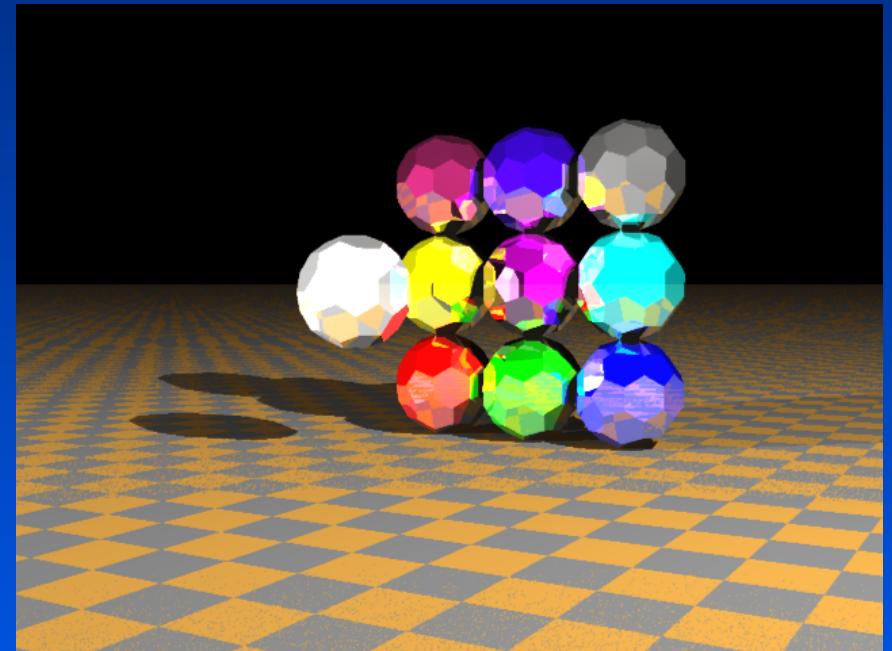


Visible Surface Algorithms

1. Back-face polygon culling
2. Z-buffer algorithms
3. 3-D depth-sort algorithm
4. Binary space partitioning trees
5. Visible-surface ray tracing



Visible Surface Determination

- Given a set of 3-D objects and a view specification, determine which lines or surfaces of the object are visible
 - From center of projection (perspective)
 - Along the direction of projection (parallel)
- Also called hidden surface removal
- Called visible-line determination when referring to lines only
- Note: lines themselves don't hide lines
Lines must be edges of opaque surfaces to hide other lines

Image Precision

- Determine which of M objects is visible at each N pixels in image
 - for each pixel in the image
 - determine object closest to viewer that is intersected by the projector through the pixel;
 - Draw the pixel in the appropriate color
- Performed at resolution of image
- Complexity: $O(N \times M)$

Object Precision

- Compare objects directly with each other
 - for each object in the world
 - Compare with all other object to determine visible part
 - Draw those parts in the appropriate color
- Performed at resolution of object
- Must be followed by scan conversion (usually image precision)
- Historically the first approach
- Complexity: $O(M^2)$

Costly operations

- Image precision :

Which objects intersect a projector?

- Object precision :

Which objects intersect each object?

Techniques for Efficient Visible Surface Algorithms♪

- Coherence: The degree to which parts of an environment exhibit logical similarities
Reuse previous calculations (e.g., depth of adjacent pixels on a plane is related by difference equation)
- Perspective transformation: Transform environment into one in which points with same (x,y) lie on same projector
Compare z's to determine the closest one
- Back-face culling: Remove surfaces that face away from point of view

- Extents and bounding volumes: Bound each complex object with a simpler one
Use to cull possible overlaps
- Spatial partitioning: Break large problem into smaller ones, e.g., partition screen into regular grid
Solve visible surface problem in each grid cell independently, only comparing with those in same grid
- Hierarchy: Build tree of nested extents
If two internal nodes don't intersect, none of their children intersect

Coherence

“local similarity”♪

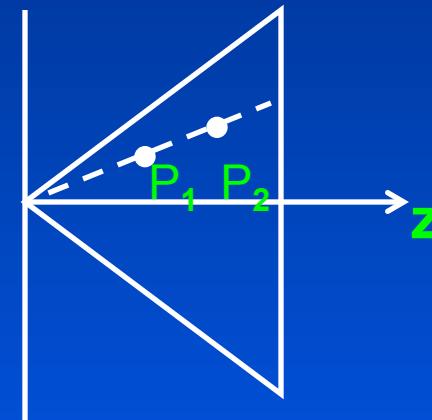
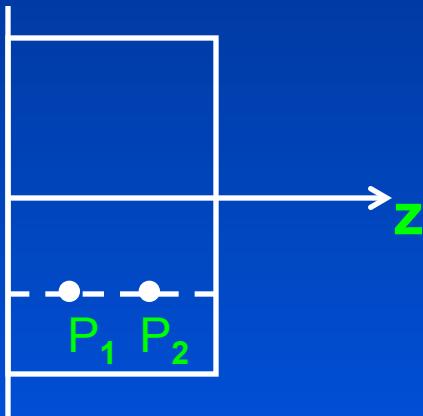
Use previous calculated value

- Object coherence
 - e.g. when 2 objects are entirely separated in x-y then need not consider polygons that make up the objects
- Face coherence
 - e.g. property at one part of face changes incrementally as you move across face
- Edge coherence
 - e.g. property along an edge changes incrementally (e.g. z)
- Implied edge coherence
 - e.g. edge formed from intersection of two faces exhibit edge coherence

- Area coherence
 - E.g. visibility at one pixel “similar” to visibility at neighboring pixel
- Depth coherence
 - E.g. depth (z) of one object similar
Depth (z) of one part of an object can be determined by depth of a neighboring part
- Temporal coherence
 - E.g. properties of one frame (at a point in time) similar to neighboring frame

Perspective transformation♪

- Hidden surface problem: given p_1 and p_2 , are p_1 and p_2 on the same projector?
 - Simple for parallel projection
 - Must divide by z for perspective projection



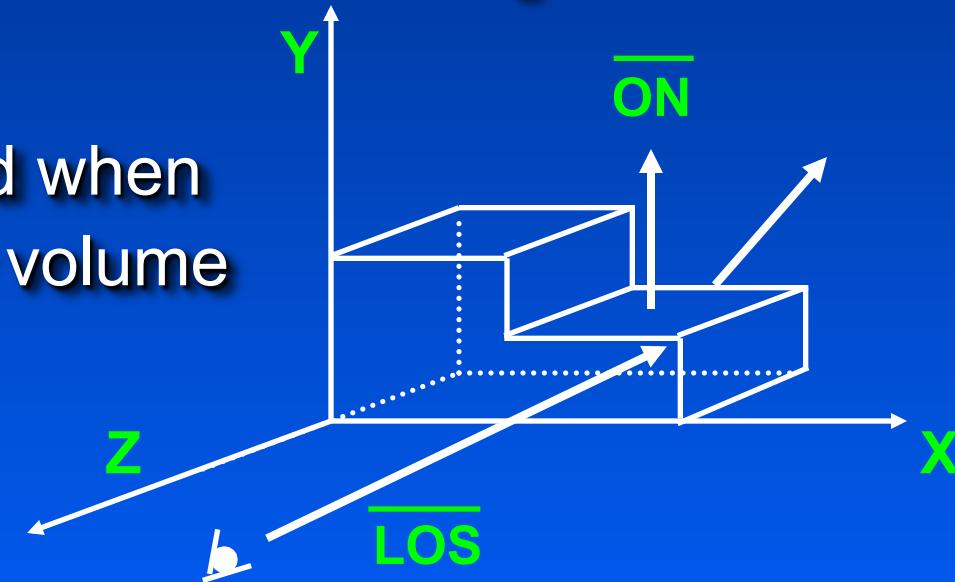
- After perspective transform, space transformed into parallel canonical view volume

Extents and Bounding volumes

- Cost of bounding box tests
- How well bounding box encloses object
 - e.g. compare bounding box of projected object on x-y plane
- If bounding box not a tight fit, may have many “false positives”

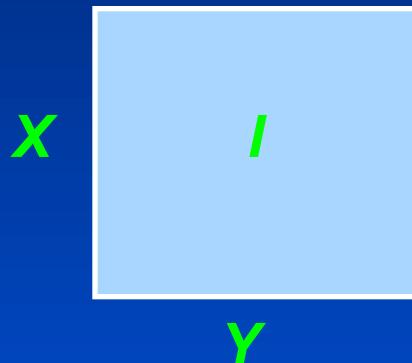
Back-Face Culling♪

- Assumes objects defined polyhedra
- When displaying a single convex polyhedron, back-face culling is the only visible surface algorithm needed!
- Dot product not needed when canonical parallel view volume is used

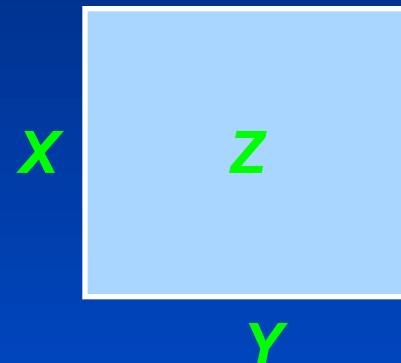


Z-Buffer Algorithm♪

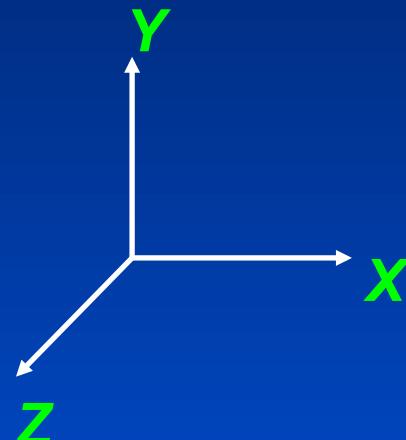
- Requires two buffer



Refresh buffer



Depth buffer



- Intensity buffer $I(x,y)$ initialized to background
- Depth buffer $Z(x,y)$ initialized to furthest z

- Polygons are scan-converted in arbitrary order
 1. Calculate the polygon depth $z(x,y)$
 2. If $z(x,y)$ closer than z-buffer at (x,y) , then
 - a) Place $z(x,y)$ into the z-buffer at (x,y) , $Z(x,y)$ and
 - b) Place pixel value of polygon at $z(x,y)$ into the refresh buffer at (x,y) , $I(x,y)$
- Used in most graphics hardware
- Image precision
- Speed tends not to explode with scene complexity
- Use difference equation to calculate z incrementally

Calculation of z

- Interpolate along z_1-z_2

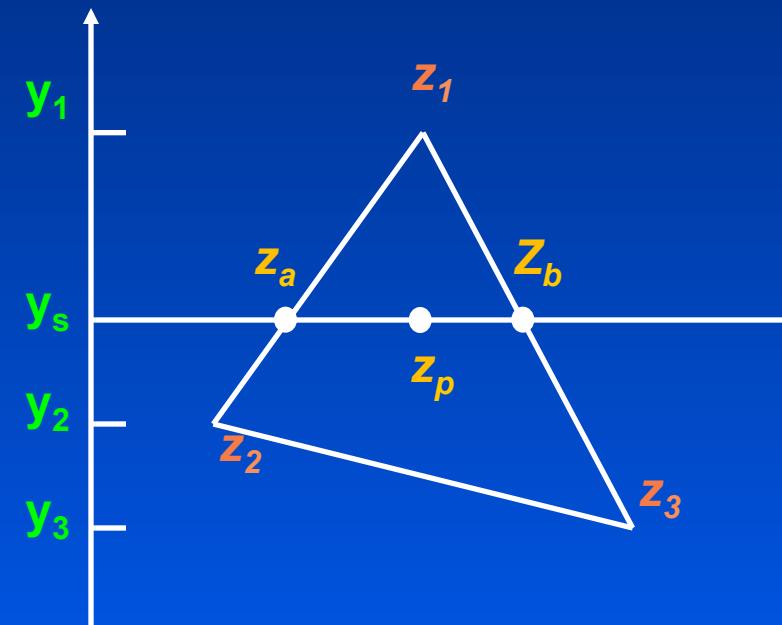
$$z_a = z_1 - (z_1 - z_2) \frac{y_1 - y_s}{y_1 - y_2}$$

- Interpolate along z_1-z_3

$$z_b = z_1 - (z_1 - z_3) \frac{y_1 - y_s}{y_1 - y_3}$$

- Interpolate along z_a-z_b

$$z_p = z_b - (z_b - z_a) \frac{x_b - x_p}{x_b - x_a}$$



- Can be done incrementally :
- If $Y_{i+1} = Y_i + 1$

$$\begin{aligned}
 Z_{i+1} &= Z_1 - \left(\frac{Z_1 - Z_2}{Y_1 - Y_2} \right) (Y_1 - Y_i) + \left(\frac{Z_1 - Z_2}{Y_1 - Y_2} \right) \\
 &= Z_i + \left(\frac{Z_1 - Z_2}{Y_1 - Y_2} \right)
 \end{aligned}$$

similary if $X_{i+1} = X_i + 1$

Summary of z-buffer algorithms

- No upper limit on complexity of scene
- Simple
- Take advantage of image space coherence
- De-facto standard for hidden surface algorithms

- Inefficient: scan convert polygons that are not visible
- Transparency is a problem where you need to process the polygons in the order: far to near
- As number of polygons increases and size decreases, setup time for each polygon dominate over per pixel cost
 - Problem for large scenes that needs to be implemented in real-time
 - Need to do processing on the scene to make hidden surface easier

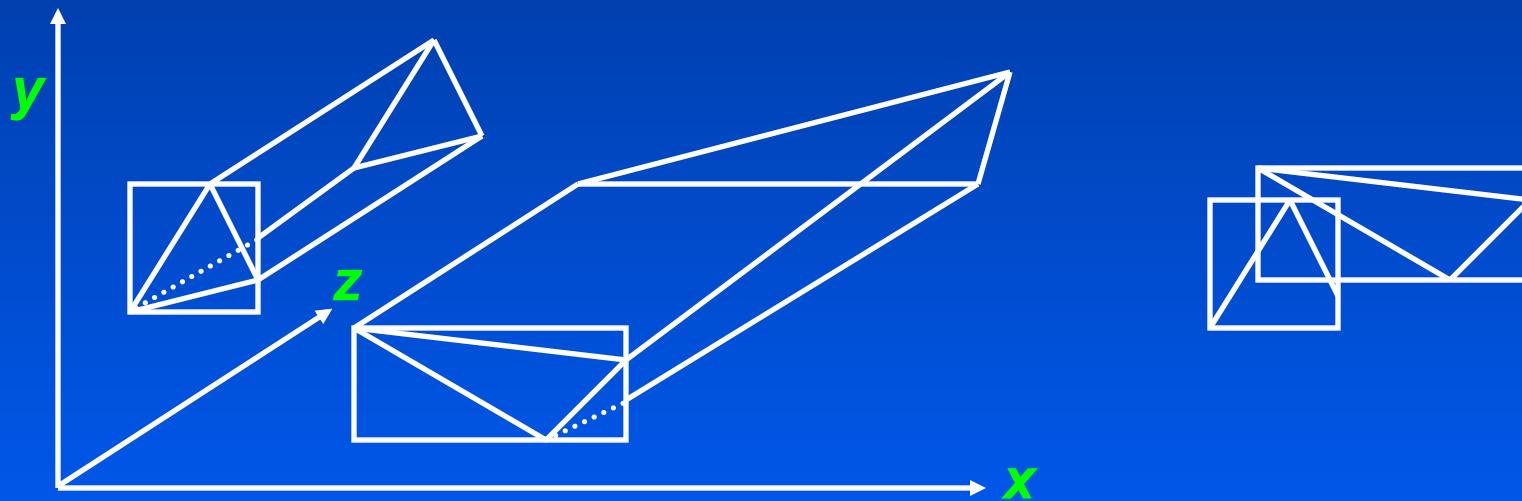
3-D Depth-Sort Algorithm,

- Strategy is to work back to front : Sort polygons based on furthest z-coordinate
- Scan convert furthest polygon first, then work forward toward viewpoint
resolve ambiguities as encountered
- Combine object-precision and image-precision operations

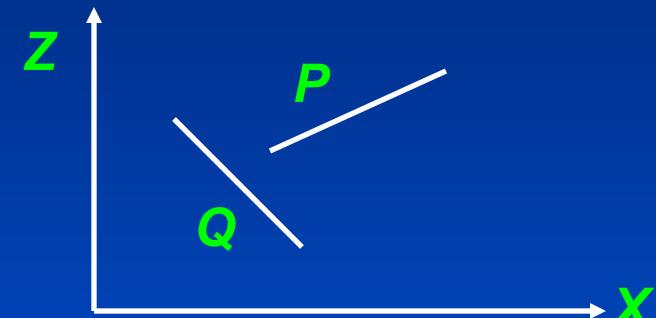
- Given preliminary sorted list of polygons (e.g. based on one of the vertex z value)
 - P: Polygon at end of list (furthest polygon in list)
 - $\{Q\}$: List of all polygons whose z-extent overlaps that of P
- Before scan-converting P, make up to five tests (in order of increasing complexity) to compare P with each polygon Q_i in $\{Q\}$
 - If all polygons pass at least one test, then scan-convert P
Means there are no Q that is in front of P
 - Next polygon becomes new P

1. Extents in x do not overlap
2. Extents in y do not overlap

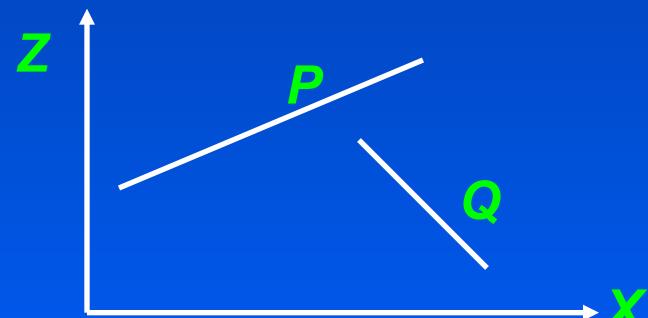
Note-the xy-extent test can cause false rejections: i.e. overlap in x and y does not imply overlap in objects themselves



3. P is completely on the "back" side
of Q_i , away from the viewpoint

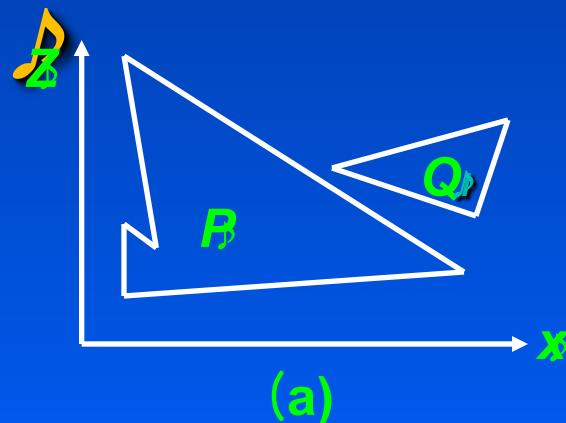
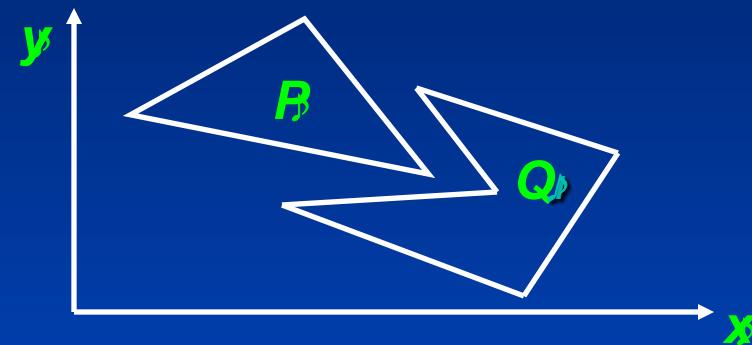


4. Q_i is completely on the "front" side
of P, toward the view point

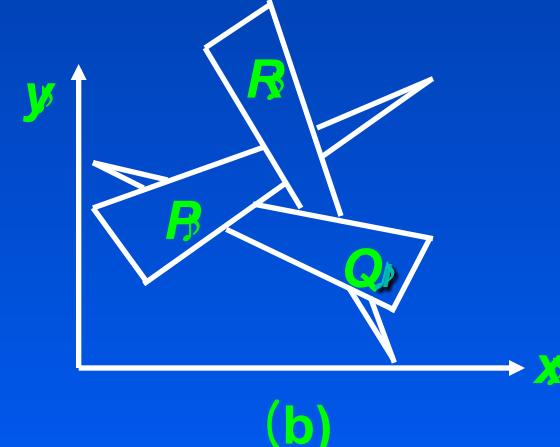


5. The projection of P and Q_i
are disjoint

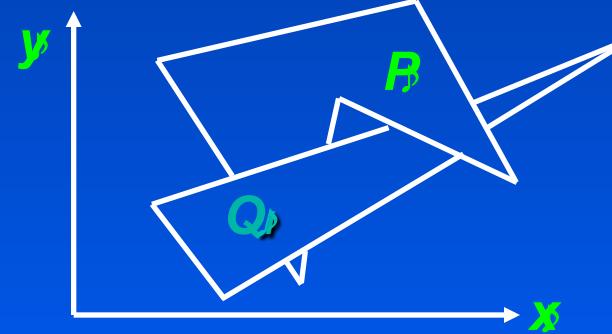
Sometimes all 5 tests fail



(a)



(b)



(c)

- IF all tests fail, move failed polygon Q_i to end of list, mark it to prevent infinite loops, repeat tests
 - Works in case (a), not in (b) or (c)
- If again all tests fail, don't move marked polygon, divide and conquer
 - Partition one polygon along plane of the other
- Once order has been determined, paint back-to-front (or front-to-back if we can determine whether a particular pixel has been written to frame buffer)
- Can be used to do transparency correctly

Binary Space Partitioning (BSP) Trees

- Divide and conquer :
 - To order any polygon correctly, order all polygons on the “far”(relative to viewpoint) half-space relative to the polygon, then that polygon, then all polygons on the polygon’s “near” half-space
 - But how to order the polygons on a half-space correctly? Choose another polygon in that region and process it recursively!

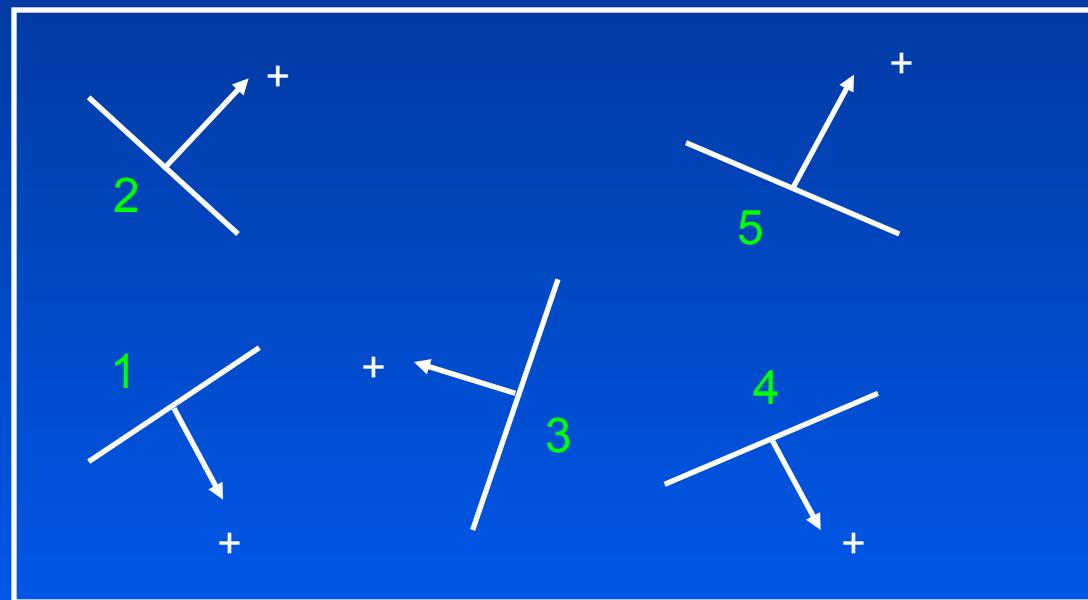


- Perform view-independent step once each time scene changes:
 - Recursively subdivide environment into a hierarchy of half-spaces by dividing polygons in a half-space by the plane of a selected polygon
 - Build a BSP tree representing this hierarchy
 - Each selected polygon is the root of the subtree

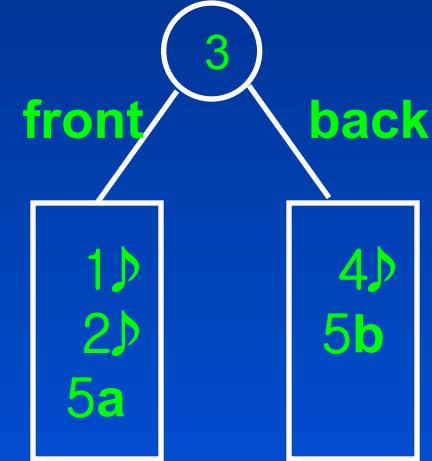
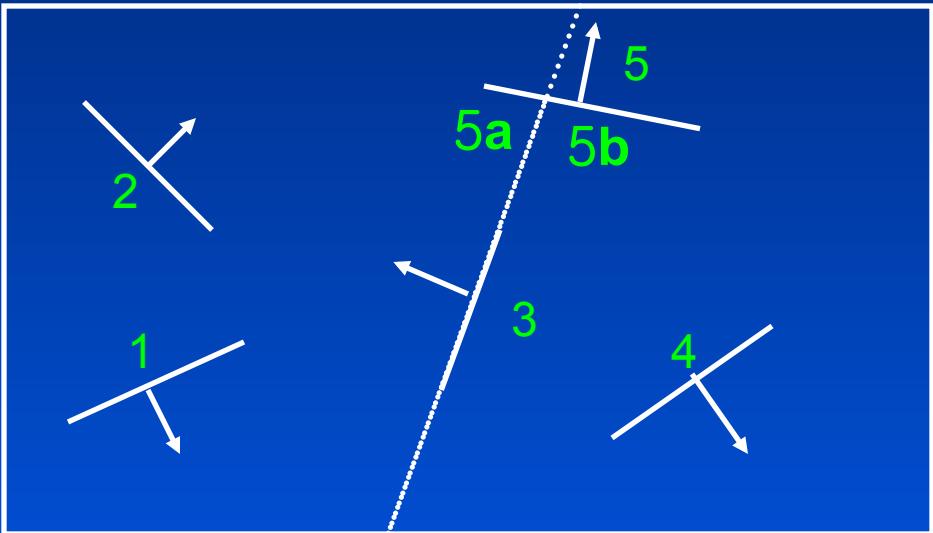
- Trades off view-independent preprocessing step (extra time and space) for low run-time overhead each time view changes
- Result is an ordered list back to front
 - Scan convert back-to-front (or front-to-back)
 - Good for transparency

Example

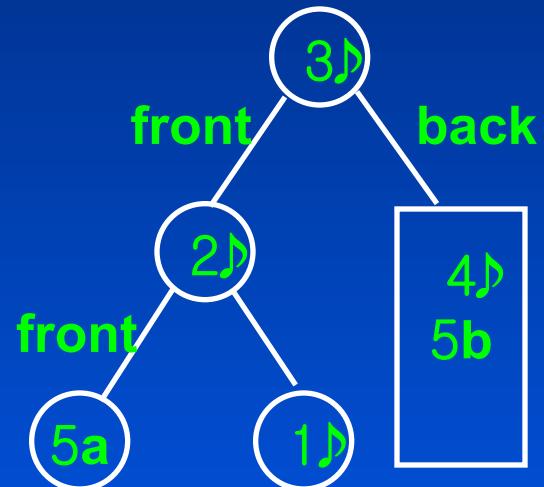
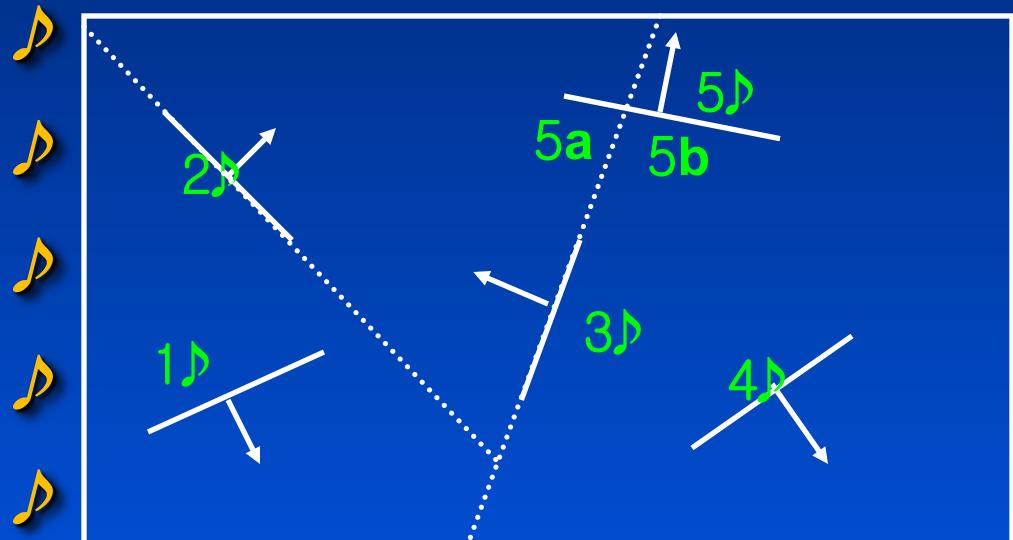
- Initial Scene looking from “above”
- Each polygon has a “+” side and a “-” side which is not related to position of camera



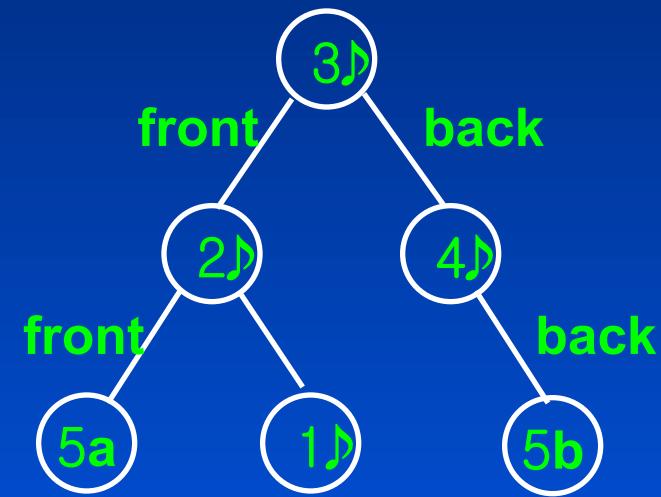
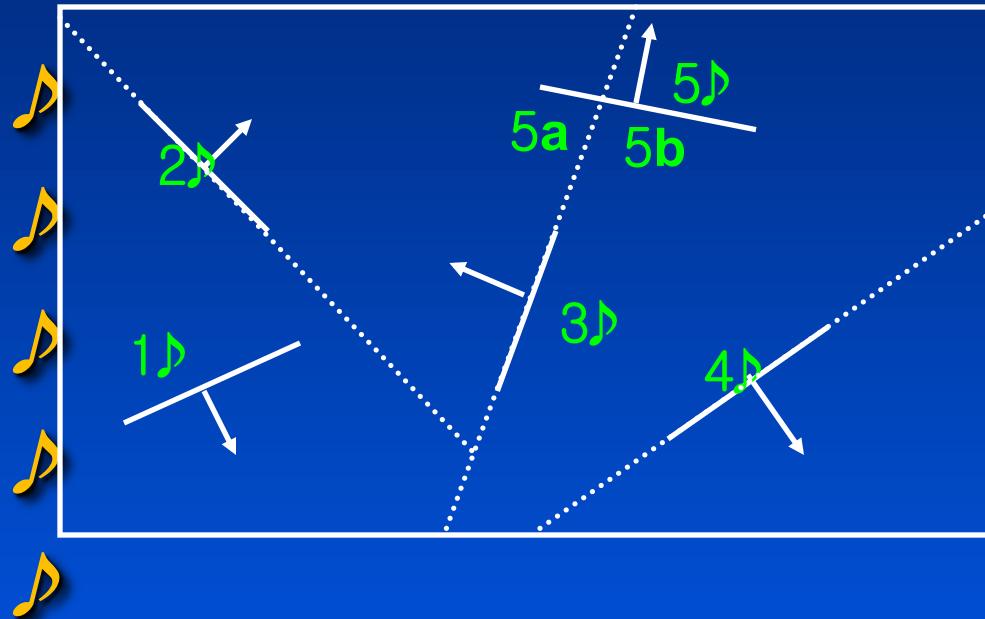
♪ ♪ ♪ ♪ ♪



- Choose any polygon (e.g., polygon3) and subdivide space by its plane, splitting polygons when necessary

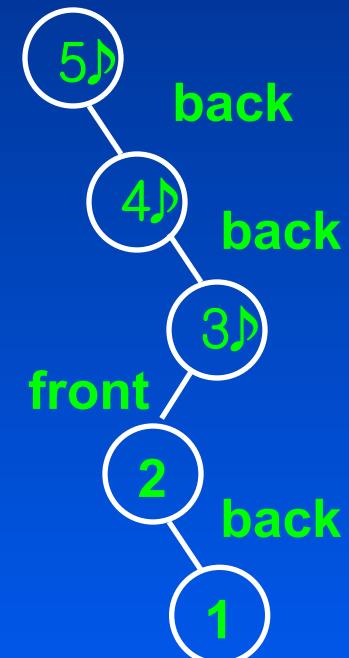
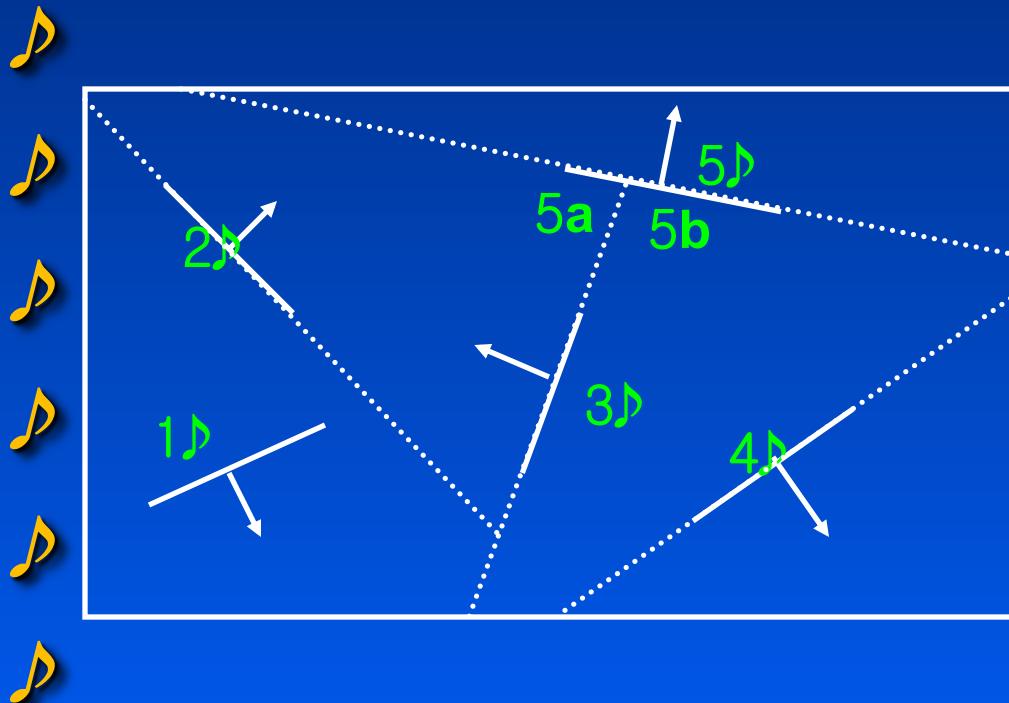


- Process front sub-tree recursively



- Process back sub-tree recursively

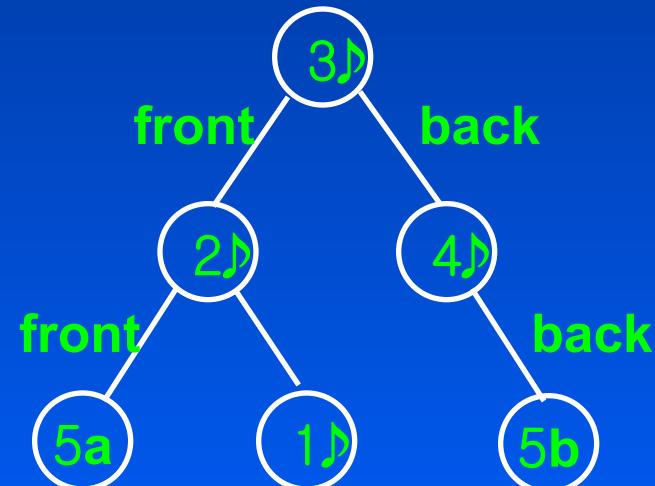
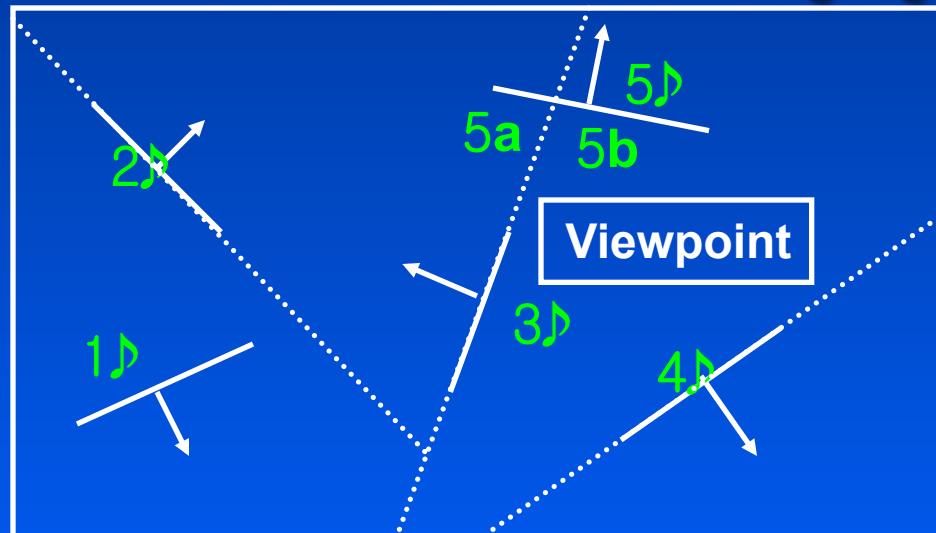
- An alternative BSP tree with polygon 5 at the root



To Generate Ordered Polygons

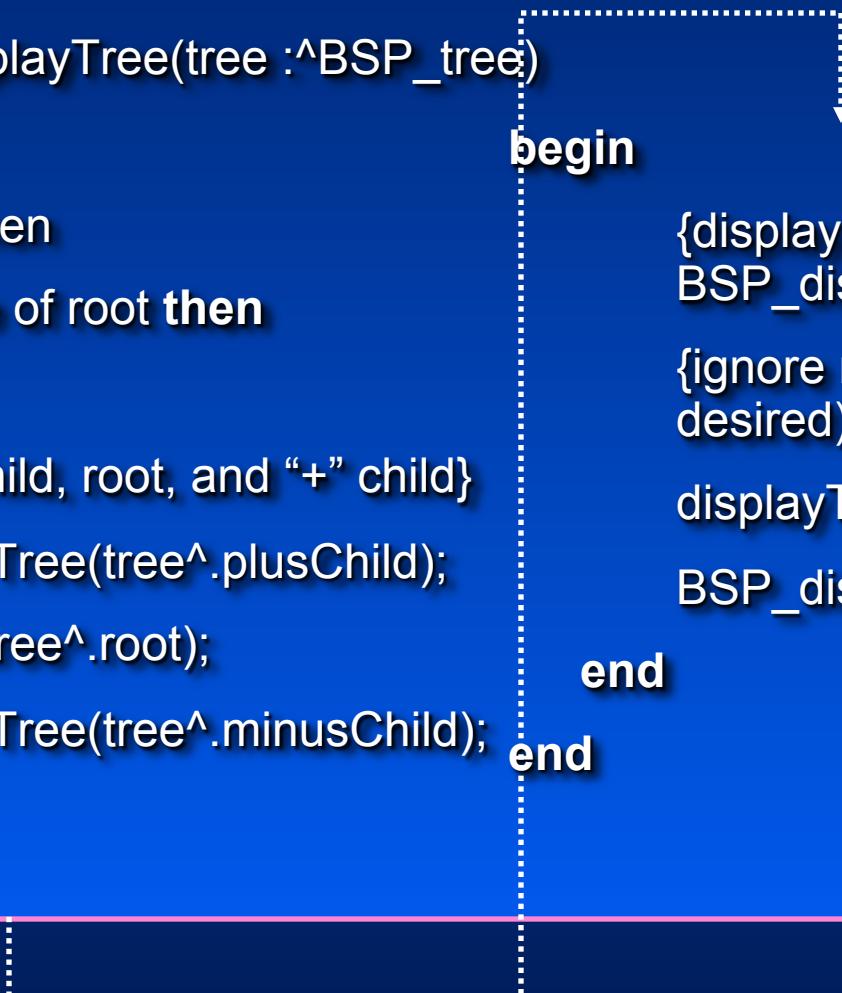
- Perform BSP tree traversal (view-dependent step) once each time viewpoint changes
 - Nodes can be visited in back-to-front order using a modified in-order traversal of the tree
 - At any node, the far side of a node's polygon is the side that the viewpoint is not in

- Given the following viewpoint:
 - 1, 2, 5a, 3, 4, 5b
 - Order not necessarily by distance



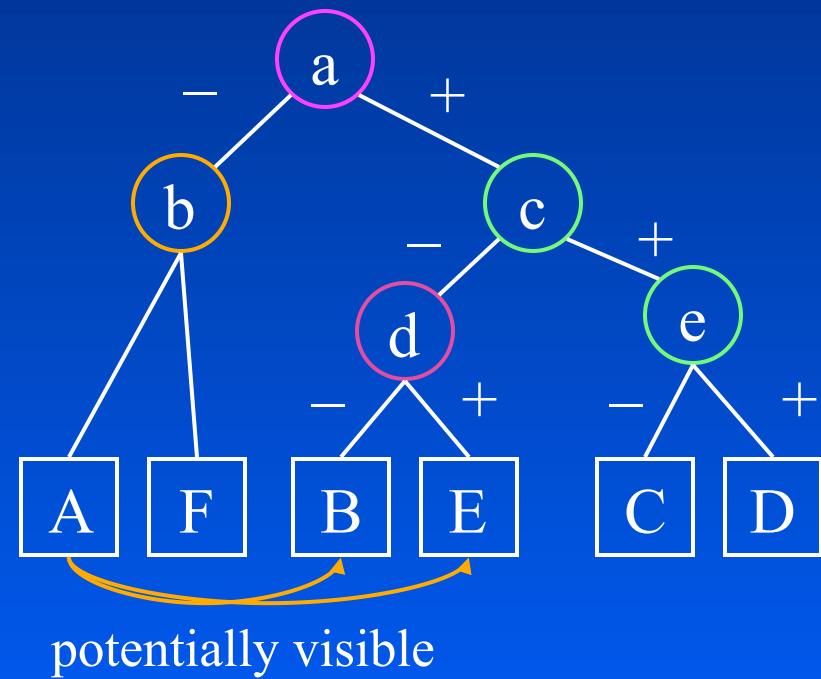
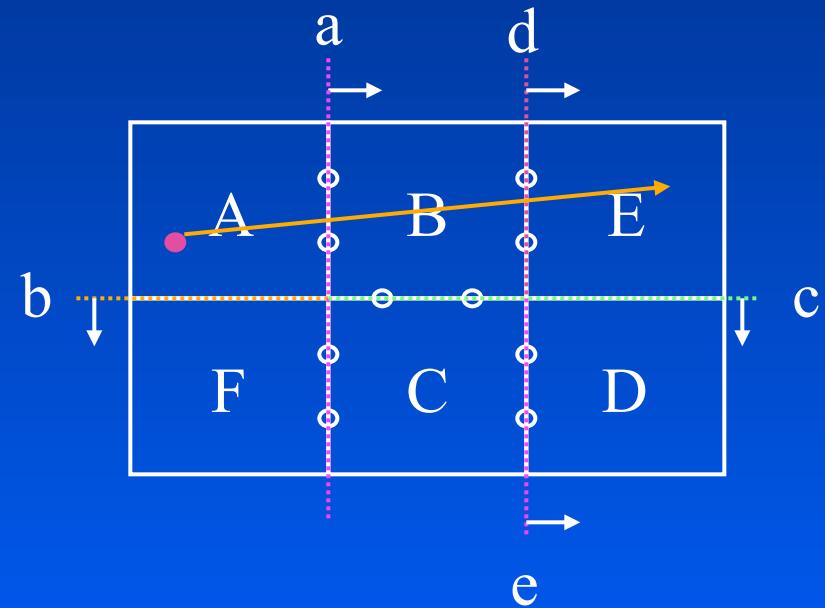
BSP Tree Traversal Algorithm

```
Procedure BSP_displayTree(tree :^BSP_tree)
Begin
  If tree is not empty then
    If viewer is in "+" side of root then
      begin
        {display “-” child, root, and “+” child}
        BSP_displayTree(tree^.plusChild);
        {ignore next line if back-face culling
         desired)
        displayTree(tree^.root);
        BSP_displayTree(tree^.minusChild);
      end
    end
  else
```



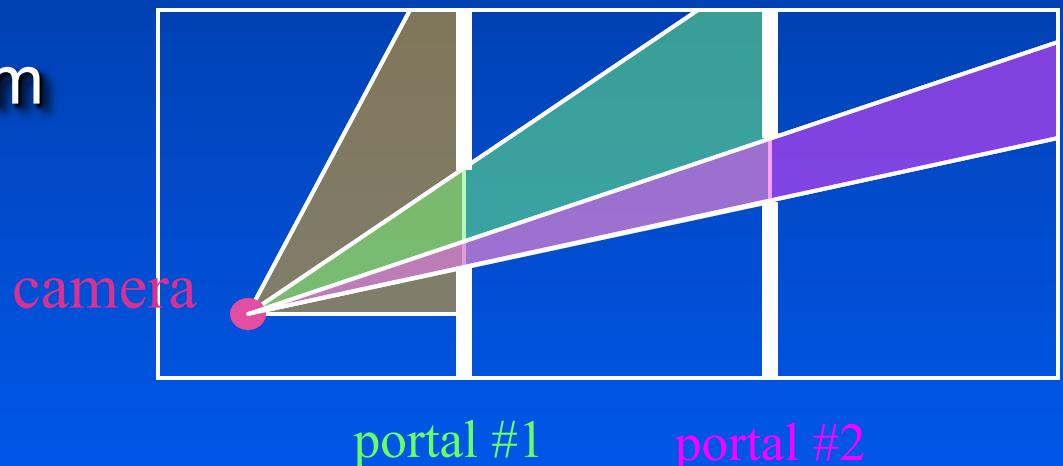
Portal with BSP

- Use BSP to quickly identify objects that are in the same room
- Divide space into BSP using walls as dividing planes
- If camera is in the space occupied by a node, objects in that node may be visible
- Additionally, can see parts of other rooms through windows/doors (portals)
- PVS (potentially visible set): set of nodes visible from a node
- When rendering
 - objects in node A + PVS(A)



Portal without BSP tree

- Cull and render the room where camera is located
- If polygon tagged as portal, recurse into next room and cull (against reduced view frustum) and render until no more portals encountered
- Mirrors portals to room itself



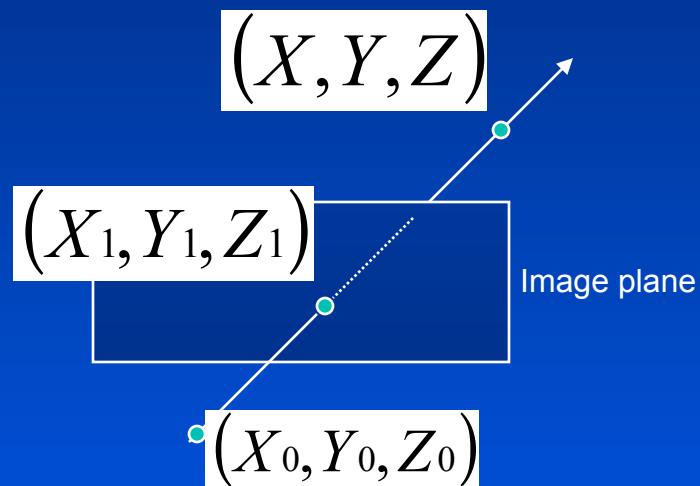
Visible-Surface Ray Tracing♪

- Shoots a ray through each pixel
- Visible object is closest intersected by ray
- This is the naïve image-precision algorithm described before
- We will revisit ray-tracing with recursive ray-tracing

Select center of Projection and window on view plane;
for each scan line in image **do**
 for each pixel in scan line **do**
 begin
 {determine ray from center of projection through pixel}
 for each object in scene **do**
 if object is intersected and
 is closest considered thus far **then**
 begin
 record intersection and object name;
 set pixel's color to that at closest object intersection
 end
 end

Ray-object intersection

- Parametric ray representation:



$$X = X_0 + t(X_1 - X_0)$$

$$Y = Y_0 + t(Y_1 - Y_0)$$

$$Z = Z_0 + t(Z_1 - Z_0)$$

or

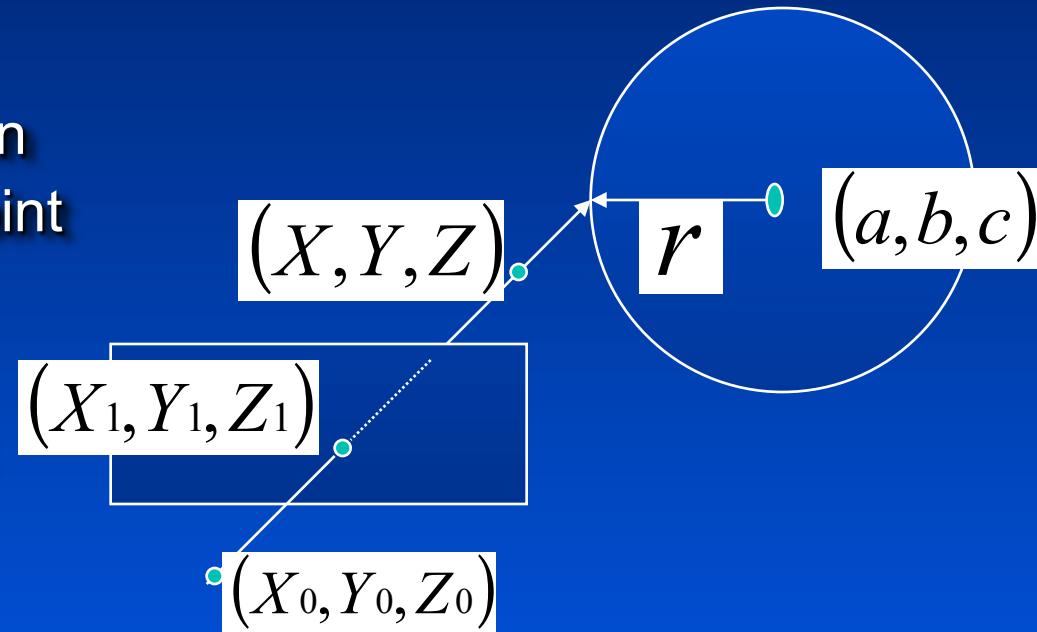
$$X = X_0 + t\Delta X$$

$$Y = Y_0 + t\Delta Y$$

$$Z = Z_0 + t\Delta Z$$

- (x_0, y_0, z_0) is the center of projection
- (x_l, y_l, z_l) is the center of a pixel on the view plane
- t is a measure of the distance along the ray from the origin

- Sphere
- Plug in expression for (x, y, z) of a point on the ray to give a quadratic in t with constant coefficients
- Ready to be solved using quadratic formula



$$(X - a)^2 + (Y - b)^2 + (Z - c)^2 = r^2$$

$$X^2 - 2aX + a^2 + Y^2 - 2bY + b^2 + Z^2 - 2cZ + c^2 = r^2$$

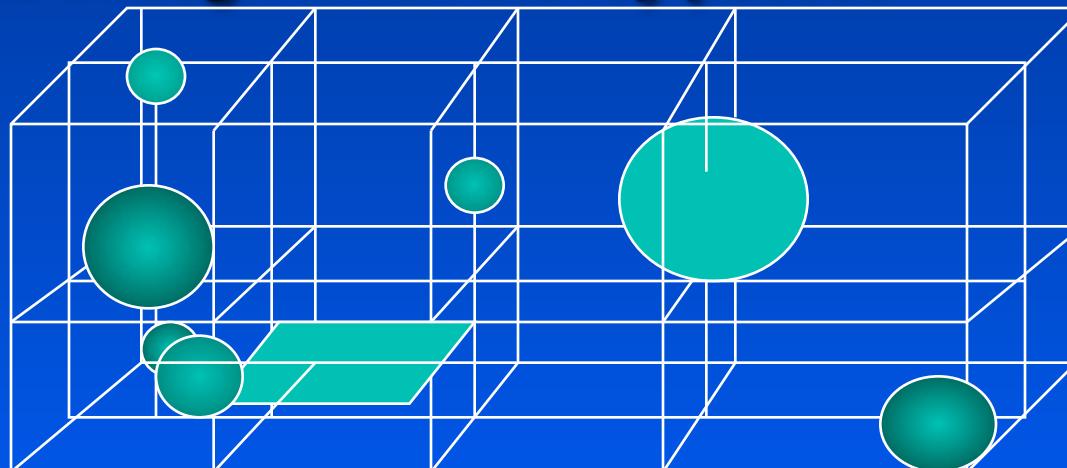
- Most useful for modeling reflections, refraction, shadows, when rays are traced recursively from closest intersection
- Expensive: intersect each ray with every object
 - E.g., for a small scene:
1024 * 1024 * 1024 << 1,000,000,000
scan pixels objects intersections
lines on a
line
- Must use techniques such as spatial partitioning and hierarchies to cut down number of intersections

Optimizing intersection calculation♪

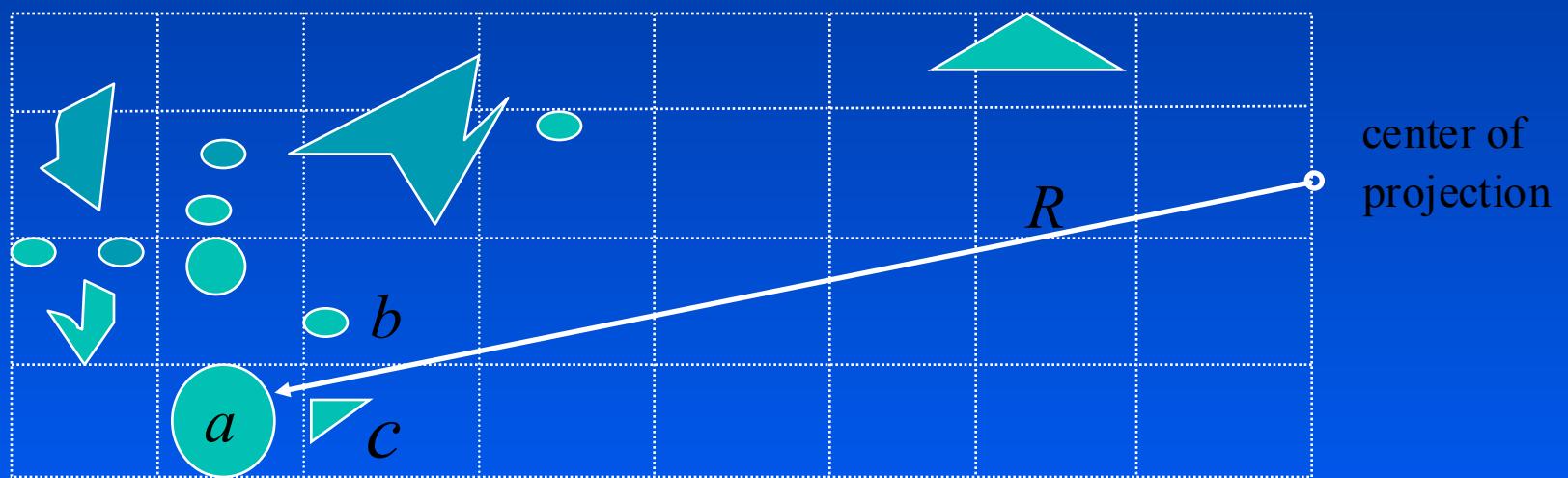
- Transform so that ray lie along z axis
- Bounding volumes
- Spatial partitioning
 - Regular voxels
 - Hierarchical partitioning (Octrees)

Regular Spatial Partitioning

- Environment is partitioned into regular grid of equal-sized volumes
- Ray intersections are calculated with objects in the partitions through which the ray passes



- Ray R need only be intersected with objects a, b, and c since the other partitions through which R passes are empty
- Use modified line rasterizer to determine the next cell



Next: Illumination Models

