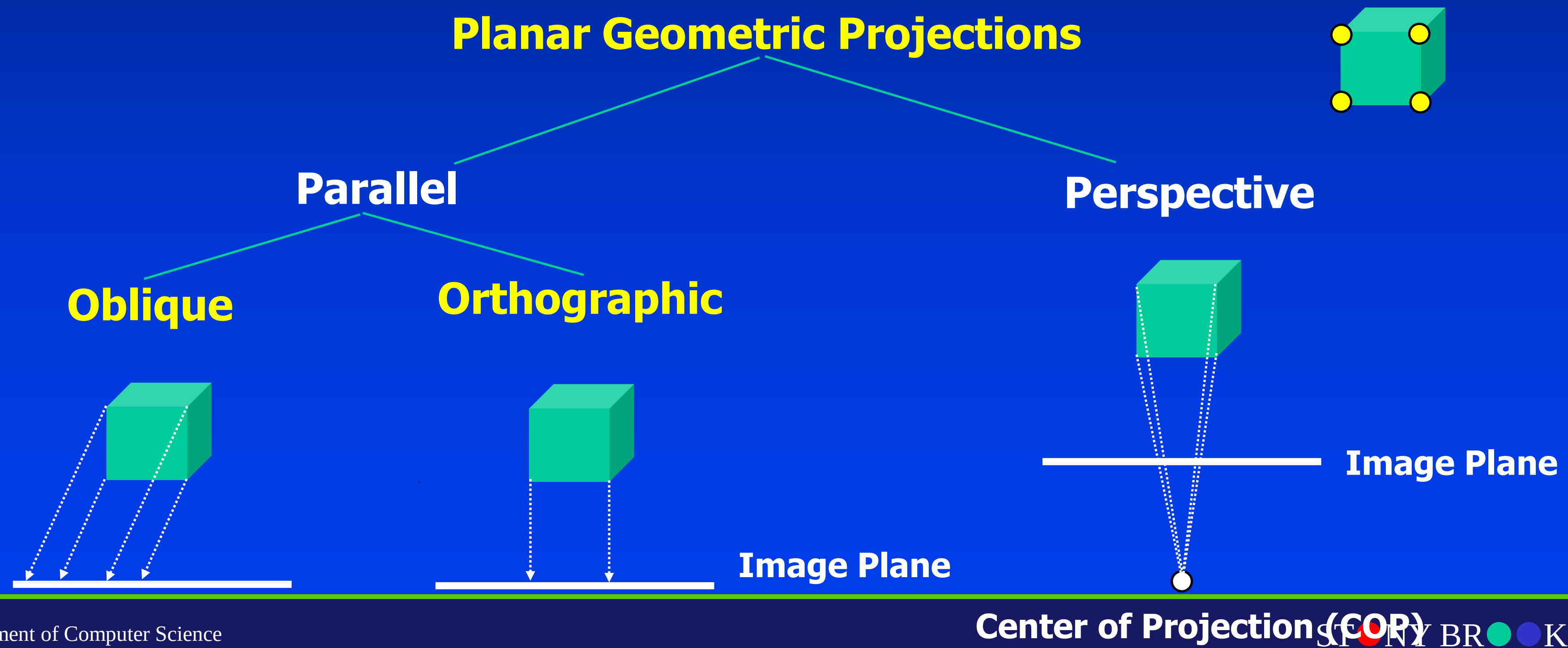
CSE328 Fundamentals of Computer Graphics

Hong Qin
Department of Computer Science
Stony Brook University (SUNY at Stony Brook)
Stony Brook, New York 11794-2424
Tel: (631)632-8450; Fax: (631)632-8334
qin@cs.stonybrook.edu
http://www.cs.stonybrook.edu/~qin

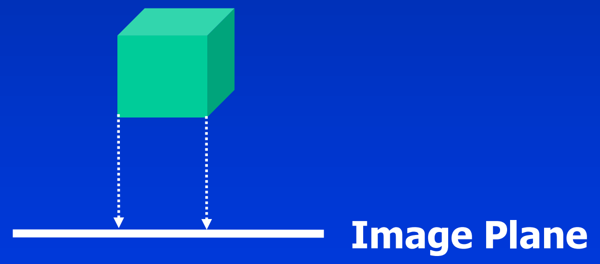
Geometric Projections

- From 3D to 2D
- Transform points from camera coordinate system to the screen (image plane of the virtual camera).



Parallel Orthographic Projection

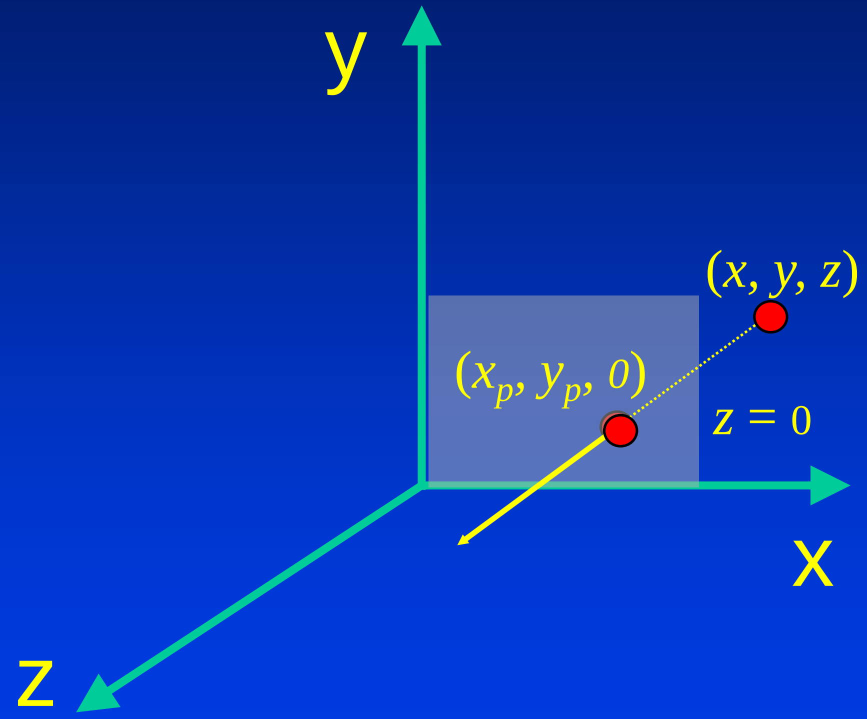
- Preserves X and Y coordinates.
- Preserves both distances and angles.



Parallel Orthographic Projection

- $x_p = x$
- $y_p = y$
- $z_p = 0$

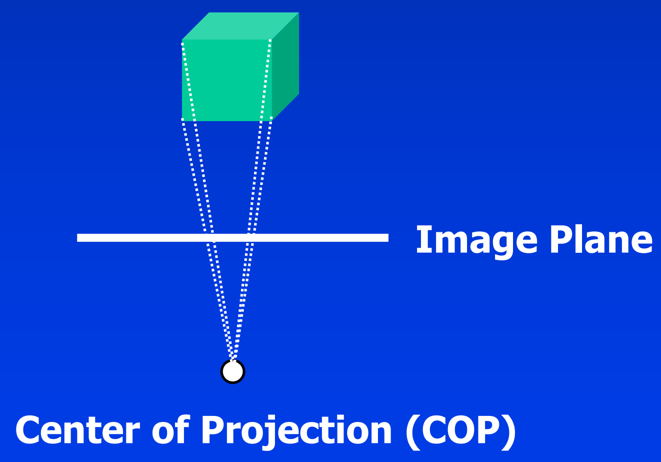
$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



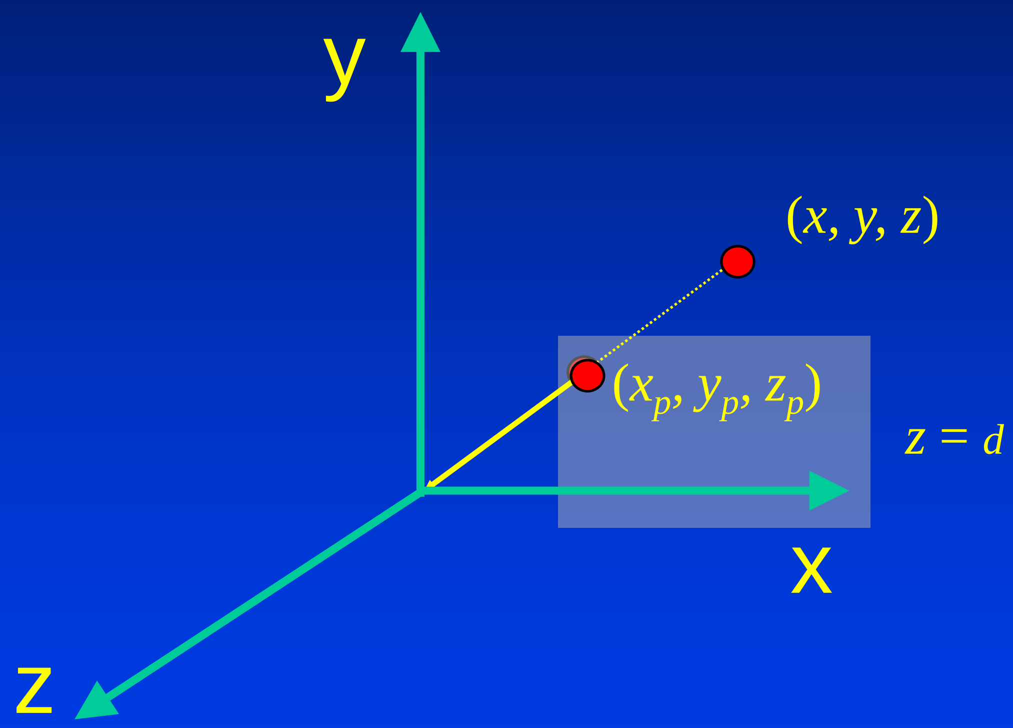
Department of Computer Science
Center for Visual Computing

Perspective Projection

- Only preserves parallel lines that are parallel to the image plane.
- Line segments are shorten by distance.

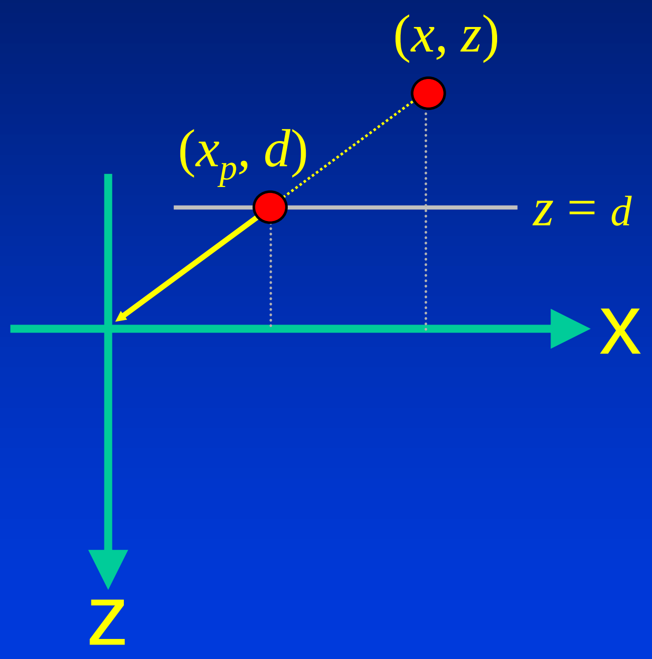


Perspective Projection



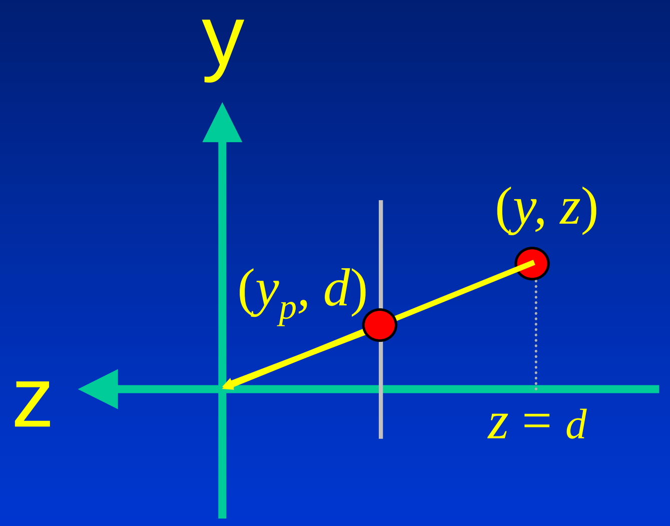
Perspective Projection

- $z_p = d$
- $x_p = (x \cdot d) / z$



Perspective Projection

- $z_p = d$
- $y_p = (y \cdot d) / z$



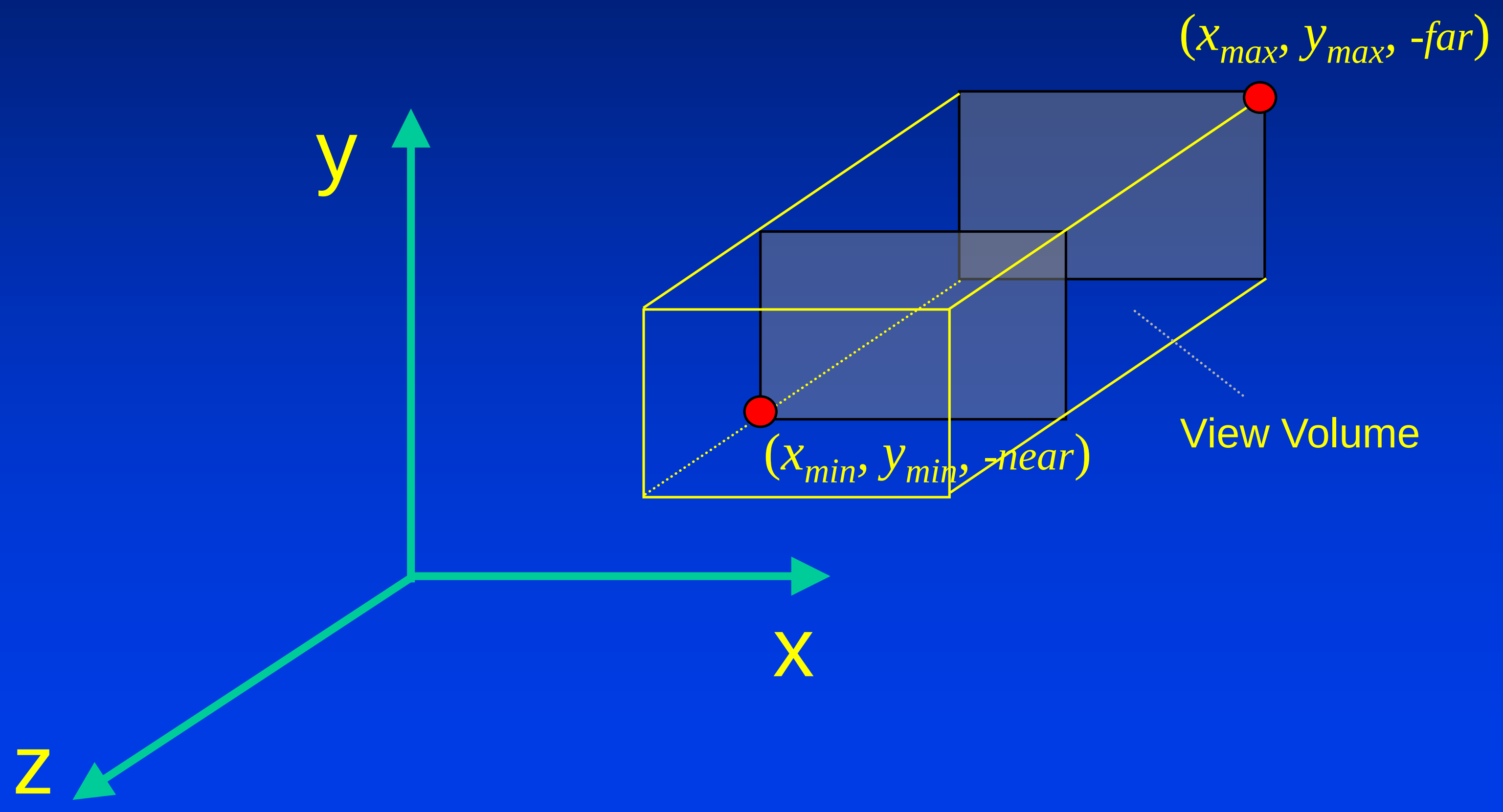
Perspective Projection

- $x_p = (x \cdot d) / z = x/(z/d)$
- $y_p = (y \cdot d) / z = y/(z/d)$
- $z_p = d \quad = z/(z/d)$

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1/h & 0 & 0 & 0 \\ 0 & 1/h & 0 & 0 \\ 0 & 0 & 1/h & 0 \\ 0 & 0 & 0 & 1/h \end{bmatrix} \begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix}$$

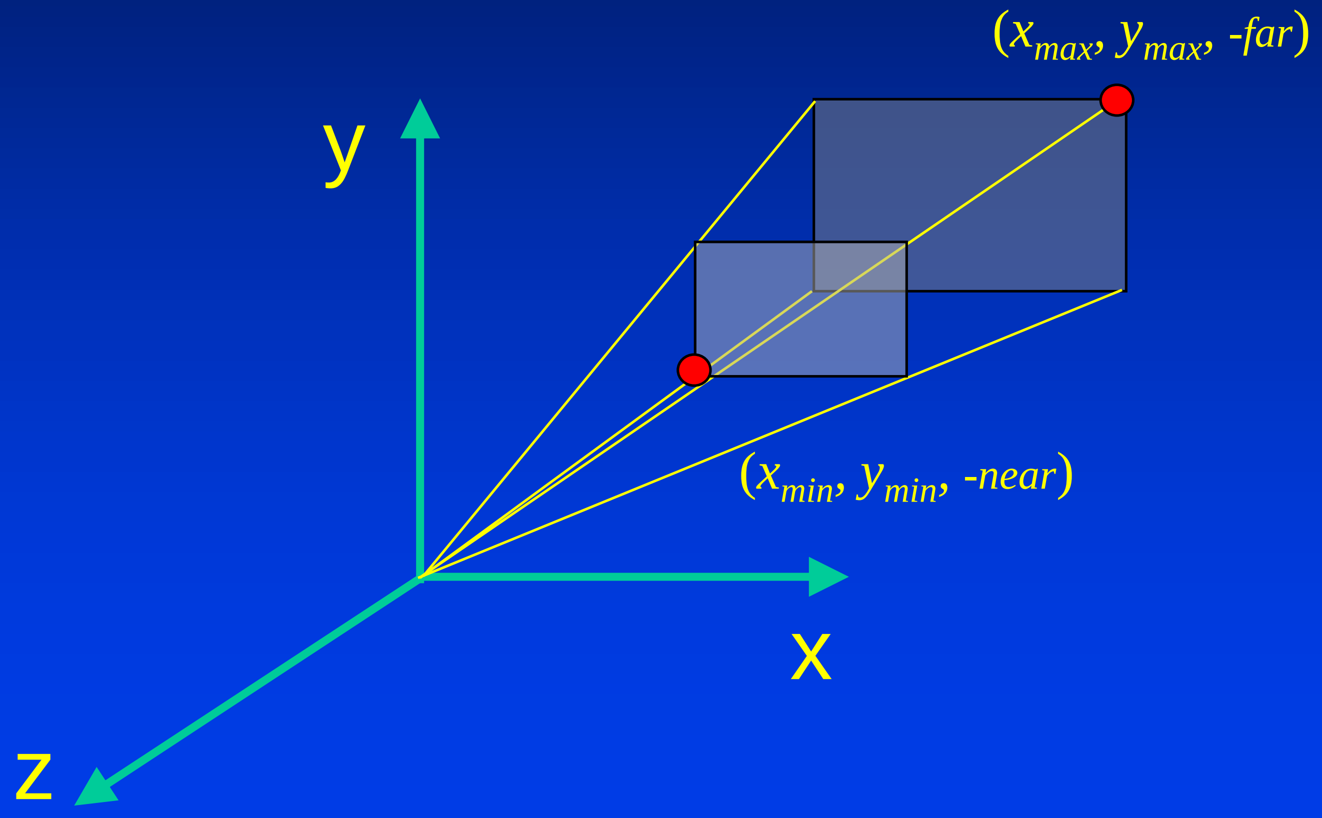
Defining the Parallel View Volume

glOrtho(xmin, xmax, ymin, ymax, near, far)



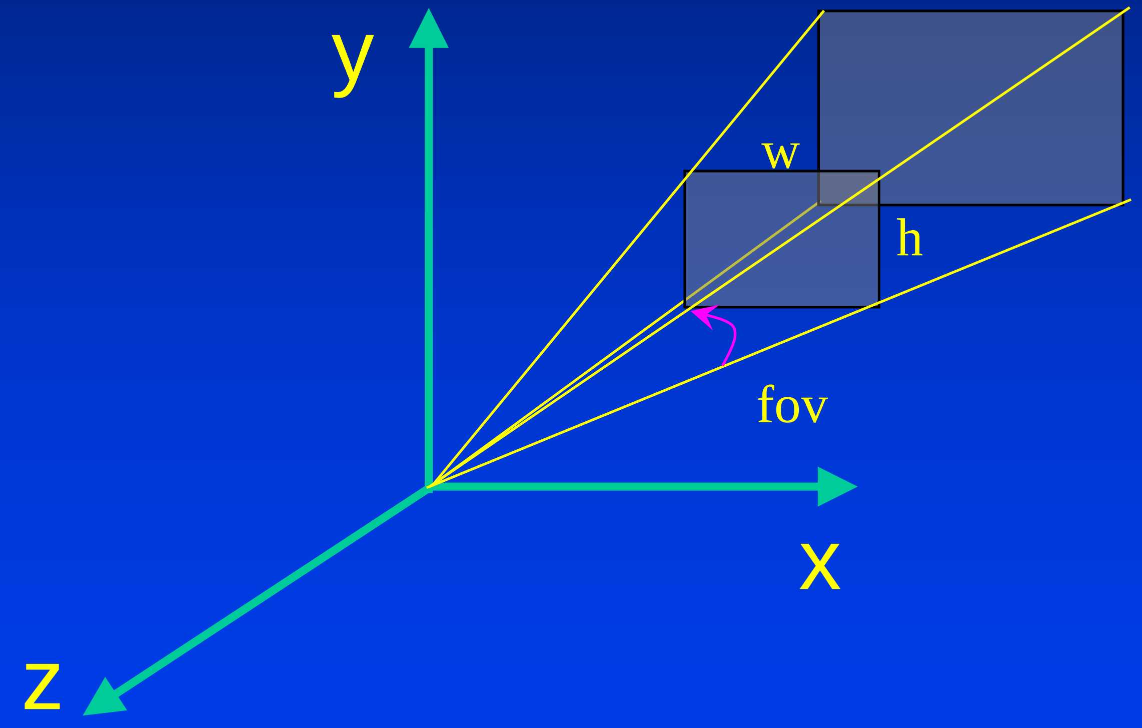
Defining the Perspective View Volume

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

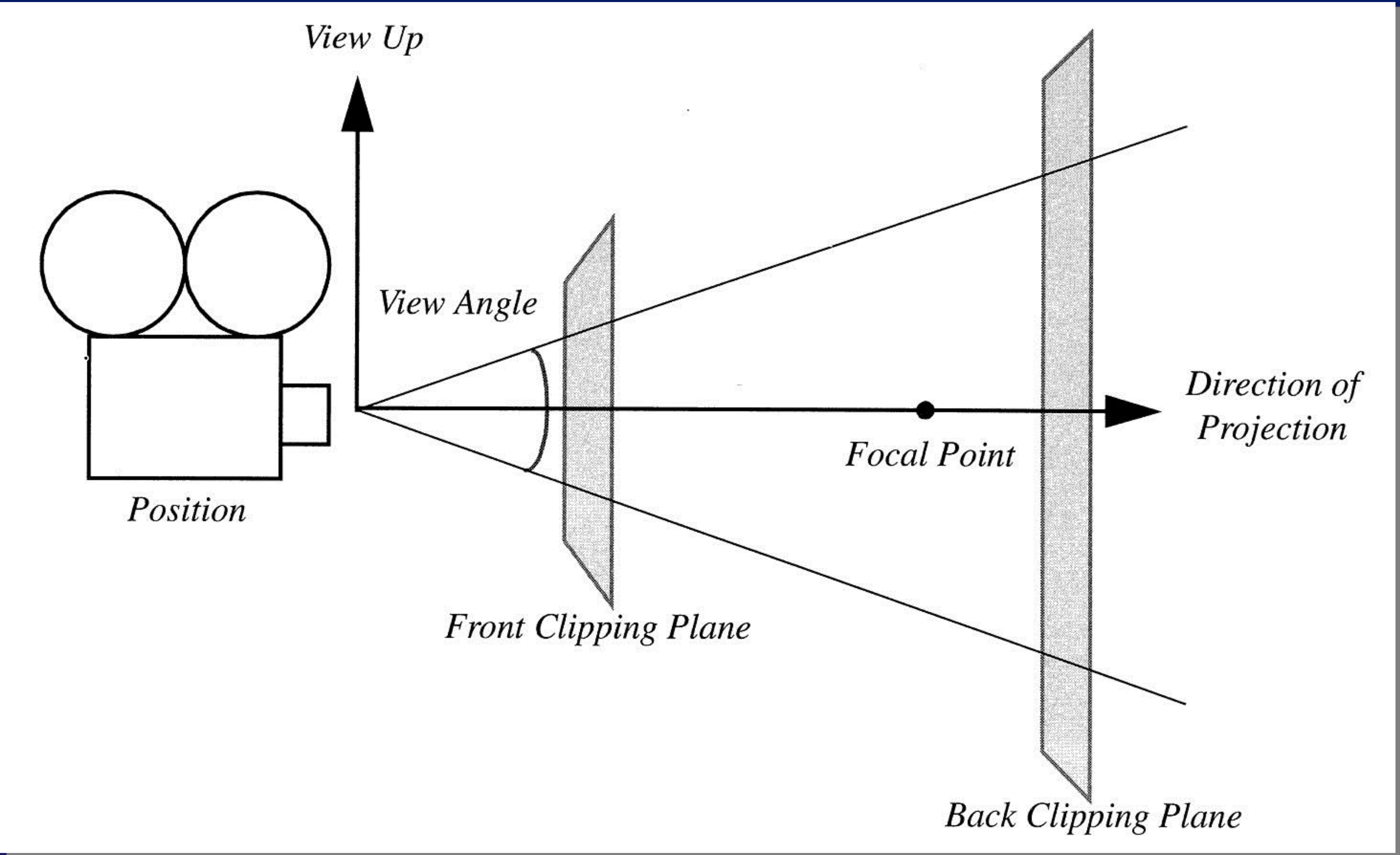


Defining the Perspective View Volume

gluPerspective(fovy, aspect, near, far)



Basic Camera Attributes



3D Graphics Viewing Pipeline

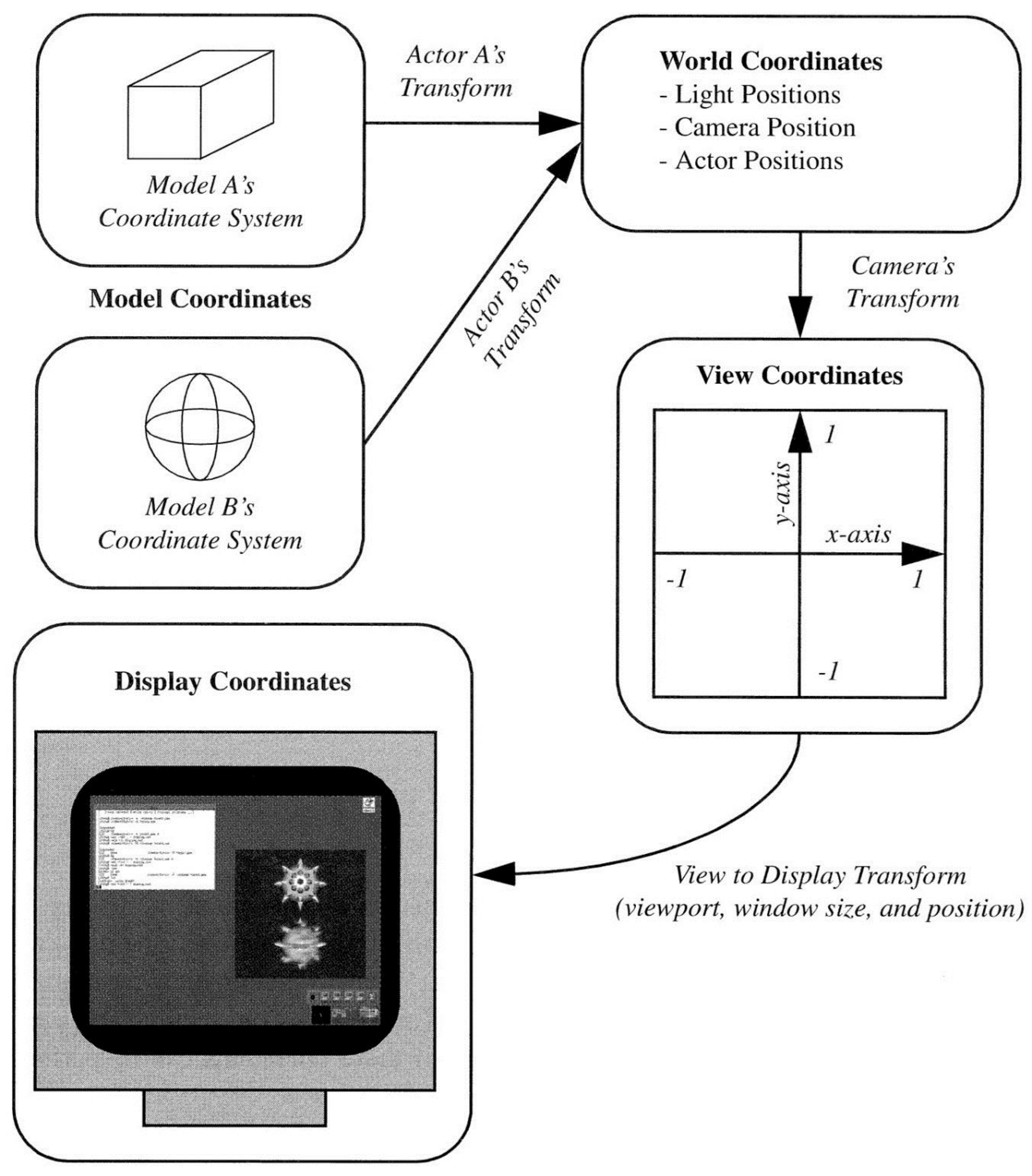
3D Viewing (Revisit the Pipeline)

- We will need to revisit the concept and the techniques for defining 3D viewing coordinate system and specifying 3D view volume (view frustum) for graphics pipeline
- We will need to convert 3D view volume (both parallel projection and perspective projection) to a canonical, normalized, device-independent coordinate system, before we can display the final picture in the specified viewport on the display device!

Coordinate Systems (Computer Graphics Pipeline)

1. Objects in model coordinates are transformed into
2. World coordinates, which are transformed into
3. View coordinates, which are transformed into
4. Normalized device coordinates, which are transformed into
5. Display coordinates, which correspond to pixel positions on the screen
6. Transformations from one coordinate system to another take place via *coordinate transformations*, which we have already discussed

Coordinate Systems



Specify a View Volume

- Reduce degrees of freedom to make the operations easier; four steps to specify a view volume

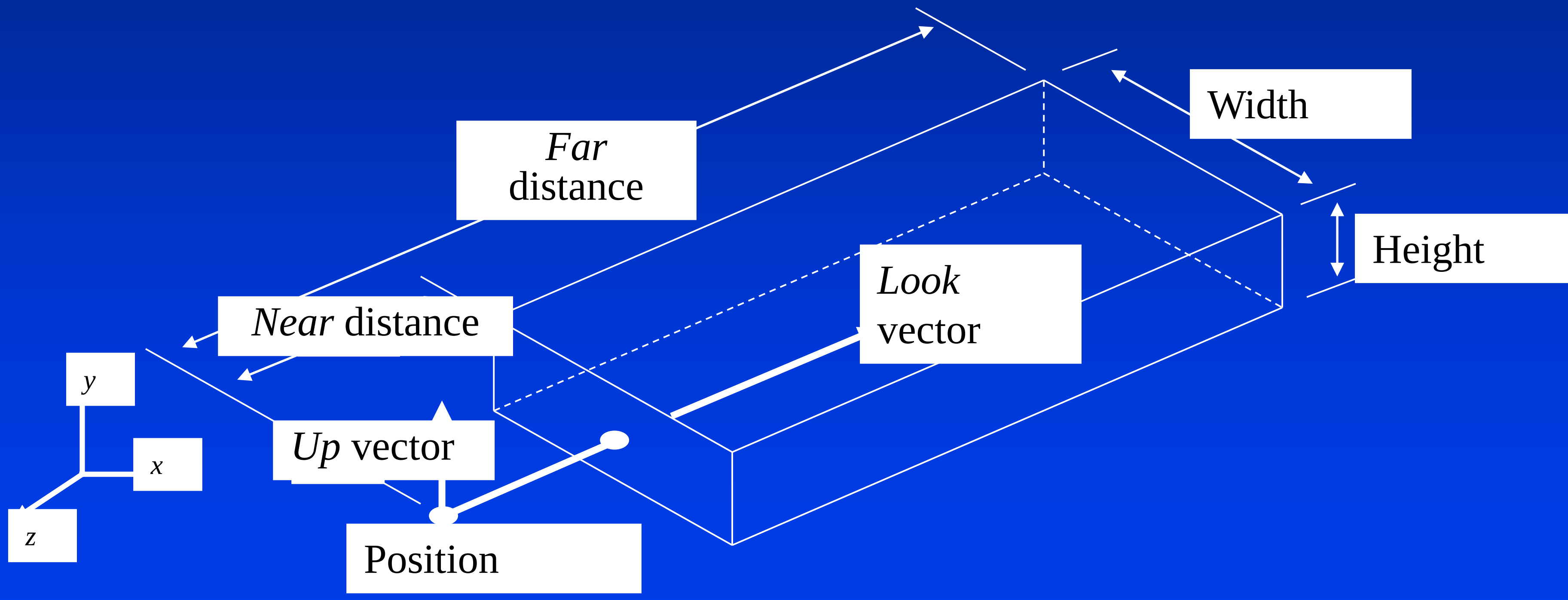
1. Position the camera (and therefore its view/image plane), the center of projection
2. Orient the camera to point at what you want to see, the view direction and the view-up direction
3. Define field of view:

   perspective: aspect ratio of image and angle of view: between wide angle, normal, and zoom

   parallel: width and height
4. Choose perspective or parallel projection

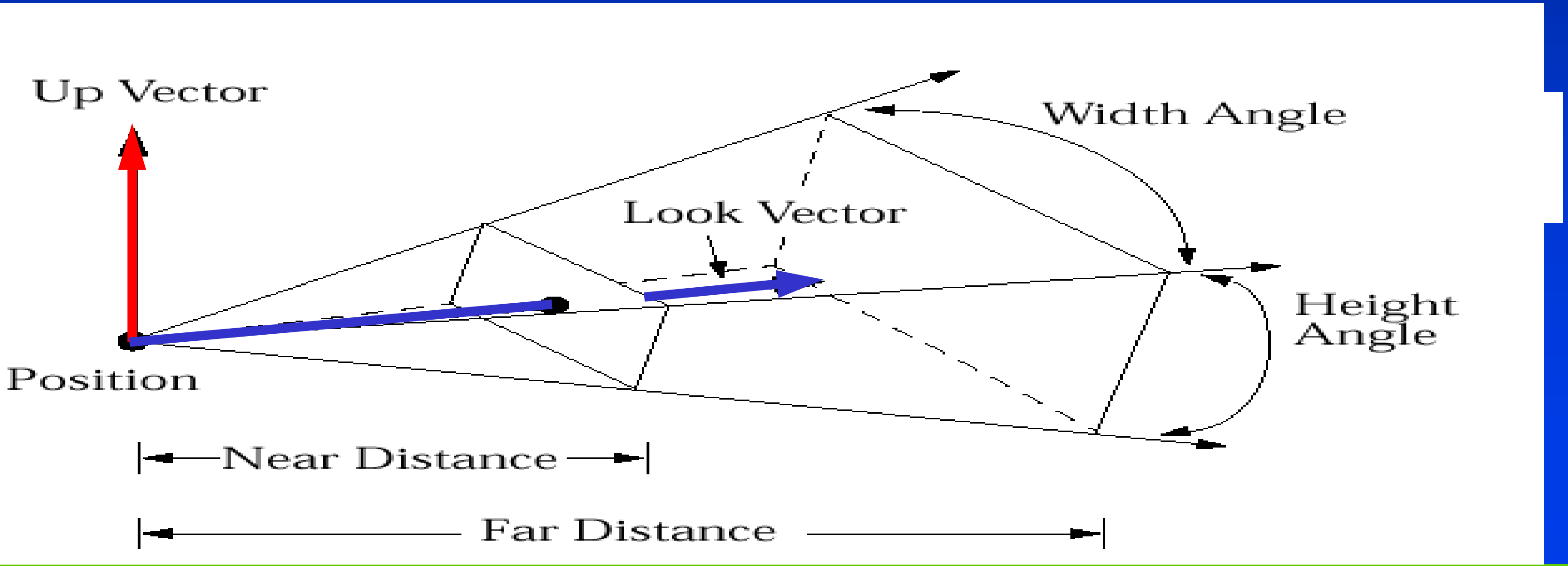
View Volume (Parallel Projection)

- For example, orthographic parallel projection: truncated view volume – Cuboid (not exactly a cube!)
- How about oblique projection???



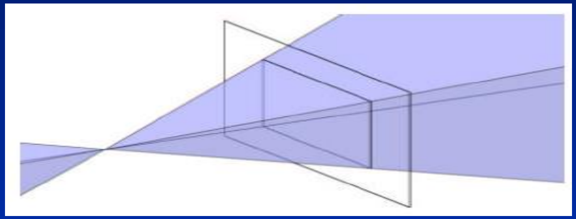
View Volume (Perspective Projection)

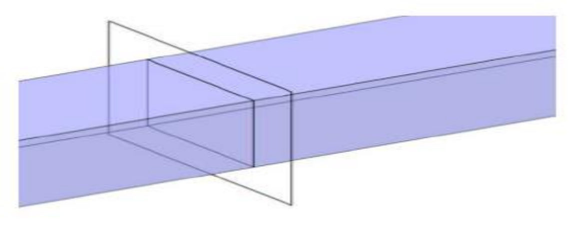
- Perspective projection: Truncated pyramid – View frustum
- How about oblique projection???



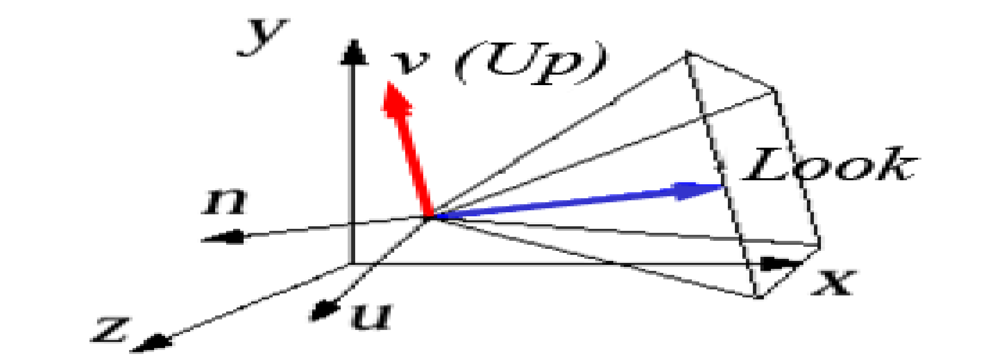
Specifying Arbitrary 3D Views

- Definition of view volume (the visible part of the virtual world) specified by camera's position and orientation
  - *Position* (a point)
  - *Look* and *Up vectors*
- **Shape** of view volume specified by
  - *Horizontal* and *vertical view angles*
  - *Front and back clipping planes*





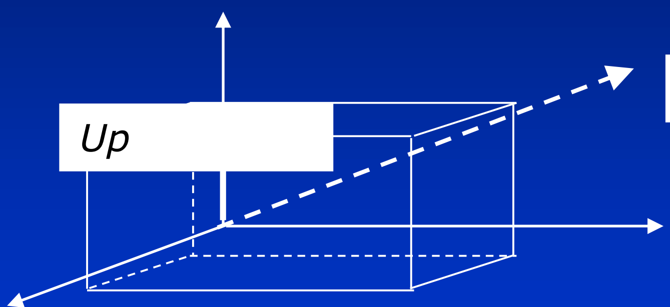
- Coordinate Systems
  - **World coordinates** – standard right-handed *xyz* 3-space
  - **Camera coordinates** – camera-space right-handed coordinate system (*u, v, n*); origin at *Position* and axes rotated by orientation; used for transforming arbitrary view into canonical view

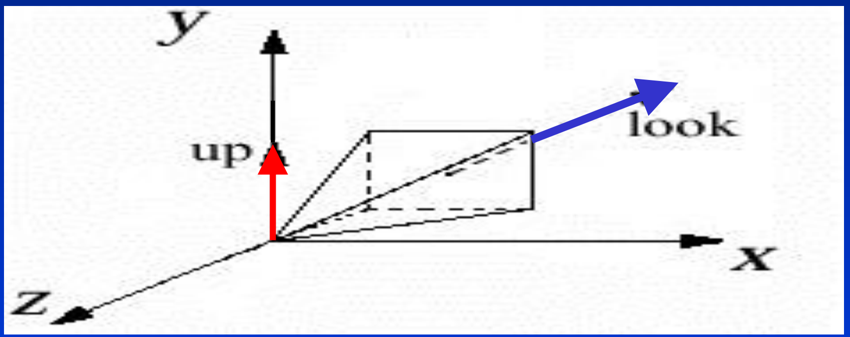


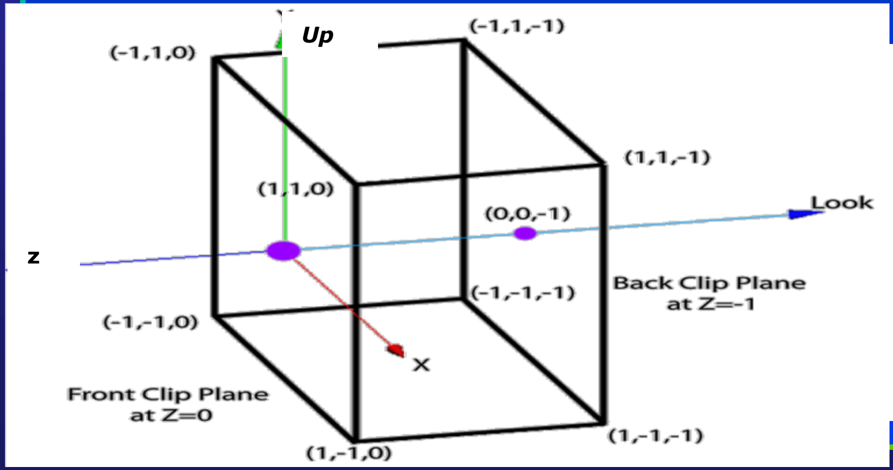
Arbitrary Perspective Frustum

Canonical View Volume

- This is the key for today's lecture







- parallel projection
- sits at origin:
  Position = (0, 0, 0)
- looks along negative *z*-axis:
  Look vector = (0, 0, −1)
- oriented upright:
  Up vector = (0, 1, 0)
- film plane extending from −1 to 1 in *x* and *y*

Normalizing to the Device Independent View Volume

- Goal: transform arbitrary view coordinate system to the canonical view volume (device independent), maintaining relationship between view volume and the normalized, device independent coordinate system, then take picture
  - For parallel view volume, transformation is ***affine*** : consisting of linear transformations (rotations and scales) and translation/shift
  - In case of a perspective view volume, it also contains a non-affine perspective transformation that turns a frustum into a parallel view volume, a cuboid
  - Composite transformation to transform arbitrary view volume to the canonical view volume, named the **normalizing transformation**, is still a 4x4 homogeneous matrix that typically has an inverse
  - Easy to clip against this canonical view volume; clipping planes are axis-aligned!
  - Projection using canonical view volume is even easier: just omit *z*-coordinates
  - For oblique parallel projection, a shearing transform is part of composite transform, to "de-oblique" view volume FIRST!!!

Viewing Coordinate System

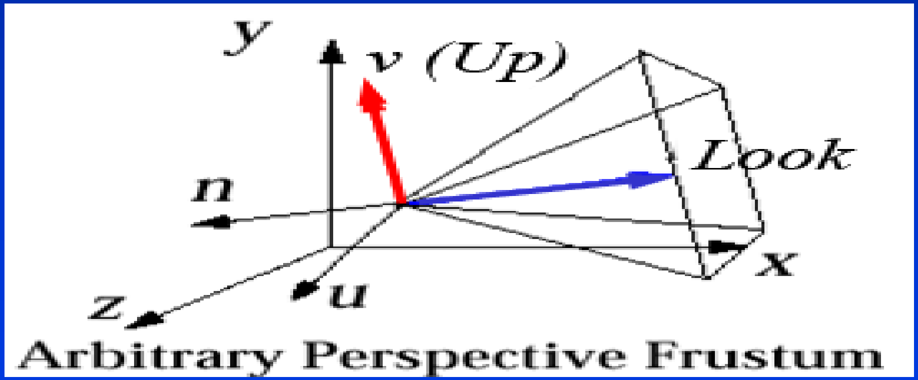
- We have specified arbitrary view with viewing parameters
- Problem: map arbitrary view specification to 2D image of scene. This is hard, both for clipping and for projection
- Solution: reduce to a simpler problem and solve it step-by-step

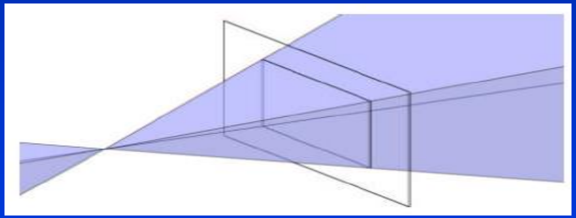
- Note: *Look vector* along negative (not positive) z-axis is arbitrary but makes math easier!

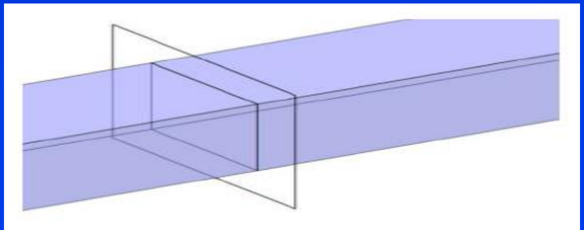
STONY BROOK
STATE UNIVERSITY OF NEW YORK

Specify Arbitrary 3D Viewing Coordinate System

- The original of coordinate system
- Three independent directions (mutually perpendicular with each other)







Viewing in Three Dimension

- The key: Mathematics of projections and its matrix operations
- How to produce 2D image from view specification?
- It is relatively easy to specify
  - **Canonical view volume** (3D parallel projection cuboid)
  - **Canonical view position** (camera at the origin, looking along the negative *z*-axis)
- A step-by-step approach
  1. Get all parameters for view specification
  2. Transform from the specified view volume into canonical view volume (This is the key step)
  3. Using canonical view, clip, project, and rasterize scene to make 2D image

From World Coordinate System to View Coordinate System

- We now know the view specification: *Position*, *Look vector*, and *Up vector*
- Need to derive an affine transformation from these parameters to translate and rotate the canonical view into our arbitrary view
  - The scaling of the image (i.e., the cross-section of the view volume) to make a square cross-section will happen at a later stage, as will the clipping operation
- Translation is easy to find: we want to translate the origin to the point *Position*; therefore, the translation matrix is

$$T(Position) = \begin{bmatrix} 1 & 0 & 0 & -Pos_x \\ 0 & 1 & 0 & -Pos_y \\ 0 & 0 & 1 & -Pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation is much harder: how do we generate a rotation matrix from the viewing specifications to turn *x*, *y*, *z*, into *u*, *v*, *n*?

Rotation Components

- We have already known how to conduct rotation operations with respects to arbitrary axis
- Also, we have already discussed the transformations between two coordinate systems earlier in our lectures
- Those techniques should be employed to define three mutually independent axes in 3D and take care of the transformation between the two coordinate systems

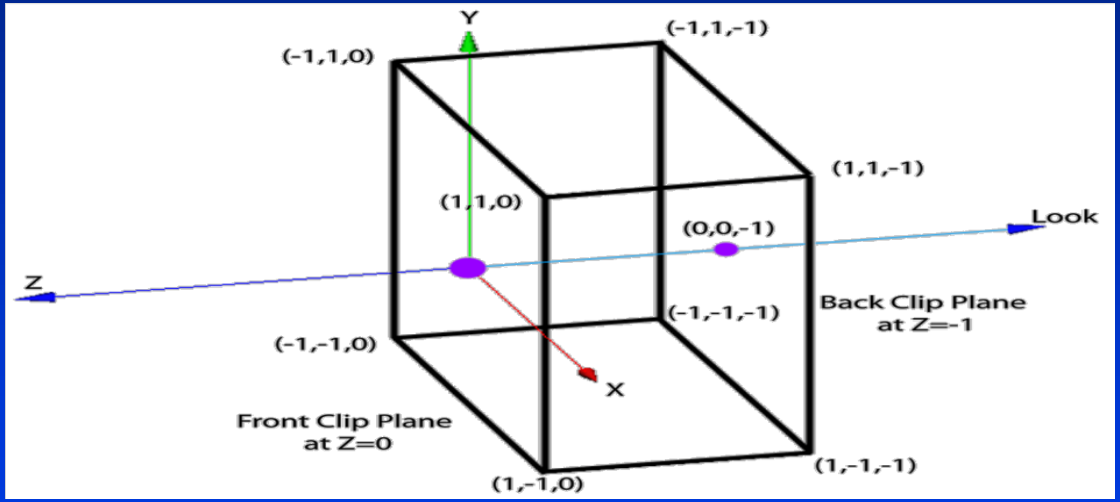
Rotation Matrix

- Want to build a rotation matrix to normalize the camera-space unit vector axes (u, v, n) into the world-space axes (x, y, z).
  - Rotation matrix M will turn (x, y, z) into (u, v, n) and has **columns (u, v, n) ➔ viewing matrix**
  - Conversely, $M^{-1}=M^T$ turns (u, v, n) into (x, y, z). M^T has **rows (u, v, n) ➔ normalization matrix**
- Reduces the problem of finding the correct rotation matrix into finding the correct perpendicular unit vectors u, v, and n
- Using *Position*, *Look vector*, and *Up vector*, compute viewing rotation matrix M with columns u, v, and n, then use its inverse, the transpose M^T, with row vectors u, v, n to get the normalization rotation matrix

Canonical View

- Given a parallel view specification and vertices of a bunch of objects, we use the normalizing transformation, i.e., the inverse viewing transformation, to normalize the view volume to a cuboid at the origin, then clip, and then project those vertices by ignoring their *z* values
- How about Perspective Projection???
- Normalize the perspective view specification to a unit frustum at the origin looking down the –z axis; then transform the perspective view volume into a parallel (cuboid) view volume, simplifying both clipping and projection

Canonical View Volume



- Note: it's a cuboid, not a cube (transformation arithmetic and clipping are easier)

Steps for Normalizing View Volume (Parallel Projection)

- We need to decompose this process into multiple steps (each step is a simple matrix)
- Each step defined by a matrix transformation
- The product of these matrices defines the entire transformation in one large, composite matrix. The steps comprise:
  - Move the eye/camera to the origin
  - Transform the view so that (u, v, n) is aligned with (x, y, z)
  - Adjust the scales so that the view volume fits between -1 and 1 in x and y, the far clip plane lies at $z = -1$, the near plane at $z = 0$

Steps for Normalizing View Volume (Perspective Projection)

- The earlier processes are the SAME AS that of the parallel projection, but we need to add one more step:
  - distort pyramid to cuboid to achieve perspective distortion to align the near clip plane with $z = 0$

Perspective Projection (Move the Eye to the Origin)

- We want to have a matrix to transform (Pos_x, Pos_y, Pos_z) to (0, 0, 0)
- Solution: it's just the inverse of the viewing translation transformation:

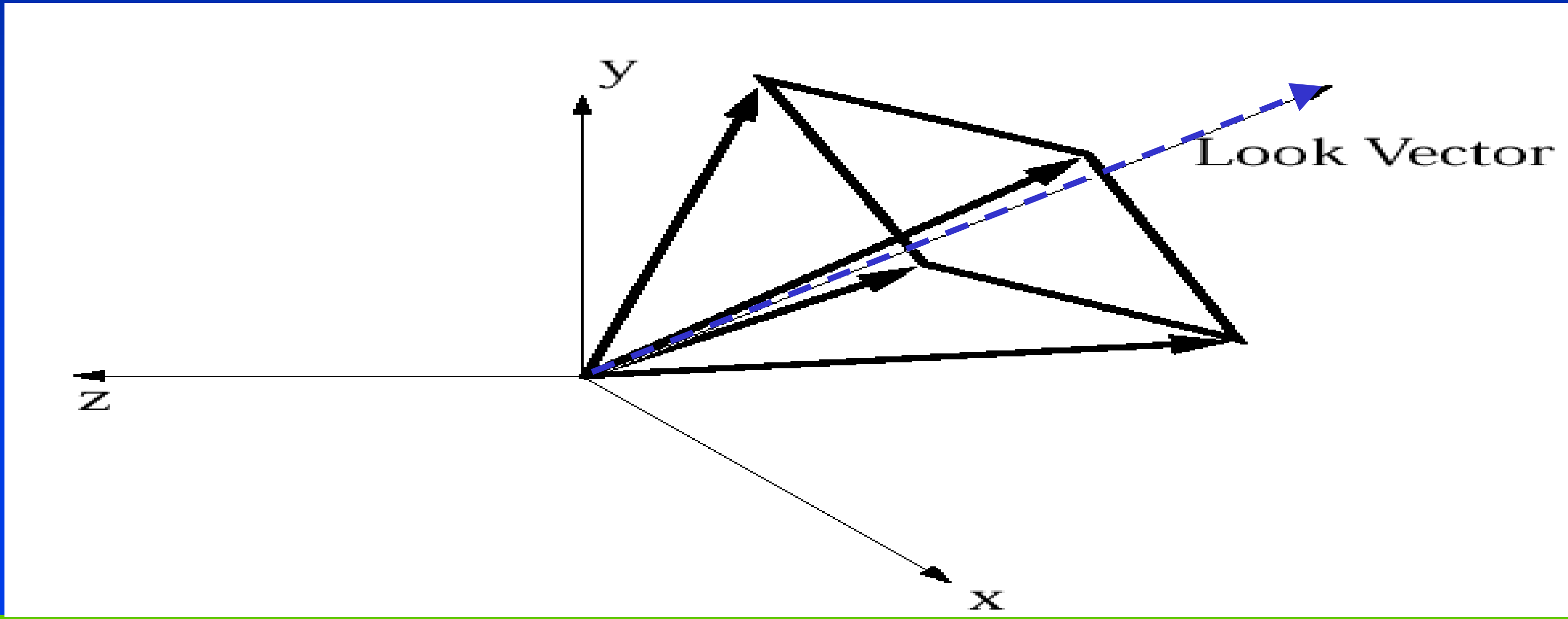
  $(t_x, t_y, t_z) = (-Pos_x, -Pos_y, -Pos_z)$
- We will take the matrix as follows, and we will multiply all vertices explicitly (and the camera implicitly) to preserve the relationship between camera and scene, i.e., for all vertices *p*
- This will move *Position* (the "eye point") to (0, 0, 0)

$$T_{trans} = \begin{bmatrix} 1 & 0 & 0 & -Pos_x \\ 0 & 1 & 0 & -Pos_y \\ 0 & 0 & 1 & -Pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$p' = T_{trans} p$$

Axis Alignment

- Align orientation with respects to (*x,y,z*) world coordinate system
- Normalize proportions of the view volume



Orientation Alignment

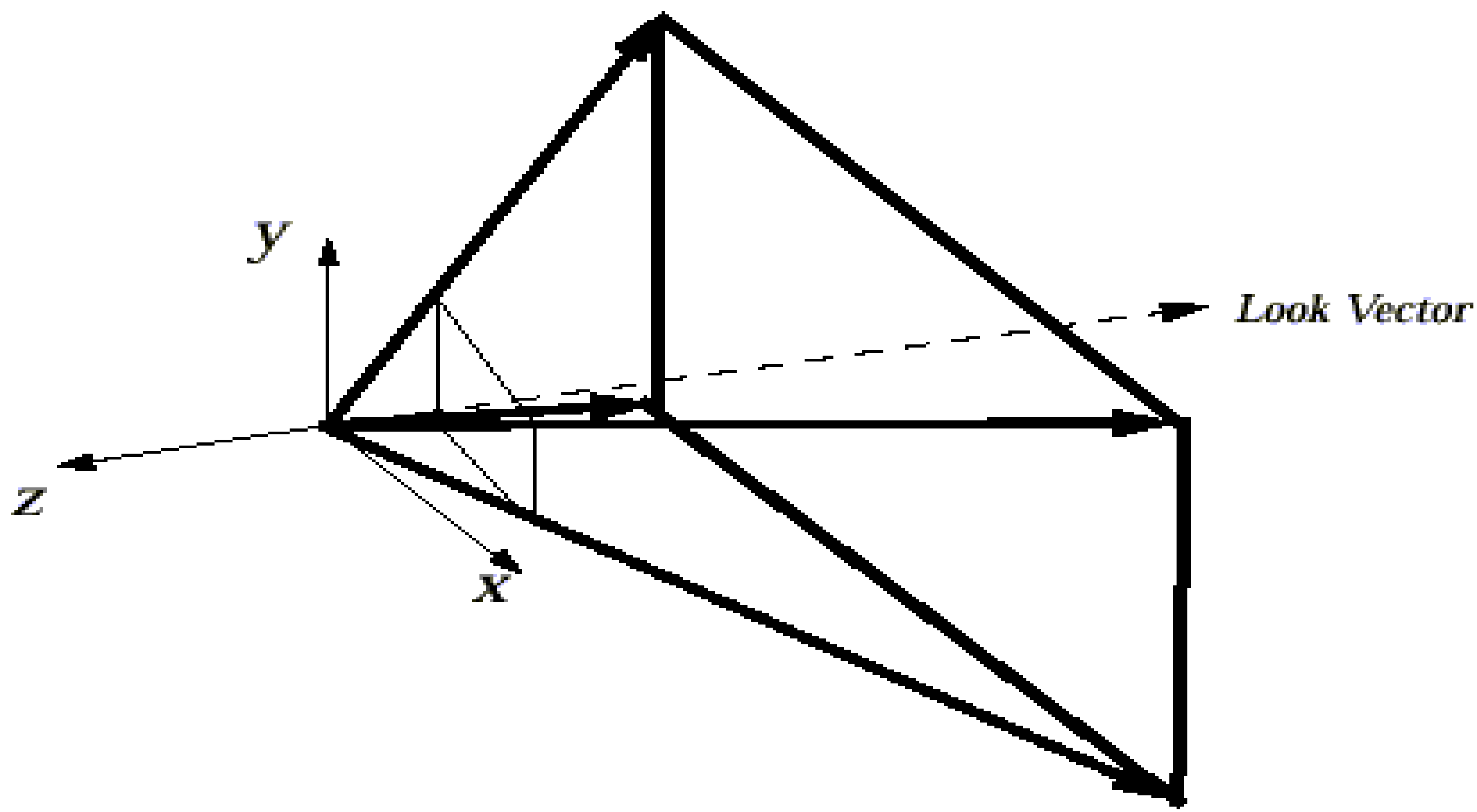
Rotate the view volume and align with the world coordinate system

- We notice that the view transformation matrix *M* with columns *u*, *v*, and *n* would rotate the *x*, *y*, *z* axes into the *u*, *v*, and *n* axes
- We now apply the inverse (transpose) of that rotation, M^T, to the scene. That is, a matrix with *rows* *u*, *v*, and *n* will rotate the axes *u*, *v*, and *n* into the axes *x*, *y*, and *z*
  - Define M_{rot} to be this rotation matrix transpose
- Now every vertex in the scene (and the camera implicitly) is multiplied by the composite matrix

$$\boxed{M_{rot} T_{trans}}$$

We have translated and rotated, so that the *Position* is at the origin, and the (*u*, *v*, *n*) axes and the (*x*, *y*, *z*) axes are aligned

Axis Alignment

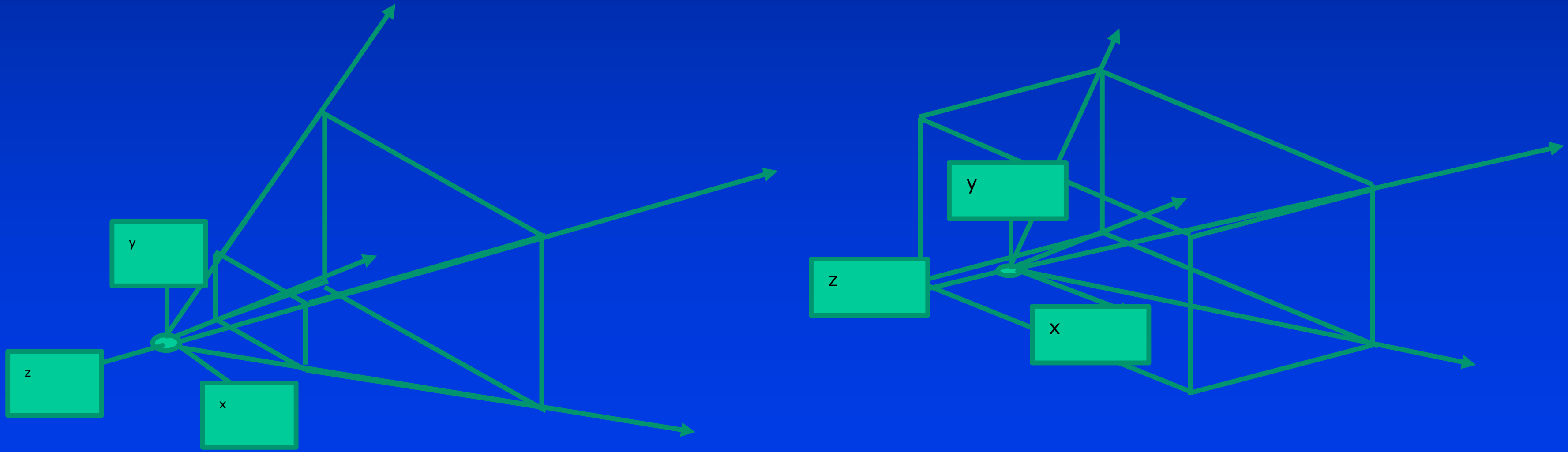


STONY BROOK
STATE UNIVERSITY OF NEW YORK

Scale the View Volume

- We have moved things more or less to the right position, but the size of the view volume needs to be normalized…
  - last affine transformation: **scaling**
- Need to be normalized to a square cross-section 2-by-2 units
  - why is that preferable to the unit square?
- Adjust so that the corners of far clipping plane eventually lie at (± 1, ± 1, -1)
- One mathematical operation works for both parallel and perspective view volumes
- Imagine vectors emanating from origin passing through corners of far clipping plane. For perspective view volume, these are edges of volume. For parallel view volume, these lie inside view volume
- First step: force vectors into 45-degree angles with *x* and *y* axes
- Solution: We shall do this by scaling in *x* and *y*

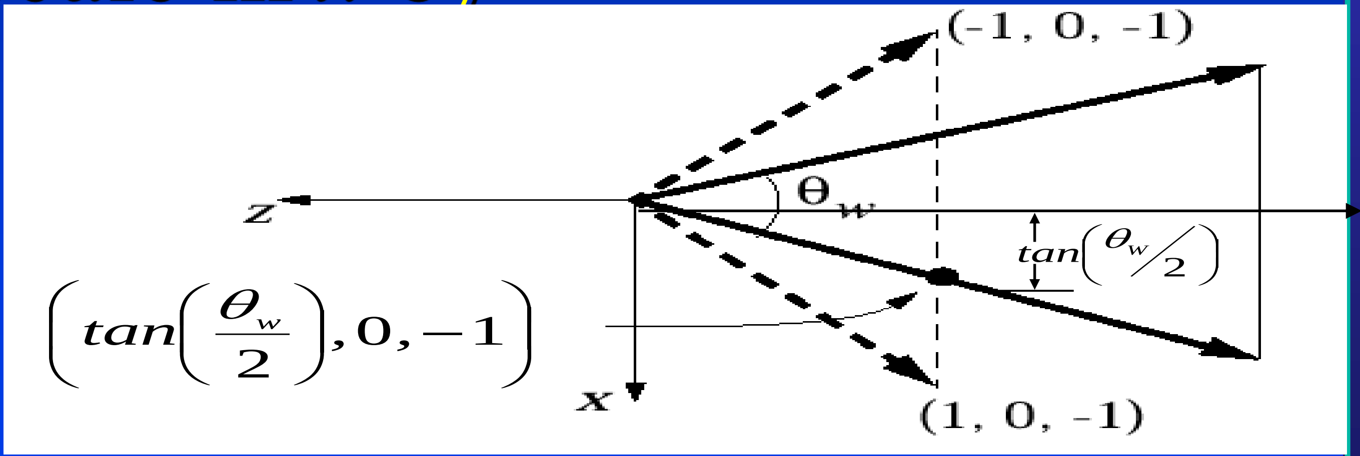
View Volume Scaling



Scale Boundary Planes

- Scale independently in *x* and *y*:
- Want to scale in *x* to make angle 90 degrees
- Need to scale in *x* by

$$\frac{1}{\left(\tan\frac{\theta_w}{2}\right)} = \cot\left(\frac{\theta_w}{2}\right)$$



Similarly in *y*

Scaling Matrix

- The scale matrix we need looks like this:

$$S_{xy} = \begin{bmatrix} \cot\left(\frac{\theta_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\theta_h}{2}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- So our current composite transformation looks like this:

$$S_{xy} M_{rot} T_{trans}$$

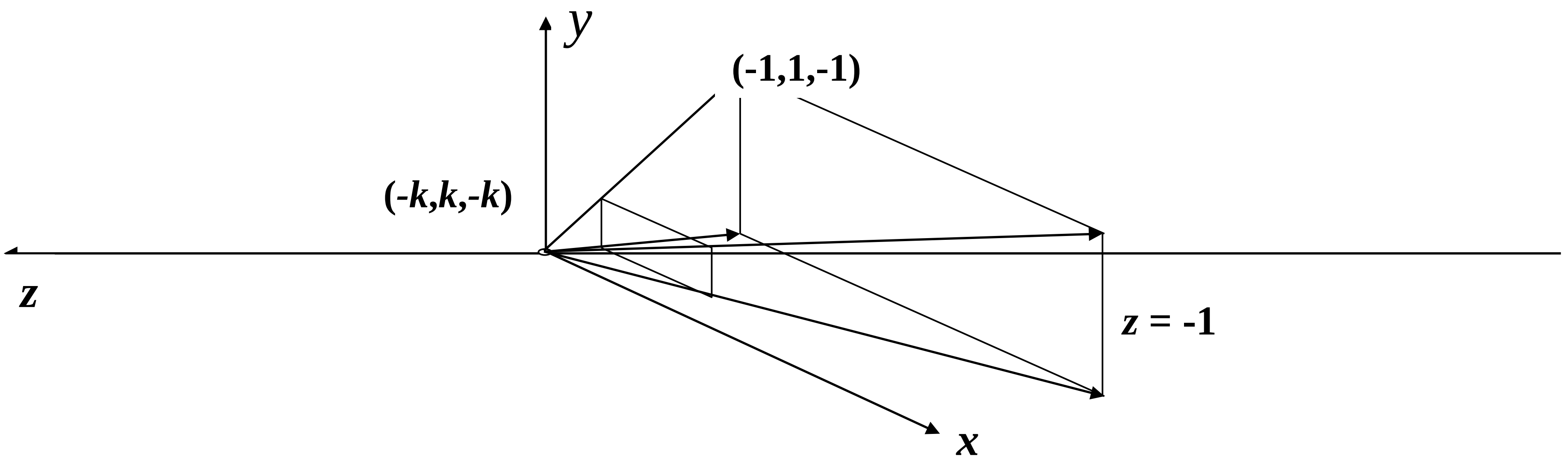
Scaling Along z-axis

- Relative proportions of view volume planes are now correct, but the back clipping plane is probably lying at some $z \neq -1$, and we want all points inside view volume to have $0 \leq z \leq -1$
- Need to shrink the back plane to be at $z = -1$
- The z distance from the eye to that point has not changed: it's still *far* (distance to the far clipping plane)
- If we scale in z only, proportions of volume will change; instead we scale uniformly:

$$S_{far} = \begin{bmatrix} \frac{1}{far} & 0 & 0 & 0 \\ 0 & \frac{1}{far} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

At Present, We Are Here

- Far plane at $z = -1$.



- Near clip plane now at $z = -k$ (note $k > 0$)

Now We Have

- Our near-final composite normalizing transformation for canonical perspective view volume:
  - T_{trans} takes the camera's *Position* and moves the camera to the world origin
  - M_{rot} takes the *Look* and *Up* vectors and orients the camera to look down the $-z$ axis
  - S_{xy} takes θ_w, θ_h and scales the clipping planes so that the corners are at $(\pm 1, \pm 1)$
  - S_{far} takes the far clipping plane and scales it to lie on the $z=-1$ plane

$$S_{far} S_{xy} M_{rot} T_{trans}$$

Perspective Transformation

- We have put the perspective view volume into the RIGHT canonical position, orientation and size
- Let's look at a particular point on the original near clipping plane lying on the *Look vector*:

$$p = Position + near \cdot Look$$

It has been changed to a new location

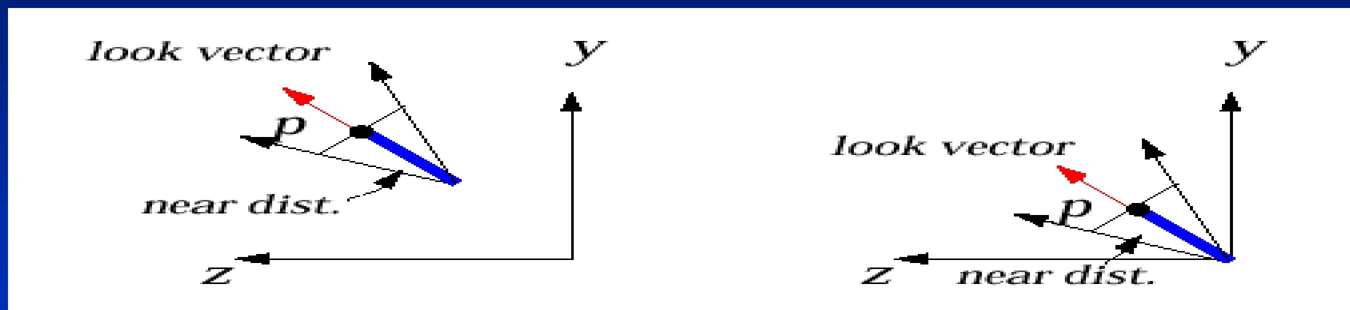
$$p' = S_{far} S_{xy} M_{rot} T_{trans} p$$

on the negative z-axis, say

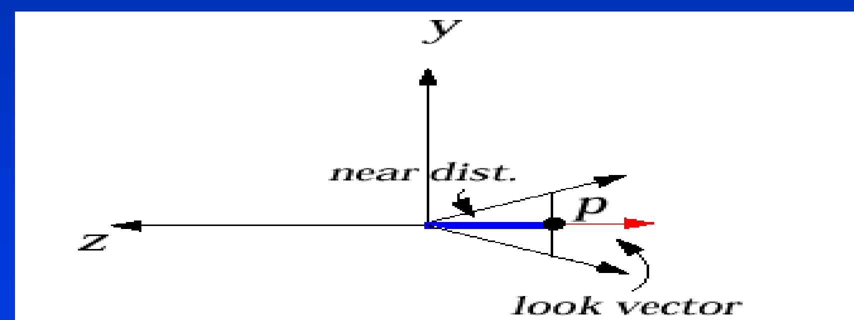
$$p' = \begin{pmatrix} 0 & 0 & -k \end{pmatrix}$$

Perspective Transformation

- What is the value of *k*? Trace through the steps. *p* first gets moved to just $near \cdot Look$



- This point is then rotated to $(near)(-e_3)$



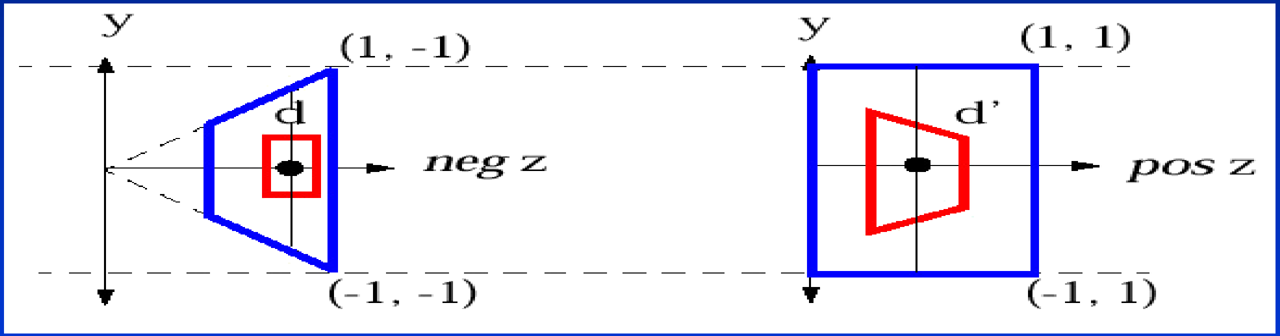
- The *xy* scaling has no effect, and the *far* scaling changes this to $\left(-\frac{near}{far}\right)e_3$, so

$$k = \frac{near}{far}$$

  - but *far* is –1, so *-near/far* is simply *near*

Perspective Transformation

- Transform points in standard *perspective* view volume between *–k* and *–1* to standard parallel view volume
- "z-buffer," used for visible surface calculation, needs z values to be [0 1], not [–1 0]. Perspective transformation must also transform scene to positive range $0 \le z \le 1$



- The matrix that does this:
- (Remember that $0 < k < 1$ ....)
- Why not originally align camera to +z axis?
  - Choice is perceptual, we think of looking through a display device into the scene that lies behind window

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{k-1} & \frac{k}{k-1} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Finally, We Have

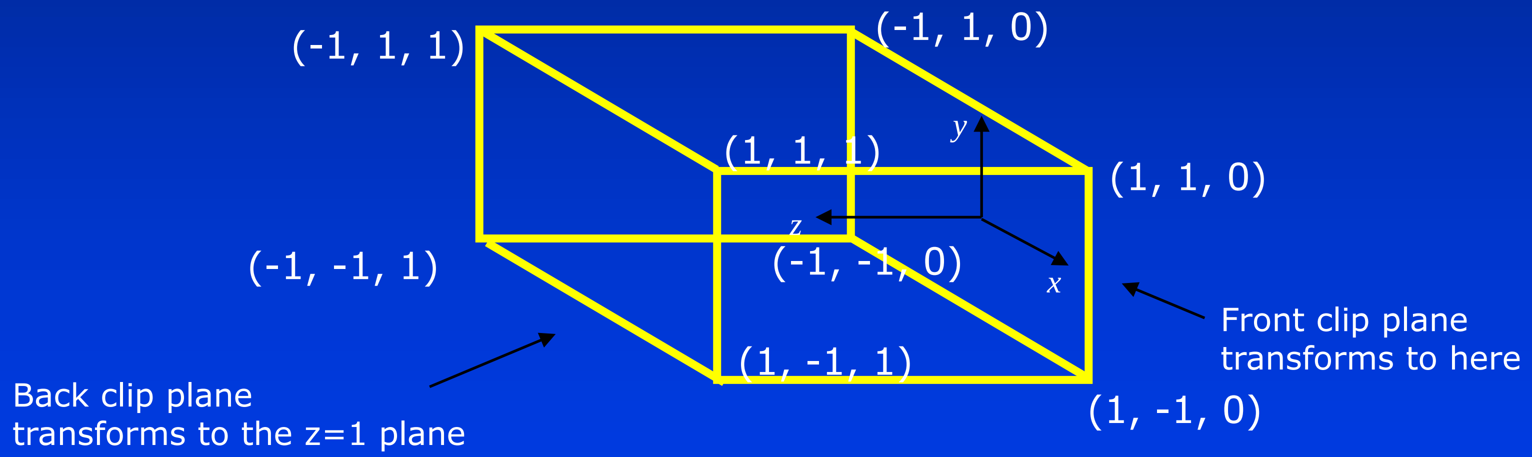
- Final transformation is here:

$$p' = D_{persp} S_{far} S_{xy} M_{rot} T_{trans} p$$

- Note that, once the viewing parameters (*Position*, *Up vector*, *Look vector*, *Height angle*, *Aspect ratio*, *Near*, and *Far*) are known, the matrices $D_{persp}, S_{far}, S_{xy}, M_{rot}, T_{trans}$
- Can all be computed and multiplied together to get a single 4x4 matrix that is applied to all points of all objects to get them from "world space" to the standard parallel view volume!!!
- What are the rationales for homogeneous coordinates???

Clipping (A Quick Review)

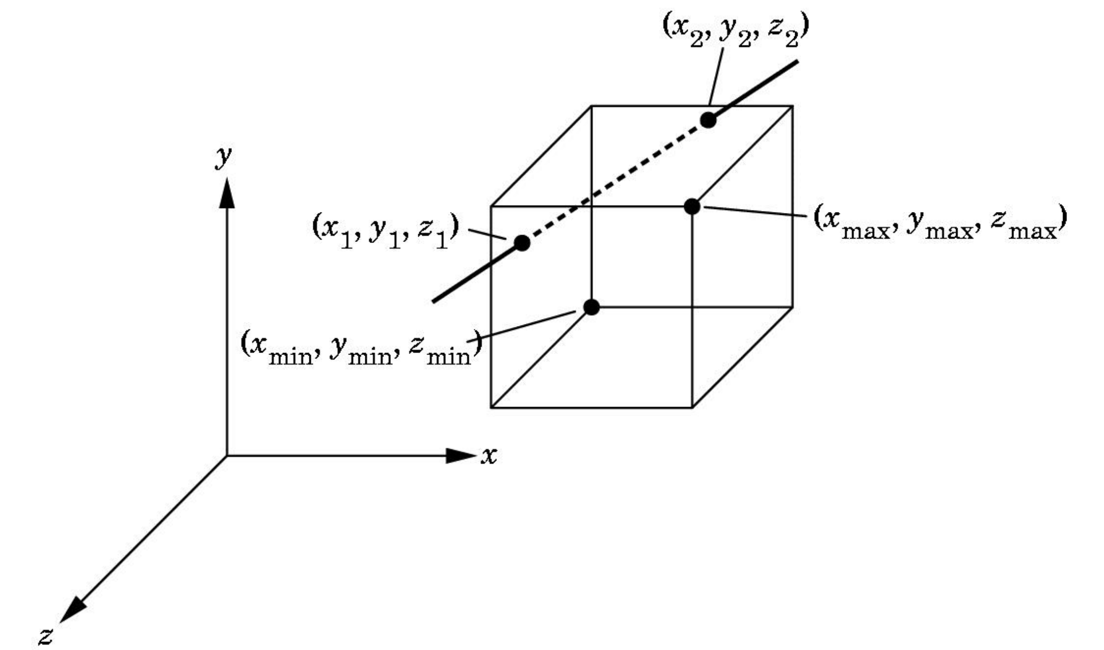
- Remaining final steps are clipping and projecting onto the image plane to produce graphical pictures
- Need to clip scene against sides of view volume
- However, we've normalized our view volume into an axis-aligned cuboid that extends from –1 to 1 in *x* and *y* and from 0 to 1 in *z*



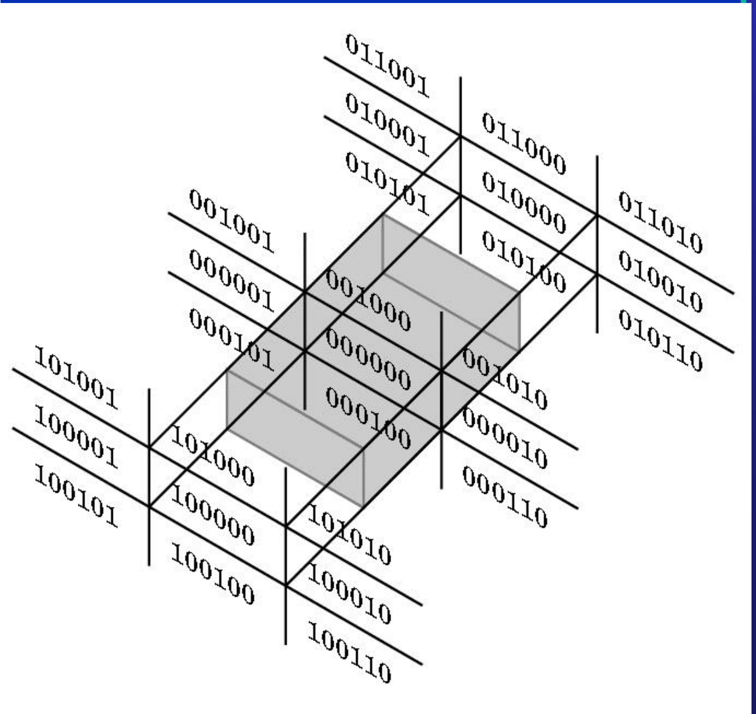
- Note that: This is the flipped (in *z*) version of the canonical view volume
- Clipping is easy! Test *x* and *y* components of vertices against +/-1. Test *z* components against 0 and 1

Clipping in 3D (Generalizations)

- Cohen-Sutherland regions



- Clip before perspective division



Department of Computer Science
Center for Visual Computing

Clipping (A Quick Review)

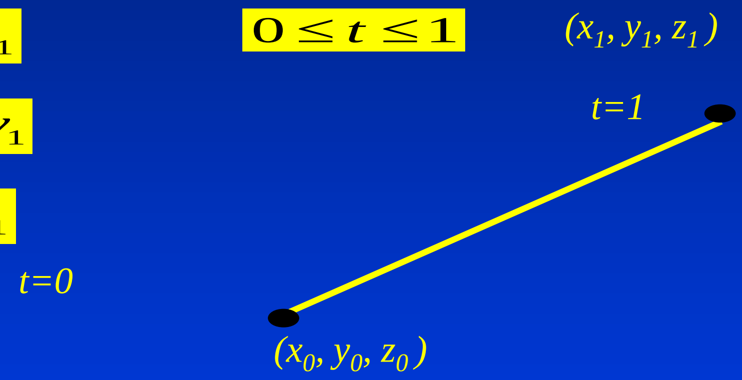
- Vertices falling within these values are saved, and vertices falling outside get clipped away; edges get clipped by knowing *x*, *y*, or *z* value at an intersection plane. Substitute *x*, *y*, or z = 1 in the corresponding parametric line equations to solve for t

$$\boxed{x = (1-t)\cdot x_0 + t\cdot x_1}$$

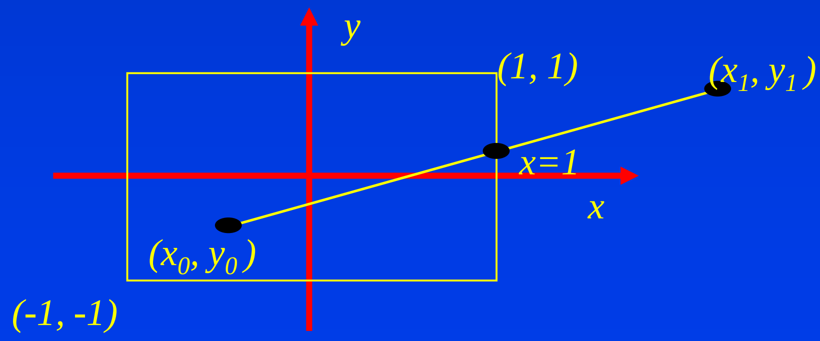
$$\boxed{y = (1-t)\cdot y_0 + t\cdot y_1}$$

$$\boxed{z = (1-t)\cdot z_0 + t\cdot z_1}$$

$$\boxed{0 \le t \le 1}$$



- In 2D:



$$1 = (1-t)x_0 + tx_1$$
$$1 - x_0 = -tx_0 + tx_1$$
$$1 - x_0 = t(x_1 - x_0)$$
$$t = \frac{1 - x_0}{x_1 - x_0}$$

Projecting to the Screen (Device Coordinates)

- Can make an image by taking each point and "ignoring *z*" to project it onto the *xy*-plane
- A point (*x*,*y*,*z*) where

$$-1 \le x, y \le 1, 0 \le z \le 1$$

$$0 \le x', y' < 1024$$

turns into the point (*x'*, *y'*) in screen space (assuming viewport is the entire screen) with

$$x' \to 512(x+1)$$
$$y' \to 512(y+1)$$

by
  - ignoring *z*
- If viewport is inside a Window Manager's window, then we need to scale down and translate to produce "window coordinates"
- Note: because it's a parallel projection we could have projected onto the front plane, the back plane, or any intermediate plane … the final pixmap would have been the same

From World to Screen

- The entire problem can be reduced to a composite matrix multiplication of vertices, clipping, and a final matrix multiplication to produce screen coordinates.
- Final composite transformation matrix (*CTM*) is composite of all modeling (instance) transformation matrices (*CMTM*) accumulated during scene graph traversal from root to leaf, composited with the final composite normalizing transformation *N* applied to the root/world coordinate system:

1) $$N = D_{persp} S_{far} S_{xy} M_{rot} T_{trans}$$

2) $$CTM = N \cdot CMTM$$

3) $$P' = CTM \cdot P$$ for every vertex P defined in its own coordinate system

4) $$P_{screen} = 512 \cdot (P' + 1)$$ for all **clipped** *P′*

Recap:

- 1) You will be computing the normalizing transformation matrix *N* in **Camtrans**
- 2) In **Sceneview**, you will extend your Camera with the ability to traverse and compute composite modeling transformations (*CMTMs*) to produce a single *CTM* for each primitive in your scene

- Aren't homogeneous coordinates wonderfully powerful?