Computer Graphics

View in 2D & 3D

Teacher: Dr. Zhuo SU (苏卓)

E-mail: suzhuo3@mail.sysu.edu.cn

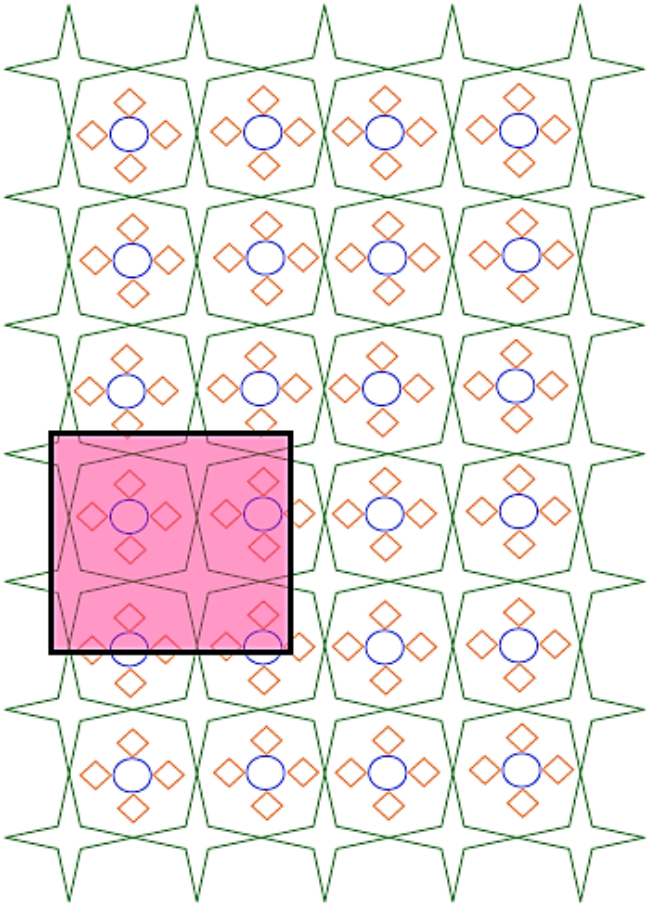
School of Data and Computer Science

Outline

- **2D Viewing**
- 3D Viewing
  - Classic view
  - Computer view
    - Positioning the camera
    - Projection

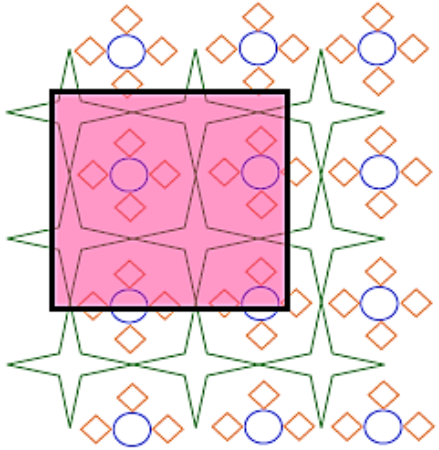
2D Viewing

- The world is infinite (2D or 3D) but the screen is finite
- Depending on the details the user wishes to see, he limits his view by specifying a window in this world

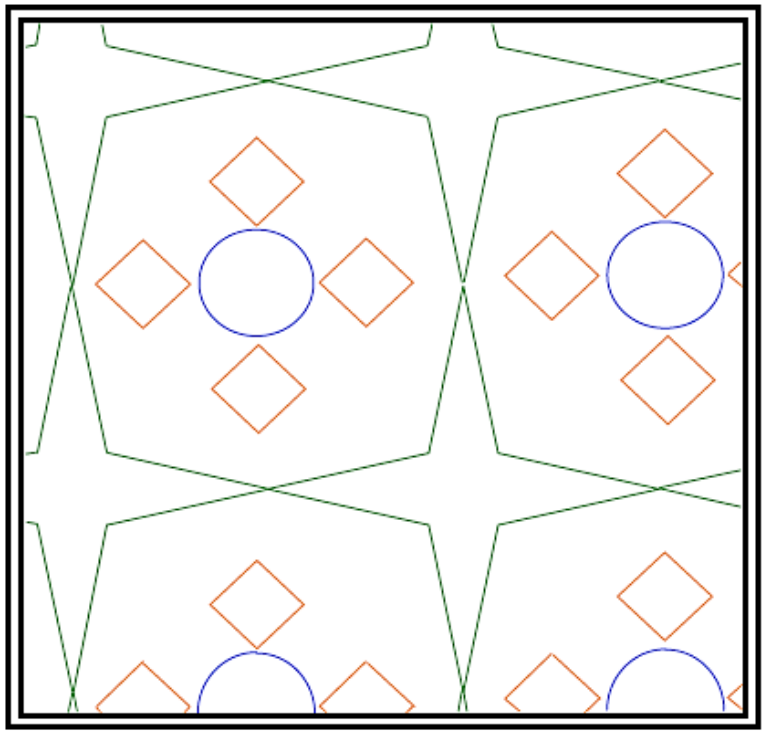


2D Viewing

- By applying ***appropriate transformations*** we can map the world seen through the window on to the screen



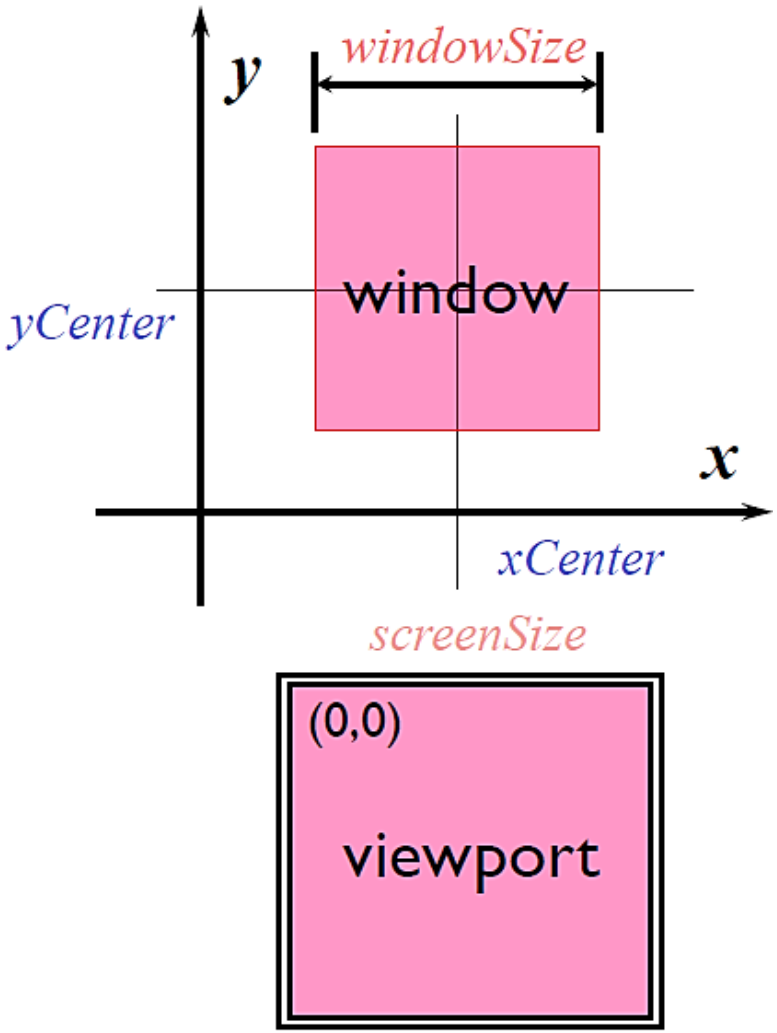
2D World



Screen

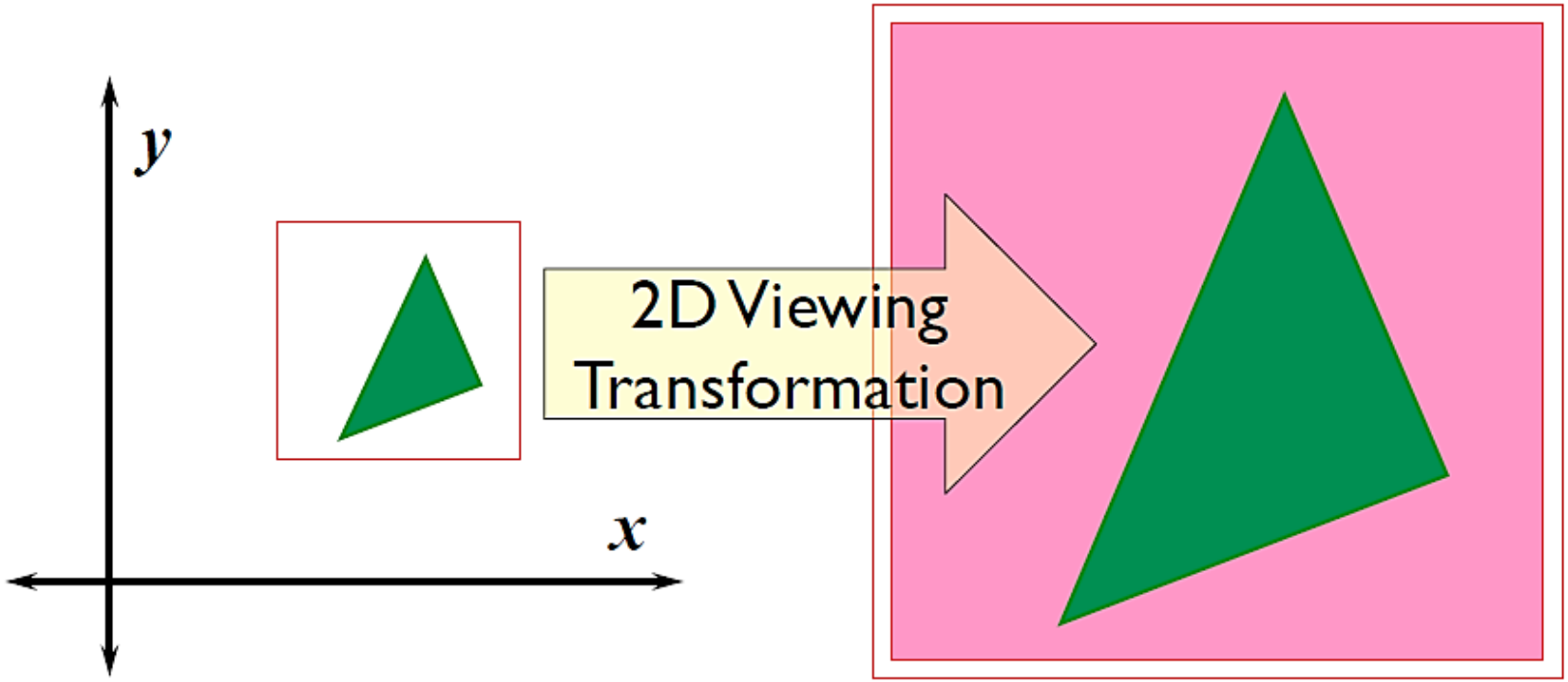
Windowing Concepts

- ***Window*** is a rectangular region in the 2D world specified by
  - a ***center*** *(xCenter, yCenter)* and
  - ***size*** *windowSize*
- Screen referred to as ***Viewport*** is a discrete matrix of pixels specified by
  - ***size*** *screenSize* (in pixels)

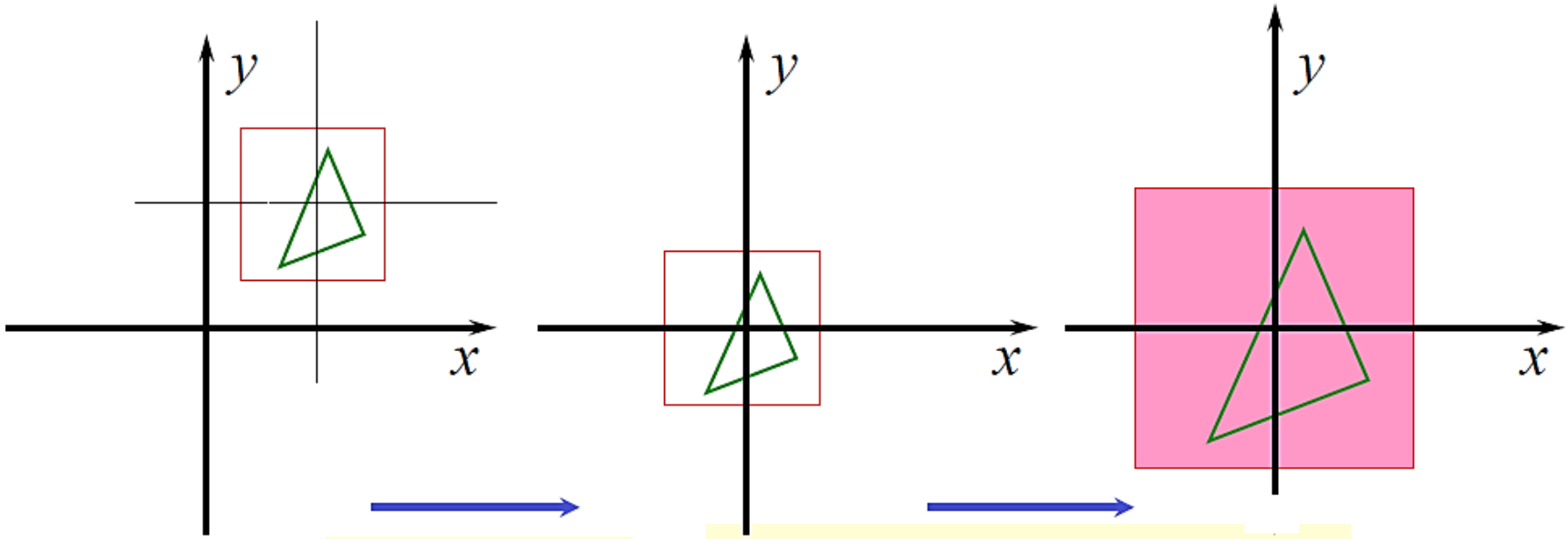


2D Viewing Transformation

- Mapping the 2D world seen in the ***window*** on to the ***viewport*** is ***2D viewing transformation***
  - also called ***window to viewport transformation***



Deriving 2D Viewing Transformation

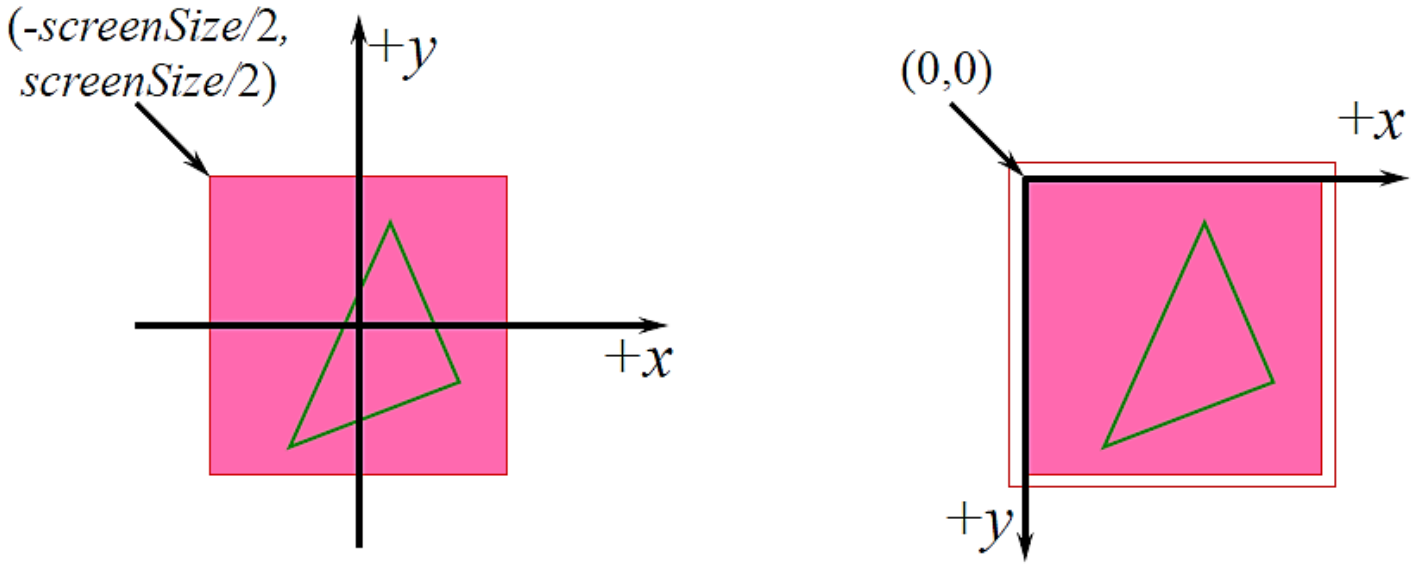


$$x - xCenter$$
$$y - yCenter$$

$$(x - xCenter) \bullet scaleFactor$$
$$(y - yCenter) \bullet scaleFactor$$

$$where, \quad scaleFactor = \frac{screenSize}{windowSize}$$

Deriving 2D Viewing Transformation



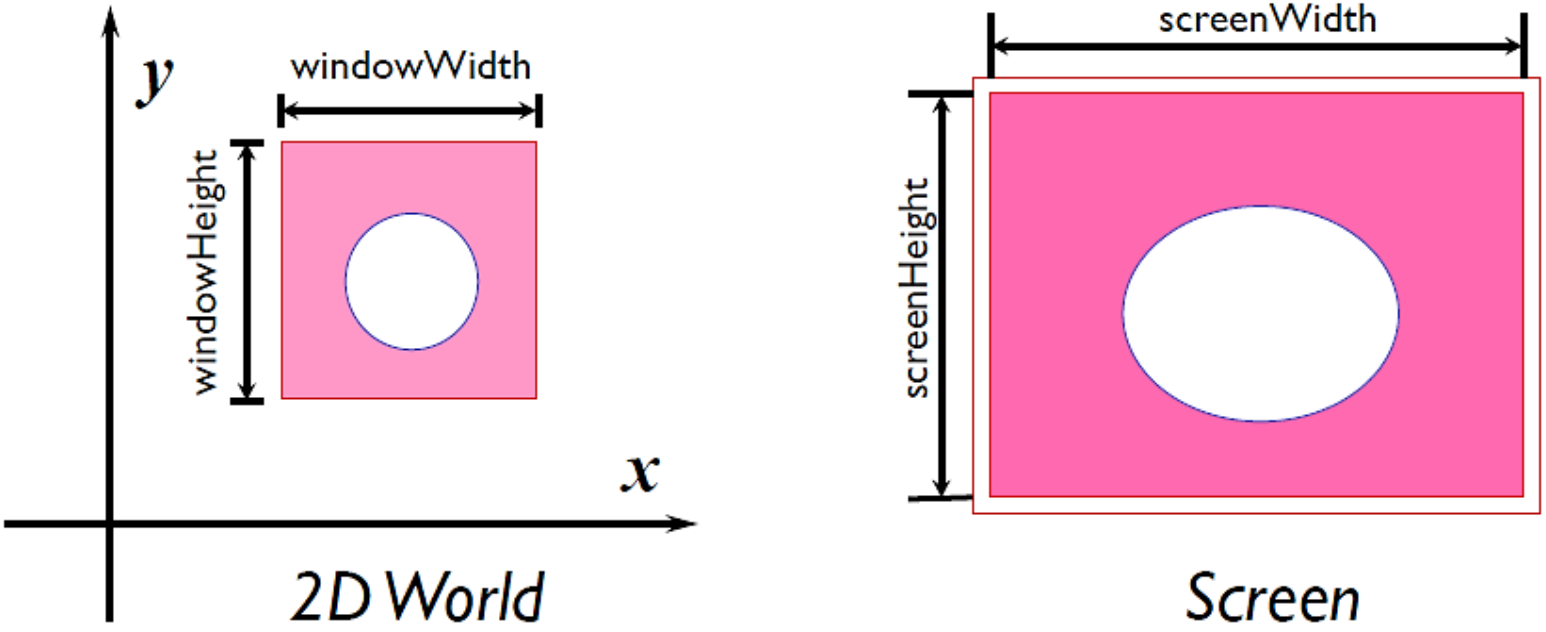
$$\frac{screenSize}{2} + (x - xCenter) \bullet scaleFactor$$

$$\frac{screenSize}{2} - (y - yCenter) \bullet scaleFactor$$

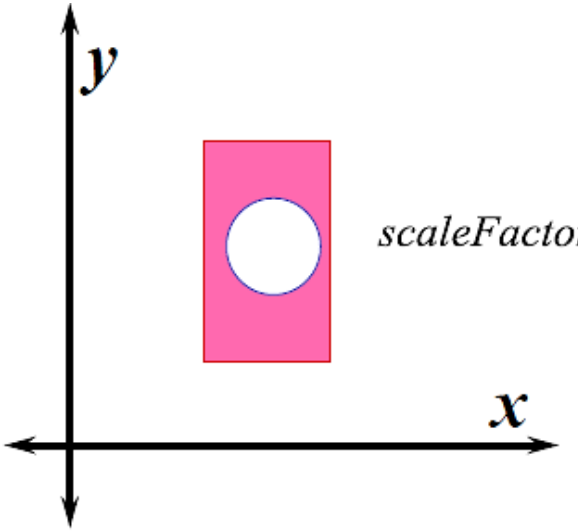
- Given any point in the 2D world, the above transformations maps that point on to the screen

The Aspect Ratio (纵横比)

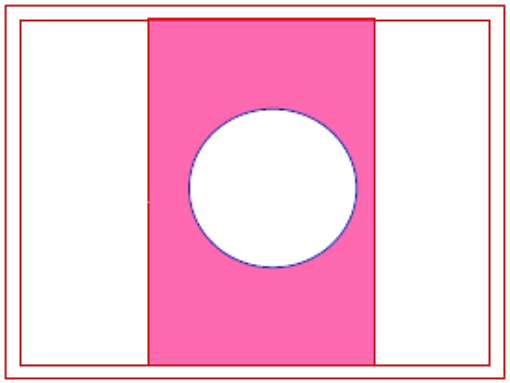
- In 2D viewing transformation the ***aspect ratio*** is maintained when the scaling is uniform
- ***scaleFactor*** is same for both x and y directions

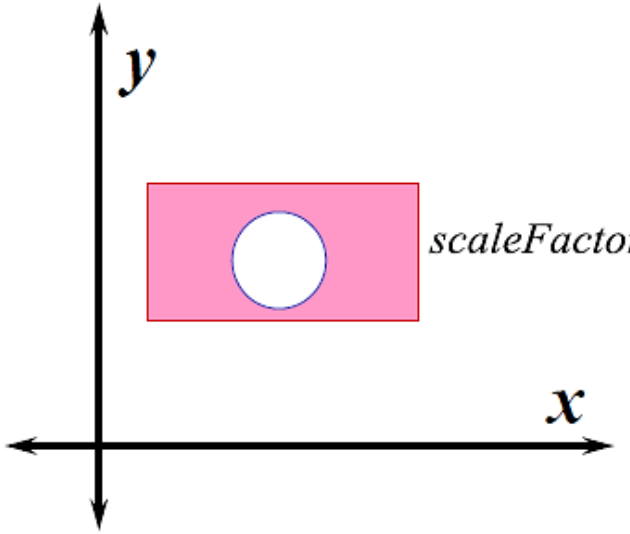


Maintaining the Aspect

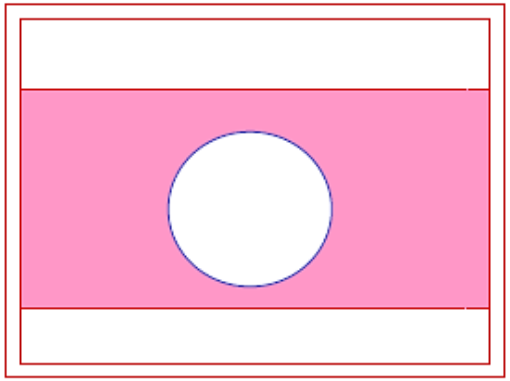


$$scaleFactor = \frac{screenHeight}{windowHeight}$$





$$scaleFactor = \frac{screenWidth}{windowWidth}$$



OpenGL Commands

gluOrtho2D(left, right, bottom, top)

Creates a matrix for projecting 2D coordinates onto the screen and multiplies the current matrix by it.

glViewport(x, y, width, height)

Define a pixel rectangle into which the final image is mapped.

(x, y) specifies the lower-left corner of the viewport.

(width, height) specifies the size of the viewport rectangle.

Outline

- 2D Viewing
- 3D Viewing
  - **Classic view**
  - Computer view
    - Positioning the camera
    - Projection

3D Viewing

- To display a 3D world onto a 2D screen
  - Specification becomes complicated because there are many parameters to control
  - Additional task of reducing dimensions from 3D to 2D (projection)
  - 3D viewing is analogous to taking a picture with a camera

Classic View

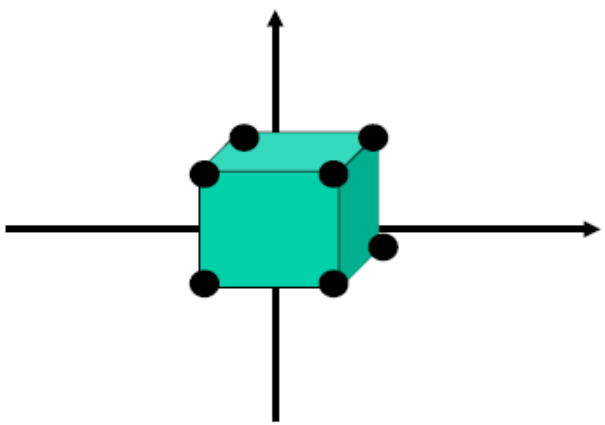
- **Three basic elements needed**
  - One or more objects
  - Observer with a projection plane
  - Projection transform: from the object to the projection plane

- **The classic view is based on the relationship between these elements**

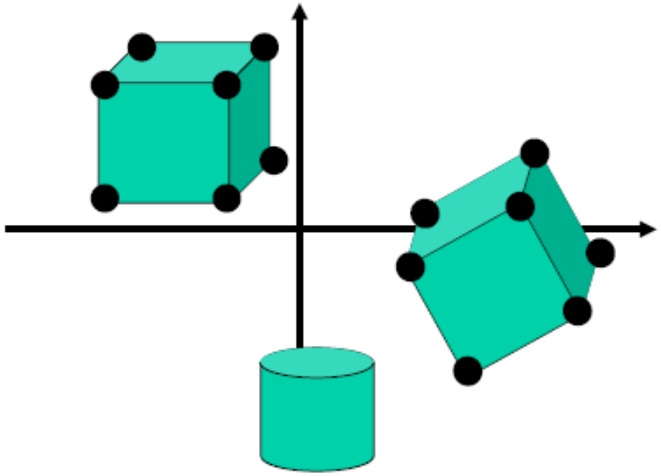
Modeling vs Viewing

- *viewing* ***transformations*** $\neq$ *modeling* ***transformations***
  - *Modeling transformations* actually position the objects in the world,
  - but *viewing transformations* are applied only to make a mapping from world to the screen
  - *Viewing transformations* do not change the actual world in any fashion

Modeling Coordinates

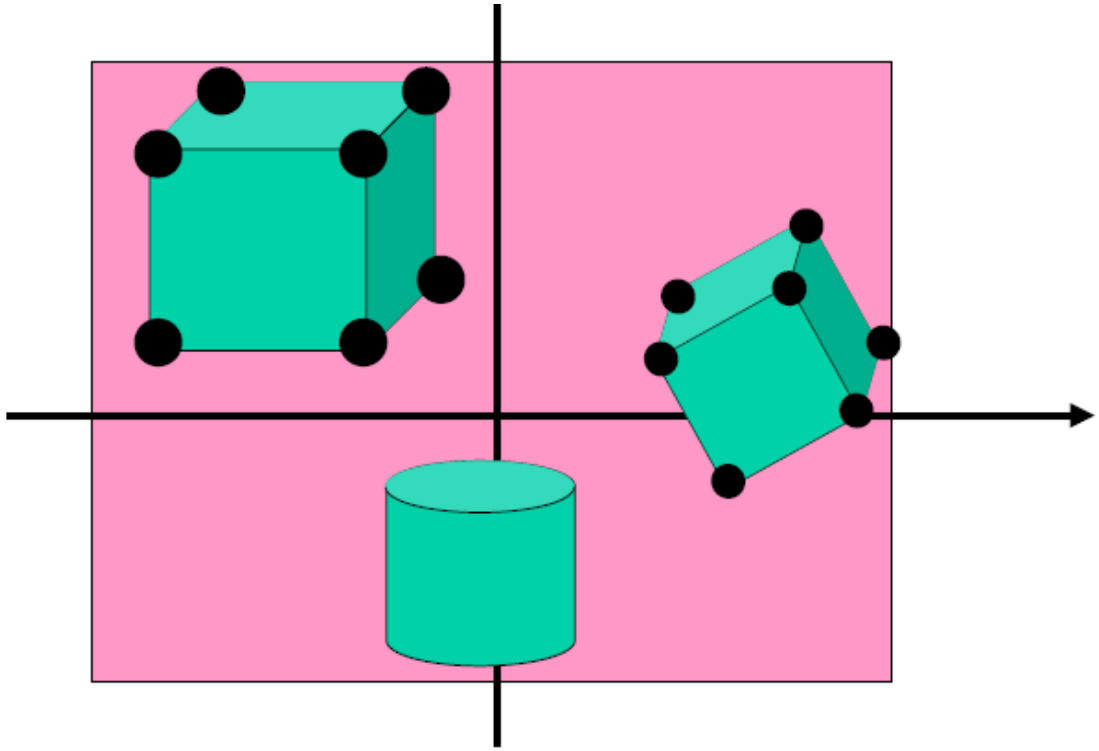


Objects are usually defined
in their own local coordinate system
(instance transformation)



Place (transform) these objects in a single scene,
in world coordinates
(modeling transformation)

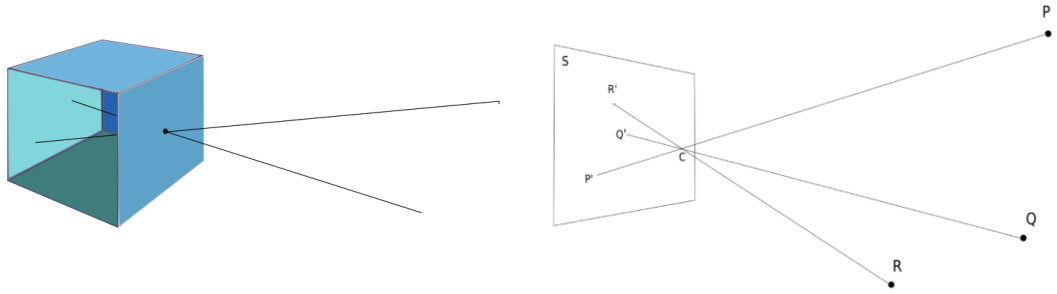
Screen Coordinates



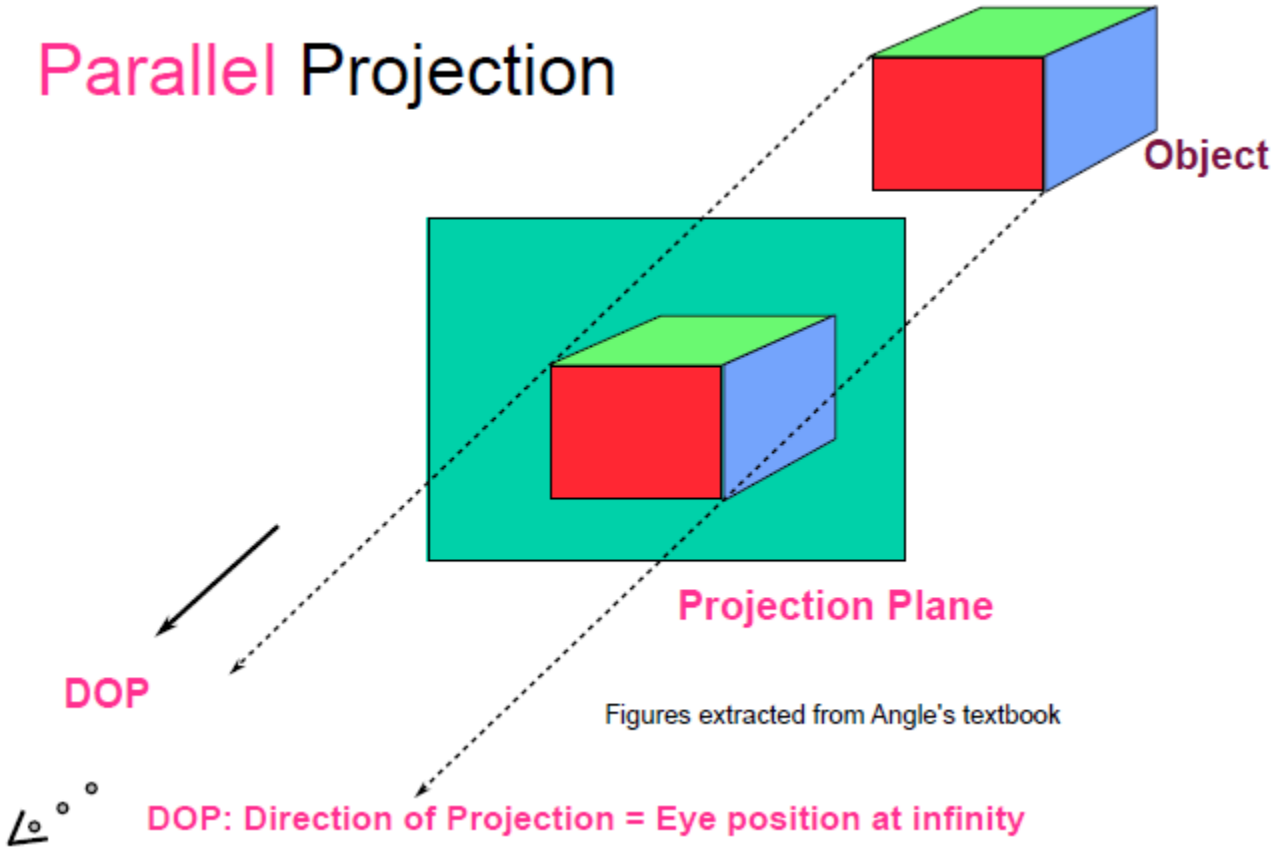
Finally, we want to project these objects onto the screen

Plane Geometry Projection

- That is projected onto the **plane** of the standard projection
- **Projection line** is a **straight** line
  - Gathering in the center of projection (汇聚)
  - Parallel to each other (平行)
- This projection preserve collinearity
- Some applications such as mapping (地图映射) need to be **non planar projection**

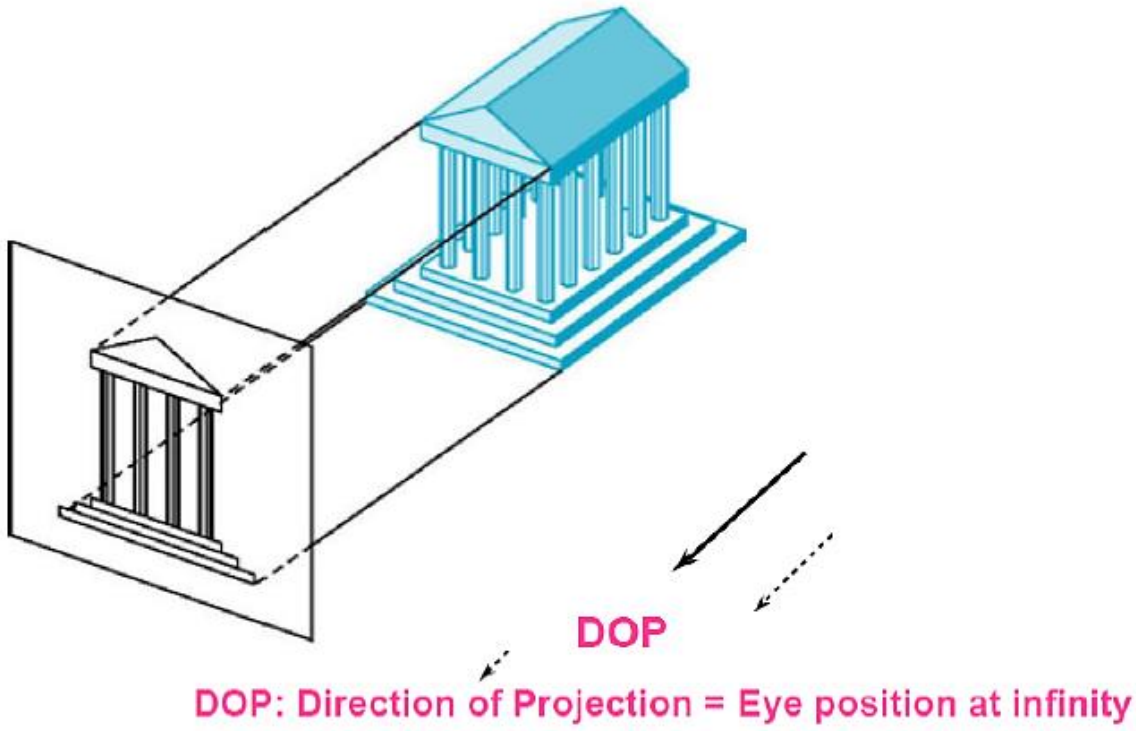


Engineering Viewing



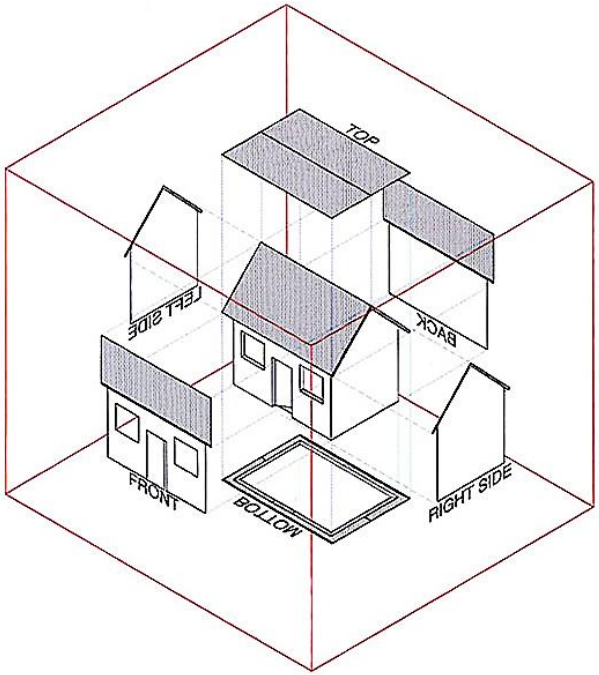
Orthogonal projection (正交投影)

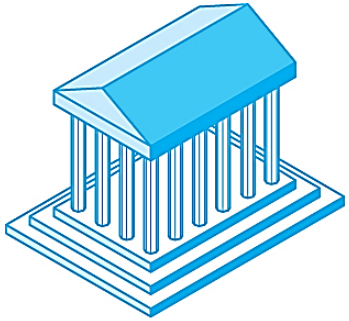
- Projection line perpendicular (垂直) to the plane of projection



Multi-view orthographic projection (多角度正交投影)

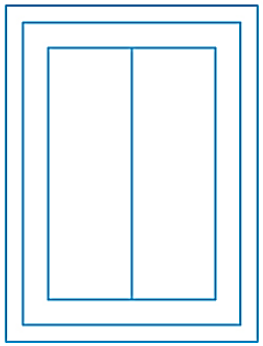
- The projection plane parallel to the reference plane(DOP is perpendicular to the view plane)
- Usually projection from the front, top and side

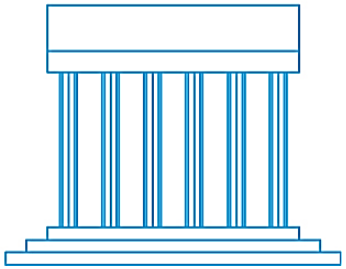




Front

Top

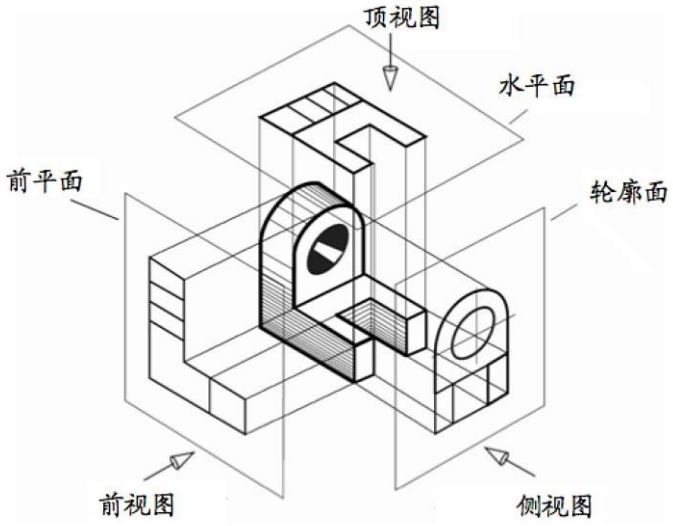


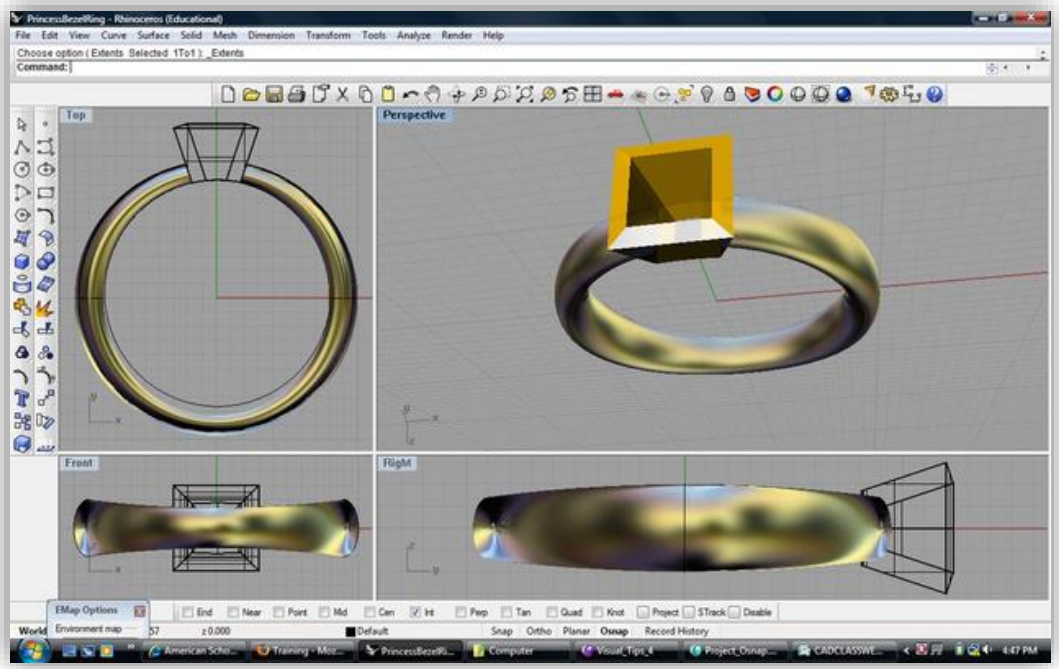


Back

Multi-view of CAD parts

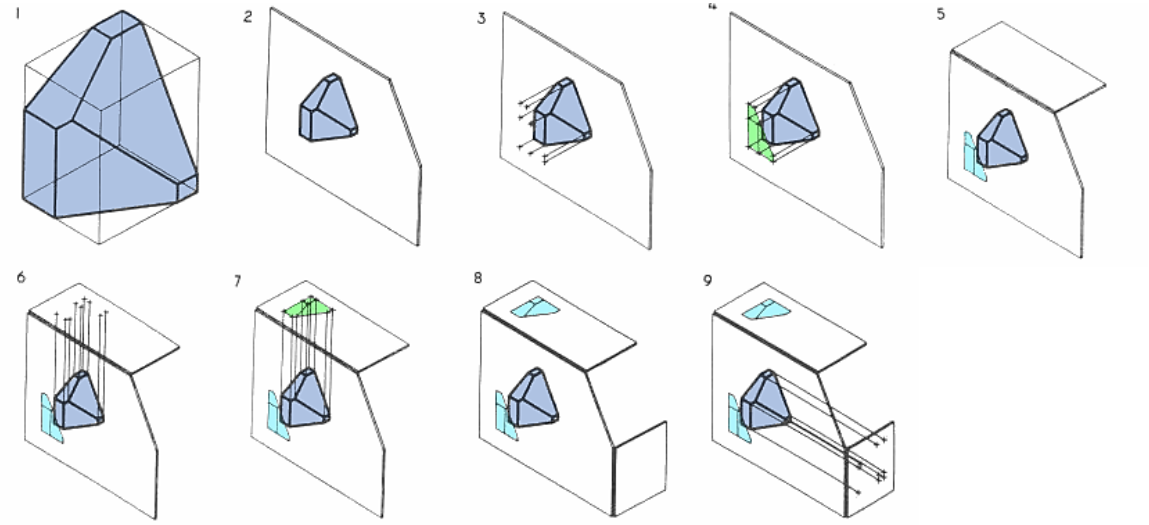
在CAD和建筑行业中，通常显示出来三个视点图以及等角投影图。





Advantages & Disadvantages

- Keep the distance and angle
  - Remain the shapes
  - Use for measurement (building, manual)
- Can't see the global real object shape, because many surface not visible in view
  - Sometimes adding isometric drawing (等角图)

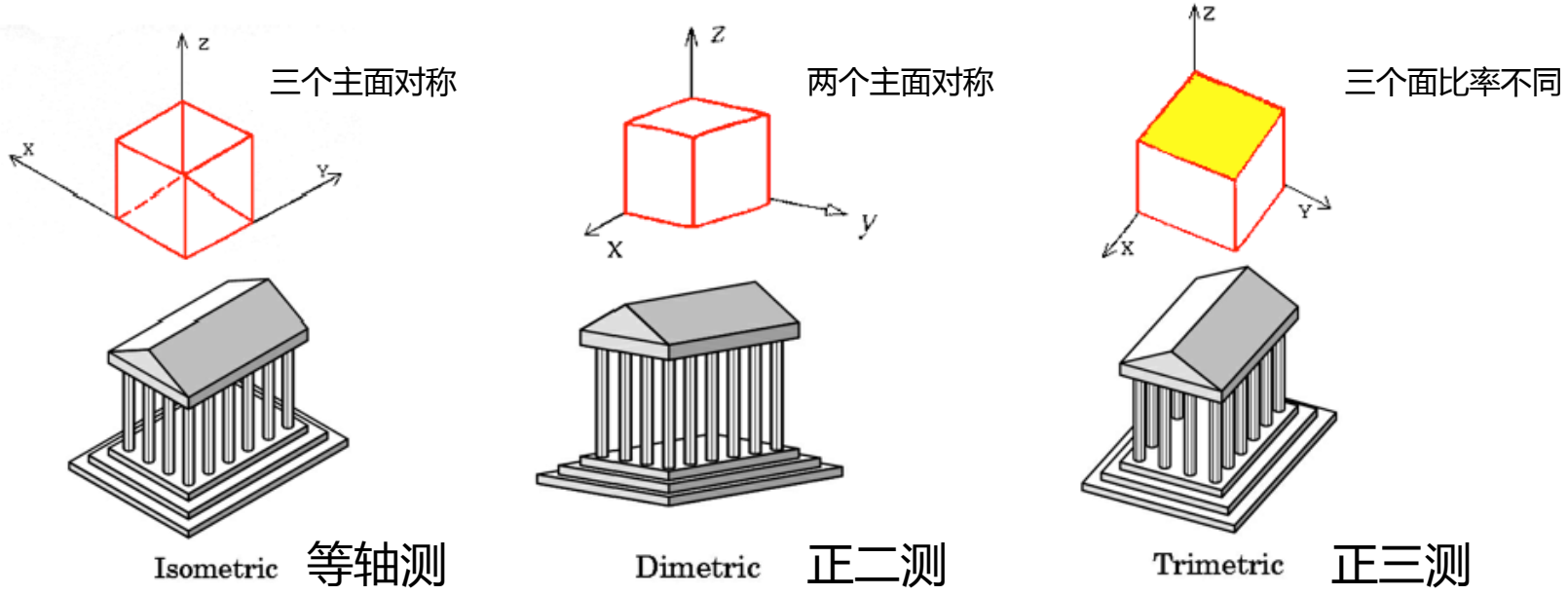


Axonometric projection (轴测投影)

- DOP orthogonal to the projection plane, but…

  …orient projection plane with respect to the object

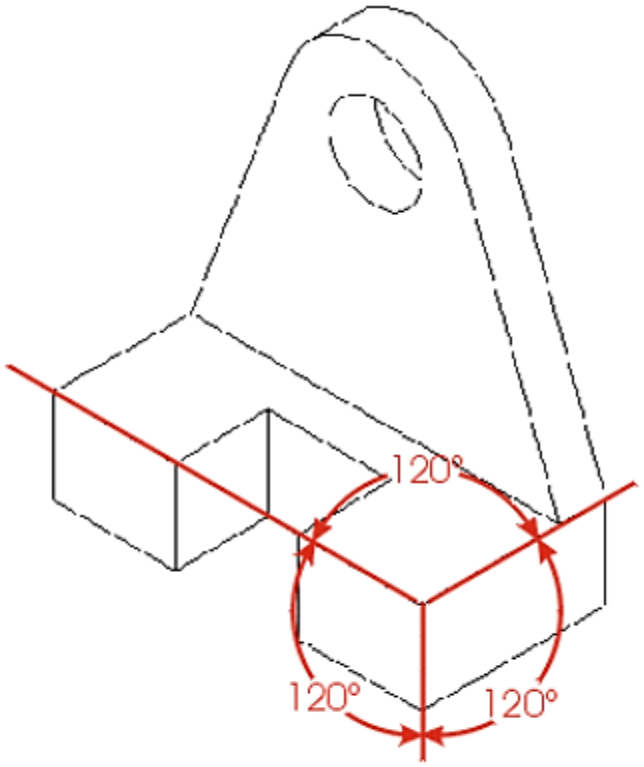
- Parallel lines remain parallel, and receding lines are equally foreshortened by some factor.



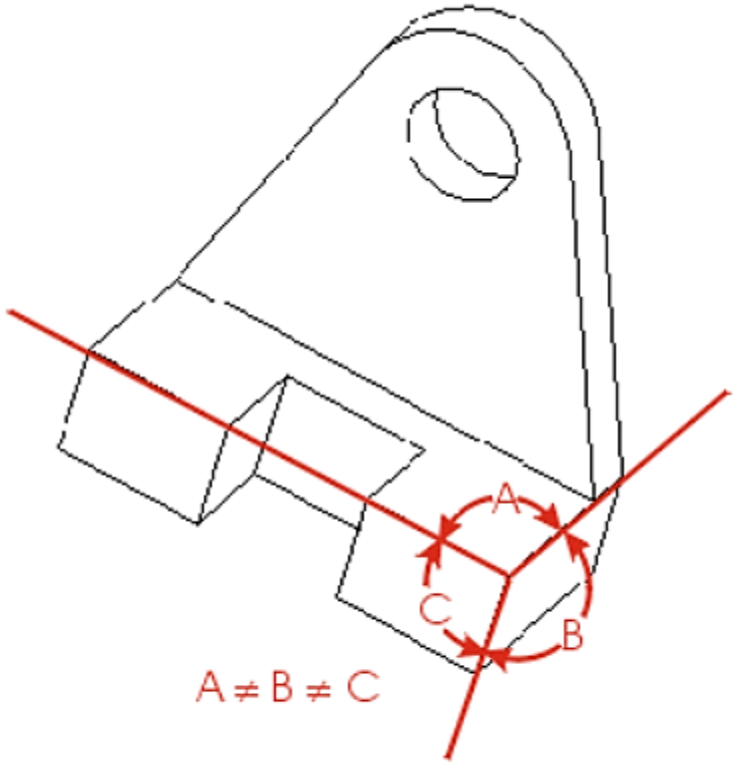
Projection type depends on angles made by projector with the three principal axes.

Figures extracted from Angle's textbook

Mechanical Drawing



isometric 等角图



trimetric 不等角图

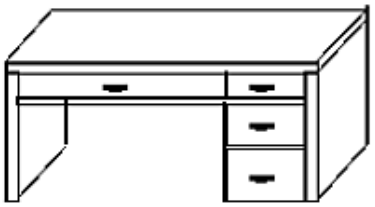
Oblique Projections (斜平行投影)

- Most general parallel views
- Projectors make an arbitrary angle with the projection plane
- Angles in planes parallel to the projection plane are preserved

斜平行投影

cavalier

斜等测



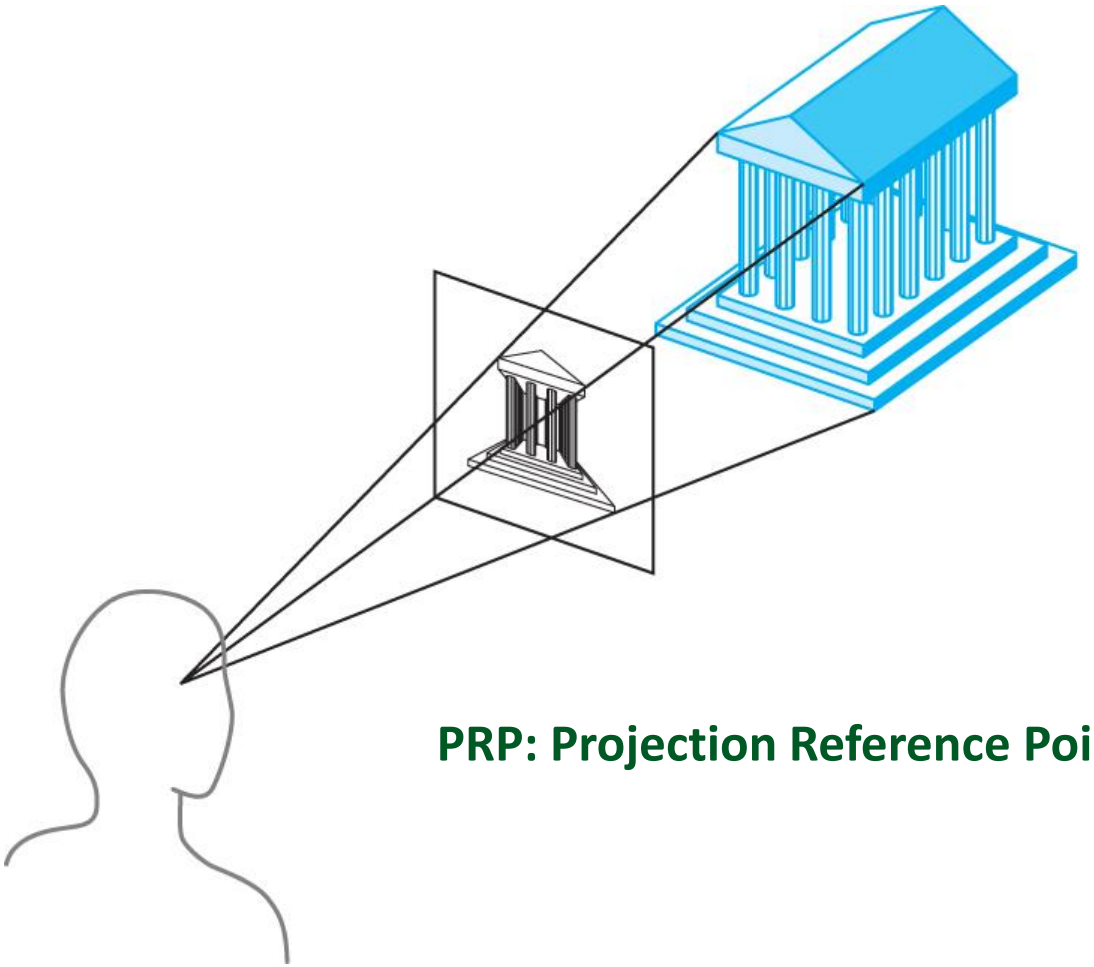
cabinet

斜二测

Cavalier
Angle between projectors and projection plane is 45°. Perpendicular faces are projected at full scale

Cabinet
Angle between projectors and projection plane is 63.4°. Perpendicular faces are projected at 50% scale

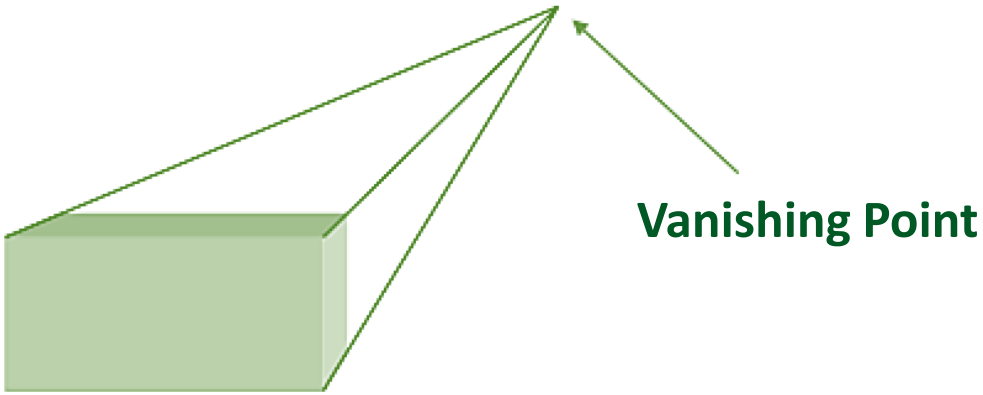
Perspective projection



PRP: Projection Reference Point = Eye position

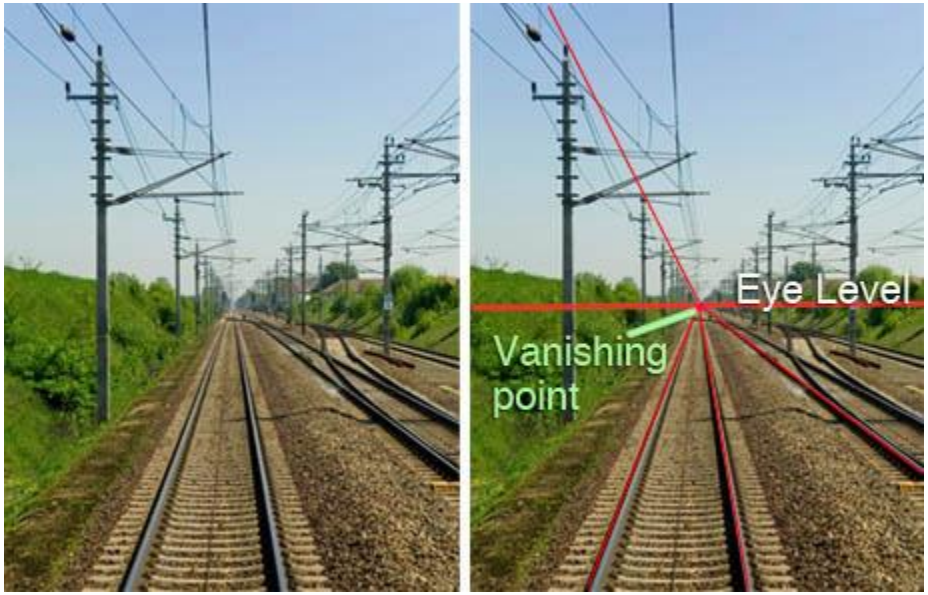
Vanishing Points

- On the object of all parallel lines (not parallel to the projection plane) projected to a point
- Hand draw simple perspective projection on the need to use the vanishing point

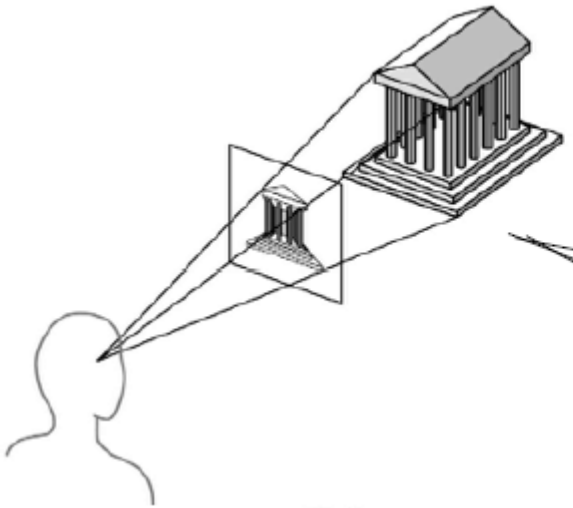


Examples

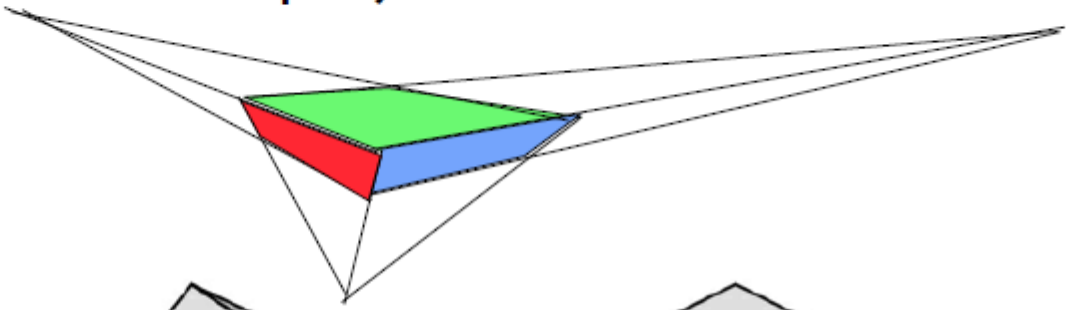
- Characterized by diminution of size
- The farther the object, the smaller the image
- Foreshortening depends on distance from viewer
- Can't be used for measurements
- Vanishing points



Perspective Viewing



Type of perspective depends on the number of principal axes cut by the projectors.



3-point

2-point

1-point

Outline

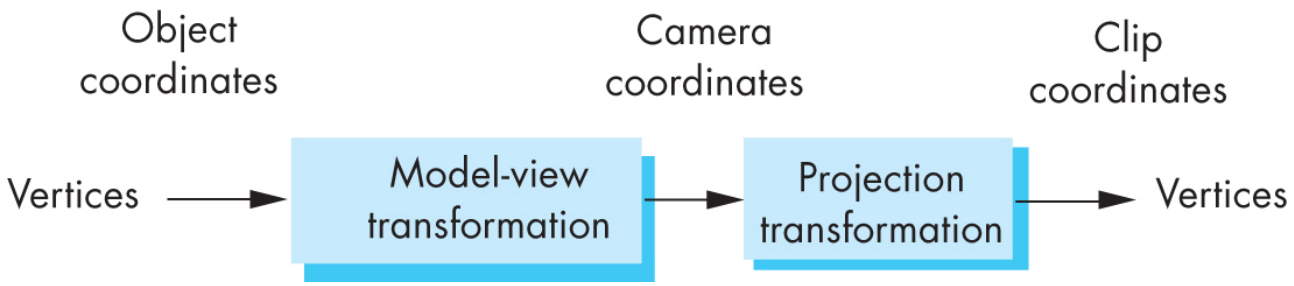
- 2D Viewing
- 3D Viewing
  - Classic view
  - **Computer view**
    - Positioning the camera
    - Projection

The fundamental difference between the classic view and computer view:

- All the classical views are based on a particular relationship among the objects (对象), the viewers (观察者), and the projectors (投影线).
- In computer graphics, we stress the independence of the object specifications (对象定义) and camera parameters (摄像机参数).

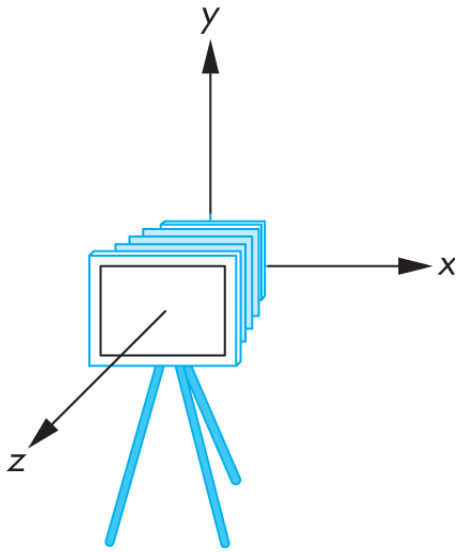
Computer view

- The view has three functions, are implemented in pipeline system
- Positioning the camera
  - Setup the model-view matrix
- Set the lens
  - Projection matrix
- Clipping
  - view frustum



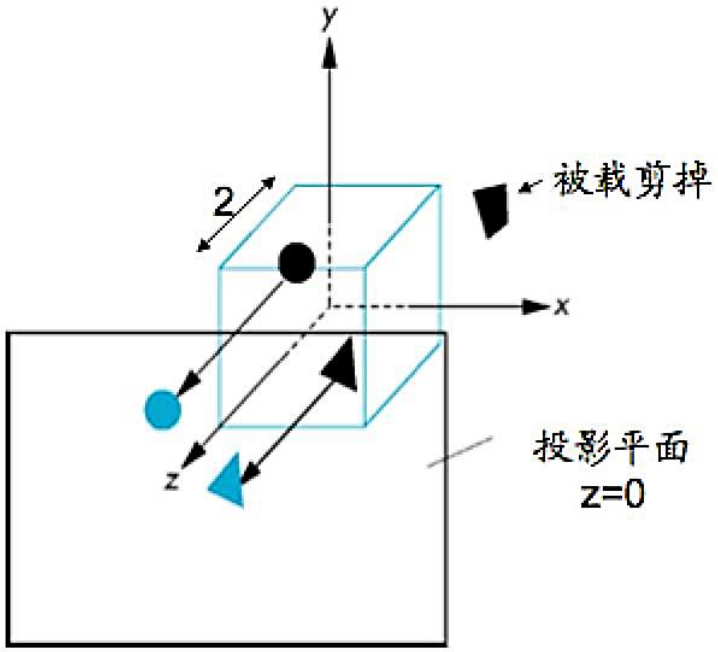
Camera in OpenGL

- In OpenGL, the initial world frame and camera frame are the same
- A camera located at the origin, and point to the negative direction of Z axis
- OpenGL also specifies the view frustum default, it is a center at the origin of the side length of 2 cube



Default projection

- Default is the projection of orthogonal projection



The definition of a visible object

- In the default camera settings, in order to make the definition of the object is visible, only to make the position and size of the object with the default view frustum matched
  - Usually, the data can be an appropriate **translation** and **isotropic scaling**
  - Note that this is not the use of OpenGL translation and scaling function operation

Outline

- 2D Viewing
- 3D Viewing
  - Classic view
  - Computer view
    - **Positioning the camera**
    - Projection

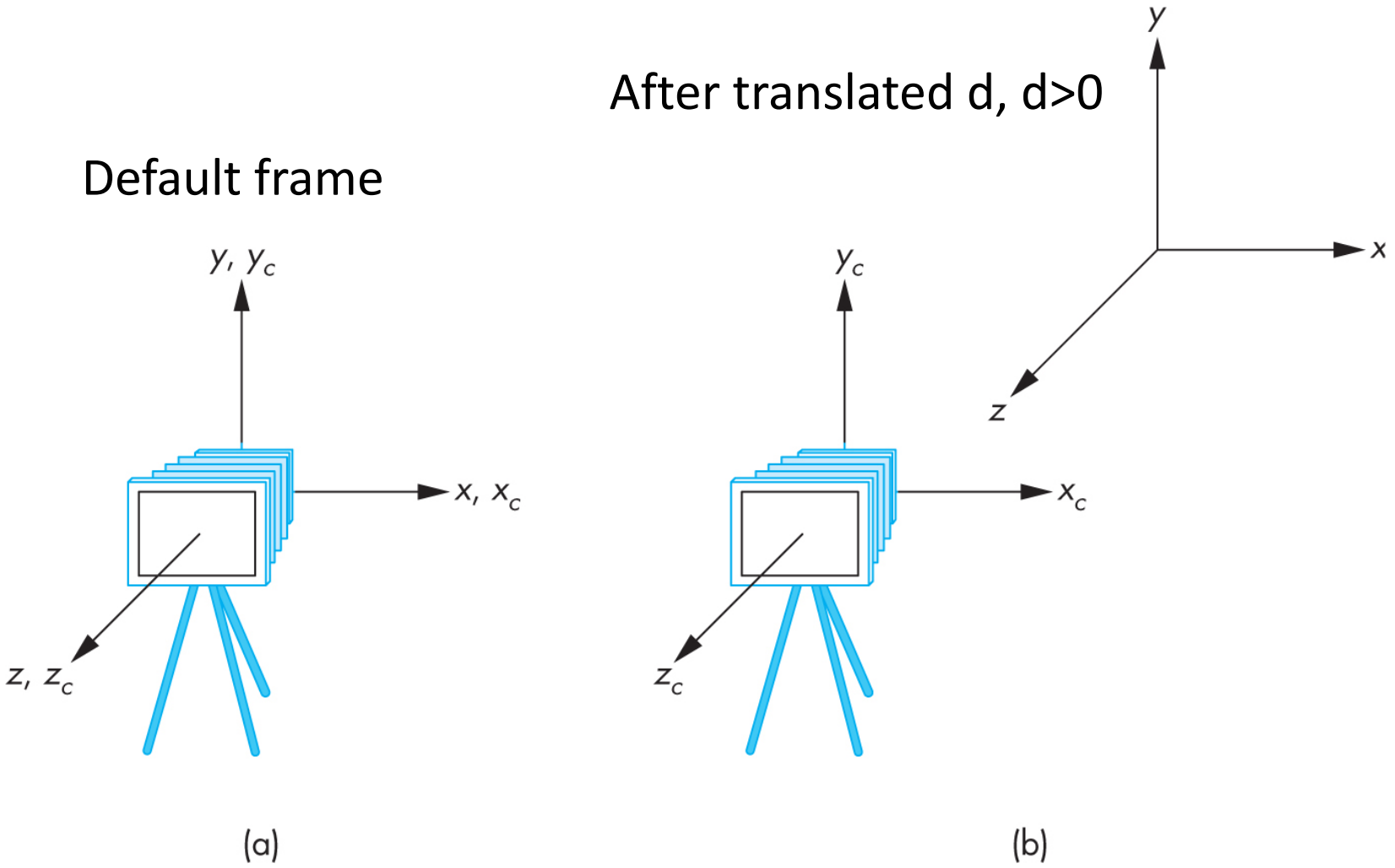
Moving the camera frame

- If you want to see objects with positive Z coordinate more, we can
  - Moves The camera along the positive Z axis
  - Moves the object along the negative Z axis
- They are equivalent, is determined by the model view matrix
  - Need a translation: glTranslated(0.0, 0.0, d);
  - Here, d > 0

Moving the camera frame

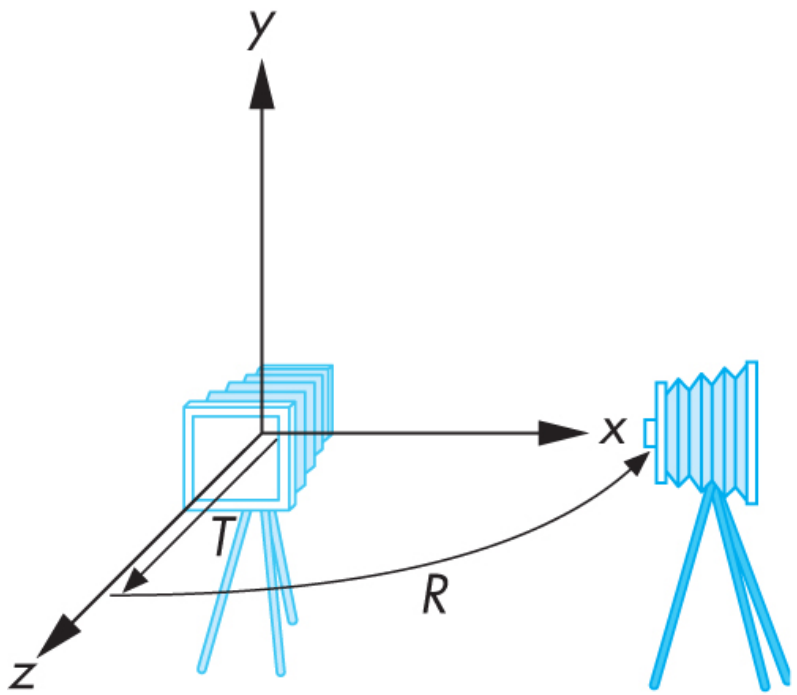
Default frame

After translated d, d>0



Moving the camera frame

- Can use a series of translation and rotation to the camera position to any position
- For example, in order to get the side view
  - Rotate the camera: R
  - Move the camera from the origin: T
  - C = TR



Viewing Specification

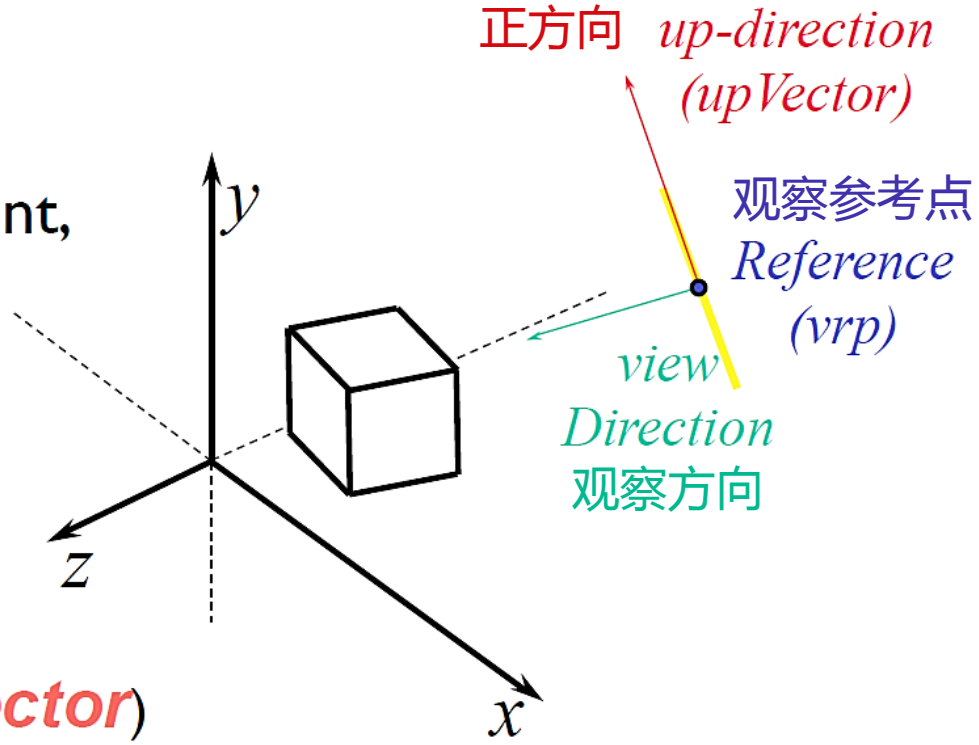
Specify

Focus point or reference point, typically on the object

(***view reference point***)

direction of viewing (***viewDirection***)

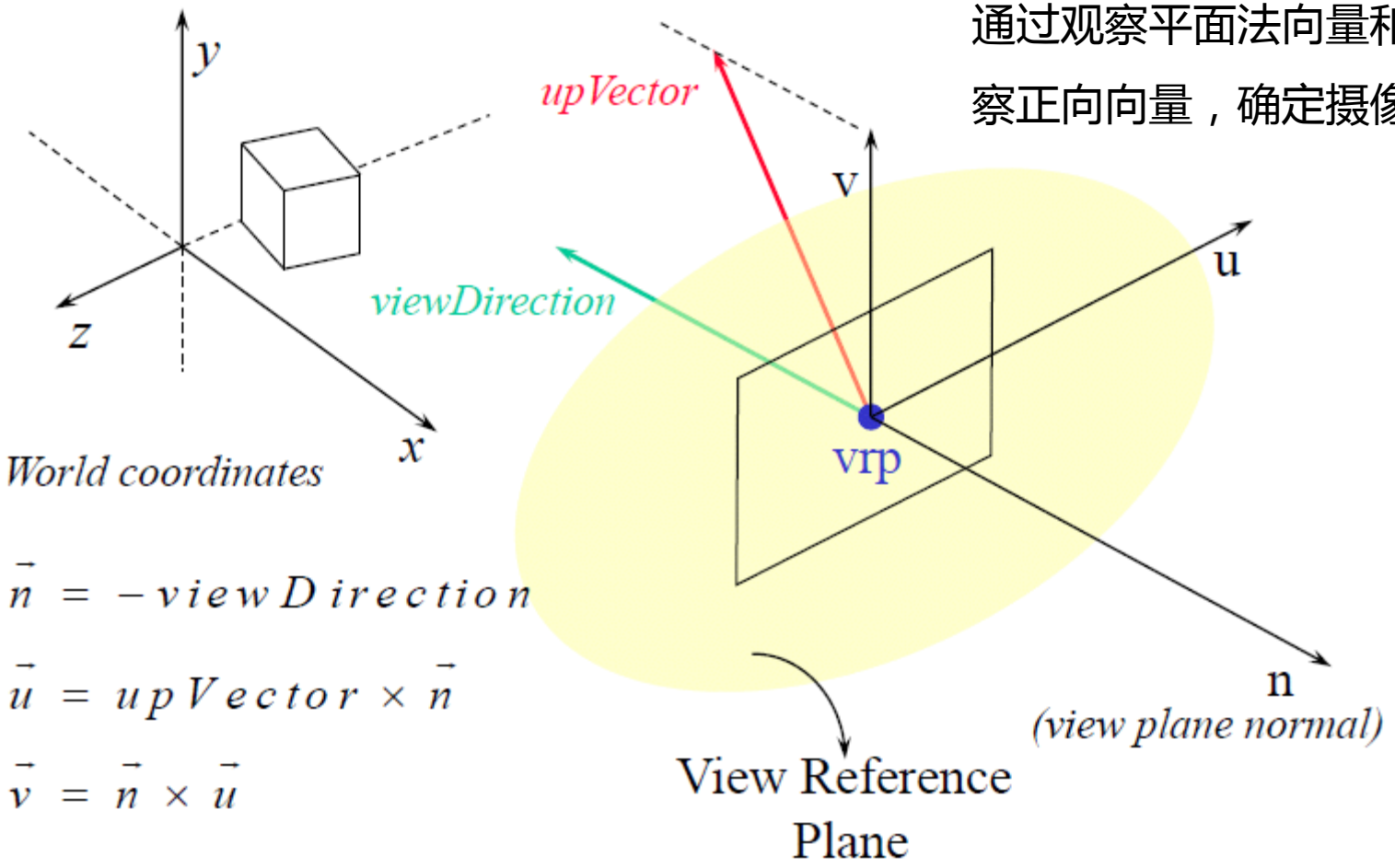
picture's up-direction (***upVector***)



All the specifications are in ***world coordinates***

View Reference Coordinate System

通过观察平面法向量和观察正向向量，确定摄像机。



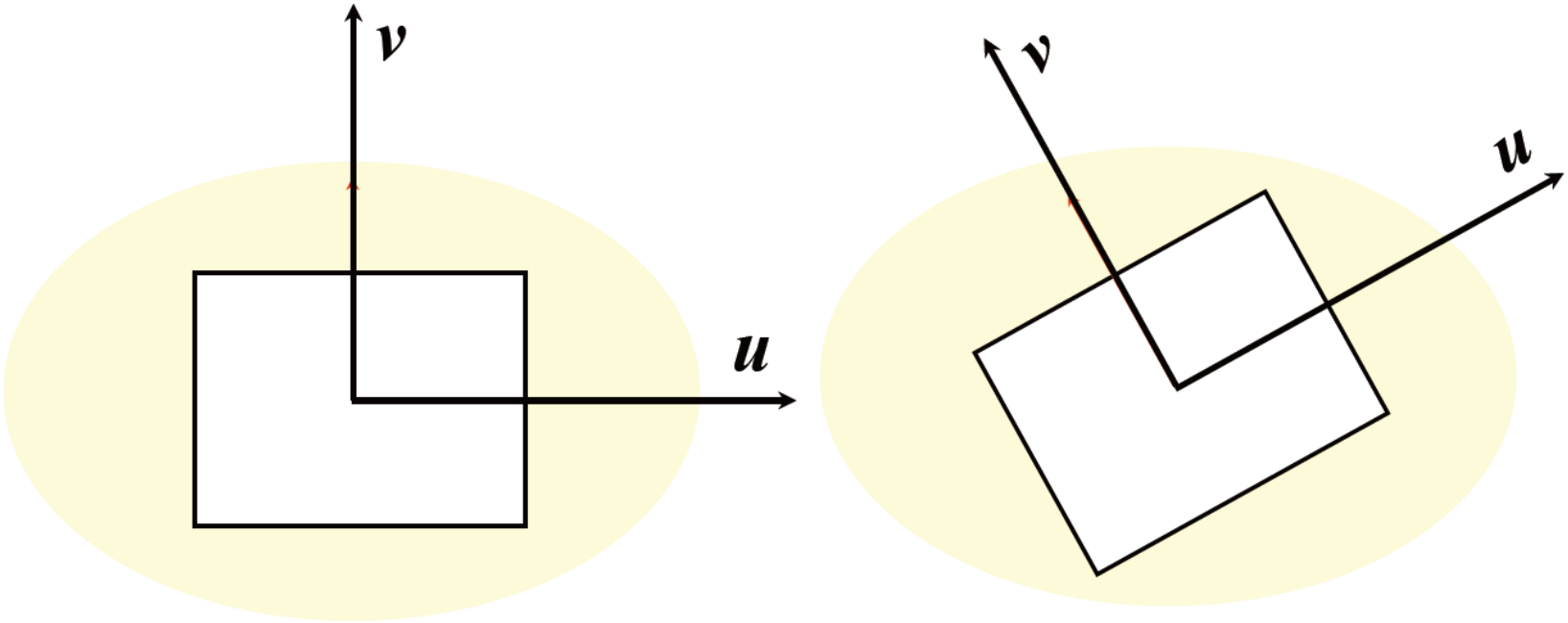
$$\vec{n} = -viewDirection$$

$$\vec{u} = upVector \times \vec{n}$$

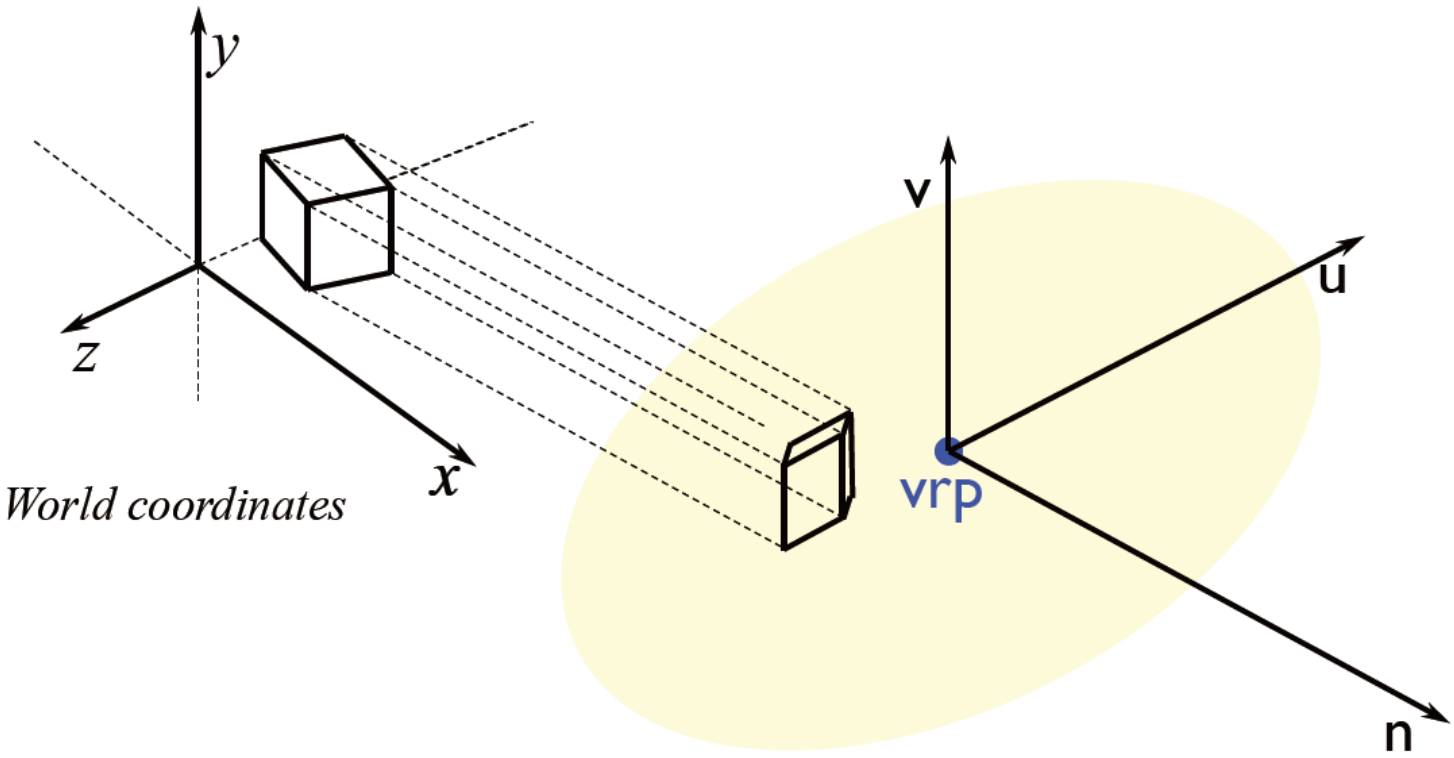
$$\vec{v} = \vec{n} \times \vec{u}$$

View Up Vector

- ***upVector*** decides the orientation of the *view window* on the *view reference plane*



View Reference Coordinate System

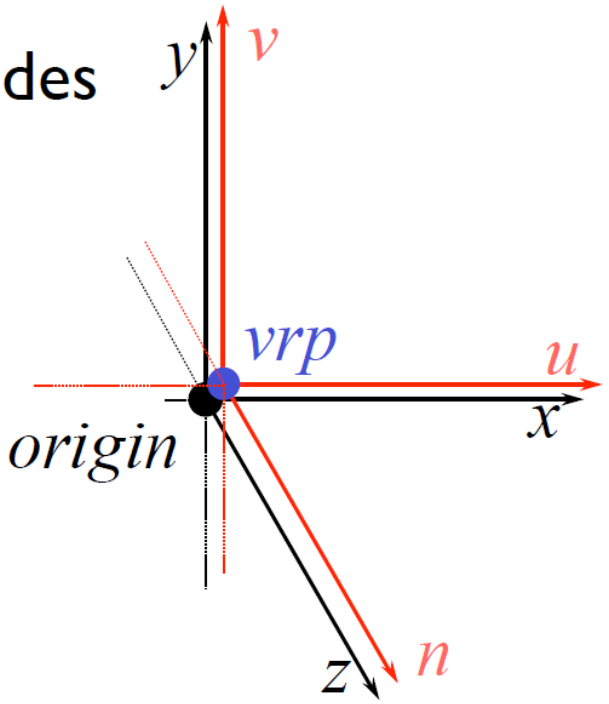


- Once the *view reference coordinate system* is defined, the next step is to project the *3D world* on to the *view reference plane*

Simplest Camera Position

- Projecting on to an arbitrary view plane looks tedious
- One of the simplest camera positions is one where ***vrp*** coincides with the ***world origin*** and

  u,v,n matches ***x,y,z***

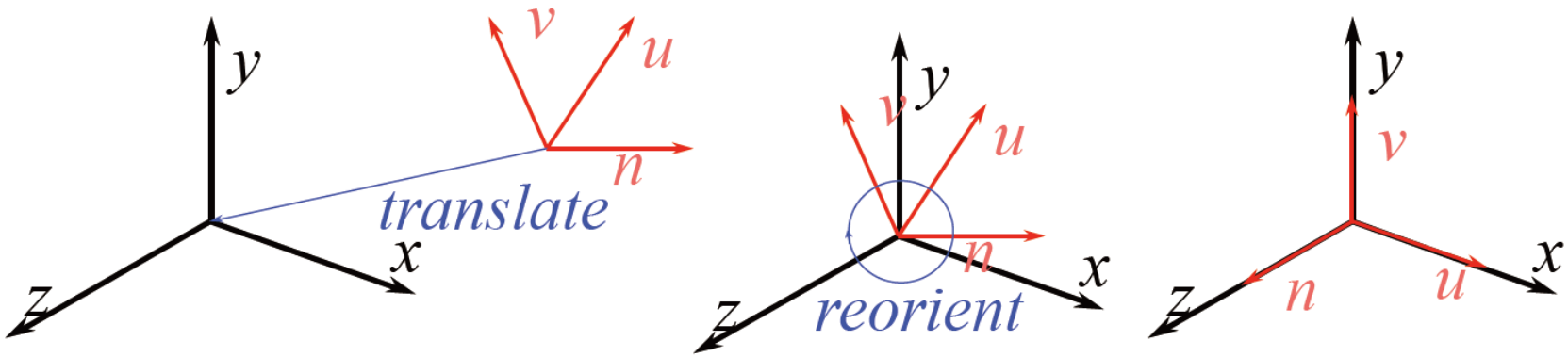


- Projection could be as simple as ignoring the z-coordinate

World to Viewing coordinate Transformation

- The world could be transformed so that the view reference coordinate system coincides with the world coordinate system
- Such a transformation is called world to viewing coordinate transformation
- The transformation matrix is also called view orientation matrix

Deriving View Orientation Matrix



- The ***view orientation matrix*** *transforms* a point from *world coordinates* to *view coordinates*

$$\begin{bmatrix} u_x & u_y & u_z & -\mathbf{r}_u \bullet vrp \\ v_x & v_y & v_z & -\mathbf{r}_v \bullet vrp \\ n_x & n_y & n_z & -\mathbf{r}_n \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A More Intuitive Approach Offered by GLU

- OpenGL provides a very helpful utility function that implements the look-at viewing specification:

```
gluLookAt ( eyex, eyey, eyez,  // eye point
            atx,  aty,  atz,   // lookat point
            upx,  upy,  upz ); // up vector
```

- These parameters are expressed in world coordinates

OpenGL Viewing Transformation

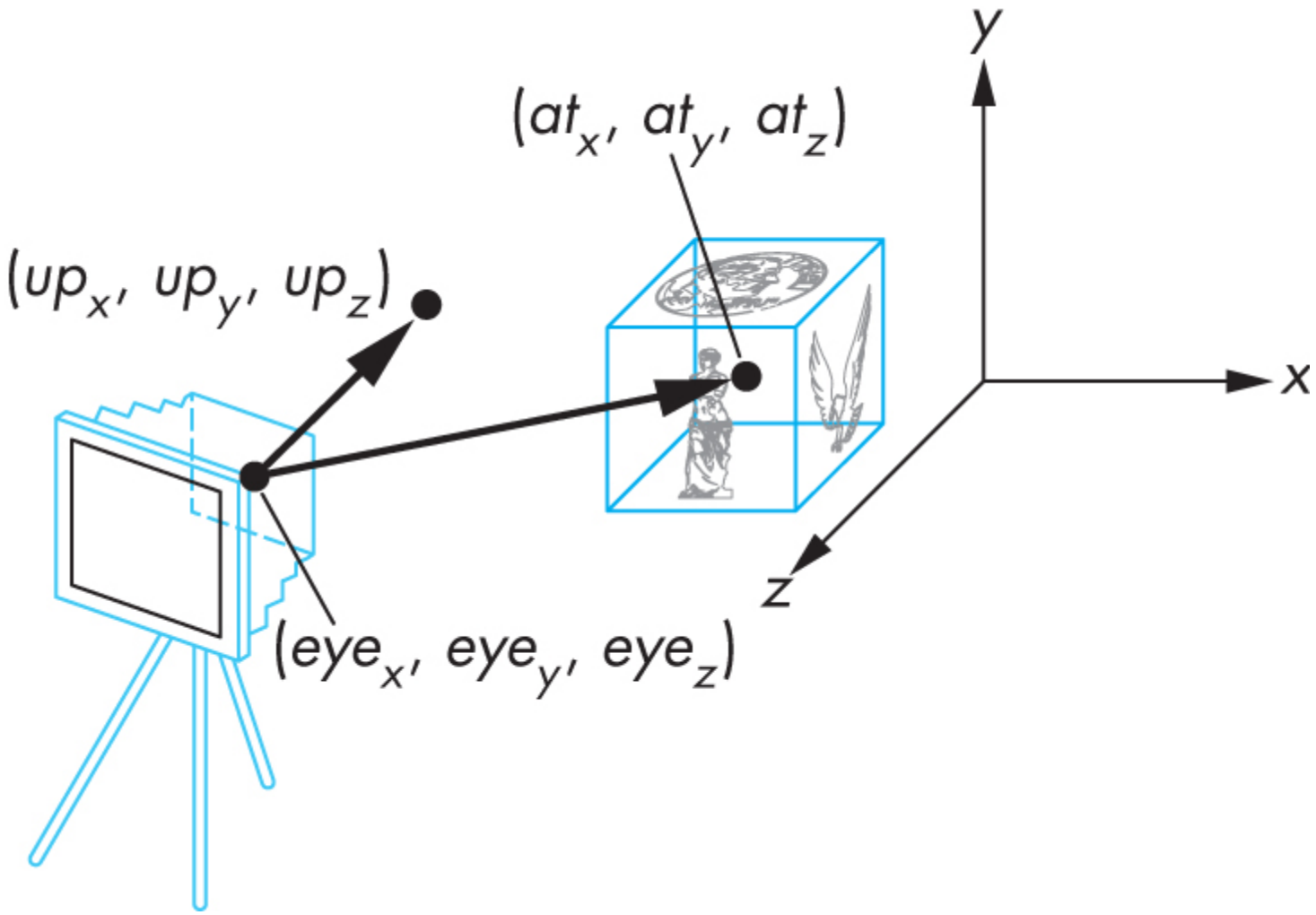
```
gluLookAt(ex,ey,ez,lx,ly,lz,ux,uy,uz)
```

- postmultiplies current matrix, so to be safe:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(ex,ey,ez,lx,ly,lz,ux,uy,uz)
// now ok to do model transformations
```

它封装了世界坐标系到观察坐标系的转换。调用之后，我们就把坐标系变换的矩阵放入了矩阵栈，后续对物体的位置描述，会通过此矩阵栈进行转换到我们的观察坐标系了。

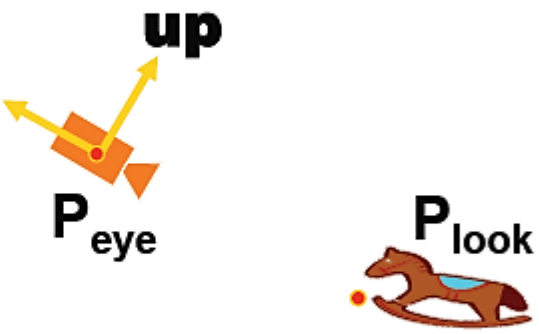
gluLookAt Illustration



Look-At Positioning

- We specify the view frame using the look-at vector **a** and the camera up vector **up**
- The vector **a** points in the negative viewing direction

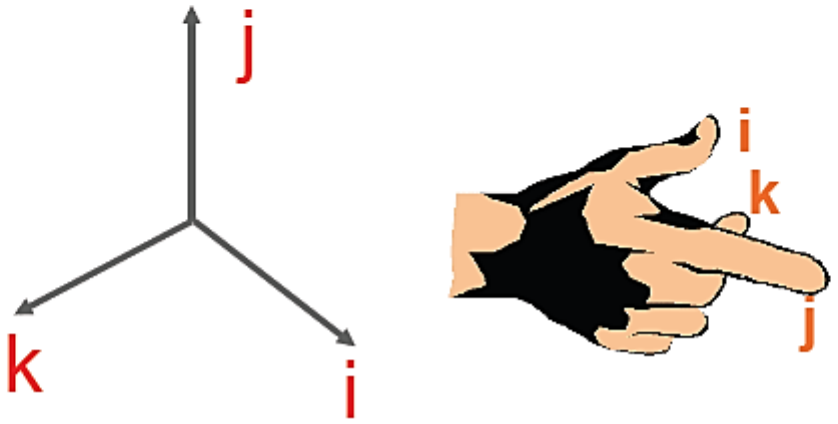
$$\mathbf{a} = P_{eye} - P_{look}$$



- In 3D, we need a third vector that is perpendicular to both **up** and **a** to specify the view frame

Where does it point to?

- The result of the cross product is a vector, not a scalar, as for the dot product
- In OpenGL, the cross product **a** × **b** yields a RHS vector. **a** and **b** are the thumb and index fingers, respectively

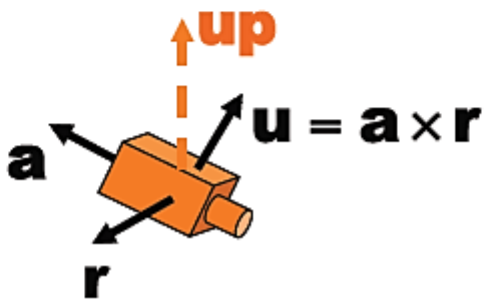


Constructing a Coordinates

- The cross product between the up and the look-at vector will get a vector that points to the right.

$$\mathbf{r} = \mathbf{up} \times \mathbf{a}$$

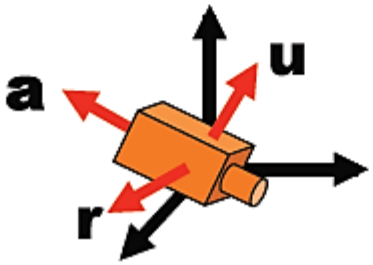
- Finally, using the vector **a** and the vector **r** we can synthesize a new vector **u** in the up direction:



Rotation

- Rotation takes the unit world frame to our desired camera frame:

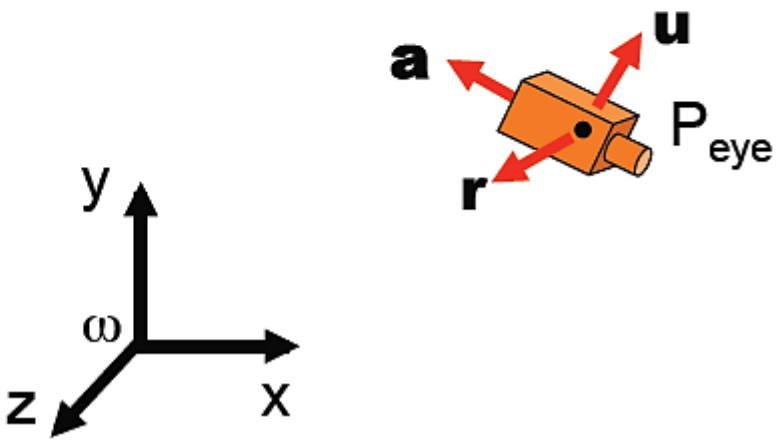
$$\begin{bmatrix} r_x & u_x & a_x & 0 \\ r_y & u_y & a_y & 0 \\ r_z & u_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{R}$$



Translation

- Translation to the eye point:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & eye_x \\ 0 & 1 & 0 & eye_y \\ 0 & 0 & 1 & eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Composing the Result

- The final camera transformation is:

$$\mathbf{E} = \mathbf{TR} = \begin{bmatrix} 1 & 0 & 0 & eye_x \\ 0 & 1 & 0 & eye_y \\ 0 & 0 & 1 & eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_x & u_x & a_x & 0 \\ r_y & u_y & a_y & 0 \\ r_z & u_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Viewing Transformation

- Transforming all points P in the world with $\mathbf{E}^{-1}$:

$$\mathbf{V} = \mathbf{R}^{-1}\mathbf{T}^{-1} = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ a_x & a_y & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Where these are normalized vectors:

$$\mathbf{a} = P_{eye} - P_{look}$$

$$\mathbf{r} = \mathbf{up} \times \mathbf{a}$$

$$\mathbf{u} = \mathbf{a} \times \mathbf{r}$$

Looking At a cube

- Setting up the OpenGL look-at viewing transformation:

```
void display(void){
        glClear(GL_COLOR_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        // Setting up the view
        gluLookAt(
                0.0, 0.0, 5.0,     // Eye is at (0,0,5)
                0.0, 0.0, 0.0,     // Center is at (0,0,0)
                0.0, 1.0, 0.);     // Up is in positive Y direction
        // Now we are using the world frame
        // Draw Object
        glColor3f (1.0, 1.0, 1.0);
        glutWireCube(1.0);
        glutSwapBuffers();
}
```

Model/View Transformation

- Combine modeling and viewing transform
  - Combine into single matrix
  - Saves computation time
    - if many points are to be transformed
  - Possible because viewing transformation directly follows modeling transformation without intermediate operations

gluLookAt() and other transformations

- The user can define the model-view matrix to achieve the same function
- But from the concept of the gluLookAt () as the camera position, while the other follow-up transformation as object position
- gluLookAt in the OpenGL () function is **the only specialized** for positioning the camera function

Outline

- 2D Viewing
- 3D Viewing
  - Classic view
  - Computer view
    - Positioning the camera
    - **Projection**

Orthogonal Projection

1. Apply the world to view transformation
2. Apply the parallel projection matrix to project the 3D world onto the view plane

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & u_y & u_z & -\vec{u} \bullet vrp \\ v_x & v_y & v_z & -\vec{v} \bullet vrp \\ n_x & n_y & n_z & -\vec{n} \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Apply 2D viewing transformations to map the view window on to the screen

Orthogonal Projection Matrix: Homogeneous coordinates

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

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

$$x_p = x$$

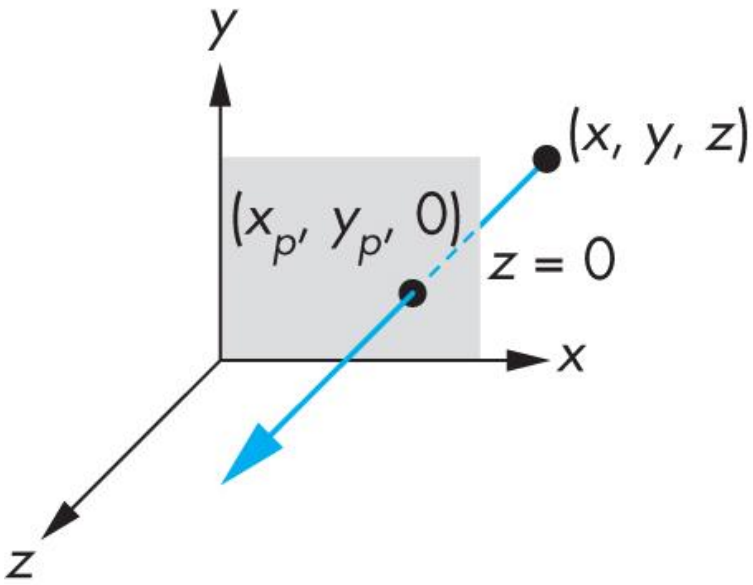
$$y_p = y$$

$$z_p = 0$$

$$w_p = 1$$

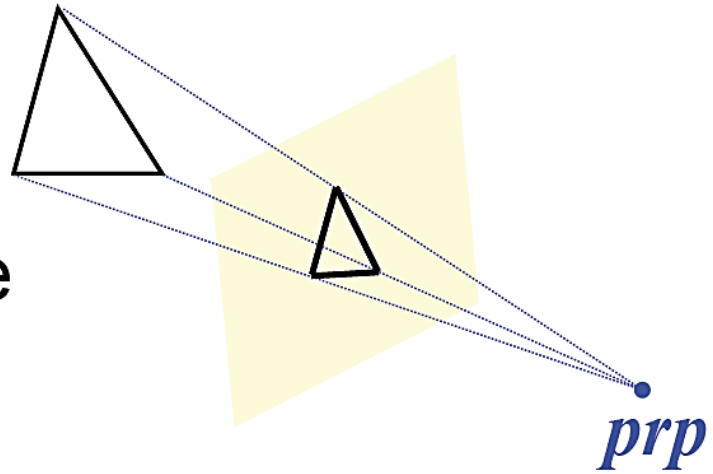
在实际应用中可以令M = I, 然后把对角线第三个元素置为零。

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$



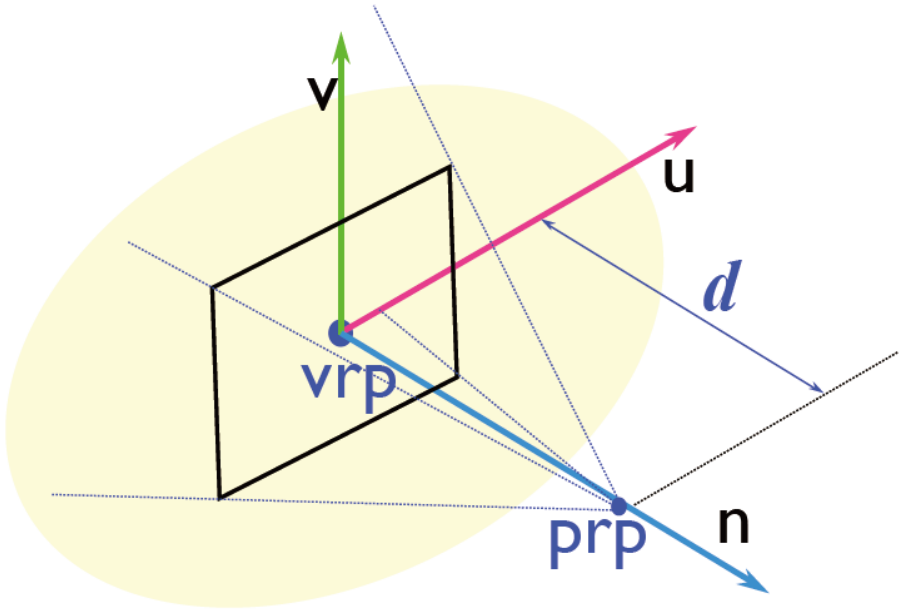
Perspective Projection

- The points are transformed to the view plane along lines that converge to a point called
  - ***projection reference point*** (***prp***) or
  - ***center of projection*** (***cop***)
- ***prp*** is specified in terms of the viewing coordinate system



Transformation Matrix for Perspective Projection

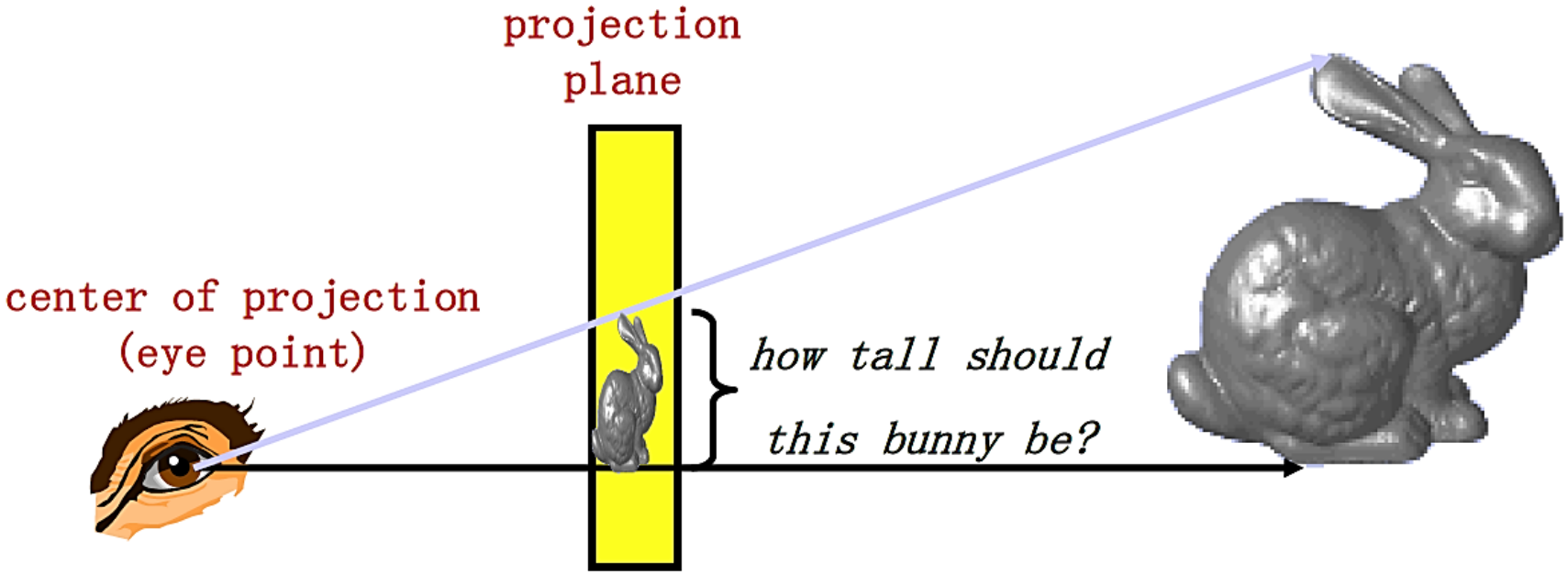
- ***prp*** is usually specified as perpendicular distance ***d*** behind the view plane



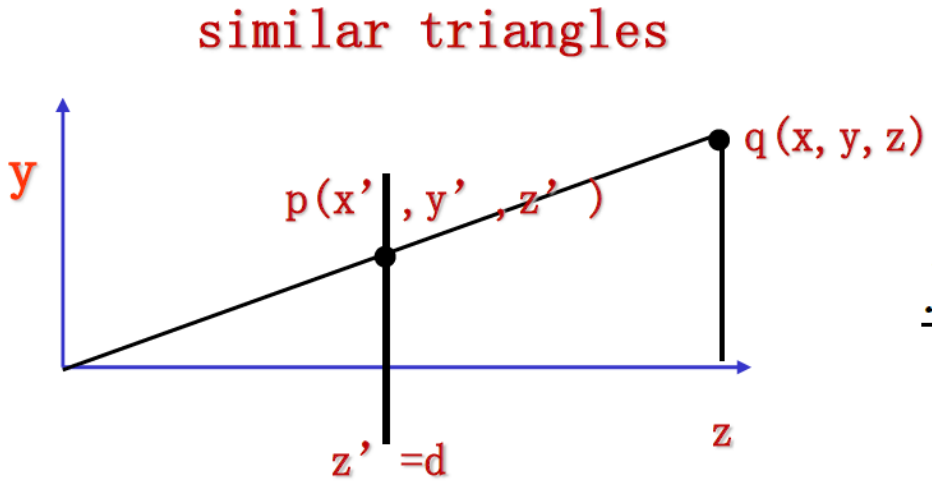
transformation matrix
for *perspective projection*

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

Perspective Projection



Basic Perspective Projection



$$\frac{y'}{d} = \frac{y}{z} \rightarrow y' = \frac{y \cdot d}{z}$$

$$\frac{x'}{d} = \frac{x}{z} \rightarrow x' = \frac{x \cdot d}{z}$$

but $z' = d$

Given p = Mq, write out the Projection Matrix M.

Homogeneous Coordinates

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

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \qquad \mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Divide

- 如果$w \neq 1$，那么必须从齐次坐标中除以w而得到所表示的点
- 这就是透视除法，结果为

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

上述方程称为透视方程。

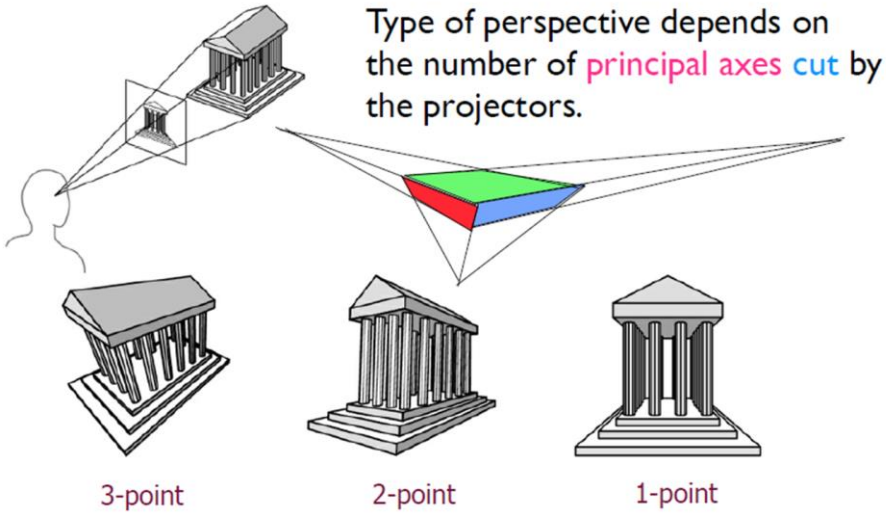
- 后面会用OpenGL函数考虑相应的裁剪体。

注：对于一个普通坐标的点 P=(Px, Py, Pz) ，有对应的一族齐次坐标 (wPx, wPy, wPz, w)，其中 w 不等于零 。

比如， P(1, 4, 7) 的齐次坐 标有 (1, 4, 7, 1) 、(2, 8, 14, 2)、(-0.1, -0.4, -0.7, -0.1) 等等 。

Perspective Projection

- Perspective Divide是非线性的，导致非均匀缩短。
  - 离投影中心（COP）远的对象投影后，尺寸缩短得比离COP近的对象大。
- 透视变换是**保直线**的，但不是**仿射变换**。
- 透视变换是不可逆的，因为沿一条投影直线上的所有点投影后的结果相同。



Parallel Viewing

1. Apply the world to view transformation
2. Apply the parallel projection matrix to project the 3D world onto the view plane

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & u_y & u_z & -\overset{\mathrm{r}}{u} \bullet vrp \\ v_x & v_y & v_z & -\overset{\mathrm{r}}{v} \bullet vrp \\ n_x & n_y & n_z & -\overset{\mathrm{r}}{n} \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Apply 2D viewing transformations to map the view window on to the screen

Perspective Viewing

1. Apply the view orientation transformation
2. Apply translation, such that the center of the view window coincide with the origin
3. Apply the perspective projection matrix to project the 3D world onto the view plane

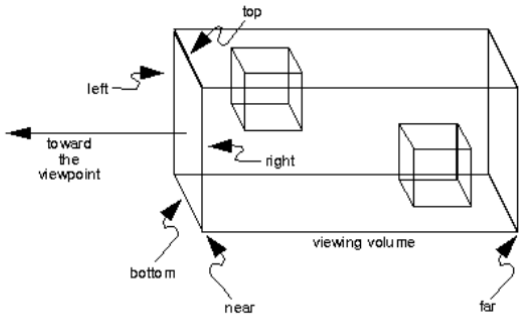
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & -cx \\ 0 & 1 & 0 & -cy \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & u_y & u_z & -\vec{u} \bullet vrp \\ v_x & v_y & v_z & -\vec{v} \bullet vrp \\ n_x & n_y & n_z & -\vec{n} \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. Apply 2D viewing transformations to map the view window (centered at the origin) on to the screen

View in OpenGL

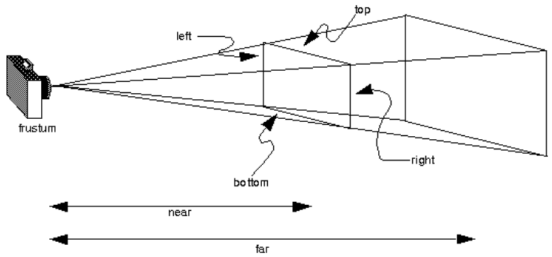
Orthogonal view

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



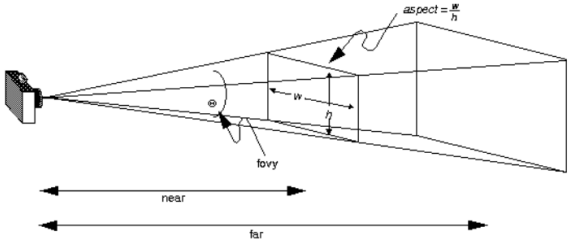
Perspective View

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




```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glFrustum(left, right, bottom, top, near, far);
```

gluPerspective(fovy, aspect, near, far);

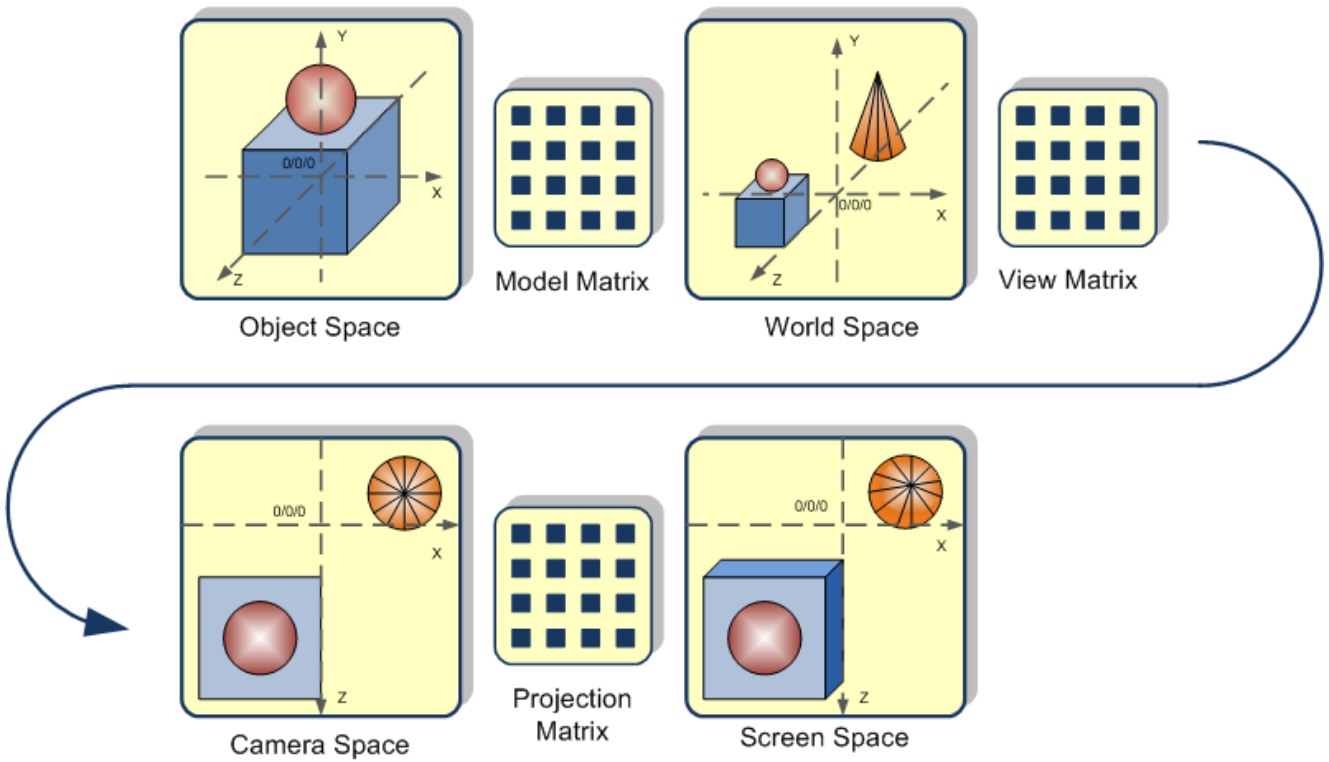


FOV is the angle between the top and bottom planes

Summary

- 实际上，从三维空间到二维平面，就如同用相机拍照一样，通常都要经历以下几个步骤 ：
  - 第一步，将相机置于三角架上，让它对准三维景物（观察(视点)变换，Viewing Transformation）。
  - 第二步，将三维物体放在适当的位置（模型变换，Modeling Transformation）。
  - 第三步，选择相机镜头并调焦，使三维物体投影在二维胶片上（投影变换，Projection Transformation）。
  - 第四步，决定二维像片的大小（视口变换，Viewport Transformation）。

Summary



1. Vertices of the Object to draw are in **Object space** (as modelled in your 3D Modeller)

2. … get transformed into World space by multiplying it with the **Model Matrix**

3. Vertices are now in **World space** (used to position the all the objects in your scene)

4. … get transformed into Camera space by multiplying it with the **View Matrix**

5. Vertices are now in **View Space** – think of it as if you were looking at the scene throught "the camera"

6. … get transformed into Screen space by multiplying it with the **Projection Matrix**

7. Vertex is now in **Screen Space** – This is actually what you see on your Display.