CS464 Oct 3rd 2017

Assignment 3 – ~~Due 10/6/2017~~
Due 10/8/2017
Implementation Outline

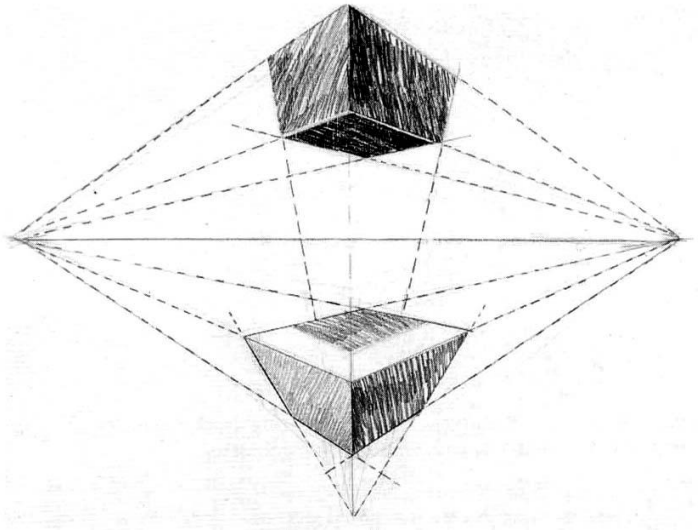
Assignment 3 Skeleton

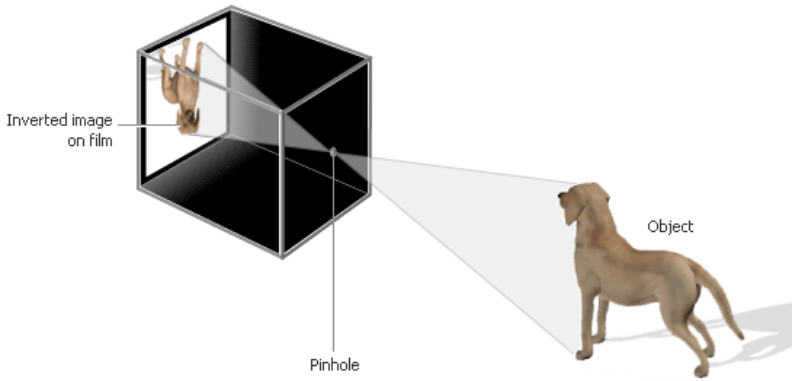
A good sequence to implement the program

1. Start with a flat terrain sitting at Y=0 and Cam at Y=0.5
2. Add in code to compute a View Matrix
3. Add in Mouse Code to handle looking around.
4. Add in Mouse Code to let you drive around.
5. Adjust the terrain to vary Y but keep values between 0 and 0.3 – so no big leaps.
6. Add code to extract Height from Texture object
7. Adjust Cam height with height from texture object.
8. Adjust Cam view angle to vary with terrain.
9. Adjust Cam up angle to vary with terrain.

Could we use Vertex Shader

Ideas for how we might use the vertex shader?





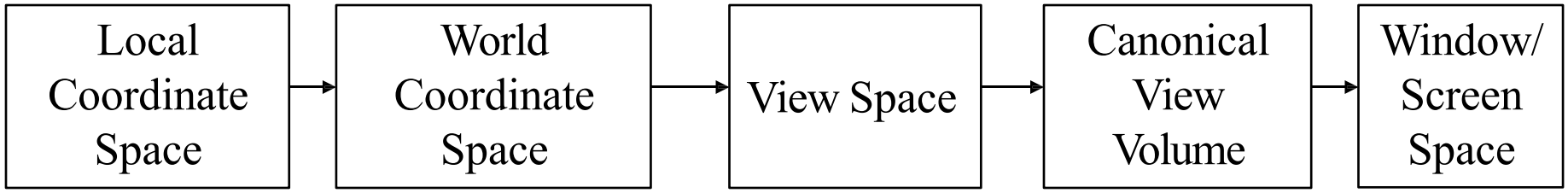
Viewing

COMPSCI 464

Image Credits: Encarta and http://www.sackville.ednet.ns.ca/art/grade10/drawing/perspective4.html

Graphics Pipeline

- Graphics hardware employs a sequence of coordinate systems
  - The location of the geometry is expressed in each coordinate system in turn, and modified along the way
  - The movement of geometry through these spaces is considered a pipeline



Standard Sequence of Transforms

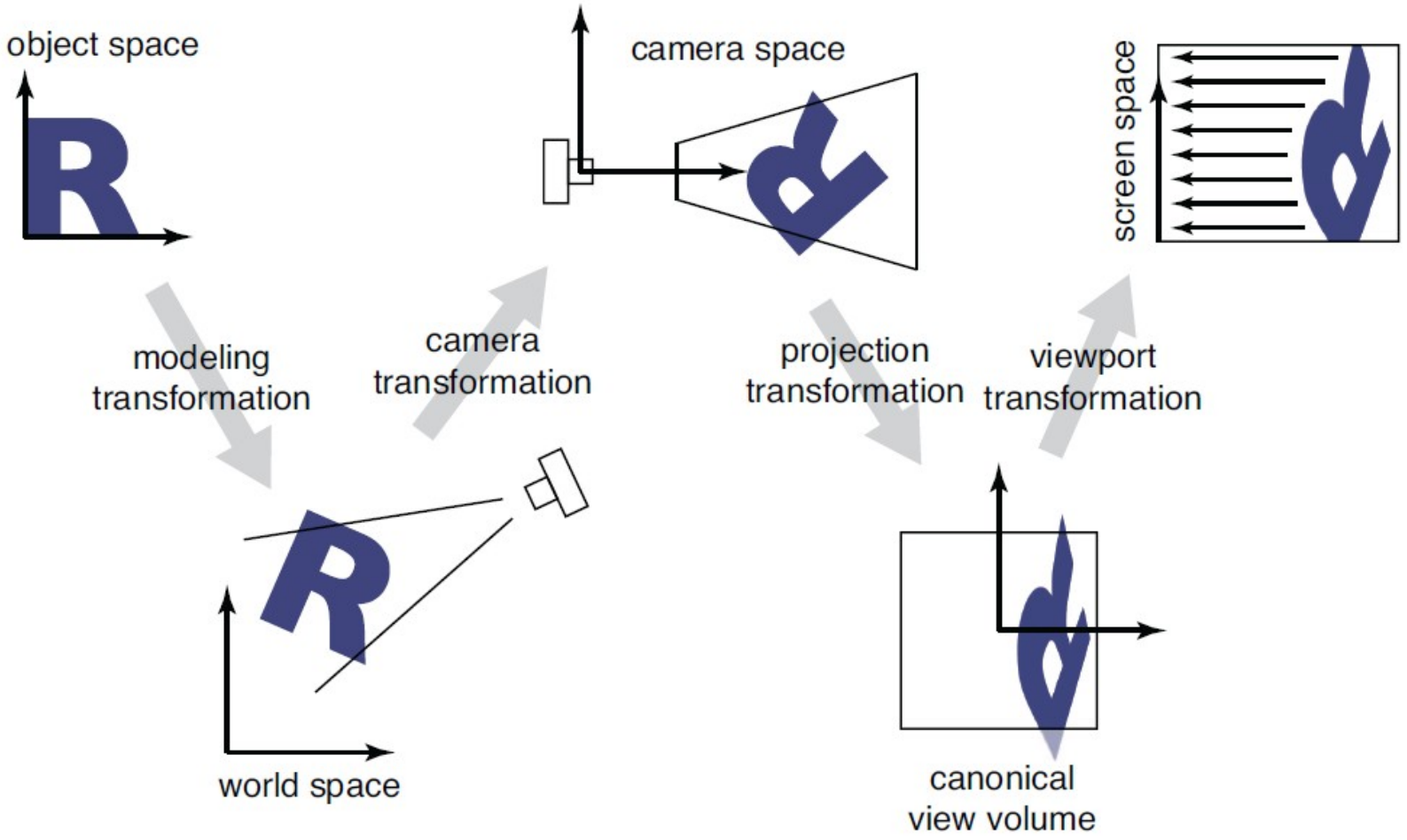
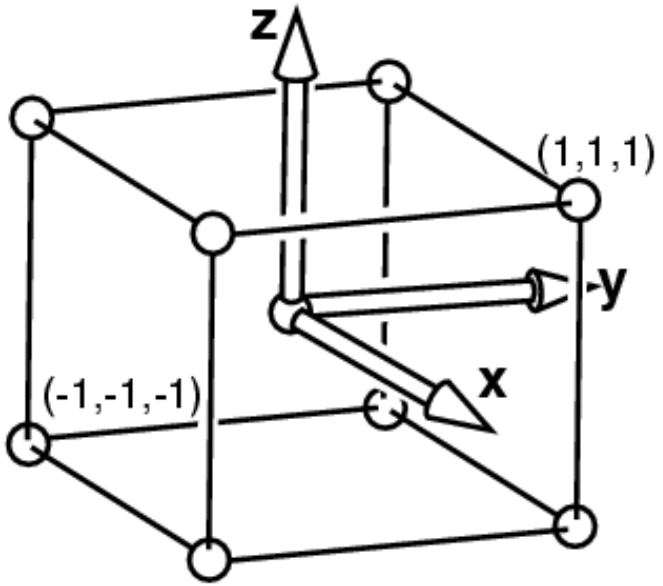


Image credits: http://www.cs.cornell.edu/courses/cs4620/2009fa/lectures/10viewing.pdf

Canonical View Volume

- CanonicalView Space: A cube, with the origin at the center, the viewer looking down –z, x to the right, and y up
  - CanonicalViewVolume is the cube: [-1,1]×[-1,1]×[-1,1]
  - Variants (later) with viewer looking down +z and z from 0-1
  - Only things that end up inside the canonical volume can appear in the window



Canonical View Volume

- Tasks: Parallel sides and unit dimensions make many operations easier
  - Clipping – decide what is in the window
  - Rasterization - decide which pixels are covered
  - Hidden surface removal - decide what is in front
  - Shading - decide what color things are

Window/Screen Space

- **Window Space:** Origin in one corner of the "window" on the screen, x and y match screen x and y

- Windows appear somewhere on the screen
  - Typically you want the thing you are drawing to appear in your window
  - But you may have no control over where the window appears

- You want to be able to work in a standard coordinate system – **your code should not depend on *where* the window is**

- You target Window Space, and the windowing system takes care of putting it on the screen

Canonical → Window Transform

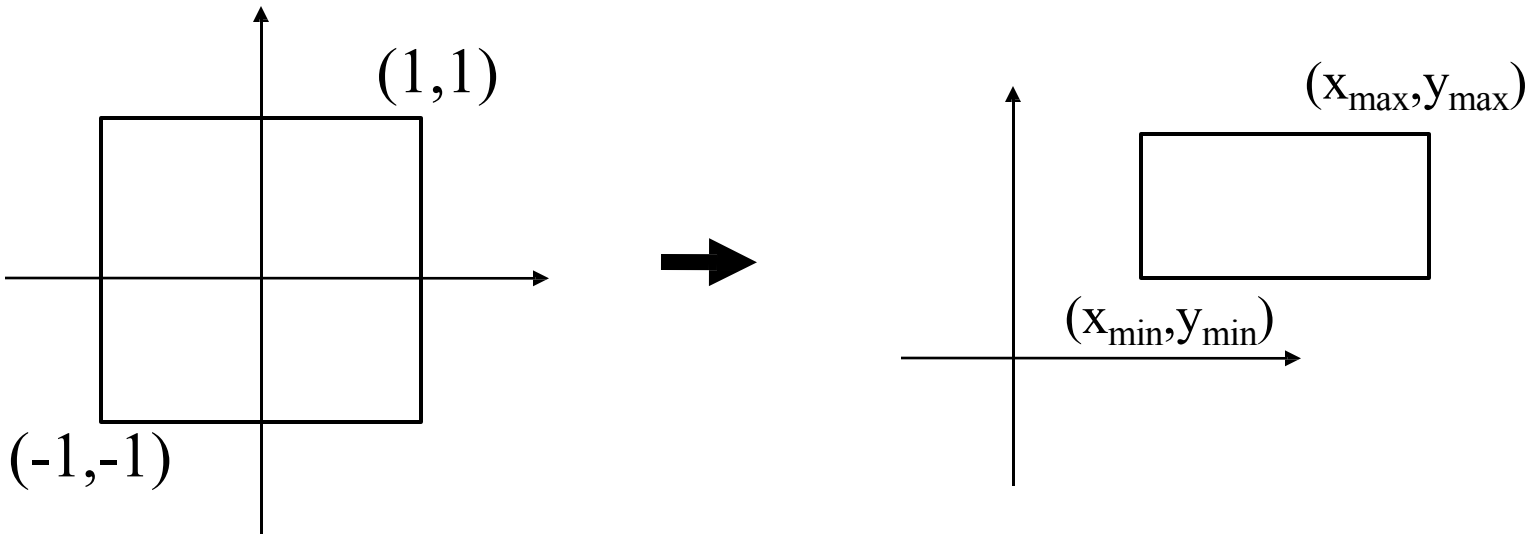
- Problem: Transform the CanonicalViewVolume into Window Space (real screen coordinates)
  - Drop the depth coordinate and translate
  - The graphics hardware and windowing system typically take care of this – but we'll do the math to get you warmed up

- The windowing system adds one final transformation to get your window on the screen in the right place
  - glutInitWindowPosition(50,50);

Canonical → Window Transform

- Typically, windows are specified by a corner, width and height
  - Corner expressed in terms of screen location
  - This representation can be converted to (x_{min}, y_{min}) and (x_{max}, y_{max})
- We want to map points in Canonical View Space into the window
  - Canonical View Space goes from (-1,-1,-1) to (1,1,1)
  - Lets say we want to leave z unchanged
- What basic transformations will be involved in the total transformation from 3D screen to window coordinates?

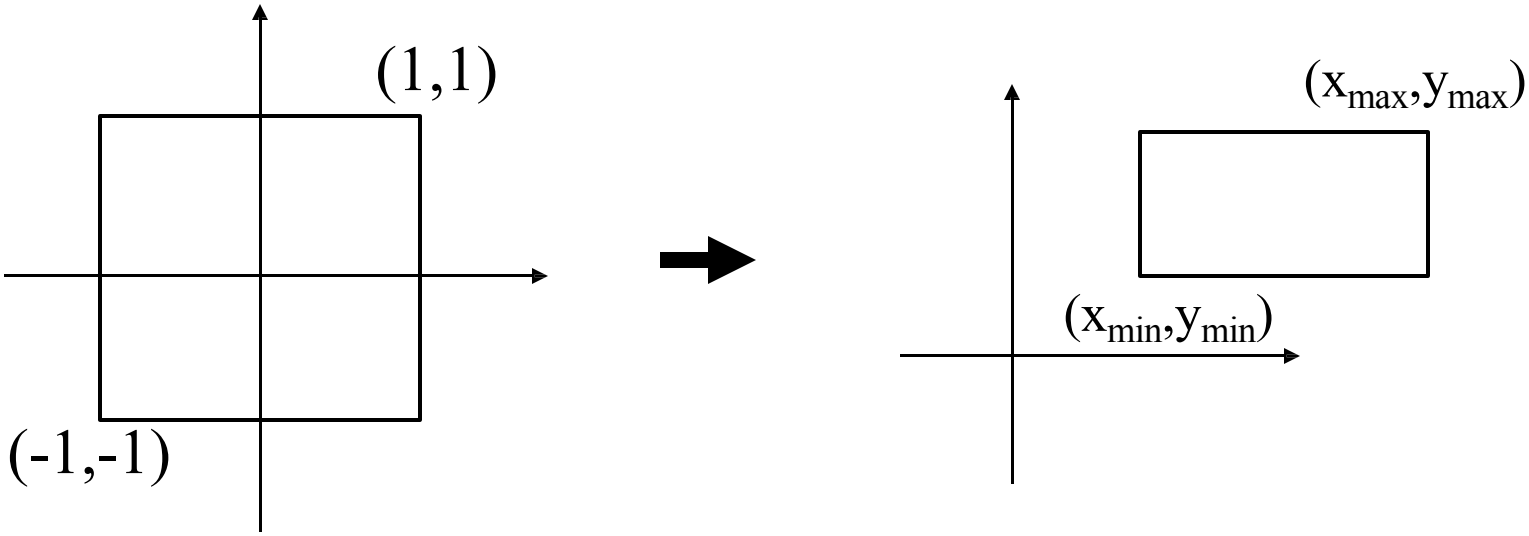
Canonical → Window Transform



Canonical View Space

Window Space

Canonical → Window Transform



$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ z_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} (x_{\max} - x_{\min})/2 & 0 & 0 & (x_{\max} + x_{\min})/2 \\ 0 & (y_{\max} - y_{\min})/2 & 0 & (y_{\max} + y_{\min})/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{bmatrix}$$

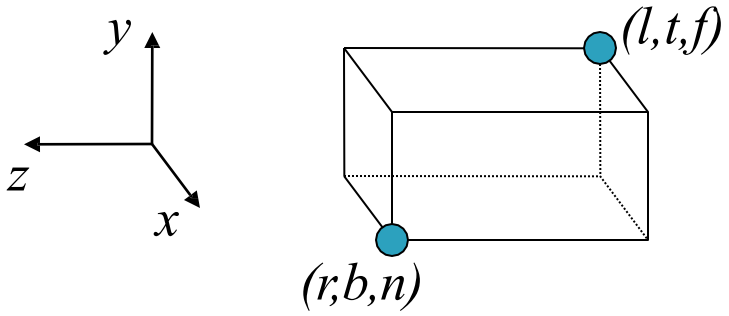
Canonical → Window Transform

- You almost never have to worry about the canonical to window transform

- In WebGL, you tell it which part of **your window** to draw in – relative to the window's coordinates
  - That is, you tell it where to put the canonical view volume
  - You must do this whenever the window changes size
  - Window (not the screen) has **origin at bottom left**
  - `gl.viewport(minx, miny, maxx, maxy)`
  - Typically: `gl.viewport(0, 0, width, height)` fills the entire *window* with the image
  - Why might you *not* fill the entire window?

Orthographic View Space

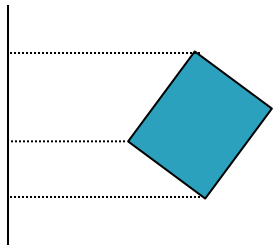
- **View Space:** a coordinate system with the viewer looking in the –z direction, with *x* horizontal to the right and *y* up
  - A right-handed coordinate system!
- The view volume is a *rectilinear box* for orthographic projection
- The view volume has:
  - a *near plane* at *z=n*
  - a *far plane* at *z=f , (f < n)*
  - a *left plane* at *x=l*
  - a *right plane* at *x=r,(r>l)*
  - a *top plane* at *y=t*
  - and a *bottom plane* at *y=b,(b<t)*

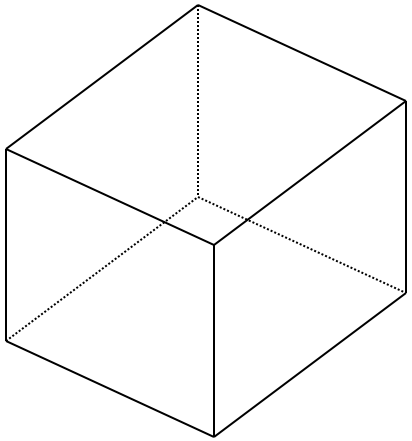


Rendering the Volume

- To find out where points end up on the screen, we must transform View Space into Canonical View Space
  - We know how to draw Canonical View Space on the screen
- This transformation is "projection"
- The mapping looks similar to the one for Canonical to Window …

Orthographic Projection

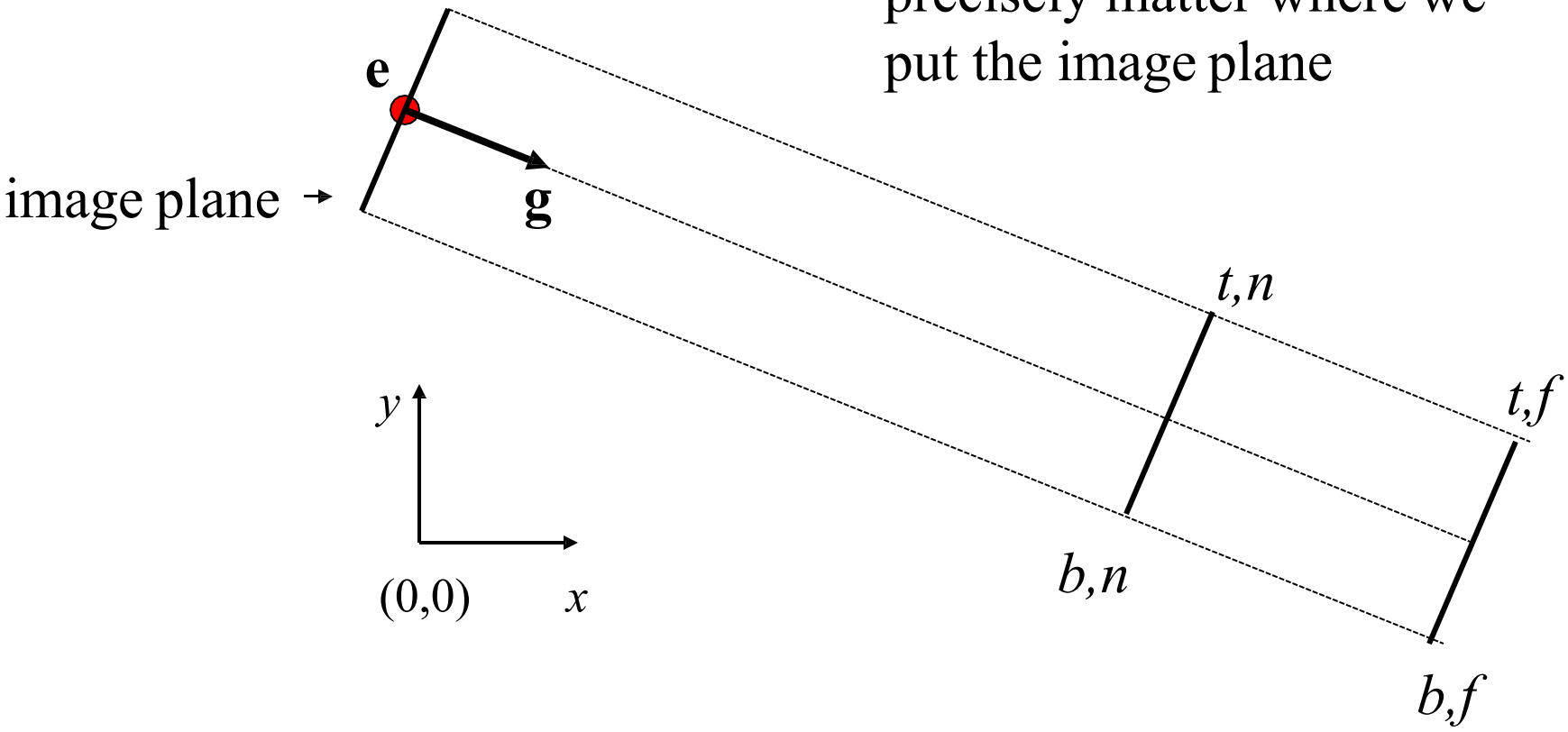




- Orthographic projection projects all the points in the world along parallel lines onto the image plane
  - Projection lines are perpendicular to the image plane
  - Like a camera with infinite focal length
- The result is that *parallel lines in the world project to parallel lines in the image*, and *ratios of lengths are preserved*
  - This is important in some applications, like medical imaging and some computer aided design tasks

General Orthographic

Subtle point: it doesn't precisely matter where we put the image plane



Orthographic Projection Matrix
(Orthographic View to Canonical Matrix)

$$\begin{bmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{bmatrix} = \begin{bmatrix} 2/(r-l) & 0 & 0 & 0 \\ 0 & 2/(t-b) & 0 & 0 \\ 0 & 0 & 2/(n-f) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -(r+l)/2 \\ 0 & 1 & 0 & -(t+b)/2 \\ 0 & 0 & 1 & -(n+f)/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{view} \\ y_{view} \\ z_{view} \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2/(r-l) & 0 & 0 & -(r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & -(t+b)/(t-b) \\ 0 & 0 & 2/(n-f) & -(n+f)/(n-f) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{view} \\ y_{view} \\ z_{view} \\ 1 \end{bmatrix}$$

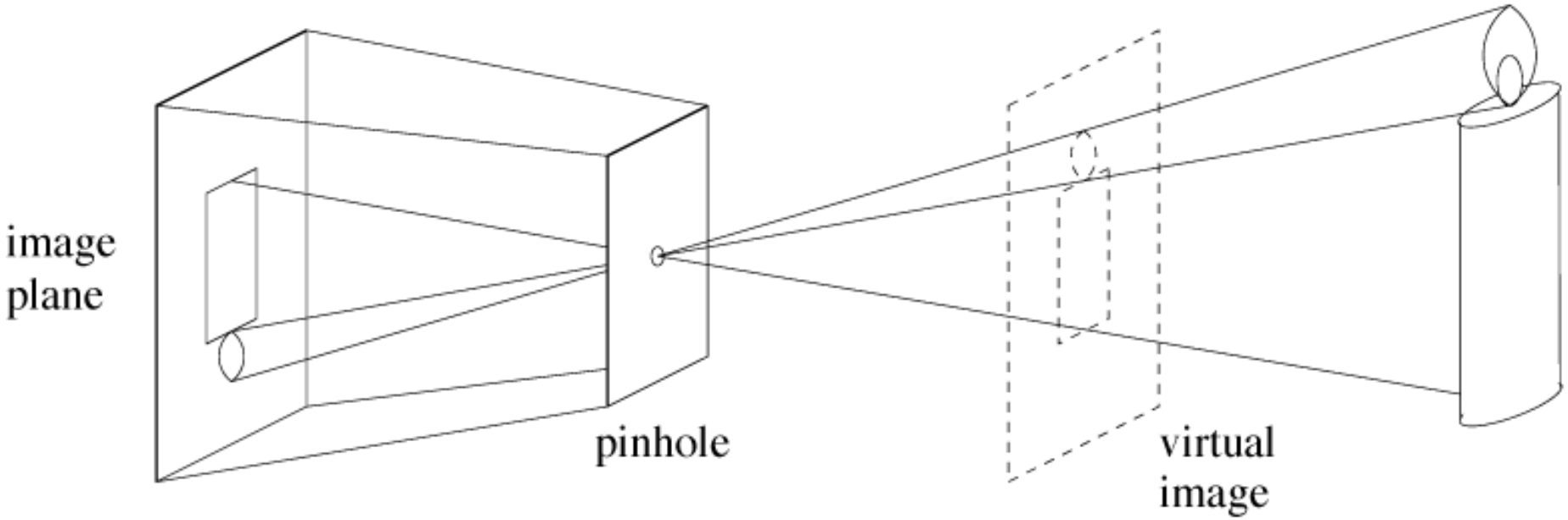
$$\mathbf{x}_{canonical} = \mathbf{M}_{view->canonical}\mathbf{x}_{view}$$

View Volumes

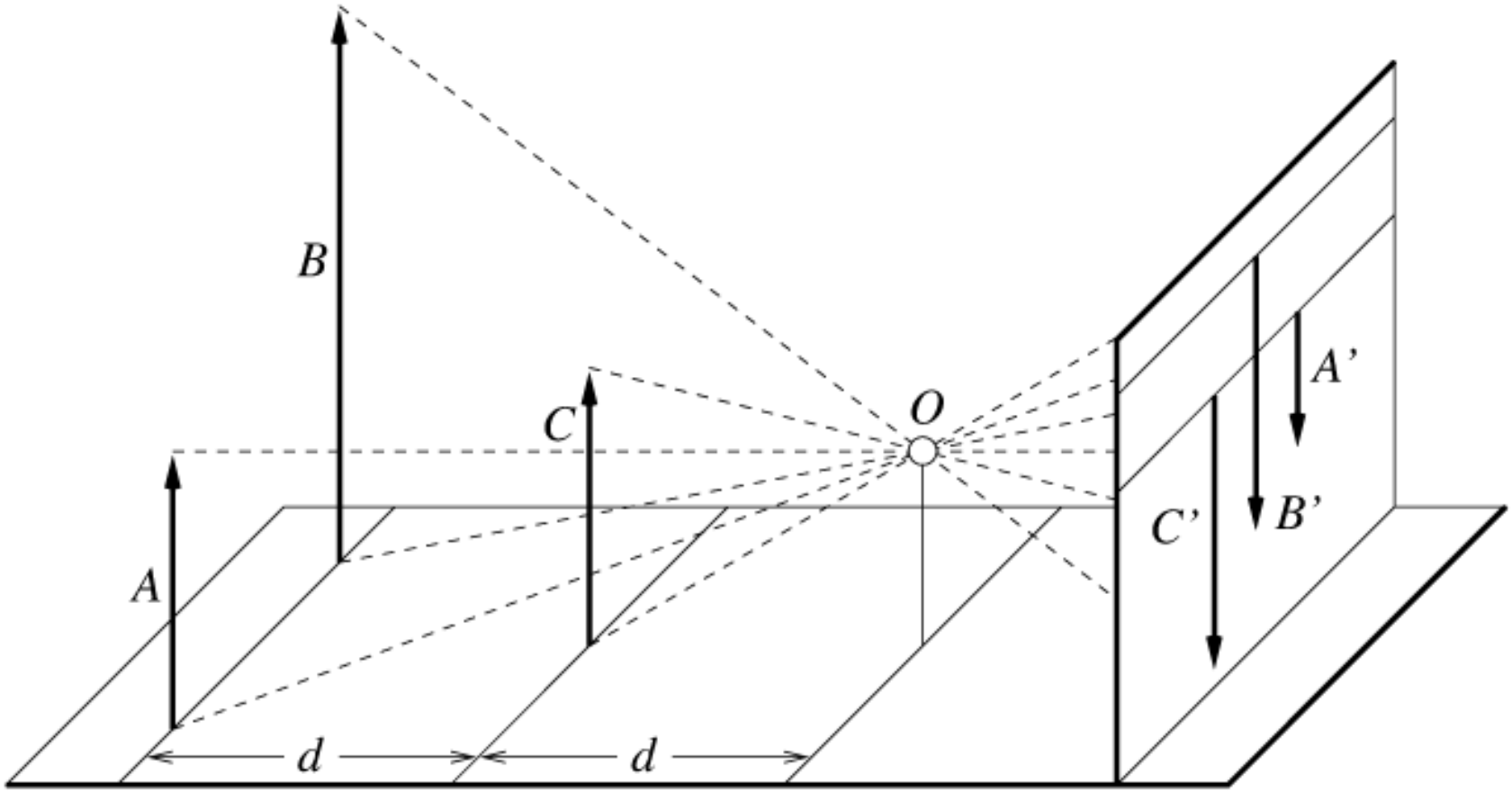
- Only stuff inside the CanonicalViewVolume gets drawn
  - The window is of finite size, and we can only store a finite number of pixels
  - We can only store a discrete, finite range of depths
    - Like color, only have a fixed number of bits at each pixel
  - Points too close or too far away will not be drawn
  - But, it is inconvenient to model the world as a unit box
- A **view volume** is the region of space we wish to *transform into* the CanonicalViewVolume for drawing
  - Only stuff inside the view volume gets drawn
  - **Describing the view volume is a major part of defining the view**

Perspective Projection

- Abstract camera model - box with a small hole in it
- Pinhole cameras work in practice



Distant Objects Are Smaller

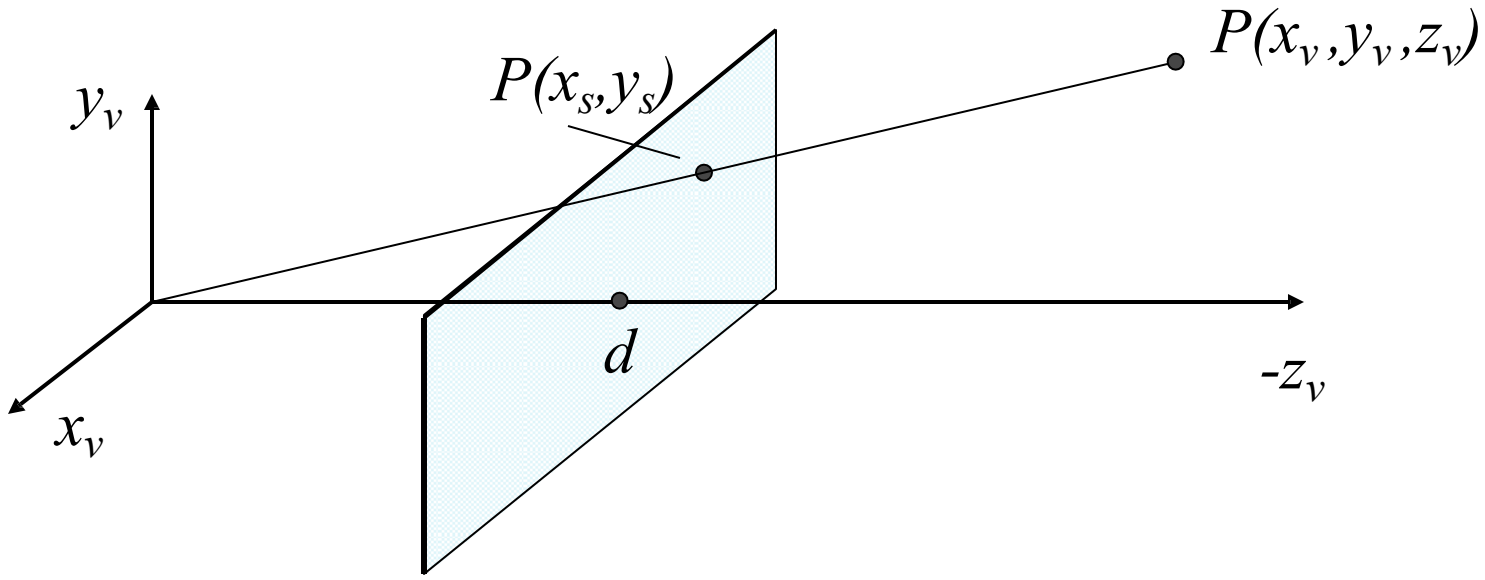


Basic Perspective Projection

- We are going to temporarily ignore canonical view space, and go straight from view to window
- Assume you have transformed to view space, with *x* to the right, *y* up, and *z* back toward the viewer
- Assume the origin of view space is at the center of projection (the eye)
- Define a focal distance, *d*, and put the image plane there (note *d* is negative)
  - You can define *d* to control the size of the image

Basic Perspective Projection

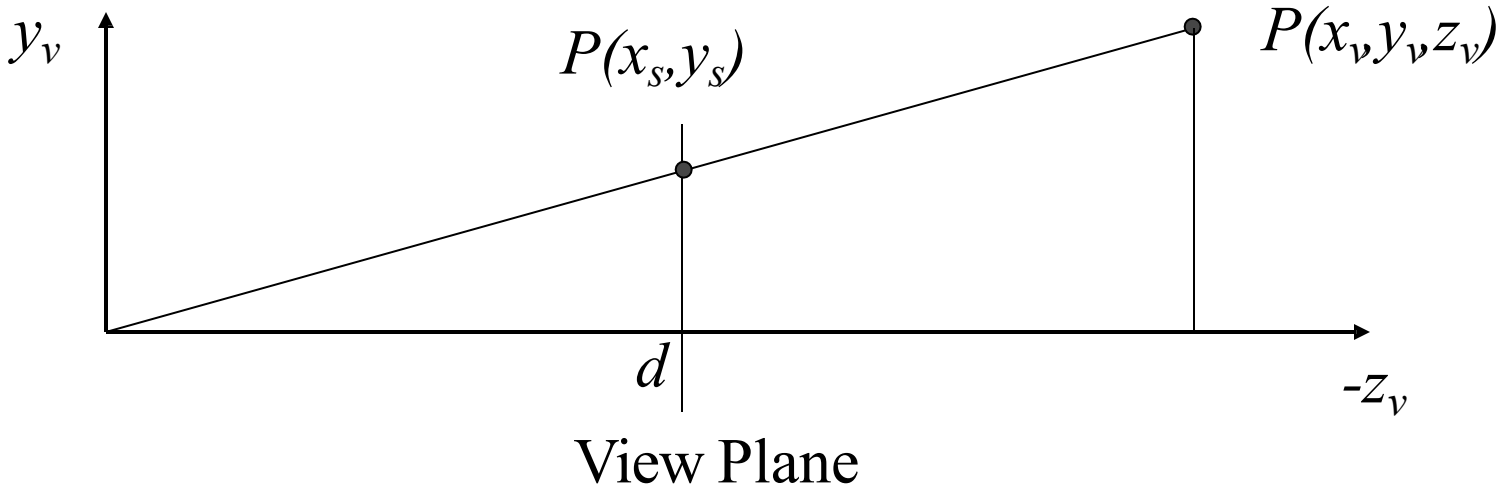
- If you know $P(x_v, y_v, z_v)$ and d, what is $P(x_s, y_s)$?
  - Where does a point in view space end up on the screen?



Basic Case

- Similar triangles gives:

$$\frac{x_s}{d} = \frac{x_v}{z_v} \qquad \frac{y_s}{d} = \frac{y_v}{z_v}$$



Simple Perspective Transformation

- Using homogeneous coordinates we can write:
  - Our next big advantage to homogeneous coordinates

$$\begin{bmatrix} x_s \\ y_s \\ d \end{bmatrix} \equiv \begin{bmatrix} x_v \\ y_v \\ z_v \\ z_v / d \end{bmatrix}$$

$$\mathbf{P}_s = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \mathbf{P}_v$$

General Perspective

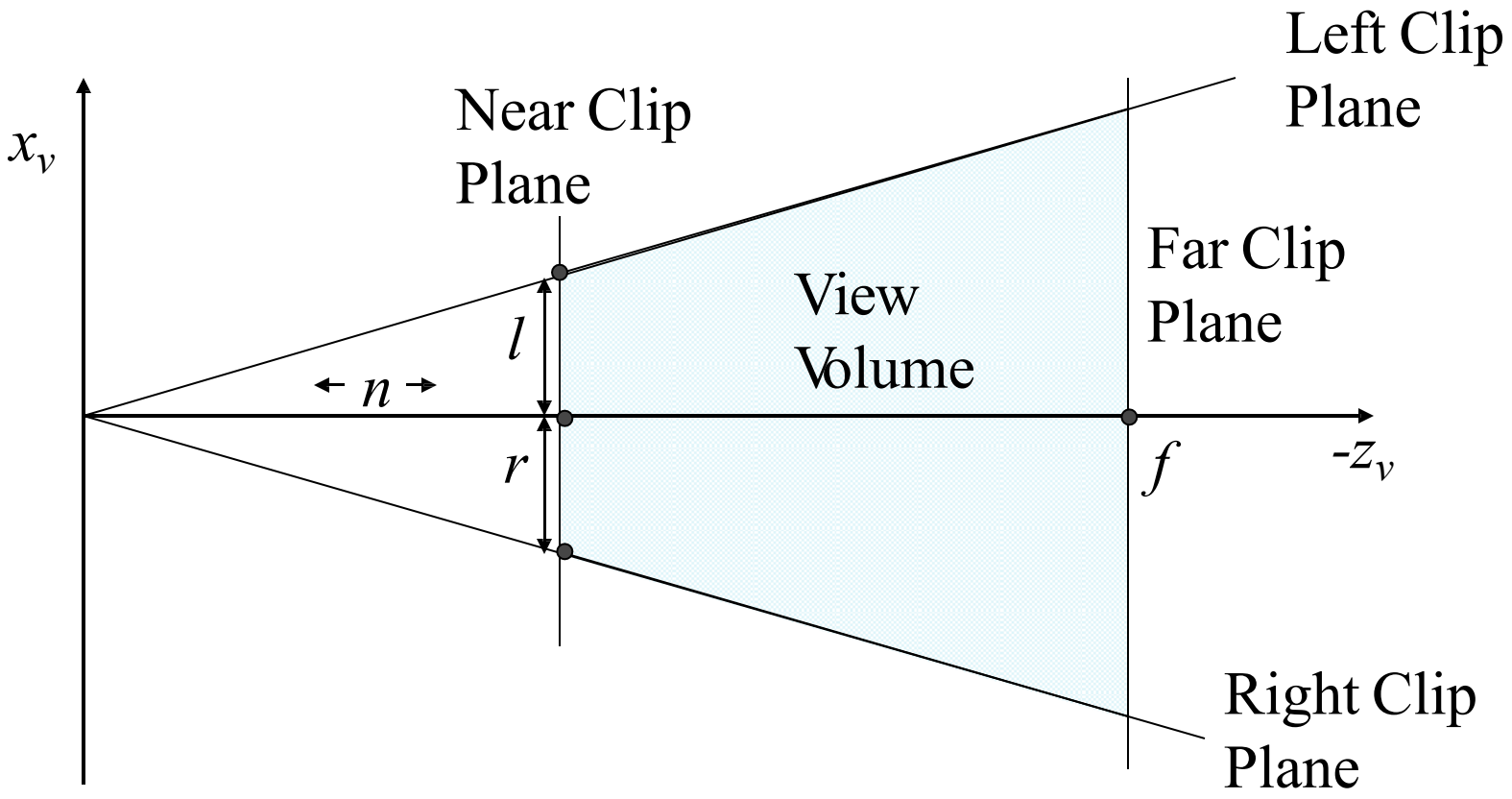
- The basic equations we have seen give a flavor of what happens, but they are insufficient for all applications
- They do not get us to a CanonicalViewVolume
- They make assumptions about the viewing conditions
- To get to a Canonical Volume, we need a Perspective Volume …

Perspective View Volume

- Recall the orthographic view volume, defined by a near, far, left, right, top and bottom plane

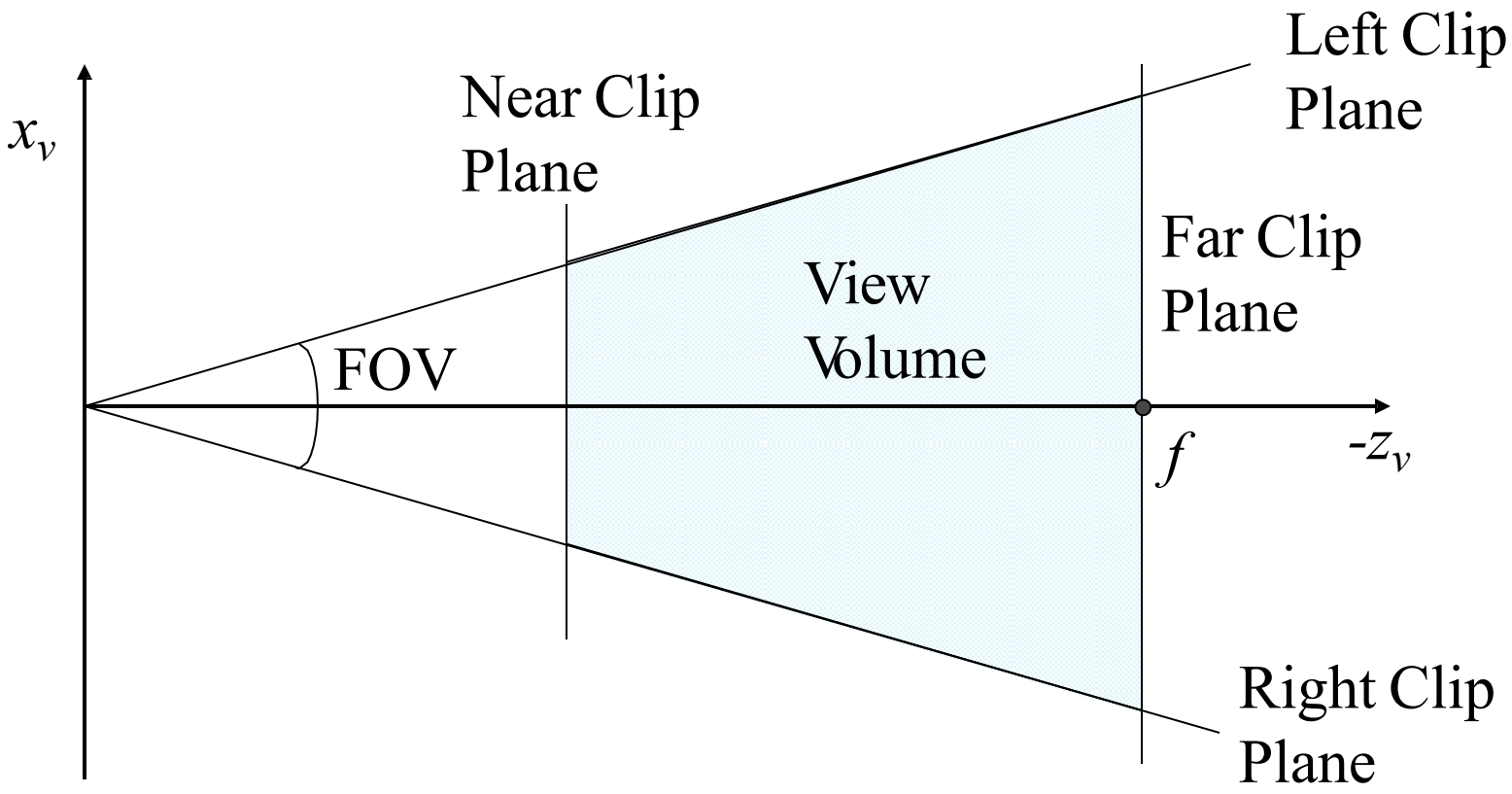
- The perspective view volume is also defined by near, far, left, right, top and bottom planes – the *clip planes*
  - **Near and far planes are parallel to the image plane**: $z_v=n, z_v=f$
  - Other planes all pass through the center of projection (the origin of view space)
  - The left and right planes intersect the image plane in vertical lines
  - The top and bottom planes intersect in horizontal lines

Clipping Planes



Field of View

- Assumes a *symmetric* view volume

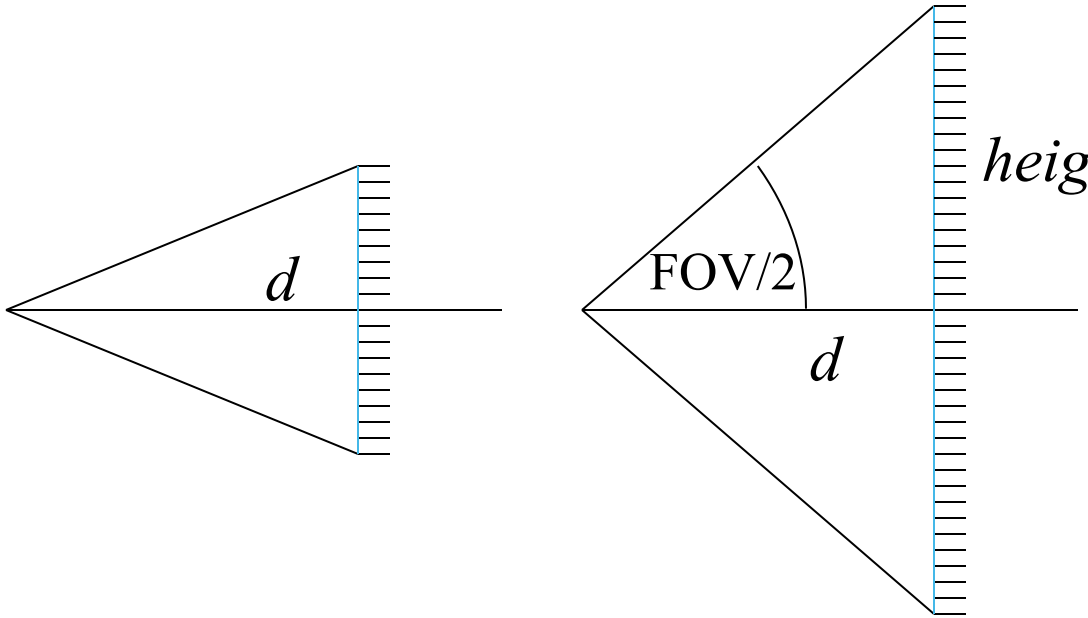


Perspective Parameters

- We have seen several different ways to describe a perspective camera
  - Focal distance, Field of View, Clipping planes
- The most general is clipping planes – they directly describe the region of space you are viewing
- For most graphics applications, field of view is the most convenient
  - It is *image size invariant* – having specified the field of view, what you see does not depend on the image size
- You can convert one thing to another

Focal Distance to FOV

- You must have the image size to do this conversion
  - Why? Same *d*, different image size,different FOV

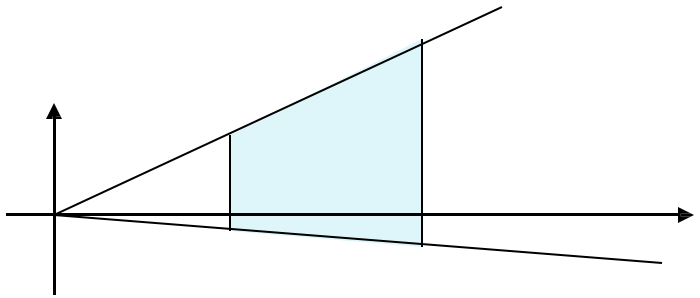


$$\tan\left(\frac{FOV}{2}\right) = \frac{height}{2d}$$

$$FOV = 2\tan^{-1}\left(\frac{height}{2d}\right)$$

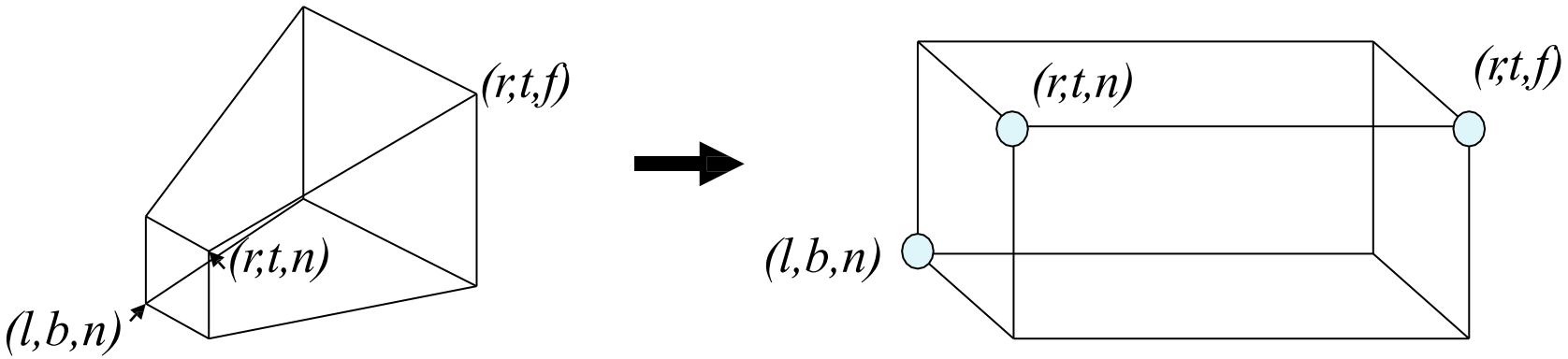
WebGL

- `mat4.perspective(...)`
  - Field of view in the *y* direction, *FOV*, (vertical field-of-view)
  - Aspect ratio, *a*, **should match window aspect ratio**
  - Near and far clipping planes, *n* and *f*
  - Defines a symmetric view volume



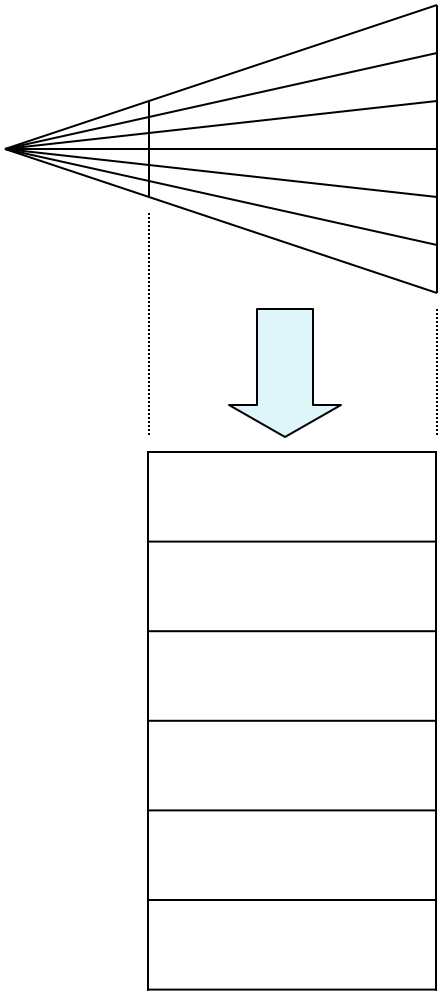
Perspective Projection Matrices

- We want a matrix that will take points in our perspective view volume and transform them into the orthographic view volume
  - This matrix will go in our pipeline before an orthographic projection matrix



Mapping Lines

- We want to map all the lines through the center of projection to parallel lines
  - This converts the perspective case to the orthographic case, we can use all our existing methods
- The relative intersection points of lines with the near clip plane should not change
- The matrix that does this looks like the matrix for our simple perspective case



General Perspective

$$\mathbf{M}_P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (n+f)/n & -f \\ 0 & 0 & 1/n & 0 \end{bmatrix} \equiv \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- This matrix leaves points with $z=n$ unchanged
- It is just like the simple projection matrix, but it does some extra things to z to map the depth properly
- We can multiply a homogenous matrix by any number without changing the final point, so the two matrices above have the same effect

Complete Perspective Projection

- After applying the perspective matrix, we map the orthographic view volume to the canonical view volume:

$$\mathbf{M}_{view->canonical} = \mathbf{M}_O \mathbf{M}_P = \begin{bmatrix} \frac{2}{(r-l)} & 0 & 0 & \frac{-(r+l)}{(r-l)} \\ 0 & \frac{2}{(t-b)} & 0 & \frac{-(t+b)}{(t-b)} \\ 0 & 0 & \frac{2}{(n-f)} & \frac{-(n+f)}{(n-f)} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & (n+f) & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{M}_{world->canonical} = \mathbf{M}_{view->canonical} \mathbf{M}_{world->view}$$

$$\mathbf{x}_{canonical} = \mathbf{M}_{world->canonical} \mathbf{x}_{world}$$

END