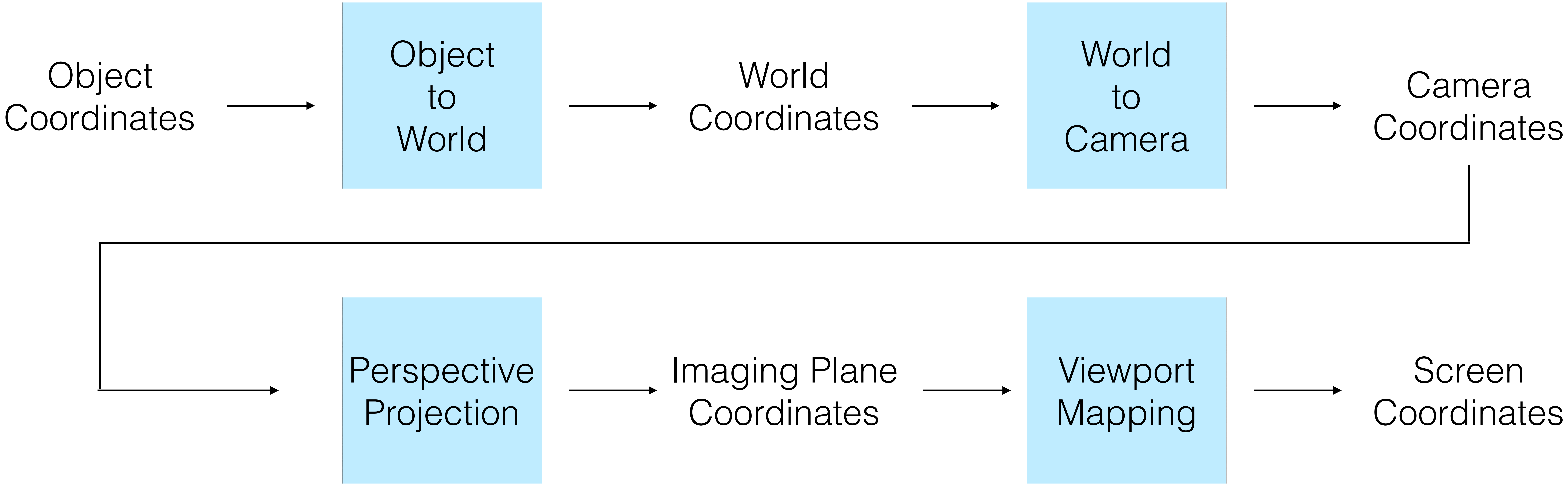
# 3D Rendering Geometry (cont’d)

CS 355: Introduction to Graphics and Image Processing

# 3D Geometry Pipeline



Let’s revisit object placement…

# 3D Linear Transformations

- Scaling has the same form as in 2D
- Translation has the same form as in 2D
- Rotation has the same form as in 2D if you begin with unit vectors for the new coordinate axes

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# 3D Rotations

- Can construct rotation matrices directly from unit vectors for the new coordinate axes
  - All rows are orthogonal
  - Any matrix with orthogonal rows is a rotation!
- Can also construct from rotation angles (looks a lot like 2D rotation matrices)
  - Around x axis (in y-z plane)
  - Around y axis (in x-z plane)
  - Around z axis (in x-y plane)

$$\mathbf{R} = \begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

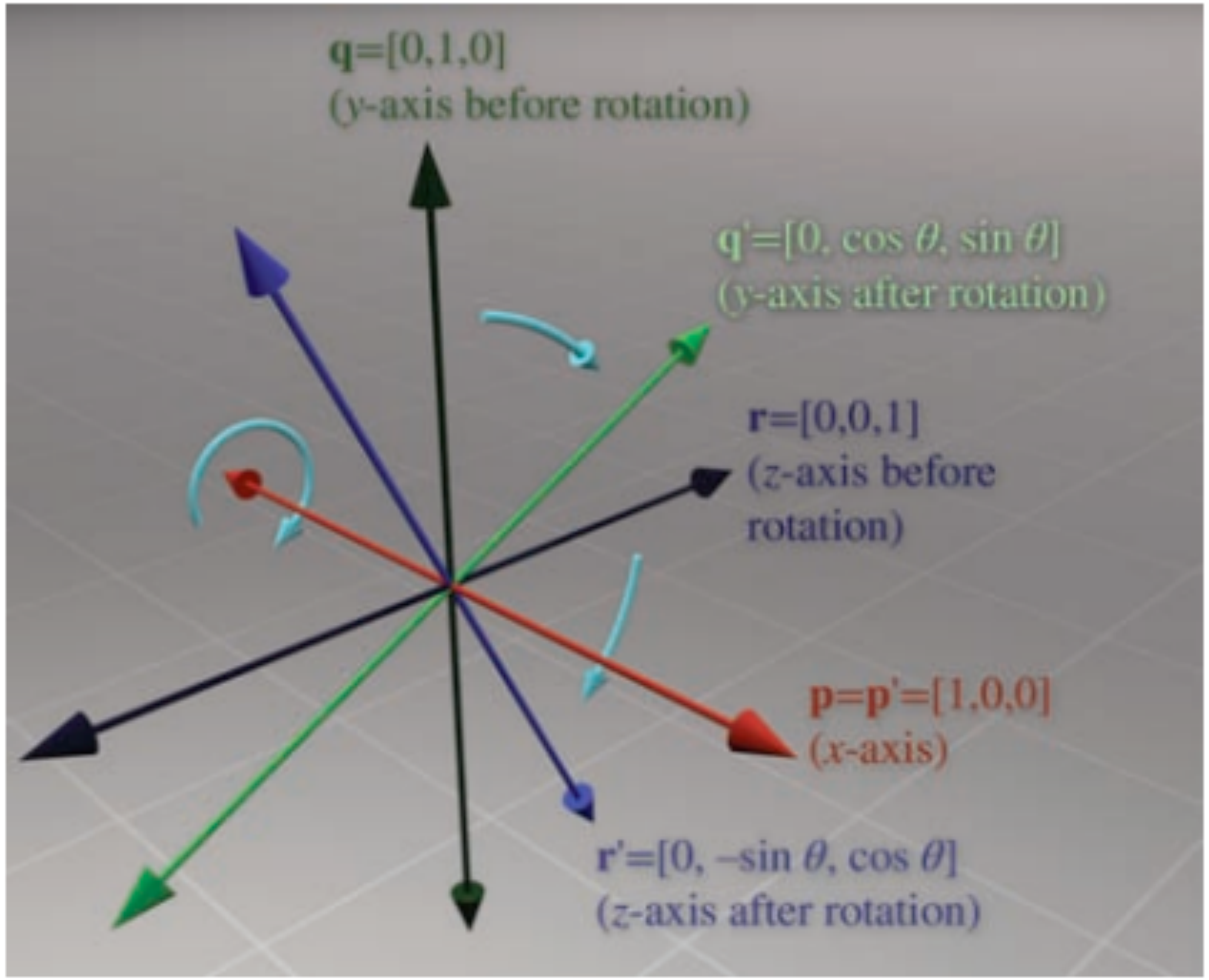
$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# 3D Rotations

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

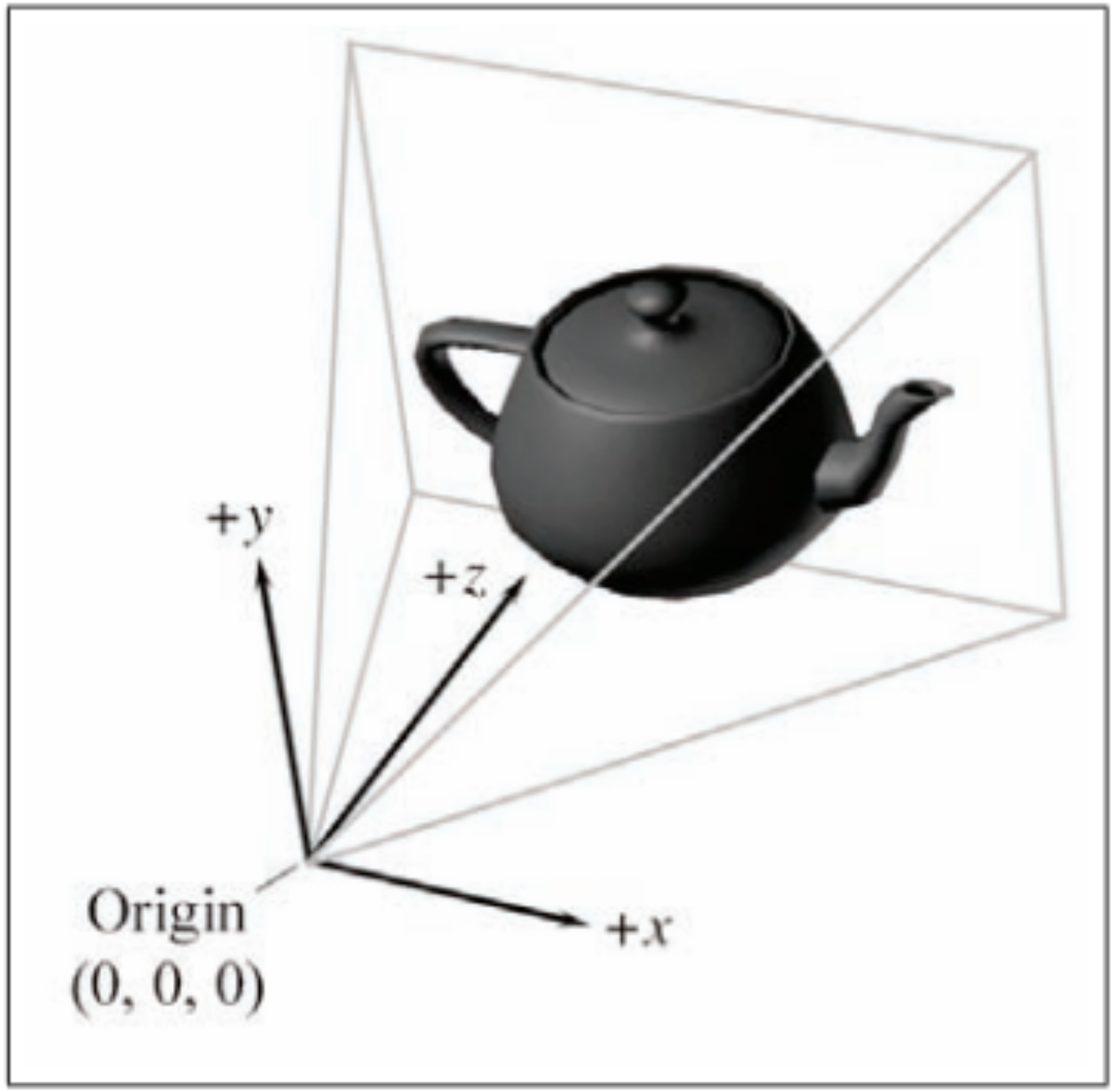
$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Let’s revisit the camera space...

# Camera Space



# World to Camera

- You need to know
  - Position of camera in world coordinates

$$\mathbf{c} = (c_x, c_y, c_z)$$

  - Orientation of camera as given by
    - a set of basic vectors in world coordinates, or
    - rotation angles

$$\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$$

Camera's x Camera's y Camera's z

Let's make it easier

# Specifying the Camera

“Look from” point $\mathbf{p}_{\text{from}}$

“Look at” point $\mathbf{p}_{\text{at}}$

“Up” vector $\mathbf{v}_{\text{up}}$

Roughly!

# Building Coordinate System

Optical axis (Z) first: $$\mathbf{e}_3 = \frac{\mathbf{p}_{\text{at}} - \mathbf{p}_{\text{from}}}{\|\mathbf{p}_{\text{at}} - \mathbf{p}_{\text{from}}\|}$$

Then side (X): $$\mathbf{e}_1 = \frac{\mathbf{e}_3 \times \mathbf{v}_{\text{up}}}{\|\mathbf{e}_3 \times \mathbf{v}_{\text{up}}\|}$$

Then straighten “up” (Y): $$\mathbf{e}_2 = \frac{\mathbf{e}_1 \times \mathbf{e}_3}{\|\mathbf{e}_1 \times \mathbf{e}_3\|}$$

“Gram - Schmidt” orthogonalization

# World to Camera

- Two steps:
  - **Translate** everything to be relative to the camera position
  - **Rotate** into the camera's viewing orientation

$$\begin{bmatrix} 1 & 0 & 0 & \boxed{\begin{matrix} -c_x \\ -c_y \\ -c_z \end{matrix}} \\ 0 & 1 & 0 & \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \boxed{e_{11} \quad e_{12} \quad e_{13} \quad 0} \\ \boxed{e_{21} \quad e_{22} \quad e_{23} \quad 0} \\ \boxed{e_{31} \quad e_{32} \quad e_{33} \quad 0} \\ 0 \quad\;\; 0 \quad\;\; 0 \quad\;\; 1 \end{bmatrix}$$

Let’s revisit the pipeline...

# Pipeline So Far

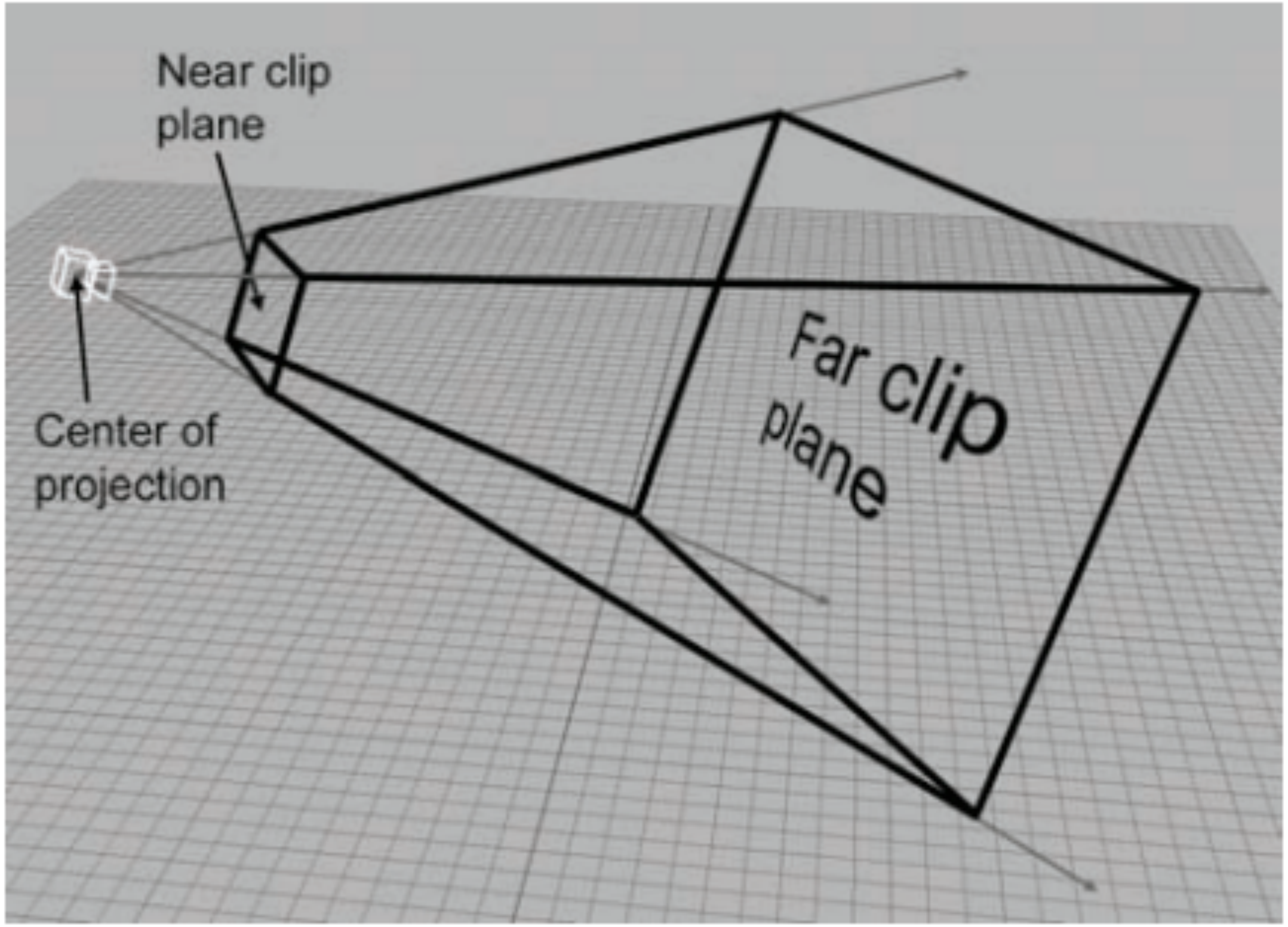
**Idea: let's cull as much as we can before dividing**

$$
\begin{bmatrix} x \\ y \\ f \\ 1 \end{bmatrix} \sim \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ Z_c/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \overbrace{\begin{bmatrix} e_{11} & e_{12} & e_{13} & 0 \\ e_{21} & e_{22} & e_{23} & 0 \\ e_{31} & e_{32} & e_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\text{World-to-camera transformation}} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}
$$

Normalize　　Project　　Rotate　　Translate

**Big problem: lots of time spent on stuff you can't see!**

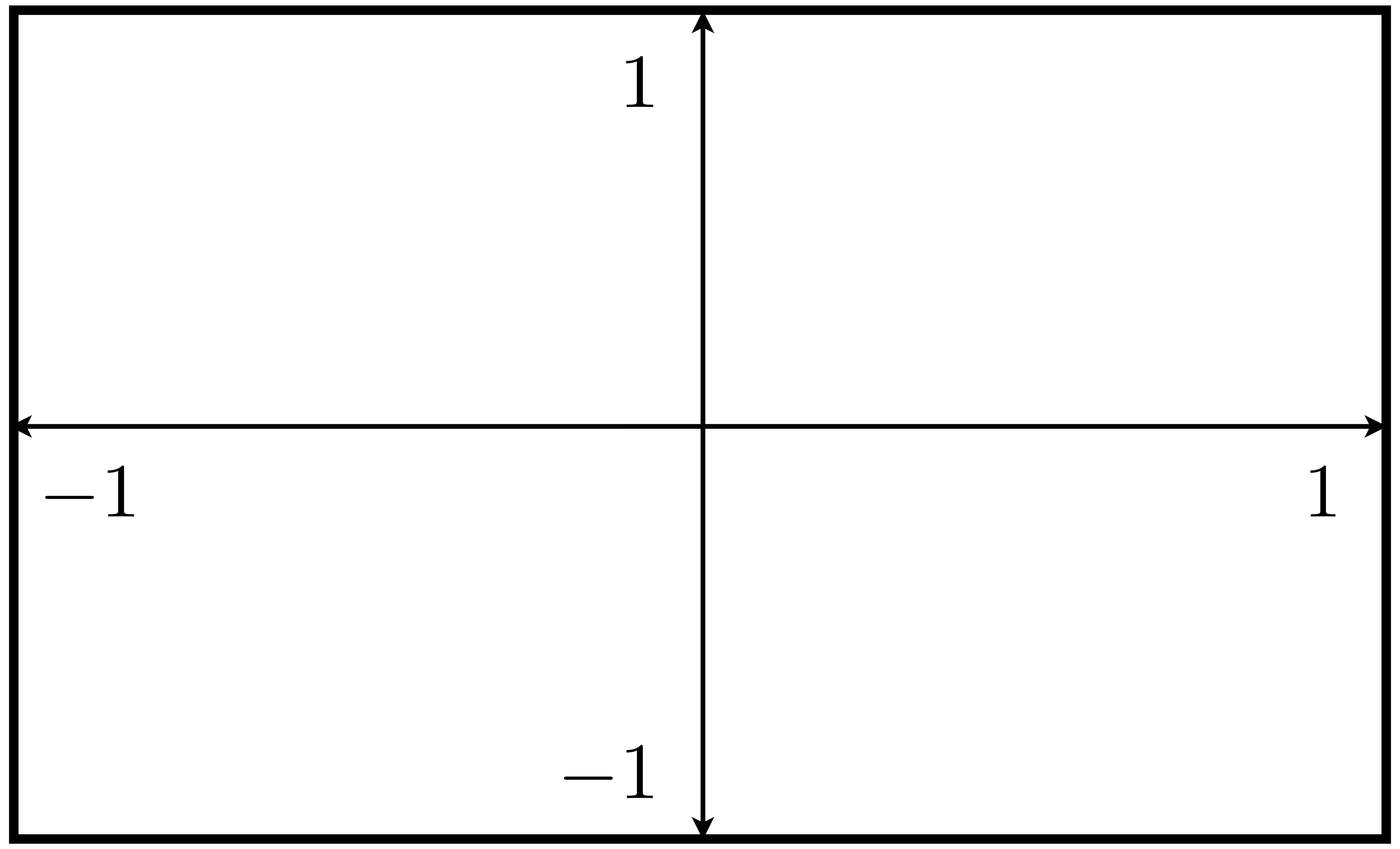
# View Frustum



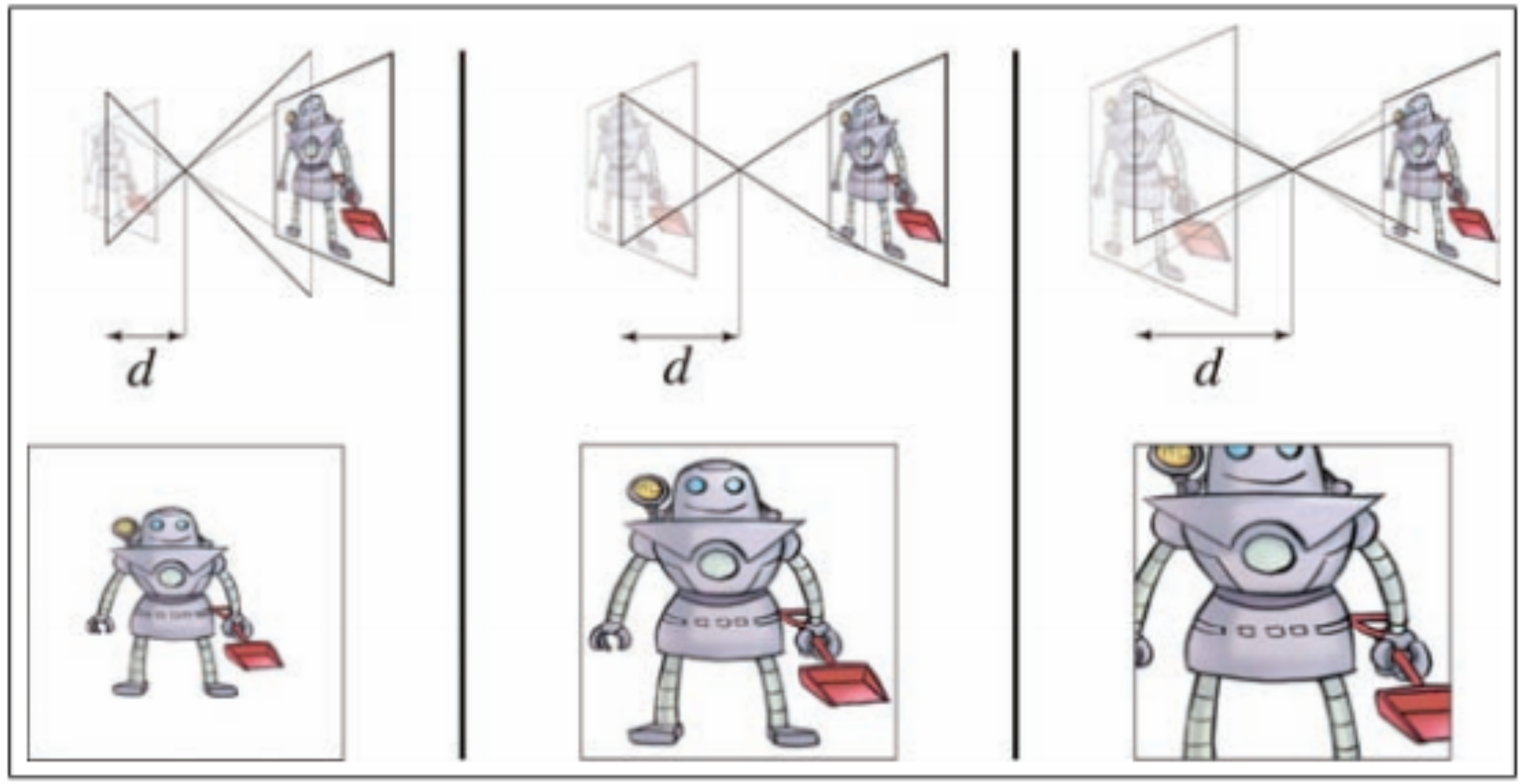
Can we clip things outside the view frustum without doing a divide?

# Canonical View

- To simplify, let's assume we map to [-1,1] in both *x* and *y* directions
- Also map [near, far] depth range to [-1,1]
- Maps frustum to $[-1,1]^3$ cube



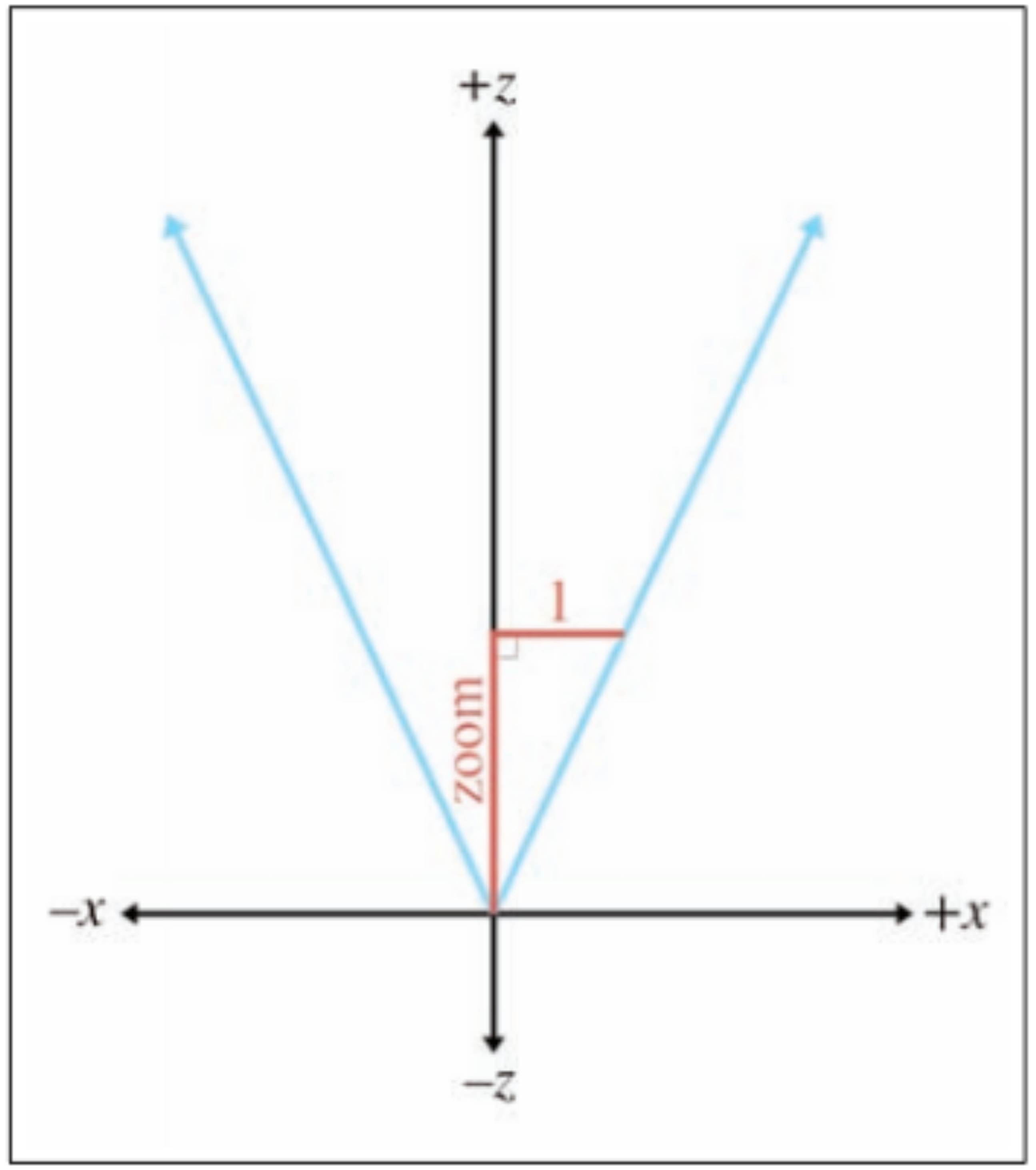
# Changing Focal Length




Changing focal length changes overall zoom,
but also affects the shape of the view frustum

# Zoom

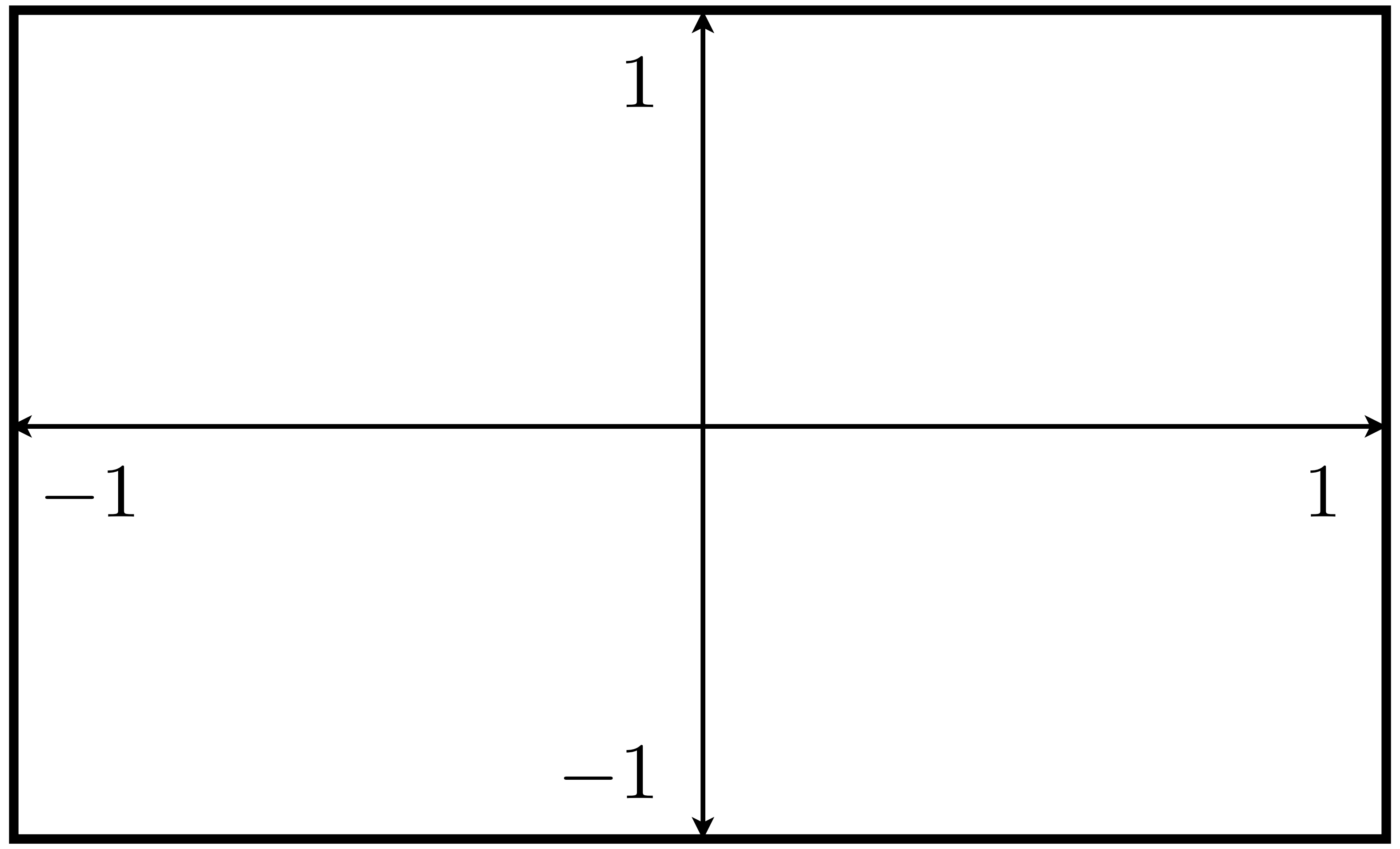
- Mapping to a canonical window loses
  - Real horizontal width
  - Real vertical width
- We'll need to fold this into our projection matrix
- Think in terms of different "zoom" levels for x and y




$$\text{zoom} = \frac{1}{\tan(\text{fov}/2)}$$

# The Clip Matrix

- Let's build a new projection matrix that
  - Scales visible x to [-1,1]
  - Scales visible y to [-1,1]
  - Scales near to far z to [-1,1]



# The Clip Matrix

- Let's build a new projection matrix that
  - Scales visible x to [-1,1]
  - Scales visible y to [-1,1]
  - Scales near to far z to [-1,1]

$$\begin{bmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$n$ = near plane distance
$f$ = far plane distance

# The Clip Matrix

$$\begin{bmatrix} \boxed{\begin{matrix} x/w \\ y/w \\ z/w \end{matrix}} \\ 1 \end{bmatrix} \sim \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \text{zoom}_x & 0 & 0 & 0 \\ 0 & \text{zoom}_y & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

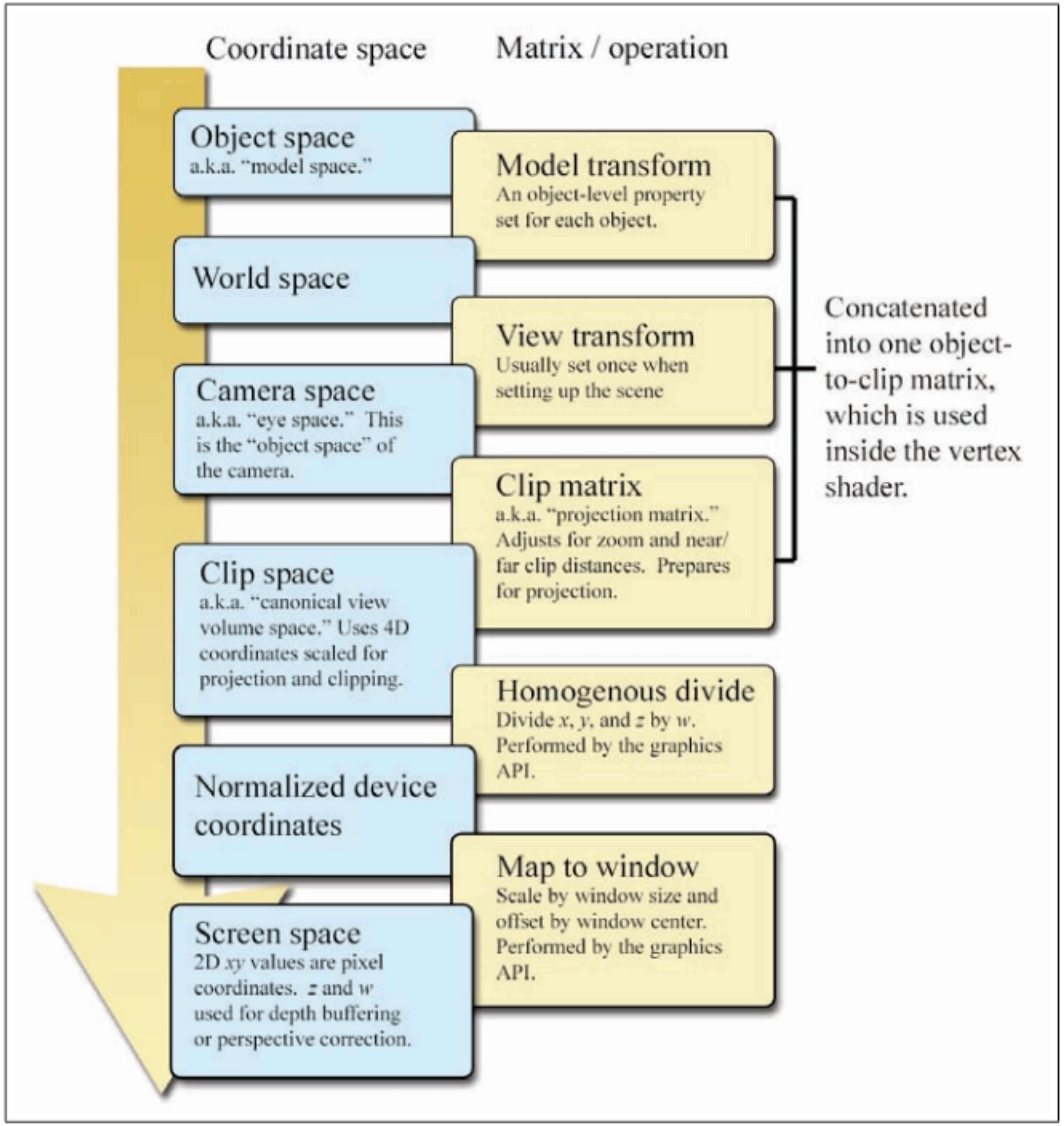
All of these are in the range [-1,1] for things in view

# Clipping Tests

| | |
|---|---|
| Left | $x < -w$ |
| Right | $x > w$ |
| Bottom | $y < -w$ |
| Top | $y > w$ |
| Near | $z < -w$ |
| Far | $z > w$ |

# Culling / Clipping

- If an entire primitive (line, polygon) fails the same clipping test, it is outside the field of view—if so, throw out
- If part fails and part passes, clip to the portion in view (create partial primitive) and process from there
- Clipping against multiple planes may not leave anything left — if so, throw out


Coordinate space
Matrix / operation
Object space
a.k.a. "model space."
Model transform
An object-level property set for each object.
World space
View transform
Usually set once when setting up the scene
Camera space
a.k.a. "eye space." This is the "object space" of the camera.
Clip matrix
a.k.a. "projection matrix." Adjusts for zoom and near/far clip distances. Prepares for projection.
Concatenated into one object-to-clip matrix, which is used inside the vertex shader.
Clip space
a.k.a. "canonical view volume space." Uses 4D coordinates scaled for projection and clipping.
Homogenous divide
Divide x, y, and z by w. Performed by the graphics API.
Normalized device coordinates
Map to window
Scale by window size and offset by window center. Performed by the graphics API.
Screen space
2D xy values are pixel coordinates. z and w used for depth buffering or perspective correction.

# To Screen Space

- Map [-1,1] x [-1,1] to screen
  - Scale x by half the width
  - Invert y and scale by half the height
  - Translate origin from center to upper left corner

$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \text{width}/2 & 0 & \text{width}/2 \\ 0 & -\text{height}/2 & \text{height}/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$$

# Rendering Geometry

- ✓ Transform from object to world coordinates
- ✓ Transform from world to camera coordinates
- ✓ Clipping: near plane, far plane, field of view
- ✓ Perspective projection
- ✓ View transformation

# Lab 7

- Repeat what you did for Lab 6
  <u>but without OpenGL</u>
- Object placement: replace OpenGL rotate/translate calls by multiplying with your own transformation matrices
- World-to-camera: likewise replace OpenGL rotate/translate calls by multiplying with your own transformation matrices
- Projection: think about how the parameters to `gluPerspective` are used to construct a clip matrix
- Clip tests: implement your own clipping tests
  - For simplicity, clip all of a line if both endpoints fail the <u>same</u> clip test
  - Except clip all of a line if either endpoint fails the near-plane test
- Divide by the homogeneous element
- Map from canonical coordinates to screen coordinates
- Draw 2D lines (see the code we give you)

# Coming up…

- Visibility
- Lighting