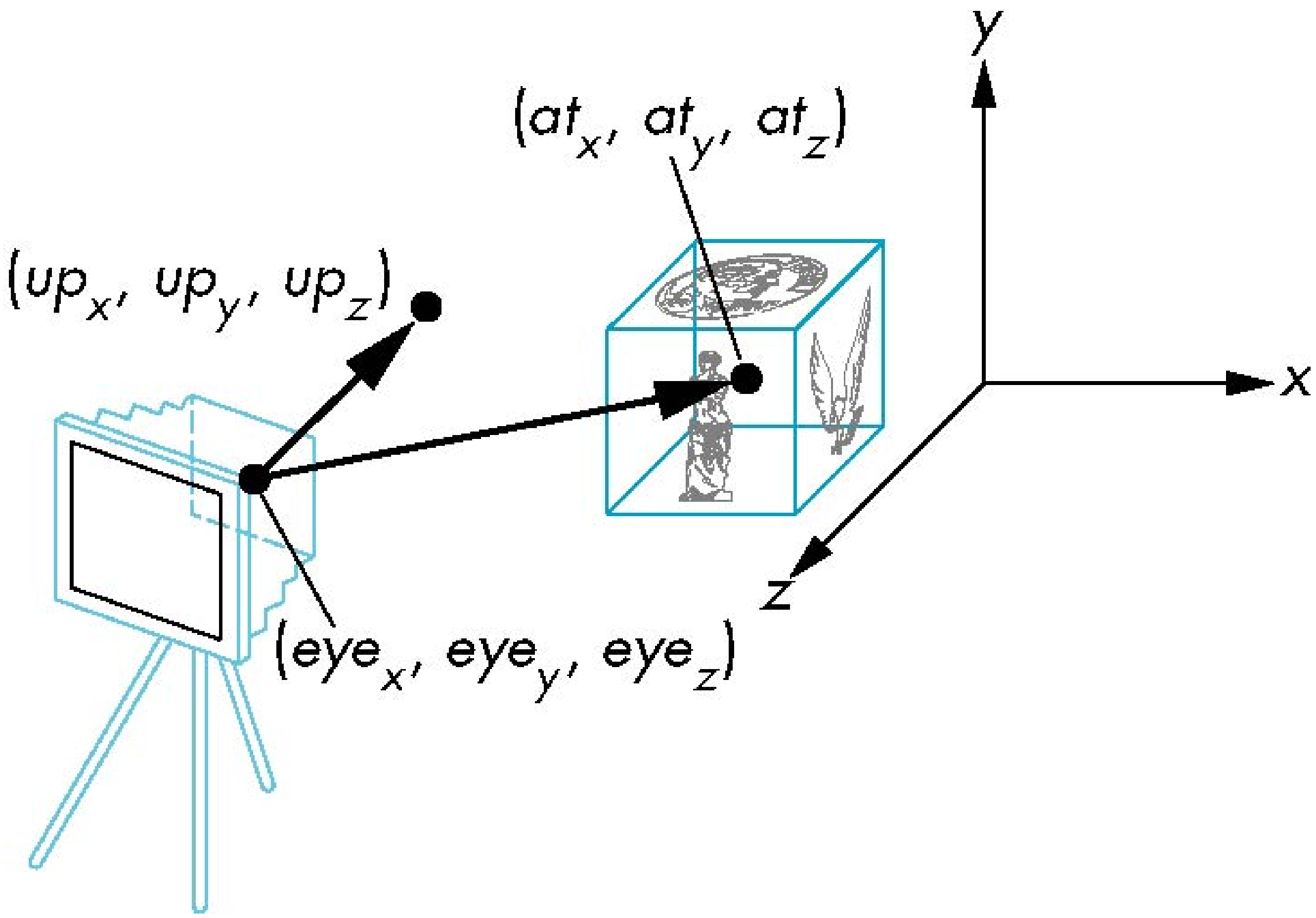
- The GLU library contained the function gluLookAt to form the required modelview matrix through a simple interface
- Replaced by **LookAt()** in mat.h
  - Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```
mat4 mv = LookAt(vec4 eye, vec4 at, vec4 up);
```

- Note the need for setting an **up direction**

gluLookAt

`LookAt(eye, at, up)`



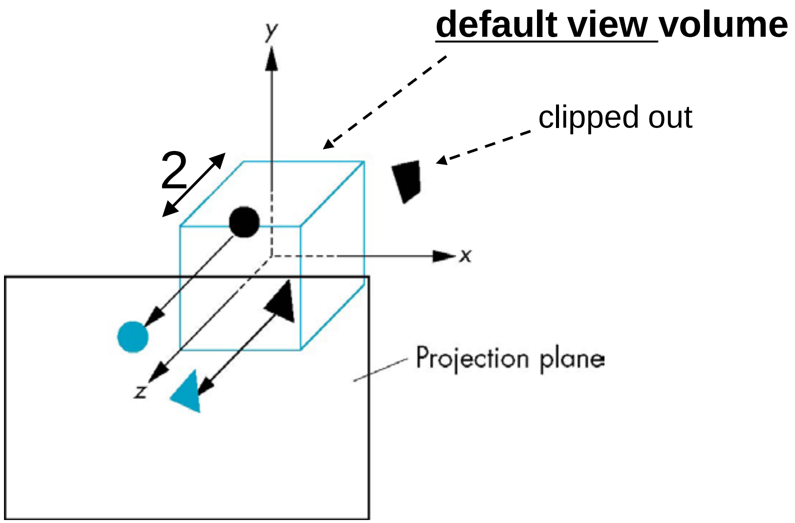
Other Viewing APIs

- 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

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:

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

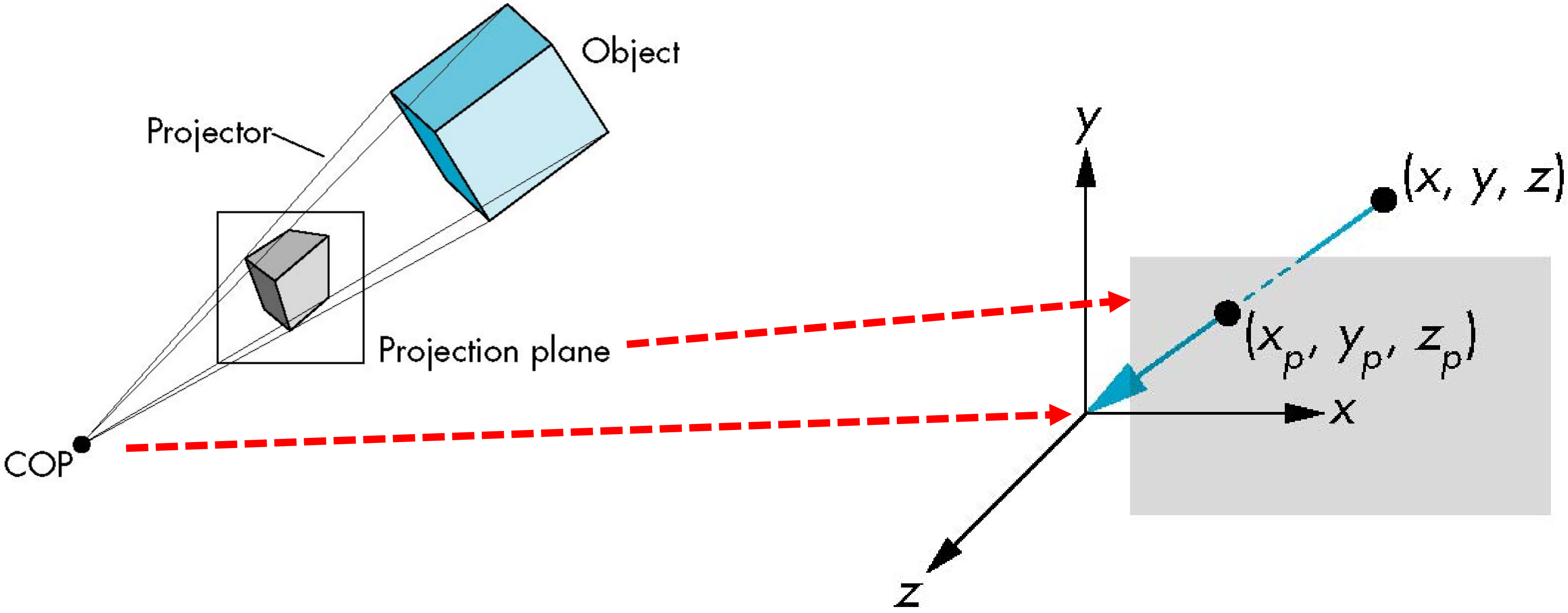
$$\mathbf{p}_p = \mathbf{Mp}$$

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

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

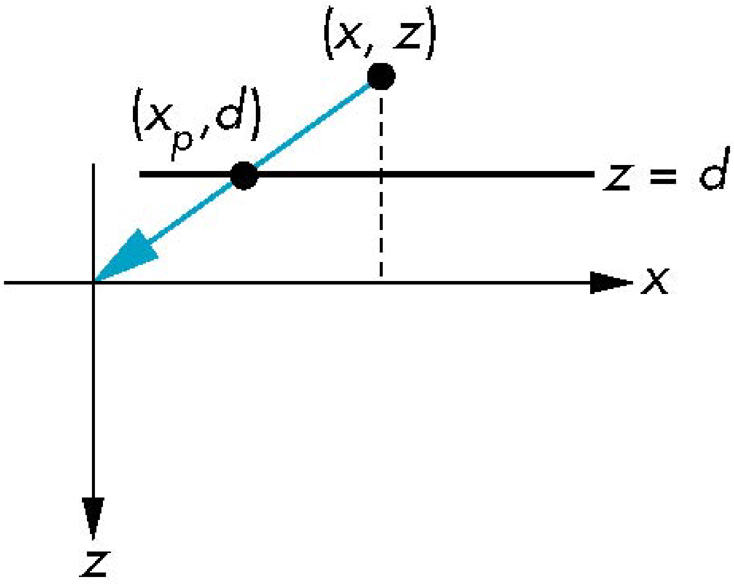
Simple Perspective

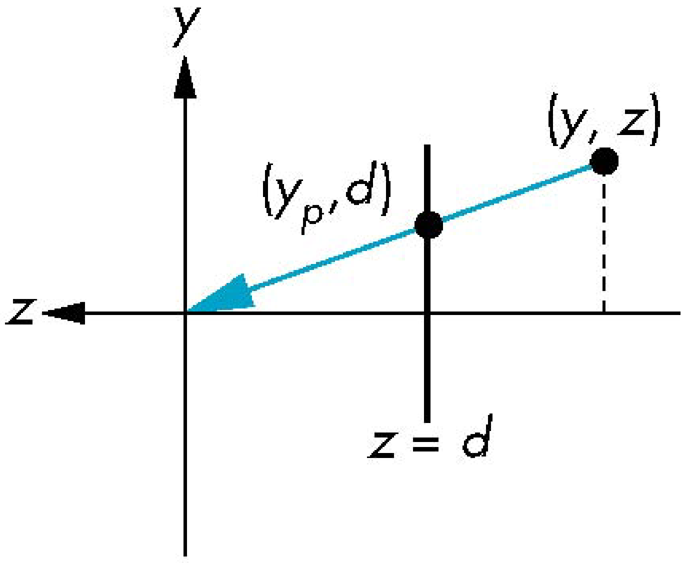
- Center of projection at the origin
- Projection plane at $z = d\,,\quad \text{where}\;\; d < 0$



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

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

Perspective Division

- However, in order to make $w = 1$,

we must divide by *w* to return from homogeneous coordinates

>> This ***perspective division*** yields:

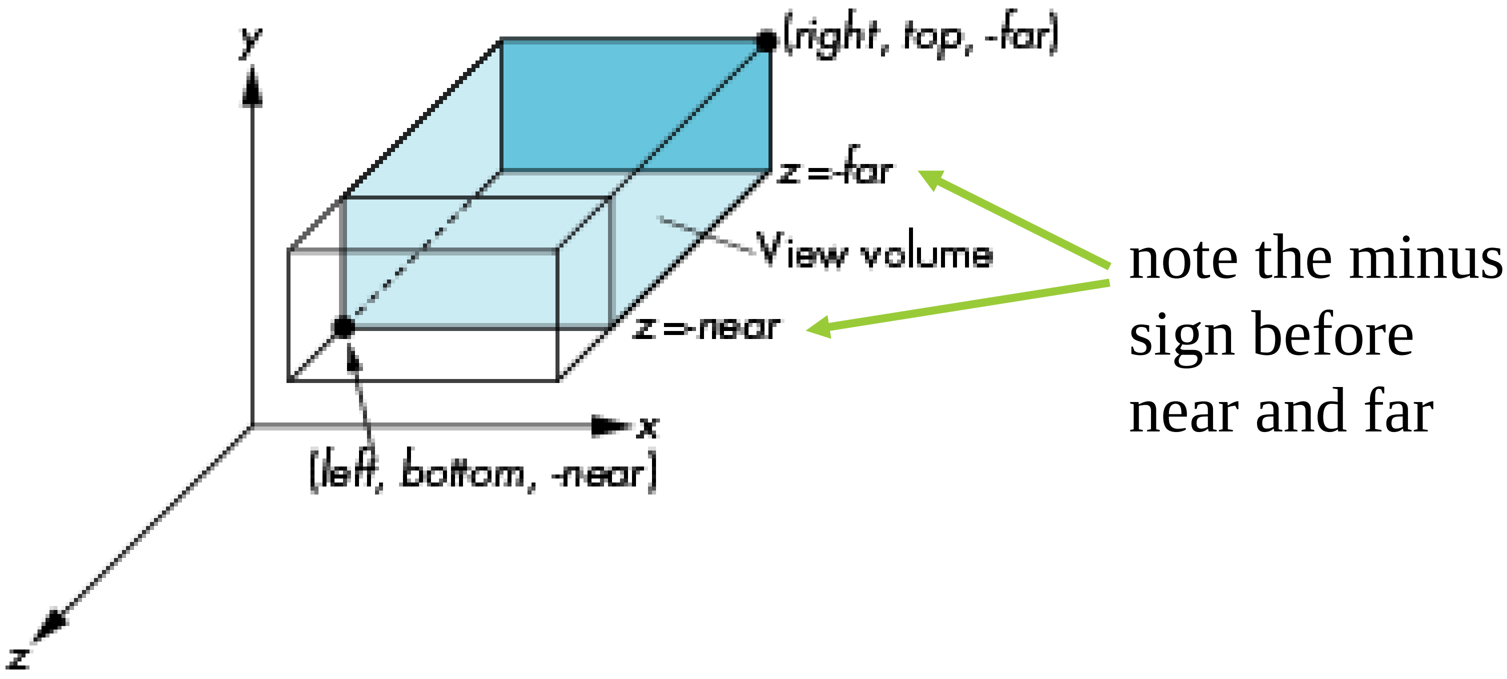
$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

the desired perspective equations.

- We will consider the corresponding clipping volume with mat.h functions

Orthogonal Viewing

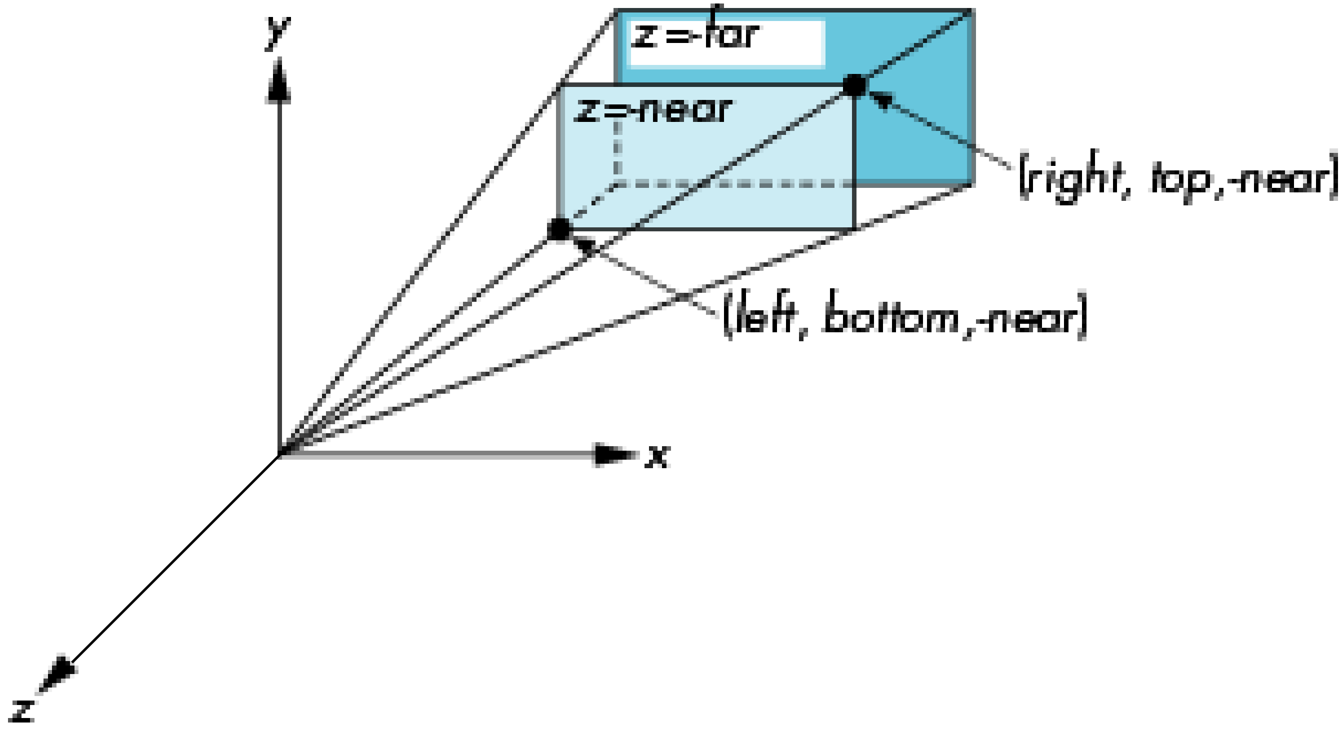
`Ortho(left,right,bottom,top,near,far)`



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

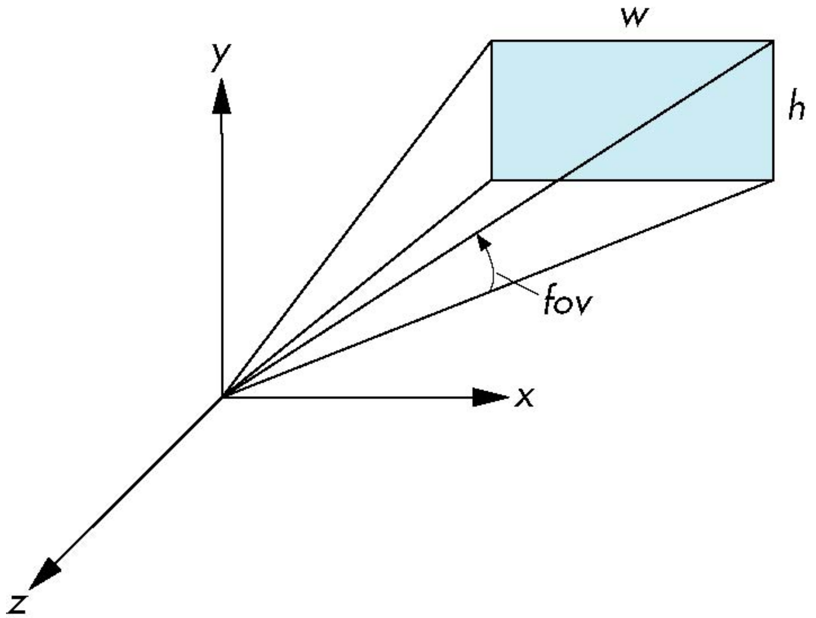
Perspective

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



Using Field of View

- With **`Frustum`** it is often difficult to get the desired view
- **`Perpective(fovy, aspect, near, far)`** often provides a better way



front plane

`aspect = w/h`

PROJECTION MATRICES

Lecturer: Asst. Prof. Ufuk Çelikcan

Based on the slides by: E. Angel and D. Shreiner

Objectives

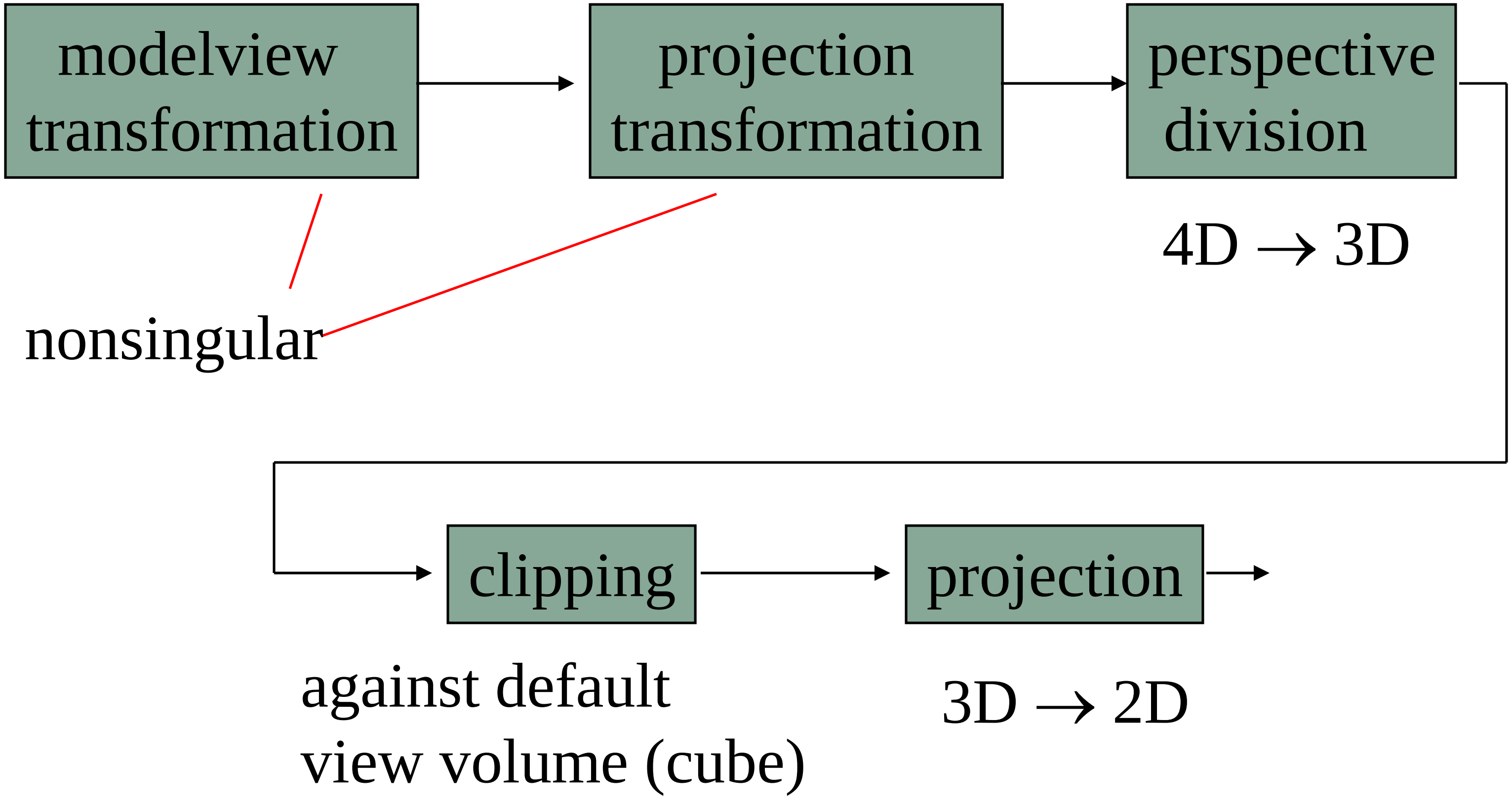
- Derive the projection matrices used for standard OpenGL projections
- Introduce oblique projections
- Introduce projection normalization

Normalization

- Rather than deriving a different projection matrix for each type of projection, we can normalization at the projection transformation stage so that we can simply use orthogonal projections within the default view volume.

- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Pipeline View



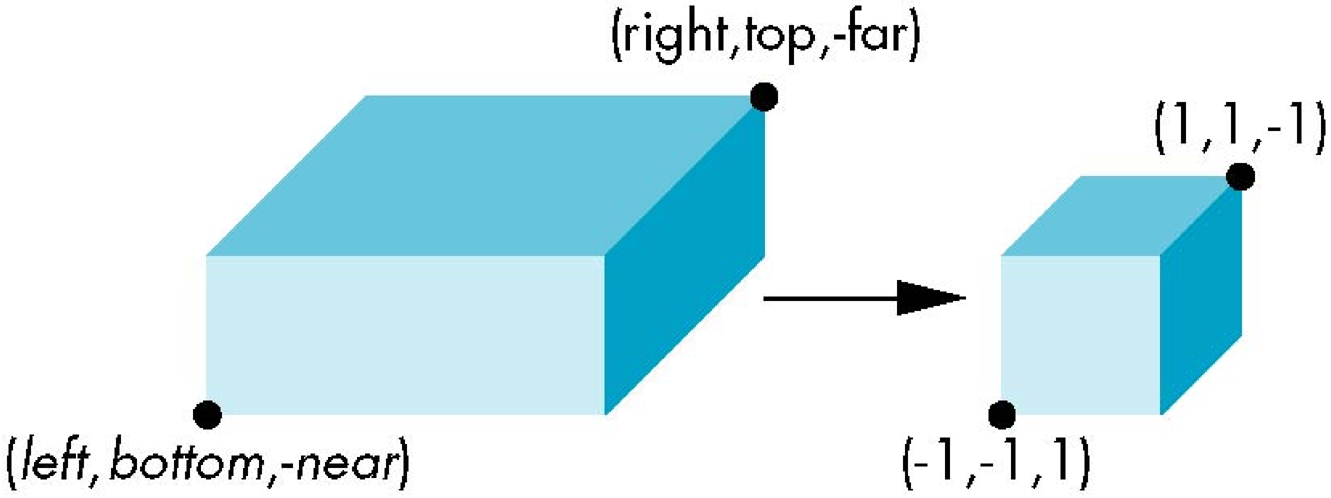
Notes

- We stay in four-dimensional homogeneous coordinates through both the model-view 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 the default cube



Orthogonal Matrix

- 2 steps
  1. Move center to origin

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

  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**

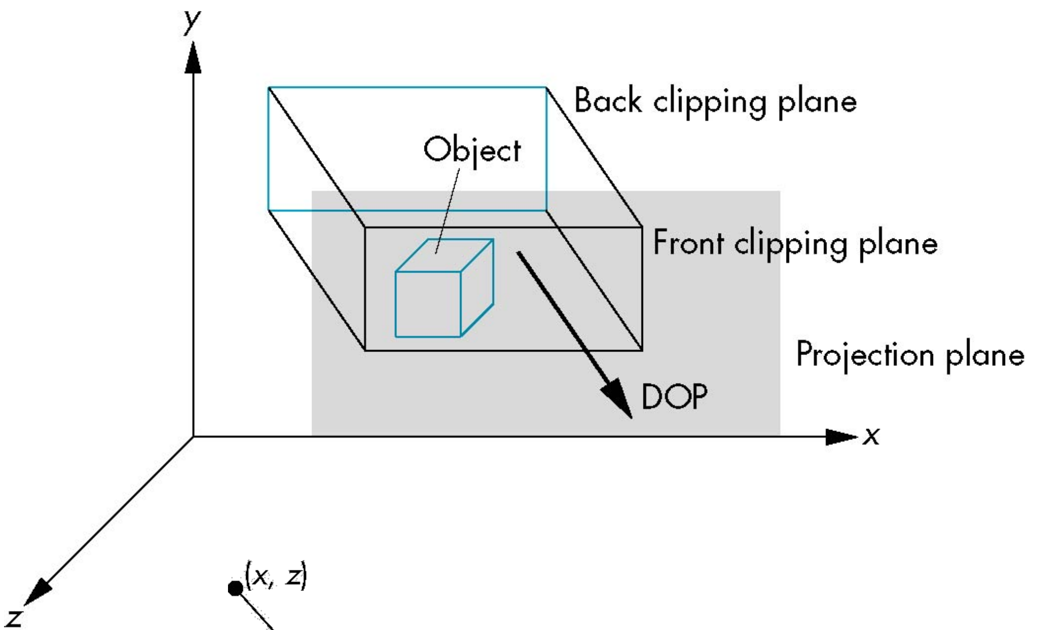
$$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{S}\mathbf{T}$$

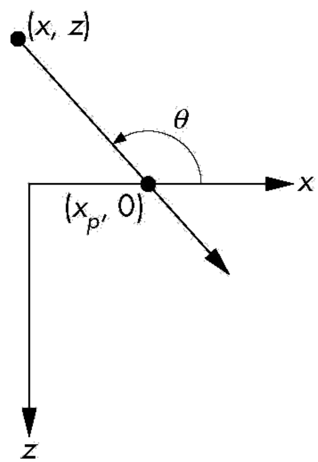
Oblique Projections

- if we look at the example of the cube, it appears as if the cube has been sheared

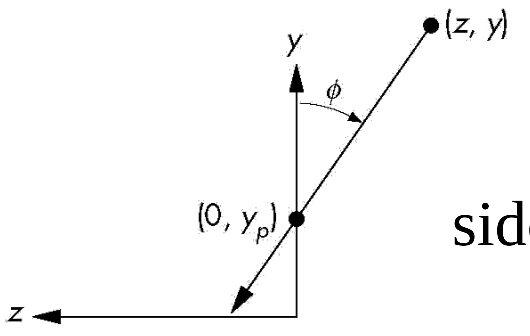
- Oblique Projection = **Shear + Orthogonal Projection**

General Shear





top view



side view

Shear Matrix

xy shear (z values unchanged)

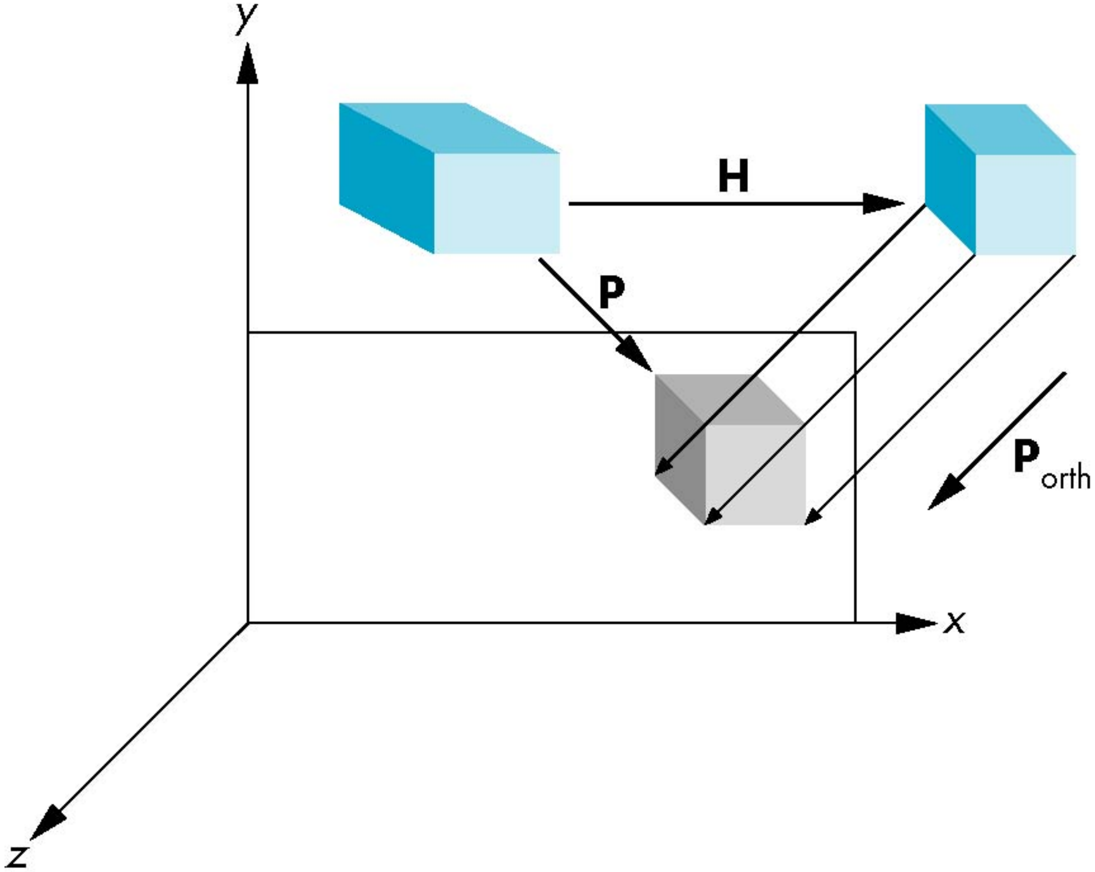
$$\mathbf{H}(\theta,\phi) = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\varphi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection matrix

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

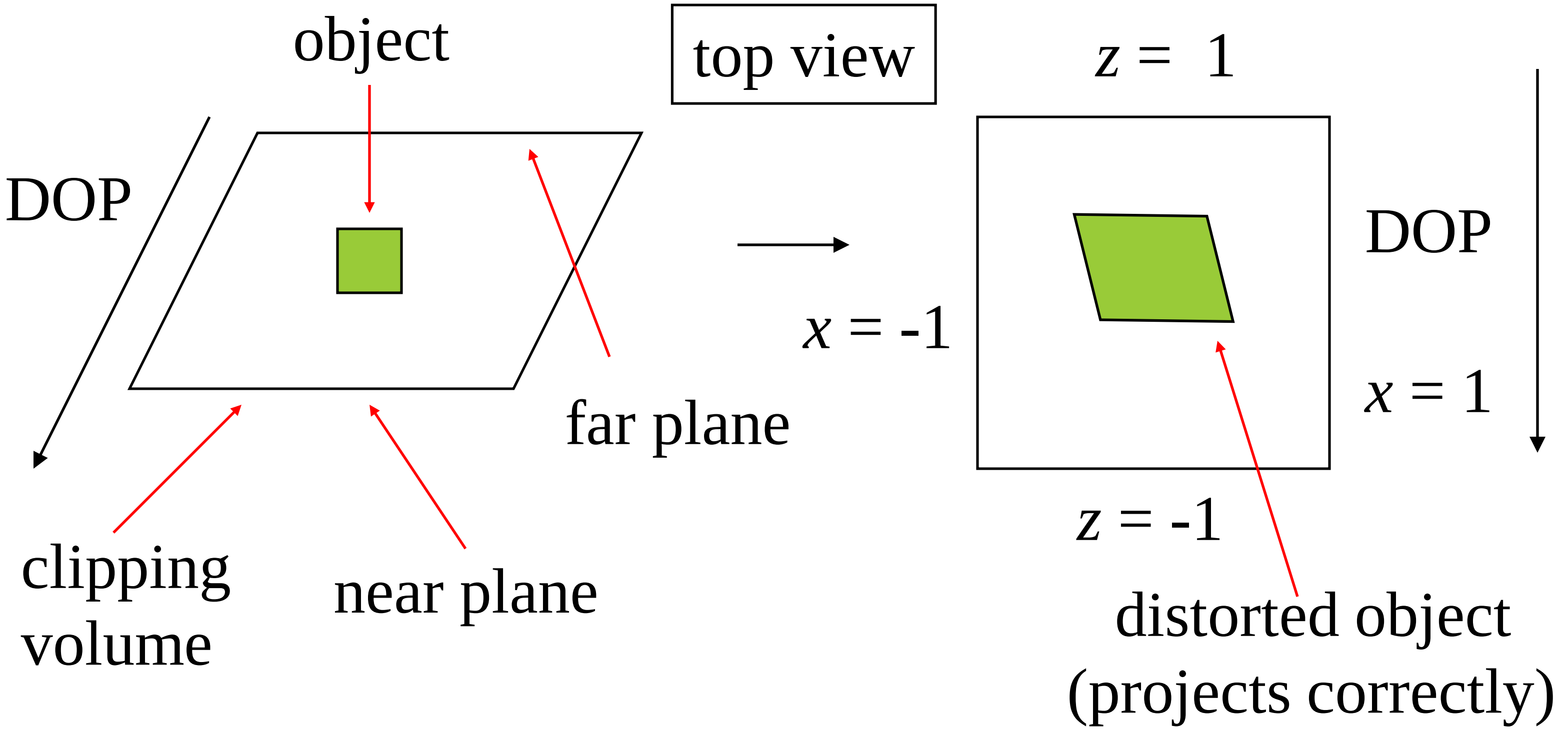
with normalization: $\mathbf{P} = \mathbf{M}_{\text{orth}}\,\mathbf{S}\,\mathbf{T}\,\mathbf{H}(\theta,\phi)$

Equivalency



Effect on Clipping

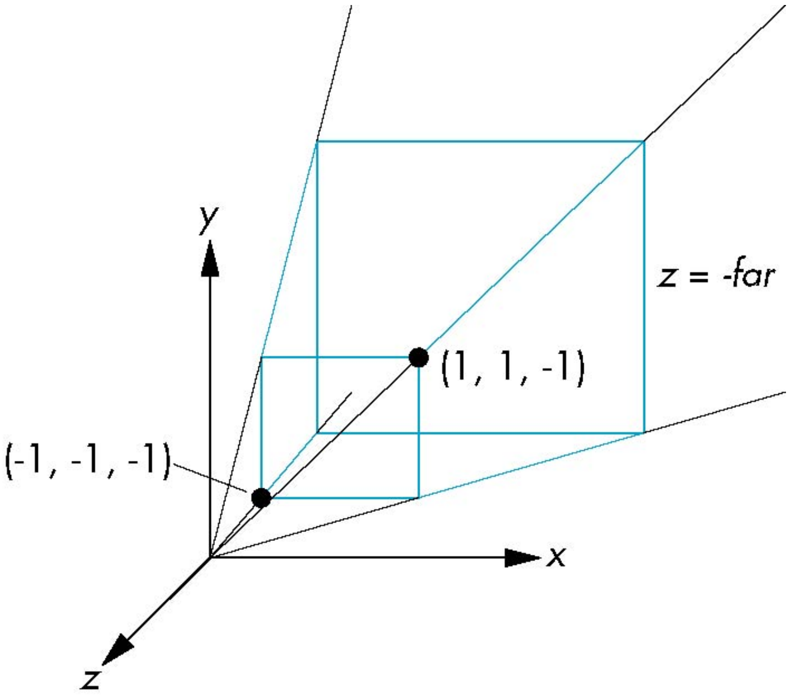
- The projection matrix $\mathbf{P} = \mathbf{STH}$ transforms the original clipping volume to the default clipping volume



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

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



Perspective Matrices

Resulting 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

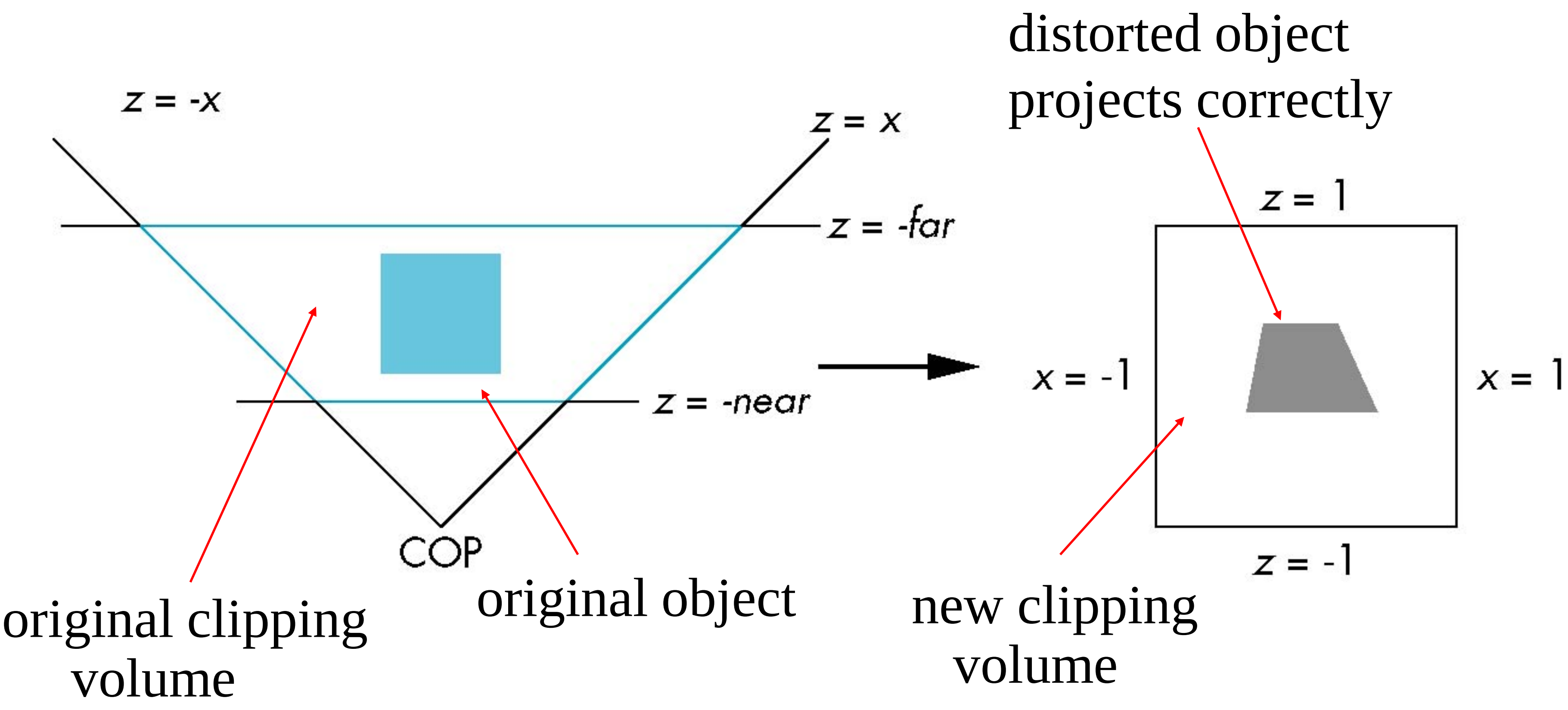
$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

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

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

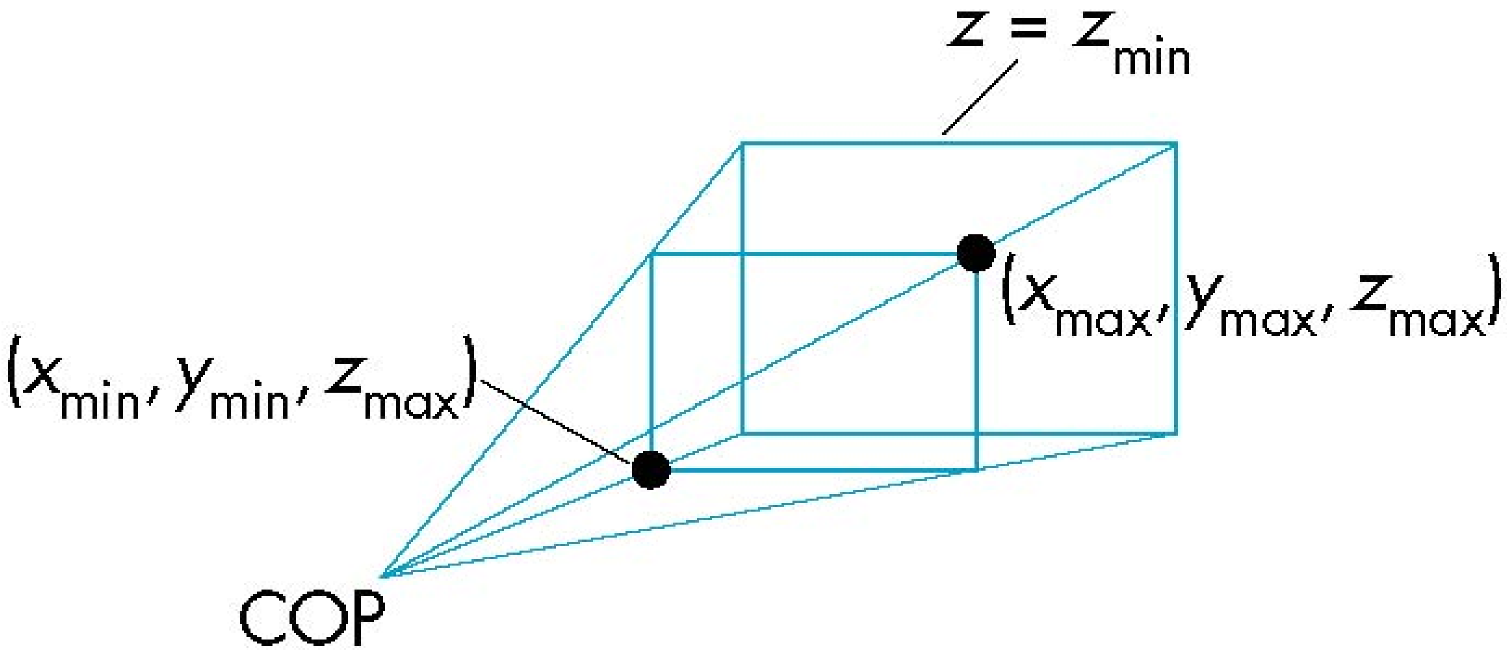


Normalization and Hidden-Surface Removal

- 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 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

- **`Frustum`** allows for an unsymmetric viewing frustum (although **`Perspective`** does not)



OpenGL Perspective Matrix

- The normalization in **`Frustum`** 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

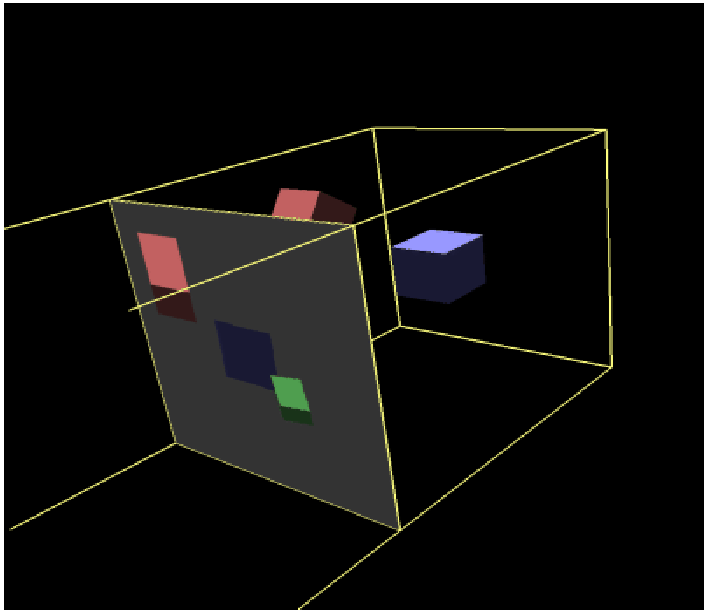
shear and scale

Why do we do it this way?

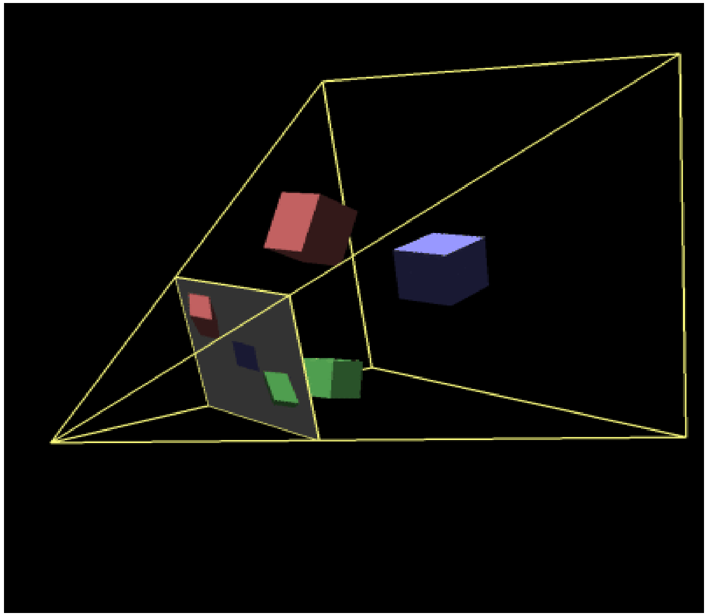
- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping

Specifying What You Can See

Orthographic View



Perspective View



Classification of Transforms

Translation

Rotation

Rigid Body preserves angles and distances

Uniform Scaling

Similarity preserves angles

Non-Uniform Scaling

Shear

Reflection

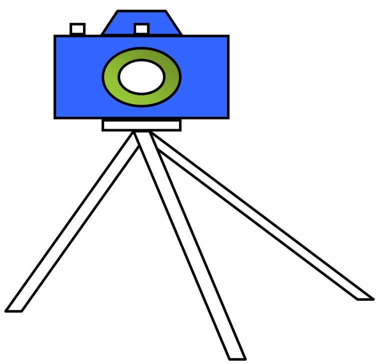
Affine preserves parallel lines

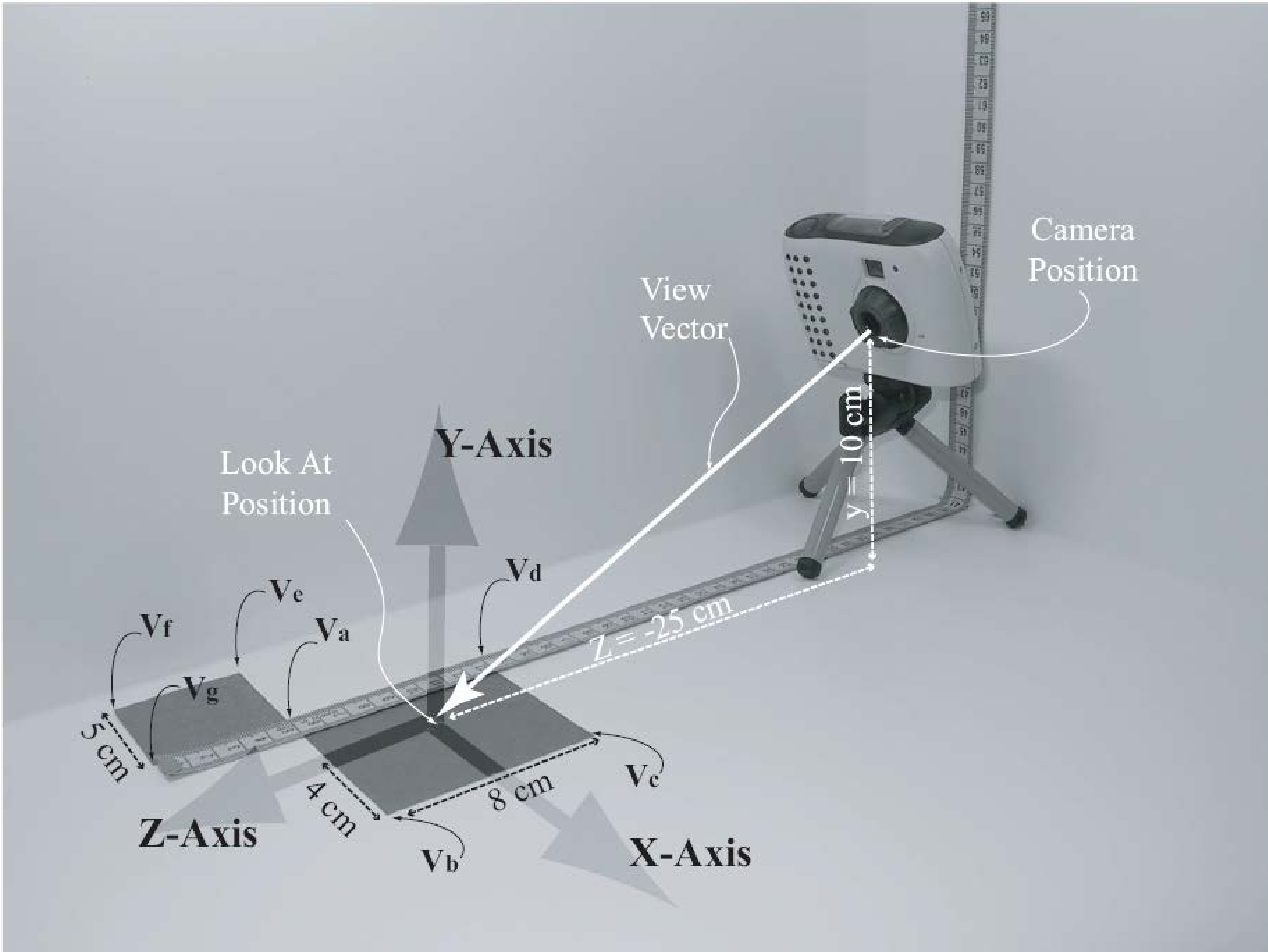
Perspective

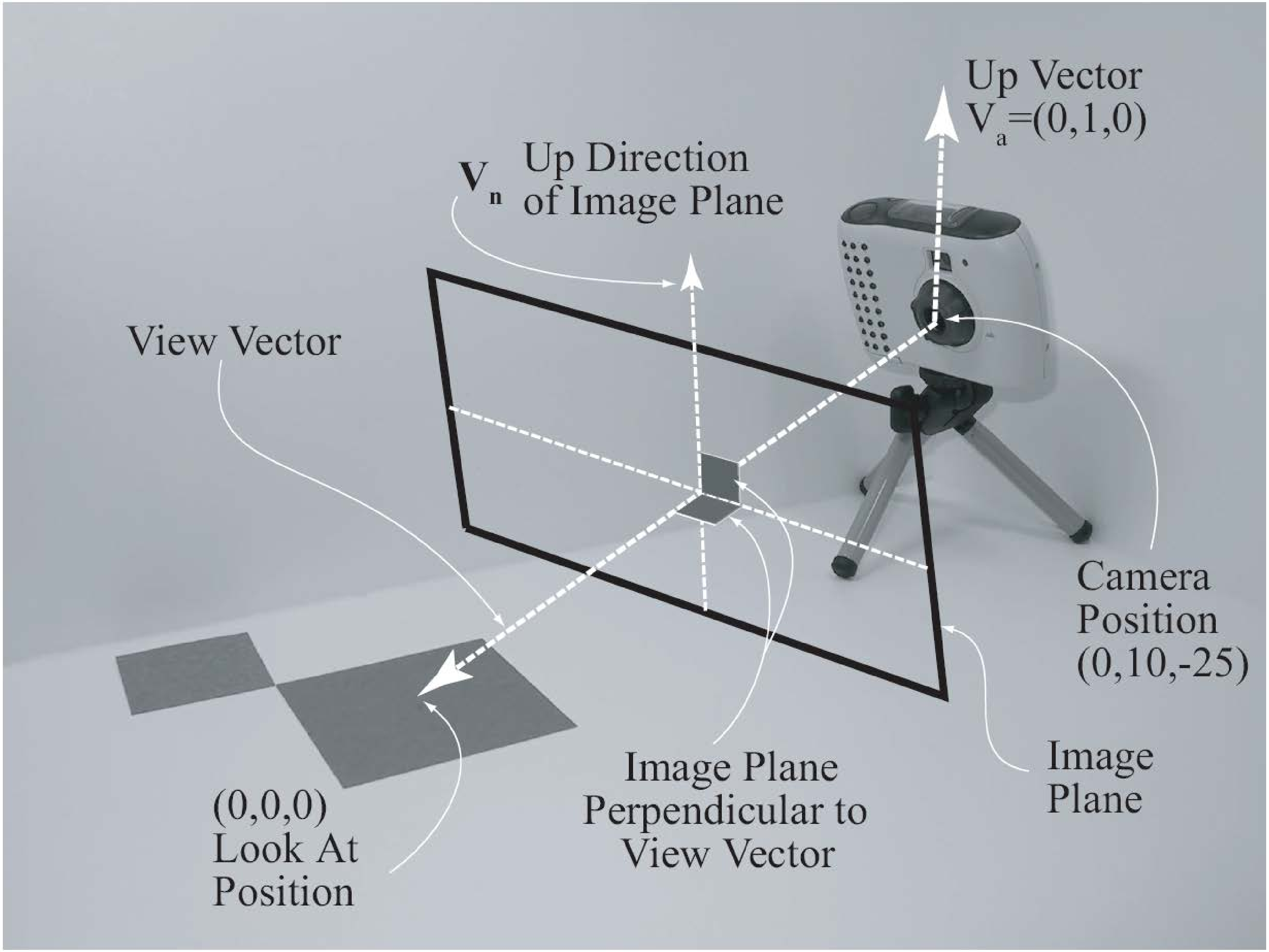
Projective preserves lines

Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene
- `LookAt( eyex, eyey, eyez,`
  `lookx, looky, lookz,`
  `upx, upy, upz )`
  - up vector determines unique orientation
  - careful of degenerate positions where the generated viewing matrix is undefined.




Camera
Position
View
Vector
Y-Axis
Look At
Position
y = 10 cm
Z = -25 cm
Vd
Ve
Vf
Va
Vg
5 cm
4 cm
8 cm
Vc
Vb
Z-Axis
X-Axis

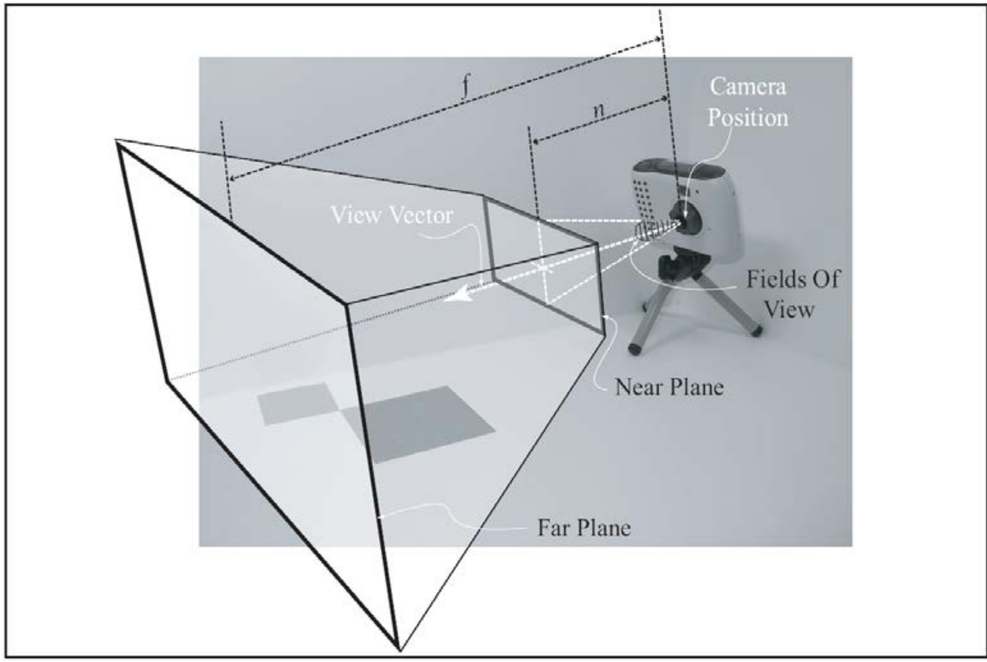

Up Vector
$V_a=(0,1,0)$
$\mathbf{V_n}$ Up Direction of Image Plane
View Vector
Camera Position (0,10,-25)
Image Plane
Image Plane Perpendicular to View Vector
(0,0,0) Look At Position

The Visible Volume

- Only geometries (primitives) *inside* the volume are visible
- All geometries (primitives) outside are ignored
- Primitives straddle the volume are Clipped!

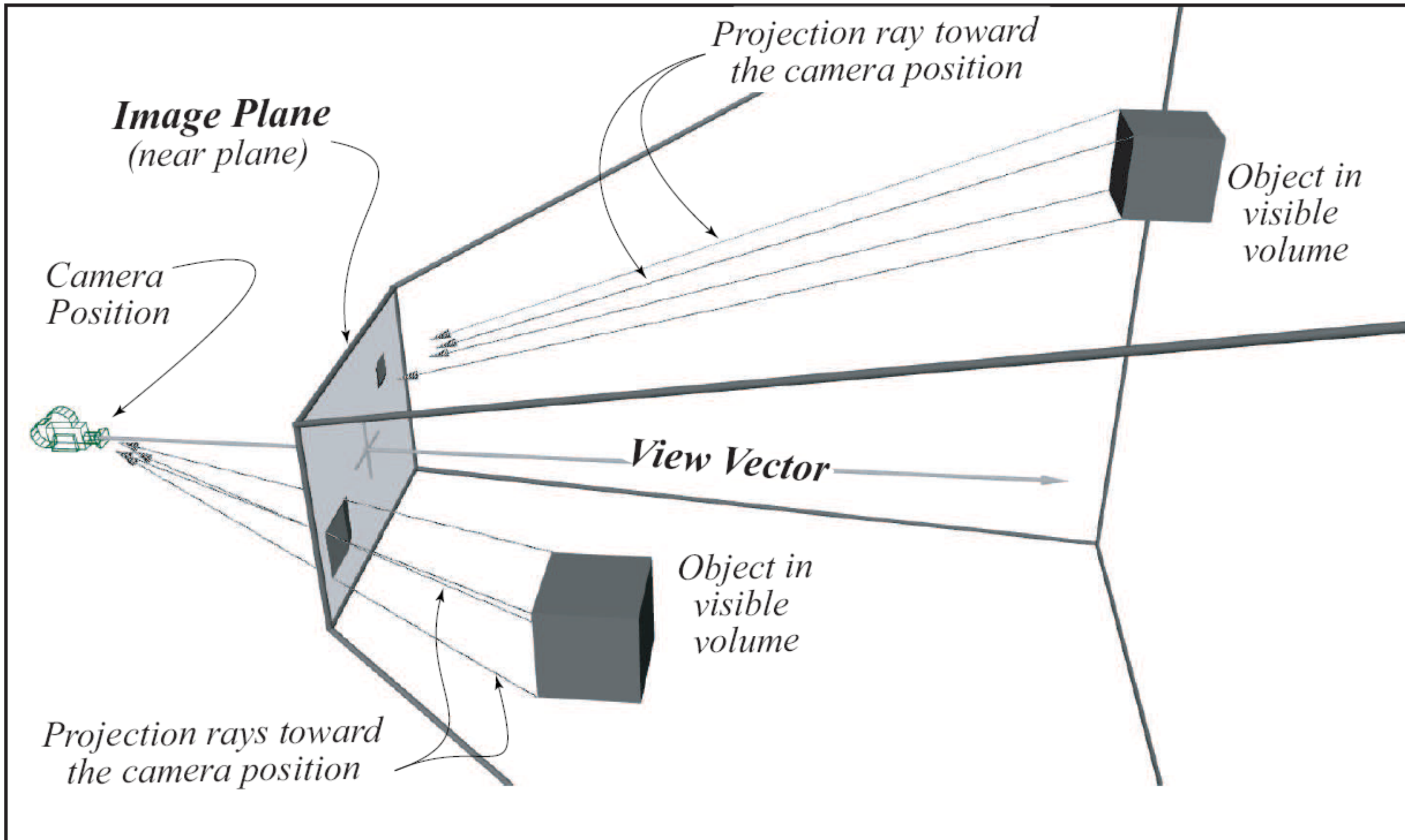
The Viewing Frustum Volume

- Volume defined by:
  - Near Plane (n)
  - Far Plane (f)
  - Field of view (fov)



- For Perspective Projection

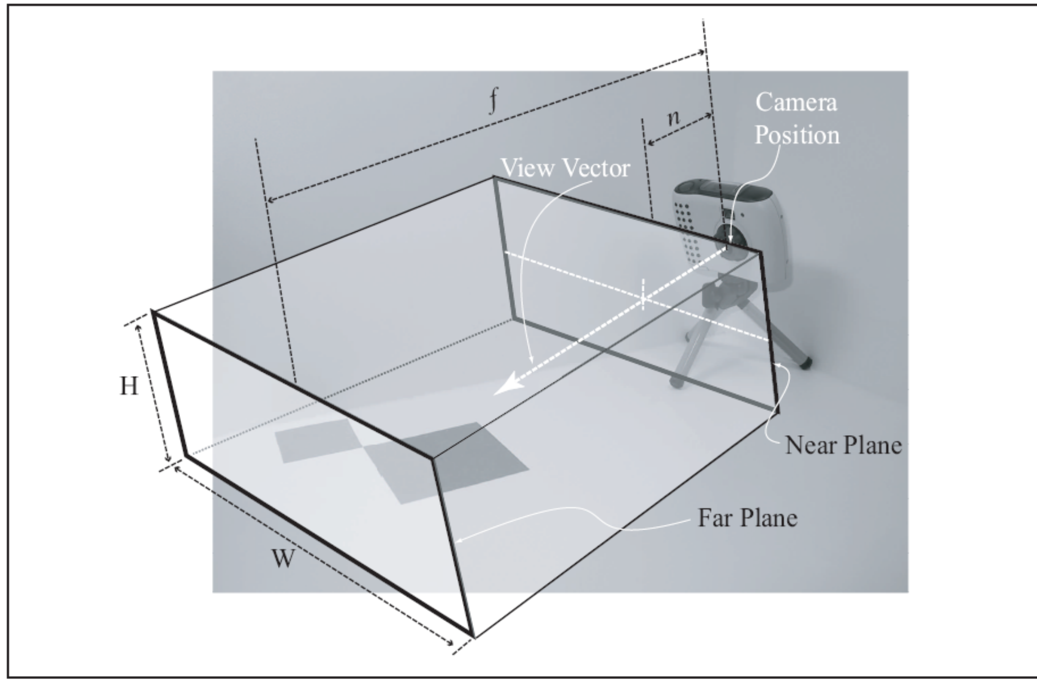
Perspective Projection



The Rectangular Visible Volume

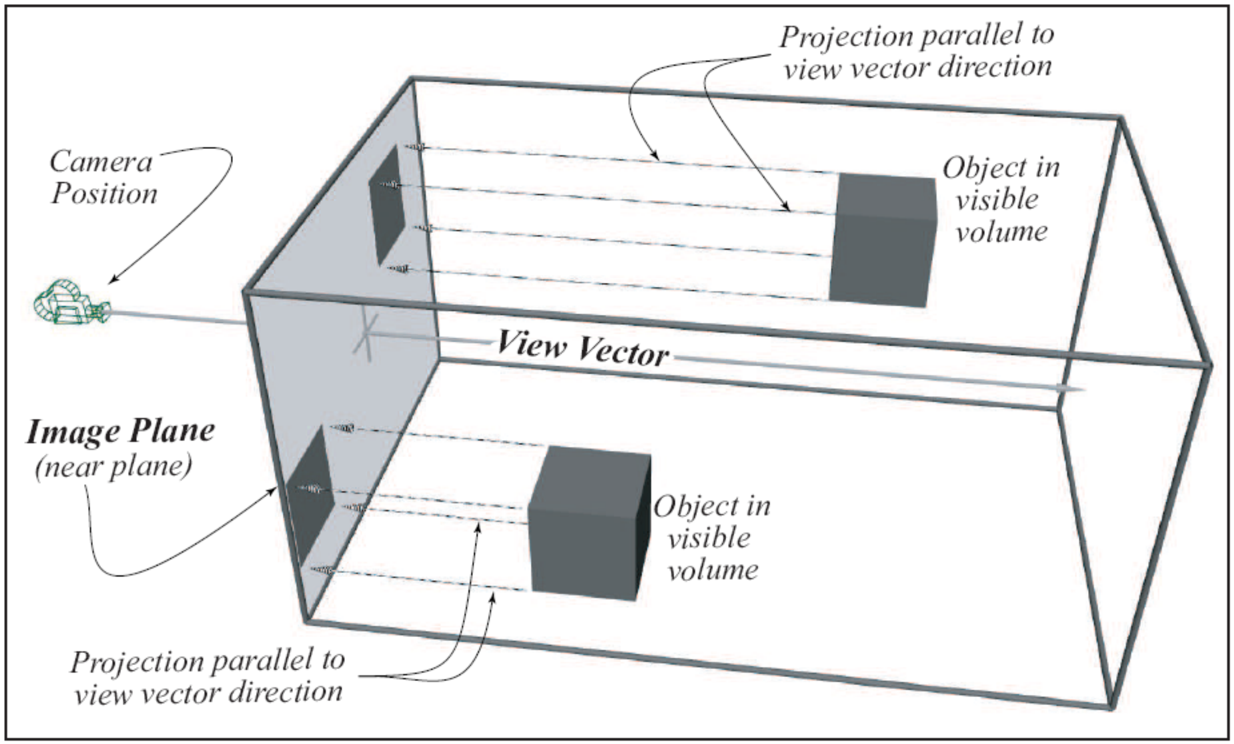
- Volume defined by:
  - Near Plane (n)
  - Far Plane (f)
  - Width (W)
  - Height (H)

$$width \times height \times depth =$$
$$W \times H \times (f - n)$$



- For Orthographic Projection

Orthographic Projection



Orthographic vs Perspective Projection

- Orthographic Projection
  - Parallel projection
  - Preserve size
    - Good for determining relative size
- Perspective Projection
  - Projection along rays
  - Closer objects appears larger
  - Human vision!

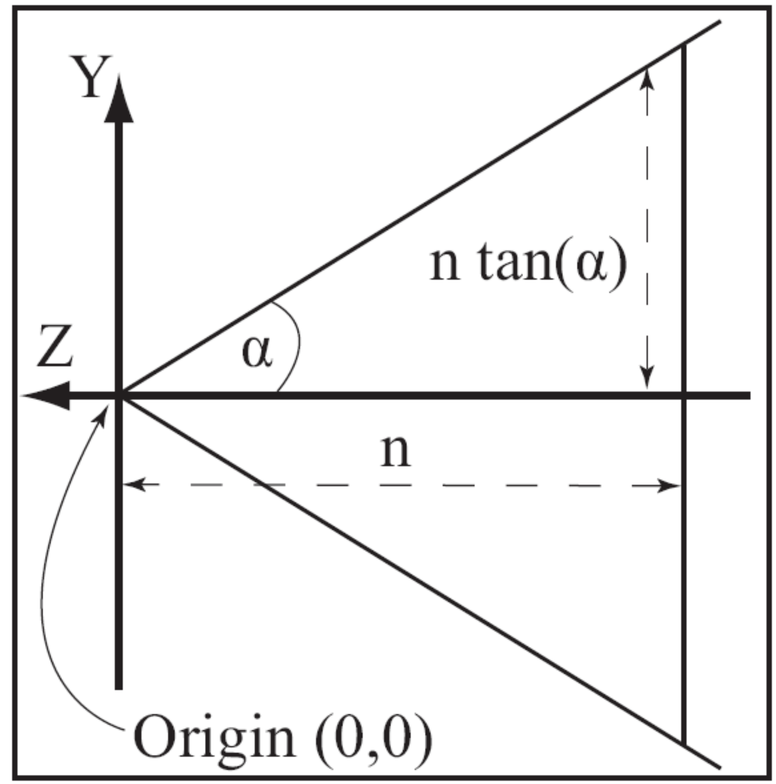
Near Plane and Aspect Ratio

- Aspect Ratio

$$AspectRatio = \frac{W_{dc}}{H_{dc}}$$

- Near Plane
  - Height (n_h) $\quad n_h = 2\ n\ tan(\alpha)$
  - Width (n_w) $\quad n_w = 2\ n\ tan(\beta)$

$$\begin{aligned} AspectRatio &= \frac{n_w}{n_h} \\ &= \frac{2\ n\ tan(\beta)}{2\ n\ tan(\alpha)} \\ &= \frac{tan(\beta)}{tan(\alpha)} \end{aligned}$$



$$tan(\beta) = AspectRatio \times tan(\alpha)$$

$$tan(\beta) = \frac{W_{dc}}{H_{dc}} \times tan(\alpha)$$

Setup Camera Matrix

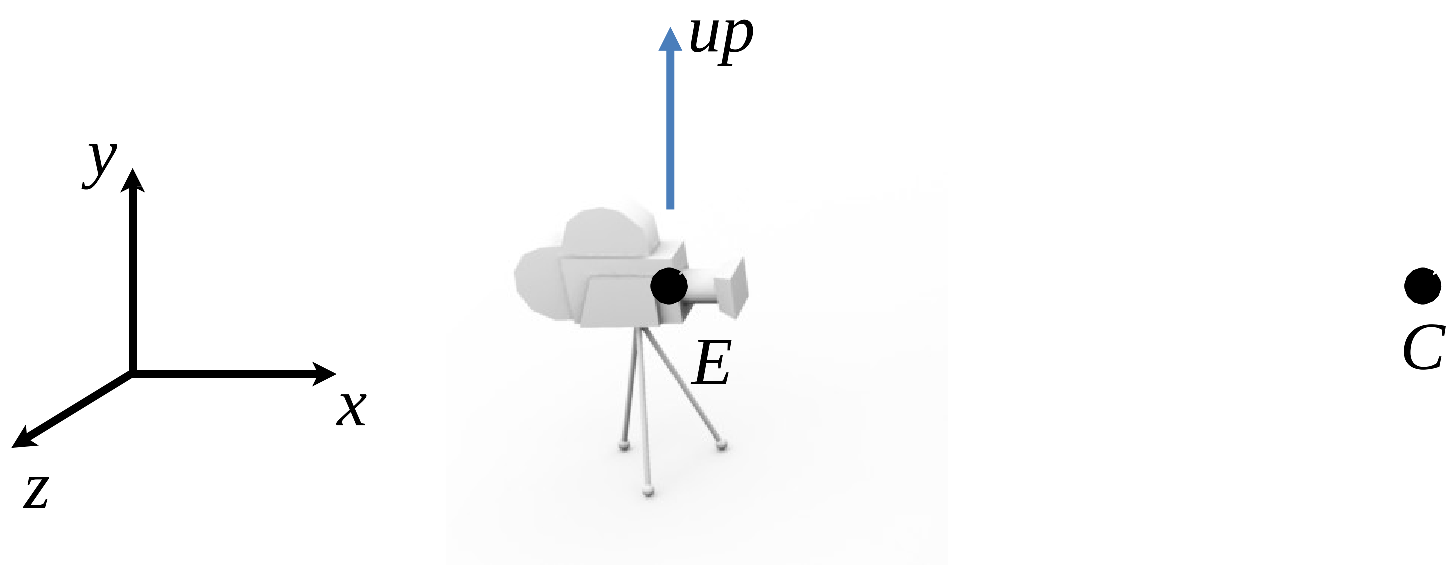
- LookAt function
  - Takes eye position (E), a position to look at (C) and an up vector (up)
  - Constructs the **View** matrix, i.e., a matrix that transforms geometry (in world space) into the camera's coordinate system (camera space)

```
mat4 View = LookAt(E.x,E.y,E.z,          // Camera position
                   C.x,C.y,C.z,          // Center of interest
                   up.x, up.y, up.z);  // Up-vector
```

Camera Placement

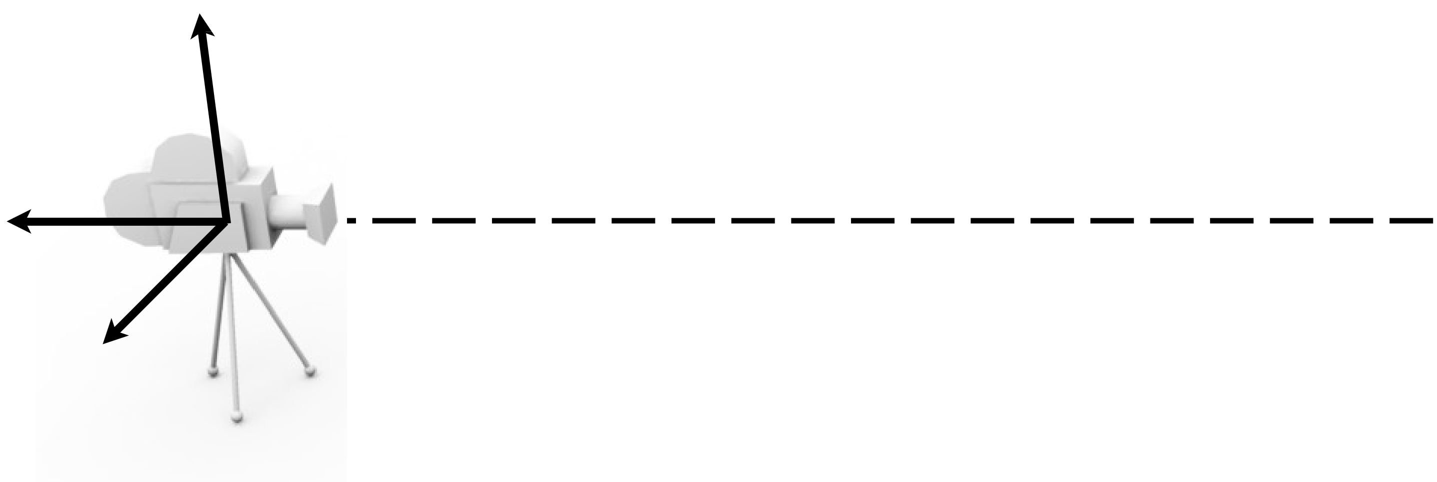
- Specify camera position (*E*), center of interest (*C*) and global up-vector (*up*)

[Up direction usually (0,1,0)]



OpenGL convention

- In OpenGL: right-hand coordinate system, looking down -z.

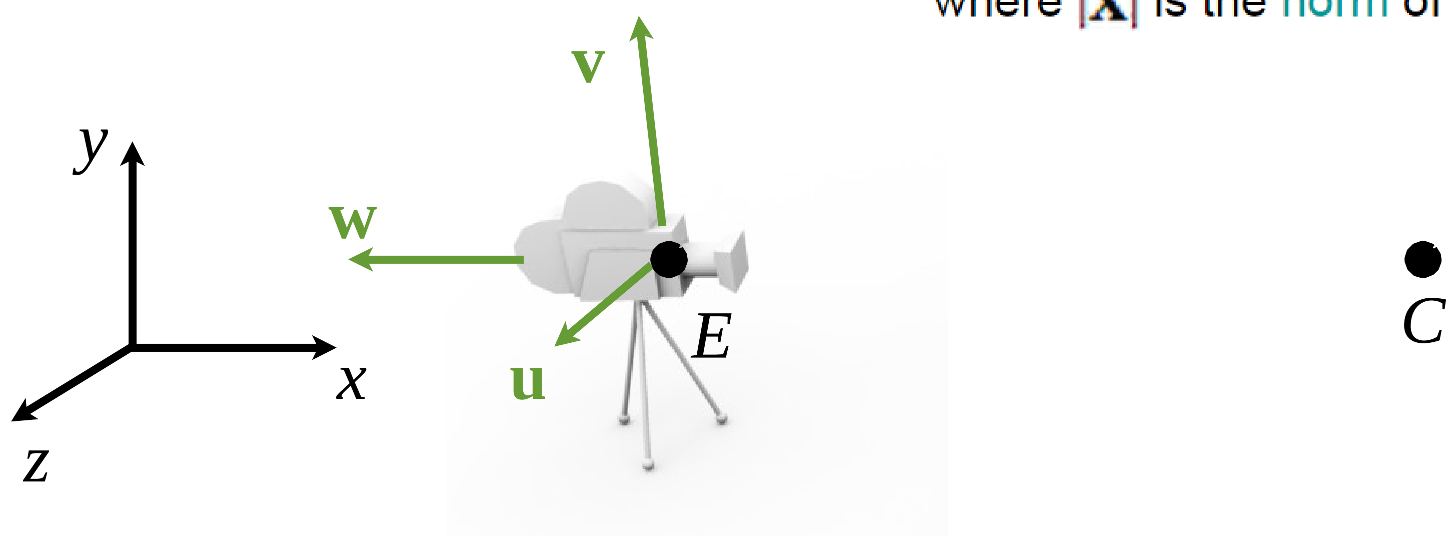


Find orthonormal basis

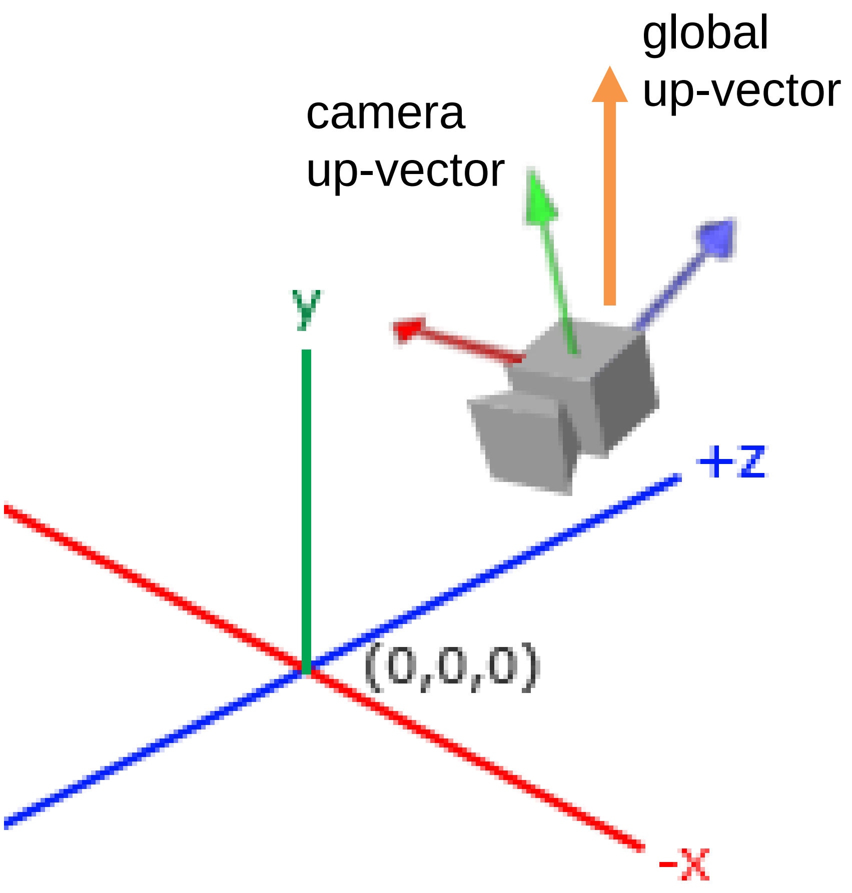
- OpenGL standard: camera looks along negative *z*. Choose **w** in direction of - (*C* - *E*) = E - C
- The coordinate system of the camera is spanned by three orthonormal vectors {**u,v,w**} s.t.

$$\boldsymbol{w} = \frac{E - C}{|E - C|} \qquad \boldsymbol{u} = \frac{up \times w}{|up \times w|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$

where $|\mathbf{X}|$ is the norm of $\mathbf{X}$

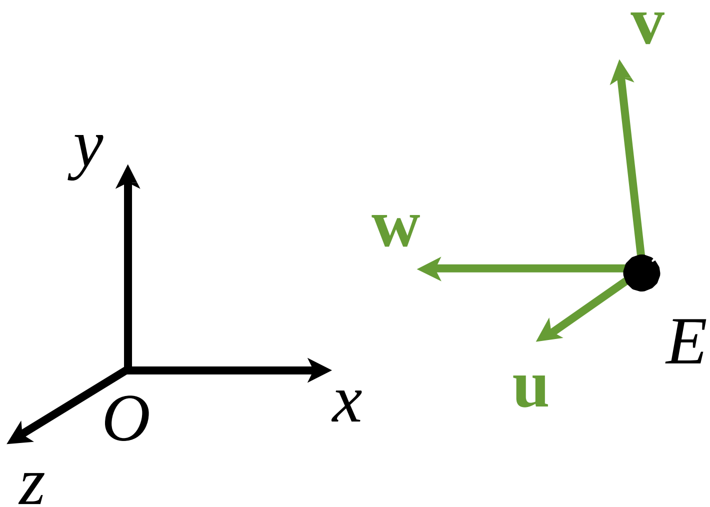


Global Up vs. Camera(Image Plane) Up



Find orthonormal basis

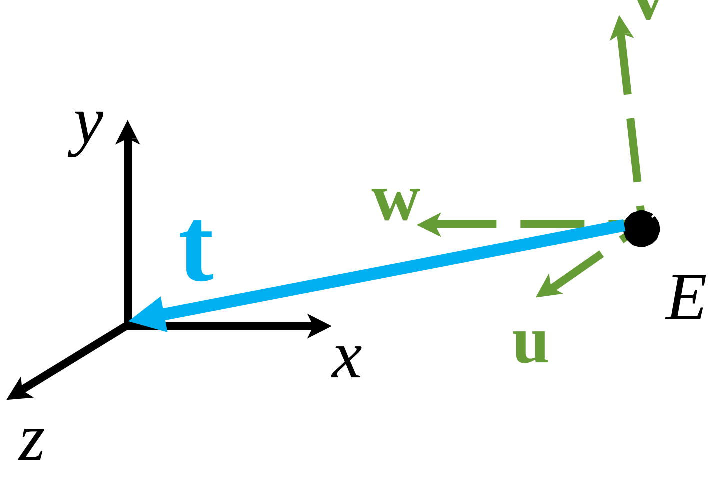
- Now, we look for matrix that transforms frame $\{\mathbf{u}, \mathbf{v}, \mathbf{w}, \mathrm{E}\}$ to $\{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathrm{O}\}$
- Translation and rotation



Find orthonormal basis

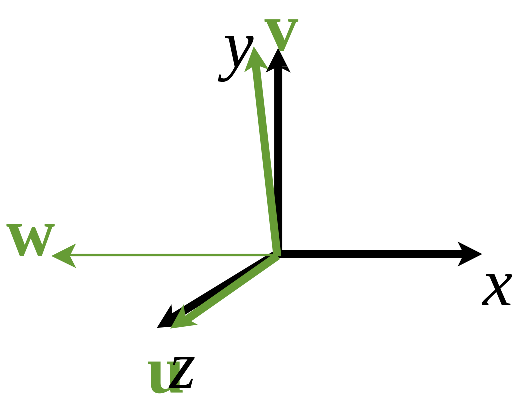
- Translate $\mathbf{uvw}\mathrm{E}$ frame so that the origin align with the $\mathbf{xyz}\mathrm{O}$ frame

$$\mathbf{t} = \begin{bmatrix} -E_x \\ -E_y \\ -E_z \end{bmatrix}$$



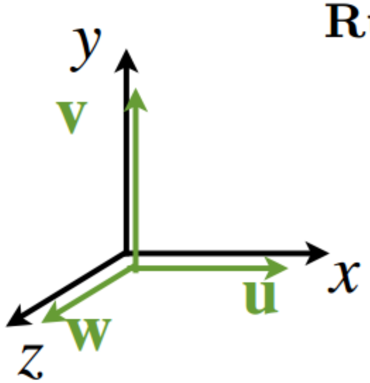
Find orthonormal basis

- Translate **uvw**E frame so that the origin align with the **xyz**O frame



Find orthonormal basis

- Then rotate $\mathbf{uvw}$ basis so that the three axes align, $\mathbf{u} \,//\, \mathbf{x}$, $\mathbf{v} \,//\, \mathbf{y}$ and $\mathbf{w} \,//\, \mathbf{z}$
- Rotation matrix given by $\mathrm{R} = \begin{bmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{bmatrix}$
- R rotates vectors $\mathbf{uvw}$ to $\mathbf{xyz}$



$$\mathbf{Ru} = \begin{bmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{bmatrix} \begin{bmatrix} | \\ \mathbf{u} \\ | \end{bmatrix} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{u} \\ \mathbf{v}\cdot\mathbf{u} \\ \mathbf{w}\cdot\mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}$$

$$\mathbf{Rv} = \mathbf{y}, \quad \mathbf{Rw} = \mathbf{z}$$

Camera Placement

- Combine the two transforms
- The view matrix M: Move to center, and then apply rotation

$$M = \underbrace{\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Rotate}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Move to center}}$$

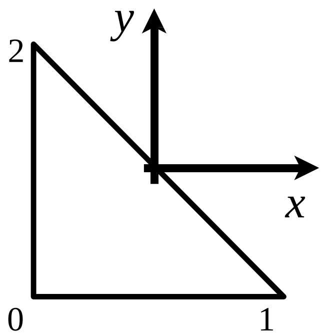
$$\boldsymbol{w} = \frac{E - C}{|E - C|} \qquad \boldsymbol{u} = \frac{up \times w}{|up \times w|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$

An OpenGL program in GLUT

Main

```
int main(int argc, char **argv)
{
   GLenum err = glewInit();  // Init GLEW
   if (GLEW_VERSION_3_0) { printf("GL version 3 supported \n"); }

   glutInit(&argc, argv); // Init GLUT
   glutInitDisplayMode(GLUT_DEPTH |GLUT_DOUBLE   | GLUT_RGBA);
   glutInitWindowPosition(100,100);
   glutInitWindowSize(512,512);
   glutCreateWindow("GLSL Test");

   // Set GLUT callbacks

   glutDisplayFunc(render);
   glutIdleFunc(render);
   glutReshapeFunc(resize);
   glutKeyboardFunc(processKeys);
   glutMouseFunc(processMouse);
   glutMotionFunc(processMouseActiveMotion);

   init();       // Create geometry and shaders
   glutMainLoop();
   cleanup();

   return 0;
}
```

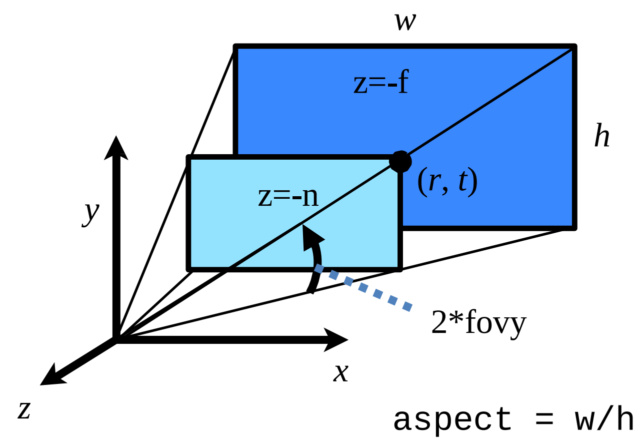
Init - Setup Geometry




```
void init()
{
     glClearColor(1.0,1.0,1.0,1.0);
    gShaderProgramID  = initShaders();

    // Create geometry (one triangle)
    vec3 vertices[3];
    vertices[0] = vec3( -0.5f, -0.5f, 1.0f);
    vertices[1] = vec3(  0.5,  -0.5f, 1.0f);
    vertices[2] = vec3( -0.5f,   0.5,  1.0f);


     // Create a vertex array object
     glGenVertexArrays( 1, &gVaoID );
     glBindVertexArray( gVaoID );
     // Create and initialize a buffer object
     glGenBuffers( 1, &gVboID );
     glBindBuffer( GL_ARRAY_BUFFER, gVboID );
     glBufferData( GL_ARRAY_BUFFER, sizeof(vertices),
                   vertices, GL_STATIC_DRAW );

      // Initialize the vertex position attribute from the vertex shader
     GLuint pos = glGetAttribLocation( gShaderProgramID, "vPosition" );
     glEnableVertexAttribArray( pos );
     glVertexAttribPointer( pos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
}
```

```
in vec4 vPosition;
uniform mat4 MVP;//ModelViewProj

void main()
{
   gl_Position = MVP*vPosition;
}
```

Resize



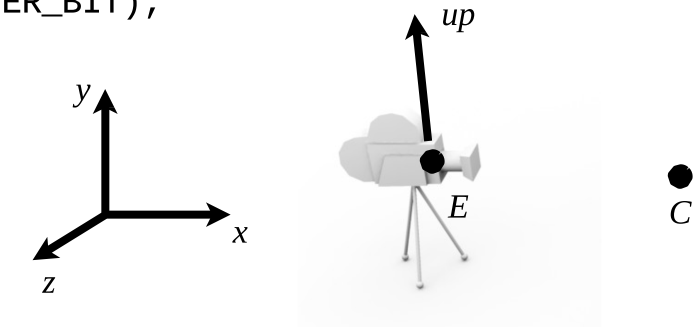

```
aspect = w/h
t = tan(fovy)*n
r = t*aspect
```

```
// If the size of the window changed,
// call this to update the GL matrices
void resize(int w, int h)
{
    if(h == 0) h = 1; // Prevent a divide by zero

    // Calculate the projection matrix
    float aspect = ((float)w) / h;
    float fovy = 45.0;
    float near = 0.01;
    float far  = 10.0;
    gProjectionMatrix = Perspective(fovy, aspect, near, far);

    glViewport(0, 0, w, h); // Set the viewport to be the entire window
}
```

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Render

```
void render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Calculate the view matrix
    vec3 at(0.0,0.0,0.0);
    vec3 up(0.0,1.0,0.0);
    mat4 View = LookAt(gEyePos, at, up );

    // Compute world matrix
    mat4 World = ...;

    // Compute ModelViewProjection matrix
    mat4 MVP = gProjectionMatrix*View*World;

    // Pass the modelview projection matrix to  the shader
    GLuint mvpID = glGetUniformLocation(gShaderProgramID,"MVP");
    glUniformMatrix4fv(mvpID, 1, GL_TRUE, (GLfloat*)MVP.getFloatArray());

    // draw a triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glutSwapBuffers();
}
```




```
in vec4 vPosition;
uniform mat4 MVP; //ModelViewProj

void main()
{
   gl_Position = MVP * vPosition;
}
```

Input Handling

```
// Mouse and keyboard handling
void processKeys(unsigned char key, int x, int y)
{
  switch (key) {
   case 27:
      exit(0);
      break;
   case 'w': case 'W':
      gEyePos.z -= 0.1;
      break;
   case 's': case 'S':
      gEyePos.z += 0.1;
      break;
   default:
      break;
  }
}
```