

# #6: Camera Perspective, Viewing, and Culling

---

CSE167: Computer Graphics

Instructor: Ronen Barzel

UCSD, Winter 2006

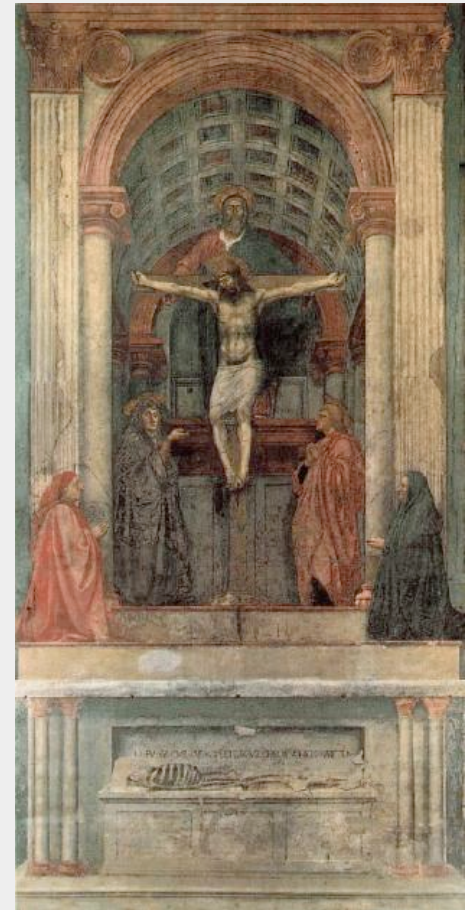
# View Projections

---

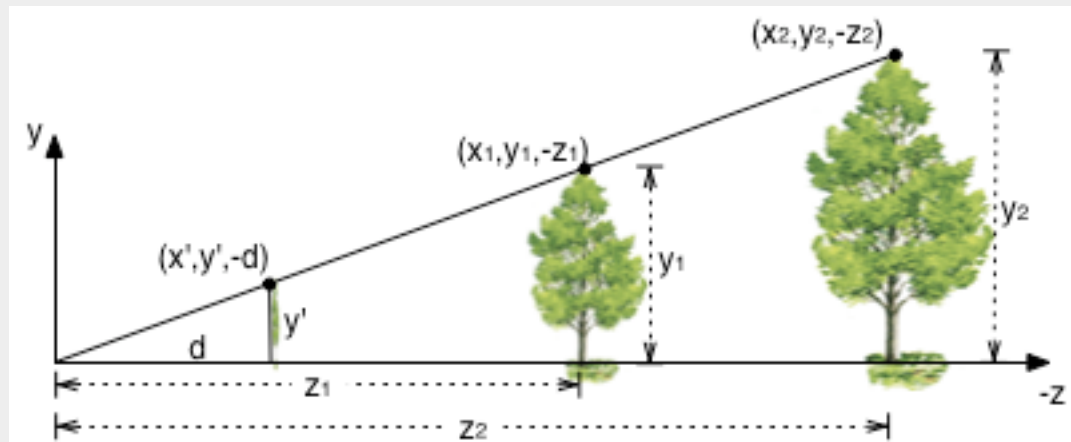
- Transform from camera space to normalized view space
- Two basic kinds:
  - *Perspective projection*: make things farther away seem smaller
    - Most common for computer graphics
    - Simple model of human eye, or camera lens
    - (Actually, a model of an ideal *pinhole camera*)
  - *Orthographic projection*: simply flatten, without any perspective
    - Used for architectural or plan views (top,side,front)
    - Not used for realistic rendering
- Others, more complex:
  - lens, with focus & depth of field
  - fish-eye lens
  - dome projection
  - computations don't easily fit into basic hardware rendering pipeline

# Perspective Projection

- Things farther away get smaller
- Parallel lines no longer parallel: vanishing point
- Discovery/formalization attributed to Filippo Brunelleschi in the early 1400's
- Earliest example: **La Trinitá** (1427) by Masaccio



# Perspective Projection



- Assume that we have “film” at distance  $d$  from the eye
- Distant tall object projects to same height as near small object
- By similar triangles, we have:
$$\frac{y'}{d} = \frac{y_1}{z_1} = \frac{y_2}{z_2}$$

Giving the transformation relations:

$$y' = d \frac{y}{z}, \quad x' = d \frac{x}{z}$$

- Notice: divide by  $z$ 
  - not a linear operation!

# Perspective Projection

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} d_1 \frac{x}{z} \\ d_2 \frac{y}{z} \\ A + \frac{B}{z} \end{bmatrix}$$

- Not a linear equation
  - not an affine transformation
  - doesn't preserve angles-but *does* preserve straight lines
  - Note: it will blow up if  $z=0$  (object at the eye)
- Z maps to *pseudo-distance*
  - necessary to preserve straight lines
  - maintains depth order when  $B < 0$ : if  $z_1 < z_2$  then  $z'_1 < z'_2$
- We'll come up with values for  $d_1$ ,  $d_2$ ,  $A$ , and  $B$ , in a little while
  - will choose them to keep area of interest within -1 to 1 in  $x, y, z$
- Ugly formula. Make it work with homogeneous matrices...

# Homogeneous Perspective Projection

- The homogeneous perspective projection matrix. Notice the last row!

$$\mathbf{P} = \begin{bmatrix} d_1 & 0 & 0 & 0 \\ 0 & d_2 & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- Multiply it by a homogeneous point

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} d_1x + 0 + 0 + 0 \\ 0 + d_2y + 0 + 0 \\ 0 + 0 + Az + B \\ 0 + 0 + z + 0 \end{bmatrix} = \begin{bmatrix} d_1x \\ d_2y \\ Az + B \\ z \end{bmatrix}$$

- Notice that the result doesn't have  $w=1$ . So *divide by w*:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} \Rightarrow \begin{bmatrix} x' / w' \\ y' / w' \\ z' / w' \\ w' / w' \end{bmatrix} = \begin{bmatrix} d_1x / z \\ d_2y / z \\ A + B / z \\ 1 \end{bmatrix}$$

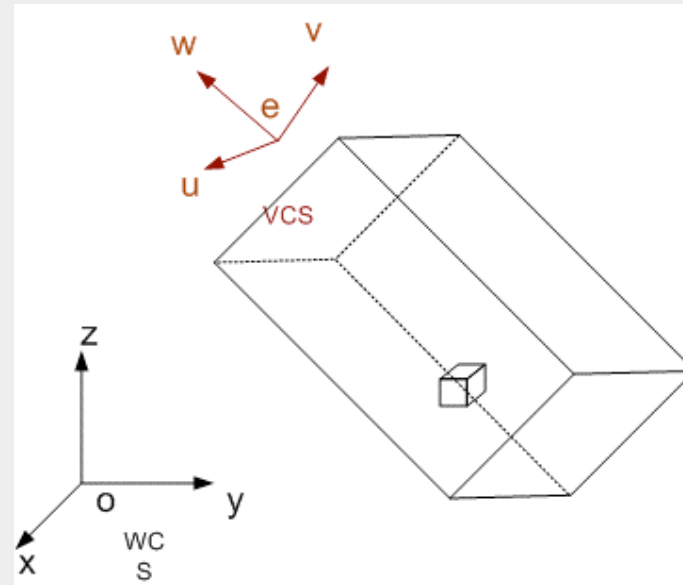
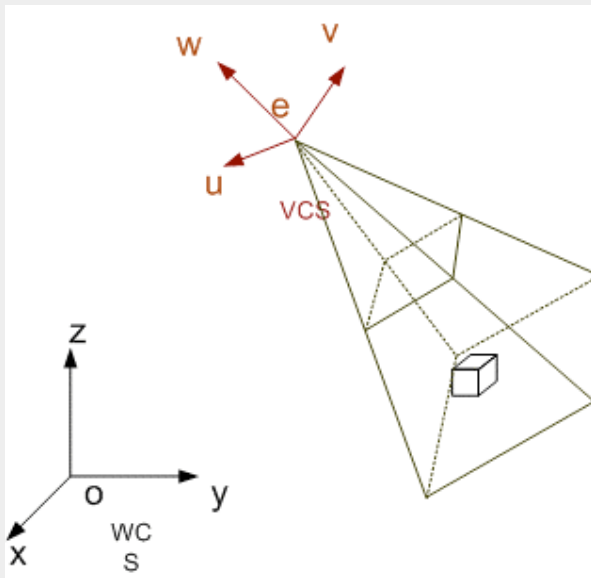
# Homogeneous Perspective Transform

---

- As always, there's some deep math behind this...
  - 3D projective space
- For practical purposes:
  - Use homogeneous matrices normally
  - Modeling & viewing transformations use *affine matrices*
    - points keep  $w=1$
    - no need to divide by  $w$  when doing modeling operations or transforming into camera space
  - Projection transform uses *perspective matrices*
    - $w$  not always 1
    - divide by  $w$  after performing projection transform
    - AKA *perspective divide*, *homogeneous divide*
  - GPU hardware does this

# View Volume

- A 3D shape in world space that represents the volume viewable by the camera



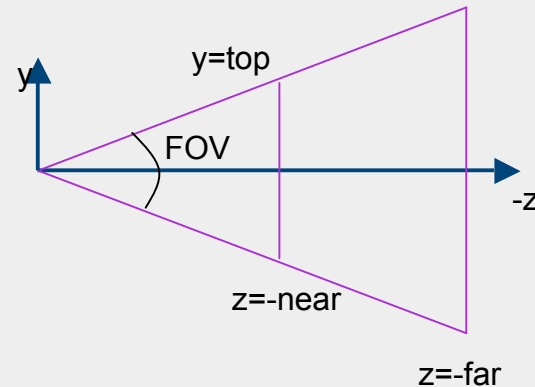
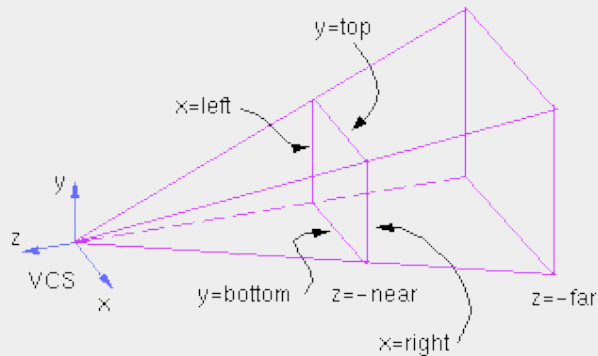


# Perspective view volume

---

- A perspective camera with a rectangular image describes a pyramid in space
  - The tip of the pyramid is at the eye point
  - The pyramid projects outward in front of the camera into space
  - Nominally the pyramid starts at the eye point and goes out infinitely...
  - But, to avoid divide-by-zero problems for objects close to the camera
    - introduce a *near clipping plane*
    - objects closer than that are not shown
    - chops off the tip of the pyramid
  - Also, to avoid floating-point precision problems in the Z buffer
    - introduce a *far clipping plane*
    - objects beyond that are not shown
    - defines the bottom of the pyramid
  - A pyramid with the tip cut off is a truncated pyramid, AKA a *frustum*
  - The standard perspective view volume is called the *view frustum*

# View Frustum



Parameterized by:

- left, right, top, bottom (generally symmetric)
- near, far

Or, when symmetric, by:

- **Field of view** (FOV), **aspect ratio**
- near, far
- Aspect ratio is the x/y ratio of the final displayed image. Common values:
  - 4/3 for TV & old movies; 1.66 for cartoons & European movies; 16/9 for American movies & HDTV; 2.35 for epic movies

$$\text{aspect ratio} = \frac{\text{right} - \text{left}}{\text{top} - \text{bottom}} = \frac{\text{right}}{\text{top}}$$

$$\tan(\text{FOV} / 2) = \frac{\text{top}}{\text{near}}$$

# Frustum Projection Matrix

- We can think of the view frustum as a distorted cube, since it has six faces, each with 4 sides
- The perspective projection warps this to a cube.
  - Everything inside gets distorted accordingly
  - By setting the parameters properly, we get the cube to range from -1 to 1 in all dimensions: i.e., normalized view space

$$\mathbf{P}_{persp}(FOV, aspect, near, far) = \begin{bmatrix} \frac{1}{aspect \cdot \tan(FOV / 2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(FOV / 2)} & 0 & 0 \\ 0 & 0 & \frac{near + far}{near - far} & \frac{2 \cdot near \cdot far}{near - far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

# The complete transform

---

- Composing the modeling matrix **M**, the camera matrix **C**, the projection matrix **P**, and the viewport matrix **D**, we have, for some point **p**:

$$\text{pixel} = (\mathbf{D} \mathbf{P} \mathbf{C}^{-1} \mathbf{M}) \mathbf{p}$$

# Spaces in OpenGL

---

- OpenGL separates it into:
  - $MODELVIEW = C^{-1}M$ 
    - It's up to you to compose the camera and model transforms.  
Typically start with the inverse camera transform, then push and pop model matrix values on top of it. `gluLookAt()` is a utility to do a look-at transformation.
  - $PROJECTION = P$ 
    - OpenGL provides utility routines to set the projection matrix:
      - `glFrustum()` lets you define the perspective view volume based on coordinates of the frustum
      - `glPerspective()` lets you specify the FOV, aspect, near clip and far clip distances
      - `glOrtho()` lets you specify a orthographic viewing transformation
  - $Viewport = D$ 
    - OpenGL provides a `glViewport()` routine to set the viewport