View in 2D & 3D

Teacher: A.prof. Chengying Gao(高成英)

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

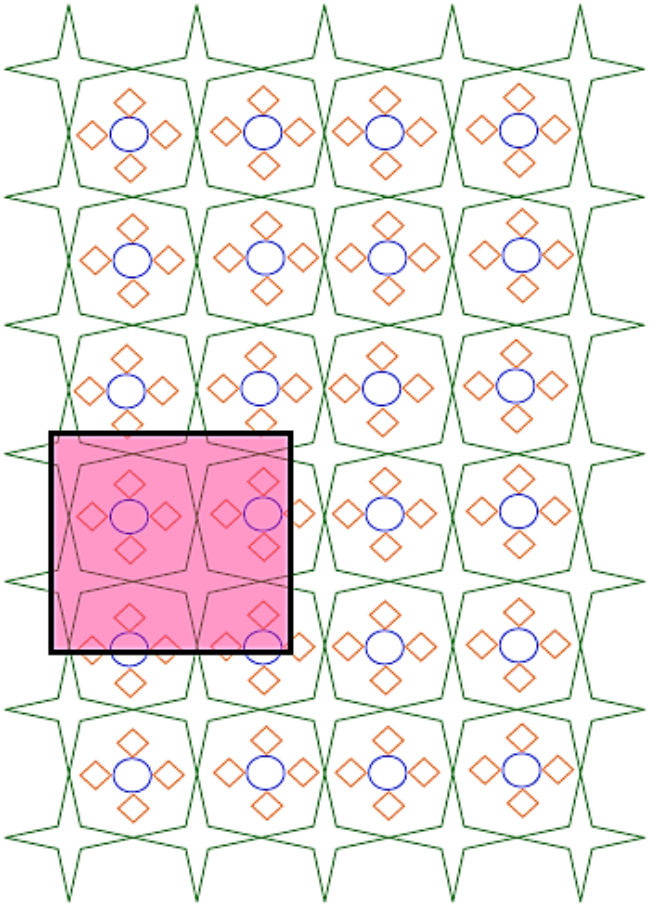
School of Data and Computer Science

Outline

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

Viewing Transformation

- 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

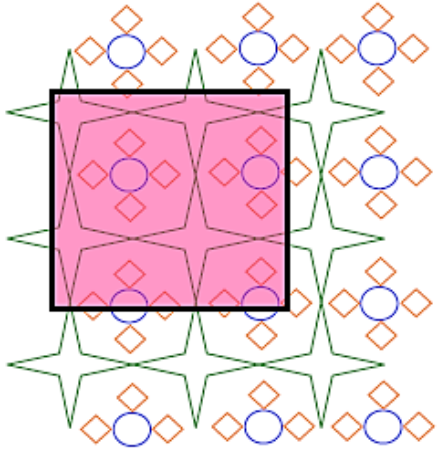


2D Viewing Transformation

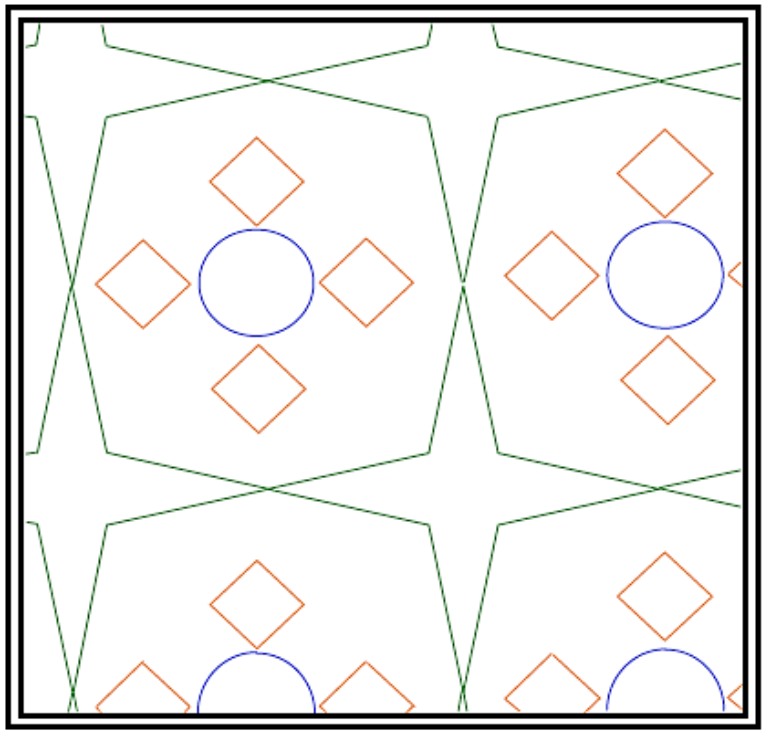
- We specify a rectangular area in the modeling coordinates (world coordinates) and a viewport in the device coordinates on the display
  - window defines what to appear
  - viewport defines where to display
- The mapping of the window (modeling coordinates) to viewport (device coordinates) is a 2D viewing transformation

2D Viewing Transformation

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

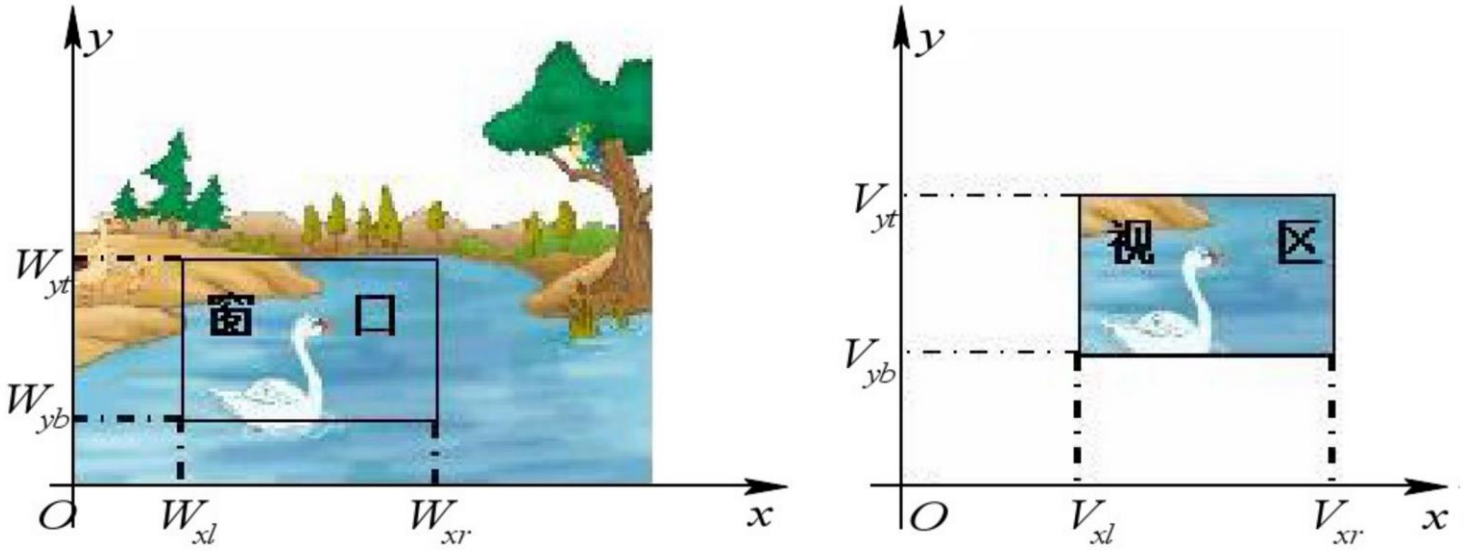


2D World



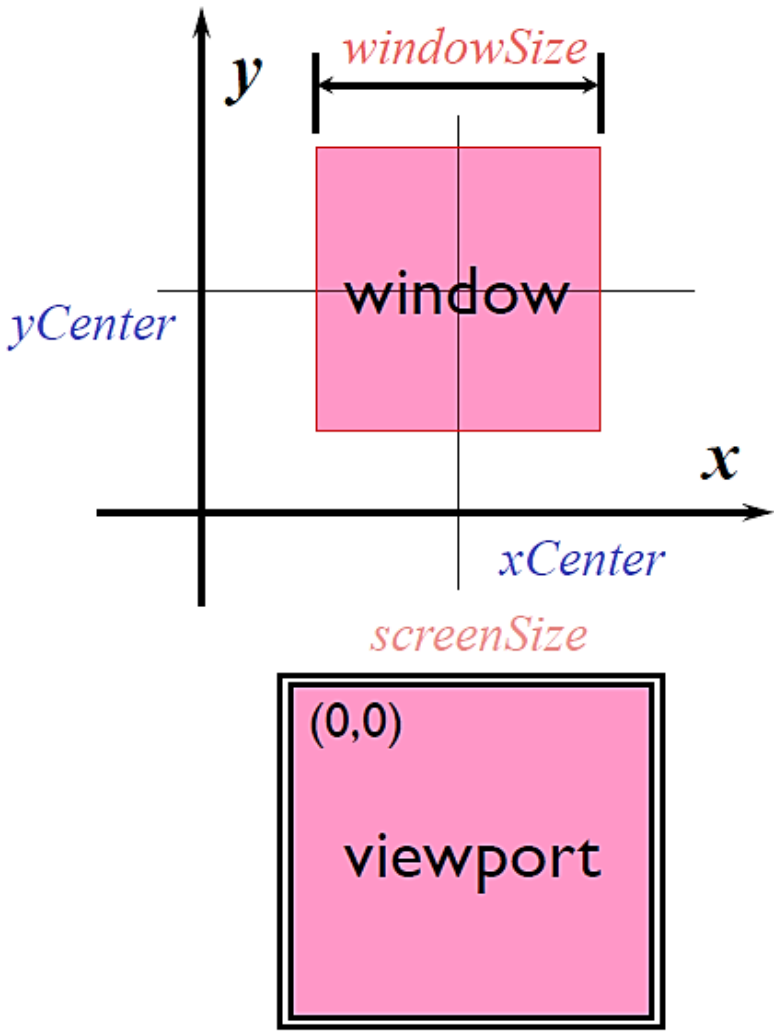
Screen

2D Viewing Transformation



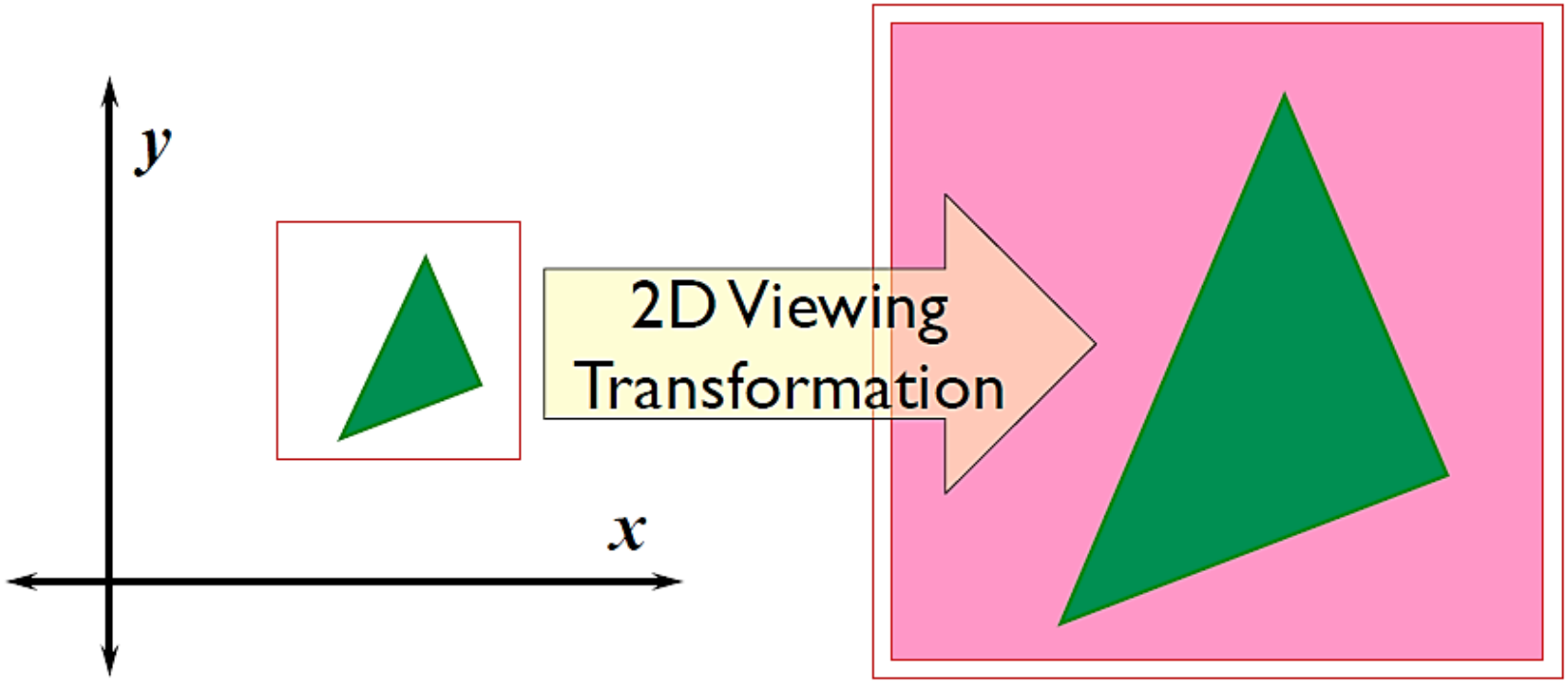
2D Viewing Transformation

- ***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)

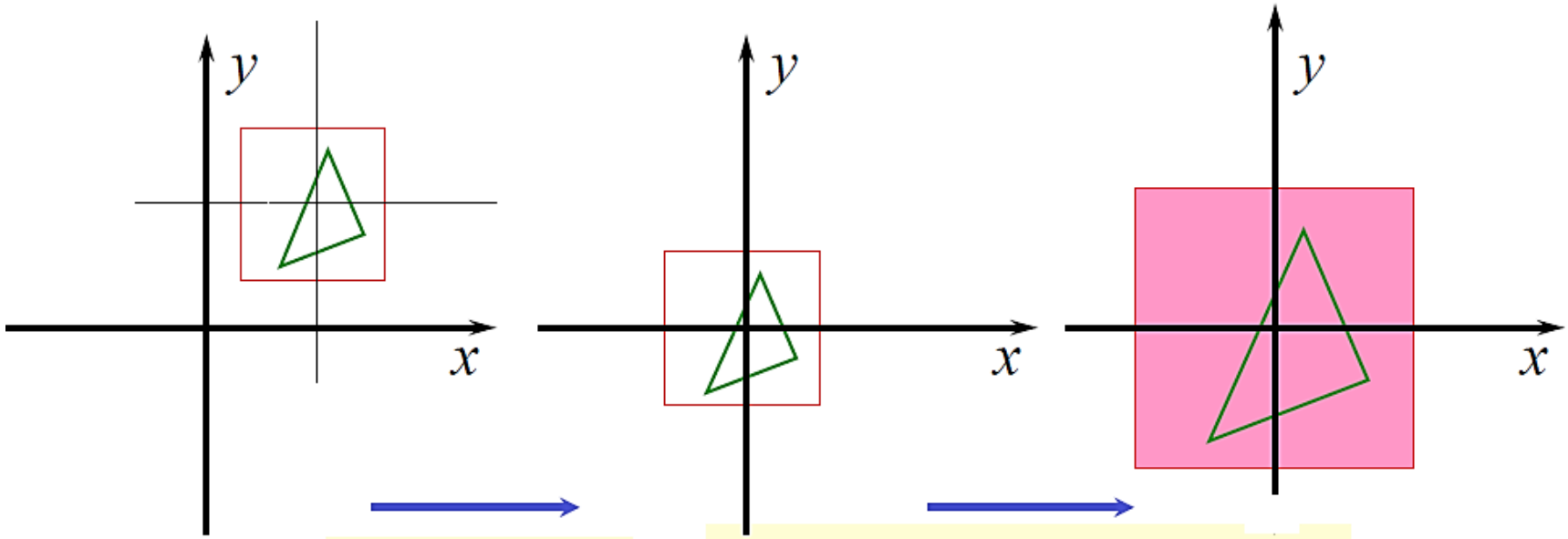


2D Viewing Transformation

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



Deriving Viewport Transformation

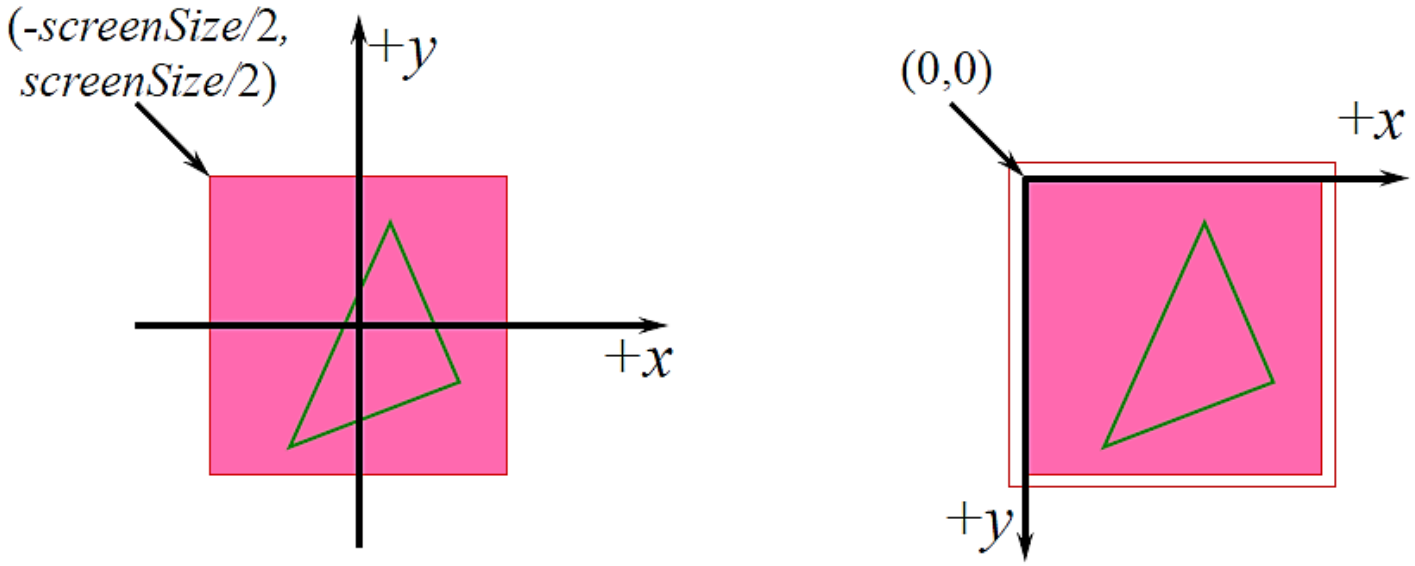


$$x - xCenter$$
$$y - yCenter$$

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

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

Deriving Viewport Transformation



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

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

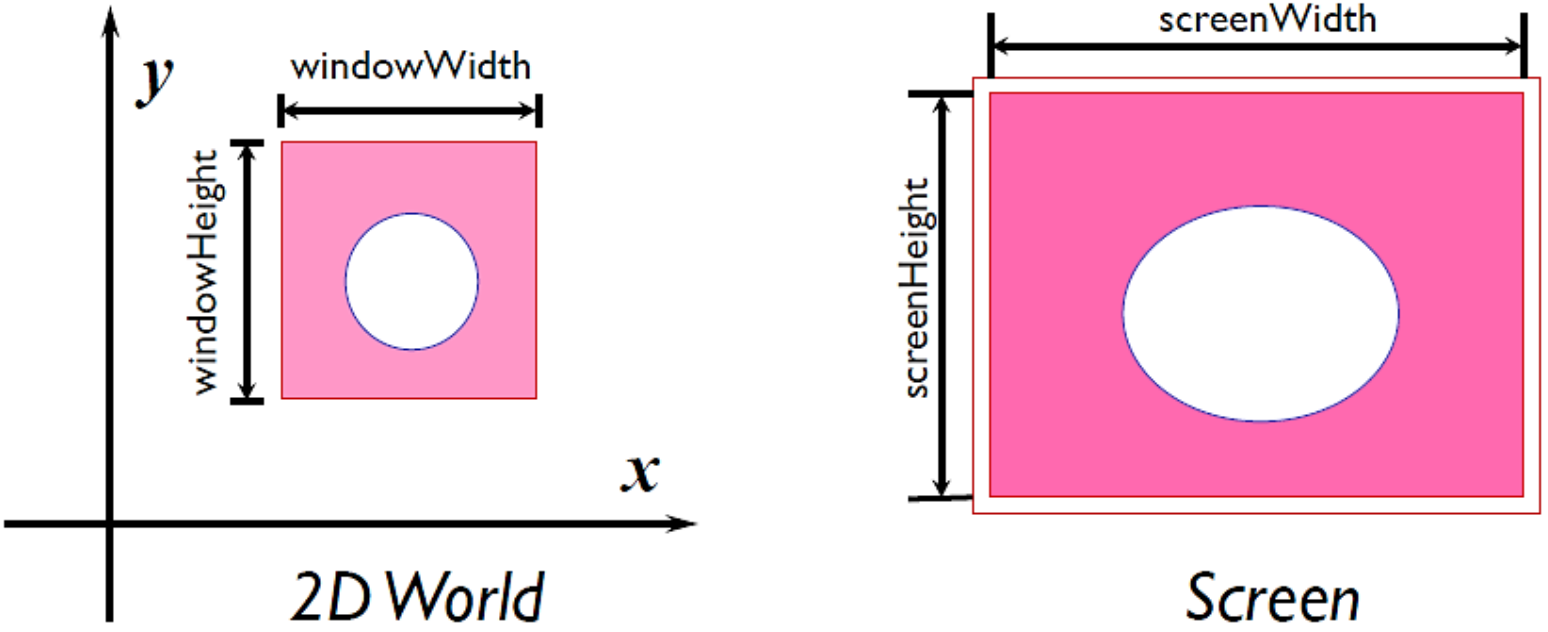
- Given any point in the 2D world, the above transformations maps that point on to the screen

Windows & Viewport

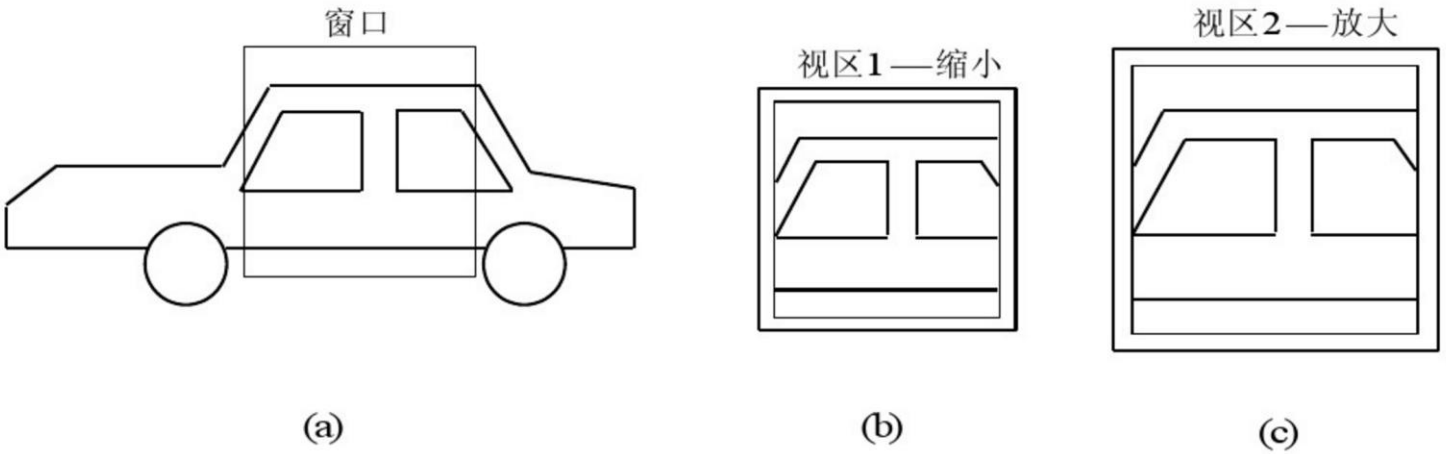
- 窗口与视区的形状相似，即二者的长与宽之比相同，变换后在视区产生均匀缩小或均匀放大的图形。
- 窗口与视区的形状不相似，即二者的长与宽之比不相等，变换后在视区产生畸变的图形。 图形将沿水平及垂直方向以不同比例发生变化——畸变。
- 窗口斜置，即窗口绕坐标原点旋转一个角度，变换后视区的图形也相应地旋转一个角度。

The Aspect Ratio (纵横比)

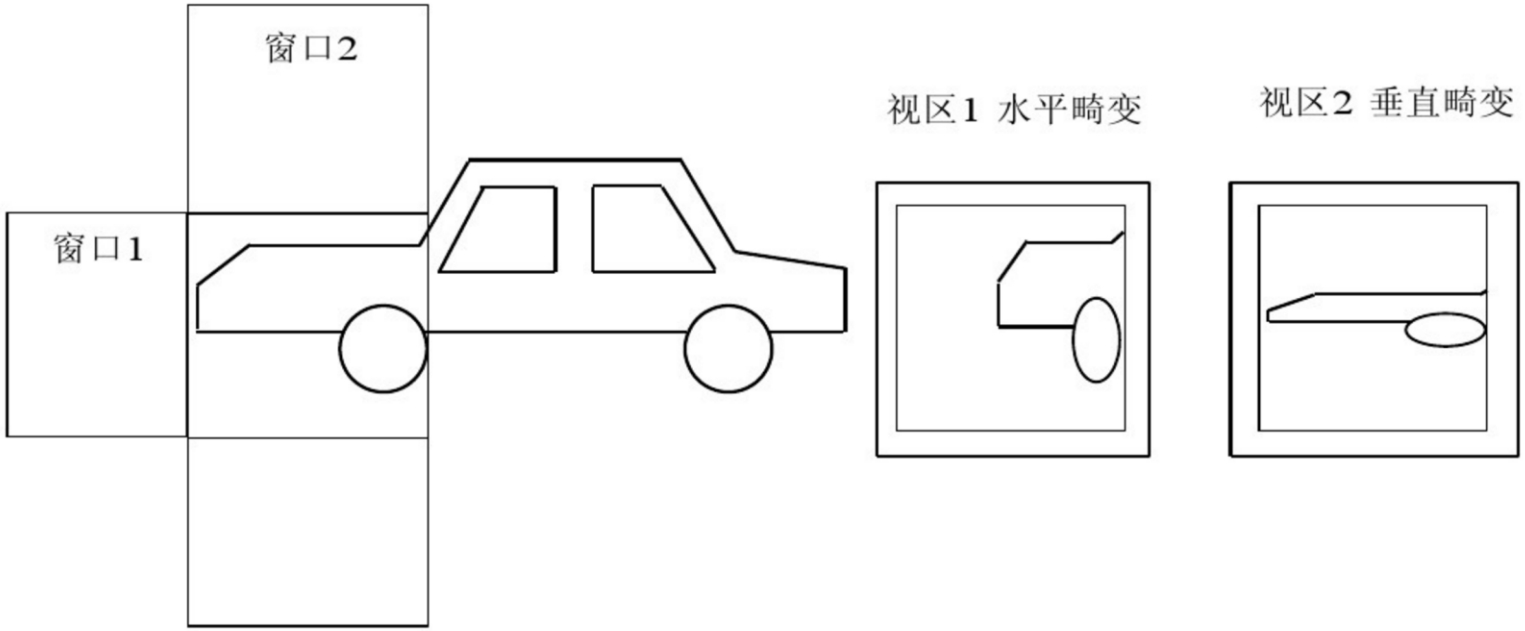
- In 2D viewing transformation the ***aspect ratio*** is maintained when the scaling is uniform
- ***scaleFactor*** is same for both x and y directions



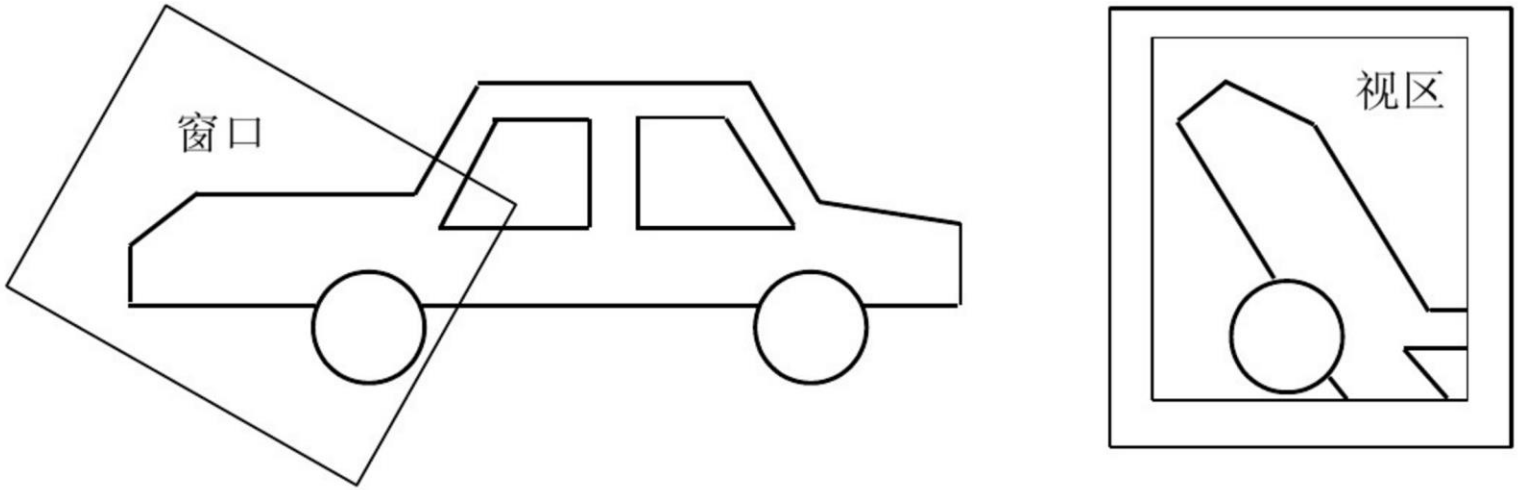
Maintaining the Aspect



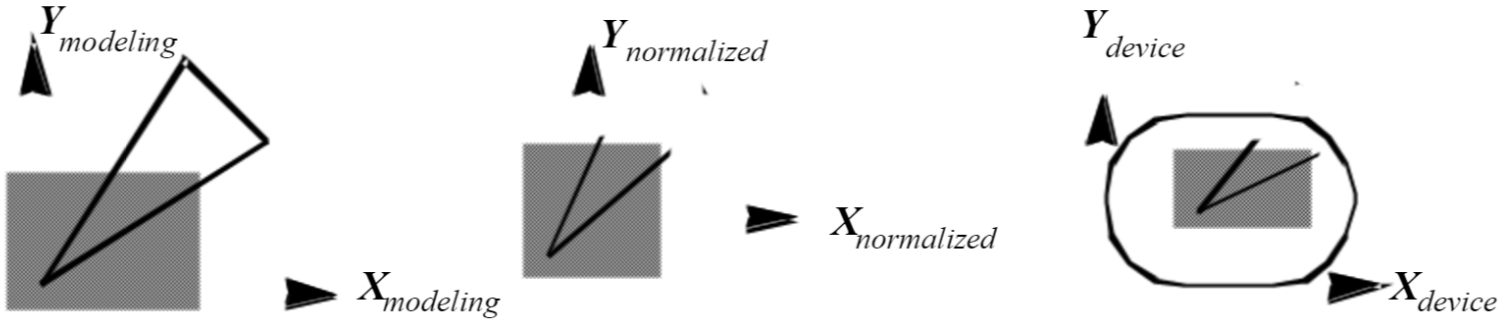
Unmaintaining the Aspect



窗口斜置



The 2D Viewing Pipeline



Specify a window in the modeling coordinates: area to be displayed

→

Transform the window and the models to the normalized coordinates.

Clip against the square

→

Transform the square and the models to the device coordinates in the display viewport.

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.

glOrtho2D是窗口变换，设置窗口的。二维绘图来说窗口由gluOrth2D()设定;
glViewport是视口变换，设置视口的。它设置的视口的左下角，以及宽度和高度。
它负责把视景体截取的图像按照怎样的高和宽显示到屏幕上。

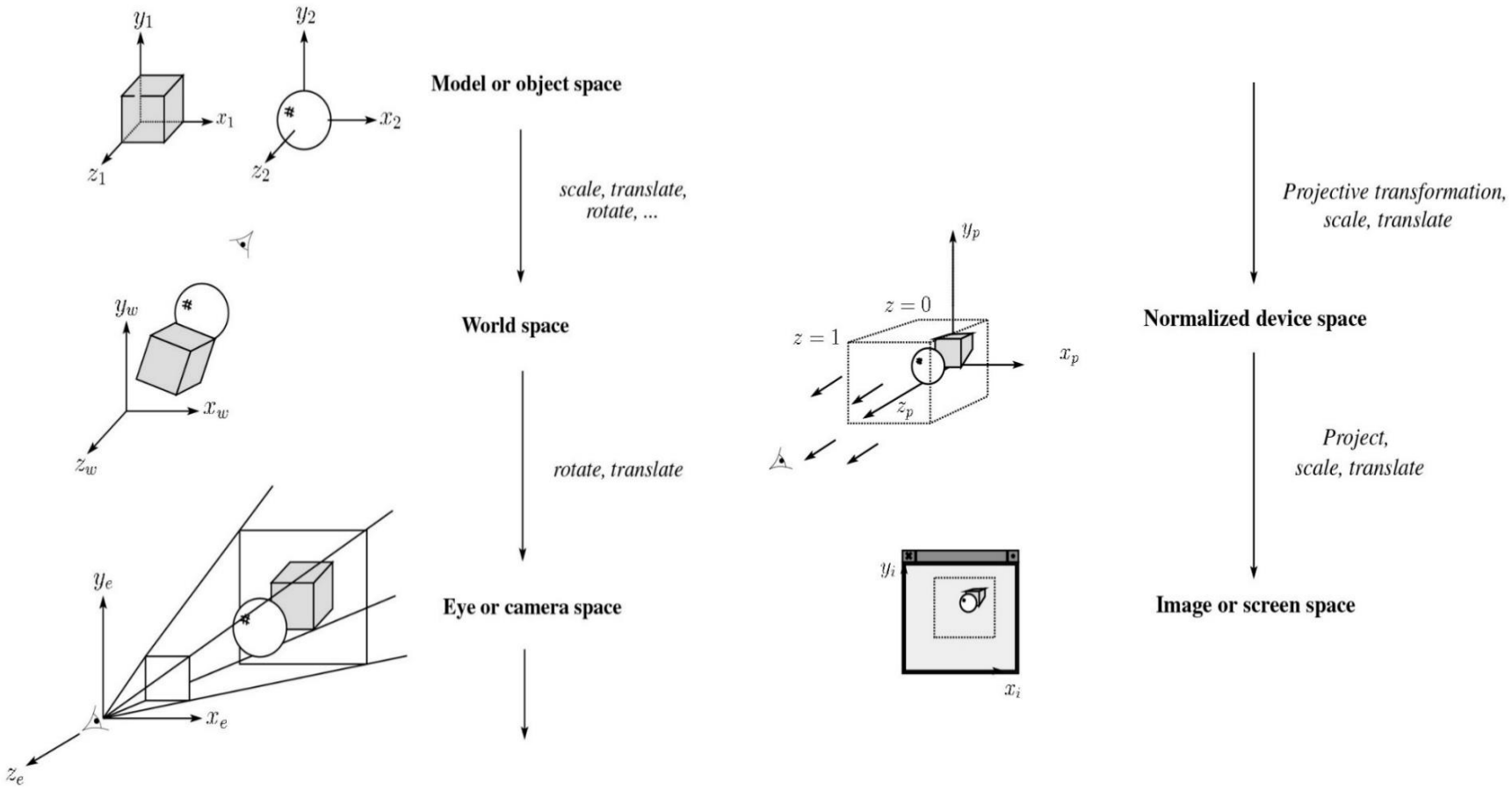
Outline

- 2D Viewing Transformation
- 3D Viewing Transfromation
  - Positioning the camera
  - Projection

3D Viewing Transformation

- 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

3D Geometry pipeline

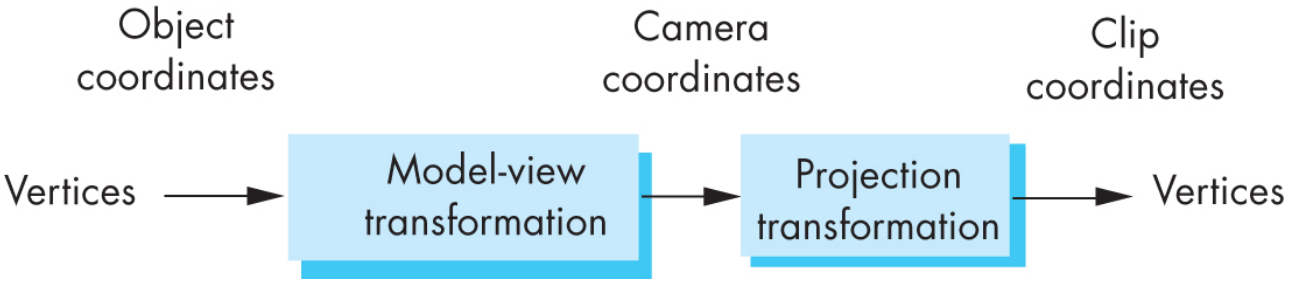


Transformation and Camera Analogy

- **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

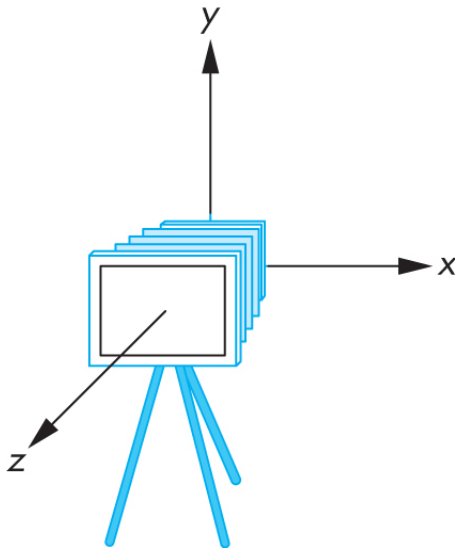
Computer view

- 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



Outline

- 2D Viewing Transformation
- 3D Viewing Transformation
  - **Positioning the camera**
  - Projection

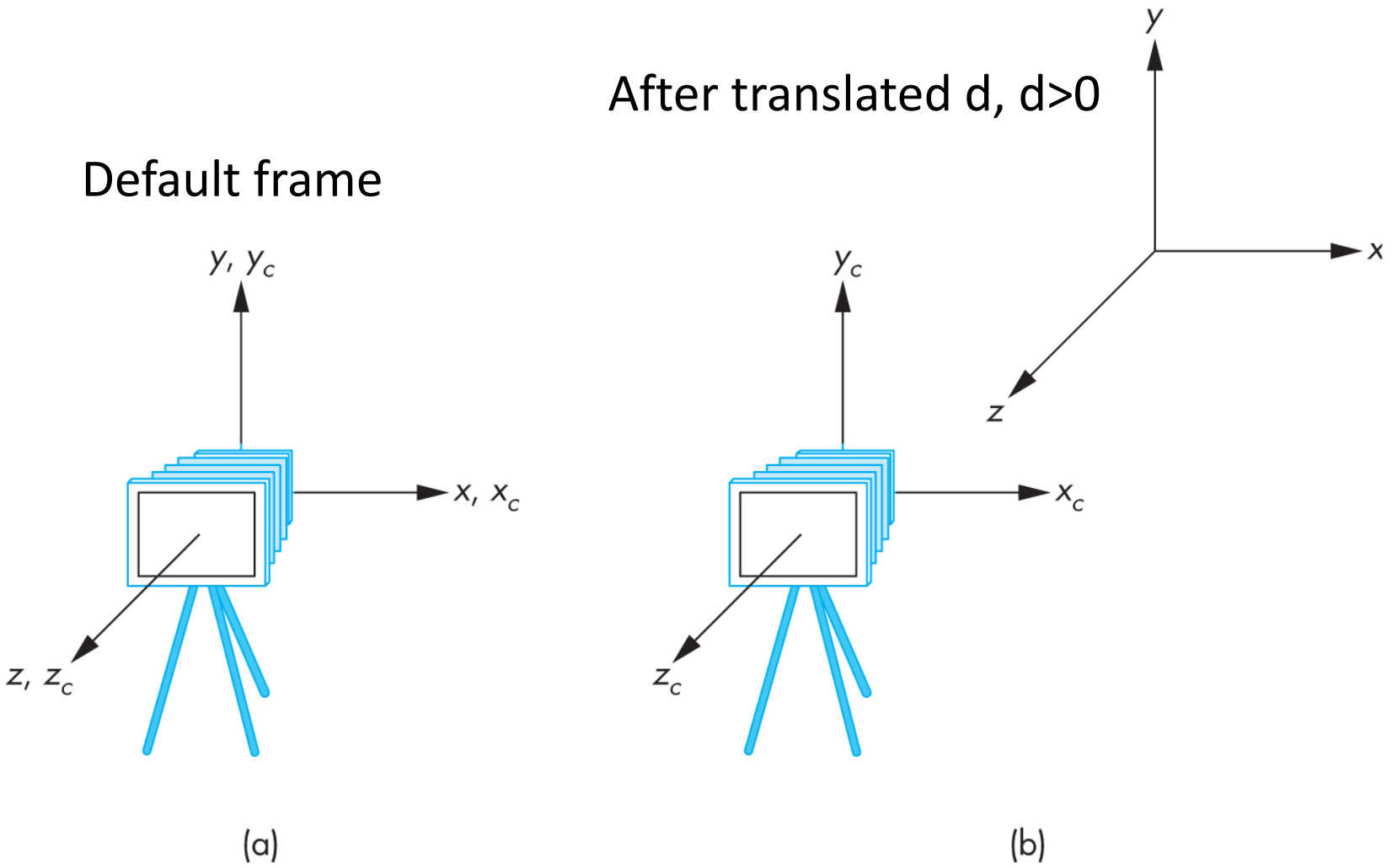
Moving the camera frame

- 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

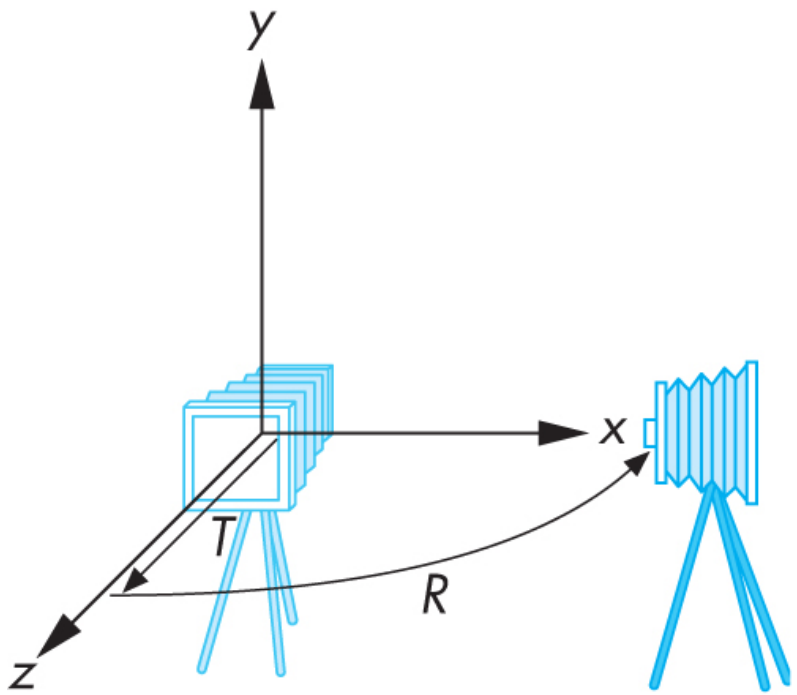
Default frame

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

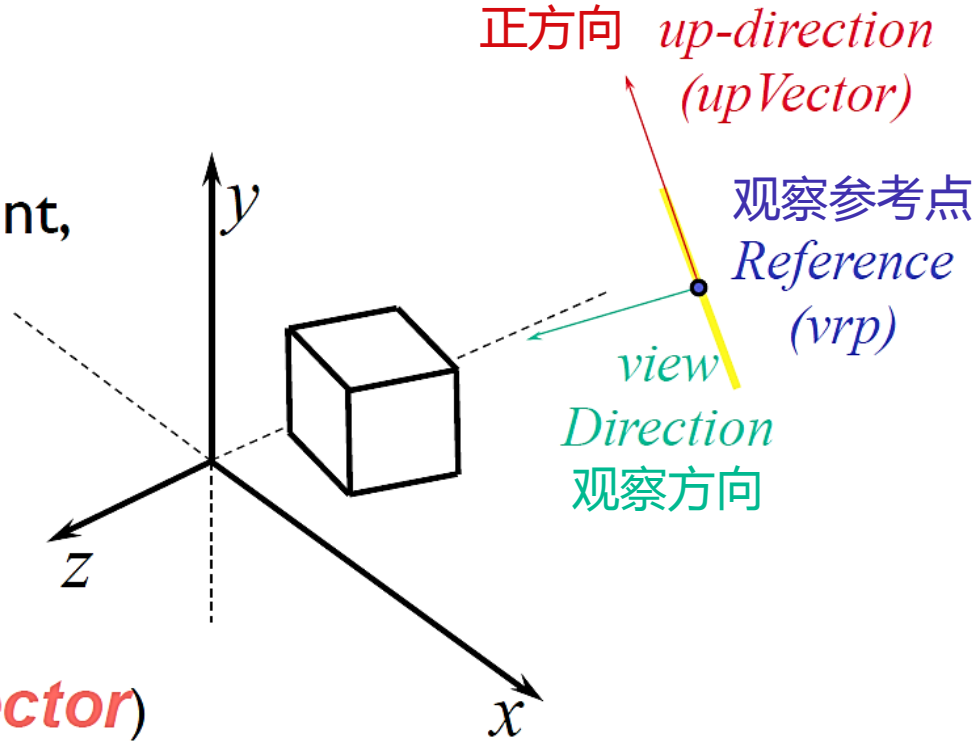
Specify

Focus point or reference point, typically on the object

(***view reference point***)

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

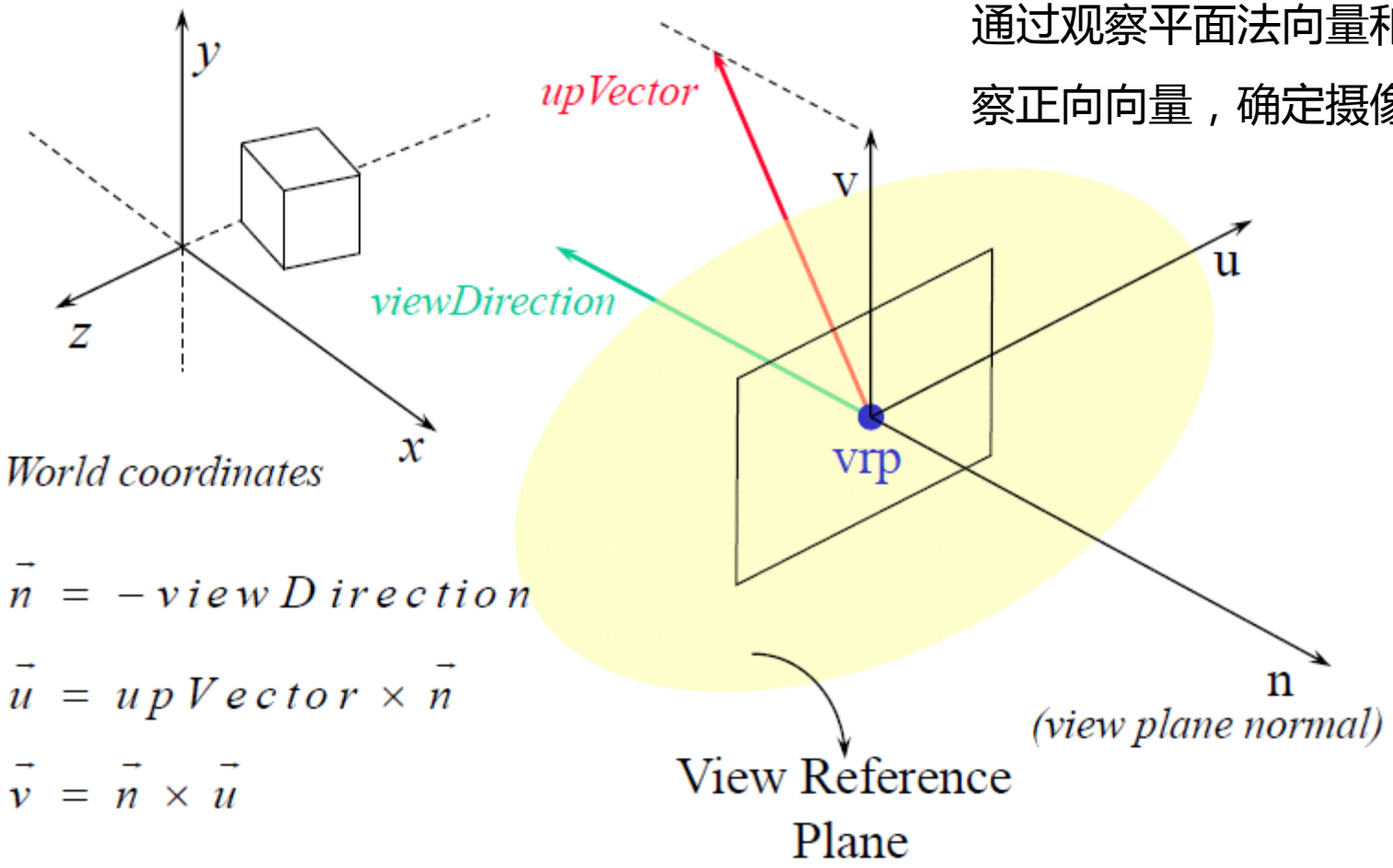
picture's up-direction (***upVector***)



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。

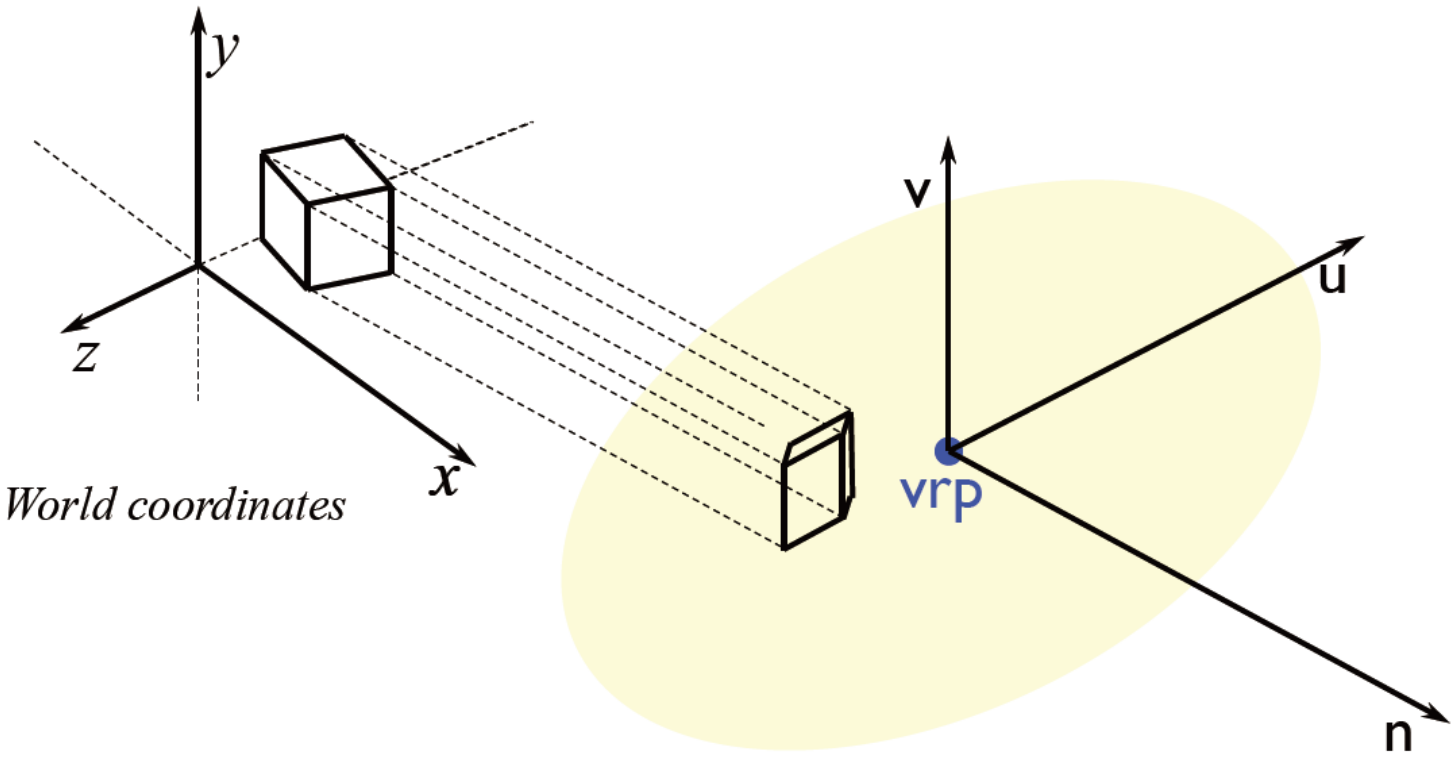


$$\vec{n} = -viewDirection$$

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

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

View Reference Coordinate System

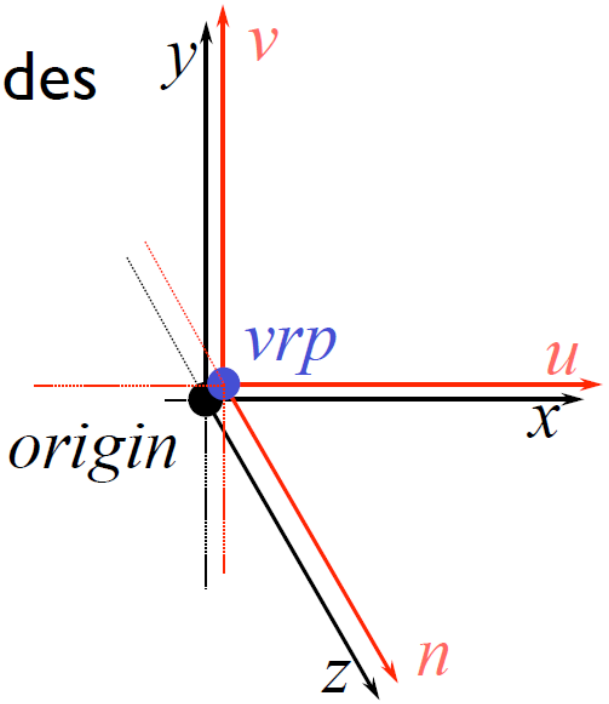


- Once the *view reference coordinate system* is defined, the next step is to project the *3D world* on to the *view reference plane*

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***

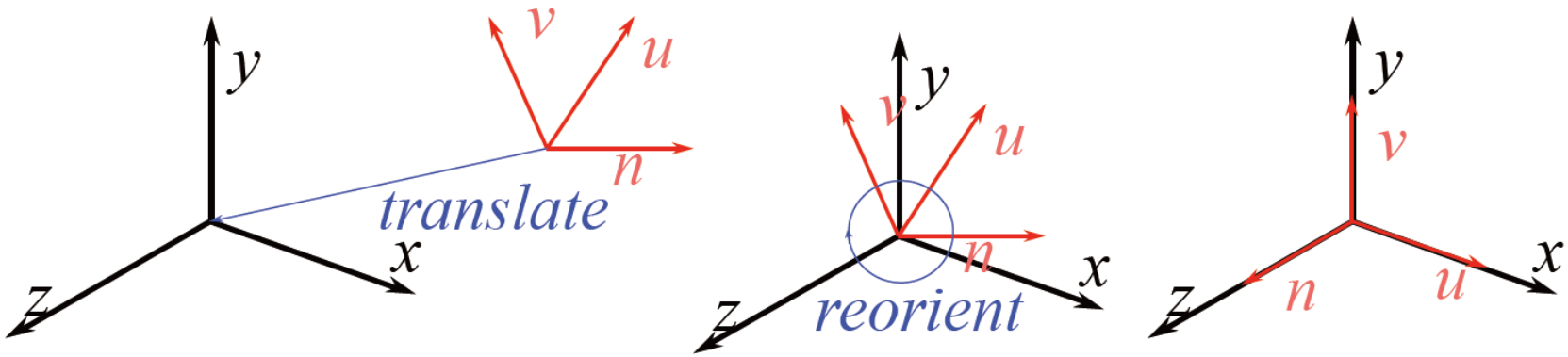


- Projection could be as simple as ignoring the z-coordinate

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*

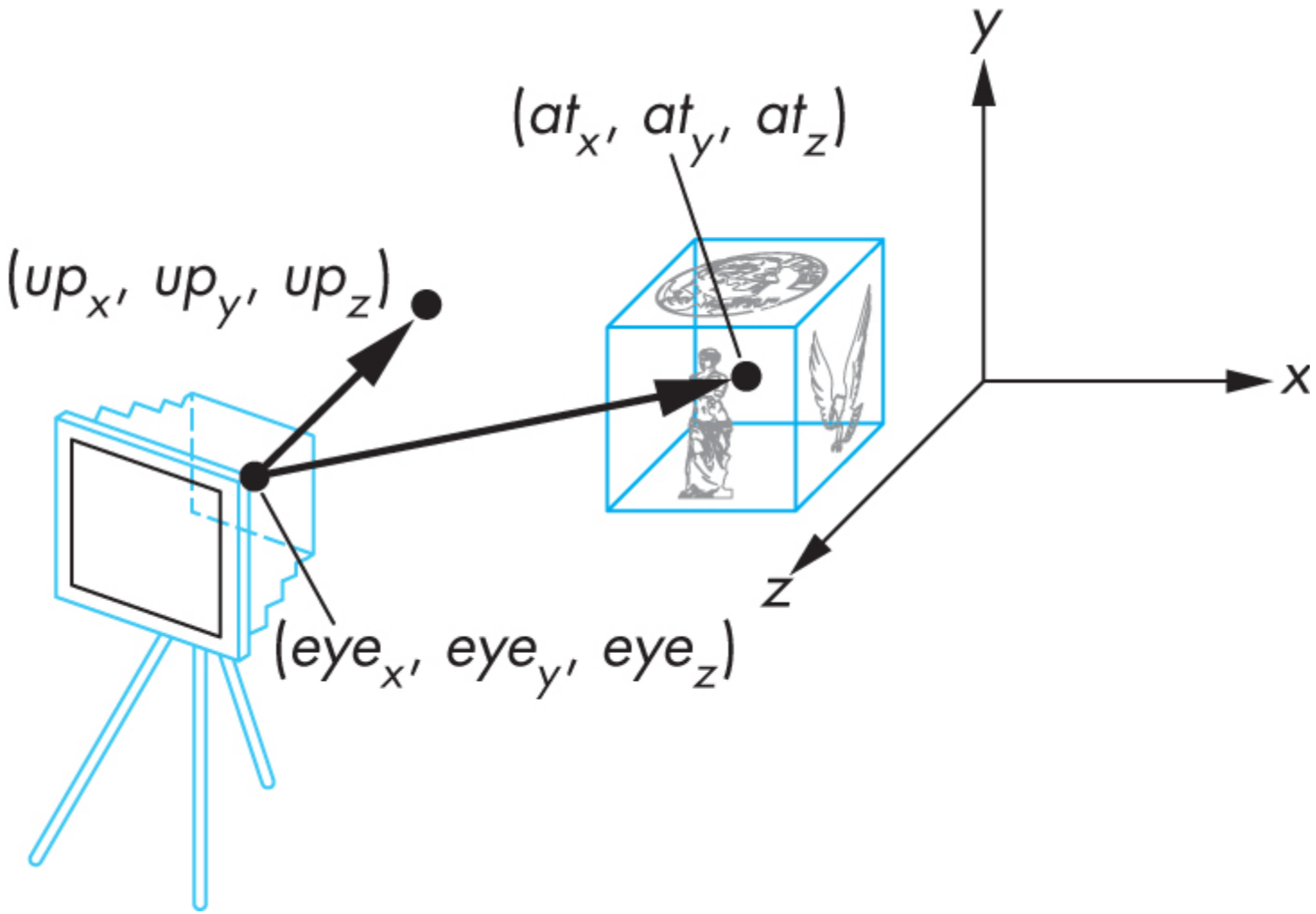
$$\begin{bmatrix} u_x & u_y & u_z & -\mathbf{r}_u \bullet vrp \\ v_x & v_y & v_z & -\mathbf{r}_v \bullet vrp \\ n_x & n_y & n_z & -\mathbf{r}_n \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A More Intuitive Approach Offered by GLM

- LookAt矩阵作为观察矩阵可以很高效地把所有世界坐标变换到刚刚定义的观察空间。LookAt矩阵就像它的名字表达的那样：它会创建一个看着(Look at)给定目标的观察矩阵。
- 幸运的是，GLM已经提供了这些支持。我们要做的只是定义一个摄像机位置，一个目标位置和一个表示世界空间中的上向量的向量（我们计算右向量使用的那个上向量）。接着GLM就会创建一个LookAt矩阵，我们可以把它当作我们的观察矩阵：

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), 摄像机位置
           glm::vec3(0.0f, 0.0f, 0.0f),  目标
           glm::vec3(0.0f, 1.0f, 0.0f)); 上向量
```

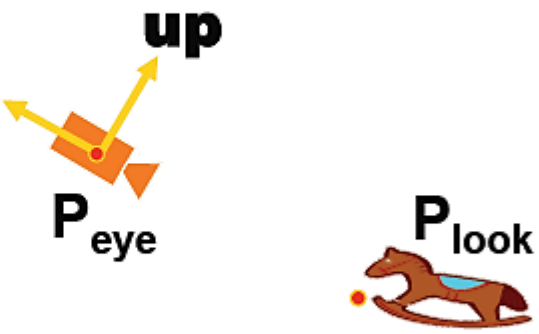
gluLookAt Illustration



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

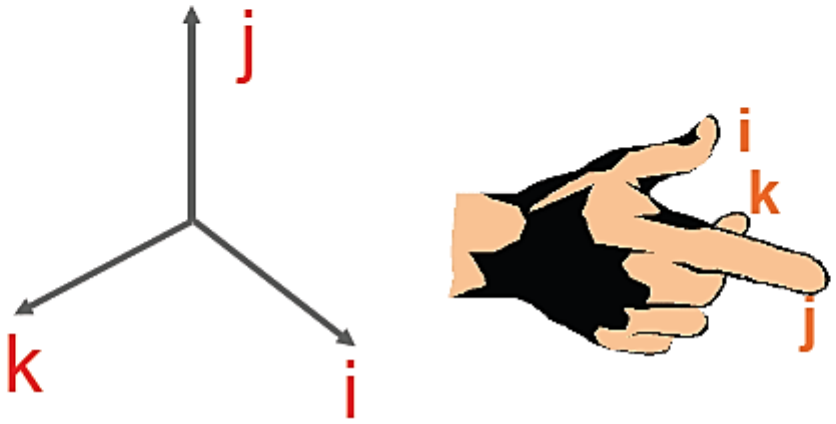
$$\mathbf{a} = P_{eye} - P_{look}$$



- 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

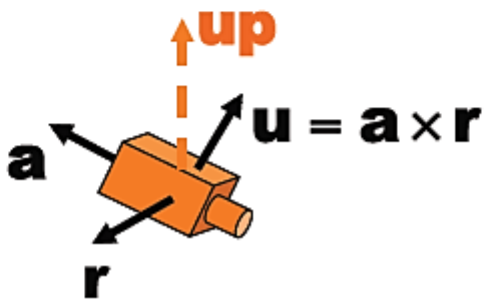


Constructing a Coordinates

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

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

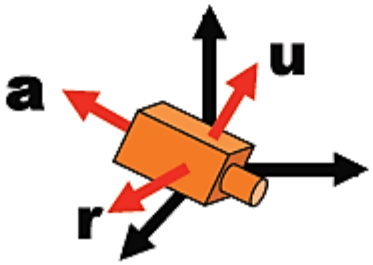
- Finally, using the vector **a** and the vector **r** we can synthesize a new vector **u** in the up direction:



Rotation

- Rotation takes the unit world frame to our desired **view reference frame:**

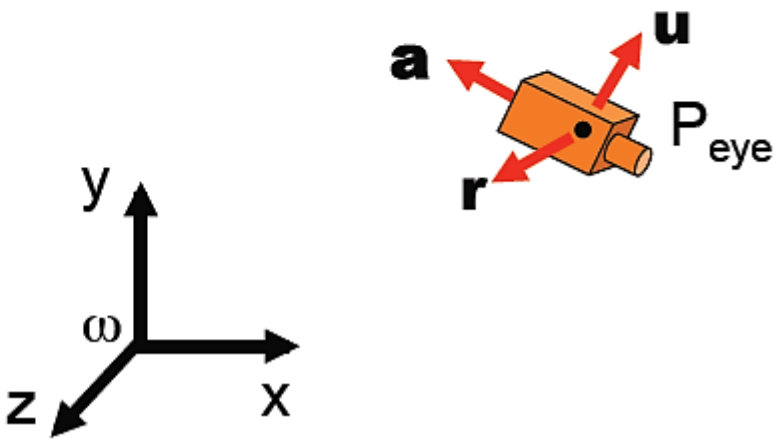
$$\begin{bmatrix} r_x & u_x & a_x & 0 \\ r_y & u_y & a_y & 0 \\ r_z & u_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{R}$$



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 **viewing coordinate 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(可编程管线)

主函数代码：

```
view = glm::lookAt(glm::vec3(camX, 0.0f, camZ), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
ourShader.setMat4("view", view);
```

顶点着色器代码：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
        gl_Position = projection * view * model * vec4(aPos, 1.0f);
        TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Model/View Transformation

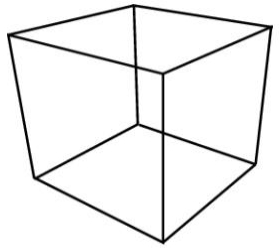
- Combine modeling and viewing transform
  - Combine into single matrix
  - Saves computation time
    - if many points are to be transformed
  - Possible because viewing transformation directly follows modeling transformation without intermediate operations

Outline

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

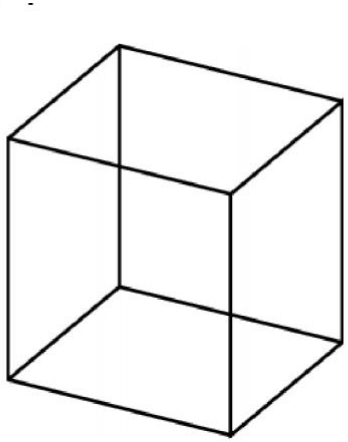
Perspective Projection

- Size varies inversely with respect to the distance form the center of projection
- Tends to look more realistic：Cannot generally measure
  - Shape
  - Object distances
  - Angles(except front faces)
  - Parallel lines appear no longer parallel



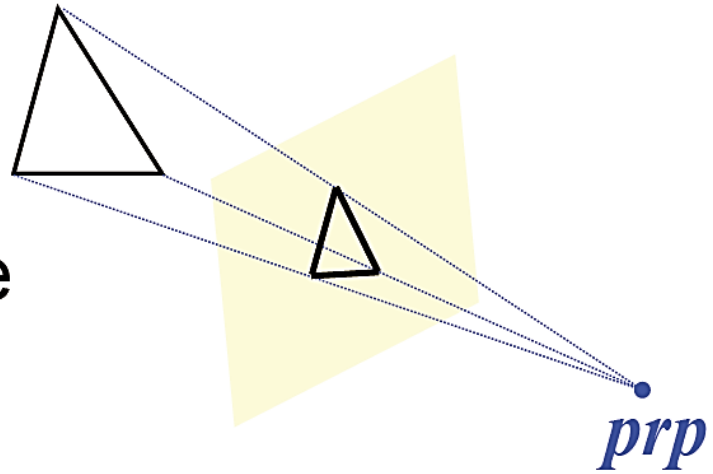
Perspective Projection

- Less realistic because perspective foreshortening is lacking
- Can however, use for exact measurements
  - Angles still only preserved for front faces
  - Parallel lines remain parallel



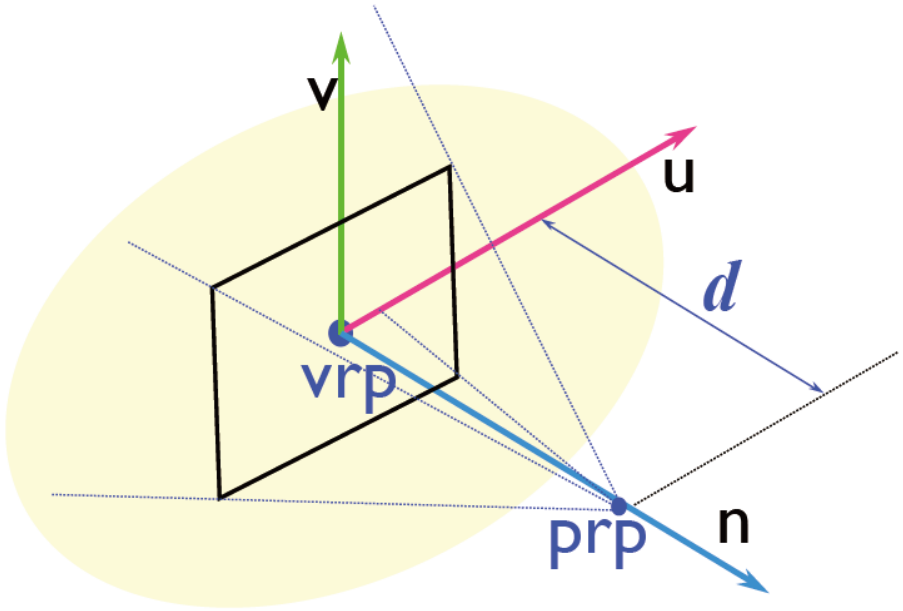
Perspective Projection

- 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

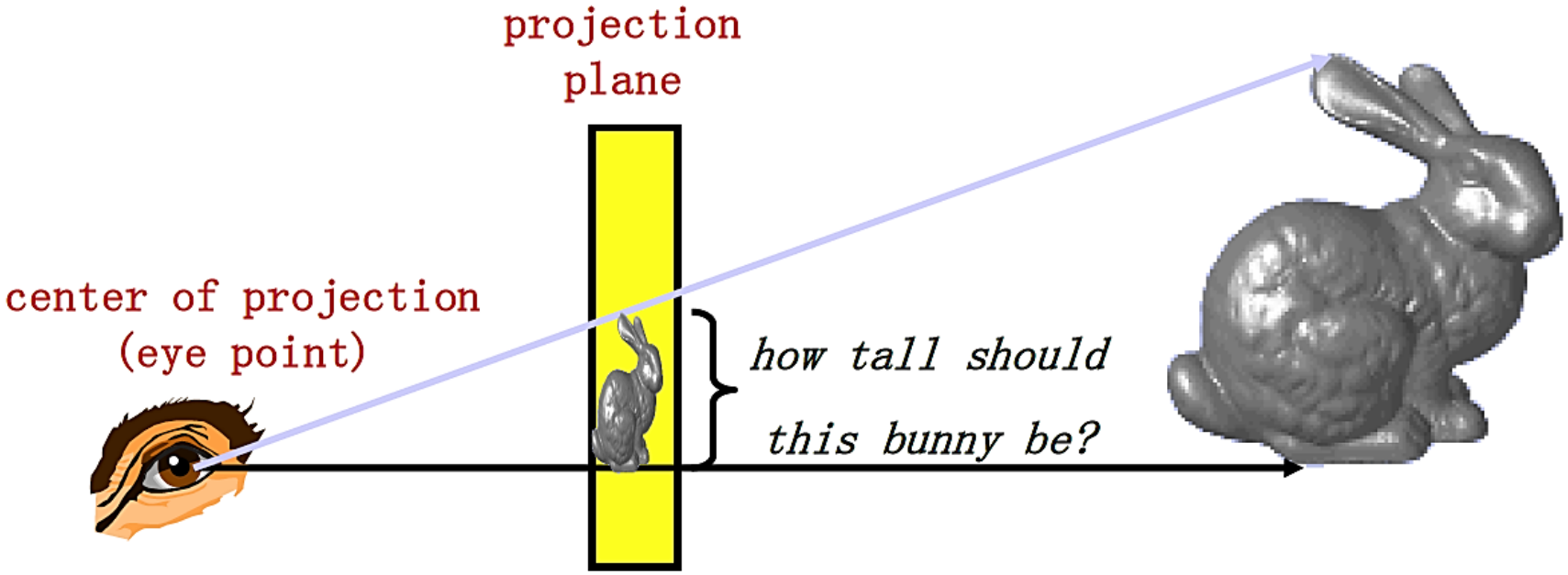


transformation matrix
for *perspective projection*

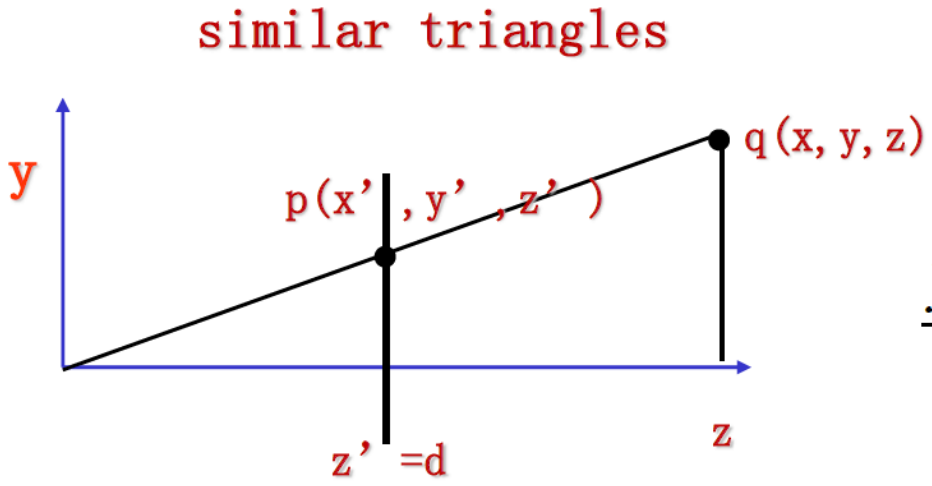
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}$$

<-应为0

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.

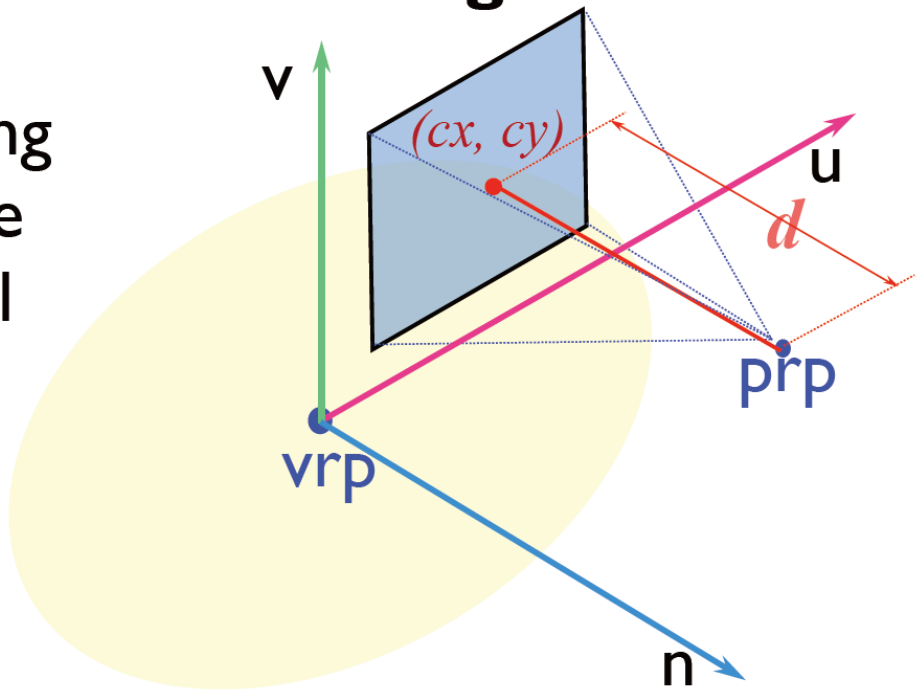
Homogeneous Coordinates

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

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

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 Projection

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

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

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

Orthogonal Projection Matrix: Homogeneous coordinates

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

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

$$x_p = x$$

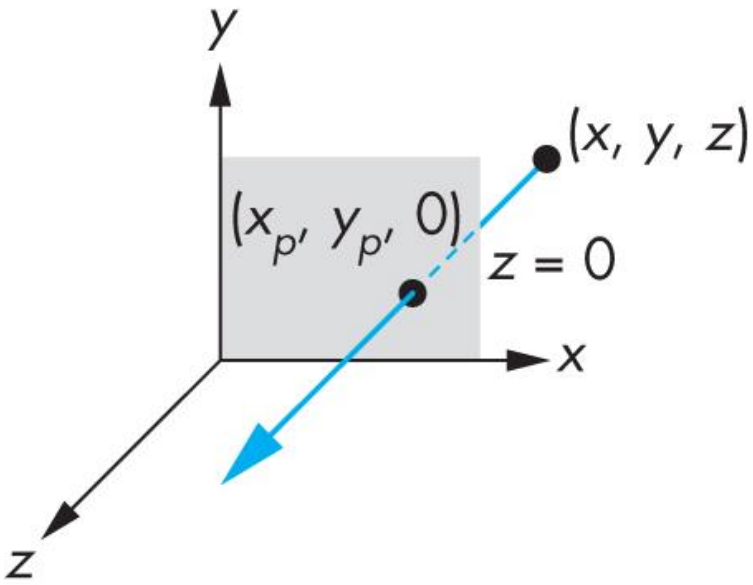
$$y_p = y$$

$$z_p = 0$$

$$w_p = 1$$

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。

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



Orthogonal Projection

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

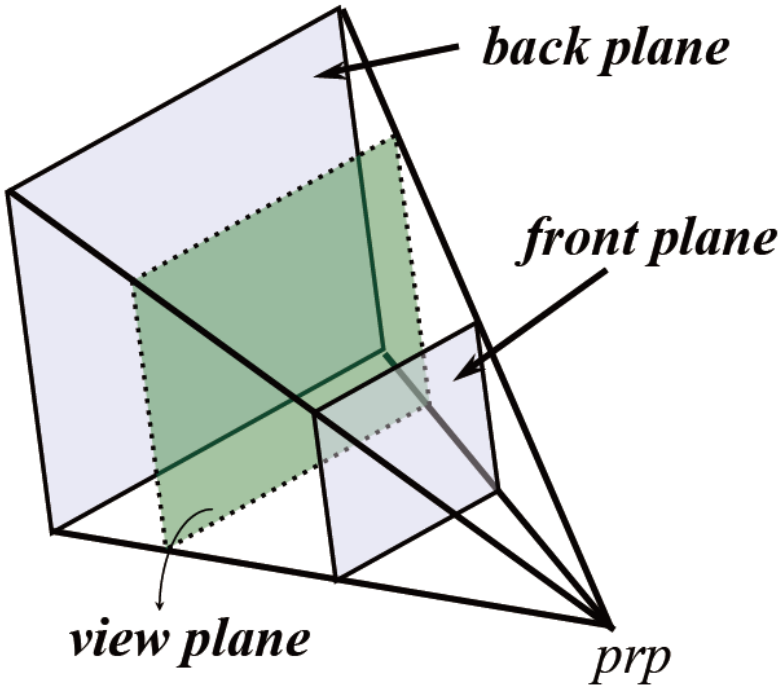
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_x & u_y & u_z & -\vec{u} \bullet vrp \\ v_x & v_y & v_z & -\vec{v} \bullet vrp \\ n_x & n_y & n_z & -\vec{n} \bullet vrp \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

View Volume & Clipping

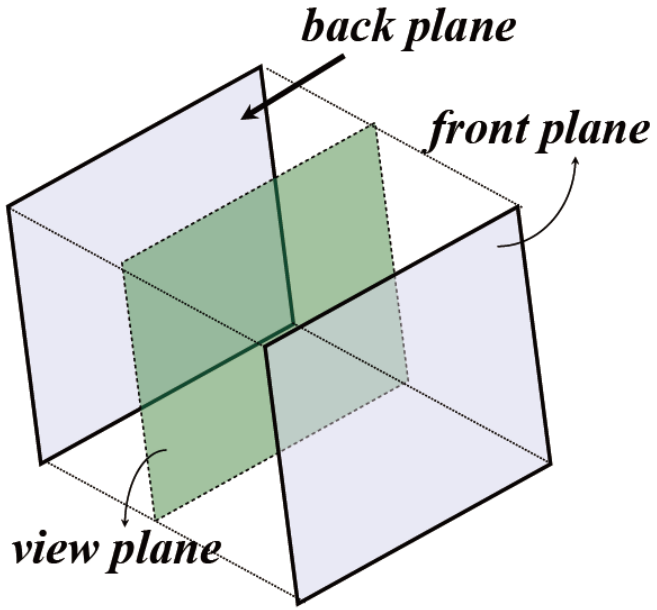
- For perspective projection the ***view volume*** is a semi-infinite pyramid with apex (顶点) at ***prp*** and edges passing through the corners of the view window

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



View Volume & Clipping

- For parallel projection the ***view volume*** is an infinite parallelepiped (平行六面体) with sides parallel to the direction of projection

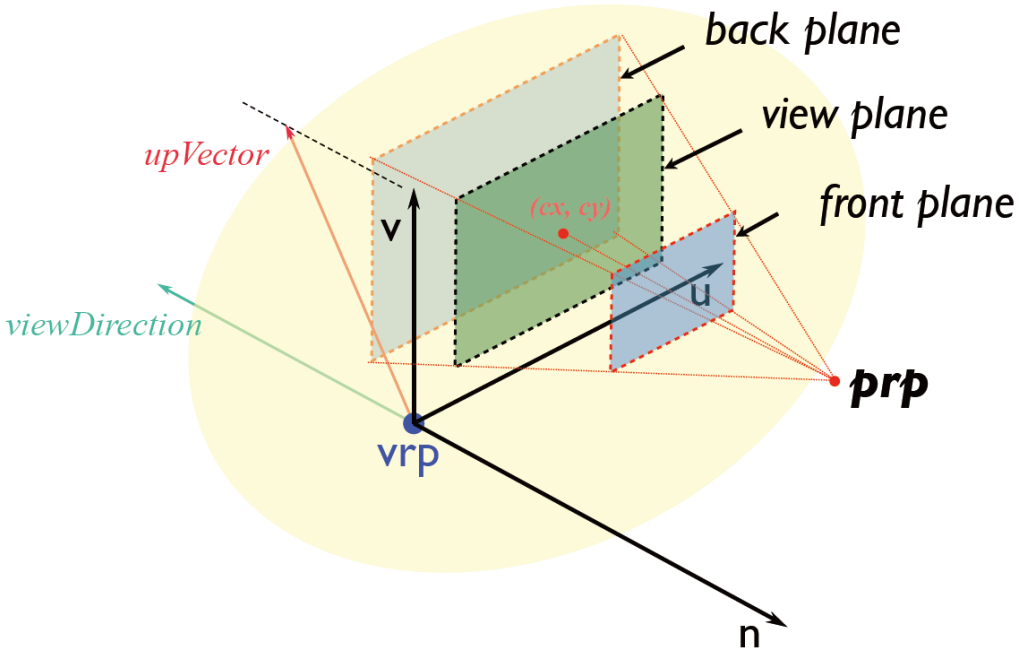


- View volume is made finite by specifying the front and back clipping plane specified as distance from the view plane

- 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

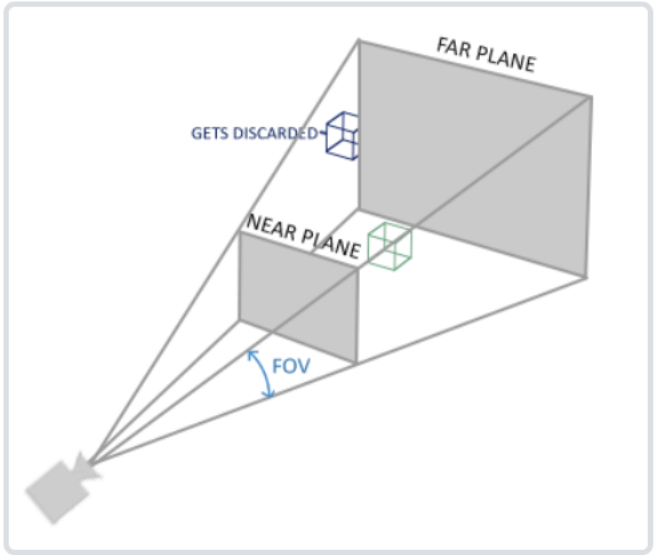


Perspective in OpenGL(可编程管线)

在GLM中可以这样创建一个透视投影矩阵：

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

同样，`glm::perspective`所做的其实就是创建了一个定义了可视空间的大**平截头体**，任何在这个平截头体以外的东西最后都不会出现在裁剪空间体积内，并且将会受到裁剪。一个透视平截头体可以被看作一个不均匀形状的箱子，在这个箱子内部的每个坐标都会被映射到裁剪空间上的一个点。下面是一张透视平截头体的图片：

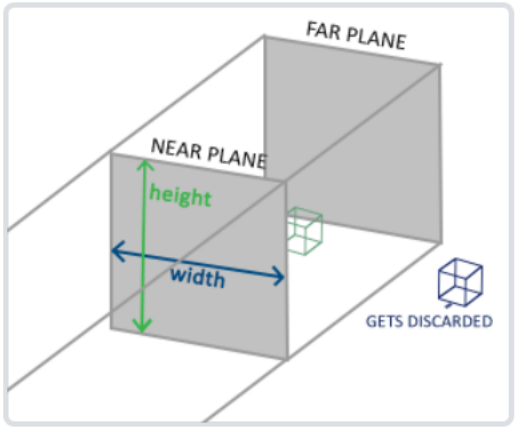


它的第一个参数定义了fov的值，它表示的是视野(Field of View)，并且设置了观察空间的大小。如果想要一个真实的观察效果，它的值通常设置为45.0f，但想要一个末日风格的结果你可以将其设置一个更大的值。第二个参数设置了宽高比，由视口的宽除以高所得。第三和第四个参数设置了平截头体的**近**和**远**平面。我们通常设置近距离为0.1f，而远距离设为100.0f。所有在近平面和远平面内且处于平截头体内的顶点都会被渲染。

Normalization

- Normalization allows for a single pipeline for both perspective and orthogonal viewing.
- It simplifies clipping.
- Projection to the image plane is simple (discard z).
- z is retained for z-buffering (visible surface determination)

Orthogonal view in OpenGL(可编程管线)



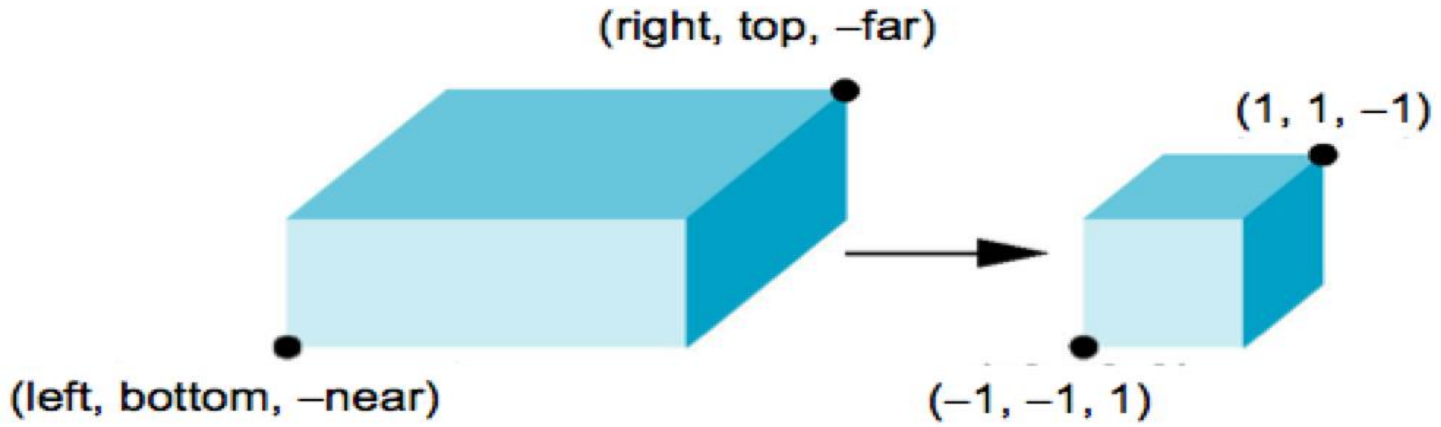
上面的平截头体定义了可见的坐标，它由由宽、高、近(Near)平面和远(Far)平面所指定。任何出现在近平面之前或远平面之后的坐标都会被裁剪掉。正射平截头体直接将平截头体内部的所有坐标映射为标准化设备坐标，因为每个向量的w分量都没有进行改变；如果w分量等于1.0，透视除法则不会改变这个坐标。

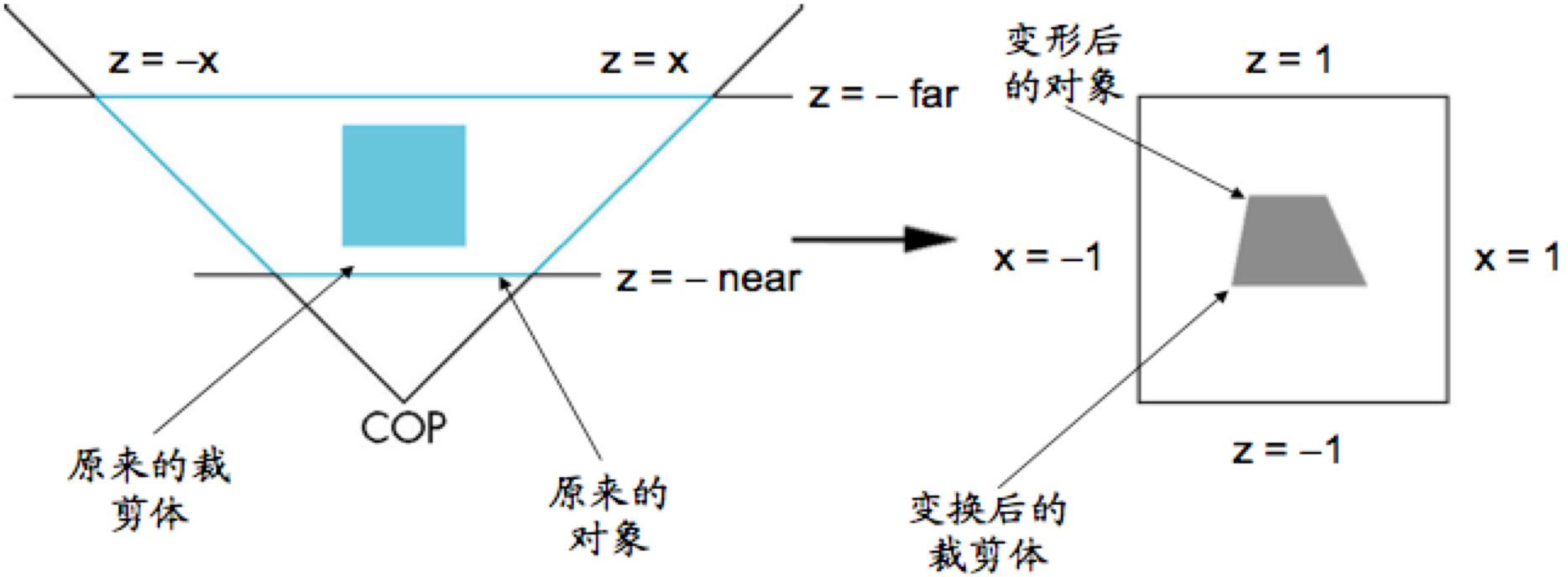
要创建一个正射投影矩阵，我们可以使用GLM的内置函数`glm::ortho`：

```
glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

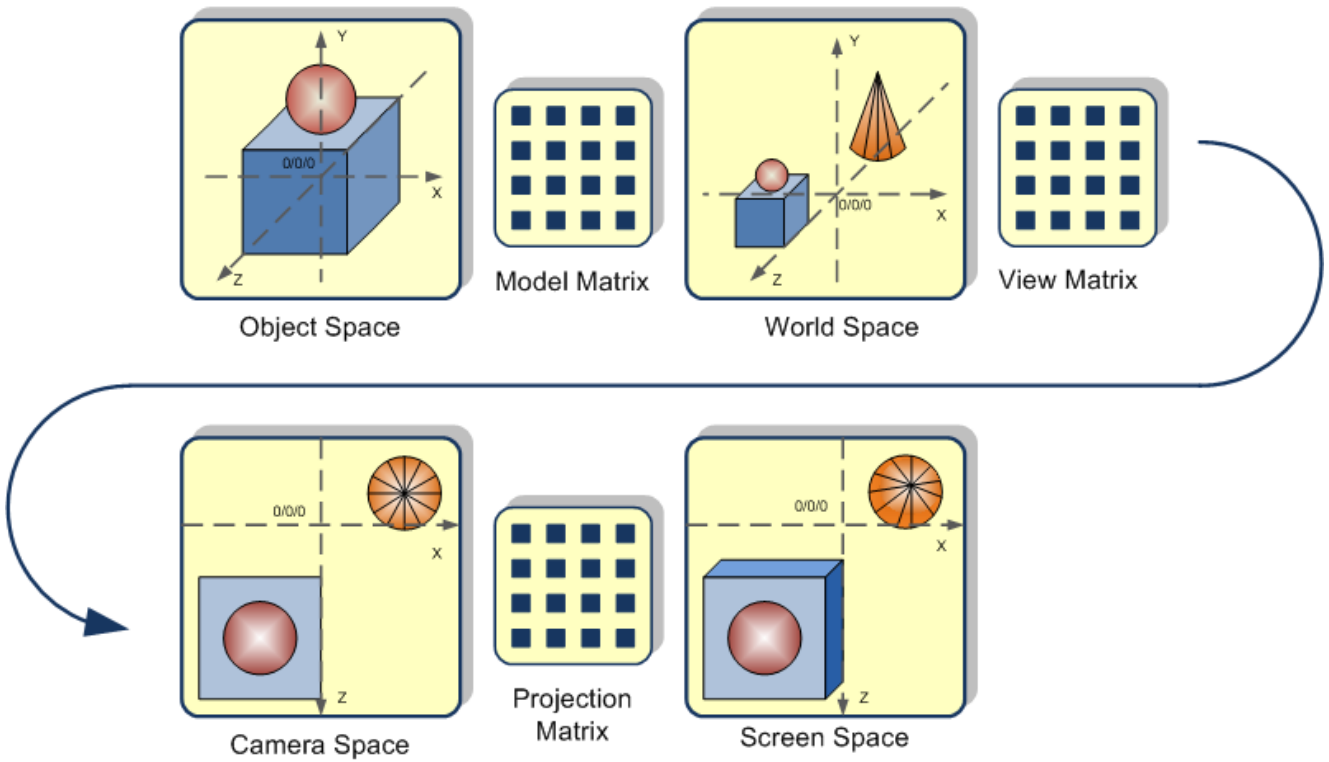
前两个参数指定了平截头体的左右坐标，第三和第四参数指定了平截头体的底部和顶部。通过这四个参数我们定义了近平面和远平面的大小，然后第五和第六个参数则定义了近平面和远平面的距离。这个投影矩阵会将处于这些x，y，z值范围内的坐标变换为标准化设备坐标。

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





Transformation Pipeline



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.