Projection and viewing

Computer Graphics

CSE 167

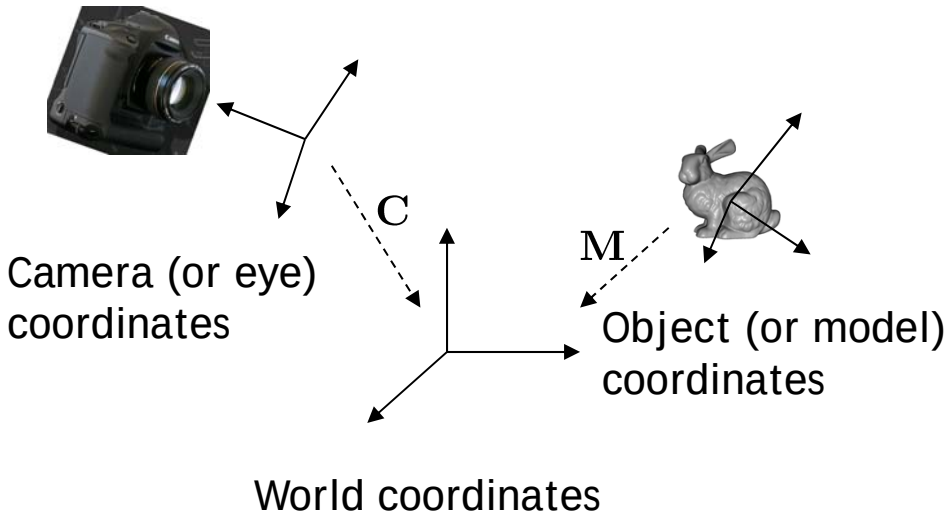
Lecture 4

CSE 167: Computer Graphics

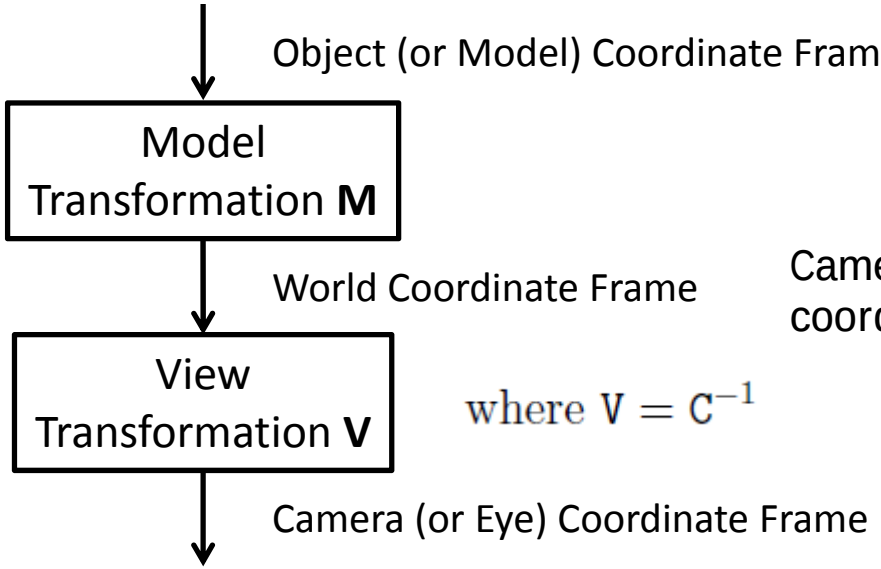
- Review: transformation from the object (or model) coordinate frame to the camera (or eye) coordinate frame
- Projection
  - Perspective projection
    - Projective homogeneous coordinates
    - Projective transformation
  - Orthographic projection
- Viewing

Review: coordinate frames

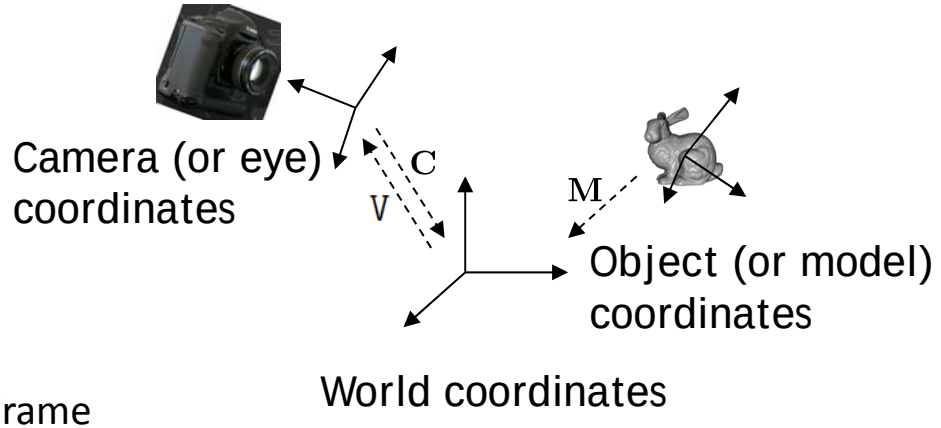
- Object (or model) coordinate frame
- World coordinate frame
- Camera (or eye) coordinate frame



Transformations



$$\text{where } \mathtt{V} = \mathtt{C}^{-1}$$



$$\mathbf{X}_{\text{camera}} = \mathtt{VM}\mathbf{X}_{\text{object}}$$

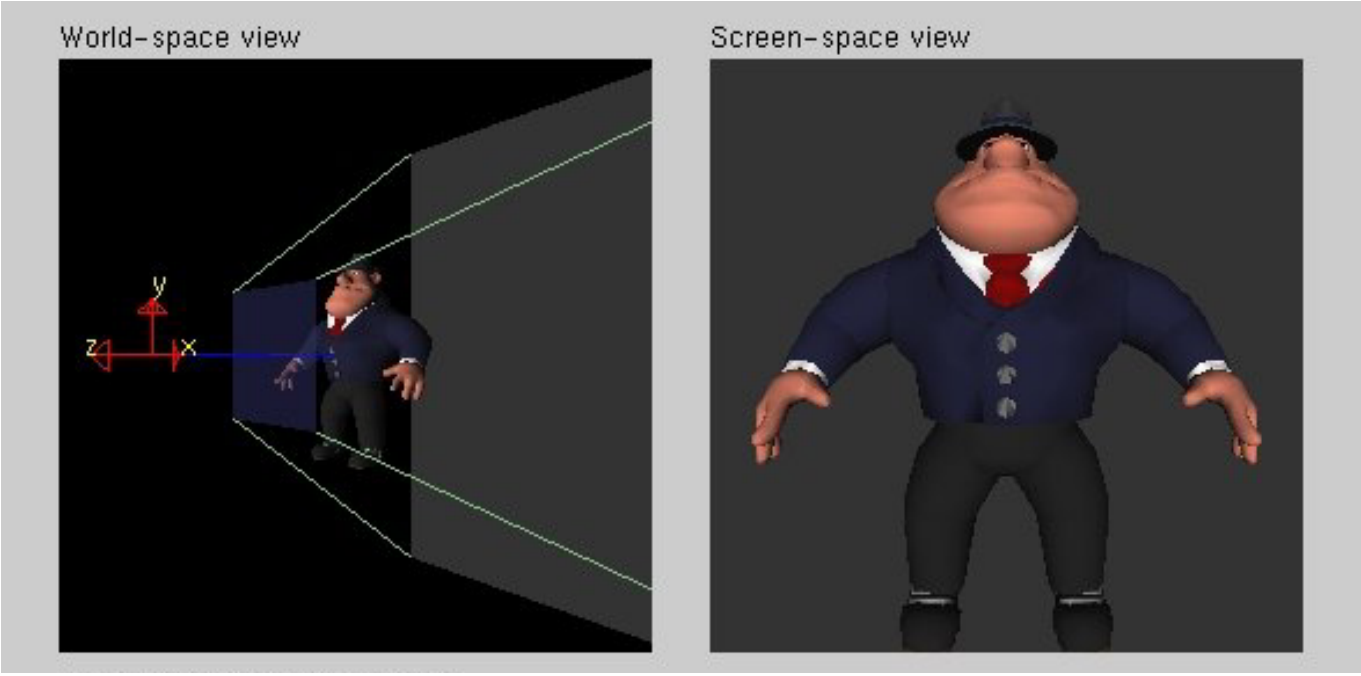
$$\mathbf{X}_{\text{camera}} = \mathtt{H}_{\text{object,camera}}\mathbf{X}_{\text{object}}$$

$$\text{where } \mathtt{H}_{\text{object,camera}} = \mathtt{VM}$$

Transform from object (or model) coordinate frame to camera (or eye) coordinate frame using a 4x4 transformation *modelview* matrix

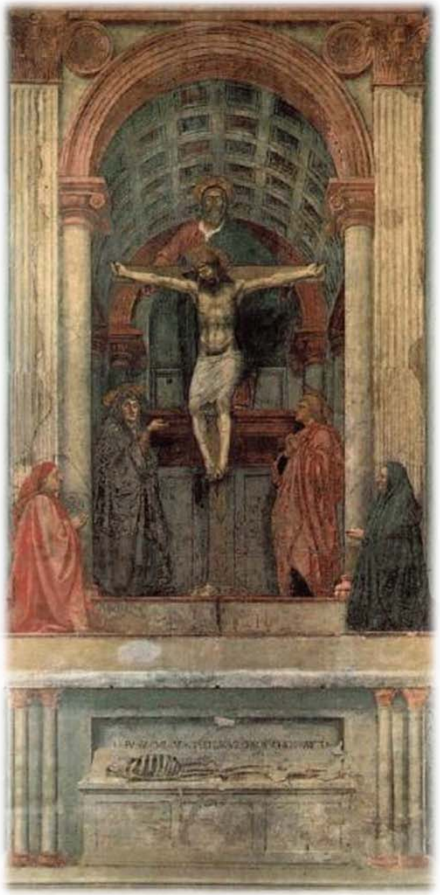
Next step: projection

- Projection of 3D objects in camera coordinate frame to 2D image



Perspective Projection

Parallel lines are
no longer parallel,
converge in one point

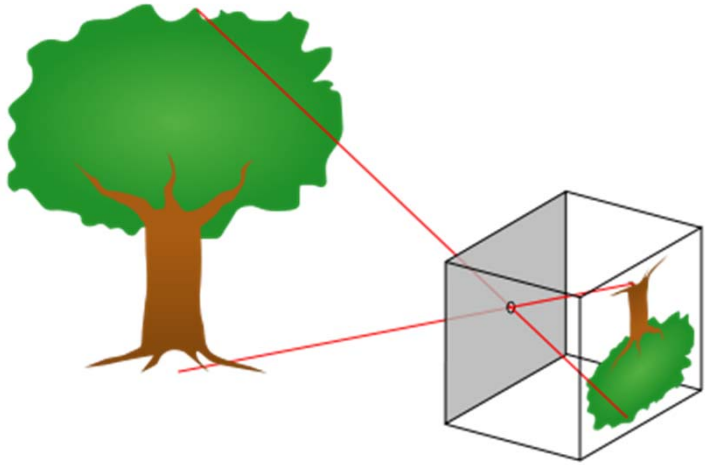


Earliest example:
La Trinitá (1427) by Masaccio

Based on slides courtesy of Jurgen Schulze

Perspective Projection

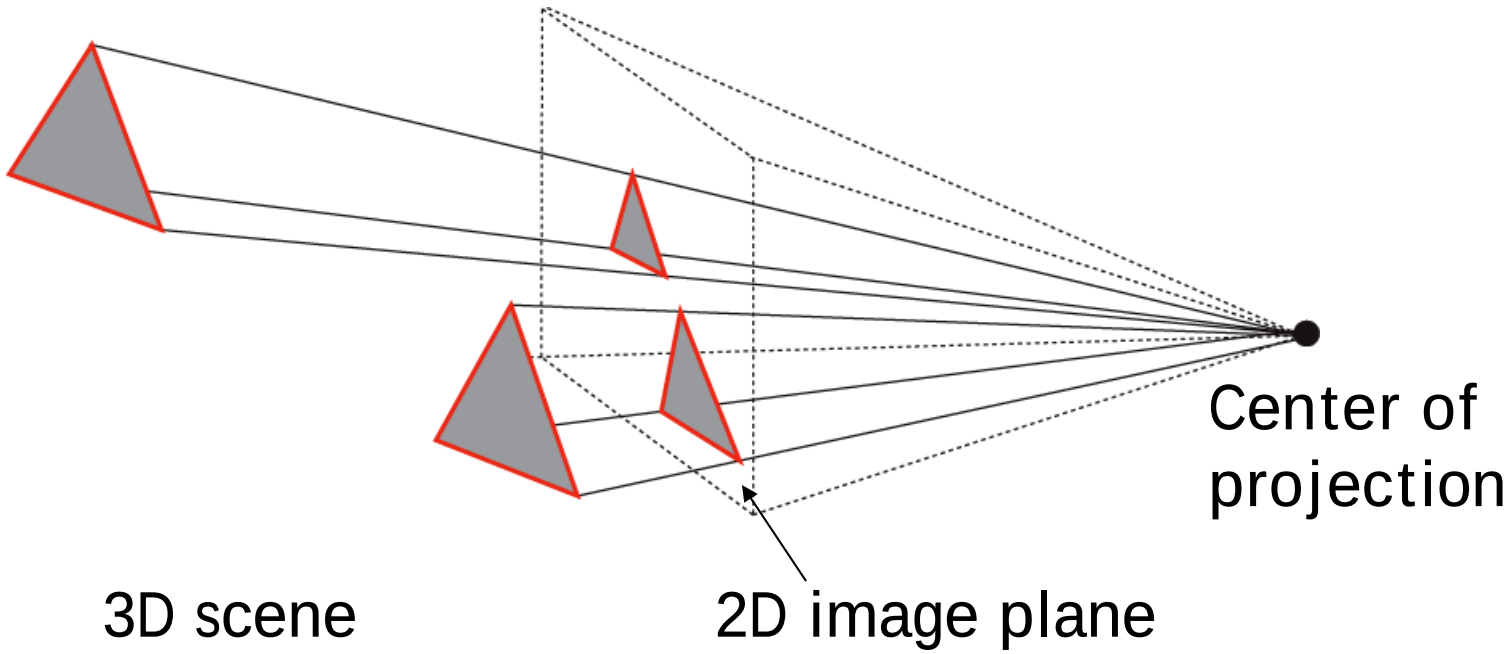
- Simplified model of pinhole camera or human eye



- Things farther away appear to be smaller
- Discovery attributed to Filippo Brunelleschi (Italian architect) in the early 1400's

Perspective Projection

- Project along rays that converge in center of projection



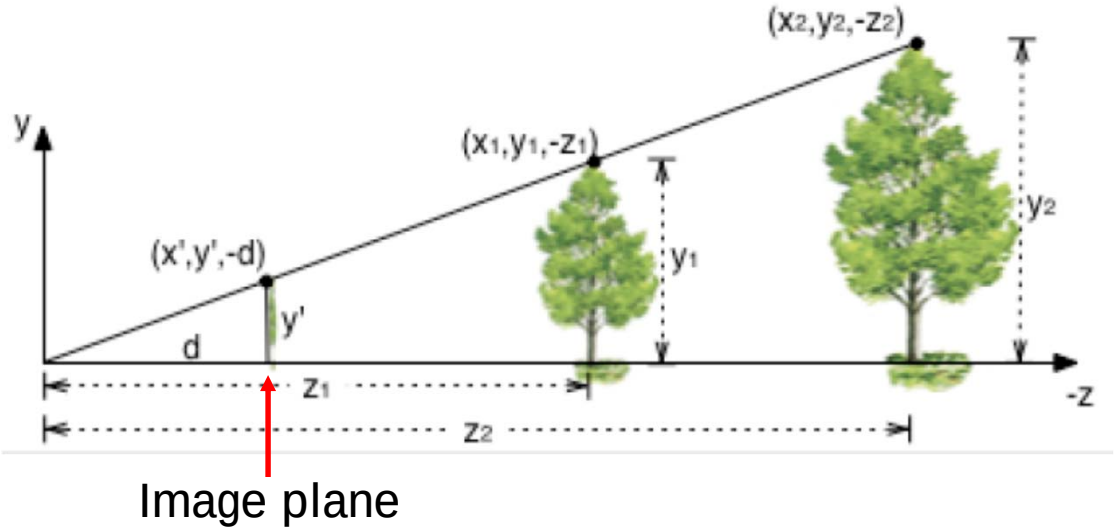
Perspective Projection

- From law of ratios in similar triangles follows:

$$\frac{y'}{d} = \frac{y_1}{z_1} \rightarrow y' = \frac{y_1 d}{z_1}$$

Similarly: $$x' = \frac{x_1 d}{z_1}$$

By definition: $$z' = d$$



- Perspective projection requires division

Homogeneous coordinates revisited

- 3D point using inhomogeneous coordinates as 3-vector

$$\tilde{\mathbf{X}} = \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \end{bmatrix}$$

- 3D point using affine homogeneous coordinates as 4-vector

$$\mathbf{X} = \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \\ 1 \end{bmatrix}$$

Homogeneous coordinates

- 3D point using *affine* homogeneous coordinates as 4-vector

$$\mathbf{X} = \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \\ 1 \end{bmatrix}$$

- 3D point using *projective* homogeneous coordinates as 4-vector (**up to scale**)

$$\mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

Homogeneous coordinates

- Projective homogeneous 3D point to affine homogeneous 3D point

$$\mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \frac{1}{W} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} \frac{X}{W} \\ \frac{Y}{W} \\ \frac{Z}{W} \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \\ 1 \end{bmatrix}$$

- Dehomogenize 3D point

$$\tilde{\mathbf{X}} = \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \end{bmatrix} = \begin{bmatrix} \frac{X}{W} \\ \frac{Y}{W} \\ \frac{Z}{W} \end{bmatrix}$$

Homogeneous coordinates

- Homogeneous points are defined up to a nonzero scale

$$\mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \lambda \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} \lambda X \\ \lambda Y \\ \lambda Z \\ \lambda W \end{bmatrix}$$

$$\tilde{\mathbf{X}} = \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \end{bmatrix} = \begin{bmatrix} \frac{\lambda X}{\lambda W} \\ \frac{\lambda Y}{\lambda W} \\ \frac{\lambda Z}{\lambda W} \end{bmatrix} = \begin{bmatrix} \frac{X}{W} \\ \frac{Y}{W} \\ \frac{Z}{W} \end{bmatrix}$$

Homogeneous coordinates

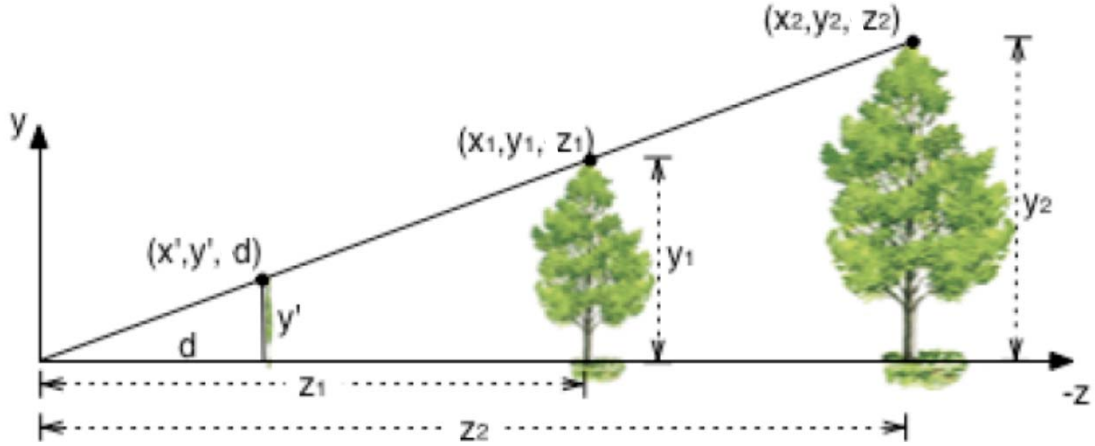
- When $W = 0$, then it is a point at infinity
- Affine homogeneous coordinates are projective homogeneous coordinates where $W = 1$
- When not differentiating between affine homogeneous coordinates and projective homogeneous coordinates, simply call them homogeneous coordinates

Perspective Projection

$$x' = \frac{x_1 d}{z_1}$$

$$y' = \frac{y_1 d}{z_1}$$

$$z' = d$$



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \rightarrow \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Projection matrix Homogeneous division

Perspective Projection

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

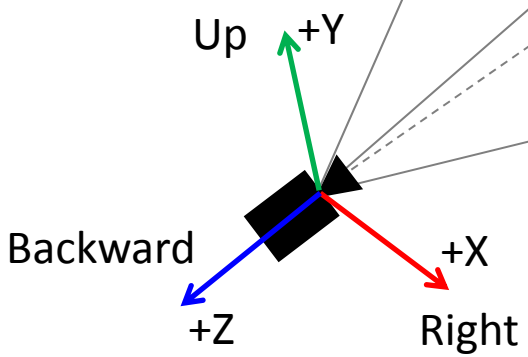
Projection matrix P

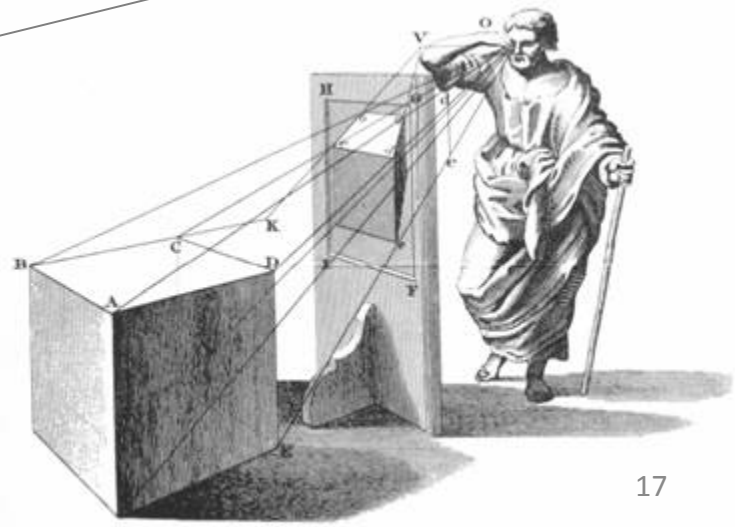
- Using projection matrix, homogeneous division seems more complicated than just multiplying all coordinates by d/z, so why do it?
- It will allow us to:
  - Handle different types of projections in a unified way
  - Define arbitrary view frustum

Camera (or eye) coordinate frame

All objects in pyramid of vision are potentially imaged by the camera or seen by the eye

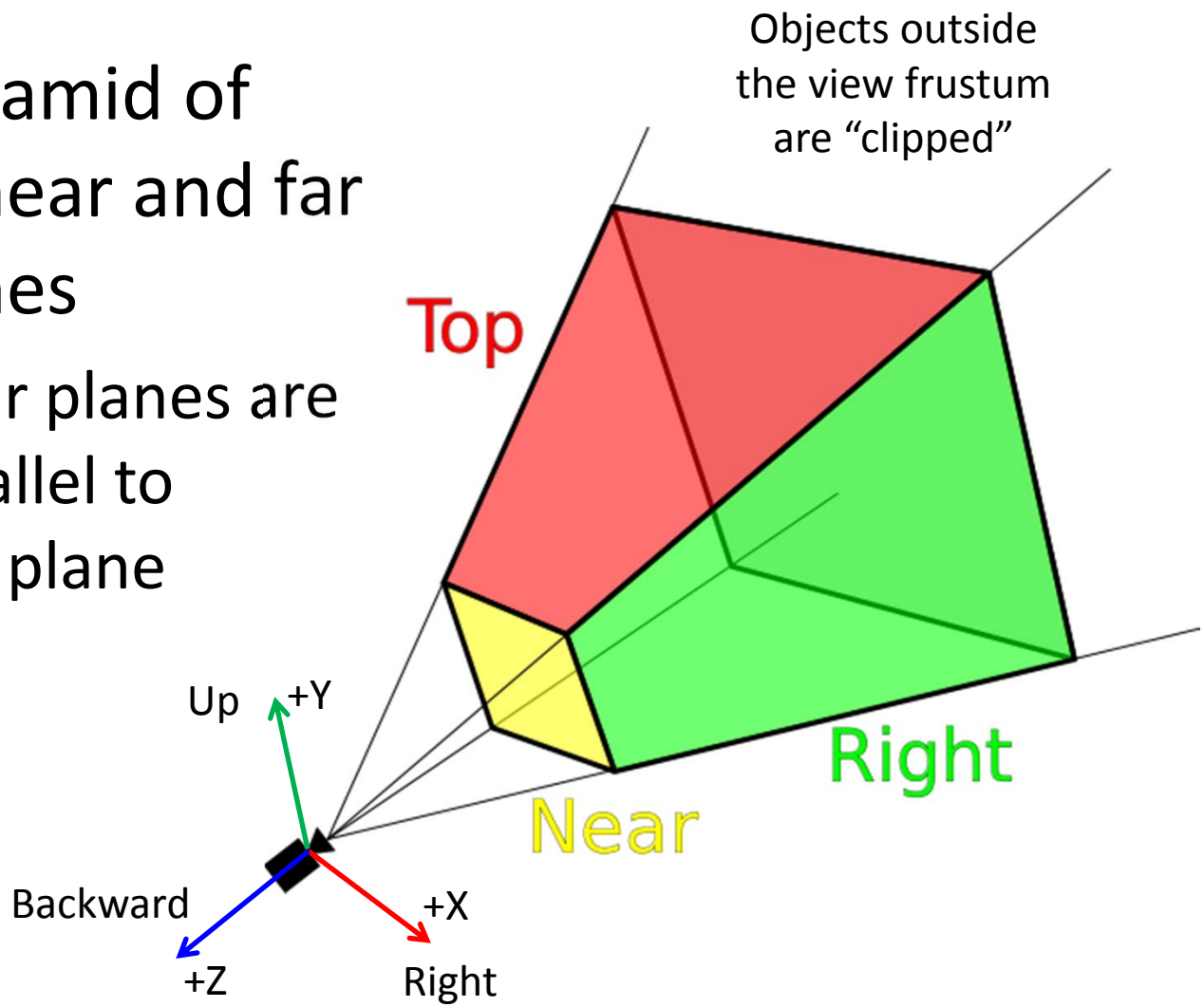
"Pyramid of vision"





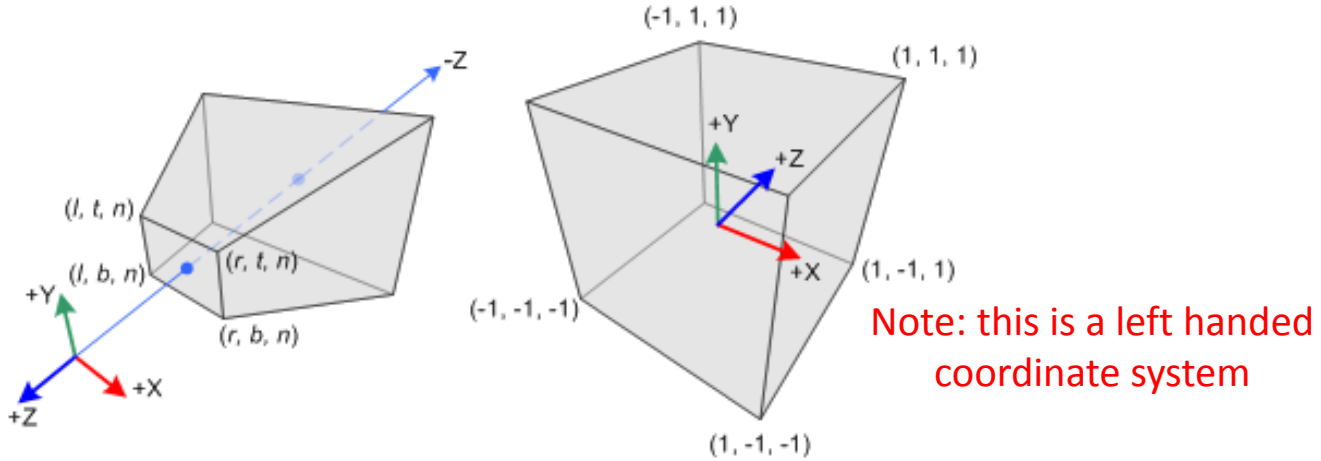
View frustum

- Truncate pyramid of vision with near and far clipping planes
  - Near and far planes are usually parallel to camera X-Y plane



Perspective projection transformation

- 3D projective transformation from the view frustum in the camera (or eye) coordinate frame to the canonical view volume in the normalized device coordinate frame



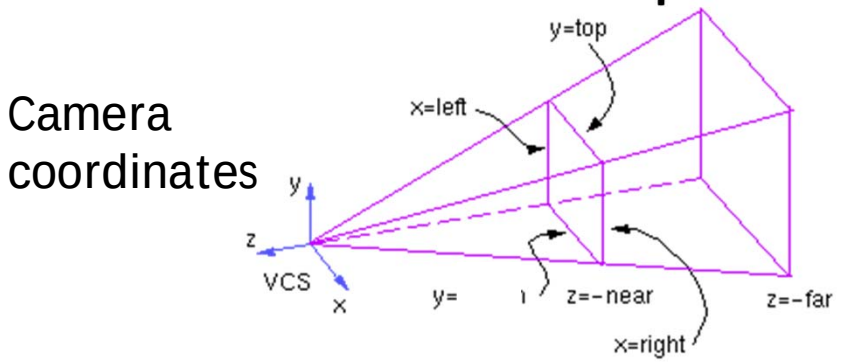
Note: this is a left handed coordinate system

View frustum in camera (or eye) coordinate frame

Canonical view volume in normalized device coordinate frame

Perspective projection transformation

- General view frustum with 6 parameters

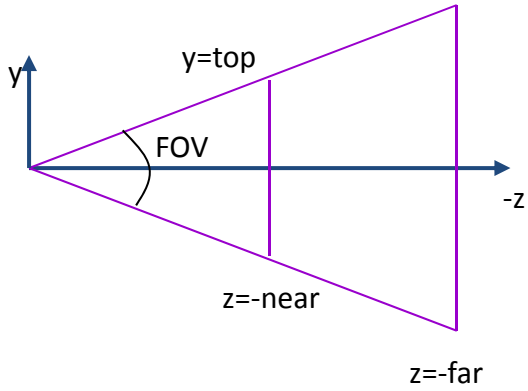


$$\mathbf{P}_{persp}(left, right, top, bottom, near, far) =$$

$$\begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective projection transformation

Symmetrical view volume



Side view of frustum

- Only 4 parameters
  - Vertical field of view (FOV)
  - Image aspect ratio (width/height)
  - Near, far clipping planes

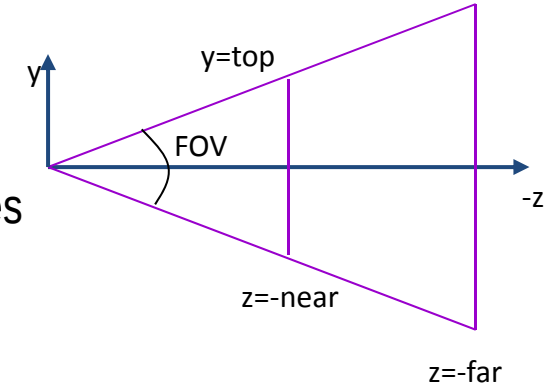
$$\text{aspect ratio} = \frac{right - left}{top - bottom} = \frac{right}{top}$$

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

Perspective projection transformation

- Symmetrical view frustum with field of view, aspect ratio, near and far clip planes

Camera coordinates



Side view of frustum

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

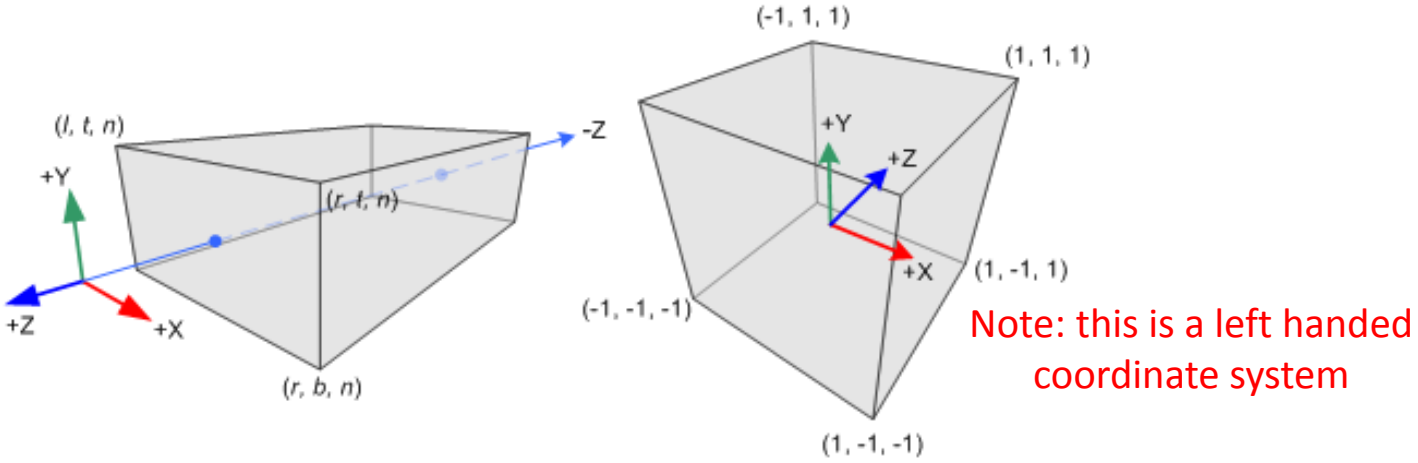
Perspective projection transformation

- Mapping of Z is nonlinear
  - Depth resolution in not uniform
- The previous projection transformations preserve depth on near and far planes
  - Very high precision at the near plane
  - Very low precision at the far plane
- The distance between near and far should be as small as possible to minimize precision issues
  - Do not set near = 0, far = infinity
- Setting near = 0 loses depth resolution

Orthographic projection transformation

- 3D projective transformation from the view frustum in the camera (or eye) coordinate frame to the canonical view volume in the normalized device coordinate frame

Parallel projection



Note: this is a left handed coordinate system

View frustum in camera (or eye) coordinate frame

Canonical view volume in normalized device coordinate frame

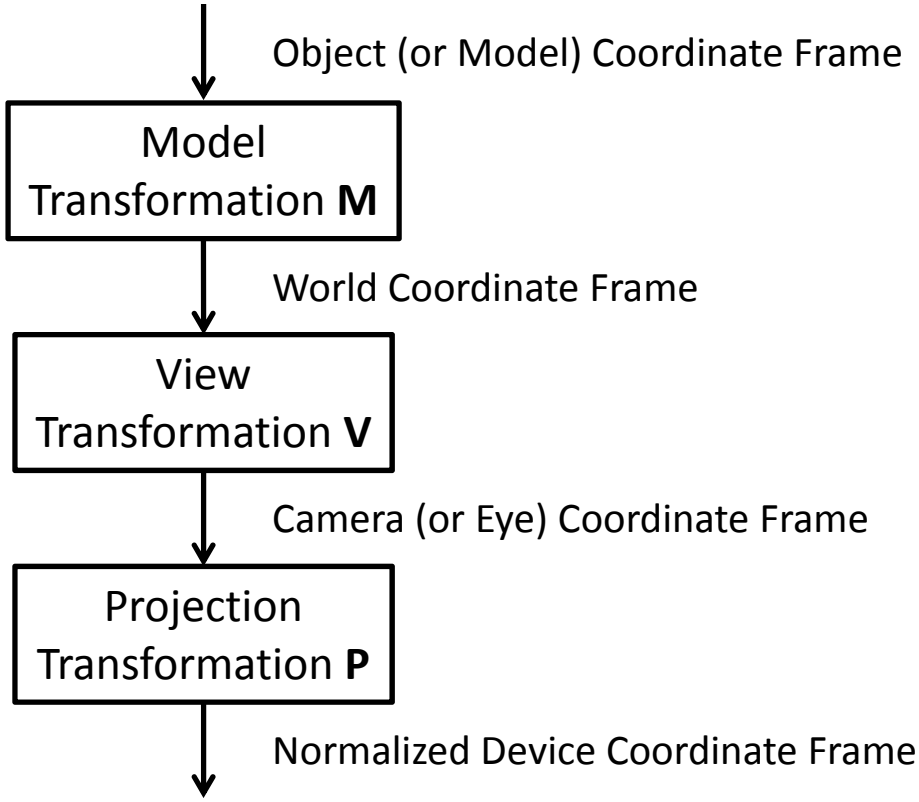
Orthographic projection transformation

- Orthographic projection transformation is much simpler than perspective projection transformation

$$\mathrm{P}_{\mathrm{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Used in many games

Transformations



$$\mathbf{X}_{\mathrm{nd}} = \mathtt{PVM}\mathbf{X}_{\mathrm{object}}$$

$$\mathbf{X}_{\mathrm{nd}} = \mathtt{H}_{\mathrm{object,nd}}\mathbf{X}_{\mathrm{object}}$$

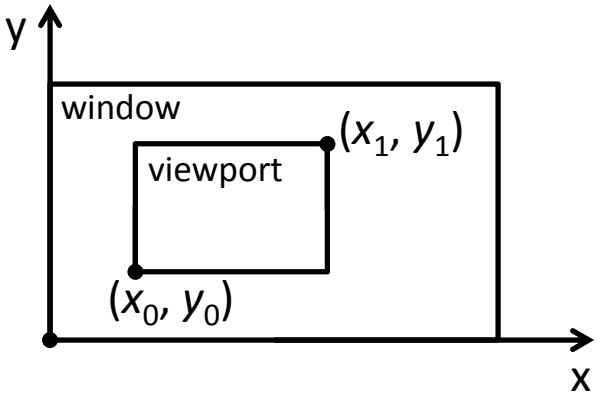
$$\text{where } \mathtt{H}_{\mathrm{object,nd}} = \mathtt{PVM}$$

Transform from object (or model) coordinate frame to normalized device coordinate frame using a 4x4 transformation *modelview projection* matrix

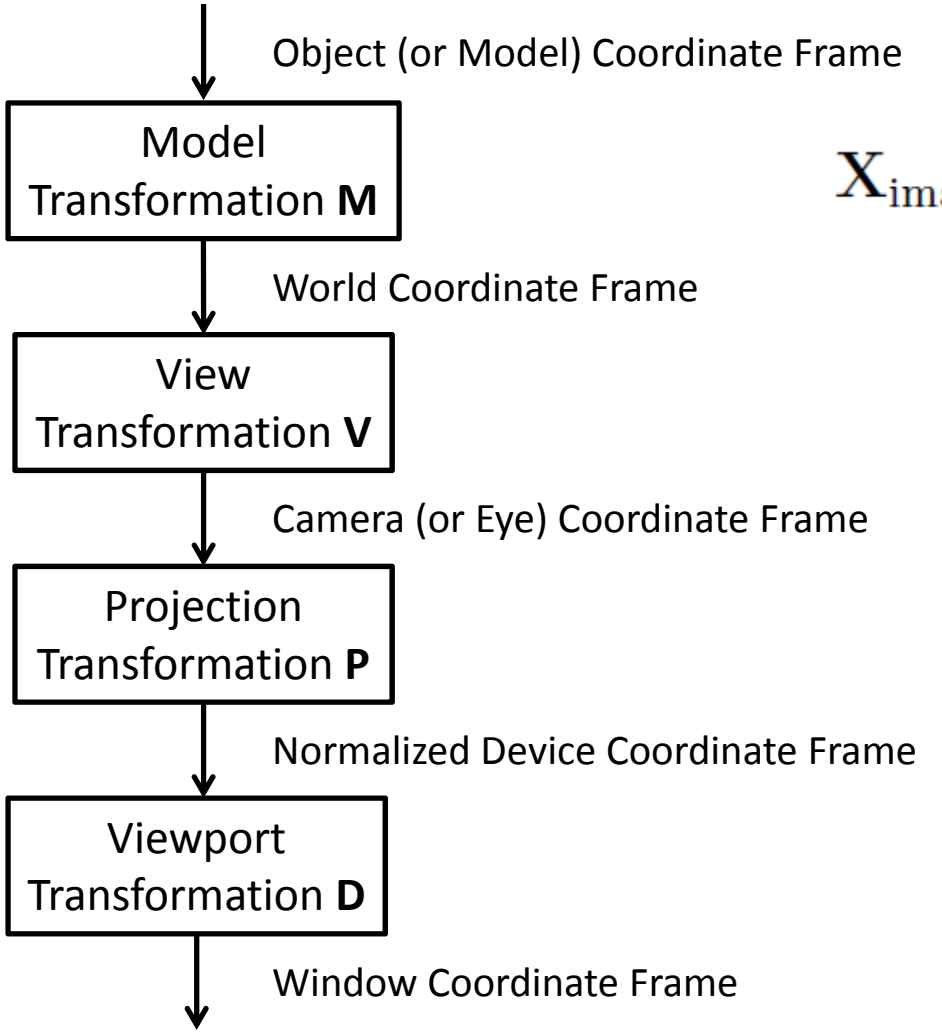
Viewport Transformation

- After applying projection matrix, scene points are in normalized device coordinates
  - Per definition within range [-1..1] x [-1..1] x [-1..1]
- Next is projection from 3D to 2D (not reversible)
- Normalized viewing coordinates can be mapped to image coordinates
  - Range depends on viewport
- Scale and translation required:

$$\mathbf{D}(x_0, x_1, y_0, y_1) = \begin{bmatrix} (x_1 - x_0)/2 & 0 & 0 & (x_0 + x_1)/2 \\ 0 & (y_1 - y_0)/2 & 0 & (y_0 + y_1)/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



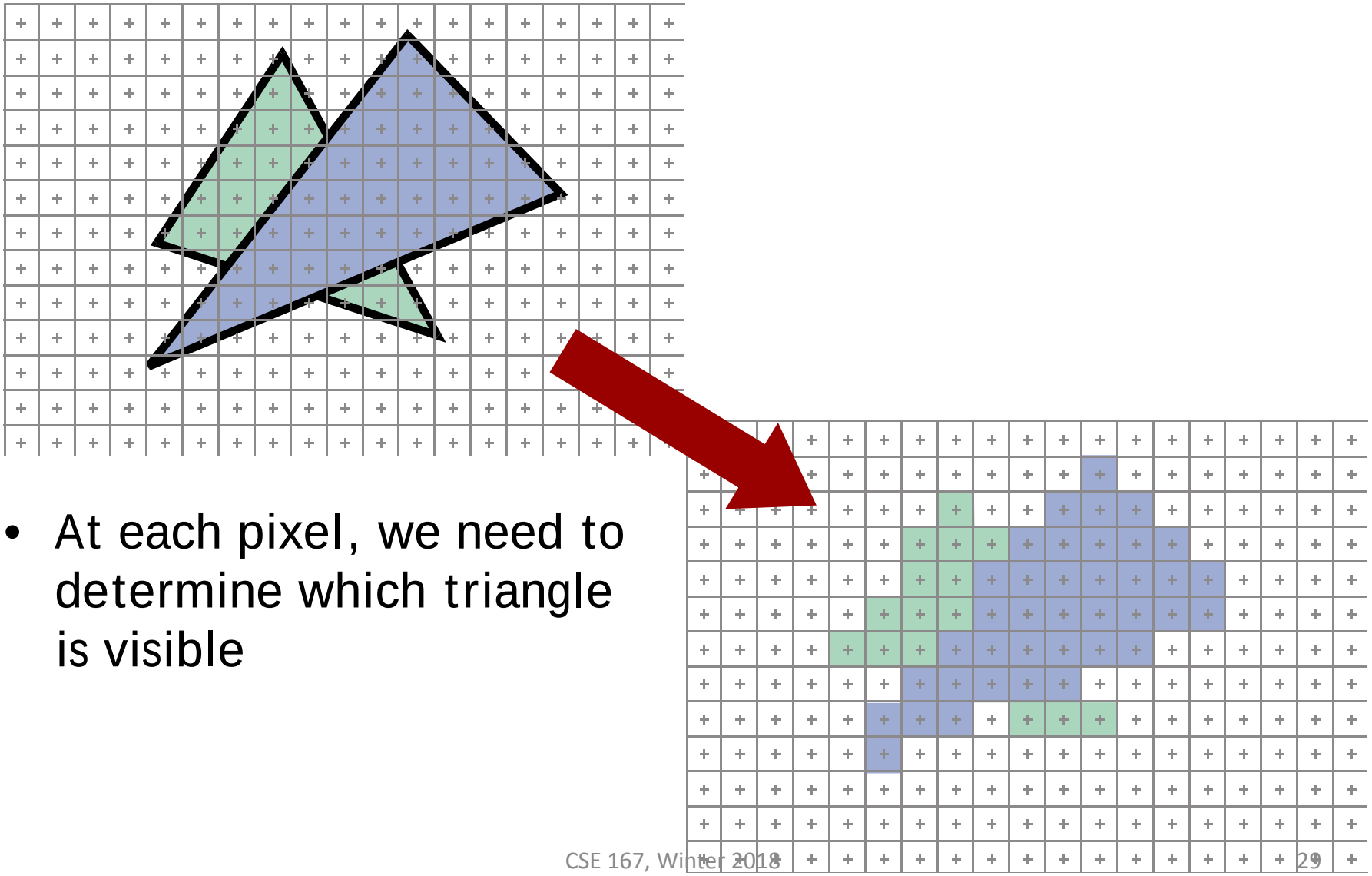
Transformations



$$\mathbf{X}_{\text{image}} = \mathbf{DPVMX}_{\text{object}}$$

- Object (or Model) coordinates
- World coordinates
- Camera (or Eye) coordinates
- Normalized device coordinates
- Window coordinates

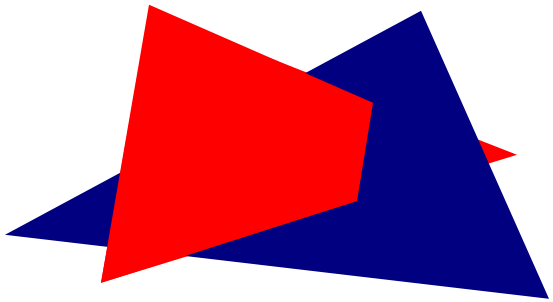
Visibility



- At each pixel, we need to determine which triangle is visible

Painter's Algorithm

- Paint from back to front
- Need to sort geometry according to depth
- Every new pixel always paints over previous pixel in frame buffer
- May need to split triangles if they intersect



- Intuitive, but outdated algorithm - created when memory was expensive
- Needed for translucent geometry even today

Z-Buffering

- Store z-value for each pixel
- Depth test
  - Initialize z-buffer with farthest z value
  - During rasterization, compare stored value to new value
  - Update pixel only if new value is smaller

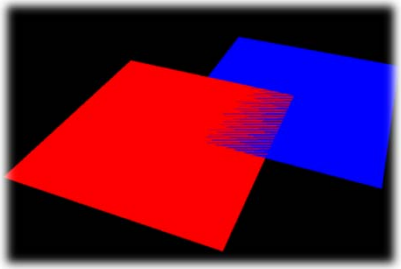
```
setpixel(int x, int y, color c, float z)
if(z<zbuffer(x,y)) then
  zbuffer(x,y) = z
  color(x,y) = c
```

- z-buffer is dedicated memory reserved in GPU memory
- Depth test is performed by GPU → very fast

Z-Buffering in OpenGL

- In OpenGL applications:
  - Ask for a depth buffer when you create your GLFW window
    - glfwOpenWindow(512, 512, 8, 8, 8, 0, **16**, 0, GLFW_WINDOW)
  - Place a call to glEnable(GL_DEPTH_TEST) in your program's initialization routine
  - Ensure that your *zNear* and *zFar* clipping planes are set correctly (glm::perspective(fovy, aspect, zNear, zFar)) and in a way that provides adequate depth buffer precision
  - Pass GL_DEPTH_BUFFER_BIT as a parameter to glClear

Z-Buffer Fighting



- Problem: polygons which are close together do not get rendered correctly
  - Errors change with camera perspective → flicker
- Cause: different colored fragments from different polygons are being rasterized to same pixel and depth → not clear which is in front of which
- Solutions:
  - Move surfaces farther apart, so that fragments rasterize into different depth bins
  - Bring near and far planes closer together
  - Use a higher precision depth buffer
    - Note that OpenGL often defaults to 16 bit even if your graphics card supports 24 bit or 32 bit depth buffers

Translucent Geometry

- Need to depth sort translucent geometry and render with Painter's Algorithm (back to front)
- Problem: incorrect blending with cyclically overlapping geometry

- Solutions:
  - Back to front rendering of translucent geometry (Painter's Algorithm), after rendering opaque geometry
    - Does not always work correctly: programmer has to weigh rendering correctness against computational effort
  - Theoretically: need to store multiple depth and color values per pixel (not practical in real-time graphics)