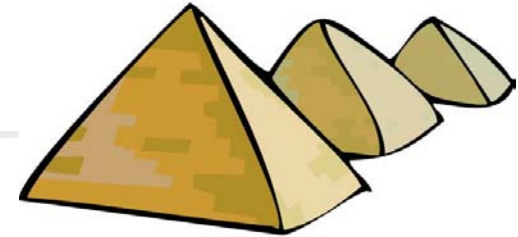


Visible-Surface Detection Methods



Chapter 9
Intro. to Computer Graphics
Spring 2009, Y. G. Shin

The Visibility Problem

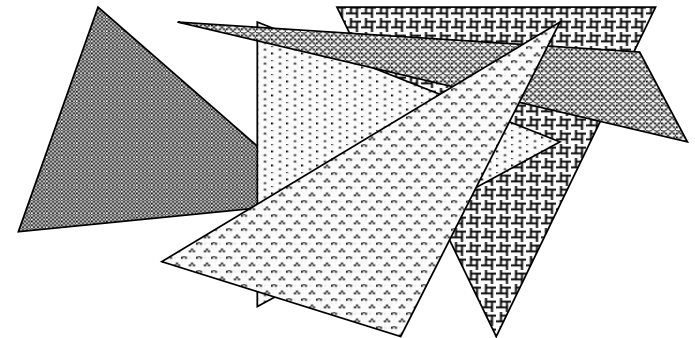


[Problem Statement]

GIVEN: a set of 3-D surfaces, a projection from 3-D to 2-D screen,

DETERMINE: the nearest surface encountered at any point on 2-D screen

- Remove of hidden parts of an object
 - Hidden-surface removal : surface rendering
 - Hidden-line removal : line drawing





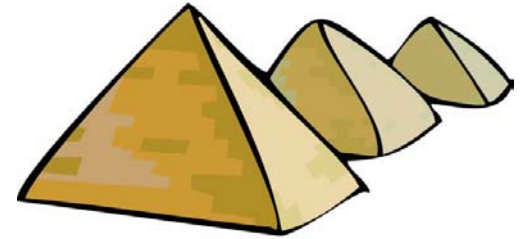
Techniques



- Visible-surface algorithms are 3D versions of sorting, i.e., depth comparison
- Avoid comparing all pairs of objects using the following coherence:
 - object coherence: no comparison between components of objects if objects are separated each other
 - face coherence: surface properties vary smoothly across a face
 - edge coherence: an edge changes its visibility not frequently



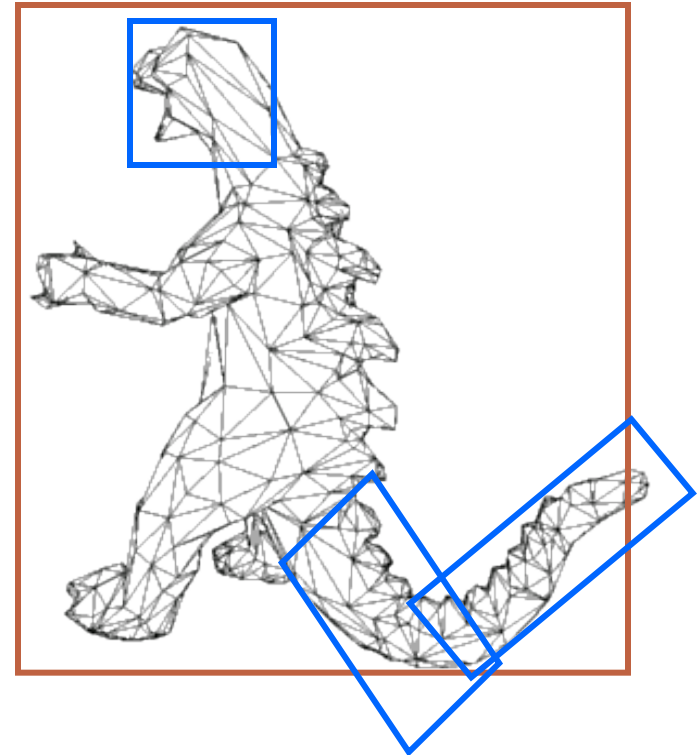
Techniques



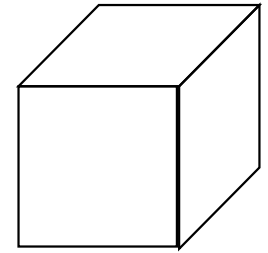
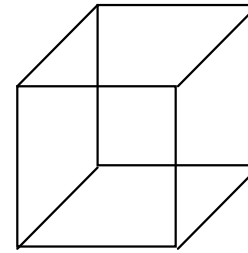
- implied edge coherence: line of intersection of two face can be determined from two intersection points
- scan-line coherence: little change in visible spans from one scanline to another
- area coherence: a group of pixels is often covered by the same visible surface
- span coherence: (special case of area coherence) homogeneous runs in a scanline
- depth coherence: adjacent parts of the same surface are typically close in depth
- frame coherence: animation frames contain small changes from the previous frame

Techniques for Efficient Algorithms

- Bounding volumes
 - approximate complex objects with simple enclosures before making comparisons.
 - the simplest approximate enclosure is a boundary box

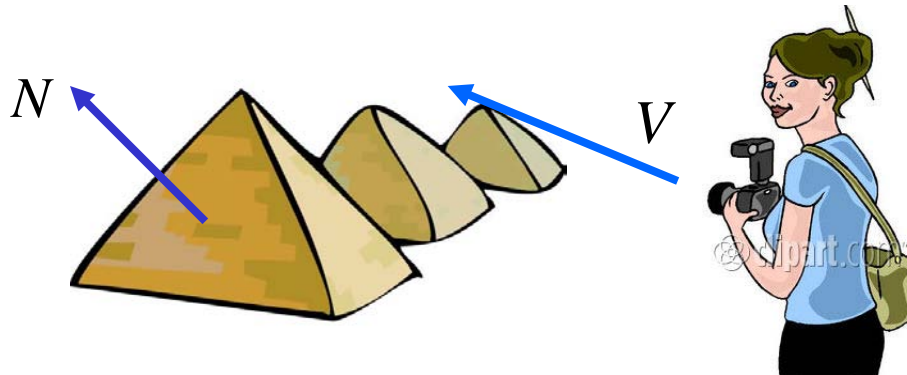


Backface Removal (Backface Culling)



- Remove entire polygons that face away from the viewer
- If we are dealing with a single convex object, culling completely solves the hidden surface problem

Geometric test for the visibility

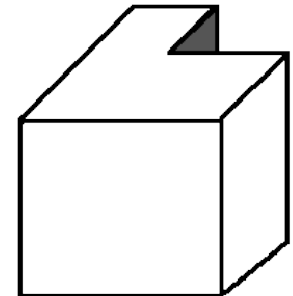


$V \cdot N < 0$: visible surface

$V \cdot N = 0$: silhouettes

Back-face Culling More..

- Vertex order in surface normal calculation
 - ← counterclockwise in the right-handed viewing system
- Backface culling after viewing transformation
 - simpler culling test (consider only z component of normal vectors since COP is at infinity after view-volume normalization)
 - more points to transform
- Partially hidden faces cannot be determined by back-face culling
- Not useful for ray-casting, radiosity



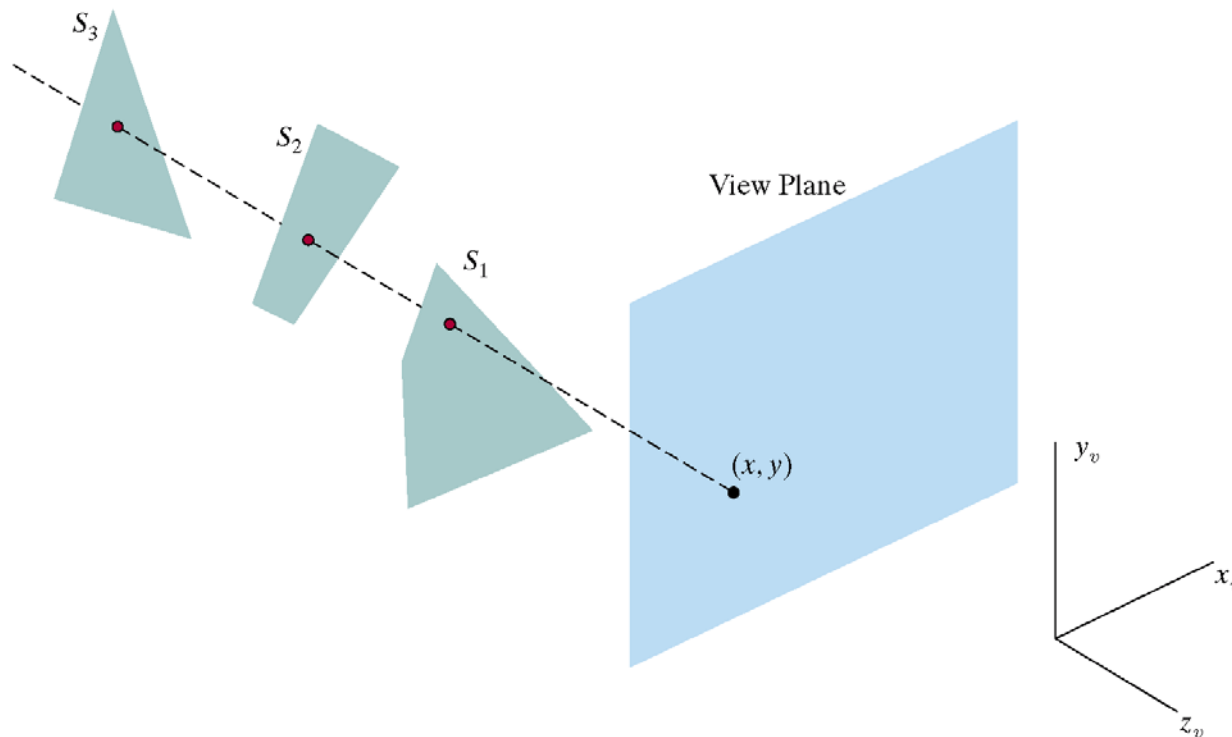


Depth-Buffer (Z-buffer)

- The basic idea is to test the z-depth of each surface to determine the closest (visible) surface
- Use two buffers
 - frame buffer : image value (intensity)
 - depth buffer (z-buffer) : z-value
- Algorithm
 1. **Initialize the depth buffer and frame buffer**
Depthbuffer(x,y)=1.0;
Framebuffer(x,y)=background;
 2. **Process each polygon in a scene, one at a time**
loop on y within y range of this object
loop on x within x range of this scan line of this object
if $z(x,y) < \text{depthbuffer}(x,y)$ then
{depthbuffer(x,y) = z(x,y);
framebuffer(x,y) = surface_color(x,y);}

Depth-Buffer (Z-Buffer)

- Z-Buffer has memory corresponding to each pixel location
 - Usually, 16 to 20 bits/location.





Z-Buffer Algorithm

Calculating z values of plane: $Ax + By + Cz + D = 0$

[WAY I]
$$z = \frac{-Ax - By - D}{C}$$

[WAY II] Use incremental calculation

$z_{(x,y)}$: the depth of position (x, y)

$$z_{(x+1,y)} = \frac{-A(x+1) - By - D}{C} = z_{(x,y)} - \frac{A}{C}$$

$$z_{(x,y+1)} = \frac{-Ax - B(y+1) - D}{C} = z_{(x,y)} - \frac{B}{C}$$

[WAY III] Use interpolation.



Z-Buffer Algorithm

- *pros:*

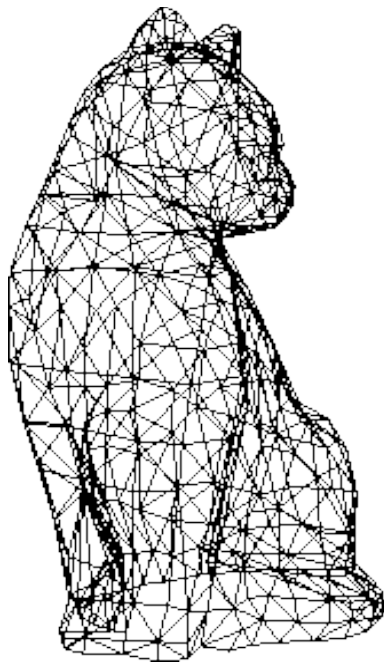
- easy implementation (directly in hardware)
- no sorting of surfaces
- $O(\# \text{ of objects} \times \text{pixels})$
- good rendering algorithm with polygon models

- *cons:*

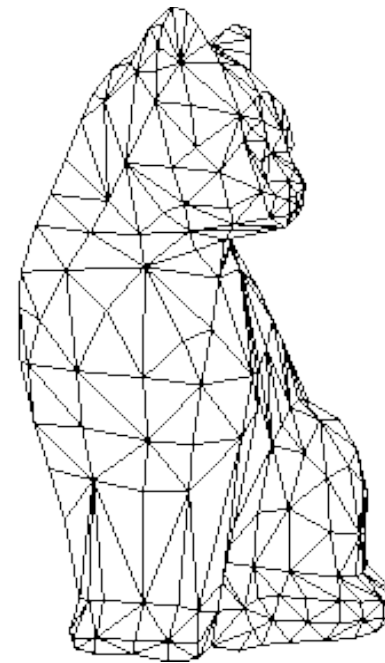
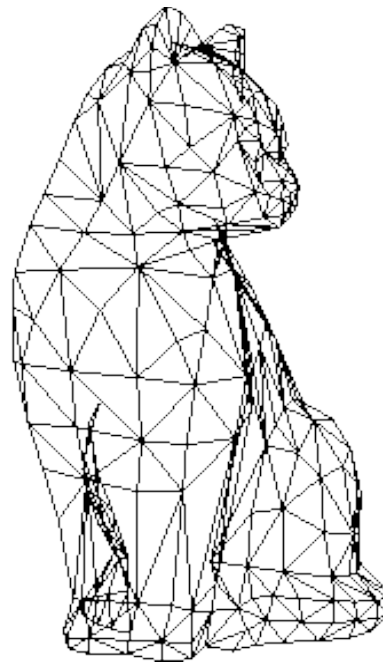
- additional buffer (z-buffer)
- aliasing (point-sampling)
- difficult to deal with transparent object



Z-Buffer Algorithm



Backface culling



Z-buffer algorithm

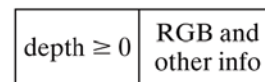


Accumulation Buffer (A-Buffer)

- An extension of the depth-buffer for dealing with anti-aliasing, area-averaging, transparency, and translucency
 - The depth-buffer method identifies only one visible surface at each pixel position : Cannot accumulate color values for more than one transparent and translucent surfaces
- The same resolution as z-buffer
- Part of OpenGL and DirectX
- Costly for real-time rendering
- Widely used for high quality rendering

Accumulation Buffer (A-Buffer)

- Each position in the A-buffer has two fields
 - Depth field: Stores a depth value
 - Surface data field:
 - RGB intensity components
 - Opacity parameter
 - Depth
 - Percent of area coverage
 - Surface identifier



(a)



(b)

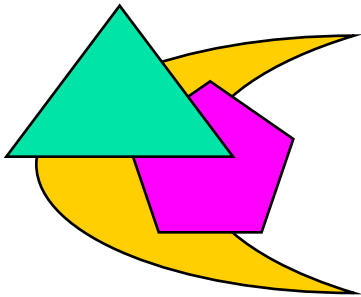
Figure 9-9

Two possible organizations for surface information in an A-buffer representation for a pixel position. When a single surface overlaps the pixel, the surface depth, color, and other information are stored as in (a). When more than one surface overlaps the pixel, a linked list of surface data is stored as in (b).



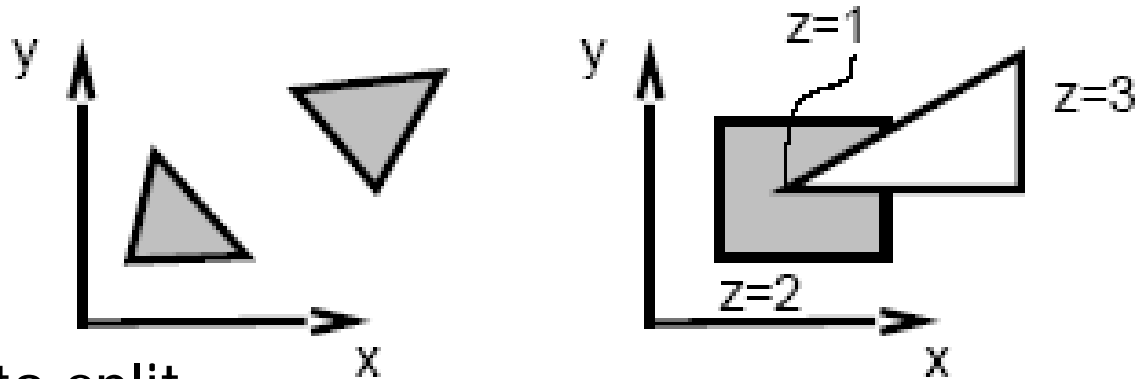
Painter's Algorithm

- Draw polygons as an oil painter might: The farthest one first. (Used in PostScript)
- [Algorithm]
 - sort objects by depth, splitting if necessary to handle intersections
 - loop on objects (drawing from back to front)
 - loop on y within y range of this object
 - loop on x within x range of this scan line of this object
 - $image[x,y] = shade(x,y)$

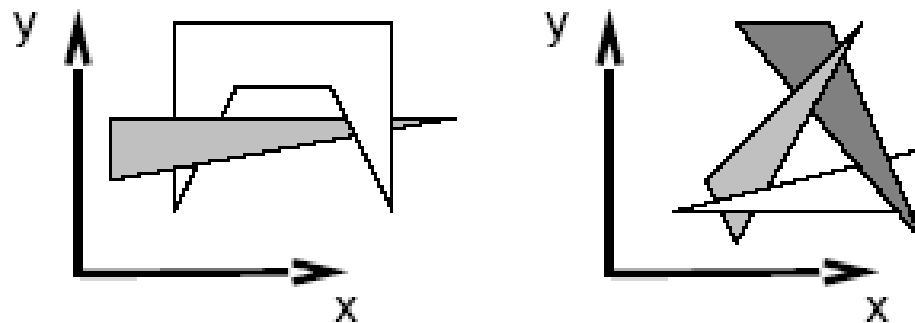


Painter's Algorithm (z-overlap case)

- Easy case of depth comparison

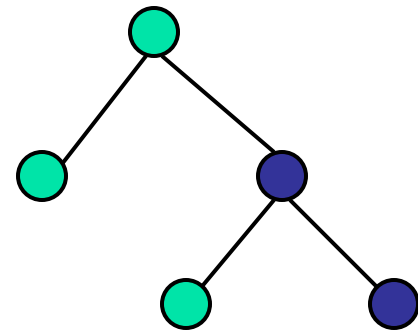
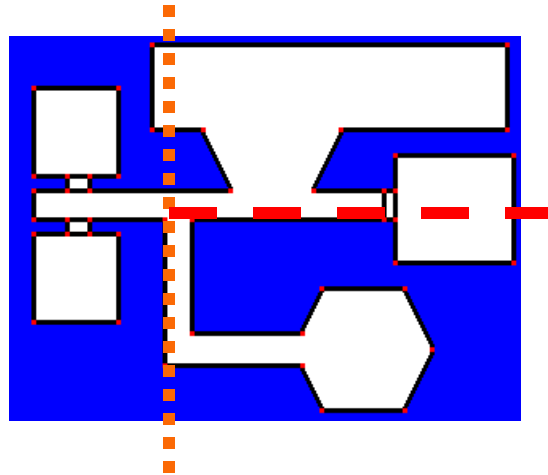


- Need to split

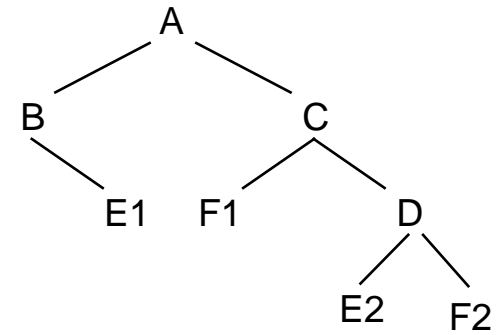
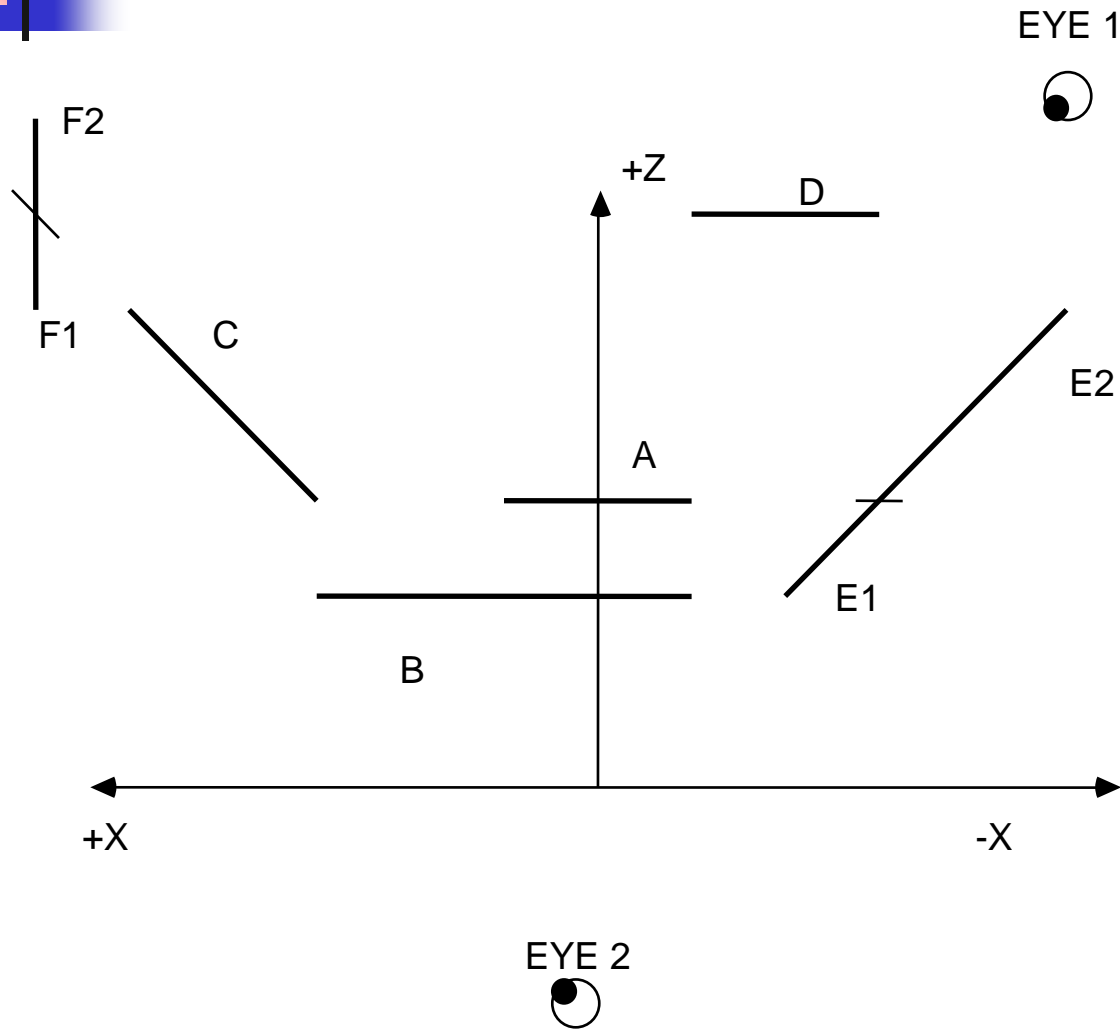


Binary Space-Partitioning Trees (BSP trees)

- Binary Space Partitioning is a relatively easy way to sort the polygons relative to the eyepoint
- Fuchs, Kedem, and Naylor
- The scan conversion order is decided by building a binary tree of polygons, the BSP tree.
- Fast traversal and viewpoint independent order
- Clusters that are on the same side of the plane as the eye-point can obscure clusters on the other side.



BSP tree





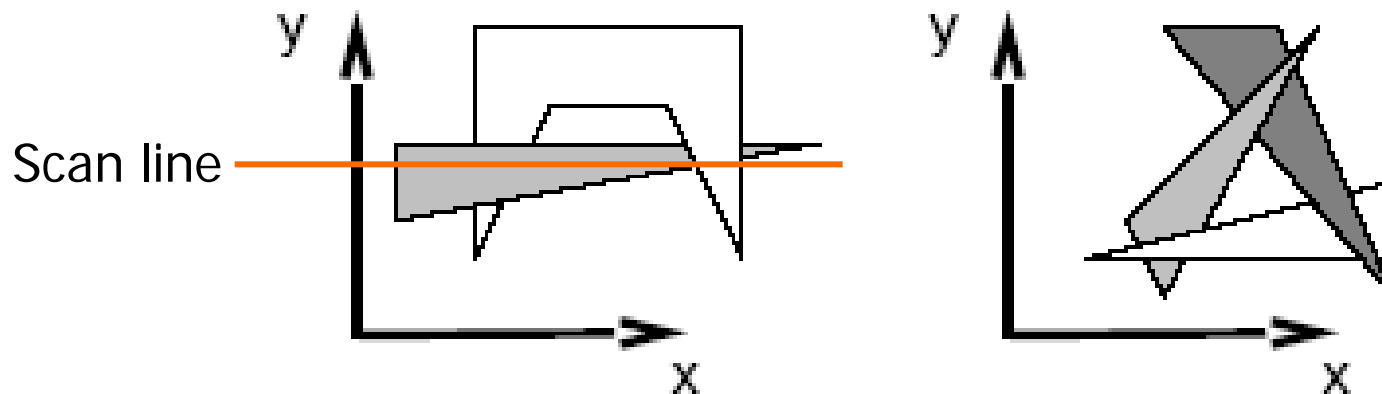
BSP trees

- Disadvantages

- significantly more than input polygons - more polygon splitting may occur than in Painter's algorithm
- appropriate partitioning hyperplane selection is quite complicated and difficult (ref. Chen in SigGraph96)

Scan-Line Method

- An extension of scan-line polygon filling (multiple surfaces)
- Idea is to intersect each polygon with a particular scanline. Solve hidden surface problem for just that scan line.
- Requires a depth buffer equal to only one scan line
- The cost of tiling scene is roughly proportional to its depth complexity
- Efficient way to tile shallowly-occluded scenes
- May need to split



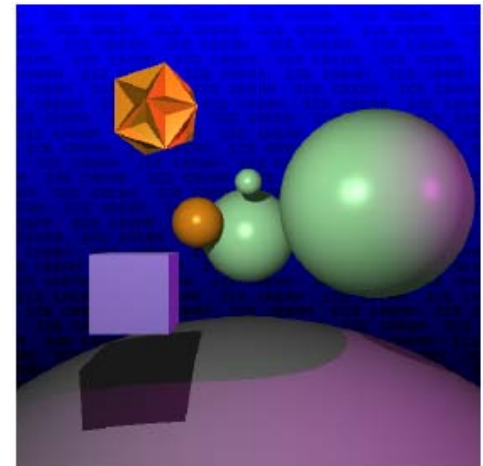
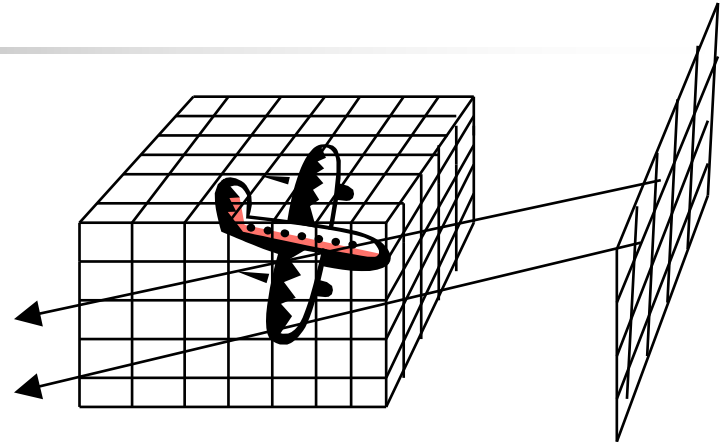
Ray Casting

- rendering + visibility
- ALGORITHM

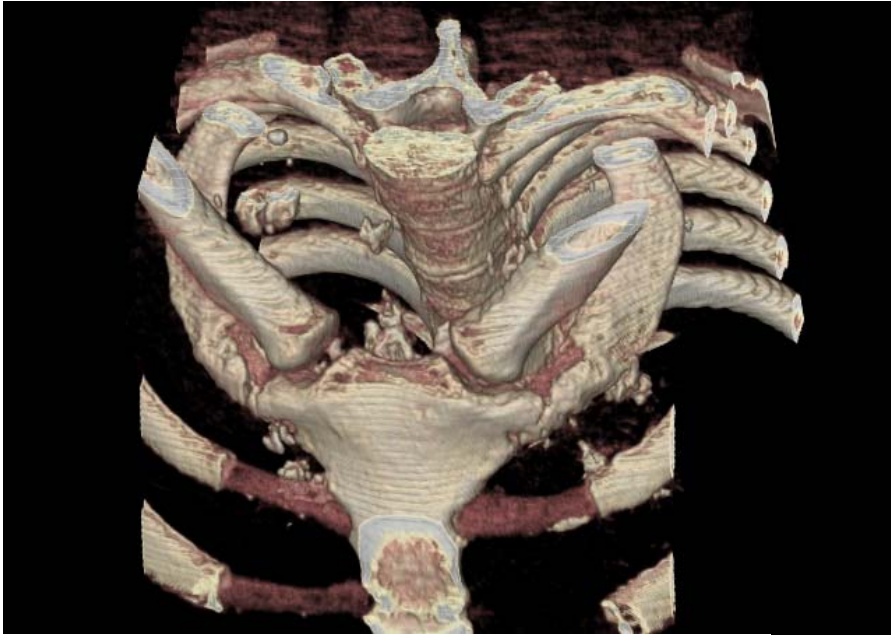
loop y

loop x

shoot ray from eye point through
pixel (x,y) into scene;
intersect with all surfaces, find
first one the ray hits;
shade that point to compute the
color of pixel (x,y) ;



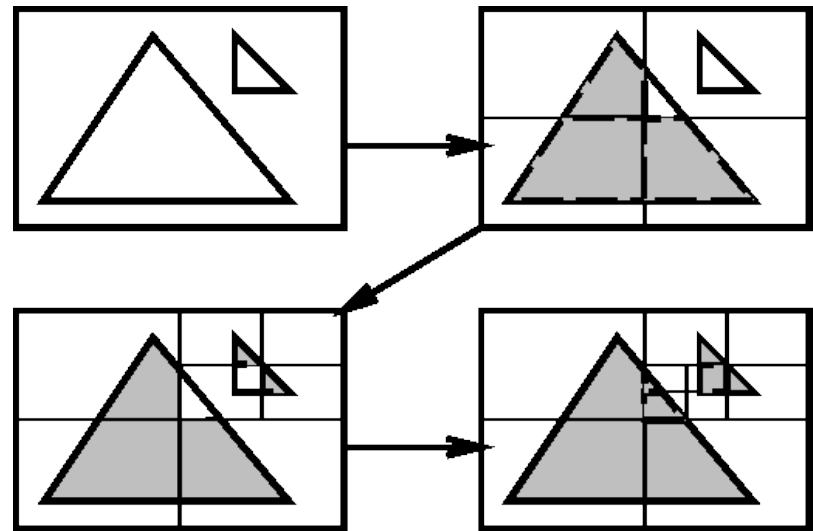
Ray Casting



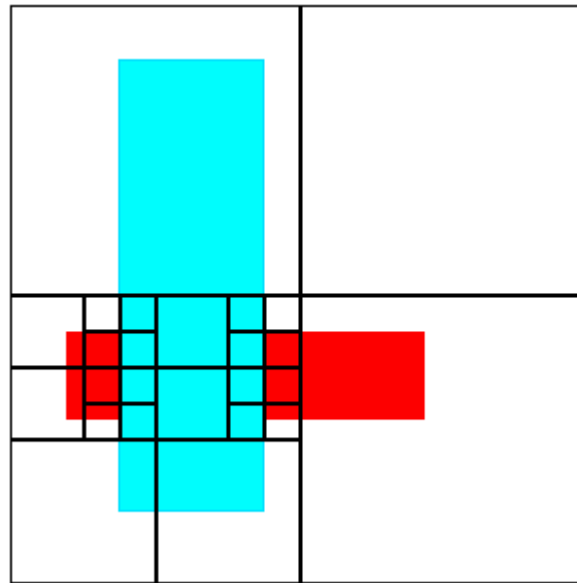
Warnock's algorithm

- Area Subdivision Algorithms
 - ✓ image space algorithms
 - ✓ divide-and-conquer : area coherence

1. Take a given section of the screen (the entire screen, in the first pass)
2. If it is "simple enough", display it;
3. If it is NOT simple enough, subdivide the screen into four sections and check each of the new sections and check each of the new sections (starting at step 1);



Warnock's algorithm



- Runtime: $O(p \times n)$

p : number of pixels

n : number of polygons



Comparisons of Hidden-Surface Algorithms

- **Z-buffer:**
 - memory: used for image buffer & z-buffer
 - implementation: moderate, requires scan conversion. It can be put in hardware.
 - speed: fast, unless depth complexity is high
 - generality: very good
- **Painter's:**
 - memory: used for image buffer
 - implementation: moderate, requires scan conversion; hard if sorting & splitting needed
 - speed: fast only *if objects can be sorted a priori*
 - generality: splitting of intersecting objects & sorting make it clumsy for general 3-D rendering



Comparisons of Hidden-Surface Algorithms

- **Ray casting:**
 - memory: used for object database
 - implementation: easy, but to make it fast you need spatial data structures
 - speed: slow if many objects: cost is $O((\#pixels) \cdot (\#objects))$
 - generality: excellent, can even do non-polygon models, shadows, transparency.
- **Others (scanline, object space):** tend to be hard to implement, and very hard to generalize to non-polygon models

Next Topics

How to get the color of a pixel.

