Computer Graphics 2014

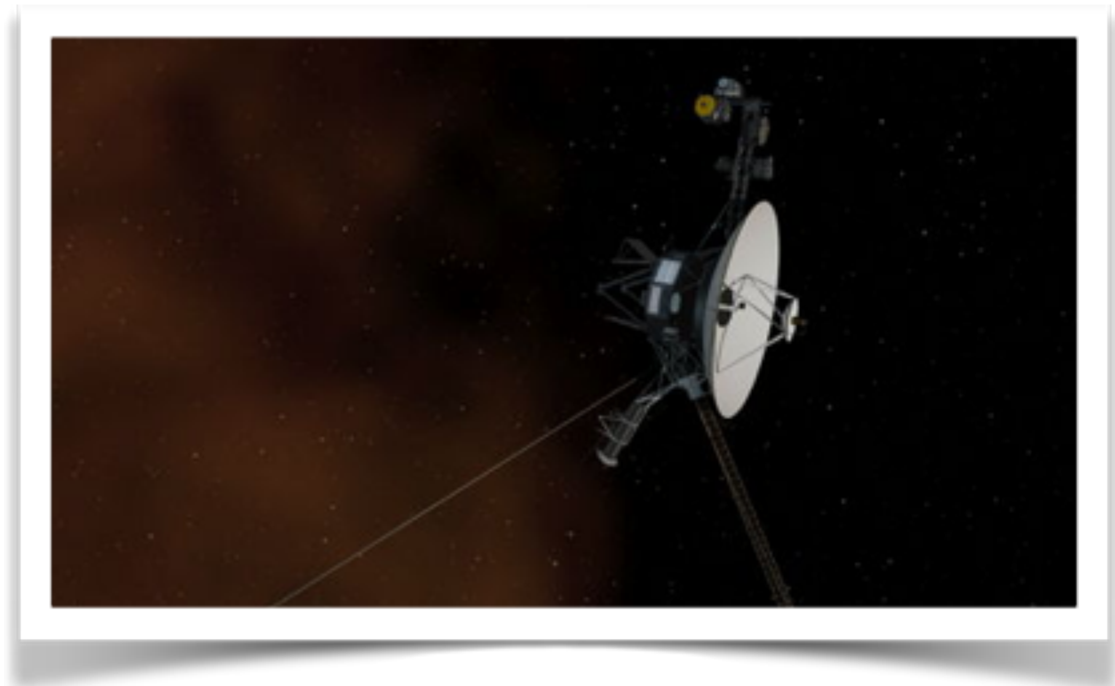
7. Viewing in 2D & 3D

Hongxin Zhang
State Key Lab of CAD&CG, Zhejiang University

2014-10-17

Homework 02

- Build a Solar System
  - requirement:
    - detailed computing steps
    - at least Earth, Moon and Sun
    - in OpenGL
  - bonus:
    - implemented demo
  - Deadline: 2014-10-24



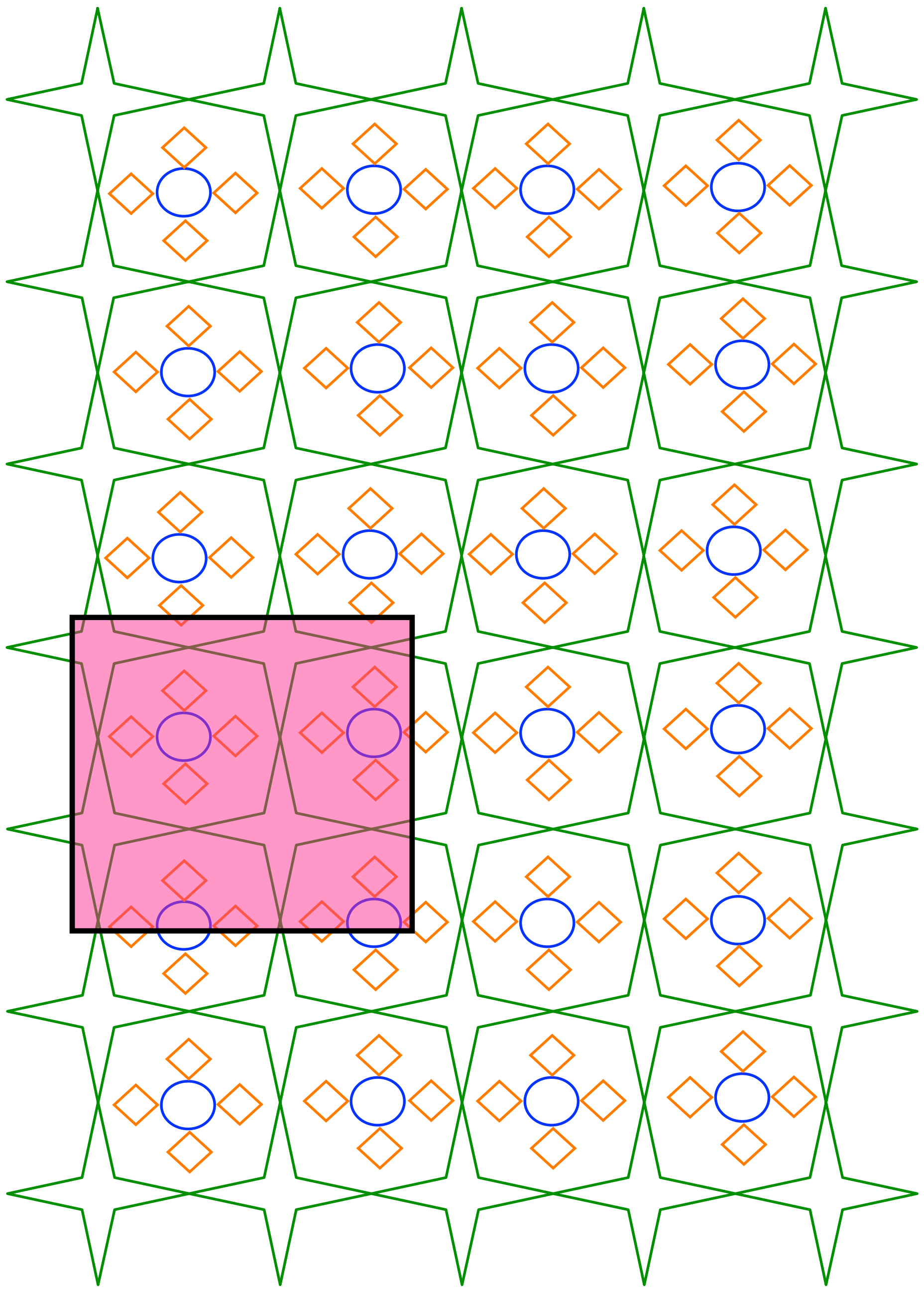
旅行者1号已迈进
星际空间

Contents

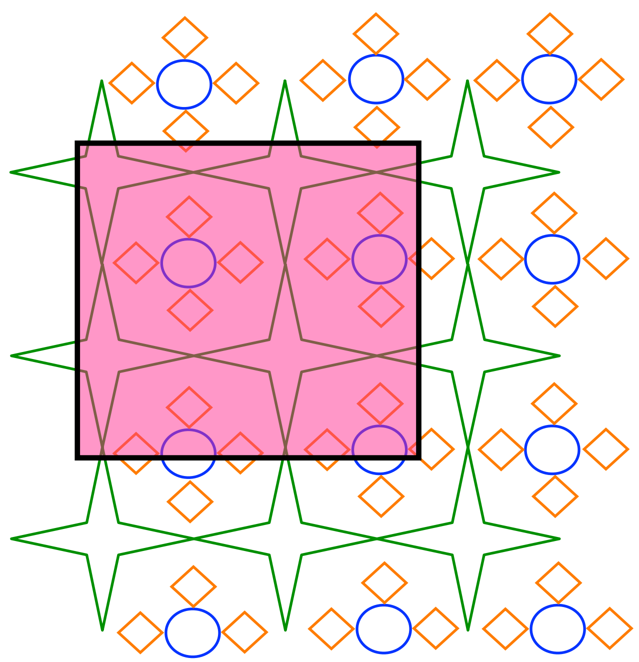
- 2D viewing
- 3D viewing

2D Viewing

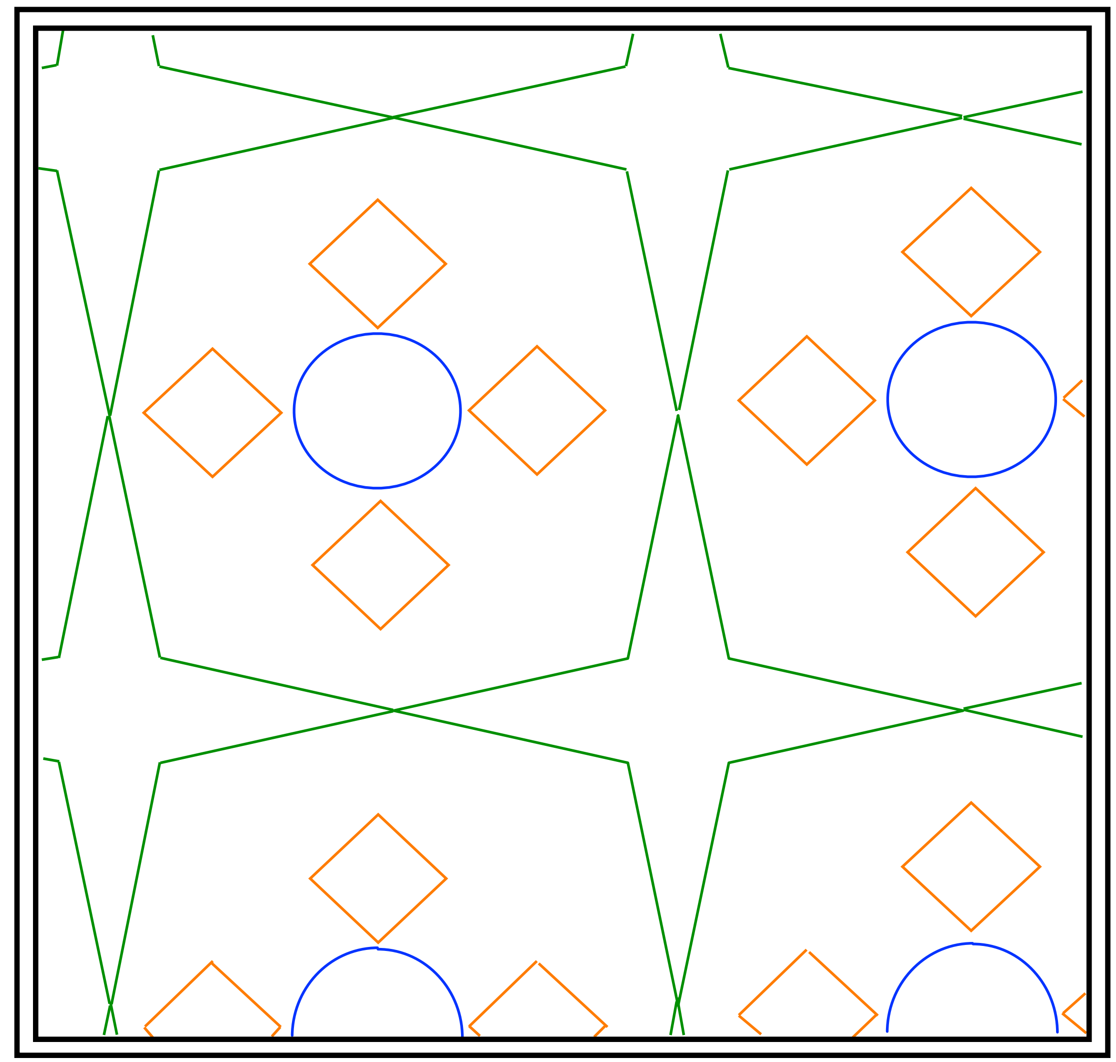
- The world is infinite (2D or 3D) but the screen is finite
- Depending on the details the user wishes to see, he limits his view by specifying a window in this world



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



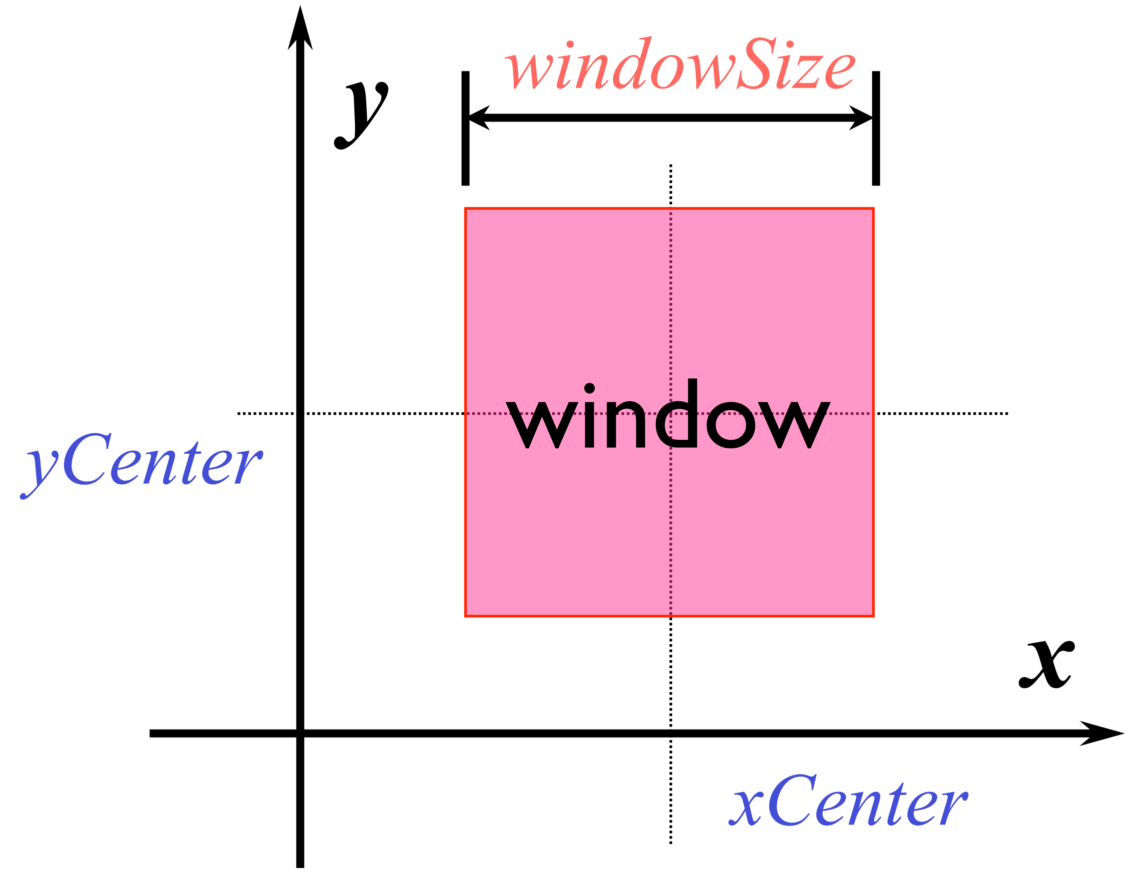
2D World

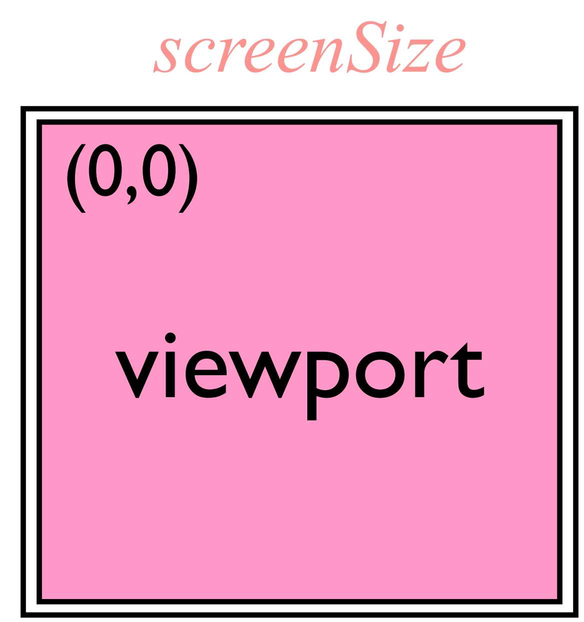


Screen

Windowing Concepts

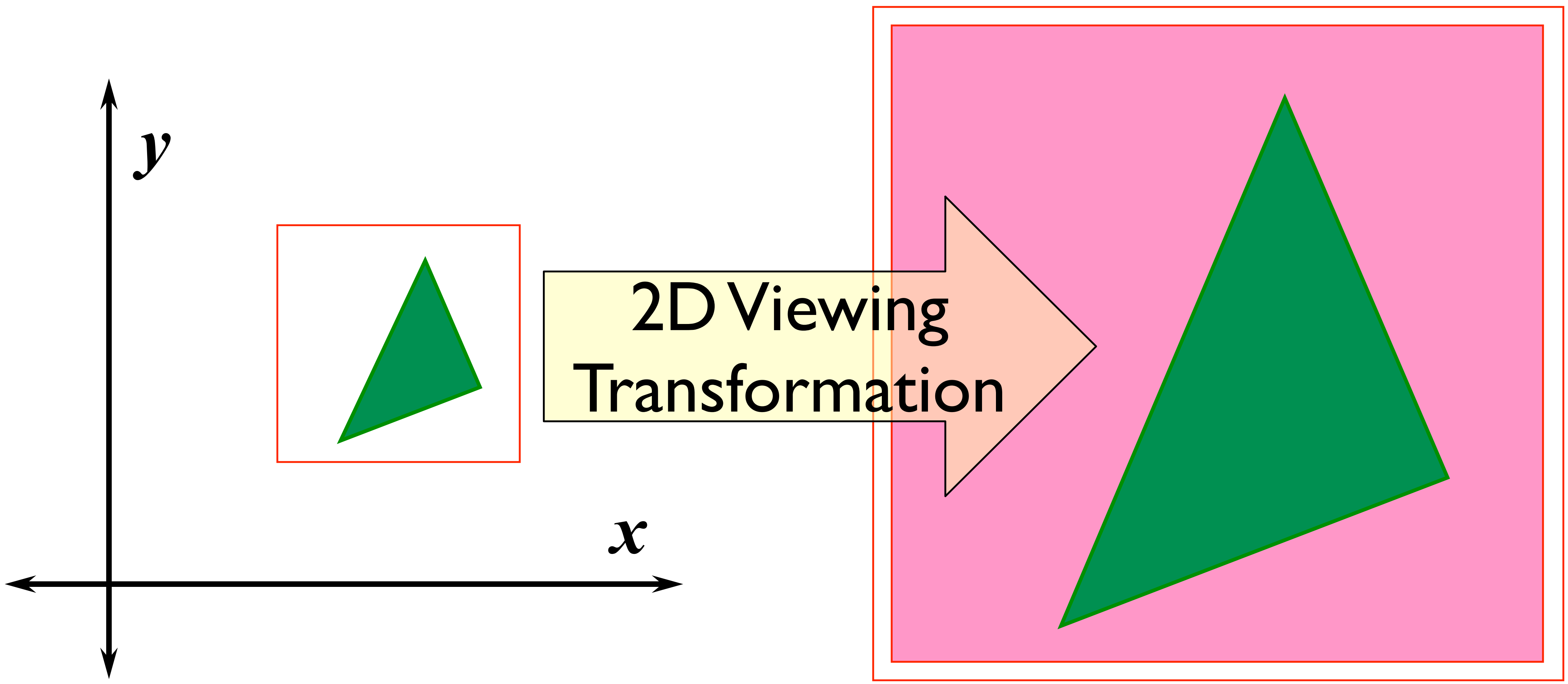
- ***Window*** is a rectangular region in the 2D world specified by
  - a ***center*** *(xCenter, yCenter)* and
  - ***size*** *windowSize*
- Screen referred to as ***Viewport*** is a discrete matrix of pixels specified by
  - ***size*** *screenSize* (in pixels)



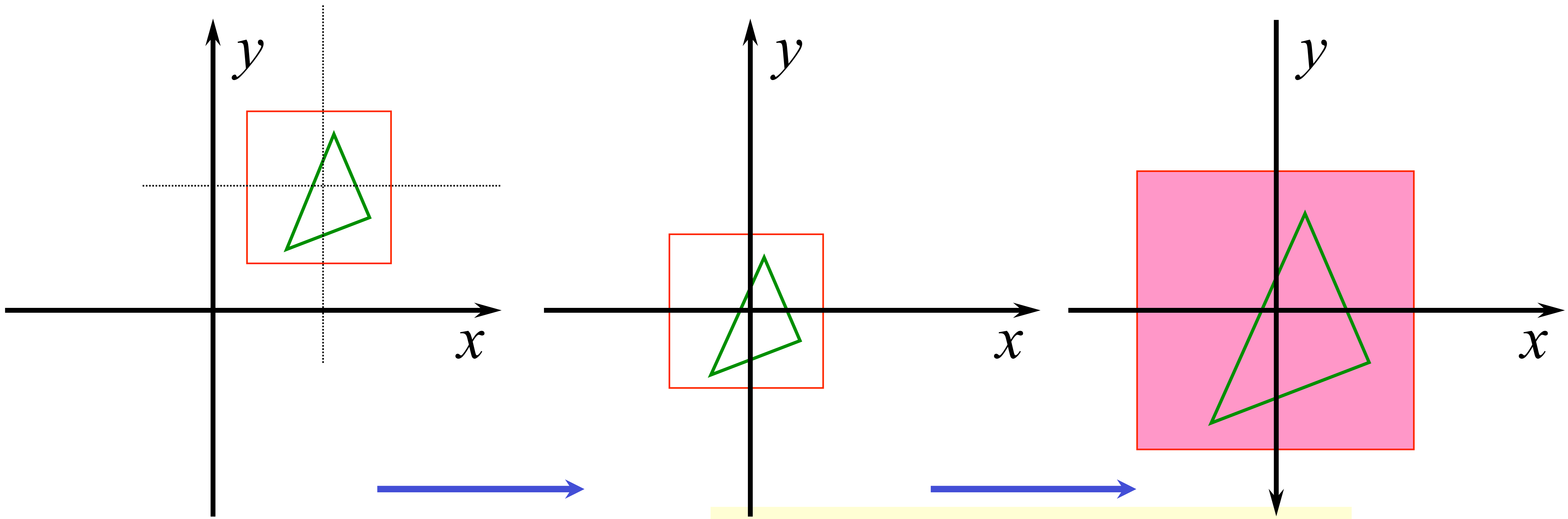


2D Viewing Transformation

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



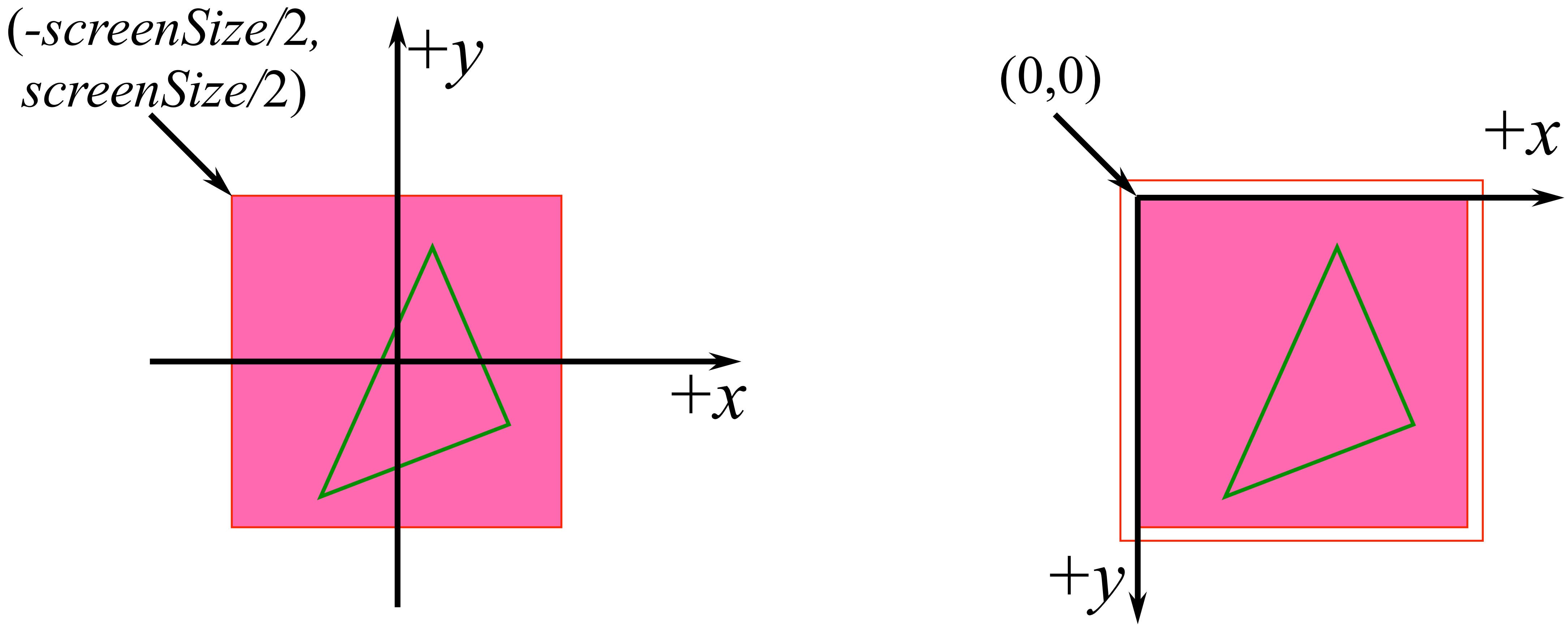
Deriving 2D Viewing Transformation



$$\begin{aligned} &x - xCenter \\ &y - yCenter \end{aligned}$$

$$\begin{aligned} &(x - xCenter) \bullet scaleFactor \\ &(y - yCenter) \bullet scaleFactor \end{aligned}$$

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

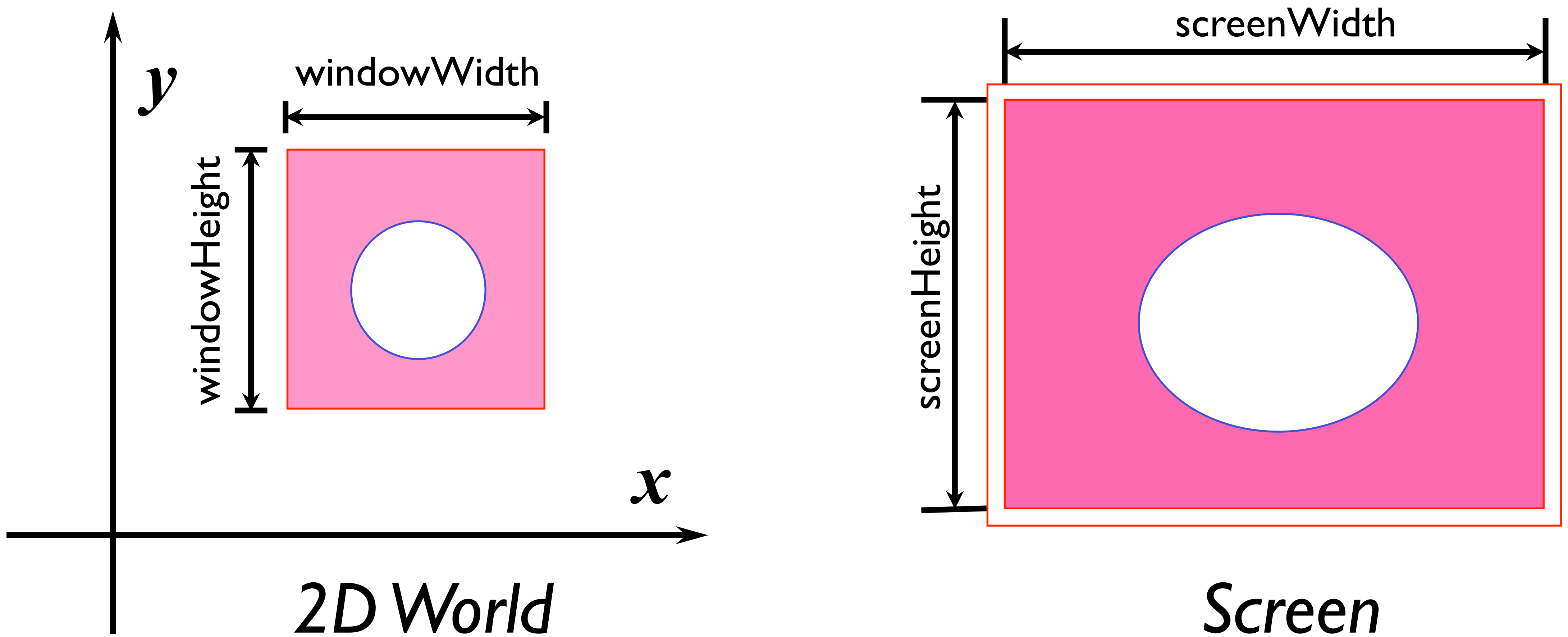


$$\frac{screenSize}{2} + (x - xCenter) \bullet scaleFactor$$
$$\frac{screenSize}{2} - (y - yCenter) \bullet scaleFactor$$

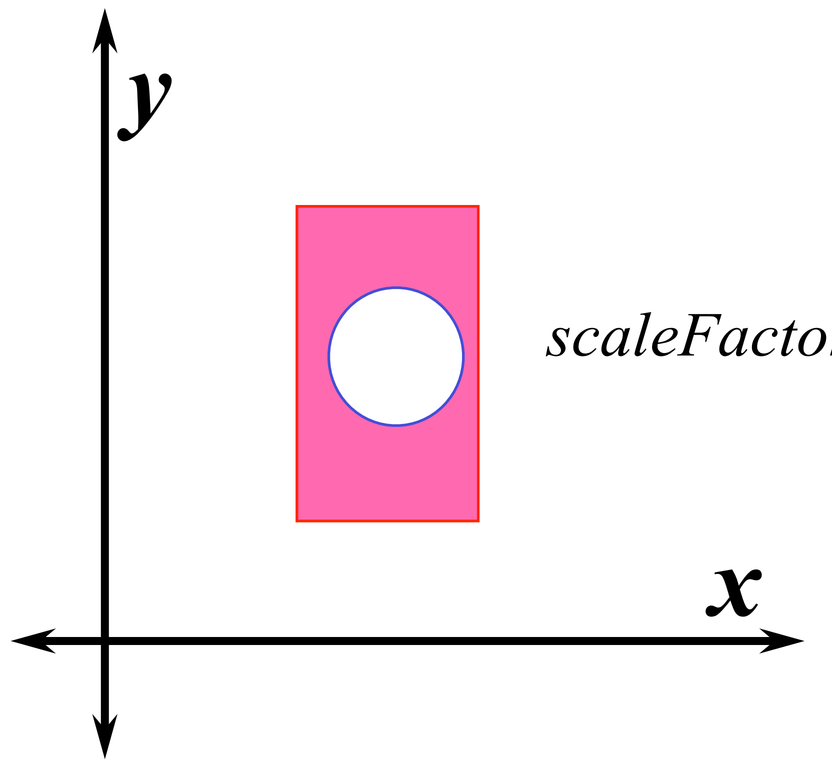
- Given any point in the 2D world, the above transformations maps that point on to the screen

The Aspect Ratio

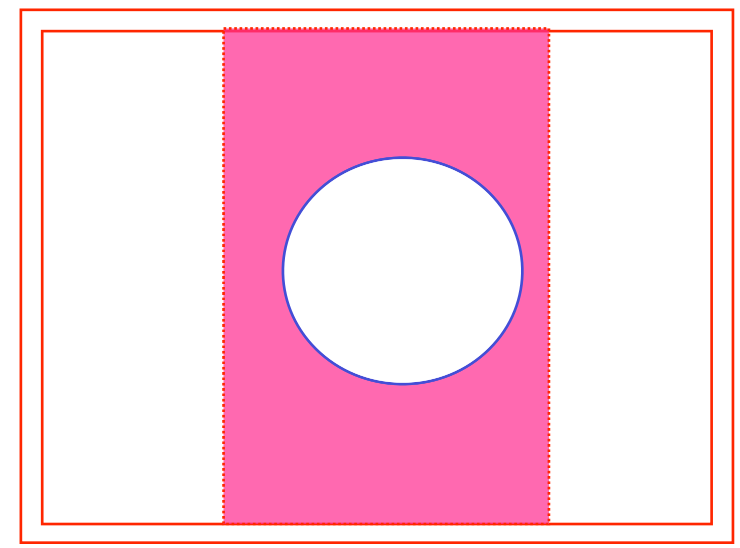
- In 2D viewing transformation the ***aspect ratio*** is maintained when the scaling is uniform
- ***scaleFactor*** is same for both x and y directions

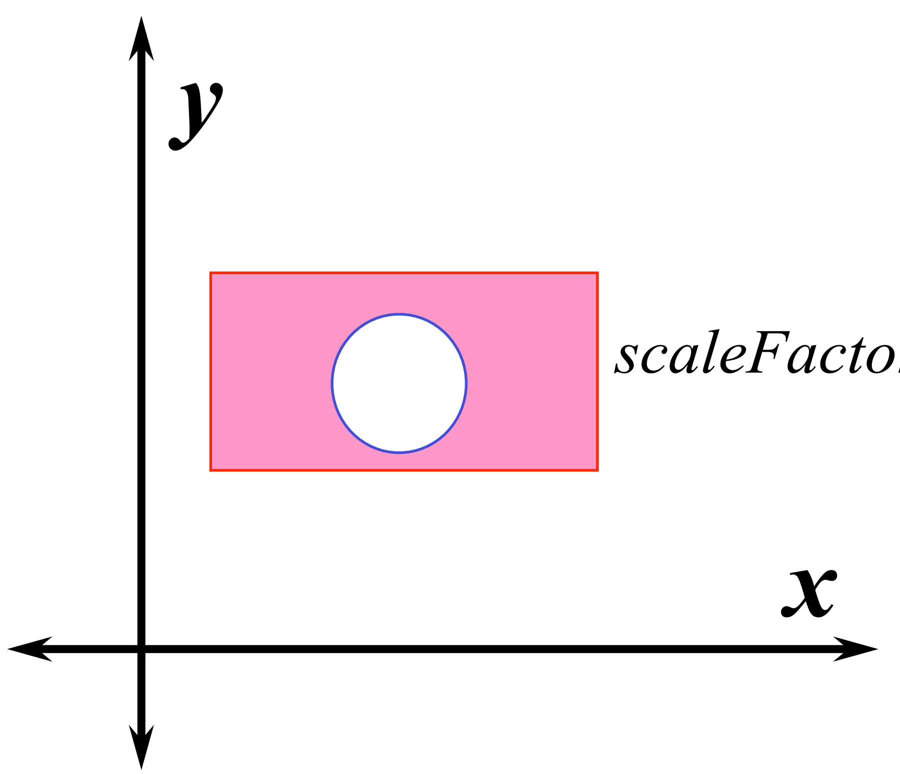


Maintaining Aspect Ratio

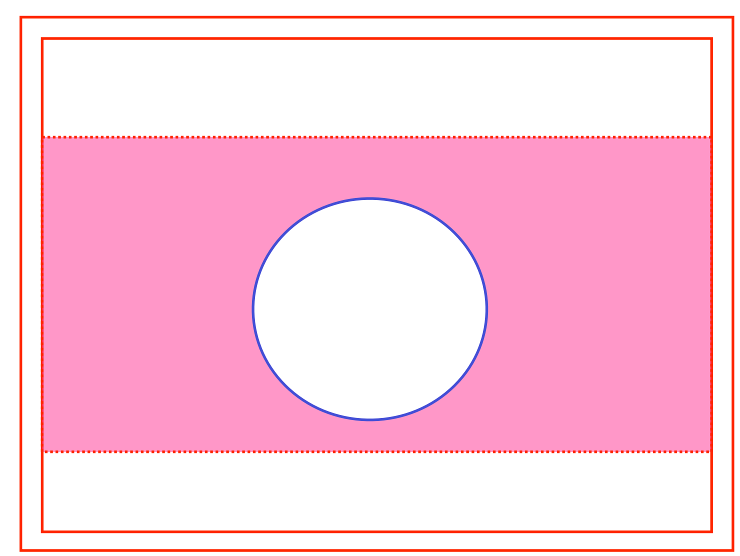


$$scaleFactor = \frac{screenHeight}{windowHeight}$$





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



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.

2D Rendering-1

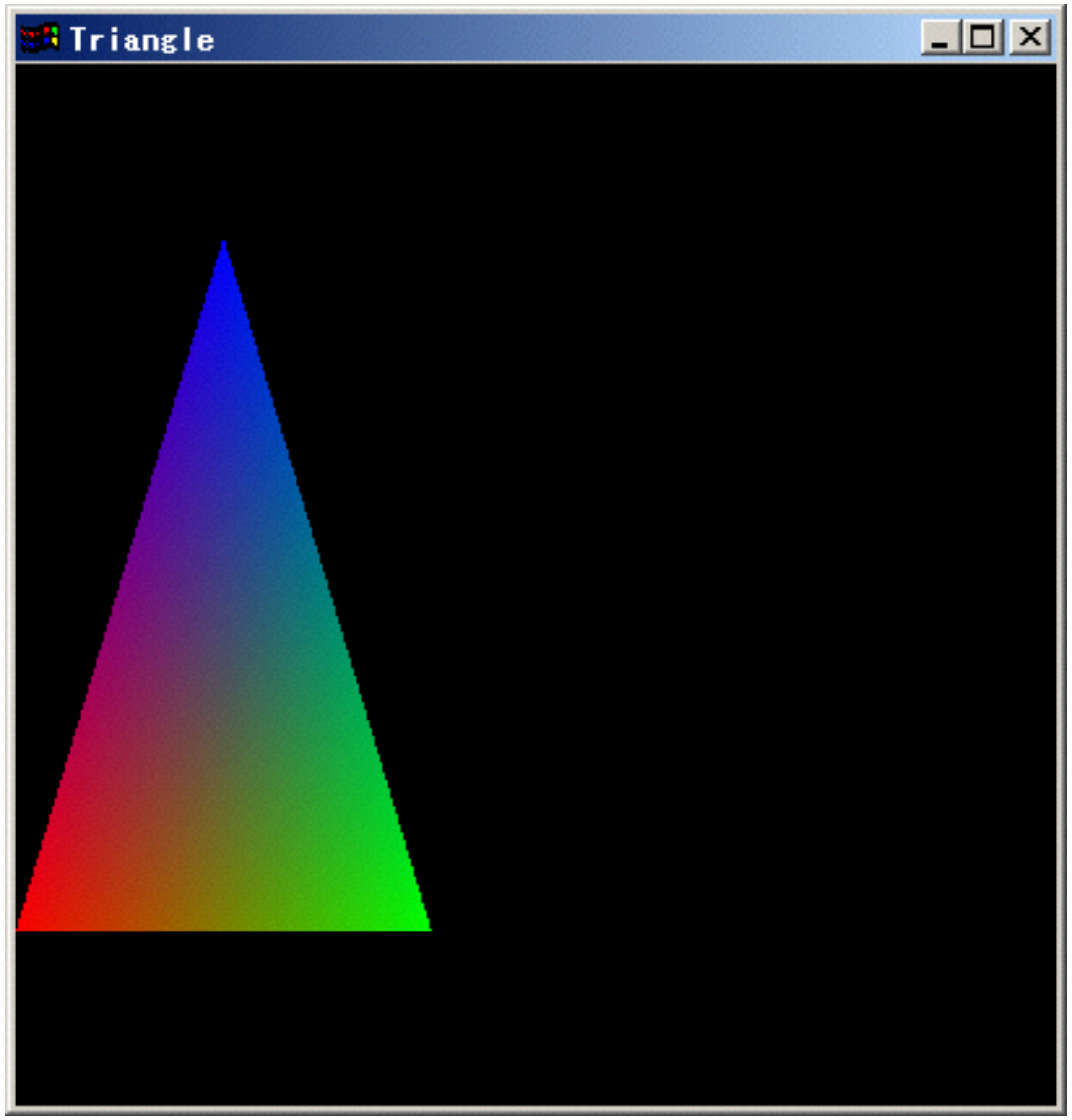
```
void myReshape(GLsizei w,GLsizei h)
{
    glViewport(0,0,w,h);//设置视口
    glMatrixMode(GL_PROJECTION);//指明当前矩阵为GL_PROJECTION
    glLoadIdentity();//将当前矩阵置换为单位阵

    //定义二维正视投影矩阵
    if(w <= h)
     gluOrtho2D(-1.0,1.5,-1.5,1.5*(GLfloat)h/(GLfloat)w);
    else
     gluOrtho2D(-1.0,1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5);
    glMatrixMode(GL_MODELVIEW); //指明当前矩阵为GL_MODELVIEW
}
```

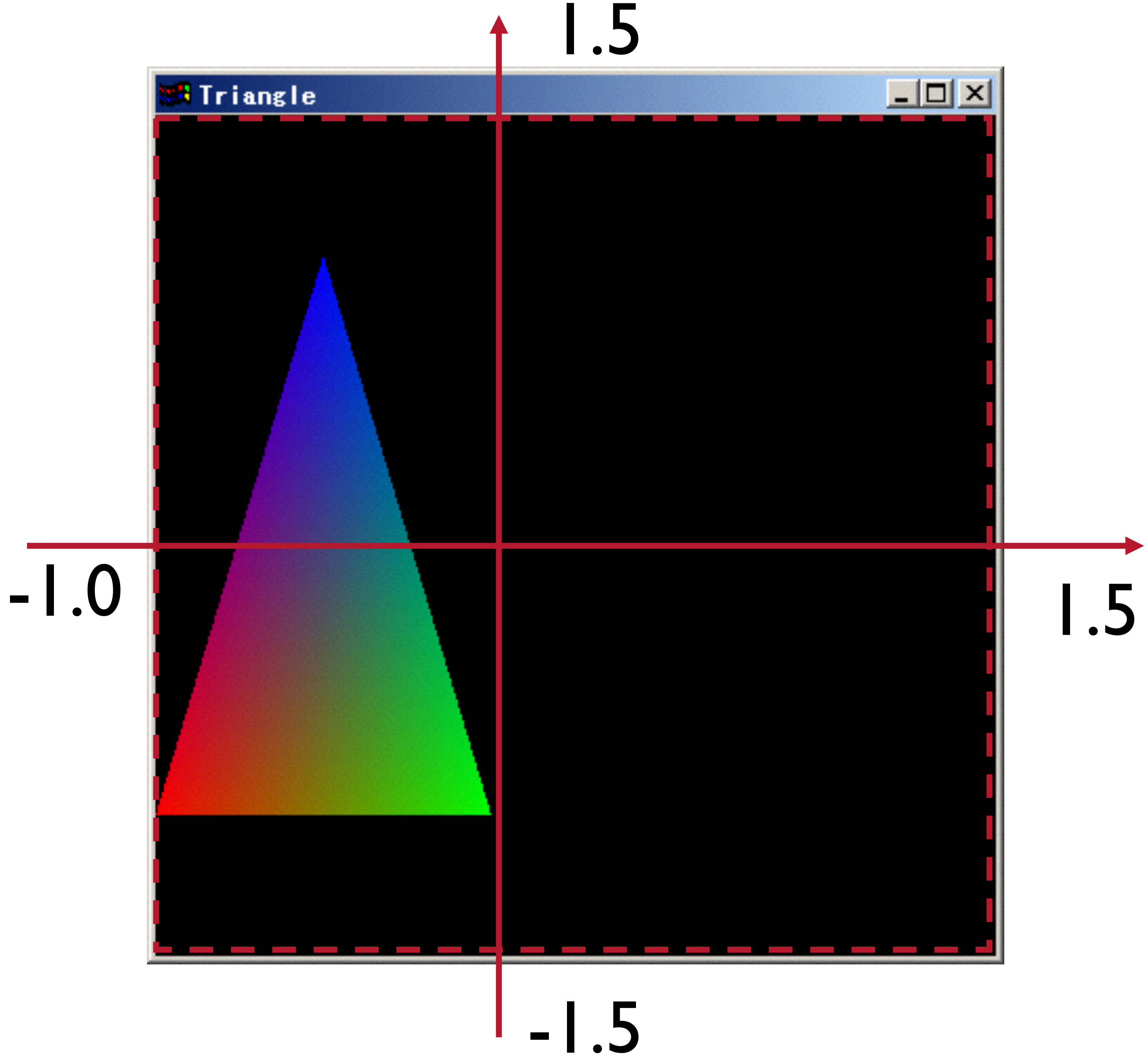
2D Rendering-2

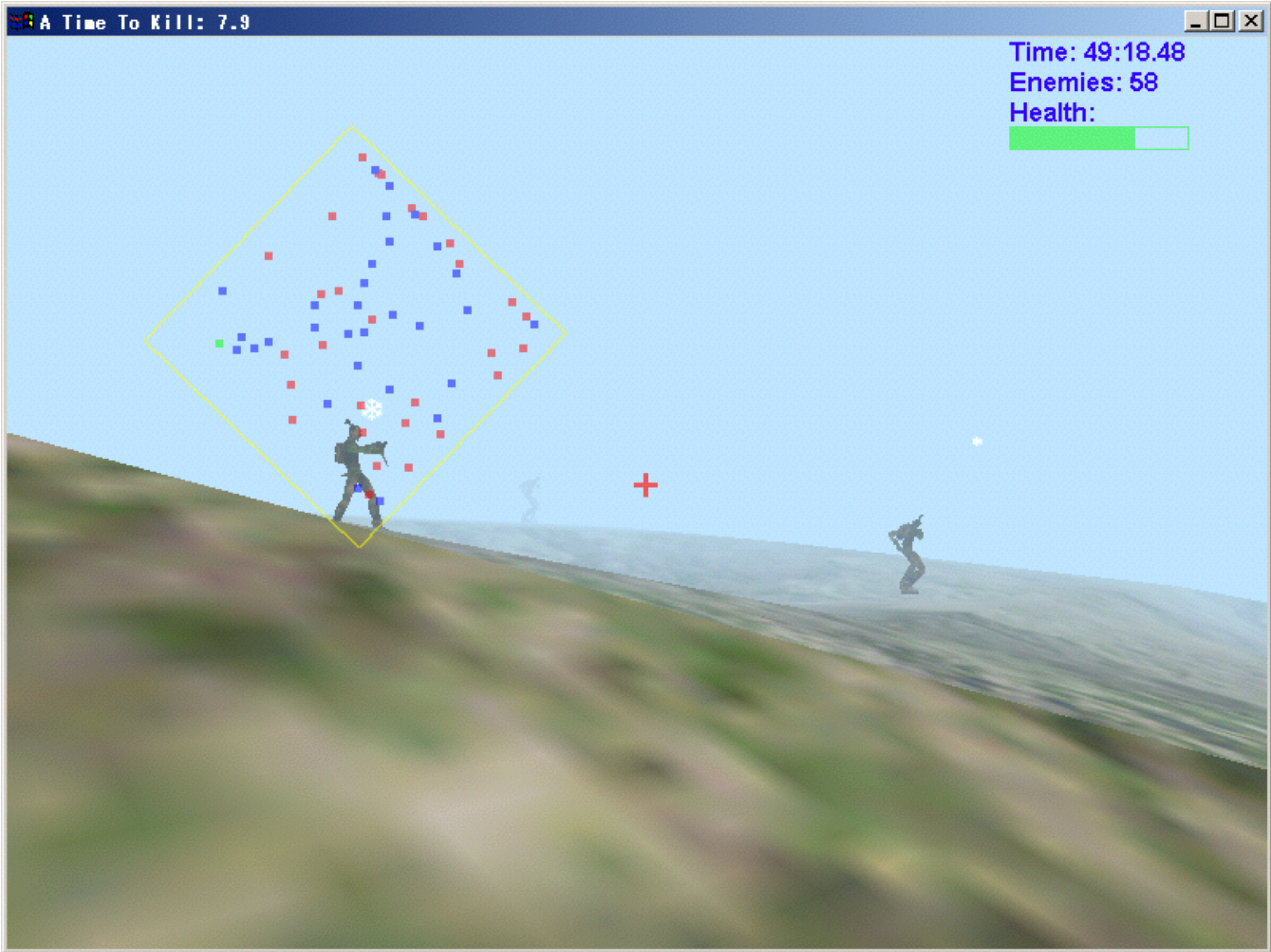
```
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT); //刷新颜色buffer
    glShadeModel(GL_SMOOTH); //设置为光滑明暗模式
    glBegin(GL_TRIANGLES);//开始画三角形
        glColor3f(1.0,0.0,0.0);//设置第一个顶点为红色
        glVertex2f(-1.0,-1.0);//设置第一个顶点的坐标为（-1.0，-1.0）
        glColor3f(0.0,1.0,0.0);//设置第二个顶点为绿色
        glVertex2f(0.0,-1.0);//设置第二个顶点的坐标为（.0，-1.0）
        glColor3f(0.0,0.0,1.0);//设置第三个顶点为蓝色
        glVertex2f(-0.5,1.0);//设置第三个顶点的坐标为（-0.5，1.0）
    glEnd(); //三角形结束
    glFlush(); //强制OpenGL函数在有限时间内运行
}
```

2D Rendering-3



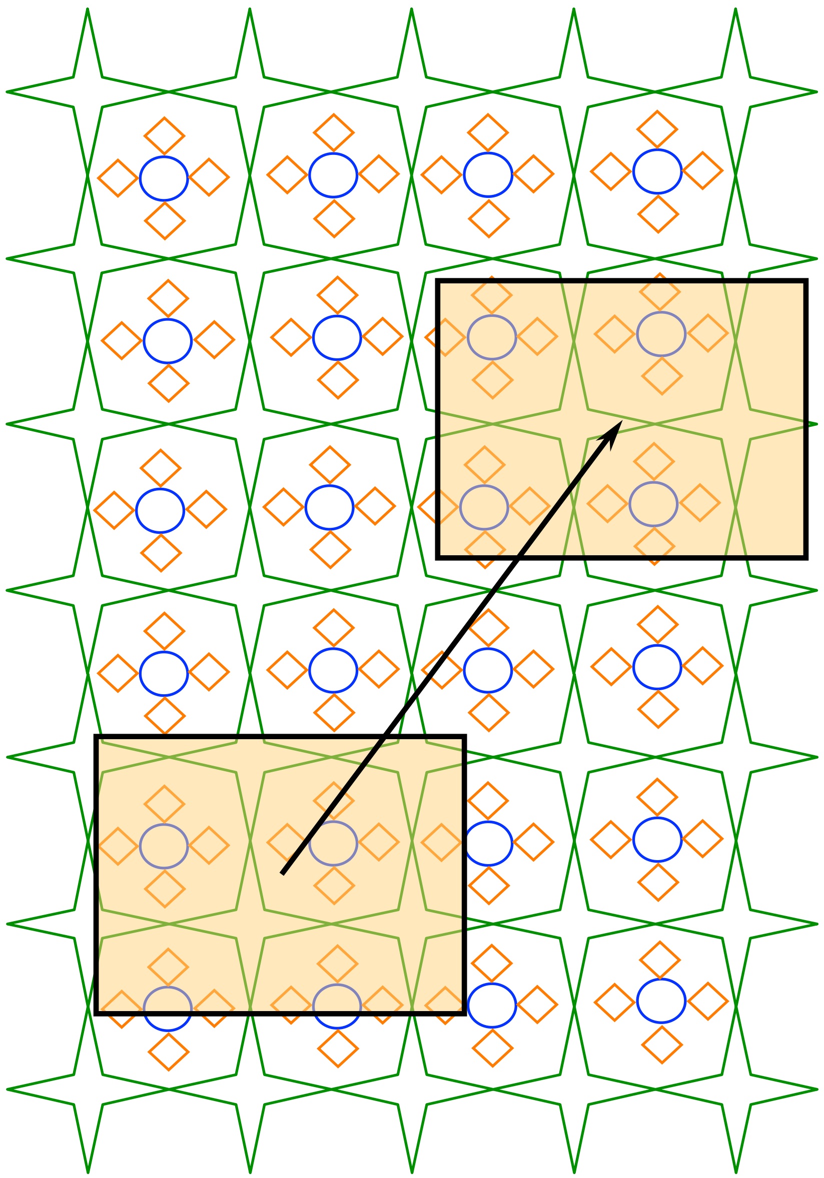
2D Rendering-3

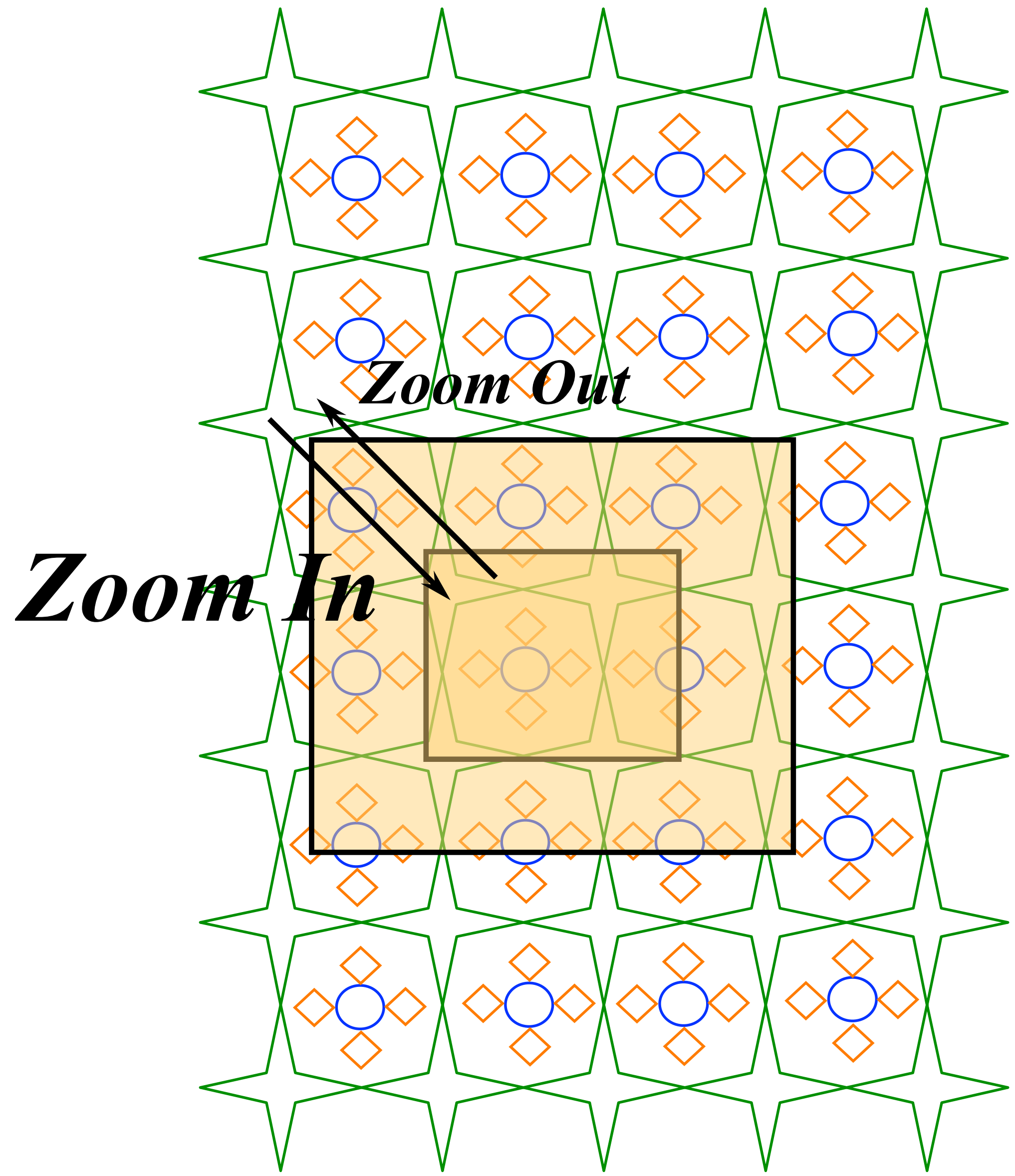



A Time To Kill: 7.9
Time: 49:18.48
Enemies: 58
Health:

User Interactions

Panning

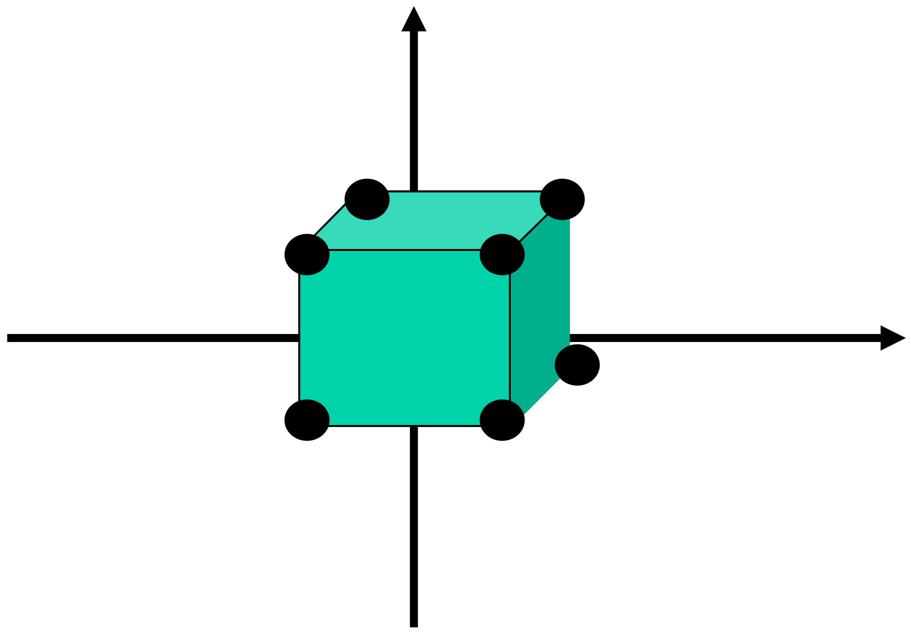


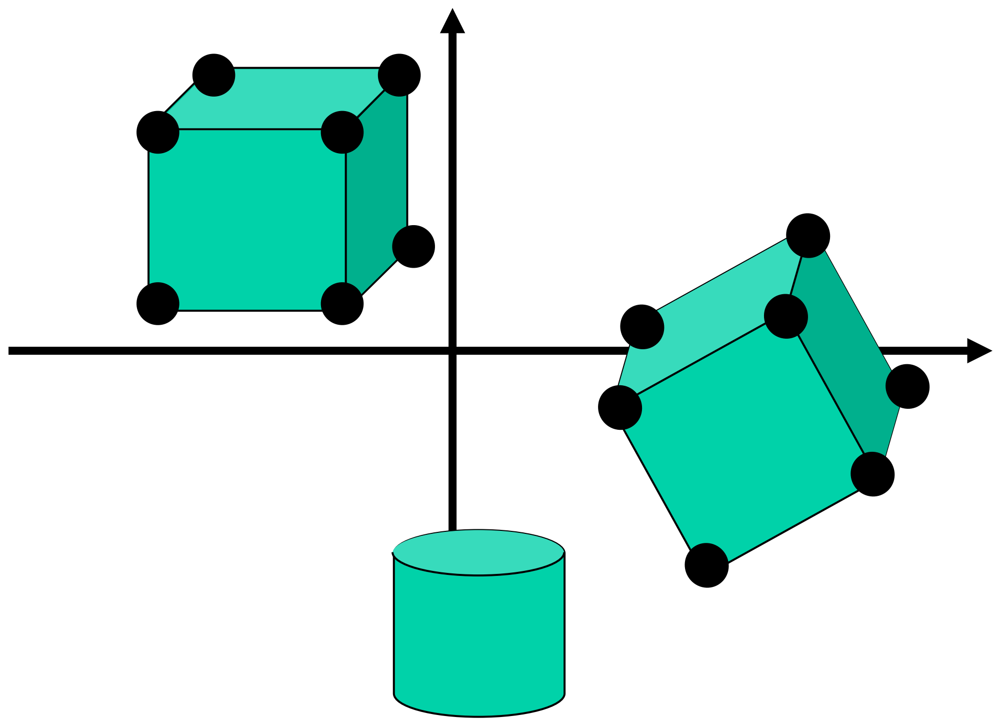


Modeling v.s. Viewing

- *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

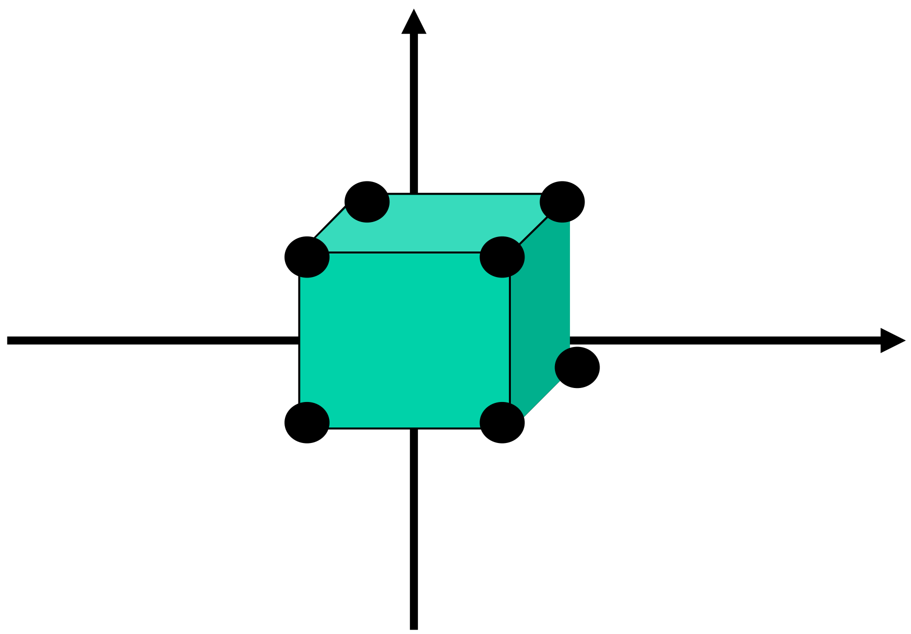
Modeling Coordinates



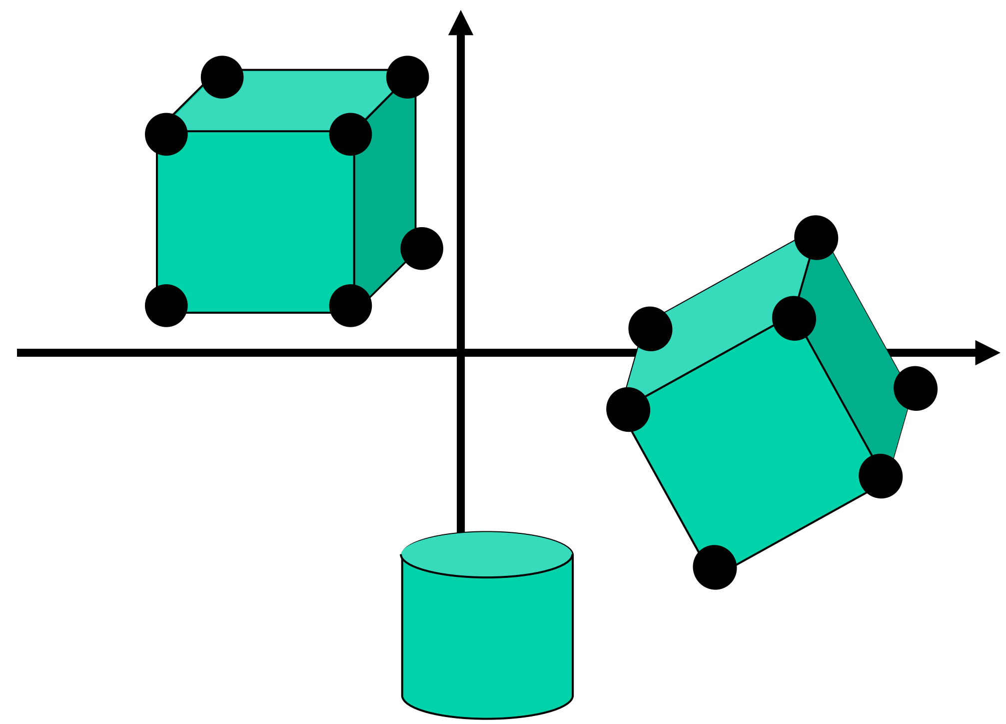


Objects are usually defined
in their own local coordinate system
(instance transformation)

Modeling Coordinates

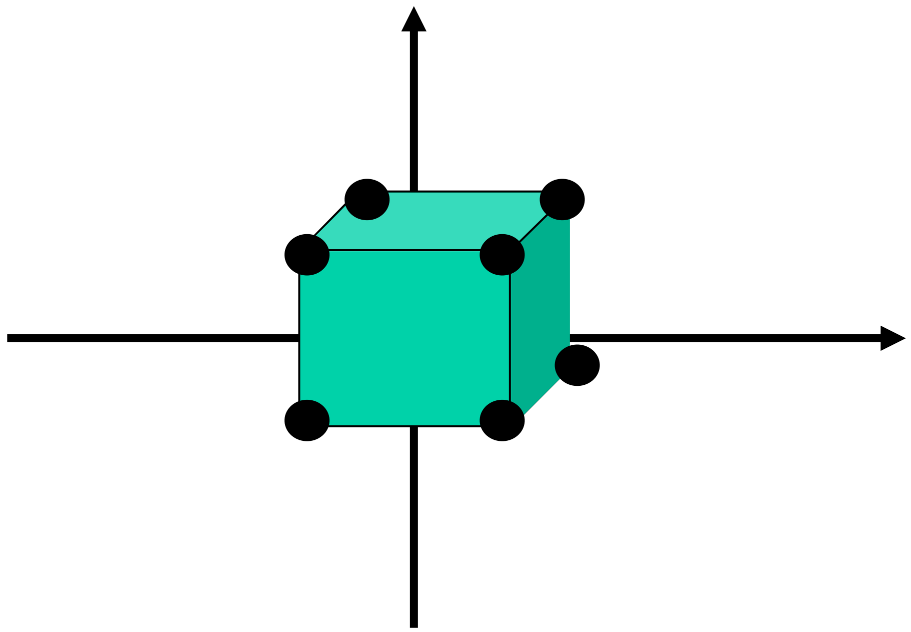


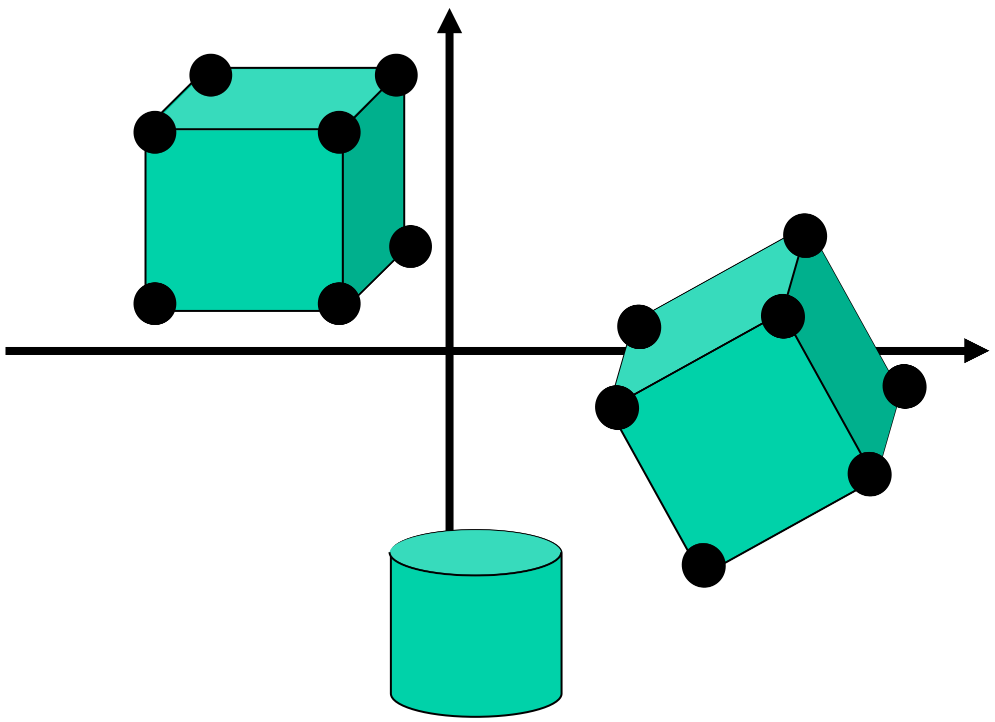
Objects are usually defined
in their own local coordinate system
(instance transformation)



Place (transform) these objects in a
single scene,

Modeling Coordinates

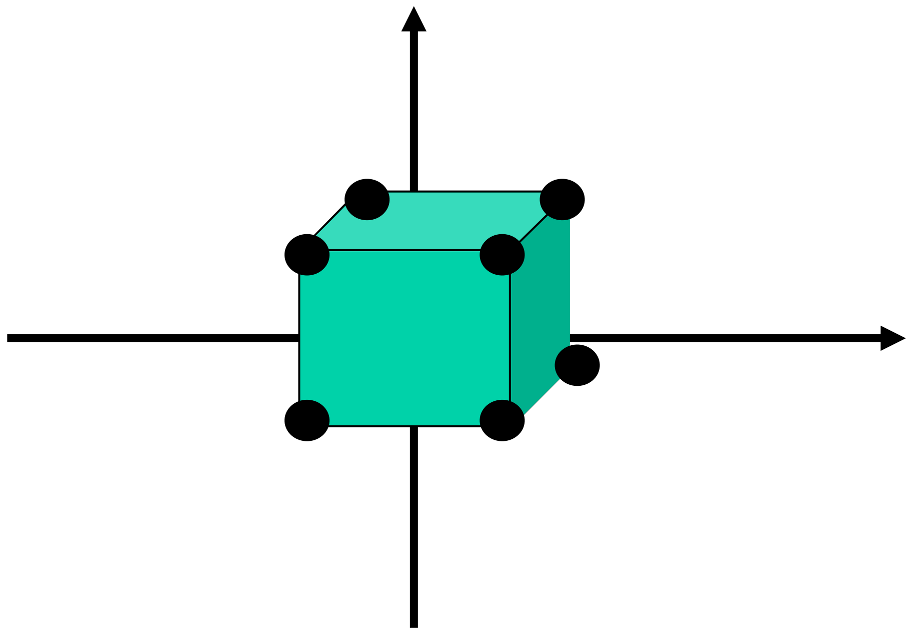




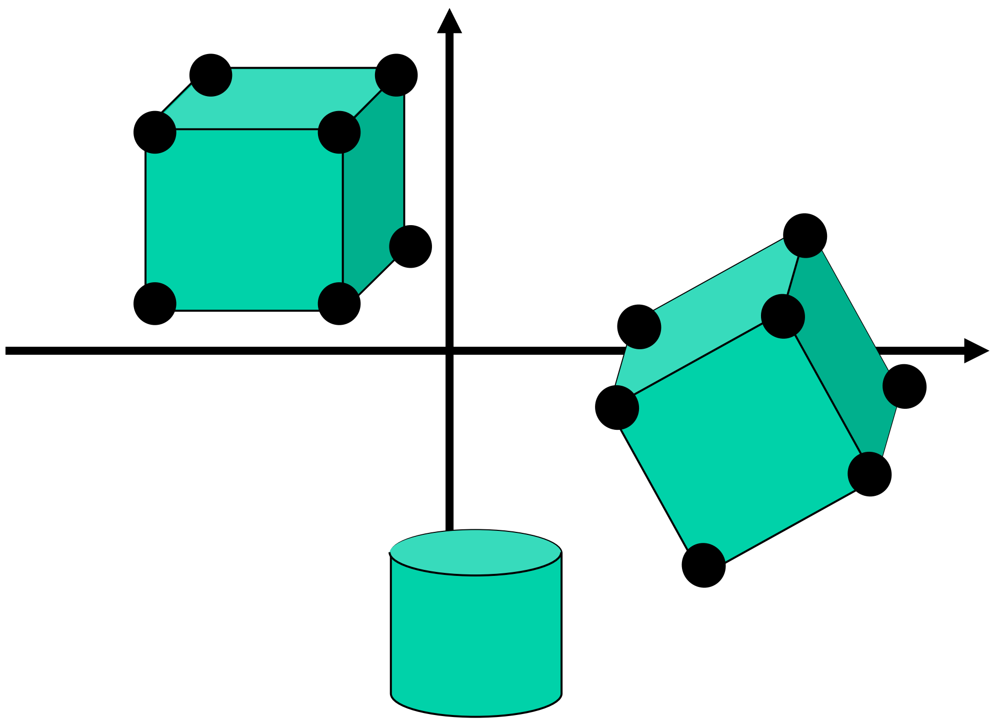
Objects are usually defined
in their own local coordinate system
(instance transformation)

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

Modeling Coordinates



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

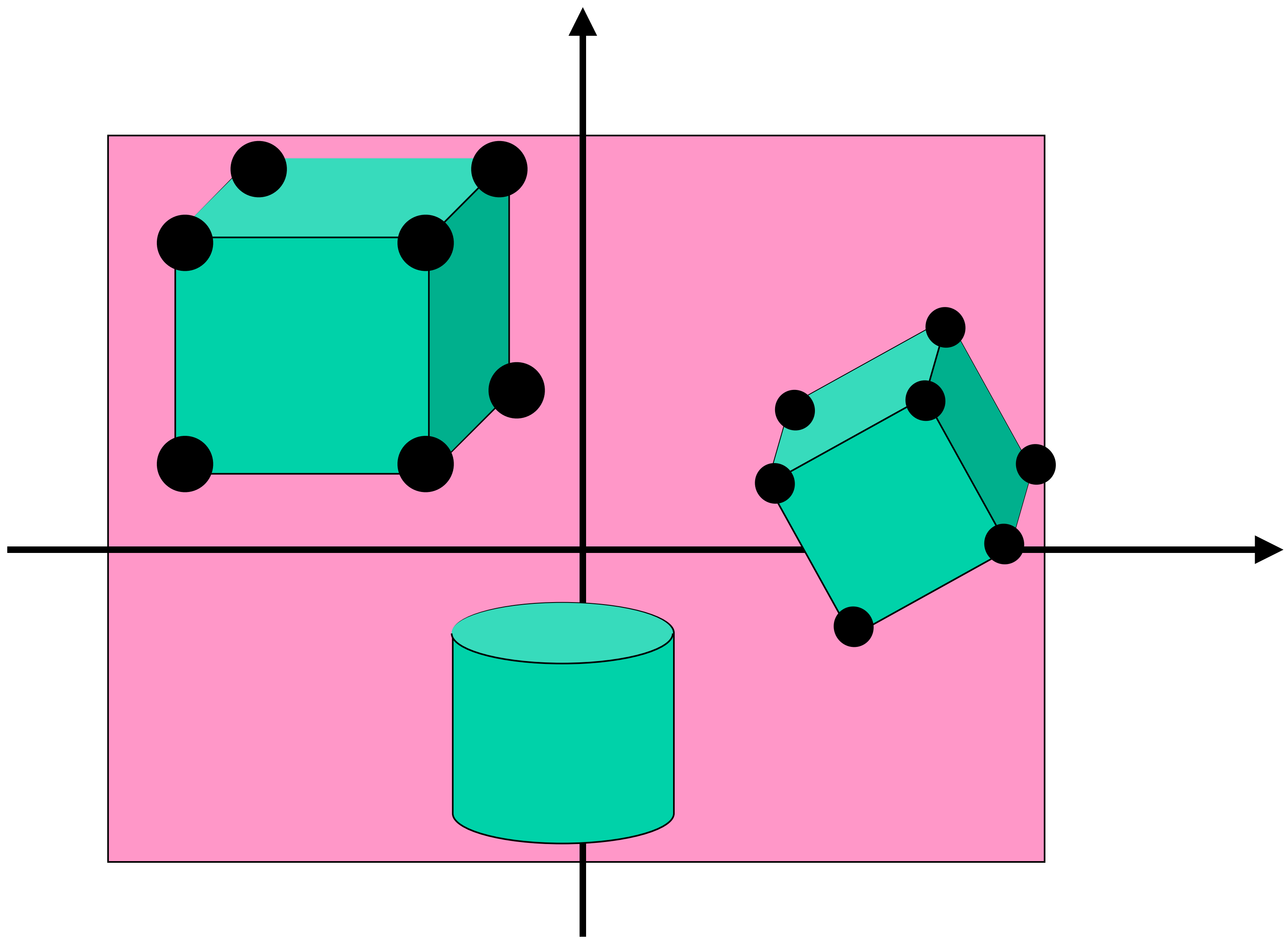


Place (transform) these objects in a single scene,

in world coordinates

(modeling transformation)

Screen Coordinates



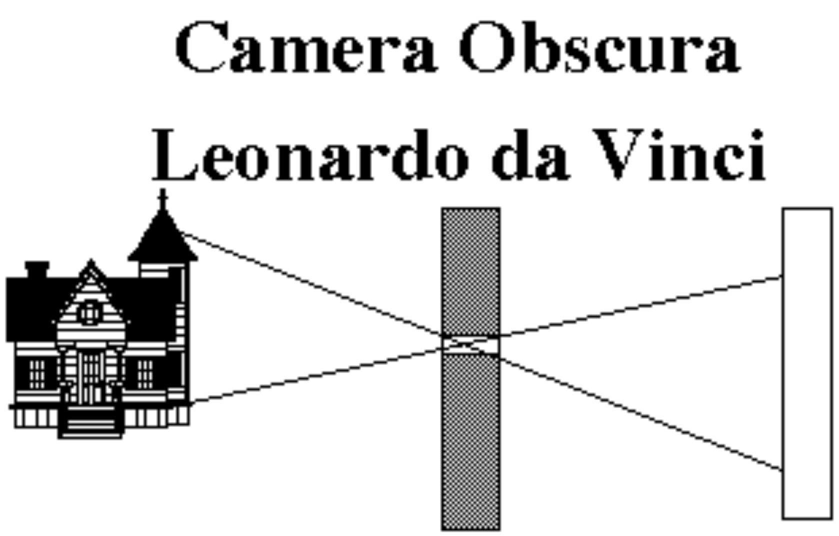
Finally, we want to project these objects onto the screen

3D Viewing

3D Viewing

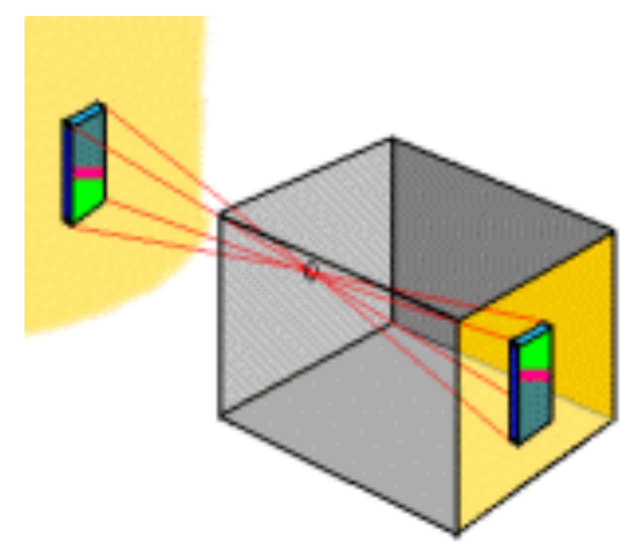
- To display a **3D world onto a 2D screen**
  - Specification becomes complicated because there are many parameters to control
  - Additional task of reducing dimensions from 3D to 2D (projection)
  - 3D viewing is analogous to taking a picture with a camera

The Pinhole Camera



The principle upon which all camera equipment works is traced to artist / inventor Leonardo da Vinci who showed that all that was needed to project an image was a small pinhole through which light could pass. The smaller the hole the sharper the image.

The basic camera, called a pinhole camera, existed in the early 17th Century. It took much longer for science to find a light sensitive material to record the image. It was not until 1826 when Joseph Niepce from France discovered that silver chloride（氯化银） could be used to make bitumen sensitive to light.



The world through a pinhole

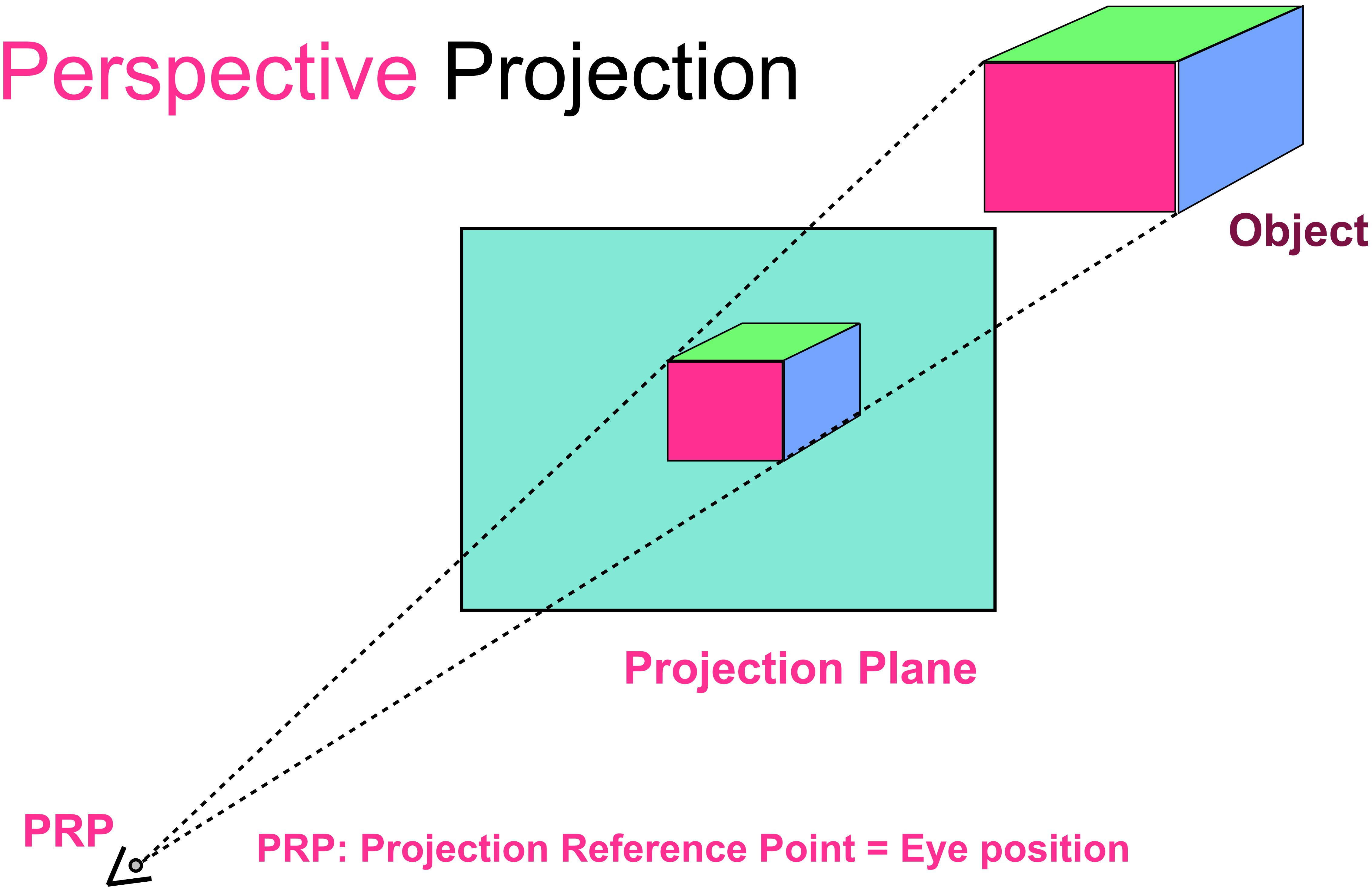
http://www.phy.ntnu.edu.tw/java/pinHole/pinhole.html

Transformations and Camera Analogy

- **Modeling transformation**
  - Shaping, positioning and moving the objects in the world scene
- **Viewing transformation**
  - Positioning and pointing camera onto the scene, selecting the region of interest
- **Projection transformation**
  - Adjusting the distance of the eye
- **Viewport transformation**
  - Enlarging or reducing the physical photograph

Classical Viewing

Perspective Projection



PRP: Projection Reference Point = Eye position

Figures extracted from Angle's textbook

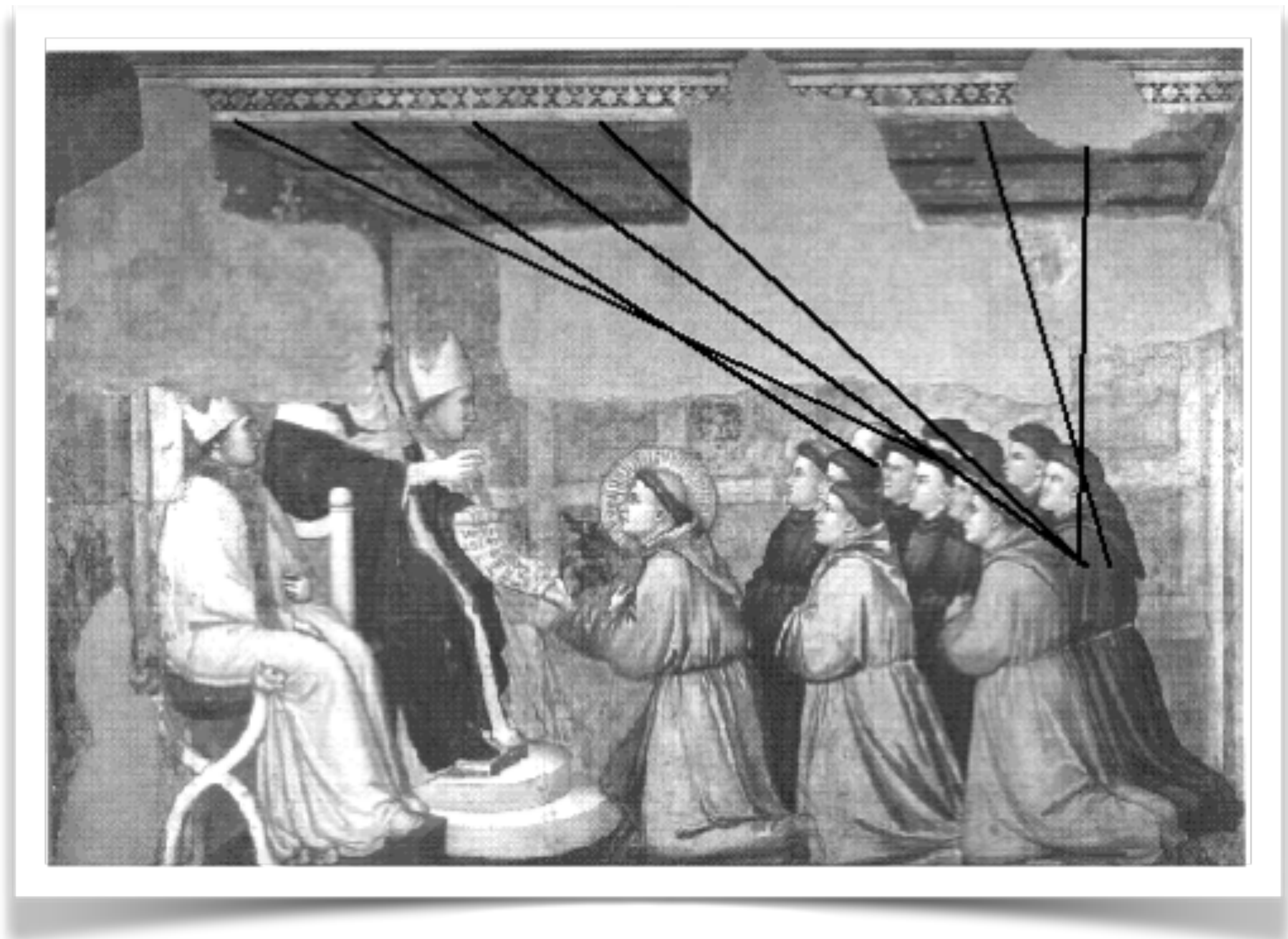
History of Perspective

"Perspective is the rein and rudder of painting"
Leonardo da Vinci

Early Perspective

Giotto di Bondone（1267~1337）

这位13世纪末、14世纪初的画家，为公认的西方绘画之父、文艺复兴的先驱。但以往因意大利官方难得愿意出借国宝外展，因此国人多不够熟悉这位西方美术史上的经典人物。



Not systematic -- lines do not converge to a single "vanishing" point

Vanishing Points

Brunelleschi

Invented systematic method of determining perspective projections in early 1400's

VPL

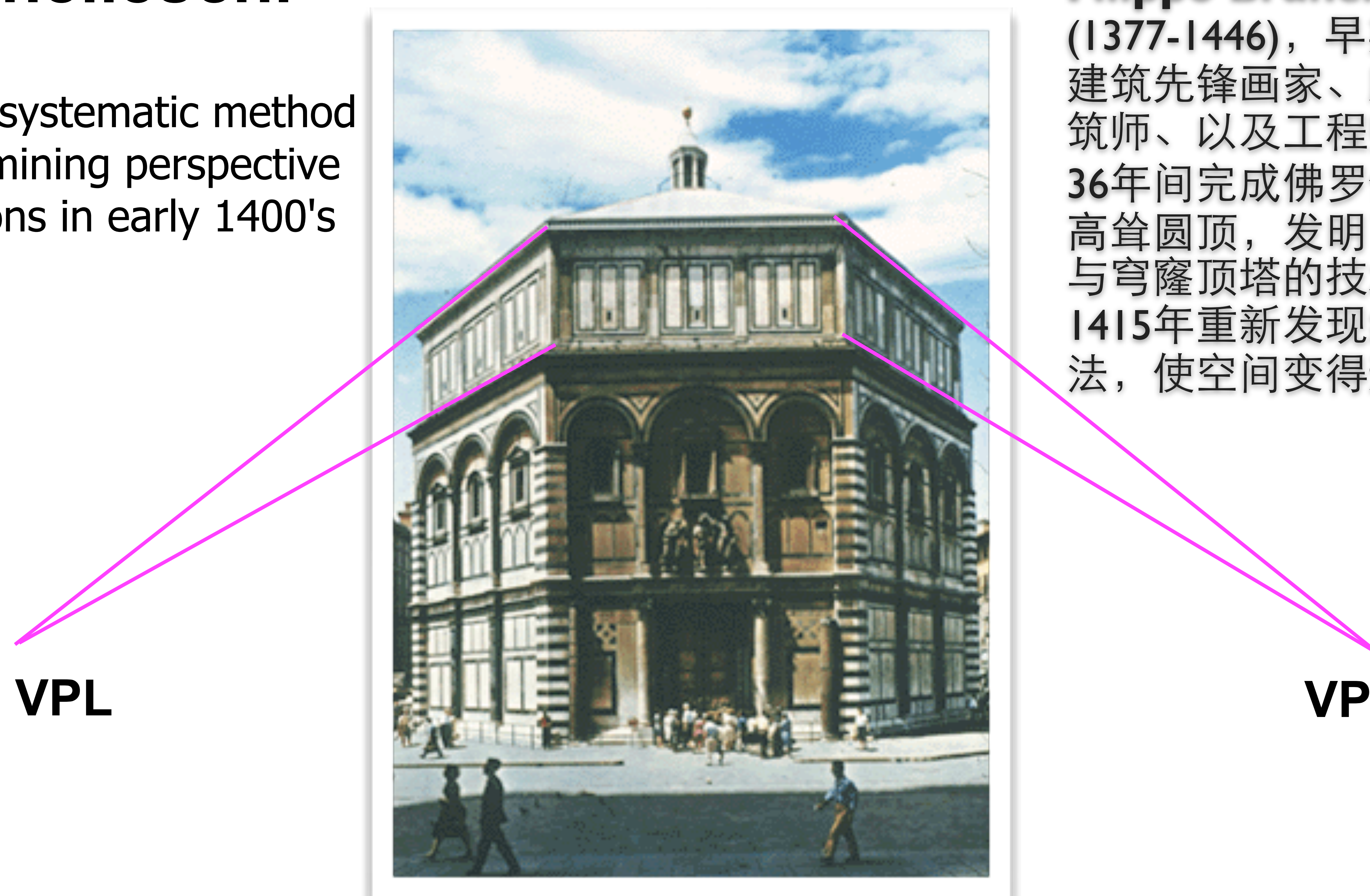
VPR

Brunelleschi's Peepshow (西洋镜)

Vanishing Points

Brunelleschi

Invented systematic method of determining perspective projections in early 1400's



Filippo Brunelleschi (1377-1446)，早期文艺复兴建筑先锋画家、雕刻家、建筑师、以及工程师。1420-36年间完成佛罗伦萨教堂的高耸圆顶，发明了完成圆顶与穹窿顶塔的技术与工程。1415年重新发现线性透视法，使空间变得逼真

Brunelleschi's Peepshow (西洋镜)

Single Vanishing Point

拉斐尔（RAFFAELLO SANZIO)（1483-1520）文艺复兴意大利艺坛三杰之一.

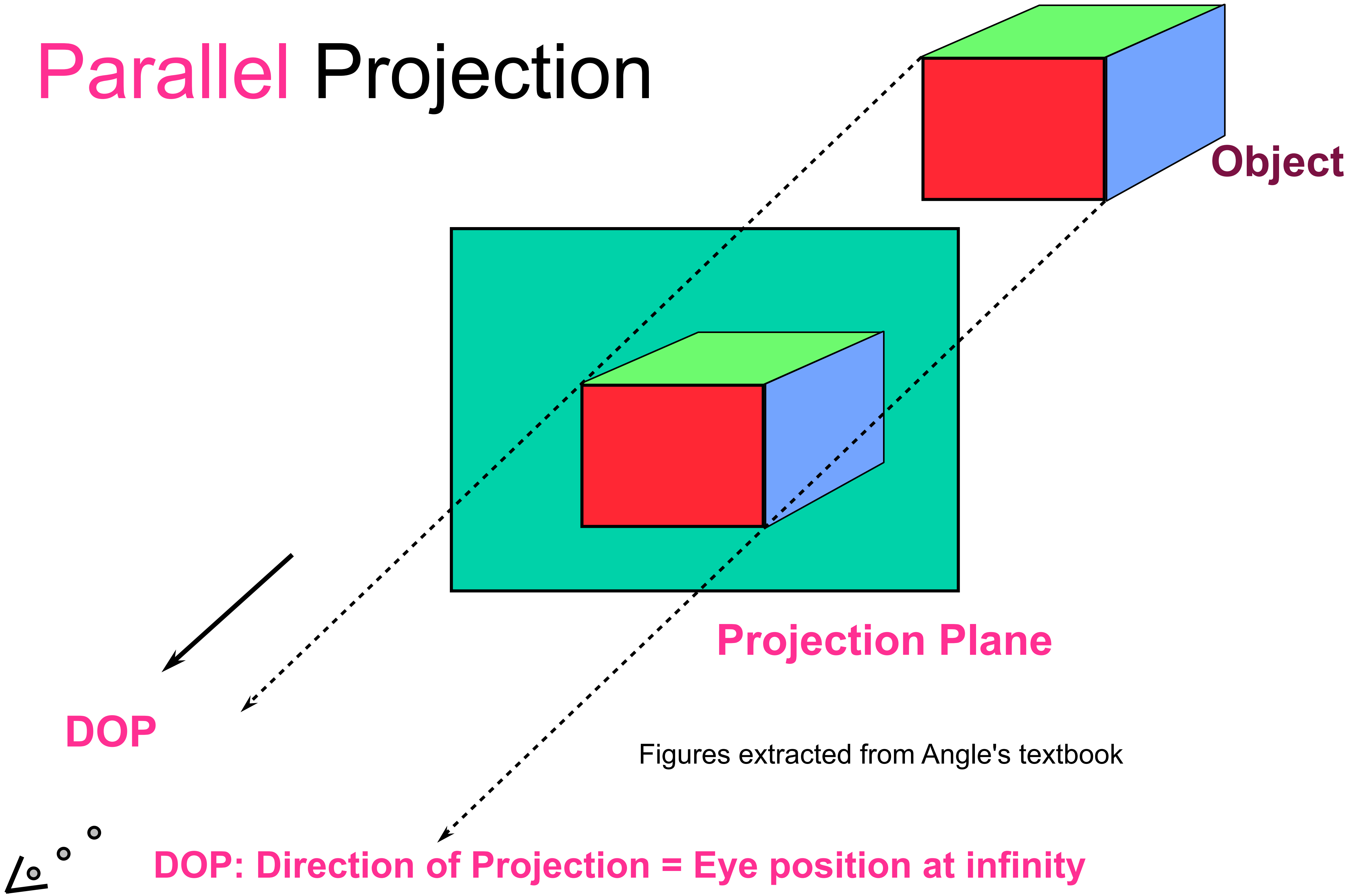
RAPHAEL: *School of Athens*

Perspective Projection

- 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

Engineering Viewing

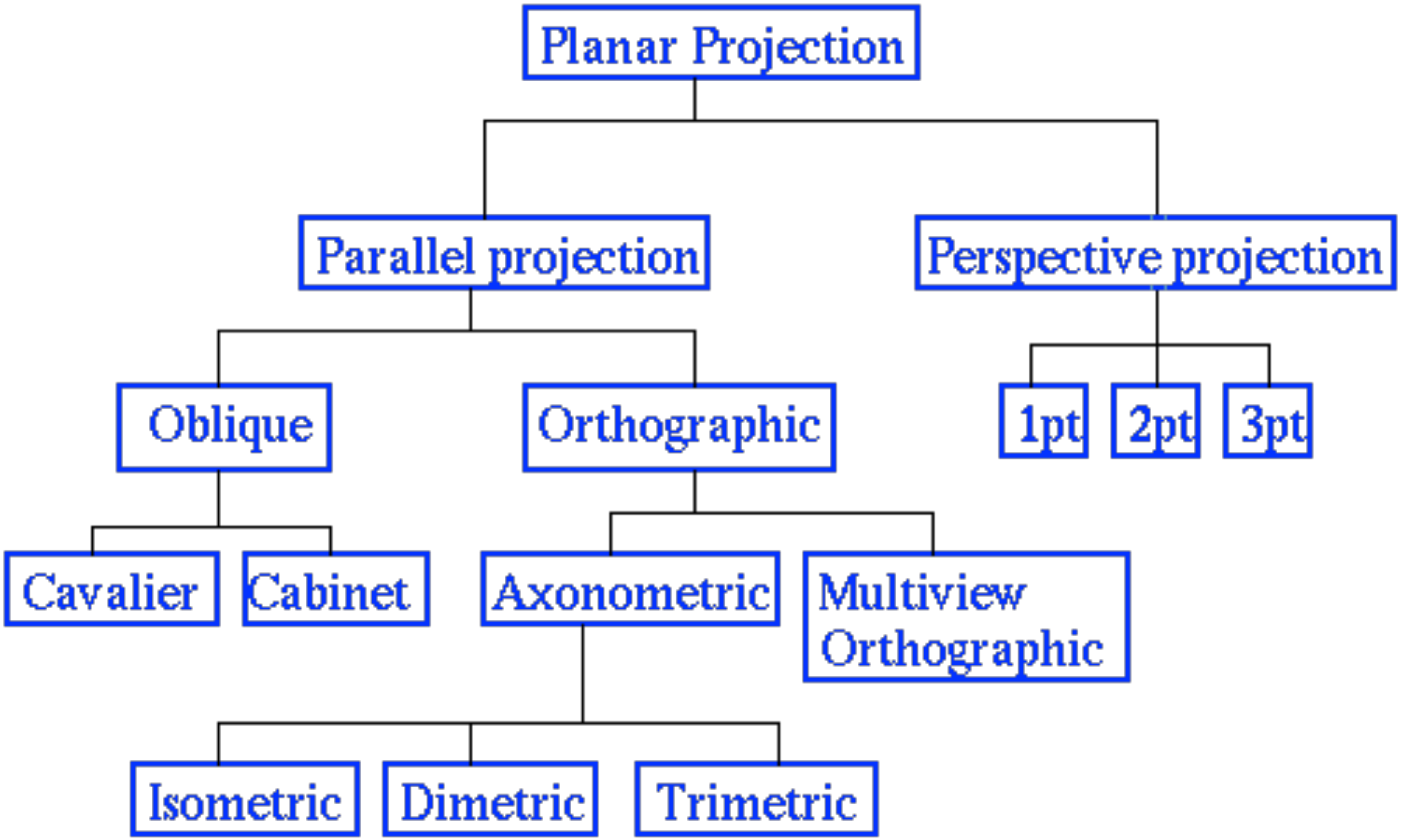
Parallel Projection



Figures extracted from Angle's textbook

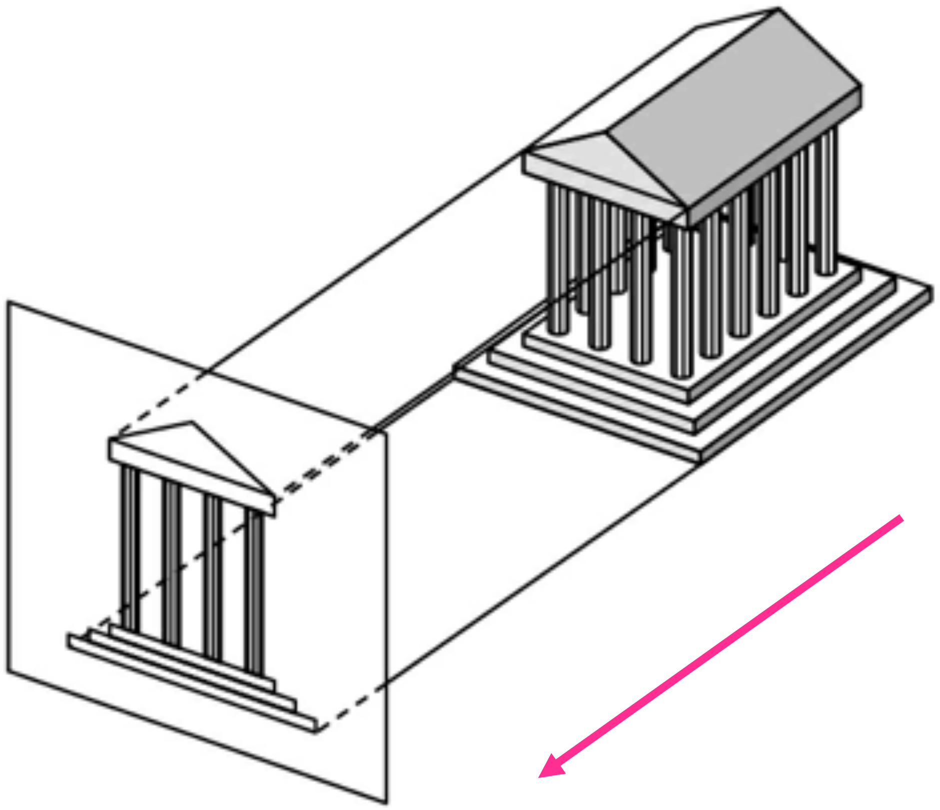
DOP: Direction of Projection = Eye position at infinity

Taxonomy of Projections

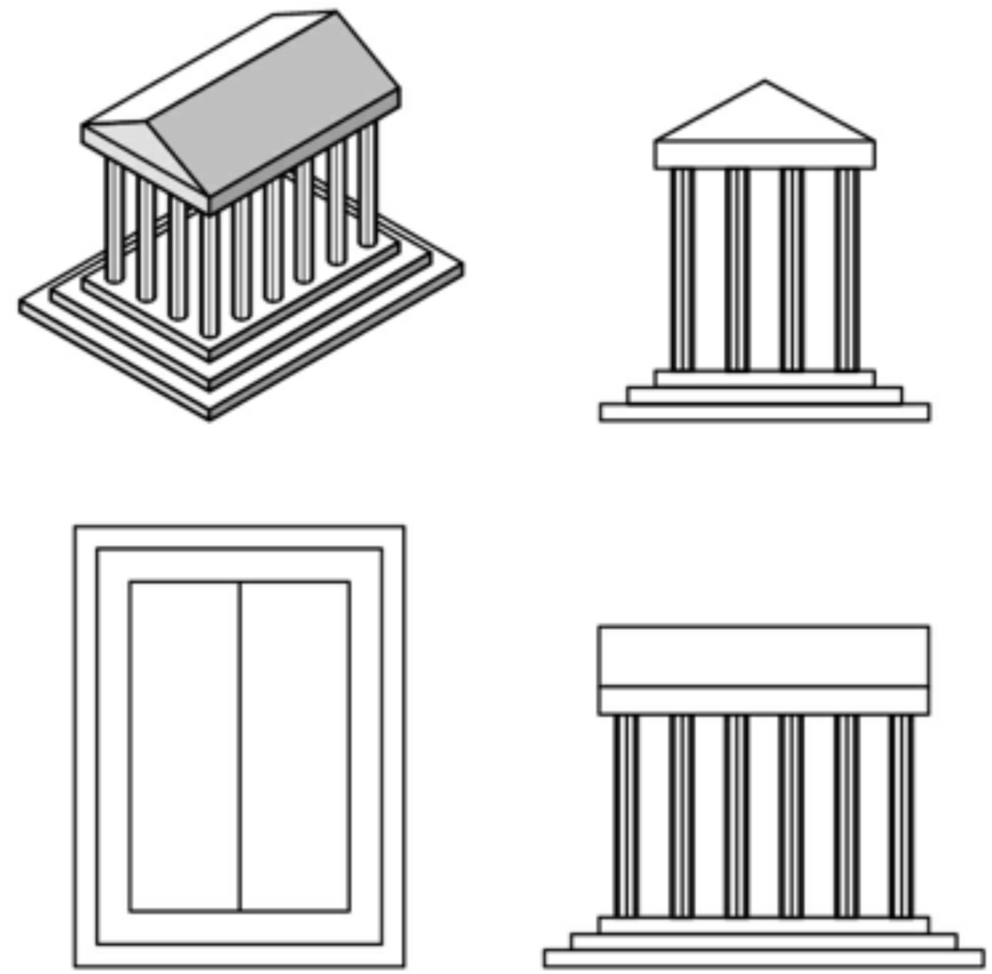


Figures extracted from Angle's textbook

Orthographic Projection 正交投影



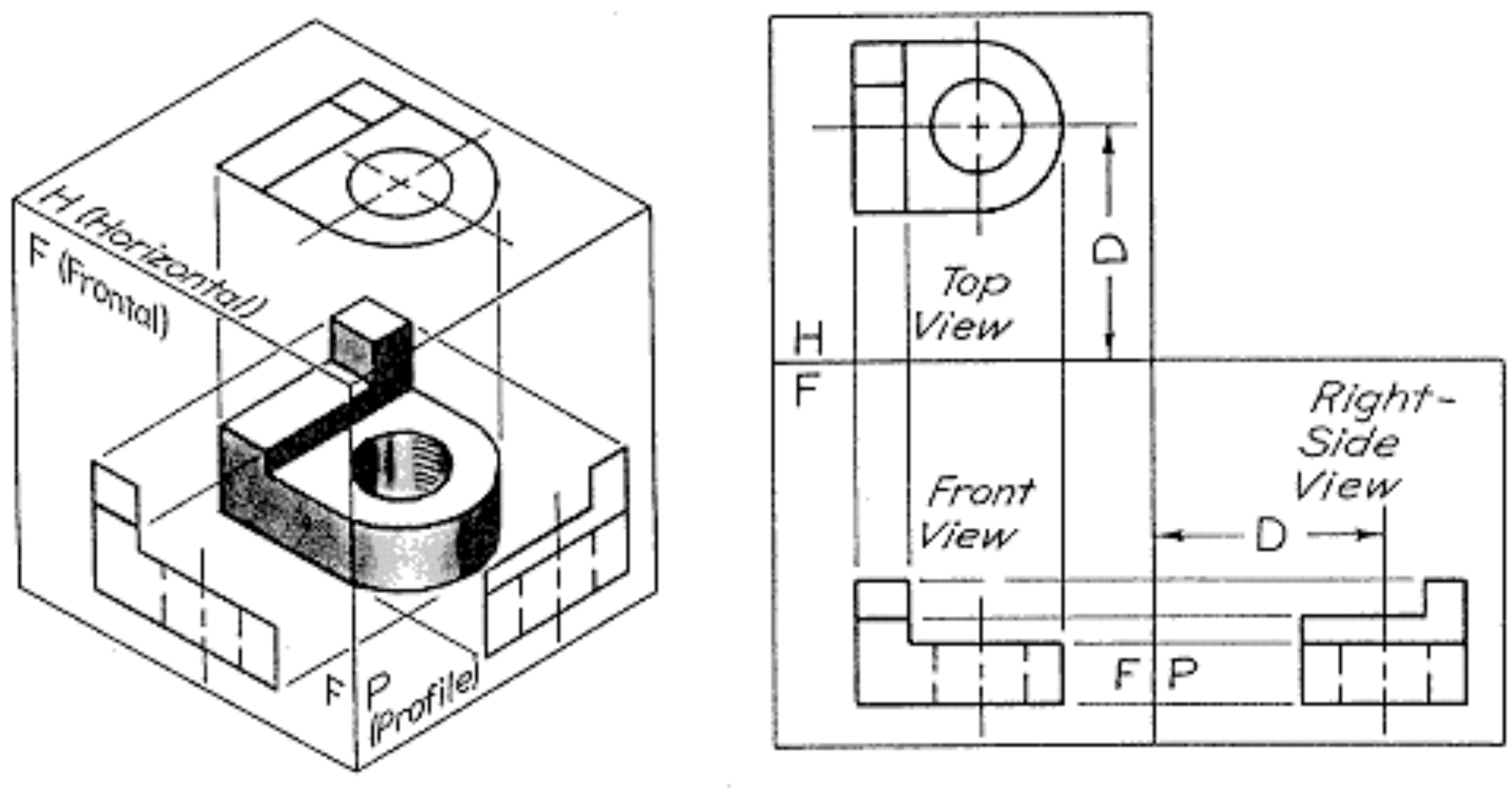
DOP
same for all points



DOP is perpendicular to the view plane

Figures extracted from Angle's textbook

Multiview Parallel Projection
多角度平行投影



(a) (b)

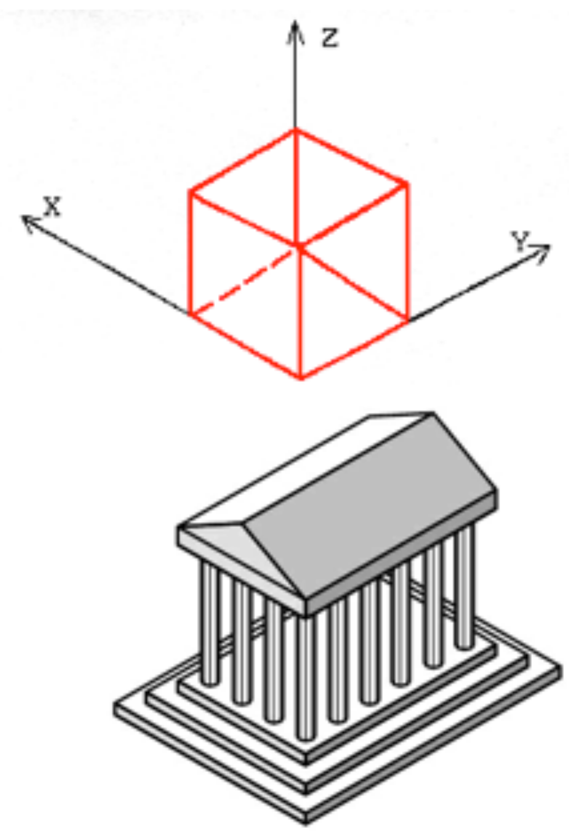
Faces are parallel to the projection plane

Figures extracted from Angle's textbook

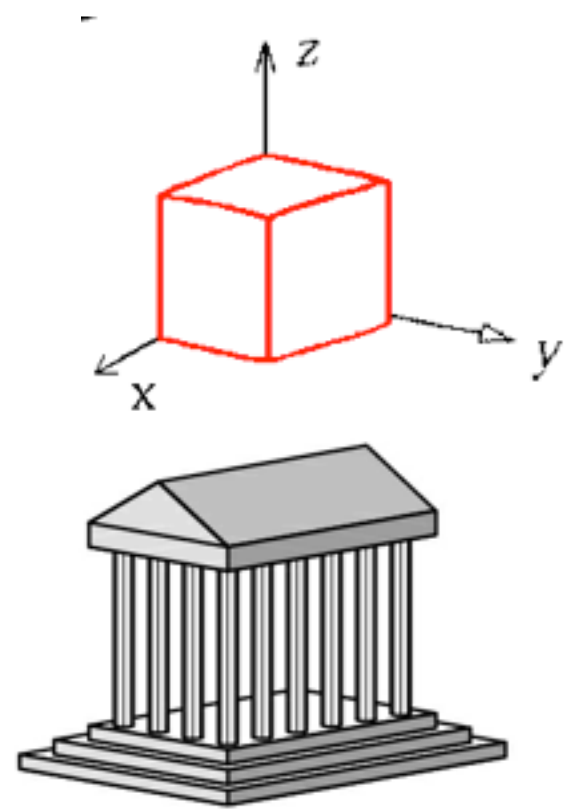
Axonometric Projections 轴侧投影

- DOP orthogonal to the projection plane, but…

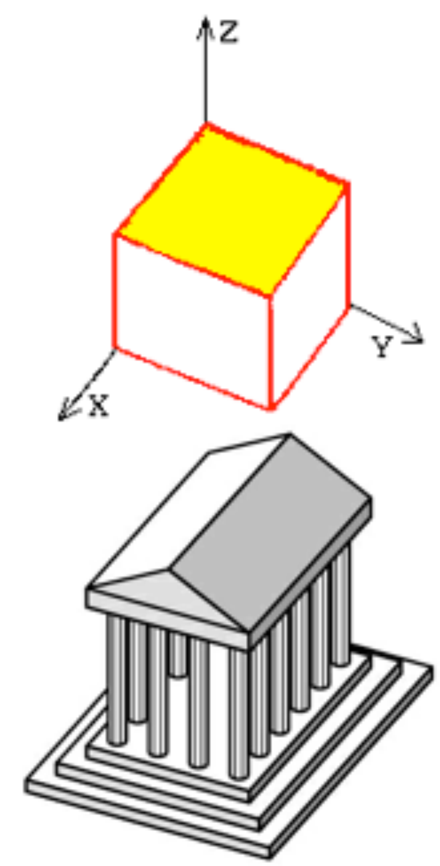
  …orient projection plane with respect to the object
- Parallel lines remain parallel, and receding lines are equally foreshortened by some factor.



Isometric 等轴侧



Dimetric 二等角



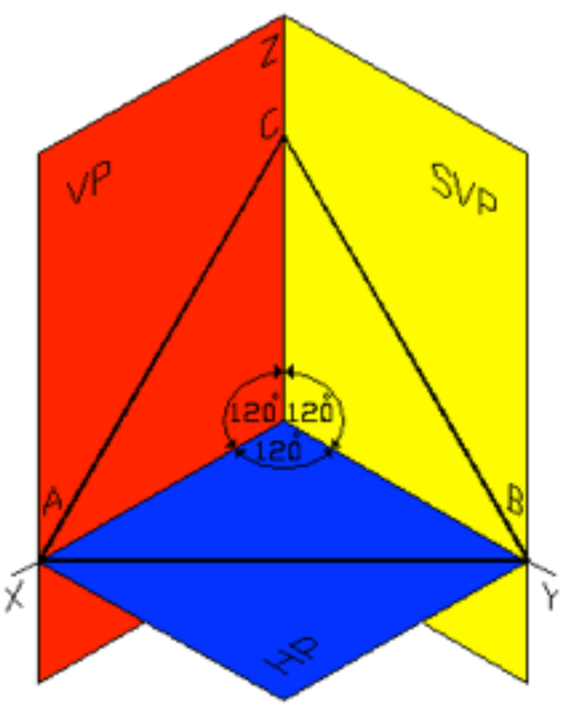
Trimetric 三度

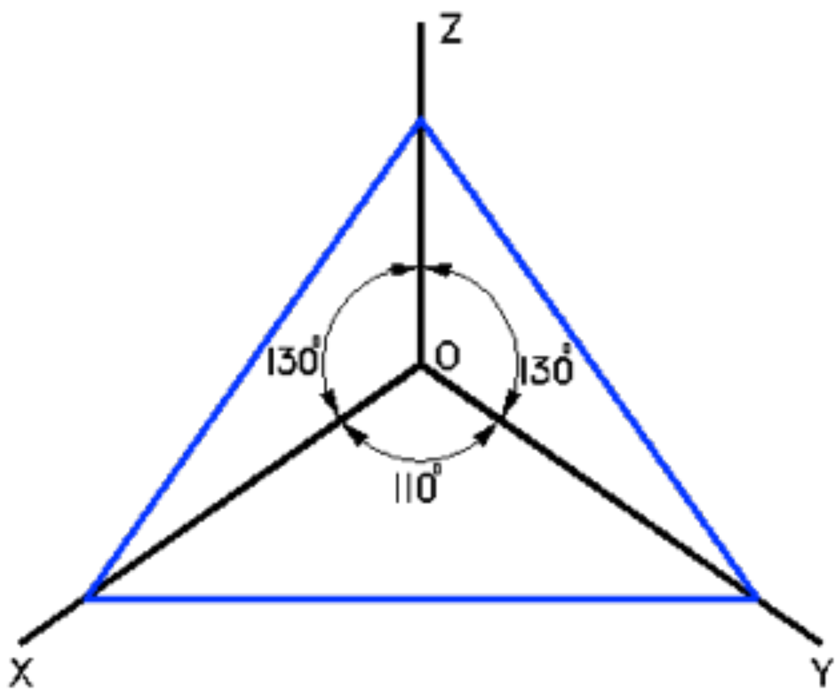
Projection type depends on angles made by projector with the three principal axes.

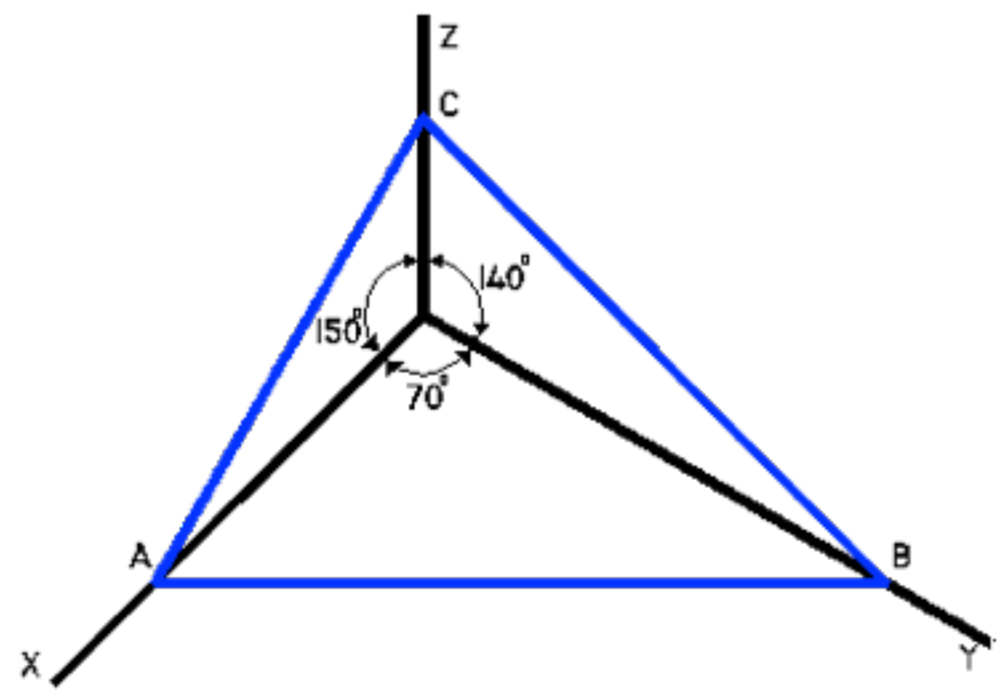
Figures extracted from Angle's textbook

Reference

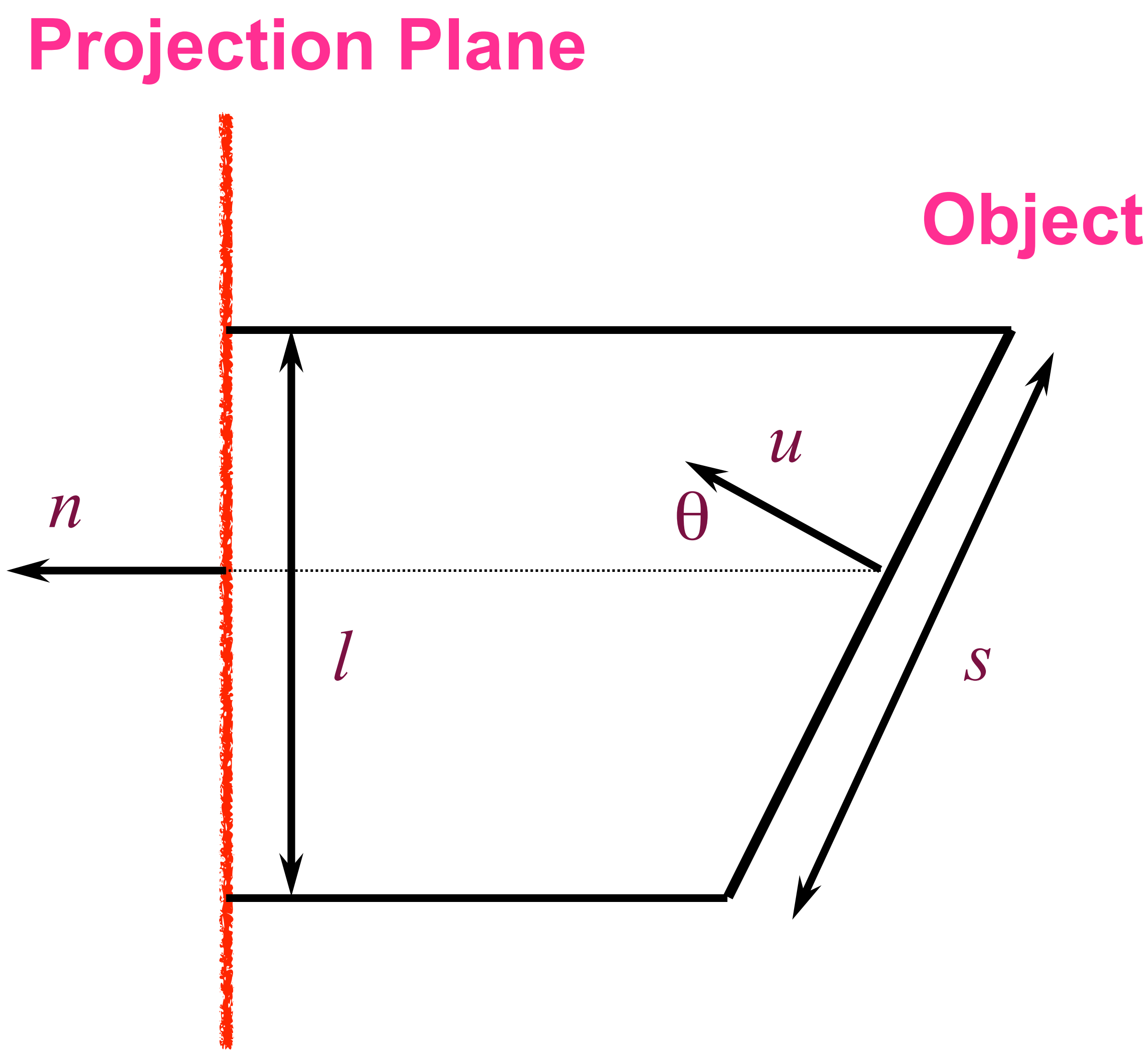
- http://www.ul.ie/~rynnet/keanea/isometri.htm







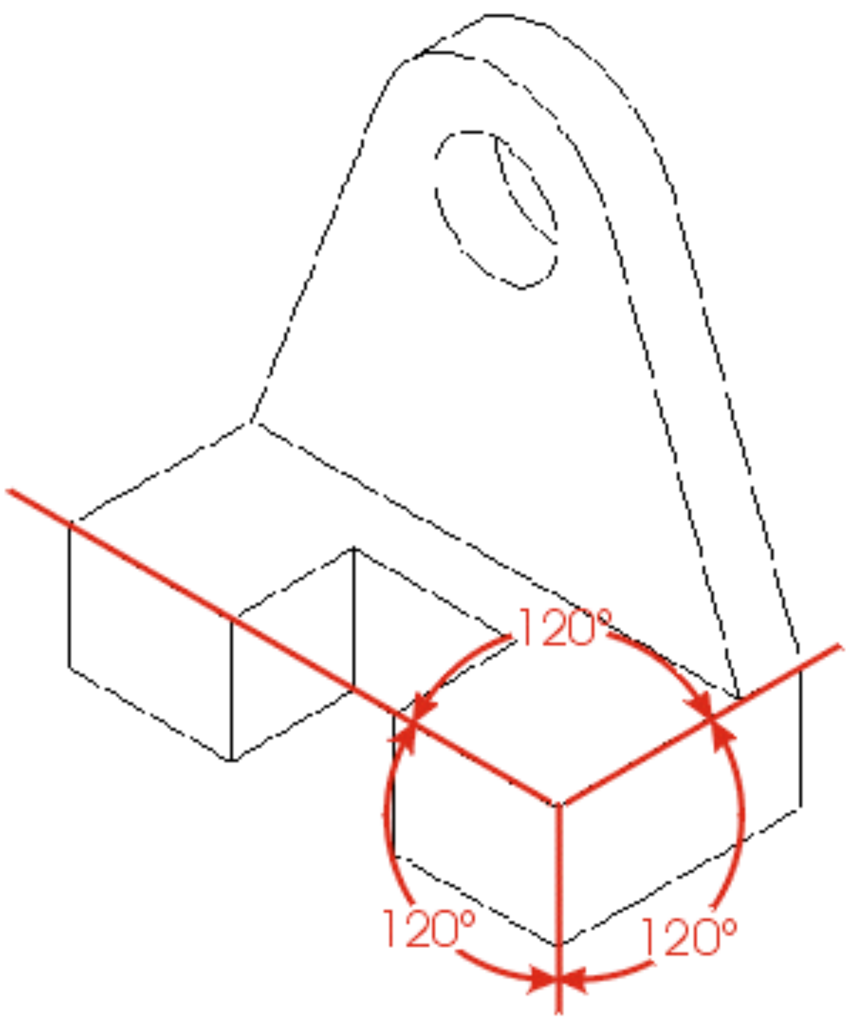
Foreshortening



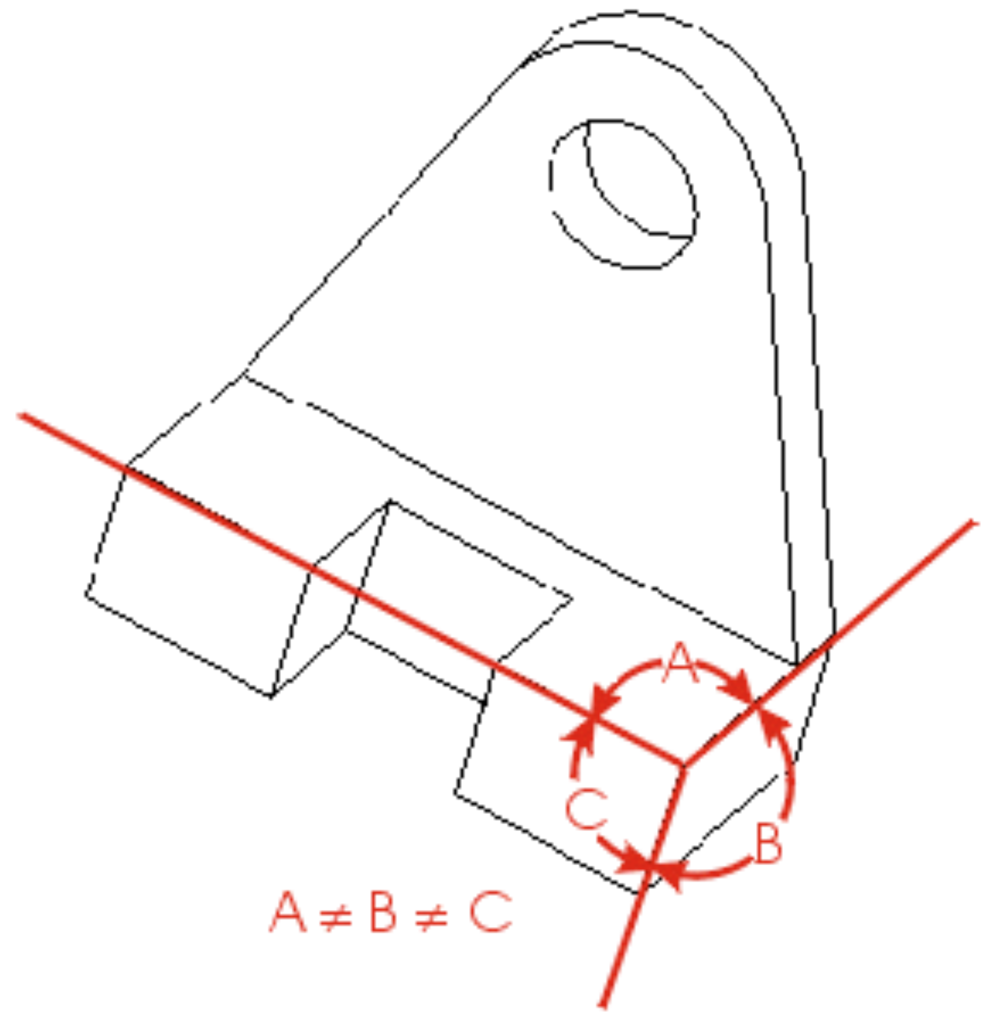
Object size s is foreshortened to l

$$l = s\cos\theta = s(u \cdot n)$$

mechanical drawing



isometric



trimetric

Oblique Projections

斜平行投影

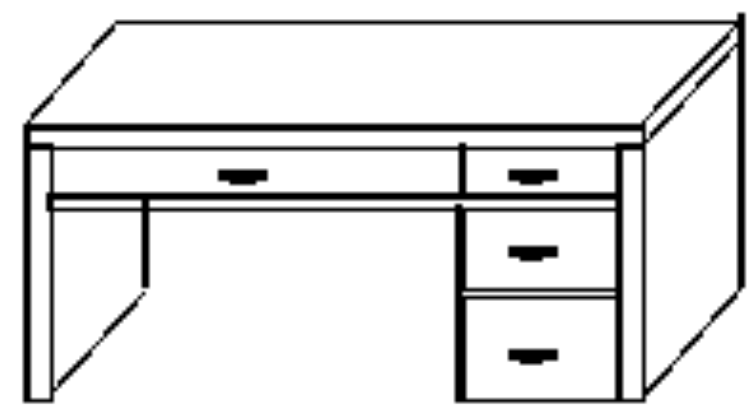
- Most general parallel views
- Projectors make an arbitrary angle with the projection plane
- Angles in planes parallel to the projection plane are preserved
- Back of view camera

Oblique Projections

斜平行投影

cavalier

斜等测



cabinet

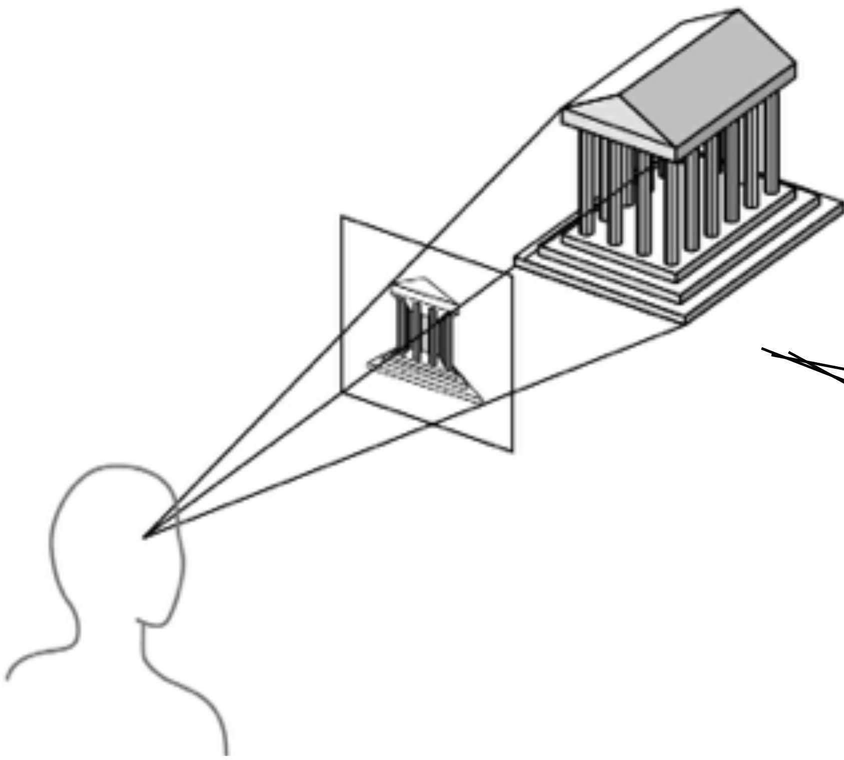
斜二测

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

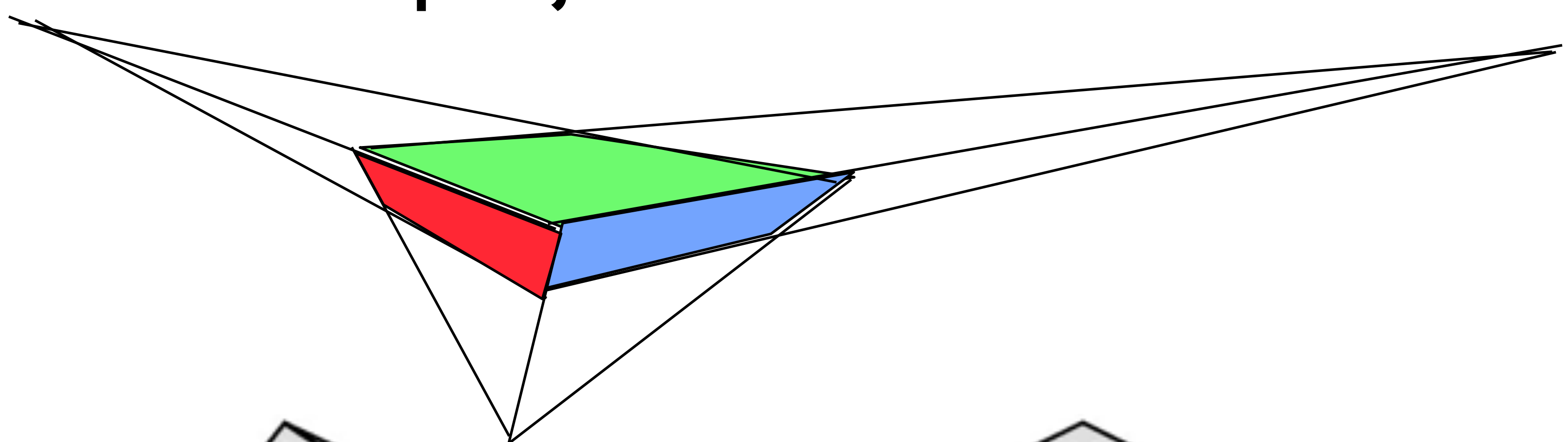
Cabinet
Angle between projectors and projection plane is 63.4°. Perpendicular faces are projected at 50% scale

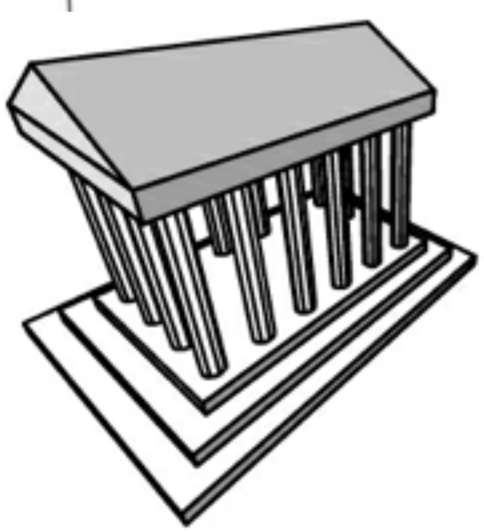
Figures extracted from Angle's textbook

Perspective Viewing

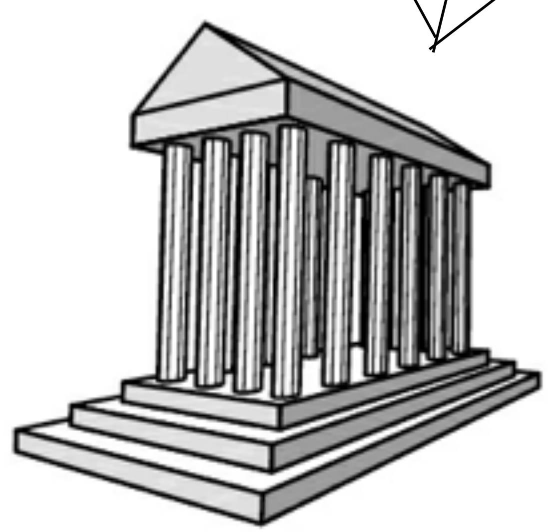


Type of perspective depends on the number of principal axes cut by the projectors.





3-point



2-point

1-point

Figures extracted from Angle's textbook

View Specification

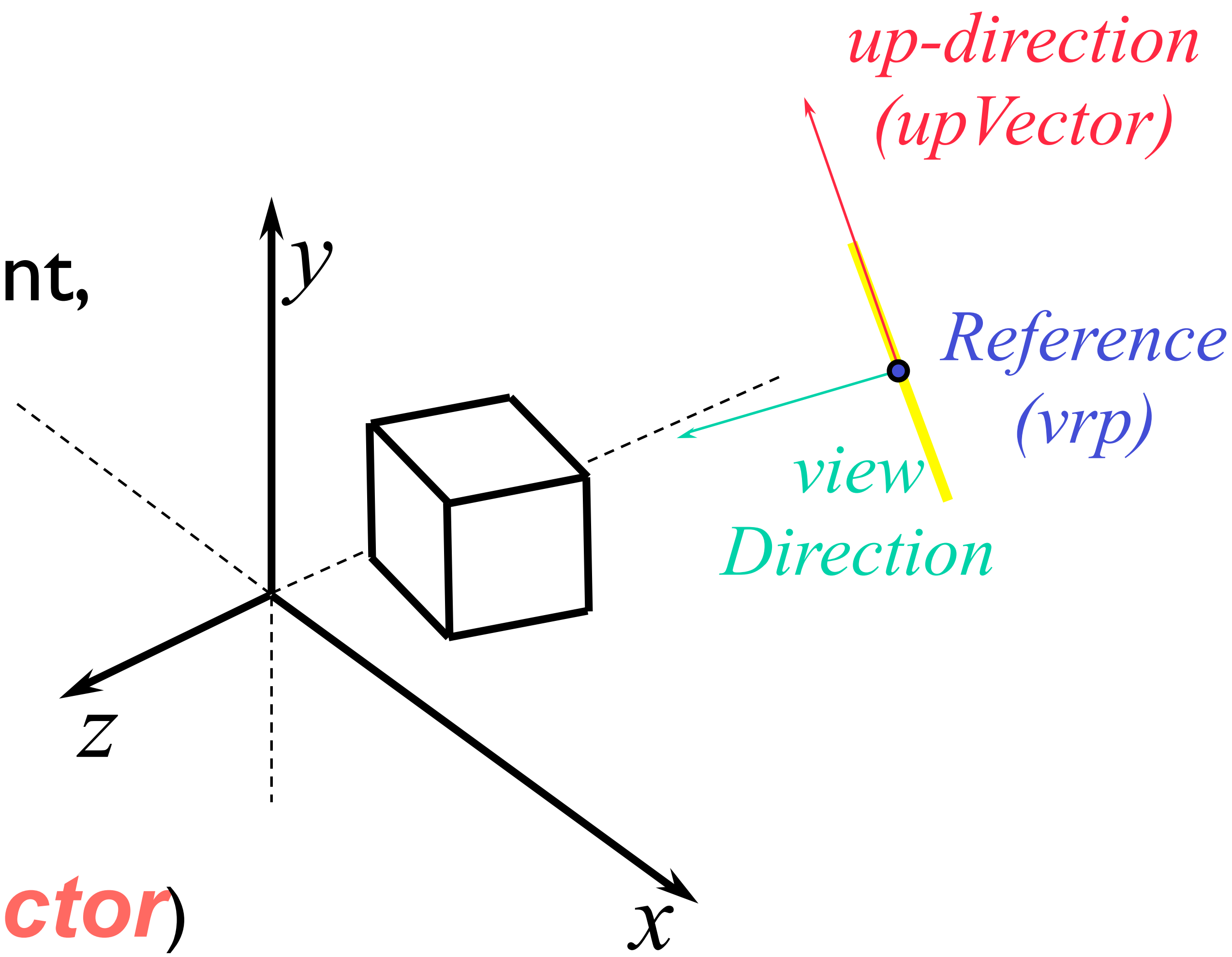
Specify

Focus point or reference point, typically on the object

(***view reference point***)

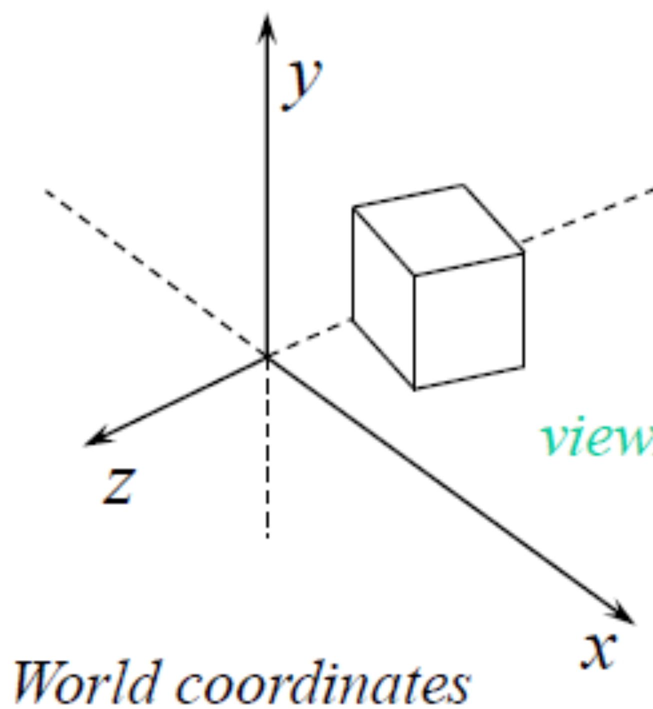
direction of viewing (***viewDirection***)

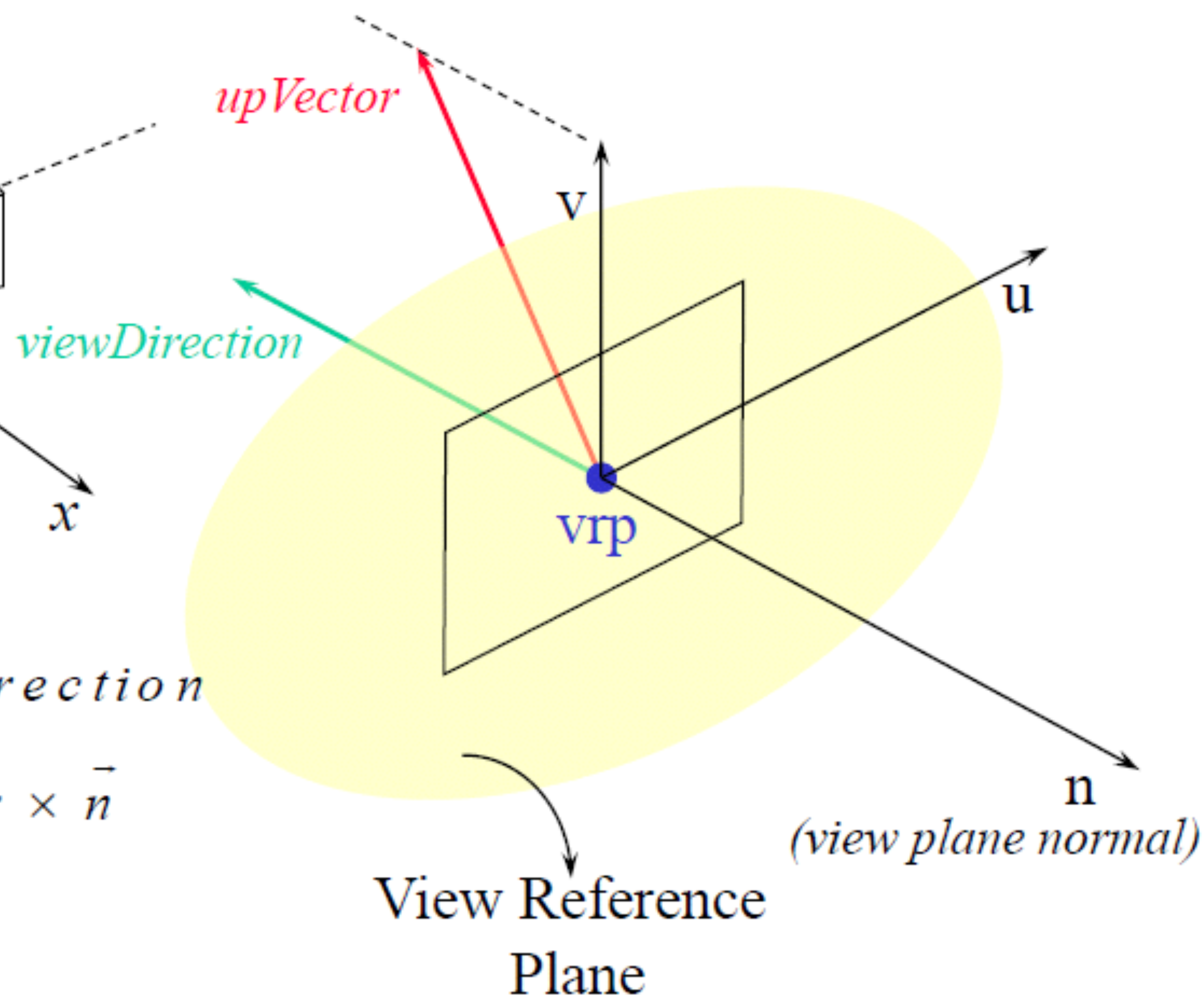
picture's up-direction (***upVector***)



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

View Reference Coordinate System





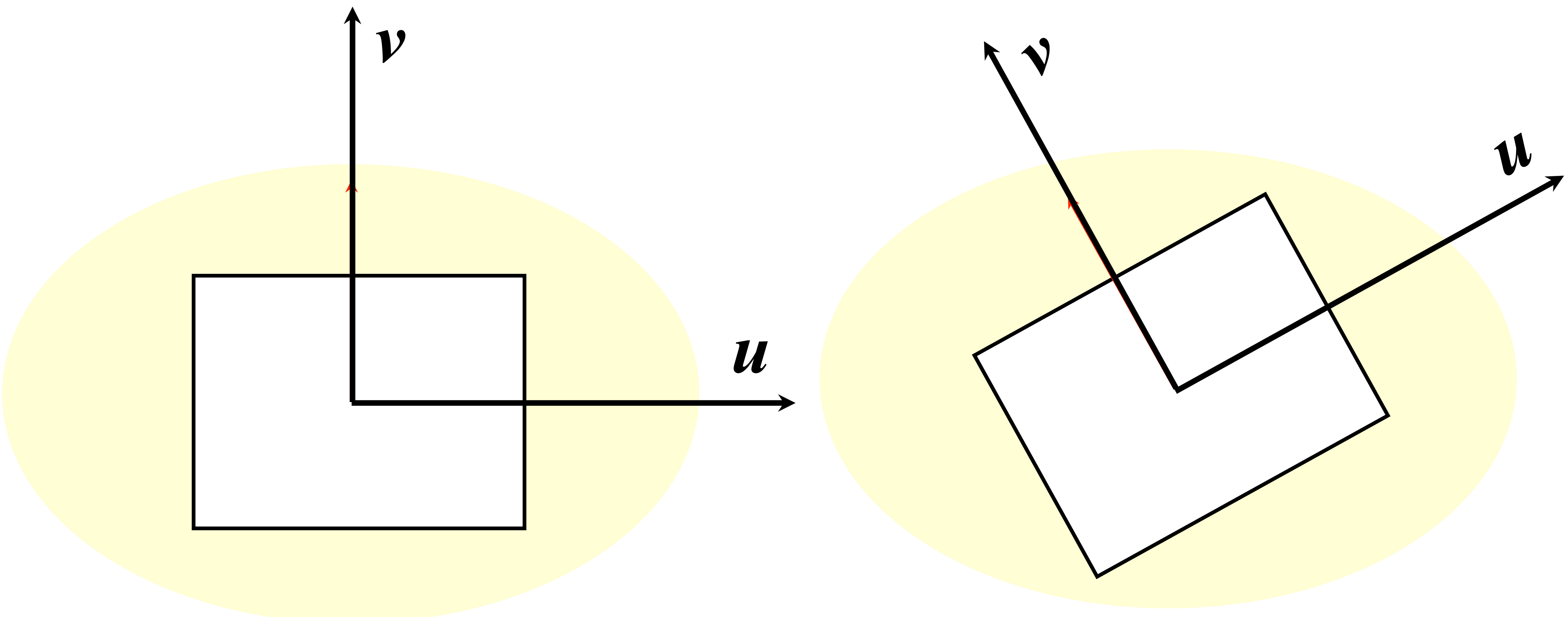
$$\vec{n} = -viewDirection$$

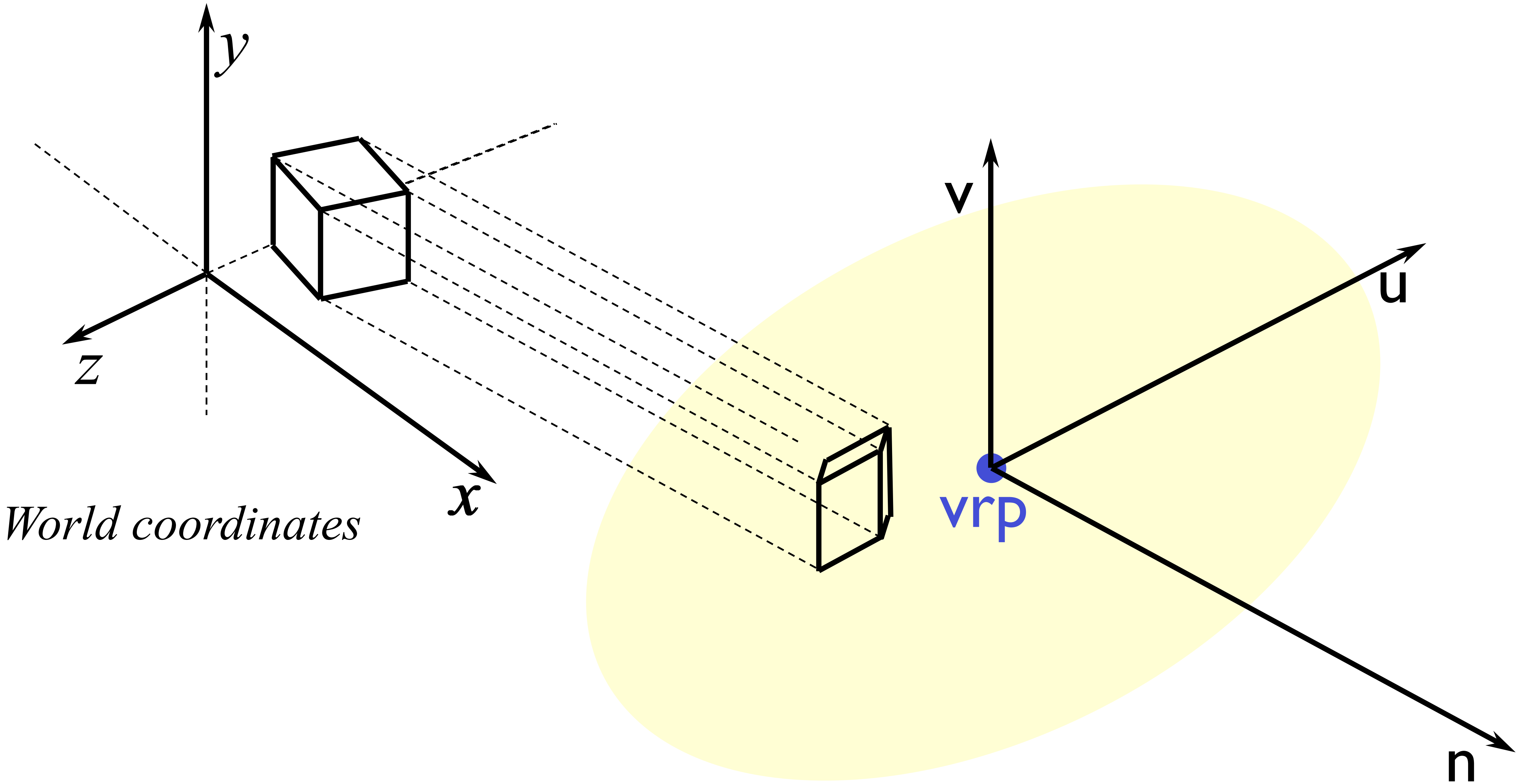
$$\vec{u} = upVector \times \vec{n}$$

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

View Up Vector

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



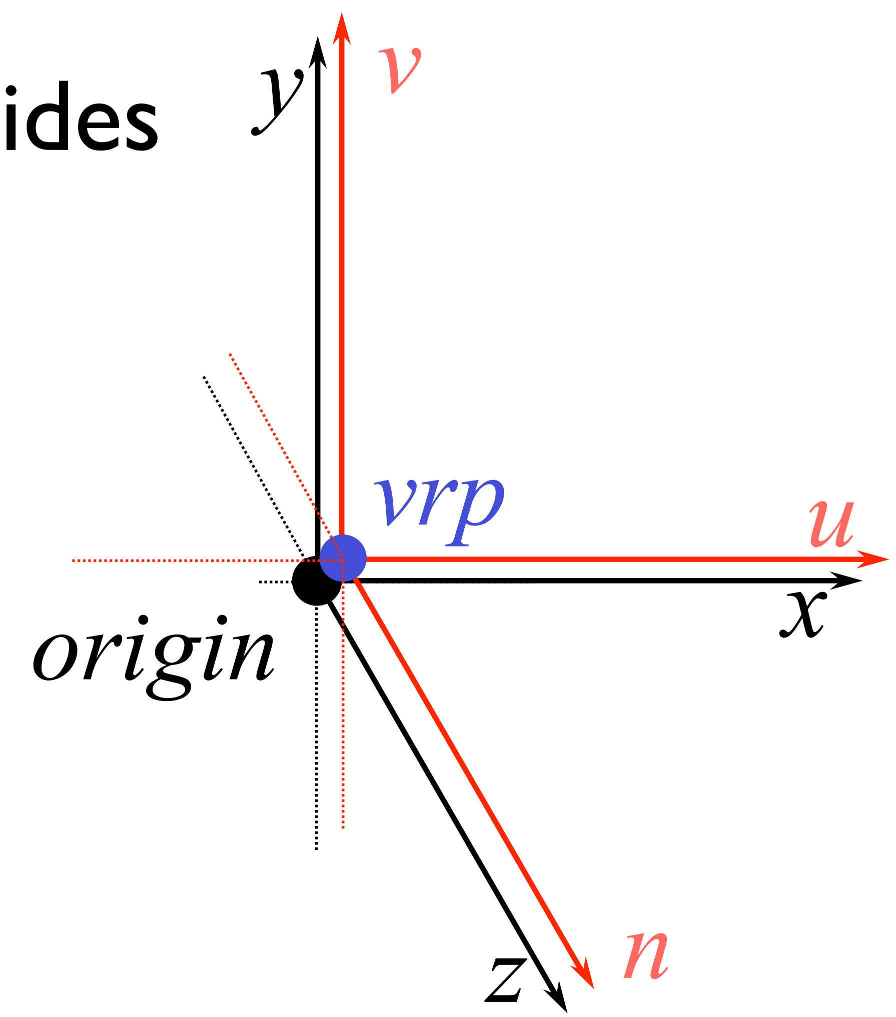


- Once the *view reference coordinate system* is defined, the next step is to project the *3D world* on to the *view reference plane*

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***

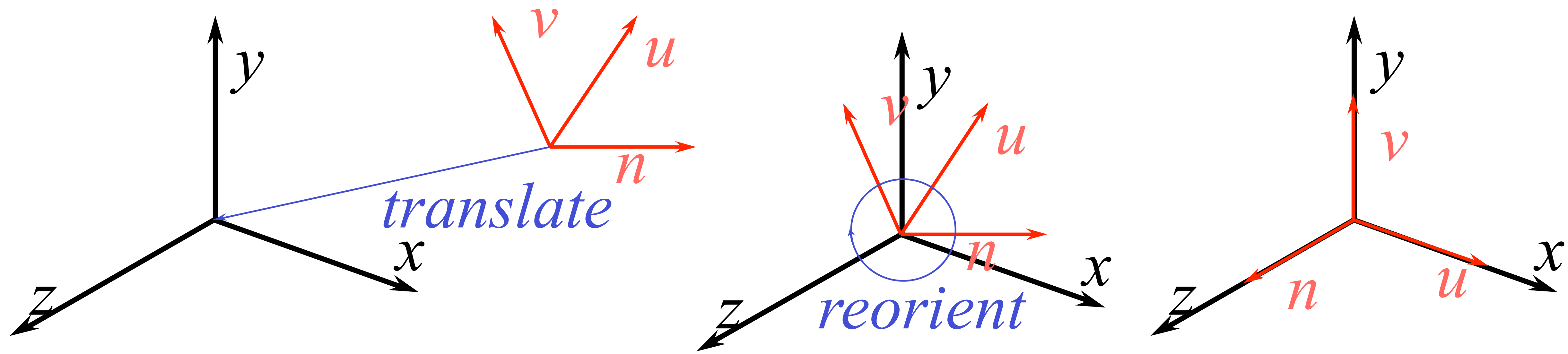


- Projection could be as simple as ignoring the z-coordinate

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***

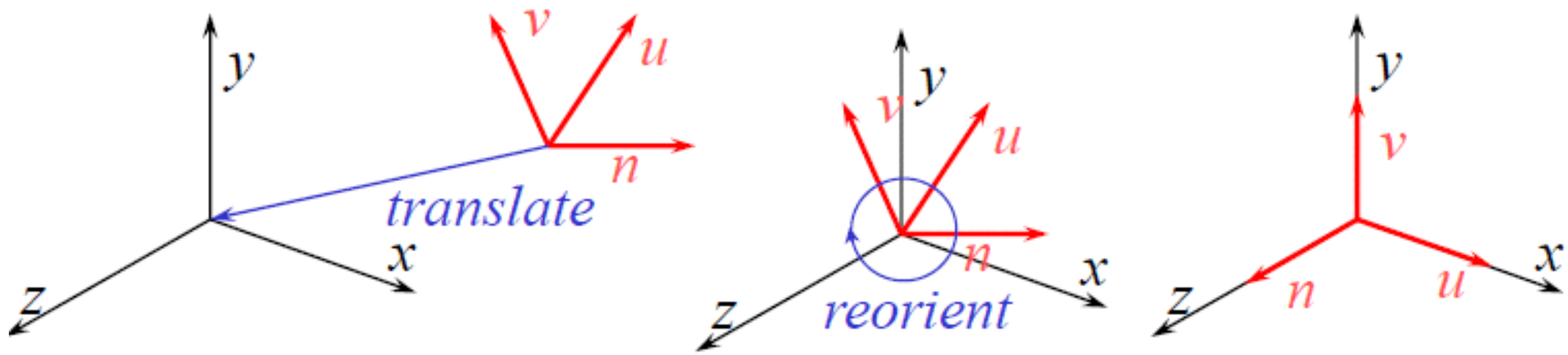
Deriving View Orientation Matrix



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

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

Deriving View Orientation Matrix

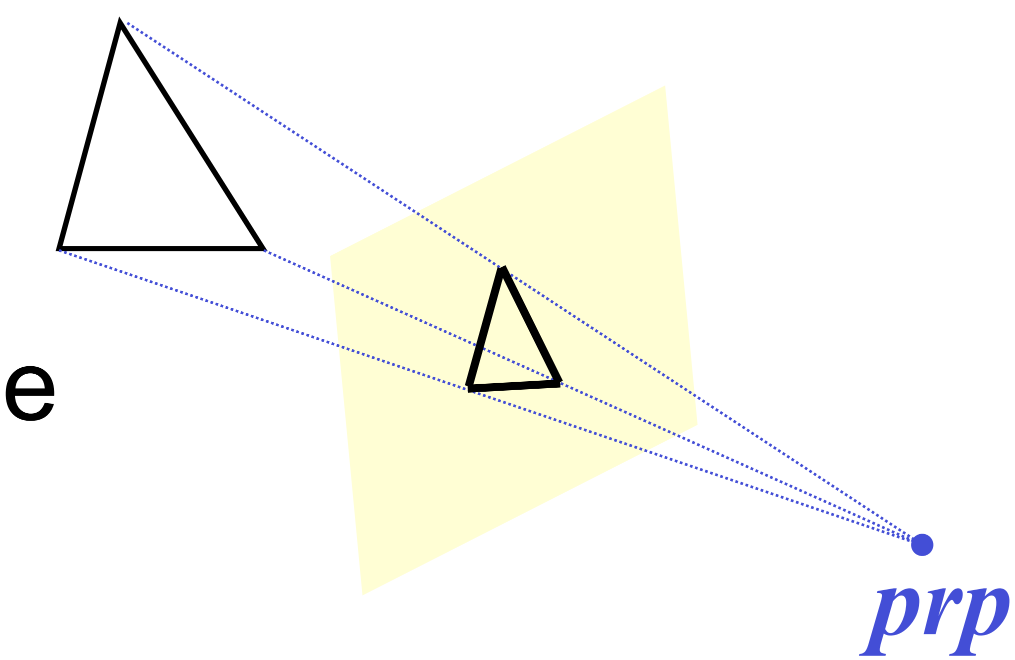


- The ***view orientation matrix*** *transforms* a point from *world coordinates* to *view coordinates*

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

Perspective Projection

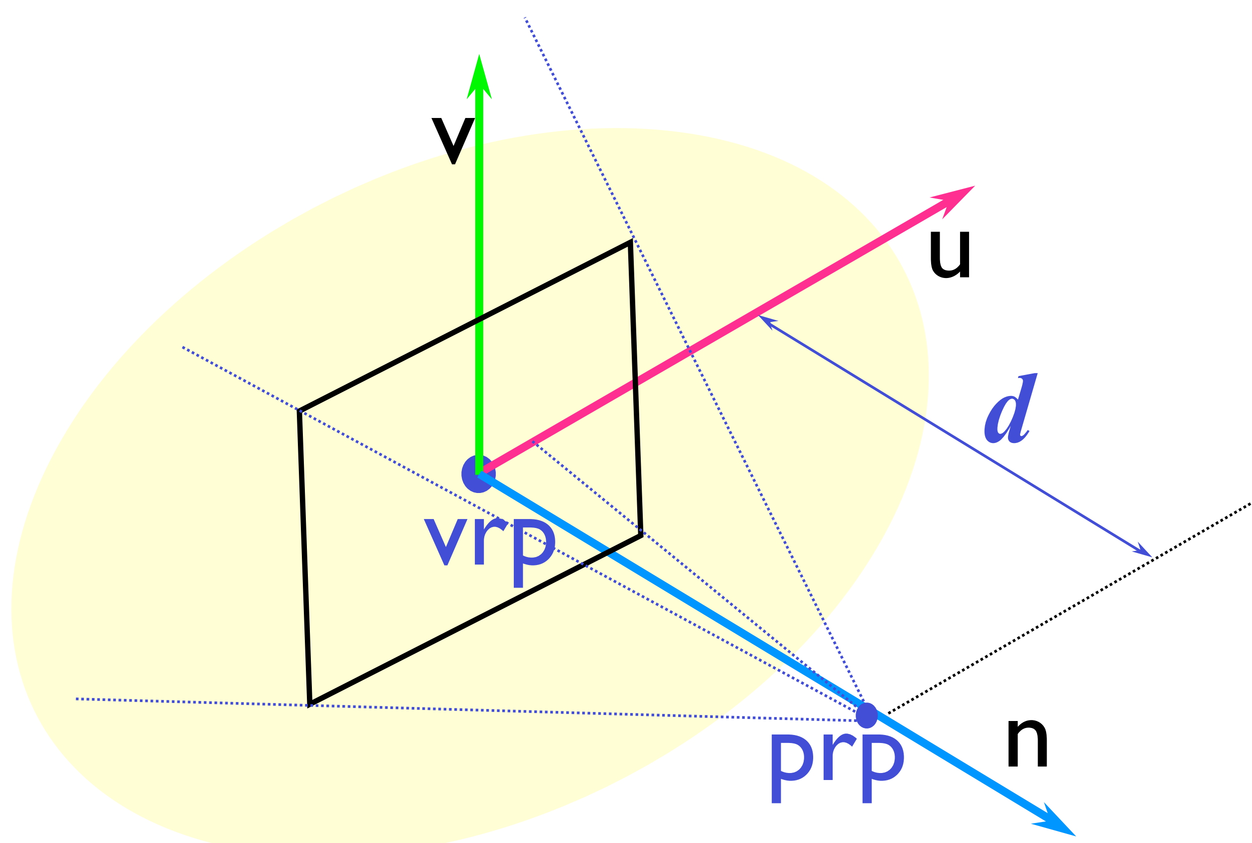
- The points are transformed to the view plane along lines that converge to a point called
  - ***projection reference point*** (***prp***) or
  - ***center of projection*** (***cop***)



- ***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

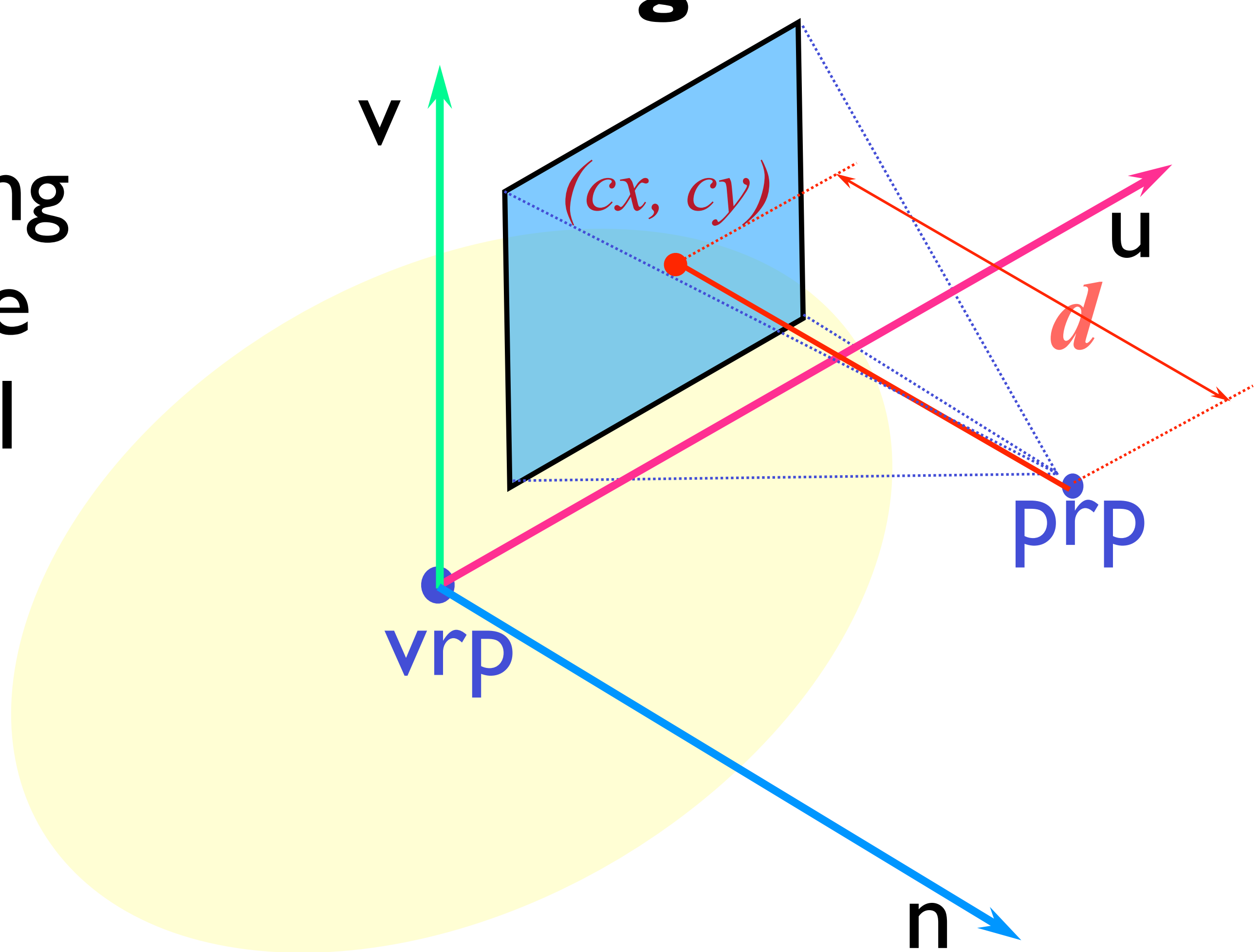


transformation matrix
for *perspective projection*

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

View Window

- ***View window*** is a rectangle in the *view plane* specified in terms of view coordinates.
- Specify ***center (cx, cy), width*** and ***height***
- *prp* lies on the axis passing through the center of the view window and parallel to the n-axis



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

Perspective Viewing

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

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

Parallel Viewing

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

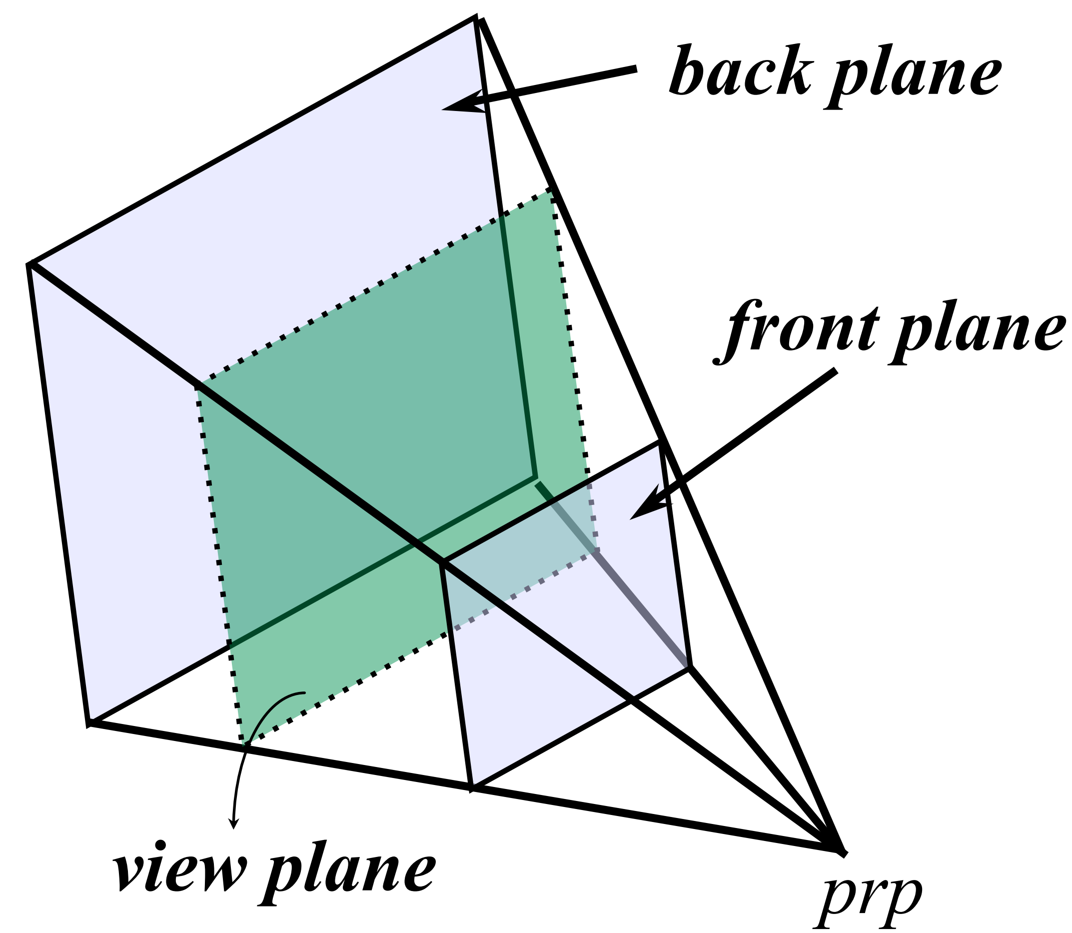
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & u_y & u_z & -\vec{u} \bullet vrp \\ v_x & v_y & v_z & -\vec{v} \bullet vrp \\ n_x & n_y & n_z & -\vec{n} \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

View Volume & Clipping

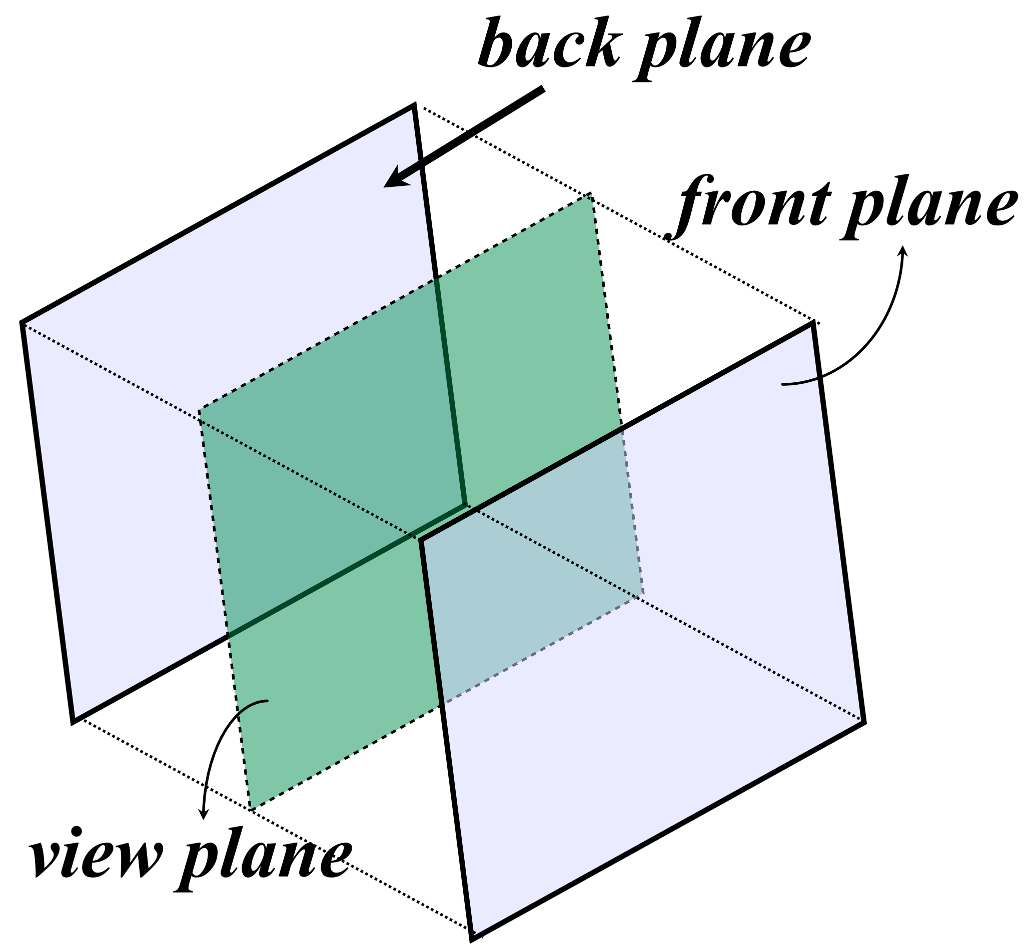
- For *perspective projection* the ***view volume*** is a semi infinite pyramid with apex at ***prp*** and edges passing through the corners of the *view window*

- For efficiency, view volume is made finite by specifying the front and back clipping plane specified as distance from the view plane



- For parallel projection the ***view volume*** is an infinite parallelepiped with sides parallel to the direction of projection

- View volume is made finite by specifying the front and back clipping plane specified as distance from the view plane



- Clipping is done in 3D by clipping the world against the front clip plane, back clip plane and the four side planes

The Complete View Specification

- Specification in world coordinates

  position of viewing (***vrp***), direction of viewing(***-n***),

  up direction for viewing (***upVector***)

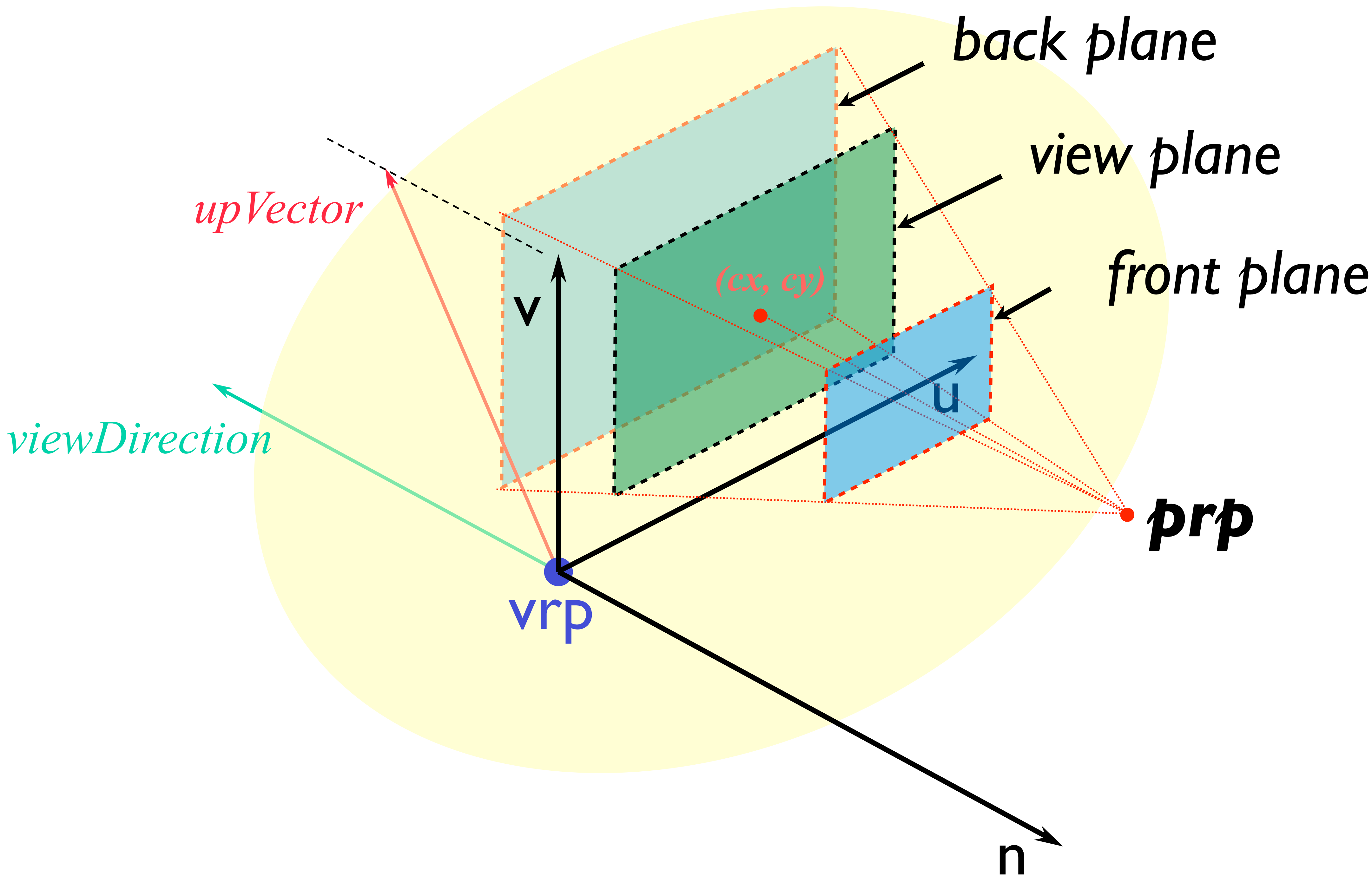
- Specification in view coordinates

  view window : center (***cx, cy***), ***width*** and ***height***,

  prp : distance from the view plane,

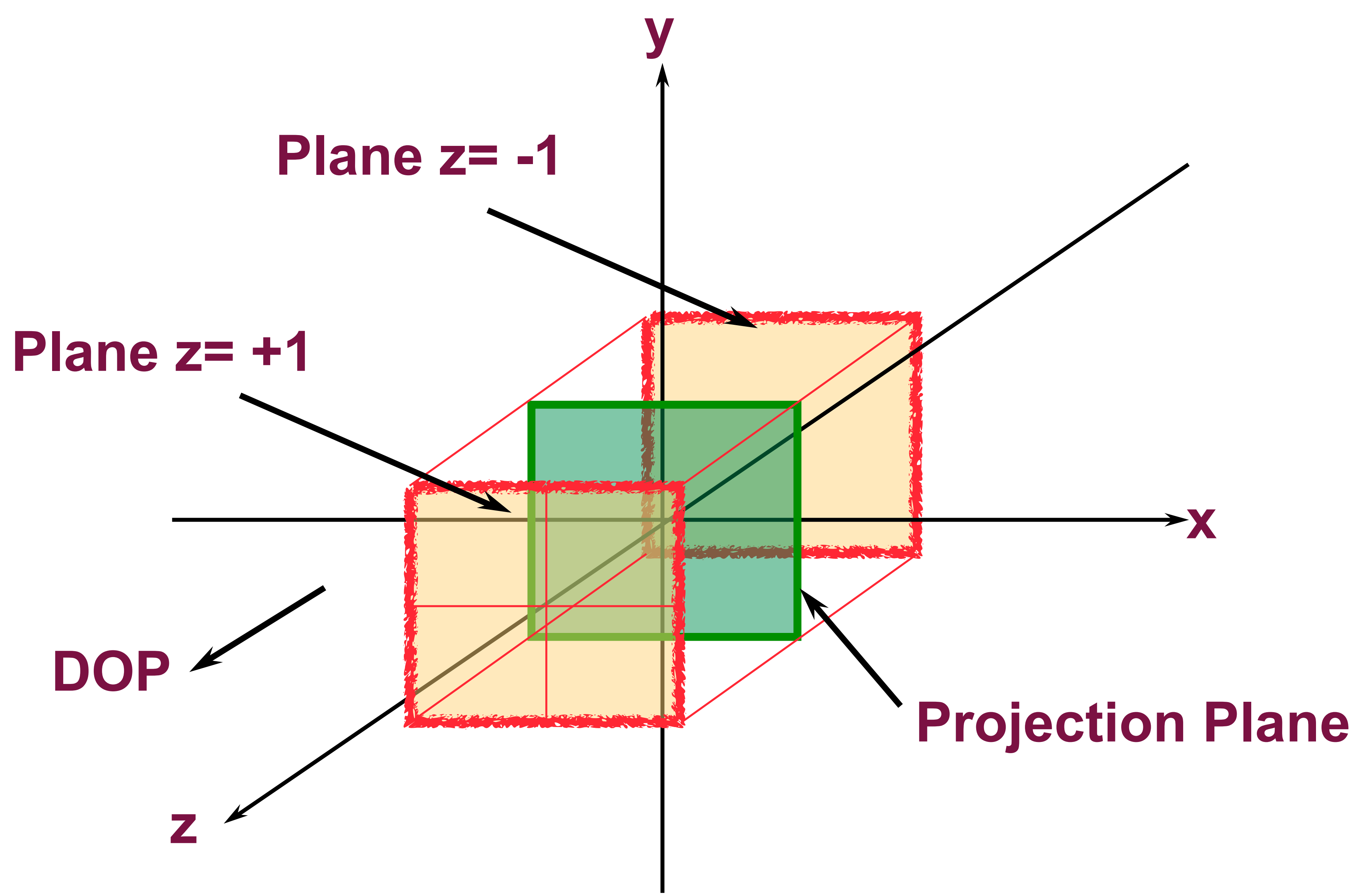
  front clipping plane : distance from view plane

  back clipping plane : distance from view plane


back plane
view plane
front plane
upVector
viewDirection
v
(cx, cy)
u
prp
vrp
n

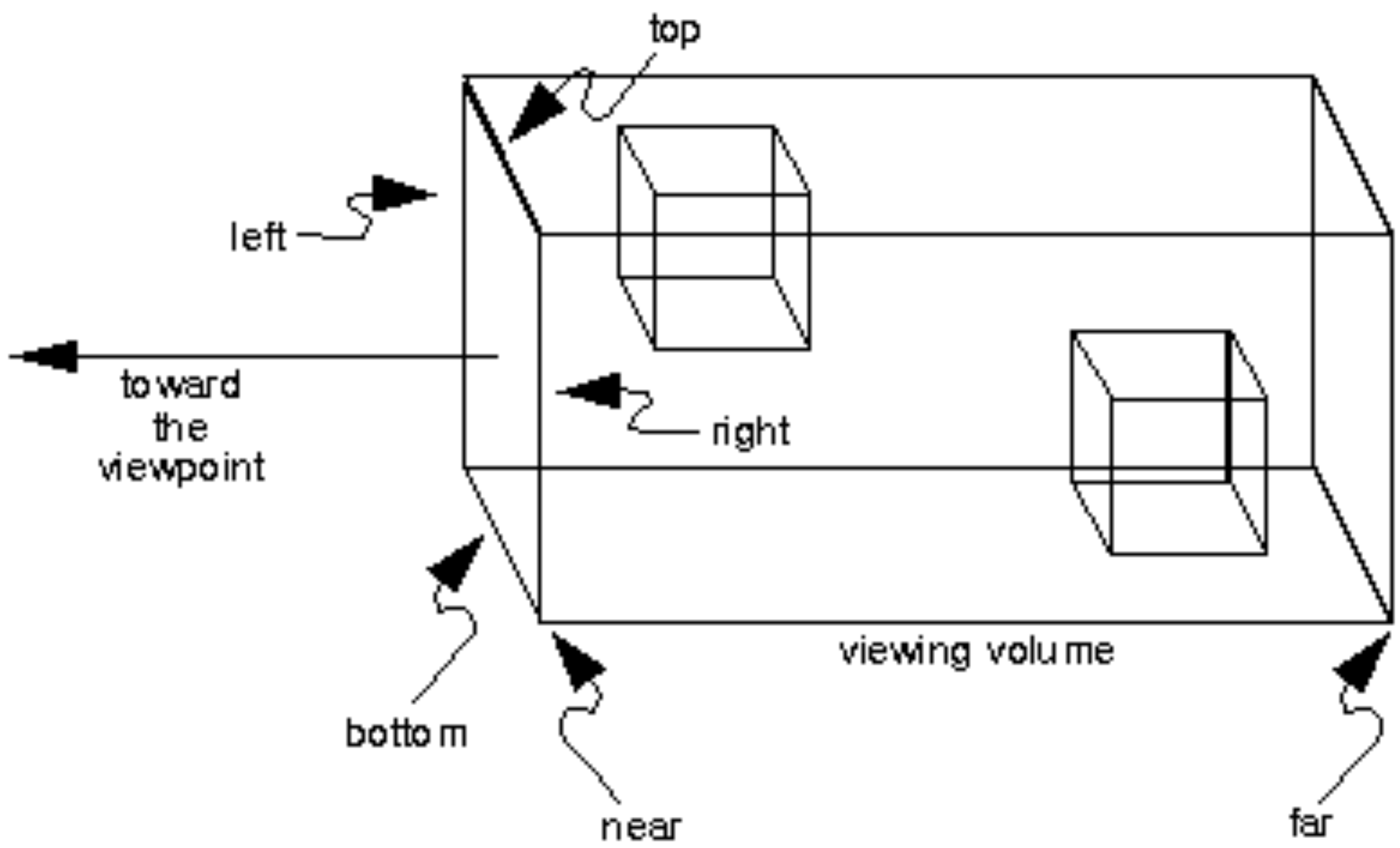
OpenGL

Orthographic Default View Volume



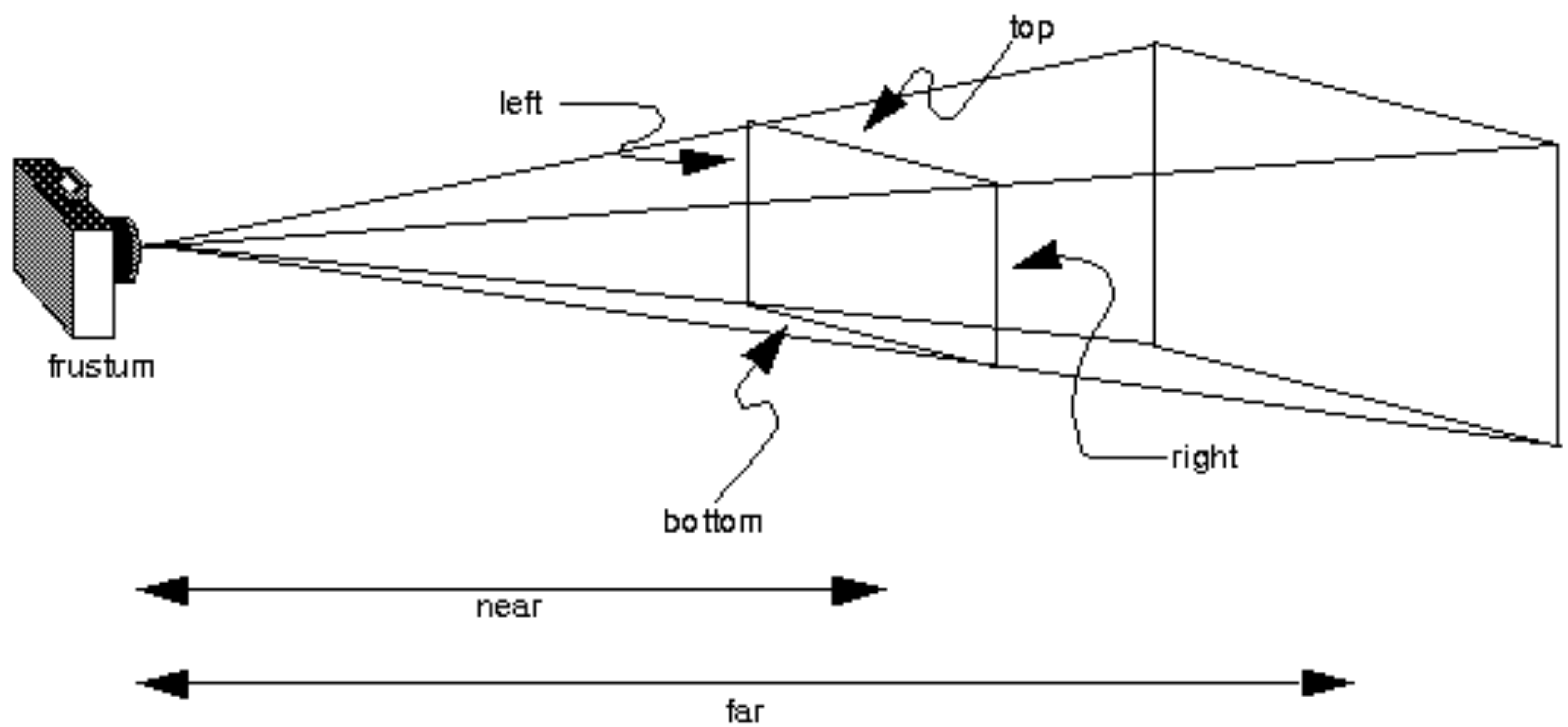
OpenGL Parallel View

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



OpenGL Perspective

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



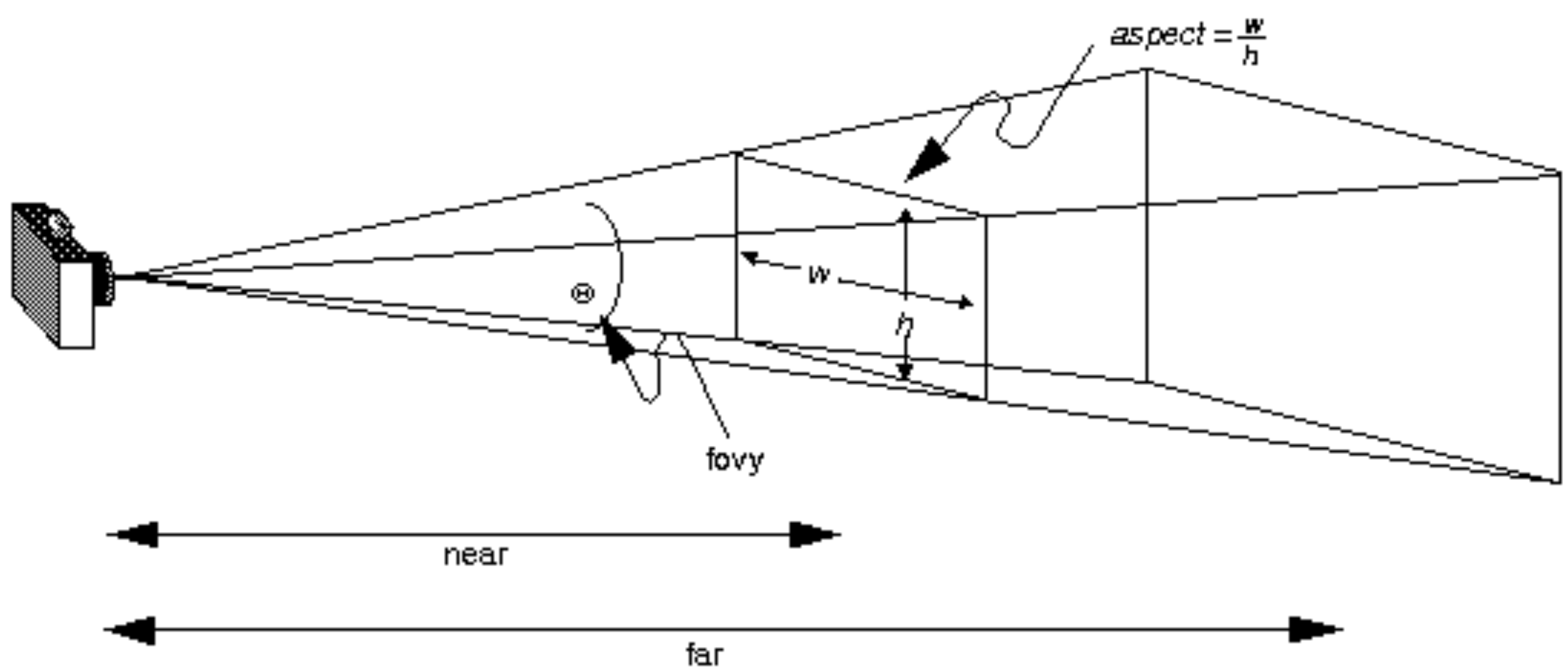

```
glMatrixMode(GL_PROJECTION);

glLoadIdentity( );

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

OpenGL Perspective

gluPerspective(fovy, aspect, near, far);



FOV is the angle between the top and bottom planes

gluPerspective(fovy, aspect, near, far)

```
{
   GLdouble m[4][4];
   double sine, cotangent, deltaZ;
   double radians = fovy / 2 * __glPi / 180;

   deltaZ = zFar - zNear;
   sine = sin(radians);
   if ((deltaZ == 0) || (sine == 0) || (aspect == 0)) {
      return;
   }
   cotangent = COS(radians) / sine;
   gluMakeIdentityd(&m[0][0]);
   m[0][0] = cotangent / aspect;
   m[1][1] = cotangent;
   m[2][2] = -(zFar + zNear) / deltaZ;
   m[2][3] = -1;
   m[3][2] = -2 * zNear * zFar / deltaZ;
   m[3][3] = 0;
   glMultMatrixd(&m[0][0]);
}
```

A More Intuitive Approach Offered by GLU

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

eyex, eyey, eyez specify the position of the eye point
and are mapped to the origin.

atx, aty, atz specify a point being looked at, which will
be rendered in center of view port. It is
mapped to the -z axis.

upx, upy, upz specify components of the camera up vector.

gluLookAt

```
{
    float forward[3], side[3], up[3];
    GLfloat m[4][4];
    forward[0] = centerx - eyex;
    forward[1] = centery - eyey;
    forward[2] = centerz - eyez;
    up[0] = upx;
    up[1] = upy;
    up[2] = upz;
    normalize(forward);
    /* Side = forward x up */
    cross(forward, up, side);
    normalize(side);
    /* Recompute up as: up = side x forward */
    cross(side, forward, up);

    __gluMakeIdentityf(&m[0][0]);
    m[0][0] = side[0];
    m[1][0] = side[1];
    m[2][0] = side[2];

    m[0][1] = up[0];
    m[1][1] = up[1];
    m[2][1] = up[2];

    m[0][2] = -forward[0];
    m[1][2] = -forward[1];
    m[2][2] = -forward[2];

    glMultMatrixf(&m[0][0]);
    glTranslated(-eyex, -eyey, -eyez);
}
```

Homework 03

- How to stitch 2x2 iPad screen (2048 x1536) together to create a larger OpenGL viewport (4K)
  - requirement:
    - detailed computing steps
  - bonus:
    - implemented demo

THANK YOU