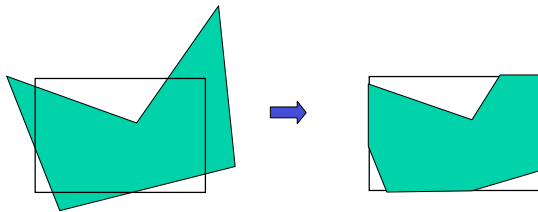


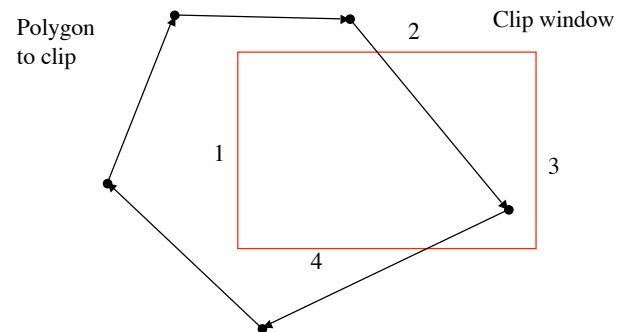
## Polygon clip (against convex polygon)



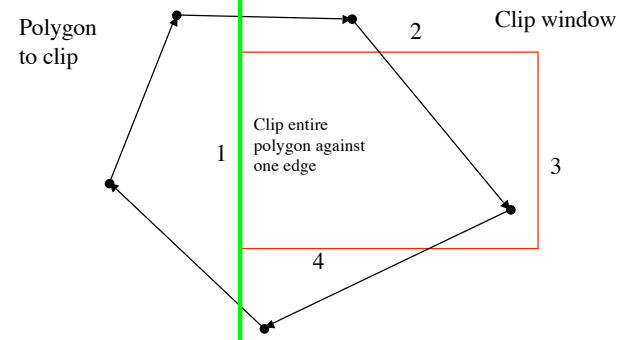
## Sutherland-Hodgeman polygon clip

- Recall: polygon is convex if any line joining two points inside the polygon, also lies inside the polygon; implies that a point is inside if it is on the right side of each edge.
- Clipping each edge of a given polygon doesn't make sense - how do we reassemble the pieces? We want to arrange doing so on the fly.
- Clipping the polygon against each edge of the clip window in *sequence* works if the clip window is *convex*.
- (Note similarity to Sutherland-Cohen line clipping)

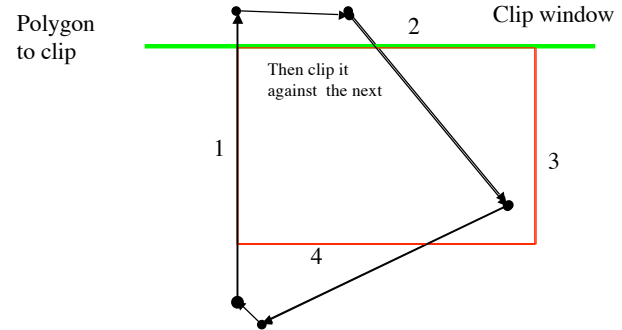
## Sutherland-Hodgeman polygon clip



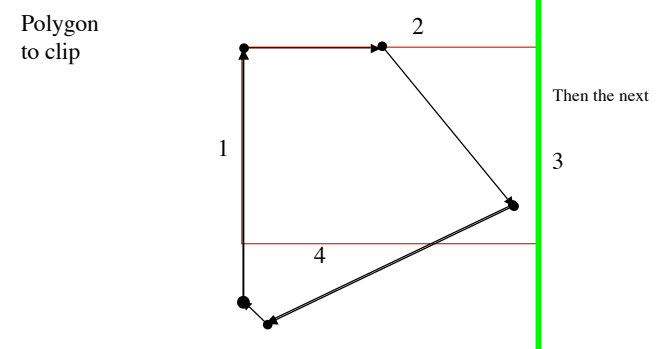
## Sutherland-Hodgeman polygon clip



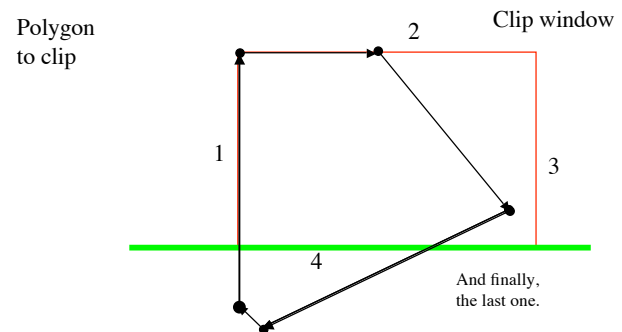
### Sutherland-Hodgeman polygon clip



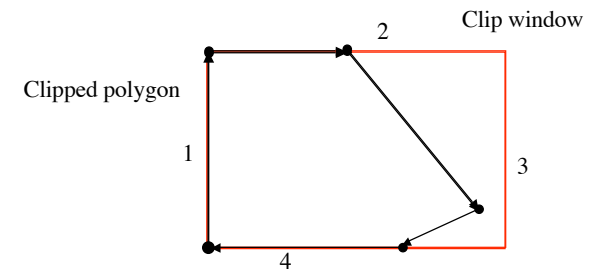
### Sutherland-Hodgeman polygon clip



### Sutherland-Hodgeman polygon clip



### Sutherland-Hodgeman polygon clip

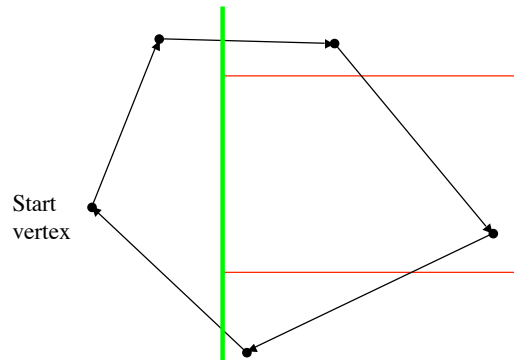


## Clipping against **current** clip edge

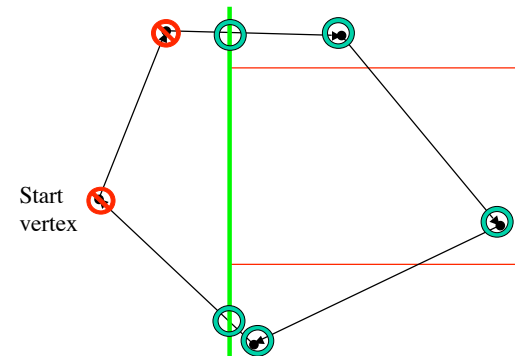
- Polygon is a list of vertices
- Think of process as rewriting polygon, vertex by vertex
- Check start vertex
  - in - emit it
  - out - ignore it
- Walk along vertices and for each edge consider four cases and apply corresponding action.
  - Four cases:
    - polygon edge crosses clip edge going from out to in
      -
    - polygon edge crosses clip edge going from in to out
      -
    - polygon edge goes from out to out
      -
    - polygon edge goes from in to in
      -

## Clipping against current clip edge

- Polygon is a list of vertices
- Think of process as rewriting polygon, vertex by vertex
- Check start vertex
  - in - emit it
  - out - ignore it
- Walk along vertices and for each edge consider four cases and apply corresponding action.
  - Four cases:
    - polygon edge crosses clip edge going from out to in
      - emit crossing, next vertex
    - polygon edge crosses clip edge going from in to out
      - emit crossing
    - polygon edge goes from out to out
      - emit nothing
    - polygon edge goes from in to in
      - emit next vertex

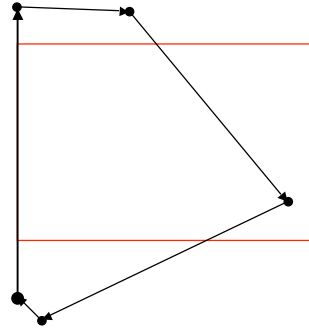


polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex  
 polygon edge crosses clip edge going from in to out ==> emit crossing  
 polygon edge goes from out to out ==> emit nothing  
 polygon edge goes from in to in ==> emit next vertex



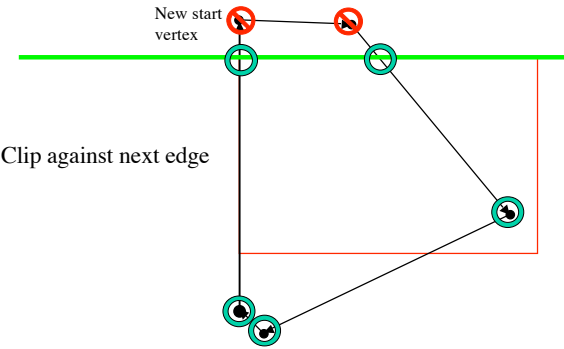
polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex  
 polygon edge crosses clip edge going from in to out ==> emit crossing  
 polygon edge goes from out to out ==> emit nothing  
 polygon edge goes from in to in ==> emit next vertex

Now have



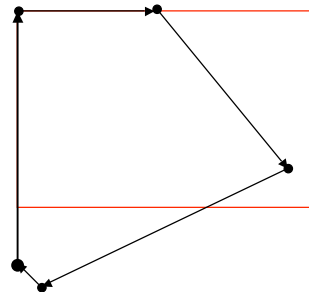
polygon edge crosses clip edge going from out to in	==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out	==> emit crossing
polygon edge goes from out to out	==> emit nothing
polygon edge goes from in to in	==> emit next vertex

Clip against next edge



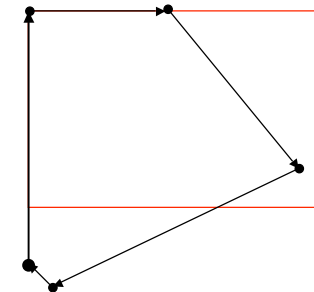
polygon edge crosses clip edge going from out to in	==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out	==> emit crossing
polygon edge goes from out to out	==> emit nothing
polygon edge goes from in to in	==> emit next vertex

Now have



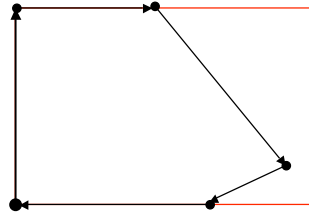
polygon edge crosses clip edge going from out to in	==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out	==> emit crossing
polygon edge goes from out to out	==> emit nothing
polygon edge goes from in to in	==> emit next vertex

Clipping against  
next edge (right)  
gives



polygon edge crosses clip edge going from out to in	==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out	==> emit crossing
polygon edge goes from out to out	==> emit nothing
polygon edge goes from in to in	==> emit next vertex

Clipping against  
final(bottom)  
edge gives



polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex  
 polygon edge crosses clip edge going from in to out ==> emit crossing  
 polygon edge goes from out to out ==> emit nothing  
 polygon edge goes from in to in ==> emit next vertex

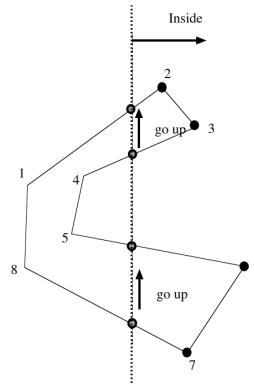
## More Polygon clipping

- Notice that we can have a pipeline of clipping processes, one against each edge, each operating on the output of the previous clipper -- substantial advantage.
- Unpleasantness can result from concave polygons; in particular, polygons with empty interior.
- Can modify algorithm for concave polygons (e.g. Weiler Atherton)

## Weiler Atherton

For clockwise polygon (starting outside):

- For out-to-in pair, follow usual rule
- For in-to-out pair, follow clip edge (clockwise) and then jump to next vertex (which is on the outside) and start again
- Only get a second piece if polygon is convex



## Additional remarks on clipping

- Although everything discussed so far has been in terms of polygons/lines clipped against lines in 2D, all - except Nicholl-Lee-Nicholl - will work in 3D against convex regions without much change.
- This is because the central issue in each algorithm is the inside outside decision as a convex region is the intersection of half spaces.
- Inside-outside decisions can be made for lines in 2D, planes in 3D. e.g. testing  $dx \cdot n \geq 0$
- Hence, all (except N-L-N) can be used to clip:
  - Lines against 3D convex regions (e.g. cubes)
  - Polygons against 3D convex regions (e.g. cubes)
- NLN could work in 3D, but the number of cases increases too much to be practical.

## 2D Transformations

- Represent **linear** transformations by matrices
- To transform a point, represented by a vector, multiply the vector by the appropriate matrix.

## 2D Transformations

- Represent **linear** transformations by matrices
- To transform a point, represented by a vector, multiply the vector by the appropriate matrix.
- Recall the definition of matrix times vector:

$$\begin{pmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

- A linear function  $f(x)$  satisfies (by definition):

$$f(ax + by) = af(x) + bf(y)$$

- Note that “x” can be an abstract entity (e.g. a vector)—as long as addition and multiplication by a scalar are defined.
- Algebra reveals that matrix multiplication satisfies the above condition

- In particular., if we define  $f(\mathbf{x}) = \mathbf{M} \cdot \mathbf{x}$ , where  $\mathbf{M}$  is a matrix and  $\mathbf{x}$  is a vector, then

$$\begin{aligned} f(a\mathbf{x} + b\mathbf{y}) &= \mathbf{M}(a\mathbf{x} + b\mathbf{y}) \\ &= a\mathbf{M}\mathbf{x} + b\mathbf{M}\mathbf{y} \\ &= af(\mathbf{x}) + bf(\mathbf{y}) \end{aligned}$$

- Where the middle step can be verified using algebra (next slide)

### Proof that matrix multiplication is linear

$$\begin{aligned}
 M(a\mathbf{x} + b\mathbf{y}) &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} ax_1 + by_1 \\ ax_2 + by_2 \end{pmatrix} \\
 &= \begin{pmatrix} a_{11}ax_1 + a_{11}by_1 + a_{12}ax_2 + a_{12}by_2 \\ a_{21}ax_1 + a_{21}by_1 + a_{22}ax_2 + a_{22}by_2 \end{pmatrix} \\
 &= a \begin{pmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{pmatrix} + b \begin{pmatrix} a_{11}y_1 + a_{12}y_2 \\ a_{21}y_1 + a_{22}y_2 \end{pmatrix} \\
 &= aM\mathbf{x} + bM\mathbf{y}
 \end{aligned}$$

- Now consider the linear transformation of a point on a line segment connecting two points,  $\mathbf{x}$  and  $\mathbf{y}$ .
- Recall that in parametric form, that point is:  $t\mathbf{x} + (1-t)\mathbf{y}$
- The transformed point is:  $f(t\mathbf{x} + (1-t)\mathbf{y}) = tf(\mathbf{x}) + (1-t)f(\mathbf{y})$
- Notice that is a point on the line segment from the point  $f(\mathbf{x})$  to the point  $f(\mathbf{y})$ ,
- This shows that a linear transformation maps line segments to line segments

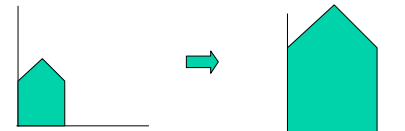
[ H&B chapter 5]

### 2D Transformations of objects

- To transform line segments, transform endpoints
- To transform polygons, transform vertices

### 2D Transformations

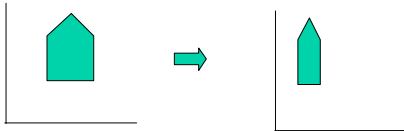
- Scale (stretch) by a factor of  $k$



$$M = \begin{vmatrix} k & 0 \\ 0 & k \end{vmatrix} \quad (k = 2 \text{ in the example})$$

## 2D Transformations

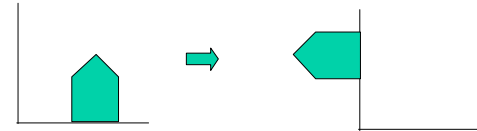
- Scale by a factor of  $(S_x, S_y)$



$$M = \begin{vmatrix} S_x & 0 \\ 0 & S_y \end{vmatrix} \quad (\text{Above, } S_x = 1/2, S_y = 1)$$

## 2D Transformations

- Rotate around origin by  $\theta$  (Orthogonal)



$$M = \begin{vmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{vmatrix} \quad (\text{Above, } \theta = 90^\circ)$$

## 2D Transformations

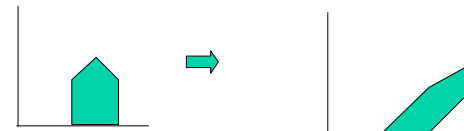
- Flip over y axis (Orthogonal)



$$M = \begin{vmatrix} -1 & 0 \\ 0 & 1 \end{vmatrix} \quad \text{Flip over x axis is ?}$$

## 2D Transformations

- Shear along x axis

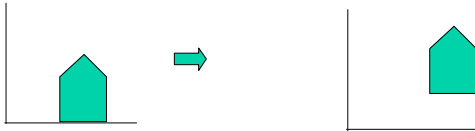


$$M = \begin{vmatrix} 1 & a \\ 0 & 1 \end{vmatrix} \quad \text{Shear along y axis is ?}$$



## 2D Transformations

- Translation ( $\mathbf{P}_{\text{new}} = \mathbf{P} + \mathbf{T}$ )



$\mathbf{M} = ?$

## Homogenous Coordinates

- Represent 2D points by 3D vectors
- $(x, y) \rightarrow (x, y, 1)$
- Now a multitude of 3D points  $(x, y, W)$  represent the same 2D point,  $(x/W, y/W, 1)$
- Represent 2D transforms with 3 by 3 matrices
- Can now do translations
- Homogenous coordinates have other uses/advantages (later)

## 2D Translation in H.C.

$$\mathbf{P}_{\text{new}} = \mathbf{P} + \mathbf{T}$$

$$(x', y') = (x, y) + (t_x, t_y)$$

$$\mathbf{M} = \begin{vmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{vmatrix}$$

## 2D Scale in H.C.

$$\mathbf{M} = \begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

## 2D Rotation in H.C.

$$M = \begin{vmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

## Composition of Transformations

- If we use one matrix,  $M_1$  for one transform and another matrix,  $M_2$  for a second transform, then the matrix for the first transform followed by the second transform is simply  $M_2 M_1$
- This generalizes to any number of transforms
- Computing the combined matrix **first** and then applying it to many objects, can save **lots** of computation

## Composition Example

- Matrix for rotation about a point, P
- Problem--we only know how to rotate about the origin.

## Composition Example

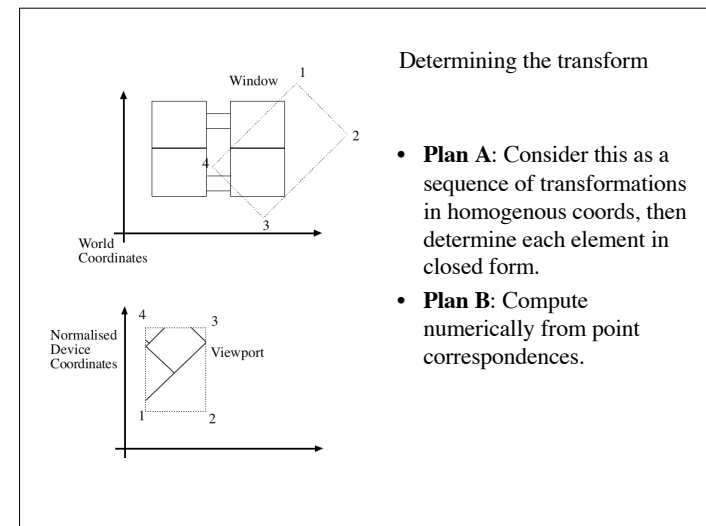
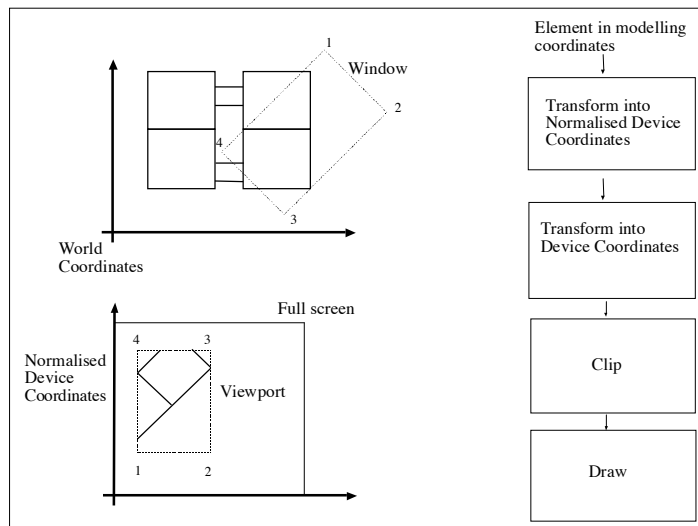
- Matrix for rotation about a point, P
- Problem--we only know how to rotate about the origin.
- Solution--translate to origin, rotate, and translate back

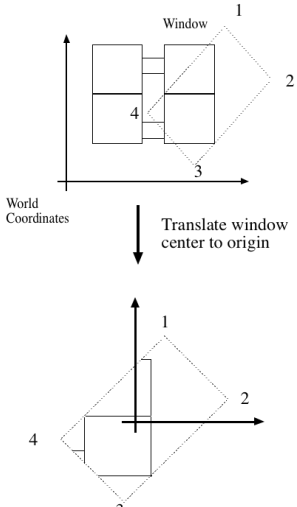
## 2D transformations (continued)

- The transformations discussed so far are invertible (why?). What are the inverses?

## 2D viewing

- Three coordinate systems are common in graphics
  - World coordinates or modeling coordinates - where the model is defined (meters, miles, etc.)
  - Normalized device coordinates; usually (0-1) in each variable.
  - Device coordinates: the actual coordinates of the pixels on the frame-buffer or the printer
- Need to construct transformations between coordinate systems
- Terminology:
  - window = region on drawing that will be displayed (rectangle)
  - viewport = region in NDC's/DC's where this rectangle is displayed (often simply entire screen).

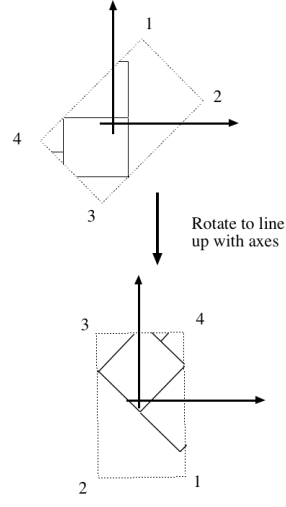




- write  $(wx_i, wy_i)$  for coordinates of  $i$ 'th point on window
- translation is:

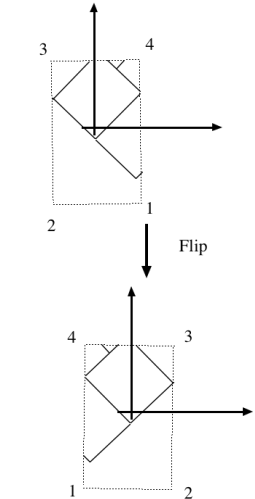
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -\overline{wx} \\ 0 & 1 & -\overline{wy} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

(overbar denotes average over vertices, i.e., 1,2,3,4)



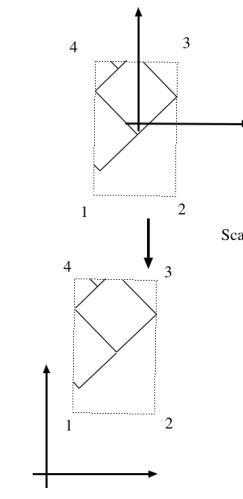
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

(Need to compute theta)



$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

(Vertex order does not correspond, need to flip)



$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{w_{new}}{w_{old}} & 0 & \overline{x_{new}} \\ 0 & \frac{h_{new}}{h_{old}} & \overline{y_{new}} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The variables labeled “new” are in either device coordinates or normalized device coordinates (depending on what you want).

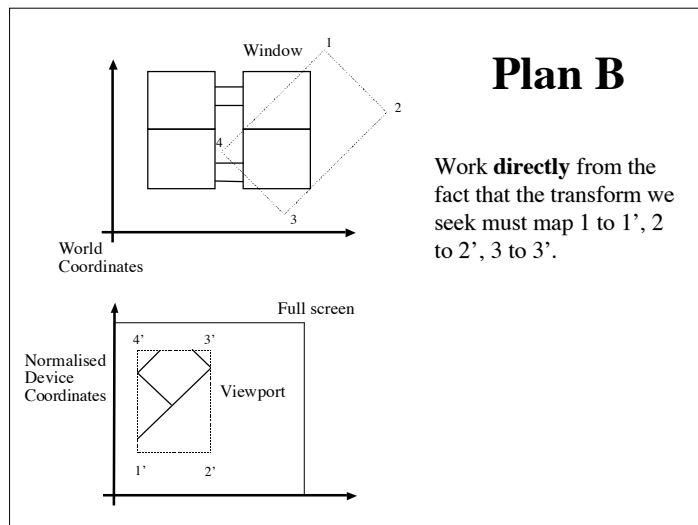
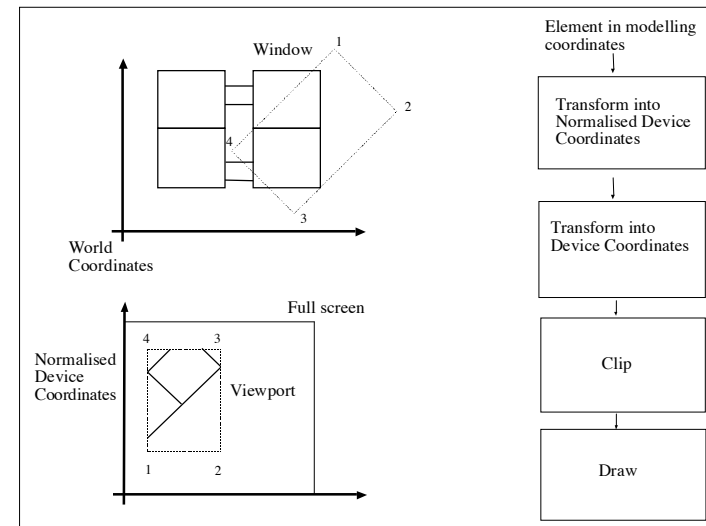
The variables labeled “old” are in world coordinates.

- Get overall transformation by multiplying transforms.
- This gives a single transformation matrix, whose elements are functions of window/viewport coordinates.

$$x' = M_{(\text{translate origin to viewport cog, scale})} M_{(\text{flip})} M_{(\text{rotate})} M_{(\text{translate window cog} \rightarrow \text{origin})} x$$

NDC's/DC's World coords

(cog==window center of gravity)



Details Optional

Plan B: Solve for the affine transformation directly.

- We know that this is an “affine” transform.
- In particular, the matrix we seek is:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$$

## More Details

Details Optional

- Consider the first mapping,  $Mp_1=q_1$
- $p_1 = (x_1, y_1, 1)^T$ ,  $q_1 = (u_1, v_1, 1)^T$

$$\begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix}$$

$$\begin{aligned} ax_1 + by_1 + c &= u_1 \\ dx_1 + ey_1 + f &= v_1 \end{aligned}$$

## More Details

Details Optional

Write

$$ax_1 + by_1 + c = u_1$$

As

$$x_1a + y_1b + 1 \bullet c + 0 \bullet d + 0 \bullet e + 0 \bullet f = u_1$$

Notice that this gives one equation in the six unknowns

## More Details

Details Optional

Similarly, write

$$dx_1 + ey_1 + f = v_1$$

As

$$0 \bullet a + 0 \bullet b + 0 \bullet c + x_1 \bullet d + y_1 \bullet e + 1 \bullet f = u_1$$

Notice that this gives a second equation in the six unknowns

## More Details

Details Optional

- $Mp_1=q_1$  gives first two rows
- $p_1 = (x_1, y_1, 1)^T$ ,  $q_1 = (u_1, v_1, 1)^T$

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix}$$

$$\begin{aligned} ax_1 + by_1 + c &= u_1 \\ dx_1 + ey_1 + f &= v_1 \end{aligned}$$

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{pmatrix}$$

$Mp_2=q_2$ ,  $Mp_3=q_3$  give other rows

## More Details

Details Optional

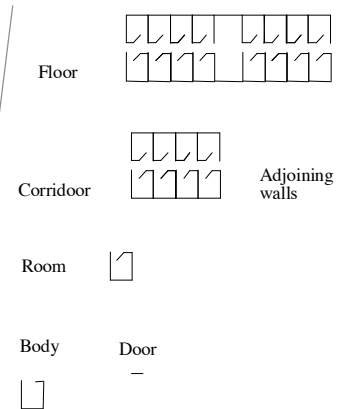
Final representation  
of six equations in  
six unknowns

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix} = \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{pmatrix}$$

This can be solved using standard methods

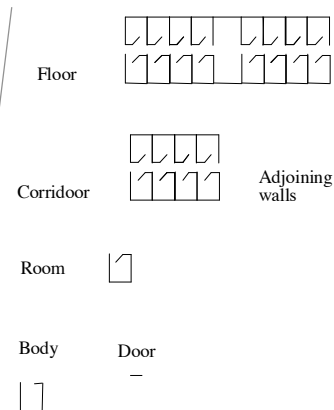
## Hierarchical modeling

- Consider constructing a complex 2d drawing: e.g. an animation showing the plan view of a building, where the doors swing open and shut.

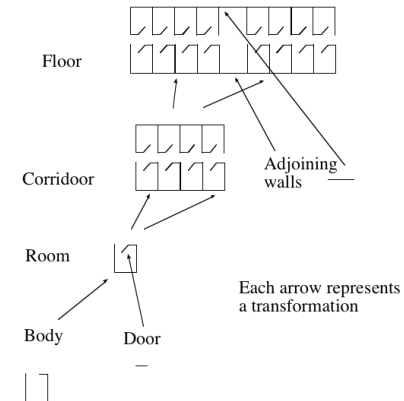


## Hierarchical modeling

- Options:
  - specify everything in world coordinate frame; but then each room is different, and each door moves differently.
  - Exploit similarities by using repeated copies of models in different places (instancing)



## Hierarchical modeling



## Hierarchical modeling

- Model form
  - Directed acyclic graph.
  - Each node consists of 0 or more objects (lines, polygons, etc).
  - Each edge is a transformation
- There can be many edges joining two nodes (e.g. in the case of the corridor - many copies of the same room model, each transformed differently).
- Every graphics API supports hierarchies - some directly (meaning you have to learn a language to express the model) some indirectly with a matrix stack

## Hierarchical modeling

Write the transformation from door coordinates to room coordinates as:

$$T_{room}^{door}$$

Then to render a door, use the transformation:

$$T_{device}^{world} T_{floor}^{corridor} T_{corridor}^{room} T_{room}^{door}$$

To render a body, use the transformation:

$$T_{device}^{world} T_{floor}^{corridor} T_{corridor}^{room} T_{room}^{body}$$

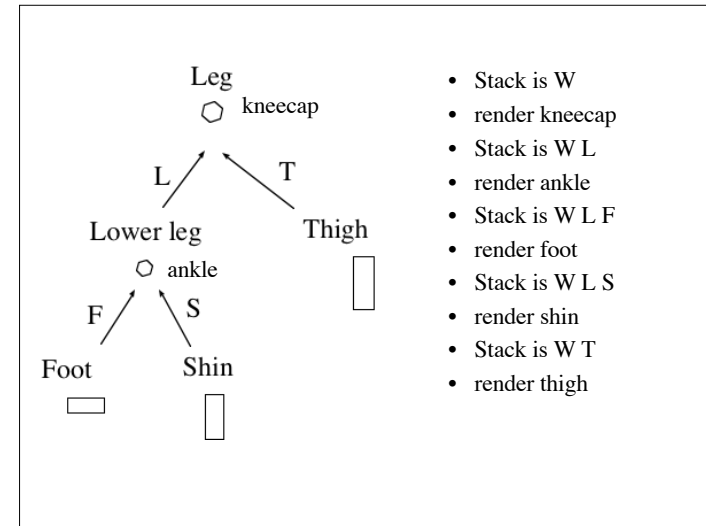
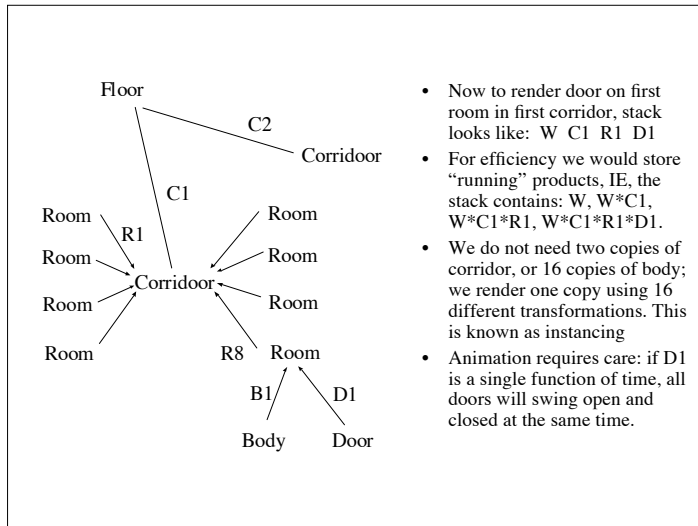
## Matrix stacks and rendering

- Matrix stack:
  - Stack of matrices used for rendering
  - Applied in sequence.
  - Pop=remove last matrix
  - Push=append a new matrix
  - In previous example, body-device transformation comes from door-device transformation by popping door-room and pushing body-room

## Matrix stacks and rendering

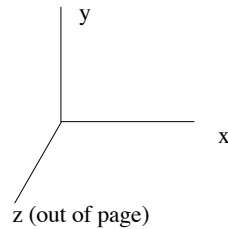
- Algorithm for rendering a hierarchical model:
  - stack is  $T_{device}^{root}$
  - render (root)
- Recursive definition of render (node)
  - if node has object, render it
  - for each child:
    - push transformation
    - render (child)
    - pop transformation



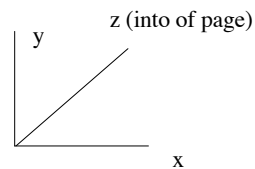


## Transformations in 3D

- Right hand coordinate system (conventional, i.e., in math)



- In graphics a LHS is sometimes also convenient (Easy to switch between them--later).



## Transformations in 3D

- Homogeneous coordinates now have four components - traditionally,  $(x, y, z, w)$ 
  - ordinary to homogeneous:  $(x, y, z) \rightarrow (x, y, z, 1)$
  - homogeneous to ordinary:  $(x, y, z, w) \rightarrow (x/w, y/w, z/w)$
- Again, translation can be expressed as a multiplication.

## Transformations in 3D

- Translation:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

## 3D transformations

- Anisotropic scaling:
- Shear (one example):

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

## Rotations in 3D

- 3 degrees of freedom
- Orthogonal,  $\det(R)=1$
- We can easily determine formulas for rotations about each of the axes
- For general rotations, there are many possible representations—we will use a **sequence** of rotations about coordinate axes.
- Sign of rotation follows the Right Hand Rule--point thumb along axis in direction of increasing ordinate--then fingers curl in the direction of positive rotation).

## Rotations in 3D

- About x-axis

$$M = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rotations in 3D

- About y-axis

$$M = \begin{vmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rotations in 3D

- About z-axis

$$M = \begin{vmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Commuting transformations

- If A and B are matrices, does  $AB=BA$ ? Always? Ever?
- What if A and B are restricted to particular transformations?
- What about the 2D transformations that we have studied?
- How about if A and B are restricted to be on of the three specific 3D rotations just introduced, such as rotation about the Z axis?

## Demo

## Commuting transformations

- If A and B are matrices, does  $AB=BA$ ? Always? Ever?
- What if A and B are restricted to particular transformations?
- What about the 2D transformations that we have studied?
- How about if A and B are restricted to be on of the three specific 3D rotations just introduced, such as rotation about the Z axis?

**Answer:** In general  $AB \neq BA$  (matrix multiplication is not commutative). But if A and B are either translations or scalings, then multiplication is commutative. The same applies to rotations restricted to be about one of the 3 axis in 3D.

## Rotations in 3D

- About X axis

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

- 90 degrees about X axis?

## Rotations in 3D

- About X axis

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

- 90 degrees about X axis

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rotations in 3D

- About Y axis

$$\begin{vmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

- 90 degrees about Y-axis?

## Rotations in 3D

- About Y axis

$$\begin{vmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

- 90 degrees about Y axis

$$\begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rotations in 3D

- 90 degrees about X then Y

$$\begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = ?$$

Y rot                      X rot

## Rotations in 3D

- 90 degrees about X then Y

$$\begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Y rot                      X rot

## Rotations in 3D

- 90 degrees about X then Y

$$\begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Y rot                      X rot

- 90 degrees about Y then X

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = ?$$

X rot                      Y rot

## Rotations in 3D

- 90 degrees about X then Y

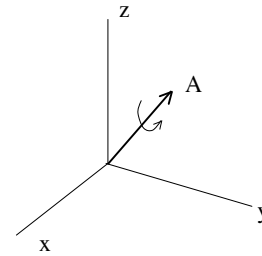
$$\begin{matrix} \begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \\ \text{Y rot} & \text{X rot} \end{matrix} = \begin{vmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

← ≠

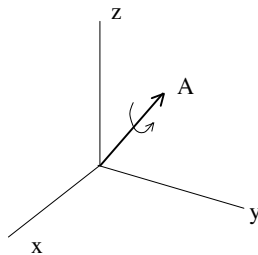
- 90 degrees about Y then X

$$\begin{matrix} \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} & \begin{vmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \\ \text{X rot} & \text{Y rot} \end{matrix} = \begin{vmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rotation about an arbitrary axis

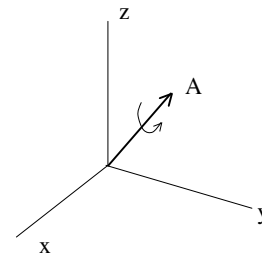


## Rotation about an arbitrary axis



Strategy--rotate A to Z axis, rotate about Z axis, rotate Z back to A.

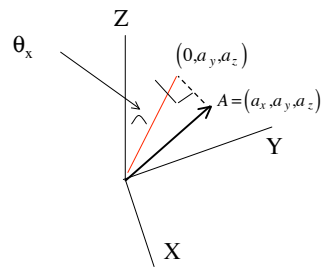
## Rotation about an arbitrary axis



Tricky part:  
rotate A to Z axis

- Two steps.
- 1) Rotate about x to xz plane
  - 2) Rotate about y to Z axis.

### Rotation about an arbitrary axis



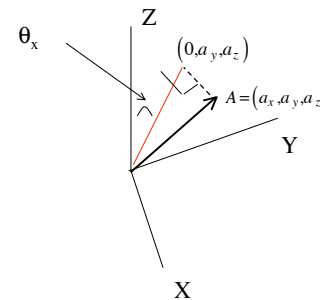
Tricky part:  
rotate A to Z  
axis

Two steps.

- 1) Rotate about X to xz plane
- 2) Rotate about Y to Z axis.

As A rotates into the xz plane, its projection (shadow) onto the YZ plane (red line) rotates through the same angle which is easily calculated.

### Rotation about an arbitrary axis



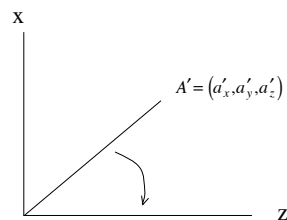
$$d = \sqrt{a_y^2 + a_z^2}$$

$$\sin \theta_x = a_y / d$$

$$\cos \theta_x = a_z / d$$

No need to compute angles,  
just put sines and cosines into  
rotation matrices

### Rotation about an arbitrary axis



Apply  $R_x(\theta_x)$  to A and renormalize to get A'

$R_y(\theta_y)$  should be easy, but note that it is clockwise.

### Rotation about an arbitrary axis

Final form is

$$R_x(-\theta_x)R_y(-\theta_y)R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

## Transforming the Normal Vector

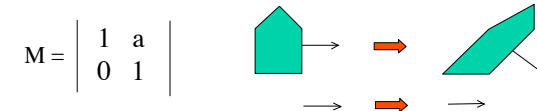
- The normal to a polygon does **not** always transform in the same way as the points on the polygon
  - One case where it does: ?
  - One case where it does not: ?

## Transforming the Normal Vector

- The normal to a polygon does **not** always transform in the same way as the points on the polygon
  - One case where it does: **Orthogonal**

Somewhat intuitive

- One case where it does not: **Shear**



## Transforming the Normal Vector

- One way to find the transformed normal is to first transform the polygon, and then re-compute the normal.
- We can often save some time by computing the transformed normal (*why can this save time?*)

Details optional

## Transforming the Normal Vector

Let  $\mathbf{t}$  be a vector tangent to the polygon, and  $\mathbf{n}$  the normal

$M\mathbf{t}$  is on the transformed polygon

Want a transformation  $N$  so that  $N\mathbf{n}$  is perpendicular to all  $M\mathbf{t}$

$$\mathbf{n} \cdot \mathbf{t} = \mathbf{n}^T \mathbf{t} = 0 \quad \text{and so (trick!)} \quad (\mathbf{n}^T M^{-1})(M\mathbf{t}) = 0$$

So,  $\mathbf{n}^T M^{-1}$  is the row vector version of the transformed normal

$$\text{So, } N\mathbf{n} = (\mathbf{n}^T M^{-1})^T = (M^{-1})^T \mathbf{n}$$

$$\text{And } N = (M^{-1})^T$$



## Transforming the Normal Vector

Derived transformation for the normal:  $N = (M^{-1})^T$

Note that  $Nn$  is not necessarily a unit vector

The formula proves that orthogonal transformations also transform the normal because orthogonal transformations satisfy:

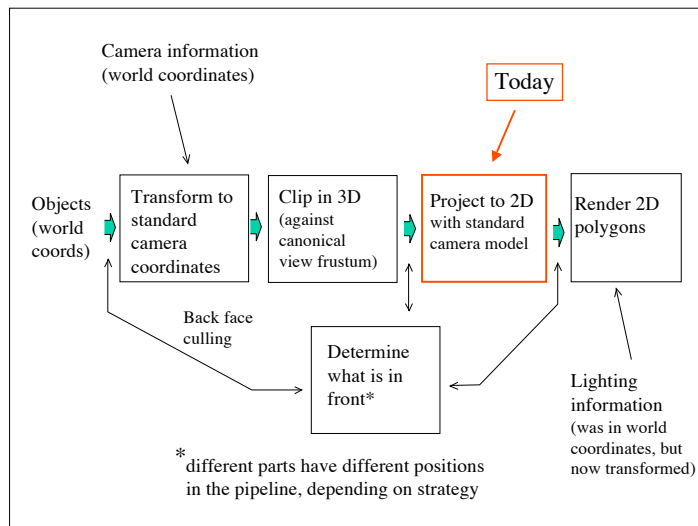
$$M = (M^{-1})^T$$

(The inverse of an orthogonal matrix is its transpose)

## 3D Graphics Concepts

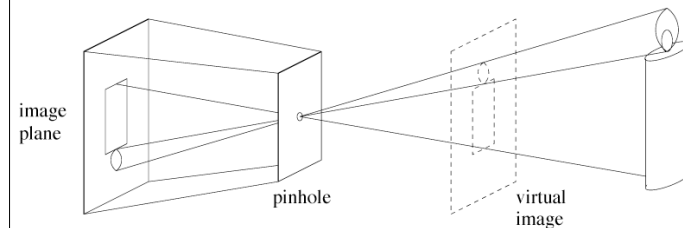
(H&B ch. 7, Watt ch. 5, Foley et al ch. 6)

- Modeling: For now, objects will be collections of polygons in 3D. Complex shapes will be many small polygons.
- Issues:
  - Which polygons can be seen? (some polygons hide others, and some are outside the relevant volume of space and need to be clipped).
  - Where do they go in the 2D image? (key abstraction is a virtual camera)
  - How bright should they be? (for example, to make it look as if we are looking at a real surface)

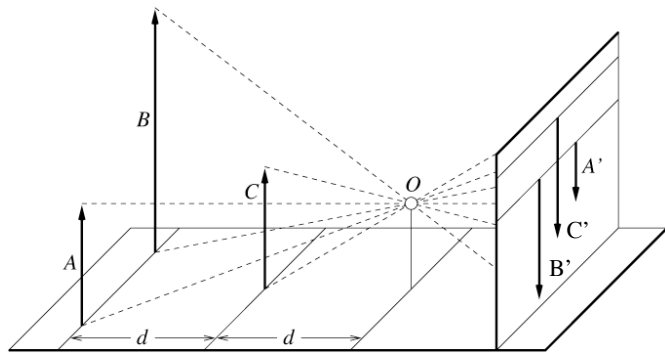


## Pinhole cameras

- Abstract camera model-- box with a small hole in it
- Pinhole cameras work for deriving algorithms--a real camera needs a lens

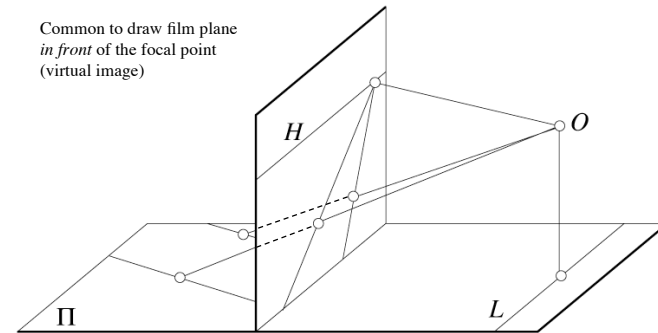


### Distant objects are smaller



### Parallel lines meet\*

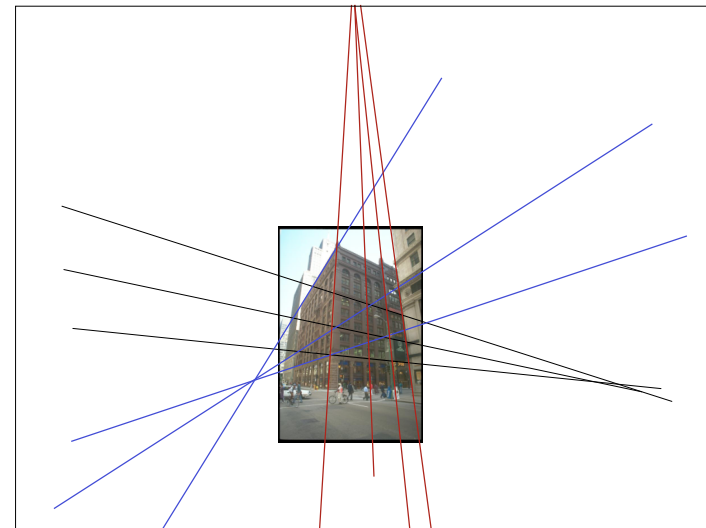
Common to draw film plane  
in front of the focal point  
(virtual image)

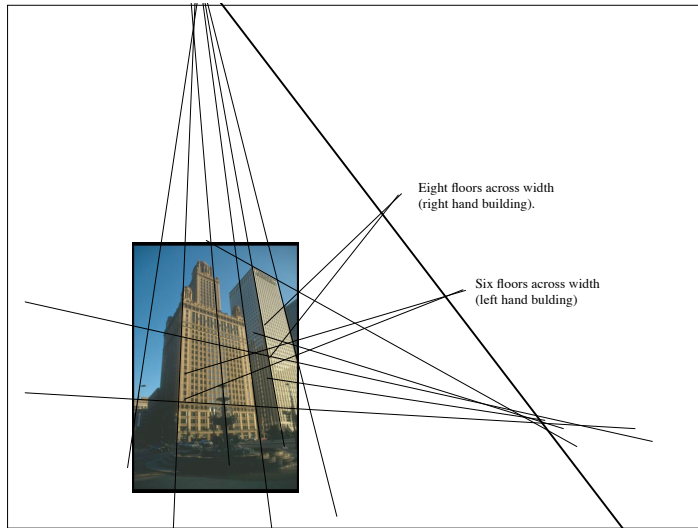


\*Exceptions?

### Vanishing points

- Each set of parallel lines (=direction) meets at a different point
  - The *vanishing point* for this direction
- Sets of parallel lines on the same plane lead to *collinear* vanishing points.
  - The line is called the *horizon* for that plane
  - Standard horizon is the horizon of the ground plane.
- One way to spot fake images

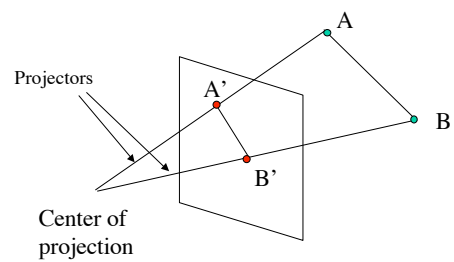




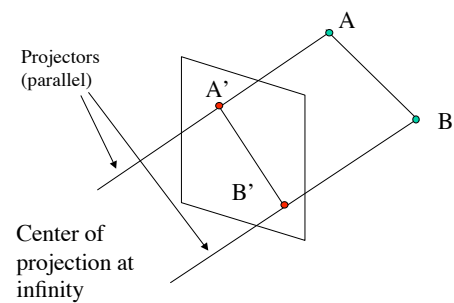
## Projections

- Mathematical definition of a projection:  $PP=P$
- (Doing it a second time has no effect).
- Generally rank deficient (non-invertable)--exception is  $P=I$
- Transformation loses information (e.g., depth)
- Given a 2D image, there are many 3D worlds that could have lead to it.

## Projections



## Parallel Projection



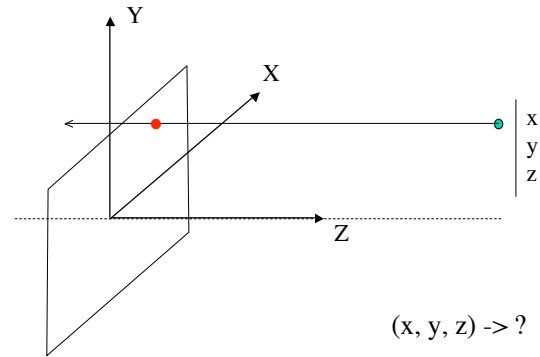
## Parallel Projection

Parallel lines remain parallel, some 3D measurements can be made using 2D picture

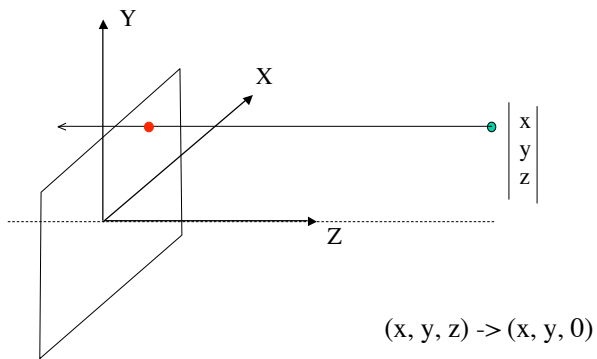
Does not give realistic 3D view because eye is more like perspective projection.

If projection plane is perpendicular to projectors the projection is *orthographic*

## Orthographic example (onto $z=0$ )



## Orthographic example (onto $z=0$ )



Can we do this with a linear transformation?

$$\begin{pmatrix} x \\ y \\ 0 \end{pmatrix} = \begin{bmatrix} \phantom{x} \\ \phantom{y} \\ \phantom{0} \end{bmatrix} ? \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

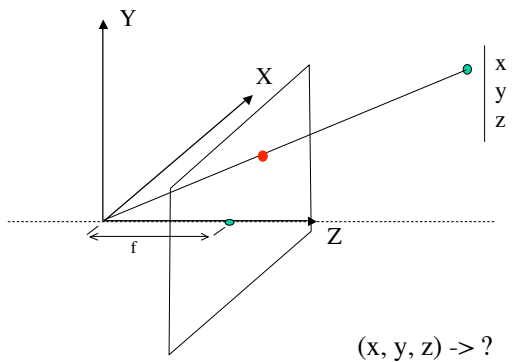
Can we do this with a linear transformation?

$$\begin{pmatrix} x \\ y \\ 0 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

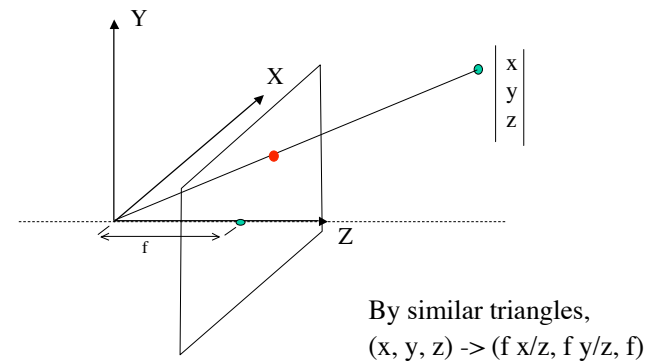
Camera matrix in homogeneous coordinates

$$\begin{pmatrix} x \\ y \\ 0 \\ w \end{pmatrix} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 0 & \\ & & & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Perspective example (onto  $z=f$ )



Perspective example (onto  $z=f$ )



Can we do this with a linear transformation?

$$f \begin{pmatrix} x/z \\ y/z \\ 1 \end{pmatrix} = ? \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Now try homogeneous coordinates

- In homogeneous coordinates

$$(x, y, z, 1) \Rightarrow (f \frac{x}{z}, f \frac{y}{z}, f, 1)$$

- Equivalently

$$(x, y, z, 1) \Rightarrow (x, y, z, \frac{z}{f})$$

- (Now H.C. are being used to store foreshortening)

Is there a linear transformation?

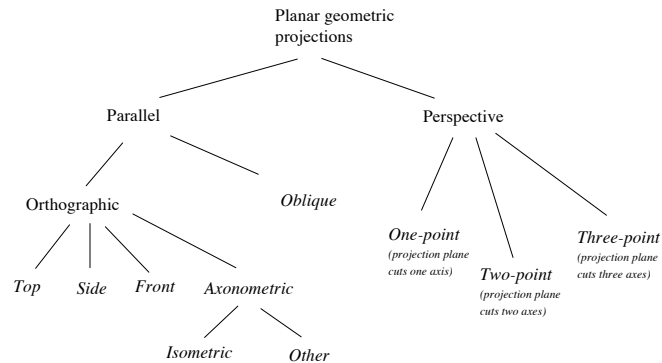
$$\begin{pmatrix} x \\ y \\ z \\ \frac{z}{f} \end{pmatrix} = ? \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Summary so far

- Parallel and perspective are projections (non-invertable)
- Perspective viewing is not a linear transform in (x,y,z)
- Perspective viewing can be encoded as a linear transformation in homogenous coordinates
- Perspective becomes parallel projection as  $f$  becomes infinite

## Projection Taxonomy

Terminology in  
italics is optional

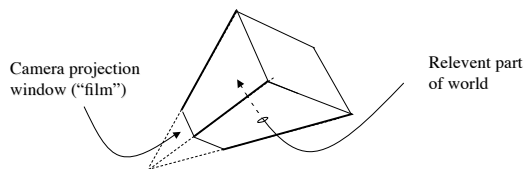


## Specifying a camera

- Makes sense to tell rendering system where camera is in world coordinates
- We want to transform the world into camera coordinates so that projection is easy
  - In particular, we want to use the projection matrix from a few slides back.
- Need to specify focal point and film plane.
- Convenient to construct a coordinate system for the camera with origin on film plane

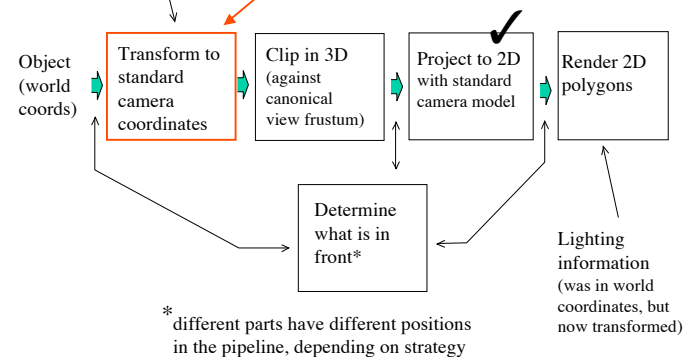
## Clipping volume

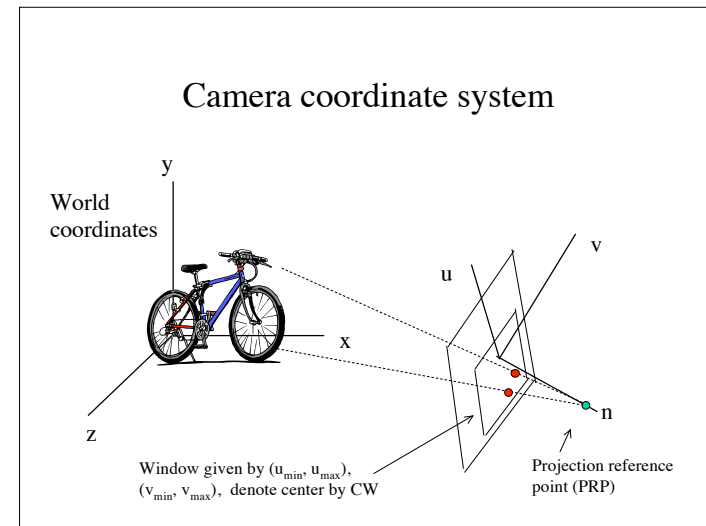
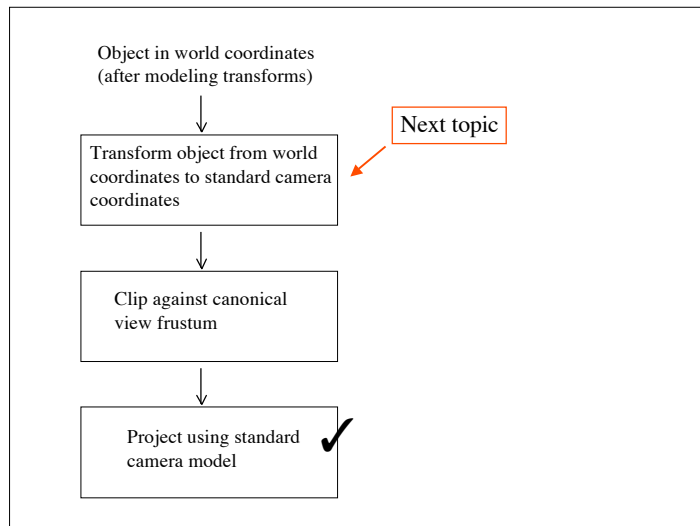
- We render only a window in the film plane
- Things beyond any of four sides don't get rendered
- Things that are too far away don't get rendered
- Things that are too near don't get rendered
- Taken together, this means that everything outside a truncated pyramid (frustum) is irrelevant.



Camera information  
(world coordinates)

Next topic





### Specifying a camera

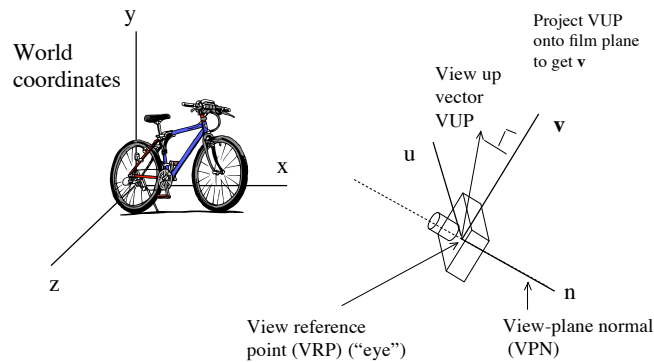
- We link camera specifications that relate to the user or the application to the  $(\mathbf{u}, \mathbf{v}, \mathbf{n})$  coordinate system
- The picture on the previous page is the most general case. Often one assumes that  $\mathbf{n}$  goes through the center of the window (CW).

### Specifying a camera

- There are many ways for the user to specify the camera. OpenGL has several. We will study one common and flexible one based on world coordinate entities “VUP” and “VPN”
- Our treatment is very similar to H&S (§7.3), but note that they call VUP,  $\mathbf{V}$



## Specifying a camera



## Specifying a camera

- Why use VUP?
  - Convenient for the user but there are other ways (OpenGL has several ways to negotiate camera parameters, including one which is very much how we are doing it).
  - A world centric coordinate system is natural for the user. In particular, the user may think in terms of the camera rotation around the axis ( $n$ ) relative to a natural horizon and/or "up" direction.
  - This will mean that VUP cannot be parallel to  $n$ . Often one will fix VUP (e.g. to the Y-axis) but this is too restrictive for some applications.
- Why use a "backwards" pointing  $n$ ?
  - It is more natural to make the camera direction point the other way, but this makes the camera coordinates left handed. (You will see it done both ways).

## Specifying a camera

- Together, the view reference point, VRP, and view plane normal,  $VPN=n$ , specify image plane.
- The up vector, VUP, gives an "up" direction in the image plane, providing for a user twist of camera about  $n$ .
- The camera coordinate system axis,  $v$ , is the projection of the up vector, VUP, into image plane.

## Specifying a camera

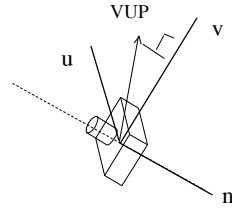
- We construct  $(u, v, n)$  so that it is a right handed coordinate system.
- This means that it is possible to map the world coordinates  $(x, y, z)$  to  $(u, v, n)$  so that  $(x \rightarrow u, y \rightarrow v, z \rightarrow n)$  using only translations and rotations.

## Computing (u,v,n) in world coordinates

$\mathbf{v}$  is the projection of VUP into the view plane which is perpendicular to  $\mathbf{n}$

$\mathbf{u}$  is perpendicular to the plane formed by  $\mathbf{n}$  and VUP

So, we can easily compute a vector parallel to  $\mathbf{u}$ .

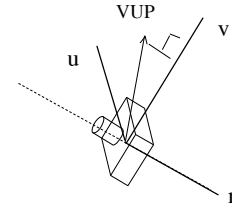


## Computing (u,v,n) in world coordinates

$$\mathbf{u} \parallel \mathbf{VUP} \times \mathbf{n}$$

$$\mathbf{u} = \frac{\mathbf{VUP} \times \mathbf{n}}{|\mathbf{VUP} \times \mathbf{n}|} = \frac{\mathbf{VUP} \times \mathbf{n}}{|\mathbf{VUP}|}$$

What about  $\mathbf{v}$ ?

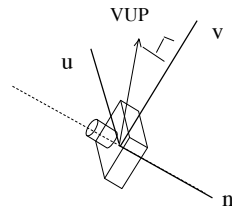


## Computing (u,v,n) in world coordinates

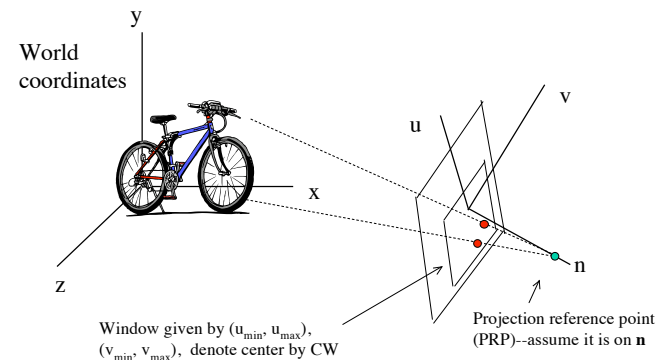
$$\mathbf{u} \parallel \mathbf{VUP} \times \mathbf{n}$$

$$\mathbf{u} = \frac{\mathbf{VUP} \times \mathbf{n}}{|\mathbf{VUP} \times \mathbf{n}|} = \frac{\mathbf{VUP} \times \mathbf{n}}{|\mathbf{VUP}|}$$

$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$



## Specifying a camera



- VRP, VPN, VUP must be in world coords;
- PRP (focal point) could be in world coords, but more commonly, camera coords (which are the same scale as world coords)
- We will use camera coords, and further assume that  $\text{PRP} = (0,0,f)$ .