3D Viewing & Clipping

Where do geometries come from?
Pin-hole camera
Perspective projection
Viewing transformation

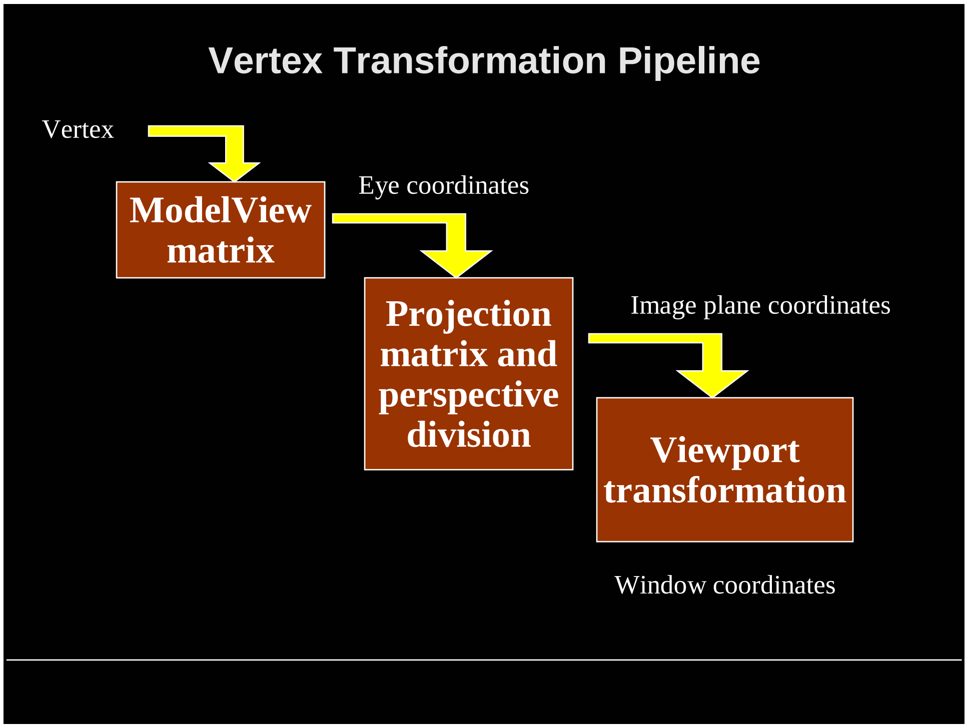
Clipping lines & polygons

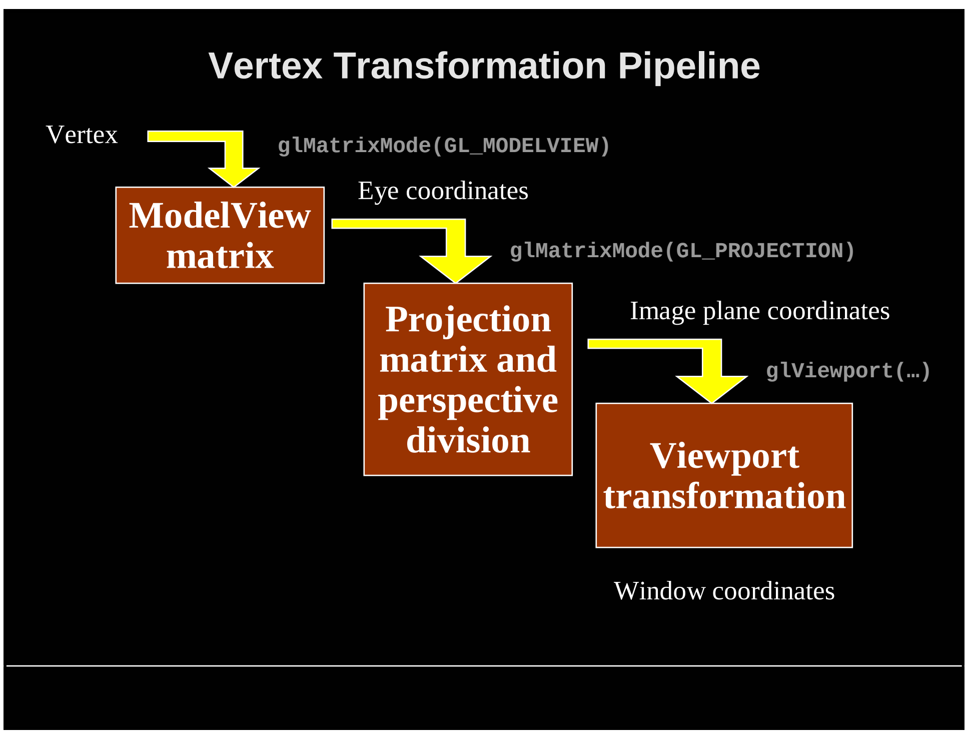
Angel Chapter 5

Getting Geometry on the Screen

Given geometry in the world coordinate system, how do we get it to the display?

- Transform to camera coordinate system
- Transform (warp) into canonical view volume
- Clip
- Project to display coordinates
- (Rasterize)


Vertex Transformation Pipeline
Vertex
ModelView matrix
Eye coordinates
Projection matrix and perspective division
Image plane coordinates
Viewport transformation
Window coordinates


Vertex Transformation Pipeline
Vertex
glMatrixMode(GL_MODELVIEW)
ModelView matrix
Eye coordinates
glMatrixMode(GL_PROJECTION)
Projection matrix and perspective division
Image plane coordinates
glViewport(…)
Viewport transformation
Window coordinates

OpenGL Transformation Overview

```
glMatrixMode(GL_MODELVIEW)
  gluLookAt(…)

glMatrixMode(GL_PROJECTION)
  glFrustrum(…)
  gluPerspective(…)
  glOrtho(…)

glViewport(x,y,width,height)
```

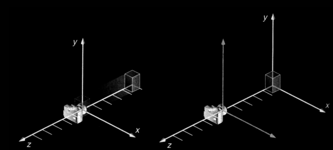


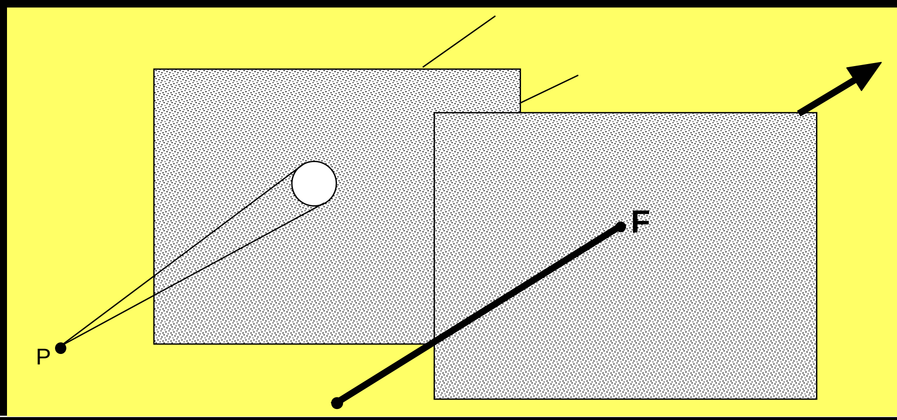
Figure 3-10 Separating the Viewpoint and the Object

Viewing and Projection

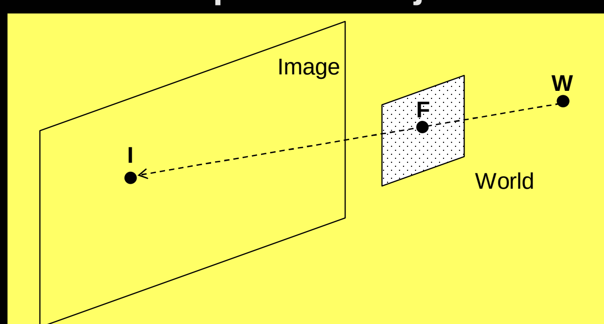
- Our eyes collapse 3-D world to 2-D retinal image (brain then has to reconstruct 3D)
- In CG, this process occurs by *projection*
- Projection has two parts:
  - *Viewing transformations:* camera position and direction
  - *Perspective/orthographic transformation:* reduces 3-D to 2-D
- Use homogeneous transformations
- As you learned in Assignment 1, camera can be animated by changing these transformations—the root of the hierarchy

Pinhole Optics

- Stand at point P, and look through the hole - anything within the cone is visible, and nothing else is
- Reduce the hole to a point - the cone becomes a *ray*
- Pin hole is the *focal point, eye point* or *center of projection.*

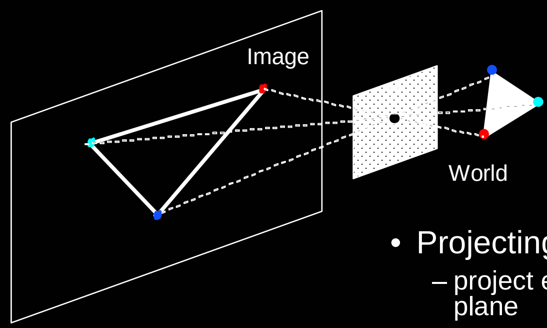


Perspective Projection of a Point

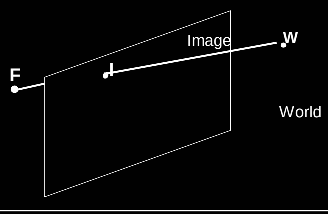


- *View plane* or *image plane* - a plane behind the pinhole on which the image is formed
  - point *I* sees anything on the line (ray) through the pinhole *F*
  - a point *W* projects along the ray through *F* to appear at *I* (intersection of WF with image plane)

Image Formation



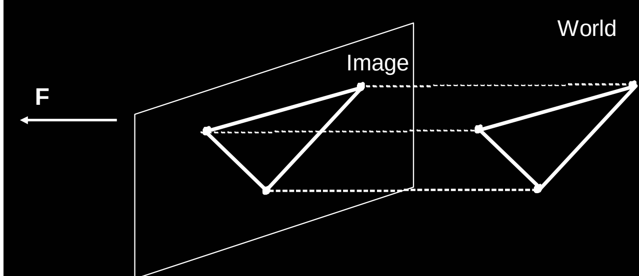
- Projecting a shape
  - project each point onto the image plane
  - lines are projected by projecting end points only

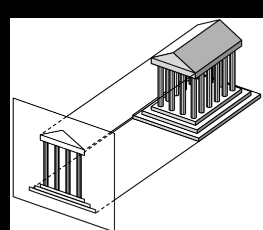


Note: Since we don't want the image to be inverted, from now on we'll put F behind the image plane.

Orthographic Projection

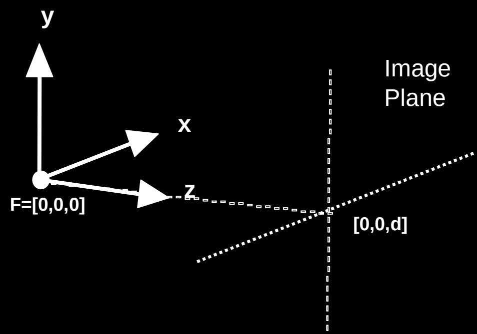
- when the focal point is at infinity the rays are parallel and orthogonal to the image plane
- good model for telephoto lens. No perspective effects.
- when *xy*-plane is the image plane (x,y,z) -> (x,y,0) front orthographic view

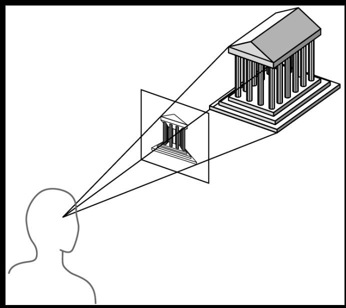




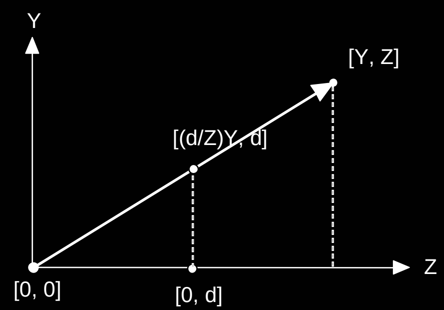
A Simple Perspective Camera

- Canonical case:
  - camera looks along the *z*-axis
  - focal point is the origin
  - image plane is parallel to the *xy*-plane at distance *d*
  - (We call d the focal length, mainly for historical reasons)





Similar Triangles



- Diagram shows *y*-coordinate, *x*-coordinate is similar
- Using similar triangles
  - point [x,y,z] projects to [(d/z)x, (d/z)y, d]

A Perspective Projection Matrix

- Projection using homogeneous coordinates:
  - transform [x, y, z] to [(d/z)x, (d/z)y, d]

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx & dy & dz & z \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{d}{z}x & \frac{d}{z}y & \mathrm{d} \end{bmatrix}$$

Divide by 4th coordinate
(the "w" coordinate)

- 2-D image point:
  - discard third coordinate
  - apply viewport transformation to obtain physical pixel coordinates

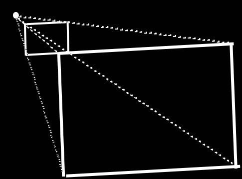
Wait, there's more!

We have just seen how to project the entire world onto the image plane..

How can we limit the portions of the scene that are considered?

The View Volume

- Pyramid in space defined by focal point and window in the image plane (assume window mapped to viewport)
- Defines visible region of space
- Pyramid edges are clipping planes
- *Frustum* = truncated pyramid with near and far clipping planes
  - Why near plane? Prevent points behind the camera being seen
  - Why far plane? Allows *z* to be scaled to a limited fixed-point value (*z*-buffering)



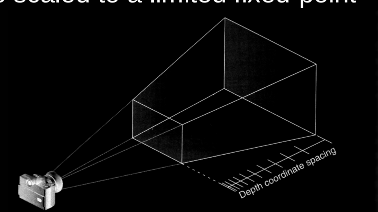


Figure 3-18 Perspective Projection and Transformed Depth Coordinates

But wait...

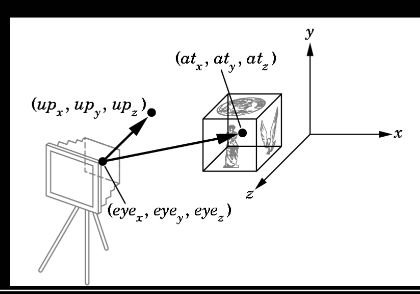
- What if we want the camera somewhere other than the canonical location?
- Alternative #1: derive a general projection matrix. (*hard*)
- Alternative #2: transform the world so that the camera is in canonical position and orientation (*much simpler*)
- These transformations are *viewing transformations*

Camera Control Values

- All we need is a single translation and angle-axis rotation (orientation), but...
- Good animation requires good camera control--we need better control knobs
- Translation knob - move to the *lookfrom* point
- Orientation can be specified in several ways:
  - specify camera rotations
  - specify a *lookat* point (solve for camera rotations)

A Popular View Specification Approach

- Focal length, image size/shape and clipping planes are in the perspective transformation
- In addition:
  - *lookfrom:* where the focal point (camera) is
  - *lookat:* the world point to be centered in the image
- Also specify camera orientation about the *lookat-lookfrom* axis



Implementation

Implementing the *lookat/lookfrom/vup* viewing scheme

(1) Translate by *-lookfrom*, bring focal point to origin

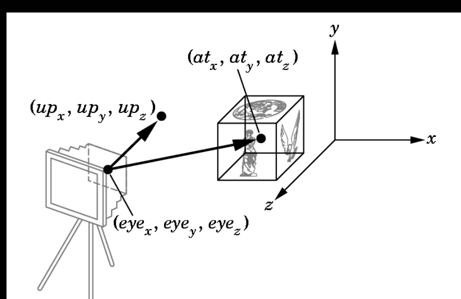
(2) Rotate *lookat-lookfrom* to the *z*-axis with matrix R:

- » v = (*lookat-lookfrom*) (normalized) and z = [0,0,1]
- » rotation axis: a = (v x z)/|v x z|
- » rotation angle: cosθ = v•z and sinθ = |v x z|

glRotate(q, a_x, a_y, a_z)

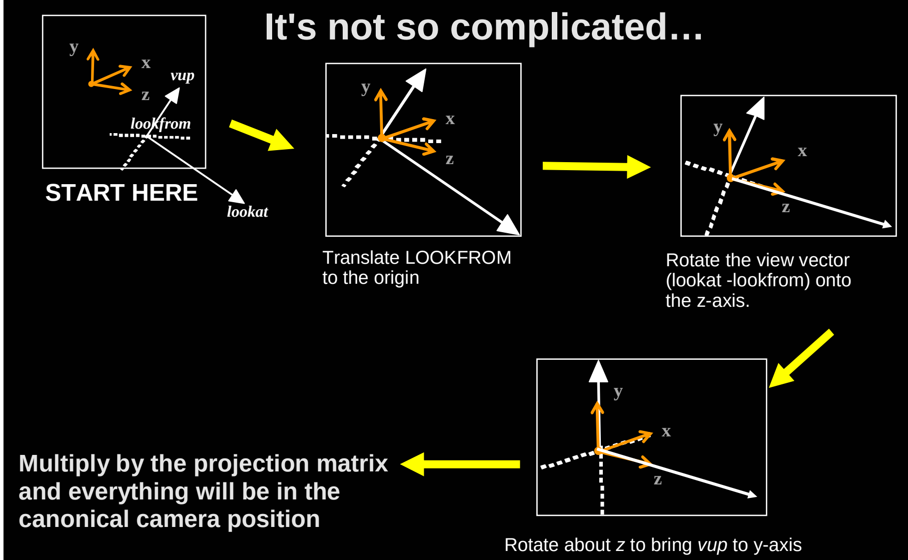
(3) Rotate about *z*-axis to get *vup* parallel to the y-axis

The Whole Picture



LOOKFROM:	Where the camera is
LOOKAT:	A point that should be centered in the image
VUP:	A vector that will be pointing straight up in the image
FOV:	Field-of-view angle.
d:	focal length
WORLD COORDINATES	

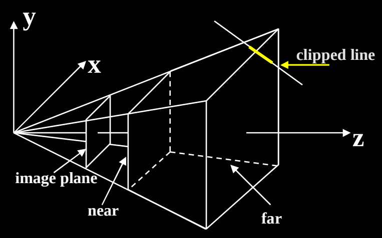
It's not so complicated…



One and Two-Point Perspective?

Clipping

- There is something missing between projection and viewing...
- Before projecting, we need to eliminate the portion of scene that is outside the viewing frustum

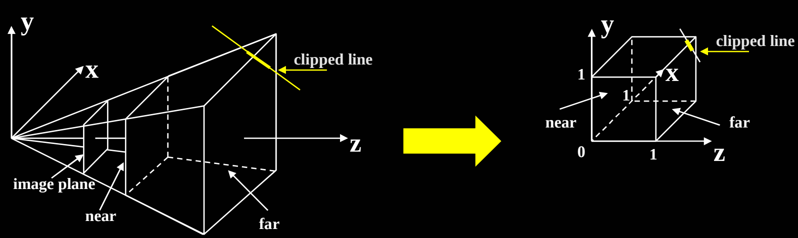


•Need to clip objects to the frustum (truncated pyramid)

•Now in a canonical position but it still seems kind of tricky...

Normalizing the Viewing Frustum

- Solution: transform frustum to a cube before clipping



- Converts perspective frustum to orthographic frustum
- This is yet another homogeneous transform!

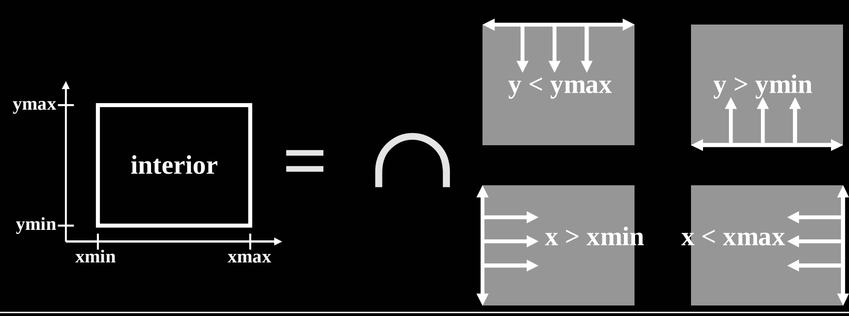
Clipping to a Cube

- Determine which parts of the scene lie within cube
- We will consider the 2D version: clip to rectangle
- This has its own uses (viewport clipping)
- Two approaches:
  - clip during scan conversion (rasterization) - check per pixel or end-point
  - clip before scan conversion

- We will cover
  - clip to rectangular viewport before scan conversion

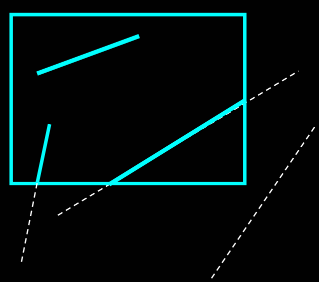
Line Clipping

- Modify endpoints of lines to lie in rectangle
- How to define "interior" of rectangle?
- Convenient definition: intersection of 4 half-planes
  - Nice way to decompose the problem
  - Generalizes easily to 3D (intersection of 6 half-planes)



Line Clipping

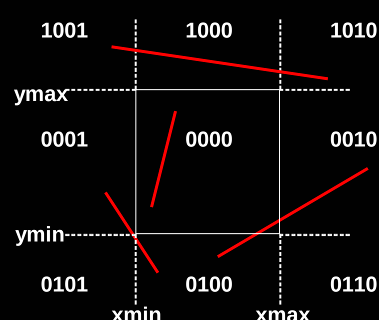
- Modify end points of lines to lie in rectangle
- Method:
  - Is end-point inside the clip region? - half-plane tests
  - If outside, calculate intersection between the line and the clipping rectangle and make this the new end point



- Both endpoints inside: trivial accept
- One inside: find intersection and clip
- Both outside: either clip or reject (tricky case)

Cohen-Sutherland Algorithm

- Uses *outcodes* to encode the half-plane tests results

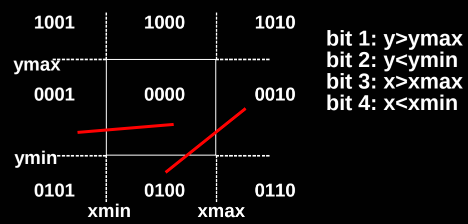


bit 1: y>ymax
bit 2: y<ymin
bit 3: x>xmax
bit 4: x<xmin

- **Rules:**
  - **Trivial accept: outcode(end1) and outcode(end2) both *zero***
  - **Trivial reject: outcode(end1) & (bitwise and) outcode(end2) *nonzero***
    - **Else subdivide**

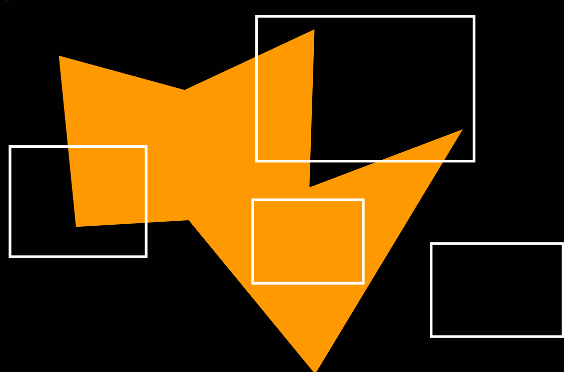
Cohen-Sutherland Algorithm: Subdivision

- If neither trivial accept nor reject:
  - Pick an outside endpoint (with nonzero outcode)
  - Pick an edge that is crossed (nonzero bit of outcode)
  - Find line's intersection with that edge
  - Replace outside endpoint with intersection point
  - Repeat until trivial accept or reject



Polygon Clipping

Convert a polygon into one *or more* polygons that form the intersection of the original with the clip window



Sutherland-Hodgman
Polygon Clipping Algorithm

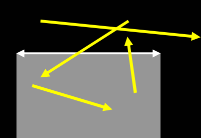
- Subproblem:
  - clip a polygon (vertex list) against a single clip plane
  - output the vertex list(s) for the resulting clipped polygon(s)
- Clip against all four planes
  - generalizes to 3D (6 planes)
  - generalizes to any convex clip polygon/polyhedron

Sutherland-Hodgman
Polygon Clipping Algorithm (Cont.)

To clip vertex list against one half-plane:

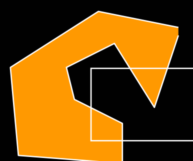
- if first vertex is inside - output it
- loop through list testing inside/outside transition - output depends on transition:

> in-to-in: **output vertex**
> out-to-out: **no output**
> in-to-out: **output intersection**
> out-to-in: **output intersection and vertex**

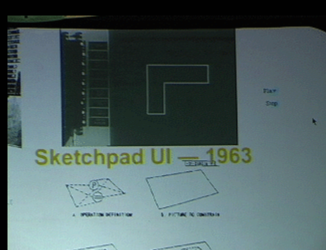


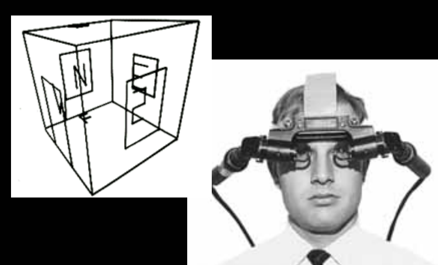
Cleaning Up

- Post-processing is required when clipping creates multiple polygons
- As external vertices are clipped away, one is left with edges running along the boundary of the clip region.
- Sometimes those edges dead-end, hitting a vertex on the boundary and doubling back
  - Need to prune back those edges
- Sometimes the edges form infinitely-thin bridges between polygons
  - Need to cut those polygons apart



Ivan Sutherland





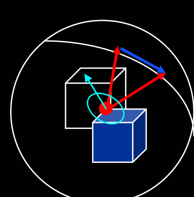
Beyond Linear Perspective...

Announcements

Written assignment #1 due next Thursday

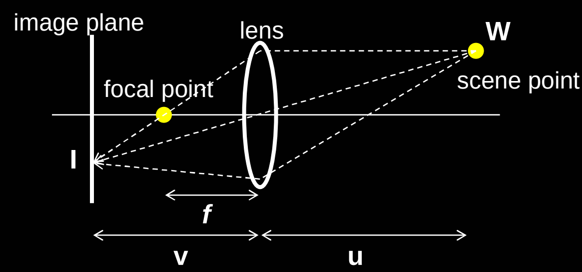
Virtual Trackballs

- Imagine world contained in crystal ball, rotates about center
- Spin the ball (and the world) with the mouse
- Given old and new mouse positions
  - project screen points onto the sphere surface
  - rotation axis is normal to plane of points and sphere center
  - angle is the angle between the radii
- There are other methods to map screen coordinates to rotations



Problems with Pinholes

- Correct optics requires infinitely small pinhole
  - No light gets through
  - Diffraction
- Solution: Lens with finite aperture



Lens Law: $$\frac{1}{u} + \frac{1}{v} = \frac{1}{f}$$