Geometry: Projection

Course web page:

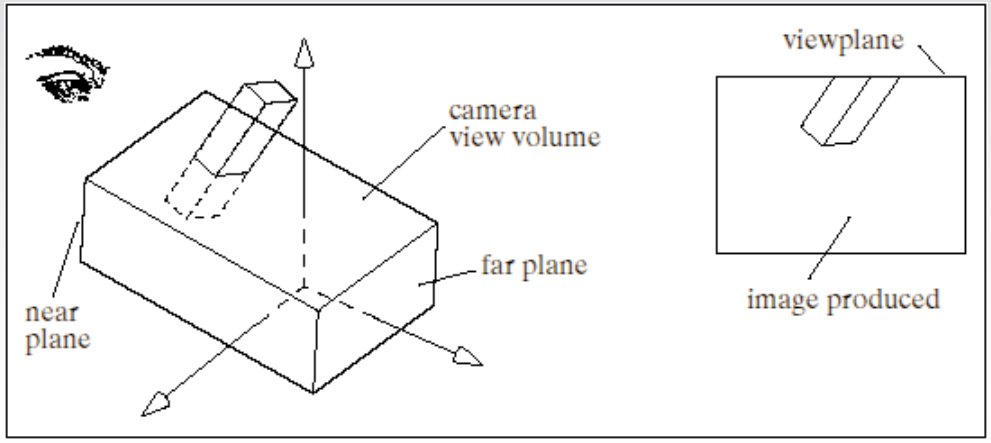
http://goo.gl/EB3aA

Outline

- Projections
  - Orthographic
  - Perspective
- Clipping

Orthographic Projection

- Projection direction **d** is aligned with Z axis
- Viewing volume is "brick"-shaped region in space
  - Not the same as image size
- No perspective effects—distant objects look same as near ones, so camera (x, y, z) ⇒ image (x, y)



from Hill

Orthographic Projection in OpenGL

- Setting up the viewing volume (VV):
  - **`glOrtho()`**
    - **`left, right, bottom, top`**: Coordinates of sides of viewing volume
    - **`znear, zfar`**: Distances to front, back sides of VV
      - Negative = Behind camera
  - **`gluOrtho2D()`**: **`glOrtho()`** with **`near = -1, far = 1`**
- Modifies top 4 x 4 matrix of **`GL_PERSPECTIVE`** matrix stack
  - Applied after **`GL_MODELVIEW`** transformation has put things in camera coordinates
  - Actual matrix scales VV to **canonical VV** (CVV): Cube extending from -1 to 1 along each dimension
    - This is properly a *transformation*; the *projection* is accomplished later by stripping off the Z coordinate

Simple Orthographic Projection Matrix

$$Pv = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 0 \\ 1 \end{bmatrix}$$

Orthographic Projection Matrix for CVV

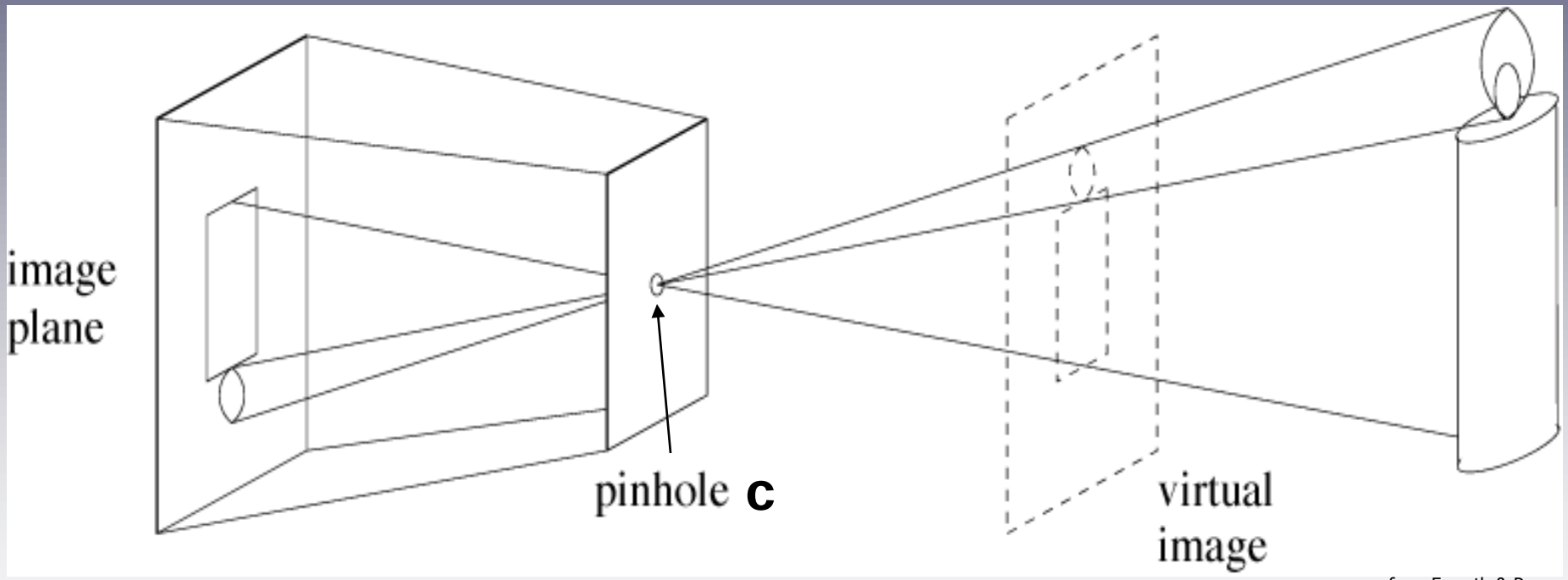
Translation to origin followed by scaling to 2 x 2 x 2 cube

(If n & f are not distances then n > f, so it should be "n – f" instead of "f – n" below — see formula 7.3 on p. 145 of Shirley):

$$P = ST = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{left+right}{2} \\ 0 & 1 & 0 & -\frac{top+bottom}{2} \\ 0 & 0 & 1 & -\frac{far+near}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Perspective with a Pinhole Camera (i.e., no lens)



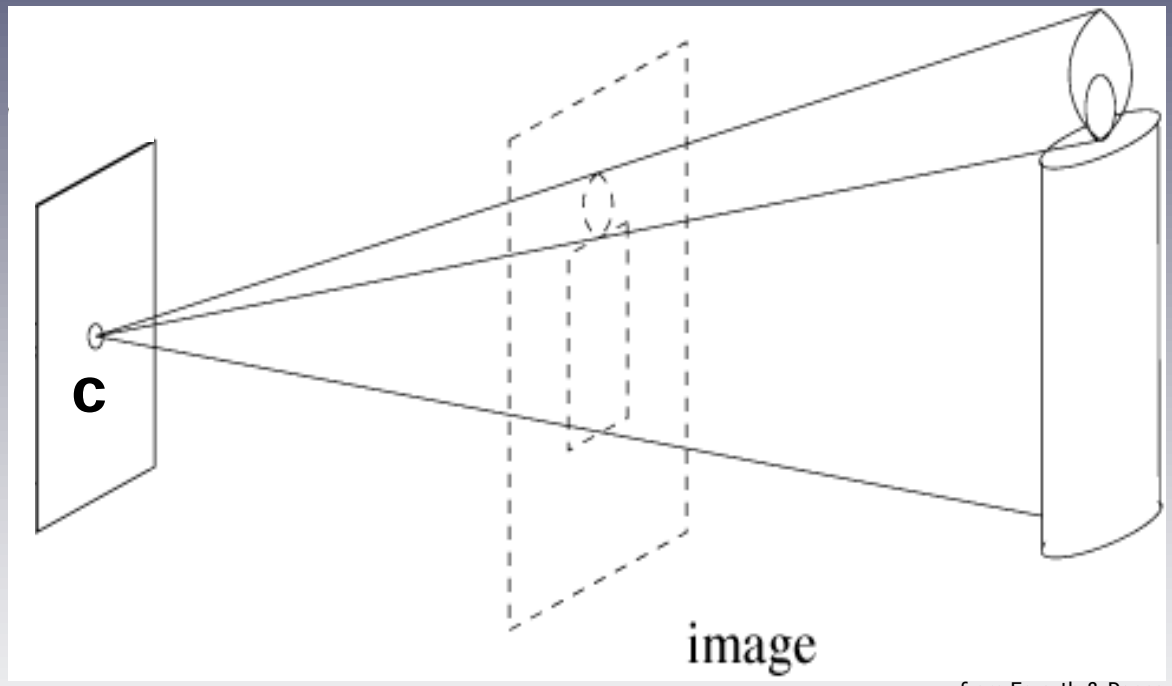
from Forsyth & Ponce

Instead of single direction **d** characteristic of parallel projections, rays emanating from single point **c** (or eye **e**) define perspective projection

Stenop.es project: Apartment as pinhole camera

More Stenop.es (note projection is upside-down)

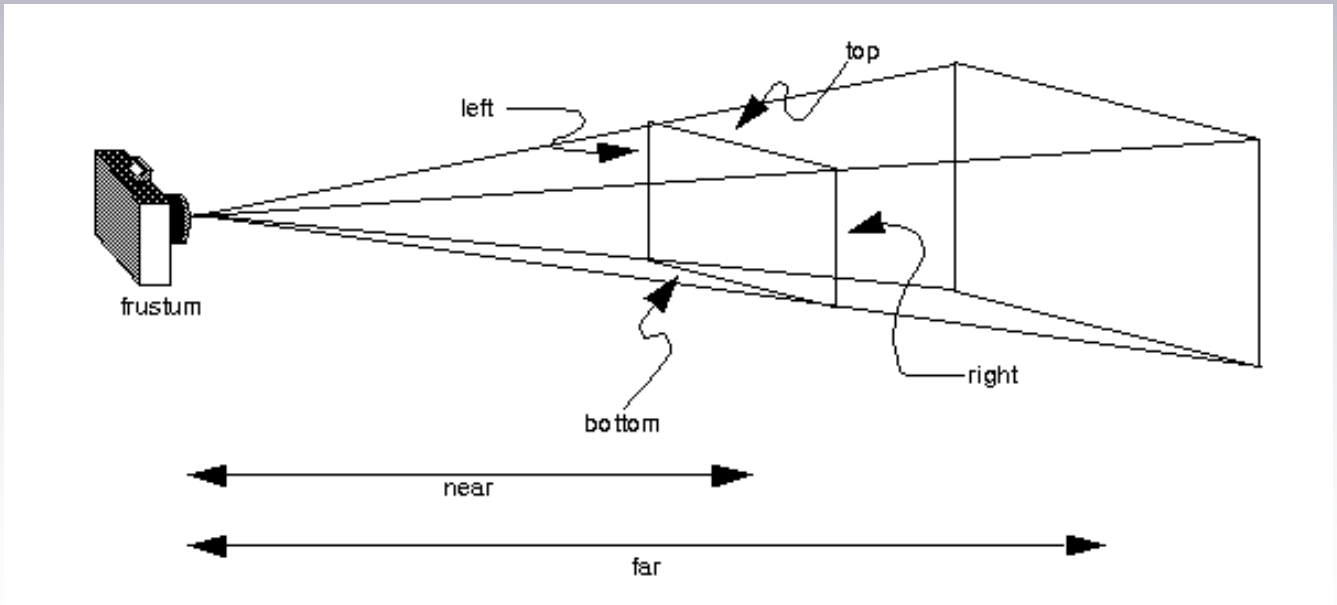
Perspective Projection



from Forsyth & Ponce

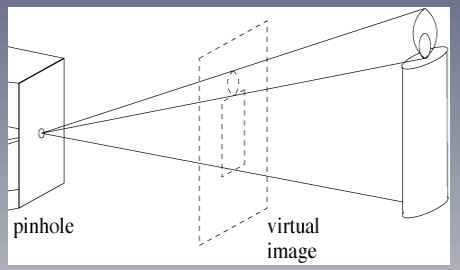
Perspective Projection: Viewing Volume

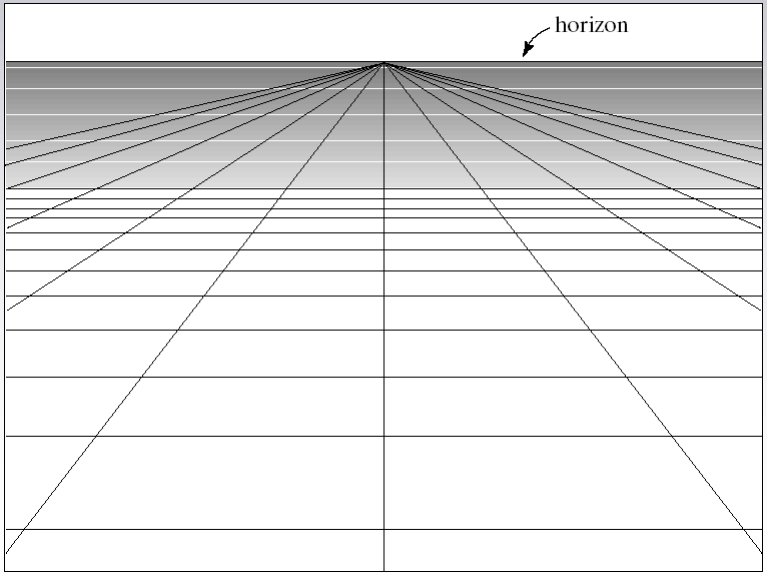
- Characteristic shape is a **frustum**—a truncated pyramid
  - Far plane is arbitrary



Perspective Projection: Properties

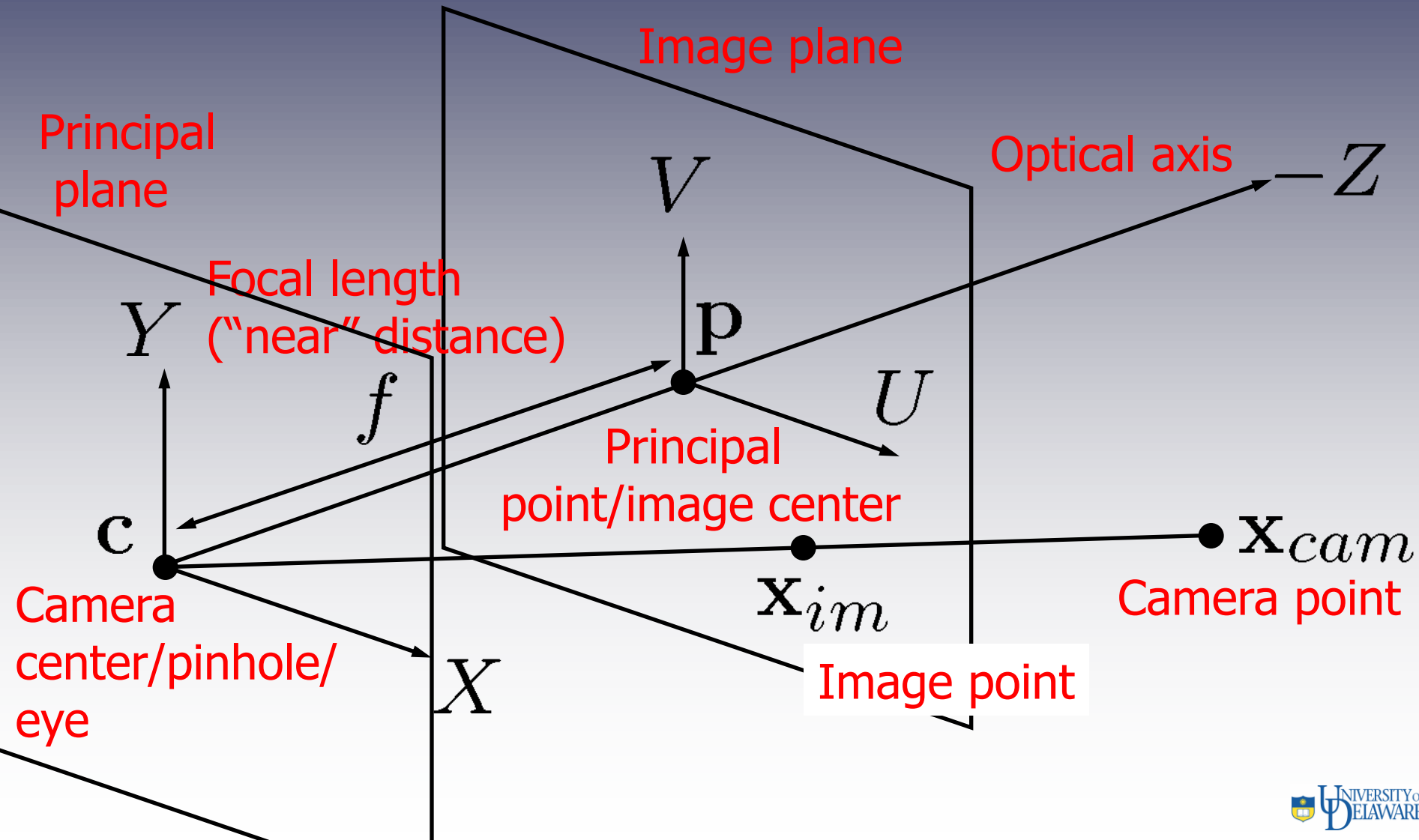
- Far objects appear smaller than near ones
- Lines are preserved
- Parallel lines in any plane Π (could be ground) converge at infinity
  - *Horizon* line defined by intersection of image plane with plane parallel to Π that passes through the pinhole





from Hill

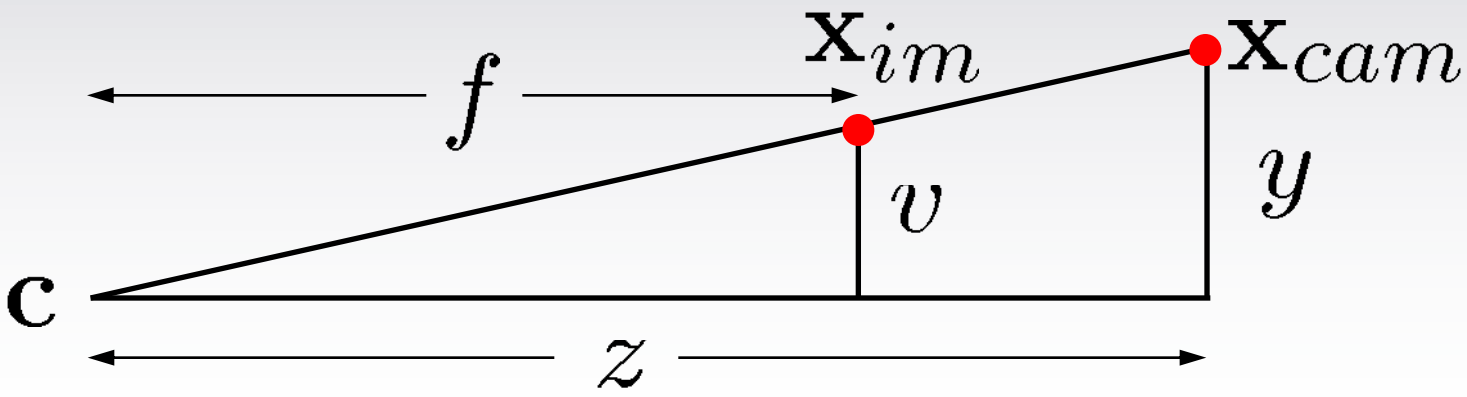
Pinhole Camera Terminology



Perspective Projection

- Letting the camera coordinates of the projected point be $\mathbf{x}_{cam} = (x, y, z)^T$ leads by similar triangles to:

$$\mathbf{x}_{im} = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} fx/z \\ fy/z \end{pmatrix}$$



Perspective Projection Matrix

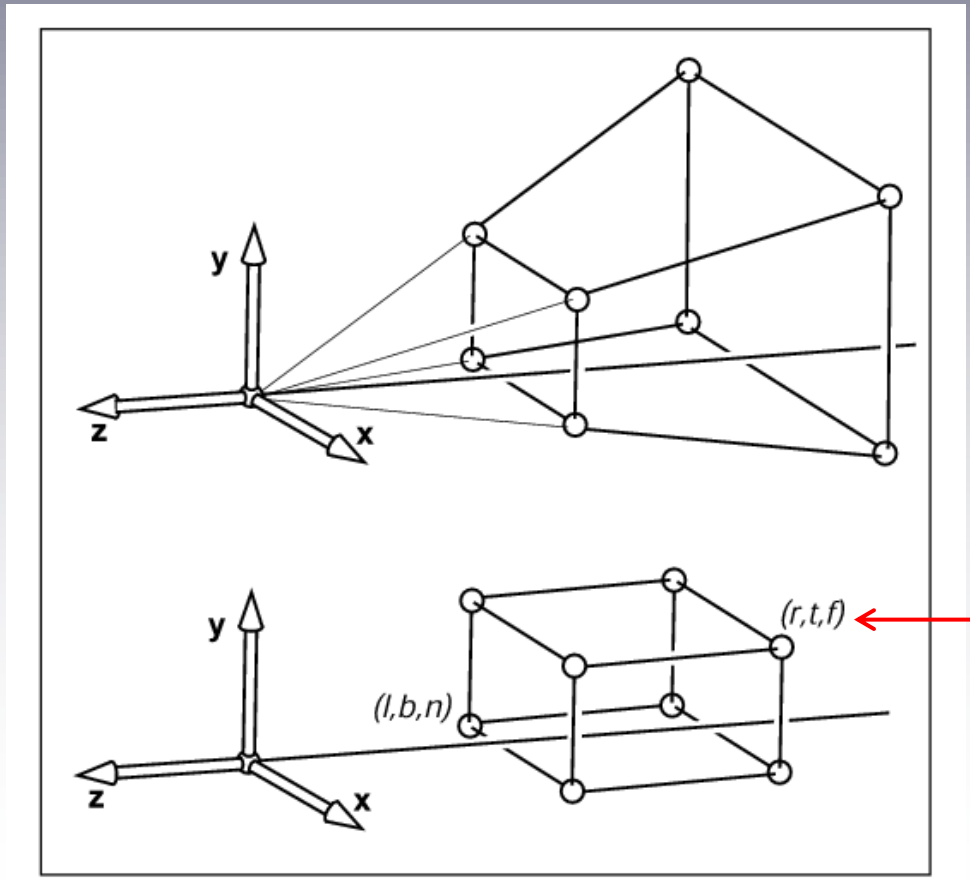
- Using homogeneous coordinates, we can describe a perspective transformation with the image plane at $z = -f$ (because $f > 0$ but $z < 0$) via a 4 x 4 matrix multiplication:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/f & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z/f \end{pmatrix} \rightarrow \begin{pmatrix} -fx/z \\ -fy/z \\ -f \\ 1 \end{pmatrix}$$

Last step accomplishes distance-dependent scaling by the rule for converting between homogeneous and regular coordinates. This is called the **perspective division**

Perspective Transformation Matrix

- Intuitively, what we're doing is transforming the viewing frustum into a "brick" (aka axis-aligned box), then doing orthographic projection



f does not mean same thing here as on previous slide!!

Perspective Transformation: Details

- Another way to write the transformation is in terms of near and far view volume planes *n* and *f* (which are negative)
- WARNING: *n* is image plane *z* value; *f* does not mean focal length on the next slide!!!

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Perspective Transformation: Details

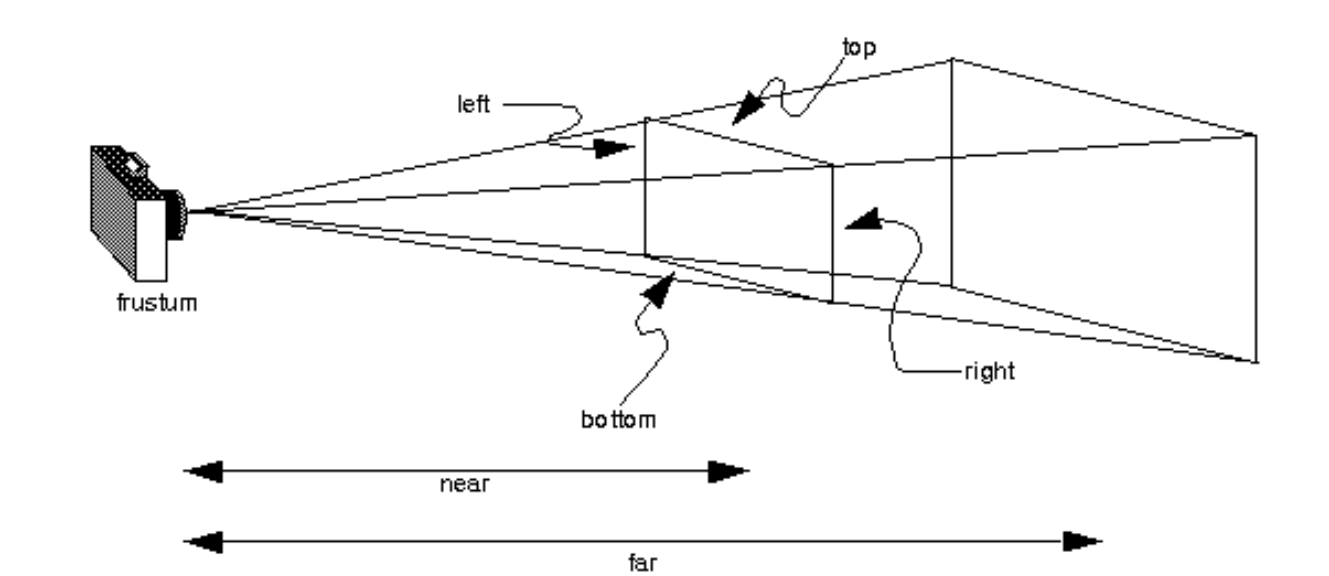
- Instead of simply projecting all *z* to near plane *n*, set matrix such that *z* = *n* maps to *n* and *z* = *f* maps to *f* (this is formula on p. 152 of Shirley):

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- After applying this, multiply by orthographic transform to scale everything to CVV and project to image coordinates
  - Project only after any steps that require **depth information**

Perspective Projections in OpenGL

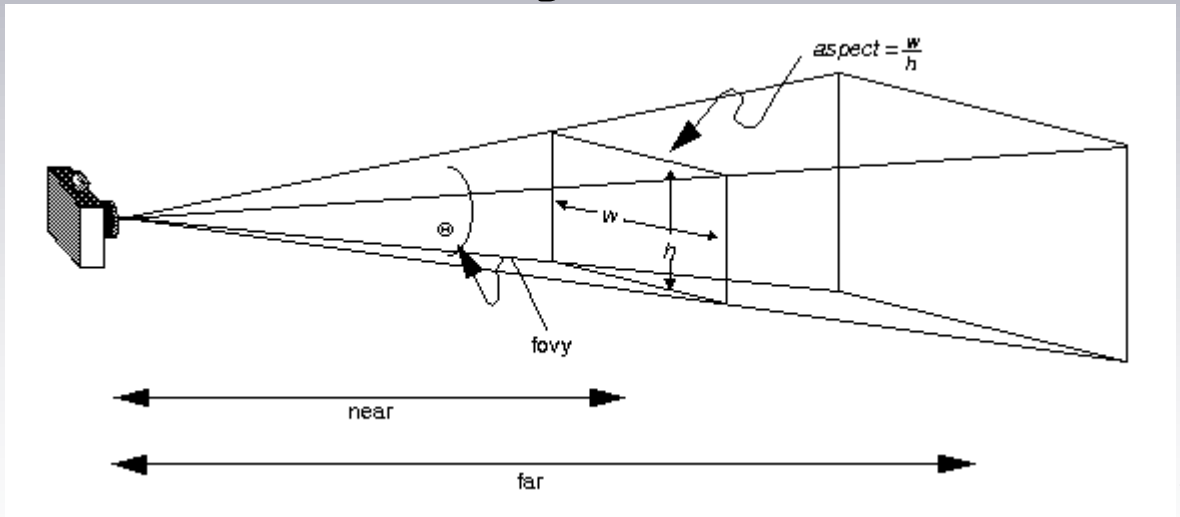
- **`glFrustum()`** sets transformation to CVV
  - Arguments like **`glOrtho()`** (set *left*, *right*, *bottom*, *top*, *near*, *far* directly) but *near*, *far* are distances and must be positive
  - **Vertical field of view (FOV)**: θ = 2 arctan(0.5 *height* / *near*), where *height* = *top* - *bottom*



from Woo *et al.*

gluPerspective()

- Simplifies call to **`glFrustum()`**
- Arguments:
  - **`fovy`**: Field of view angle (degrees) in Y direction
  - **`aspect`**: Ratio of width to height of viewing frustum
  - **`near, far`**: Same as **`glFrustum()`**



from Woo *et al.*

- Example: Robins' `projection` tutor

Field of View

- Controls "strength" of perspective effects

courtesy of S. Marschner

Viewport Transform

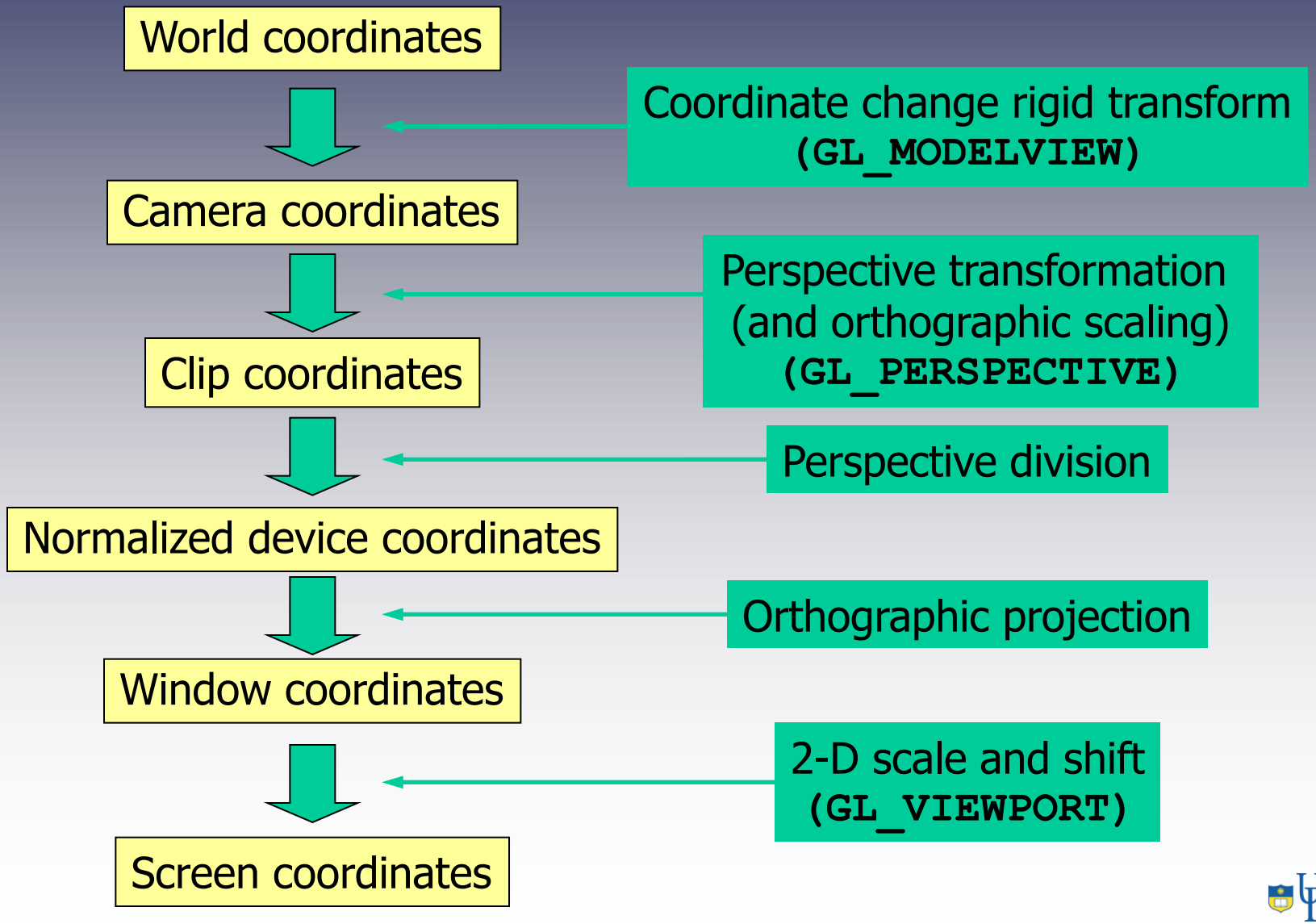
- A final transform, **`GL_VIEWPORT`**, shifts normalized device coordinates (NDC—what you have after perspective divide) to image coordinate origin and scales to fit window
- This is what makes x = 0, y = 0 go to corner of window instead of center
- **`glViewport(x, y, w, h)`**
  - Lower left corner and width, height of viewport
  - Default is (0, 0) and width, height of window
  - Often put in GLUT **`resize()`** callback for when window size changes

Applications of viewport transform

- Drawing space vs. controls
  - Separate viewports for rendered image and area where buttons, text, etc. are grouped
- Stereo
  - Draw scene twice—once for each "eye"
  - Change viewport for left, right views:

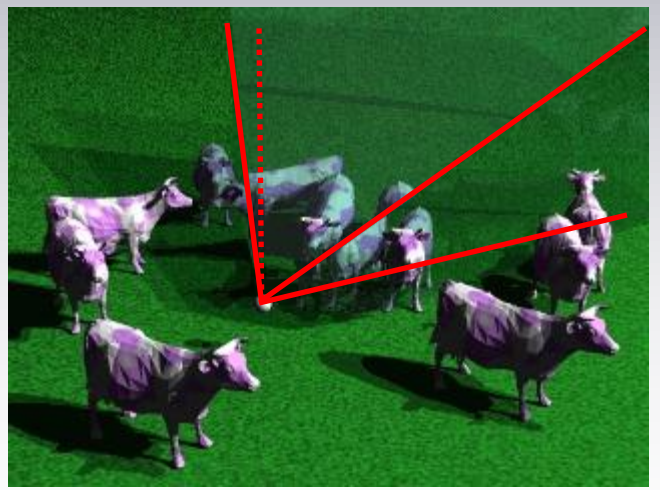
```
glViewport(0, 0, w/2, h);
...
glViewport(w/2, 0, w/2, h);
```

Geometry pipeline



Clipping

- Removal of portions of geometric primitives outside viewing volume (VV)
- Why?
  - Optimization that saves computation which would otherwise be wasted on lighting, texturing, etc.
- Cases
  - Trivial acceptance: Complete inside VV
  - Trivial rejection: Completely outside VV
  - Crossing **clip plane(s)**: Partially outside, so must trim to fit
- Different primitives require different methods
  - Points: Only trivial accept/reject
  - Lines: Chop at intersection with clip plane
  - Polygons: Must trim so as to maintain connectivity



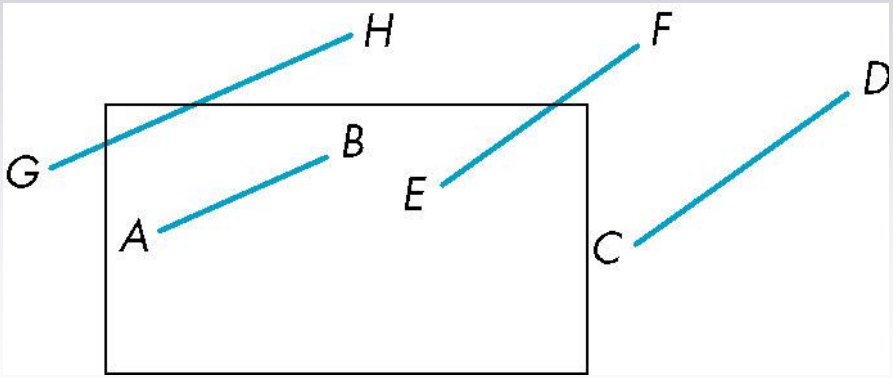
courtesy of L. McMillan

When to clip?

- The earlier in the pipeline invisible primitives are removed, the less computation is wasted on them
- However...difficulty of clipping problem depends on stage of geometry pipeline
  - **Camera coordinates**: VV is a frustum, so clip planes are angled (dot product necessary for inside/outside (IO) test)
  - **Clip coordinates**: CVV is an axis-aligned box (cube with corners at ±1 after perspective division), so clip planes are simple (IO test is simple greater-than/less-than comparison)
  - **NDC**: Perspective division destroys true sign of point's z---can't distinguish between point at (x, y, z) and one at (-x, -y, -z). This allows points behind eye to be erroneously displayed
  - **Screen coordinates**: Too late—nearly all of the work has already been done

Clipping: Basic issues

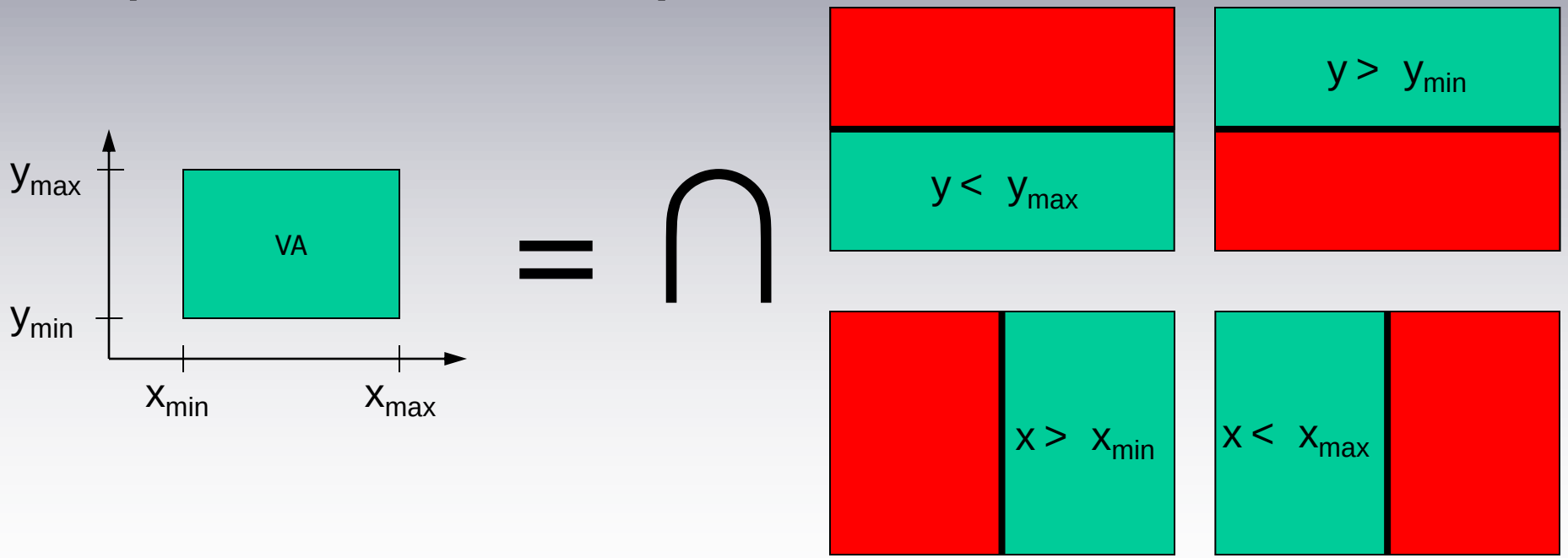
- Testing which side of a clip plane a point is on
  - Which side of a line in 2-D
- How to trim primitives that span VV border
  - Find the intersection of a line segment and a clip plane



from E. Angel

Cohen-Sutherland 2-D line clipping (not in textbook)

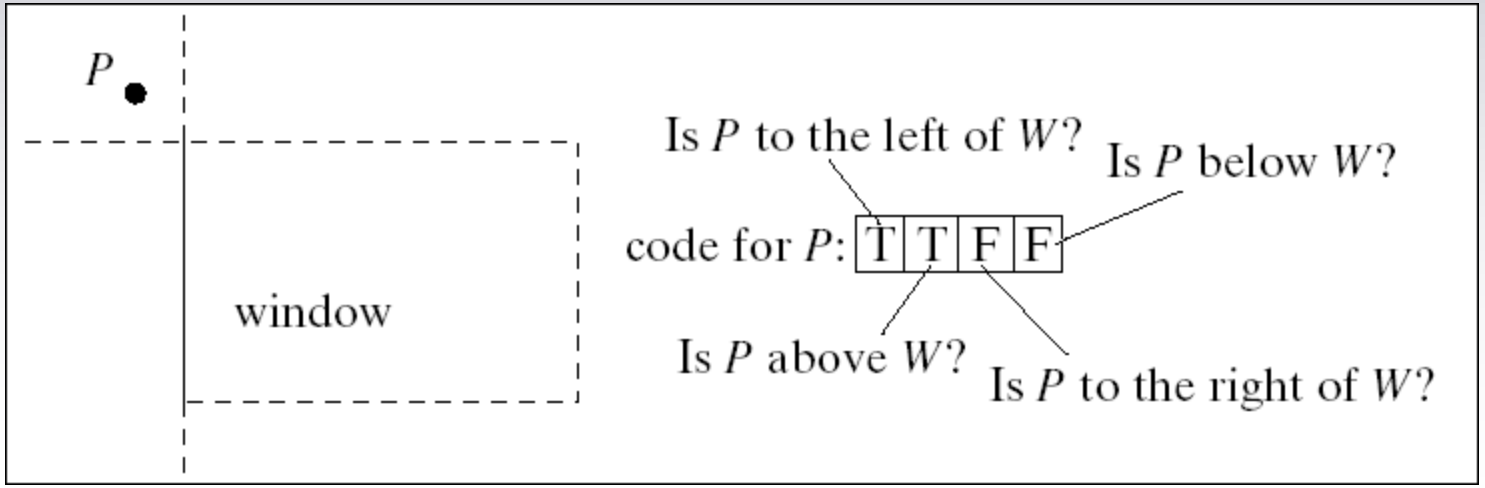
- Idea: Consider rectangle (the "viewing area" (VA)) as intersection of 4 half-planes defined by sides



adapted from F. Pfenning

Cohen-Sutherland clipping: Outcodes

- Test a line endpoint **p** = (x, y) against each of 4 half-planes and record whether it is **outside** the half-plane or not (**T** or **F**)
- These four **T**/**F** bits are **p**'s **outcode** o(**p**)

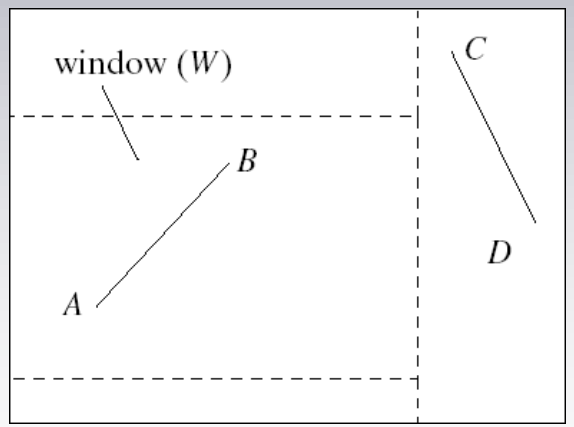


from Hill

Cohen-Sutherland clipping

- Outcodes partition plane around the viewing area
- Trivial line clipping cases
  - **Accept** line ($\mathbf{p}_1$, $\mathbf{p}_2$): Both endpoints are inside the rectangle
    - In terms of outcodes, this means o($\mathbf{p}_1$) = **FFFF** and o($\mathbf{p}_2$) = **FFFF**
  - **Reject** line: Both endpoints outside rectangle **on same side**
    - This means both points' outcodes have a **T** at the same bit position—e.g., o($\mathbf{p}_1$) = **FTTF** and o($\mathbf{p}_2$) = **FFTF**

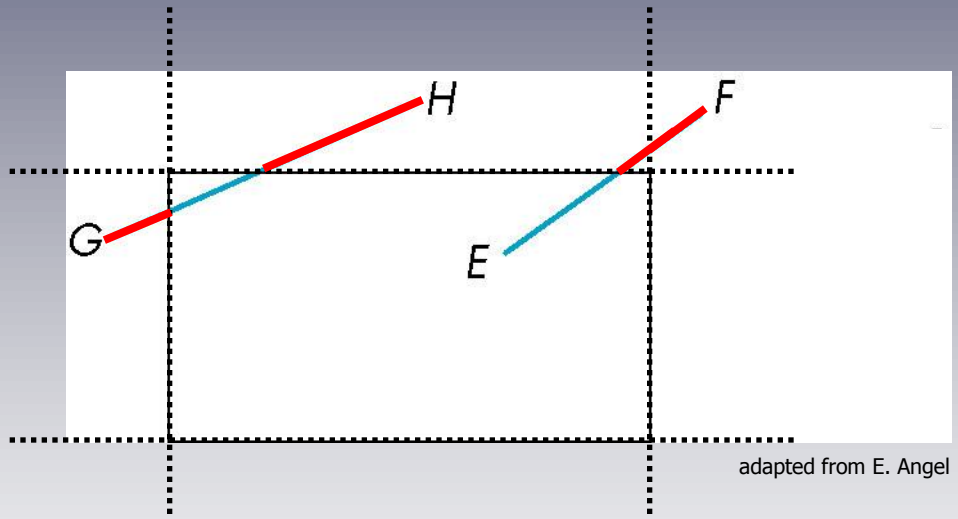
TTFF	FTFF	FTTF
TFFF	FFFF window	FFTF
TFFT	FFFT	FFTT



from Hill

Cohen-Sutherland clipping

- Tougher cases are what's left



adapted from E. Angel

- Basic idea: Subdivide non-trivial lines by sequentially removing (aka clipping) portions outside rectangle edge lines until what's left is trivial
  - Arbitrary order: Left, right, bottom, top

Cohen-Sutherland: Algorithm

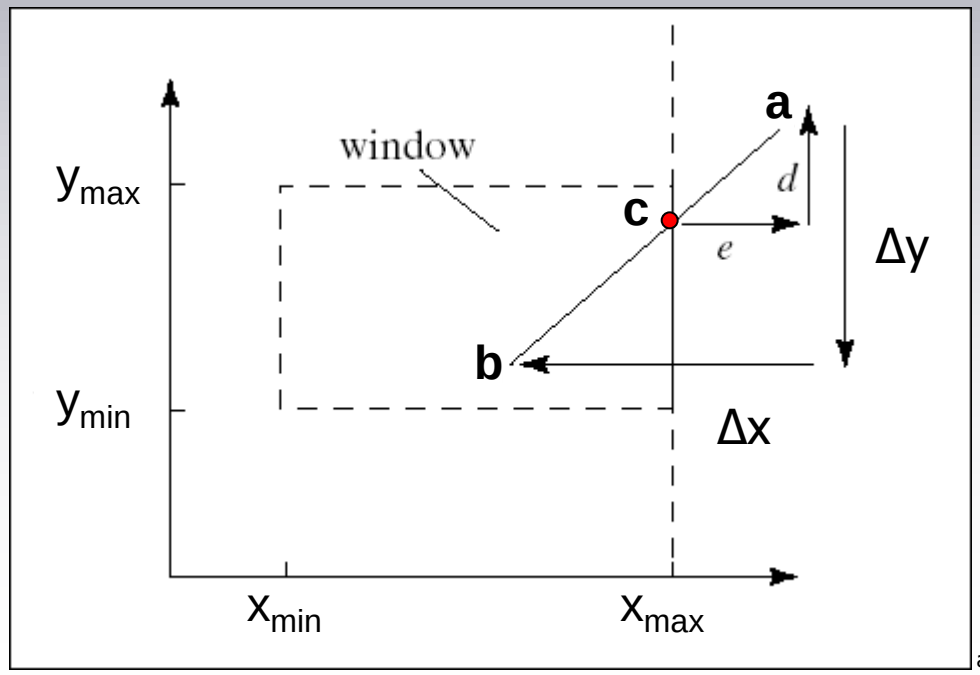
```
int clipSegment(Point2& p1, Point2& p2, RealRect W)
{
    do{
        if(trivial accept) return 1; // some portion survives
        if(trivial reject) return 0; // no portion survives

        if(p1 is outside )
        {
            if(p1 is to the left ) chop against the left edge
            else if(p1 is to the right ) chop against the right edge
            else if(p1 is below) chop against the bottom edge
            else if(p1 is above) chop against the top edge
        }
        else      // p2 is outside
        {
          if(p2 is to the left ) chop against the left edge
               else if(p2 is to the right ) chop against the right edge
          else if(p2 is below) chop against the bottom edge
          else if(p2 is above) chop against the top edge
        }
    }while(1);
}
```

compute outcodes

compute intersection

Computing the intersection point

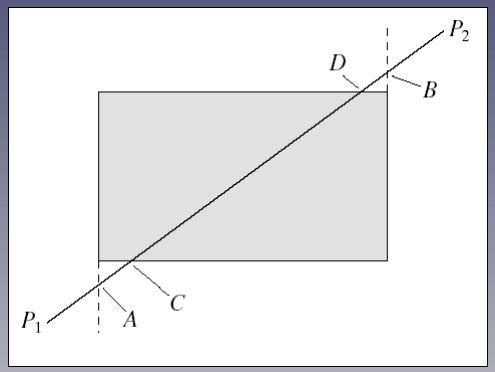
- What is **c** = (c_x, c_y)?
  - Obviously, in this case $c_x = x_{max}$ and $c_y = a_y - d$
  - Noting that $e = a_x - x_{max}$ and $e/\Delta x = d/\Delta y$, we can compute $d = e\,\Delta y/\Delta x$ and obtain c_y
- A similar approach works for the other clip lines



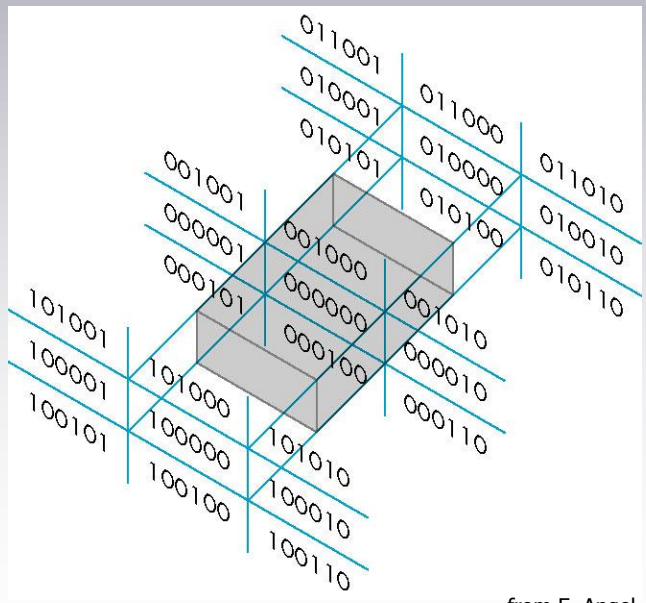
adapted from Hill

Line clipping: Notes

- C-S's recursive clipping (up to 4 passes) is not optimal
  - E.g., **Liang-Barsky** method can be faster
- C-S generalizable to 3-D
  - Instead of 4 half-planes there are 6 half-spaces (3-D volumes)
  - Outcode has 6 bits (two more for near and far Z clip planes)



from Hill



from E. Angel