Visible Surface Detection

CEng 477
Introduction to Computer Graphics
Fall 2007

Visible Surface Detection

- Visible surface detection or hidden surface removal.
- Realistic scenes: closer objects occludes the others.
- Classification:
  - Object space methods
  - Image space methods

Object Space Methods

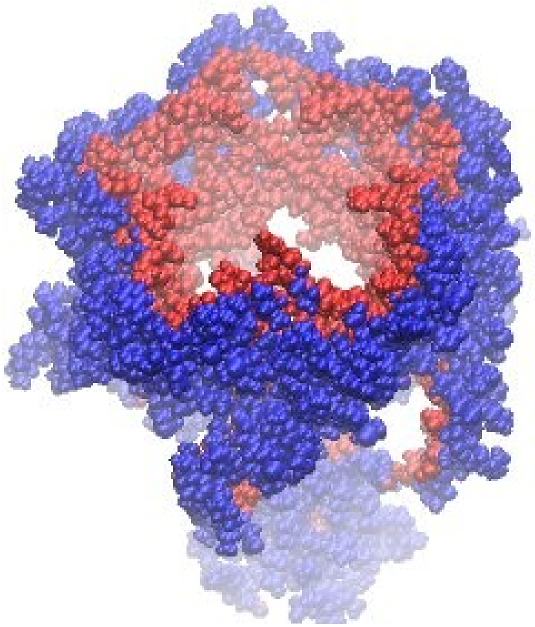
- Algorithms to determine which parts of the shapes are to be rendered in 3D coordinates.
- Methods based on comparison of objects for their 3D positions and dimensions with respect to a viewing position.
- For N objects, may require N^*N comparision operations.
- Efficient for small number of objects but difficult to implement.
- Depth sorting, area subdivision methods.

Image Space Methods

- Based on the pixels to be drawn on 2D. Try to determine which object should contribute to that pixel.
- Running time complexity is the number of pixels times number of objects.
- Space complexity is two times the number of pixels:
  - One array of pixels for the frame buffer
  - One array of pixels for the depth buffer
- Coherence properties of surfaces can be used.
- Depth-buffer and ray casting methods.

Depth Cueing

- Hidden surfaces are not removed but displayed with different effects such as intensity, color, or shadow for giving hint for third dimension of the object.
- Simplest solution: use different colors-intensities based on the dimensions of the shapes.



Back-Face Detection

- Back-face detection of 3D polygon surface is easy
- Recall the polygon surface equation:

$$Ax + By + Cz + D < 0$$

- We need to also consider the viewing direction when determining whether a surface is back-face or front-face.
- The normal of the surface is given by:

$$N = (A, B, C)$$

Back-Face Detection

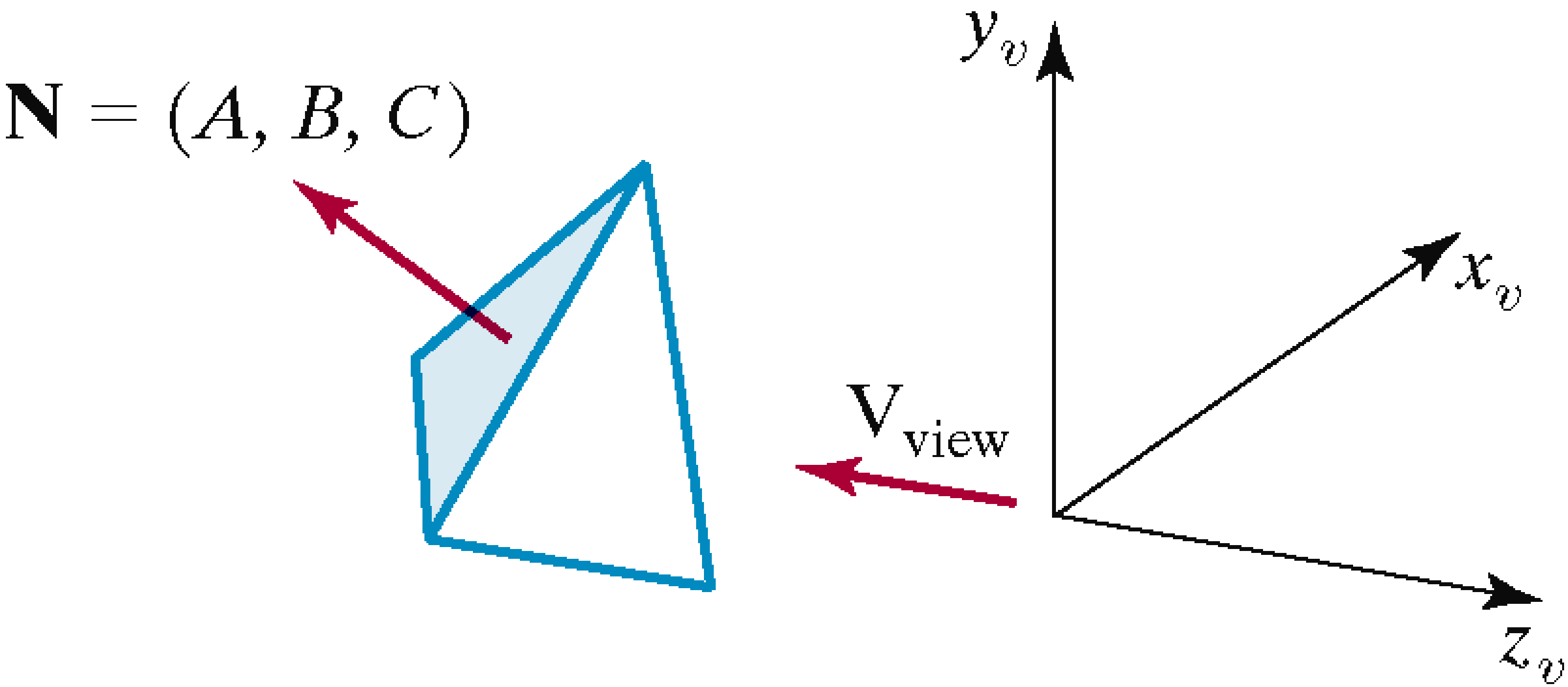
- A polygon surface is a back face if:

$$V_{\text{view}} \cdot N > 0$$

- However, remember that after application of the viewing transformation we are looking down the negative *z*-axis. Therefore a polygon is a back face if:

$$(0,0,-1) \cdot N > 0$$
$$\text{or if } C < 0$$

Back-Face Detection



- We will also be unable to see surfaces with C=0. Therefore, we can identify a polygon surface as a back-face if:

$$C \leq 0$$

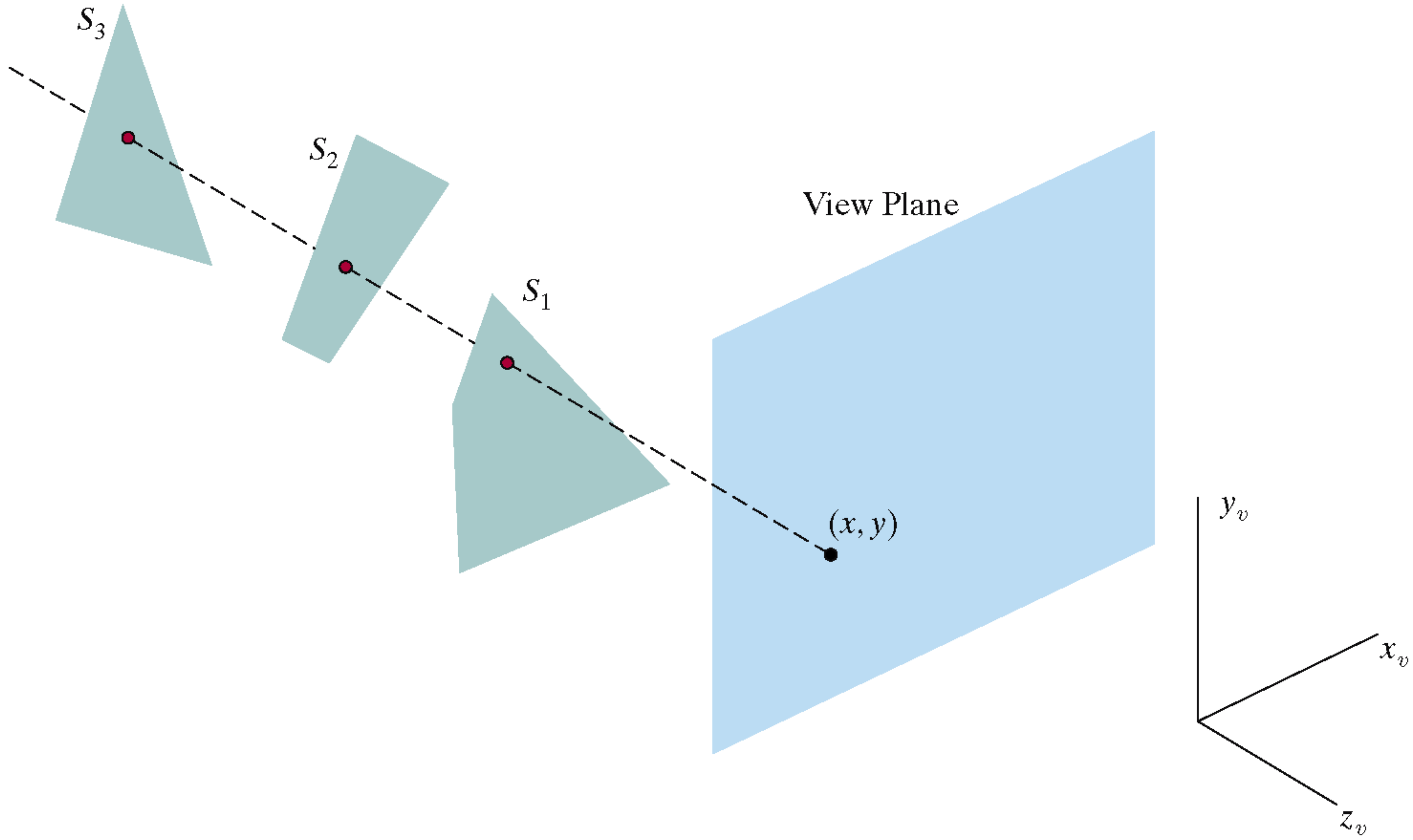
Back-Face Detection

- Back-face detection can identify all the hidden surfaces in a scene that contain non-overlapping convex polyhedra.
- But we have to apply more tests that contain overlapping objects along the line of sight to determine which objects obscure which objects.

Depth-Buffer Method

- Also known as *z*-buffer method.
- It is an image space approach
  - Each surface is processed separately one pixel position at a time across the surface
  - The depth values for a pixel are compared and the closest (smallest z) surface determines the color to be displayed in the frame buffer.
  - Applied very efficiently on polygon surfaces
  - Surfaces are processed in any order

Depth-Buffer Method



Depth-Buffer Method

- Two buffers are used
  - Frame Buffer
  - Depth Buffer
- The z-coordinates (depth values) are usually normalized to the range [0,1]

Depth-Buffer Algorithm

- Initialize the depth buffer and frame buffer so that for all buffer positions (x,y),

  depthBuff (x,y) = 1.0, frameBuff (x,y) =bgColor

- Process each polygon in a scene, one at a time
  - For each projected (x,y) pixel position of a polygon, calculate the depth z.
  - If z < depthBuff (x,y), compute the surface color at that position and set

  depthBuff (x,y) = z, frameBuff (x,y) = surfCol (x,y)

Calculating depth values efficiently

- We know the depth values at the vertices. How can we calculate the depth at any other point on the surface of the polygon.
- Using the polygon surface equation:

$$z = \frac{-Ax - By - D}{C}$$

Calculating depth values efficiently

- For any scan line adjacent horizontal *x* positions or vertical *y* positions differ by 1 unit.
- The depth value of the next position (x+1,y) on the scan line can be obtained using

$$z' = \frac{-A(x+1) - By - D}{C}$$

$$= z - \frac{A}{C}$$

Calculating depth values efficiently

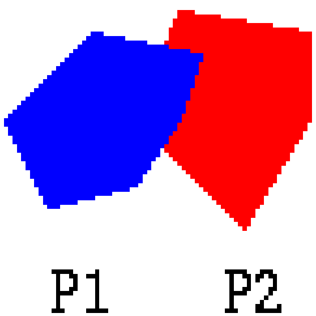
- For adjacent scan-lines we can compute the x value using the slope of the projected line and the previous x value.

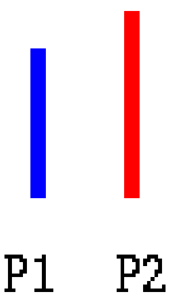
$$\{ x' = x - \frac{1}{m}$$

$$¿ \Rightarrow z' = z + \frac{A/m + B}{C} ¿¿$$

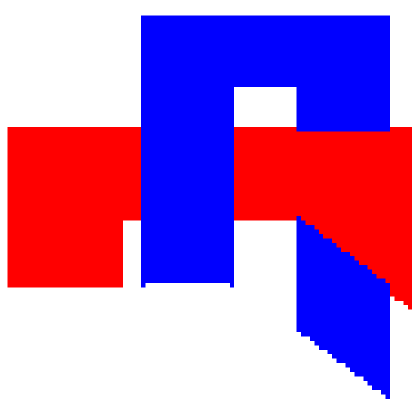
Depth-Buffer Method

- Is able to handle cases such as





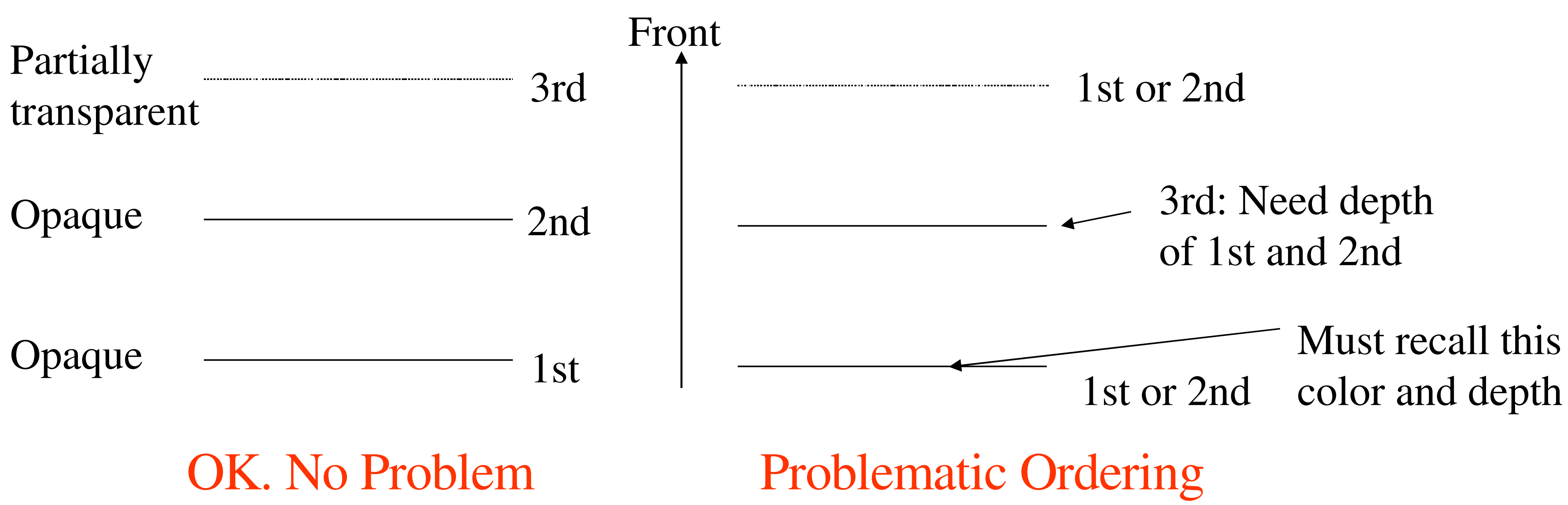
View from the Right-side



These polygons are both in front of and behind one another.

Z-Buffer and Transparency

- We may want to render transparent surfaces (alpha $\neq 1$) with a z-buffer
- However, we must render in back to front order
- Otherwise, we would have to store at least the first opaque polygon behind transparent one

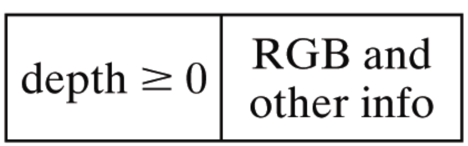


A-Buffer Method

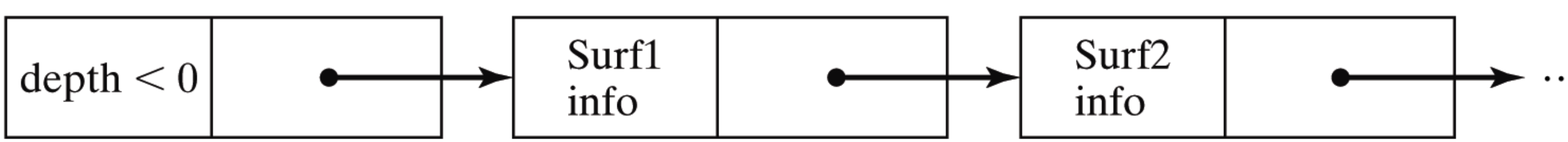
- Extends the depth-buffer algorithm so that each position in the buffer can reference a linked list of surfaces.
- More memory is required
- However, we can correctly compose different surface colors and handle transparent surfaces.

A-Buffer Method

- Each position in the A-buffer has two fields:
  - a depth field
  - surface data field which can be either surface data or a pointer to a linked list of surfaces that contribute to that pixel position



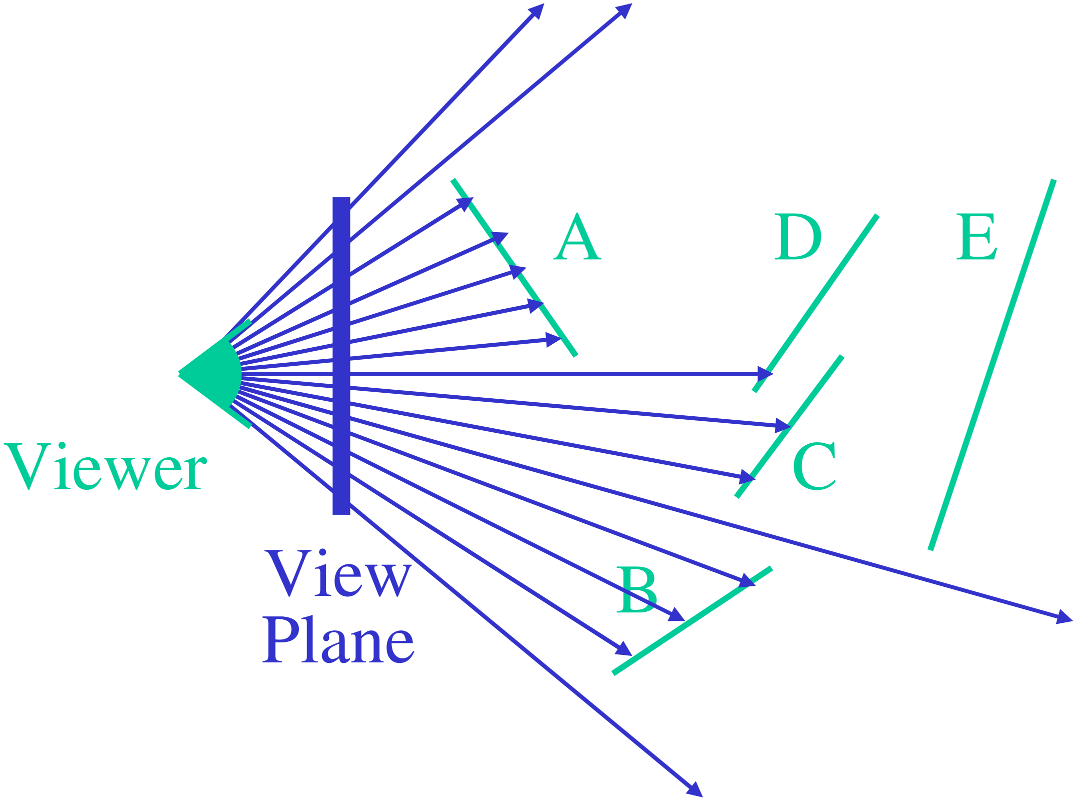
(a)



(b)

Ray Casting Algorithm

- Algorithm:
  - Cast ray from viewpoint through each pixel to find front-most surface

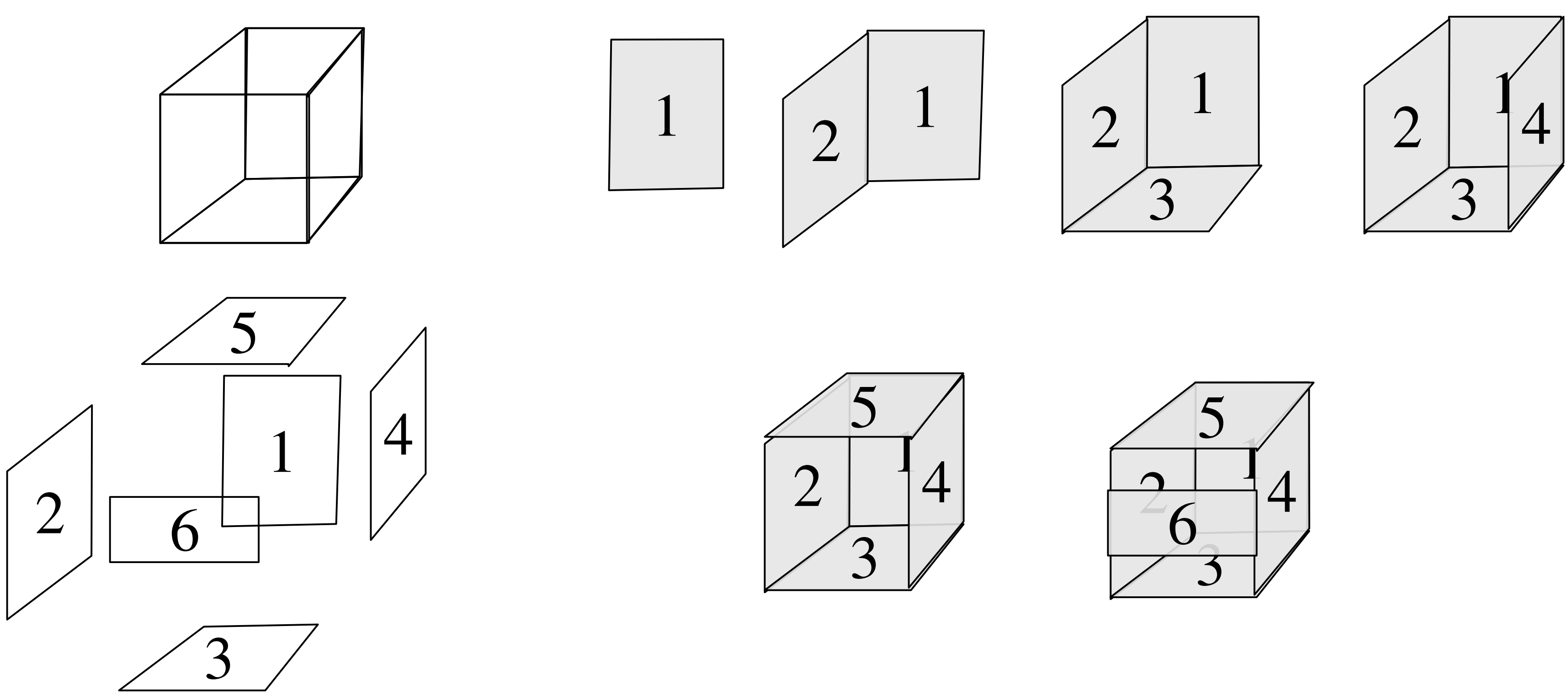


It is like a variation of the depth-buffer algorithm, in which we proceed pixel by pixel instead of proceeding surface by surface.

Object Space Methods

Depth Sorting

- Also known as painters algorithm. First draw the distant objects than the closer objects. Pixels of each object overwrites the previous objects.

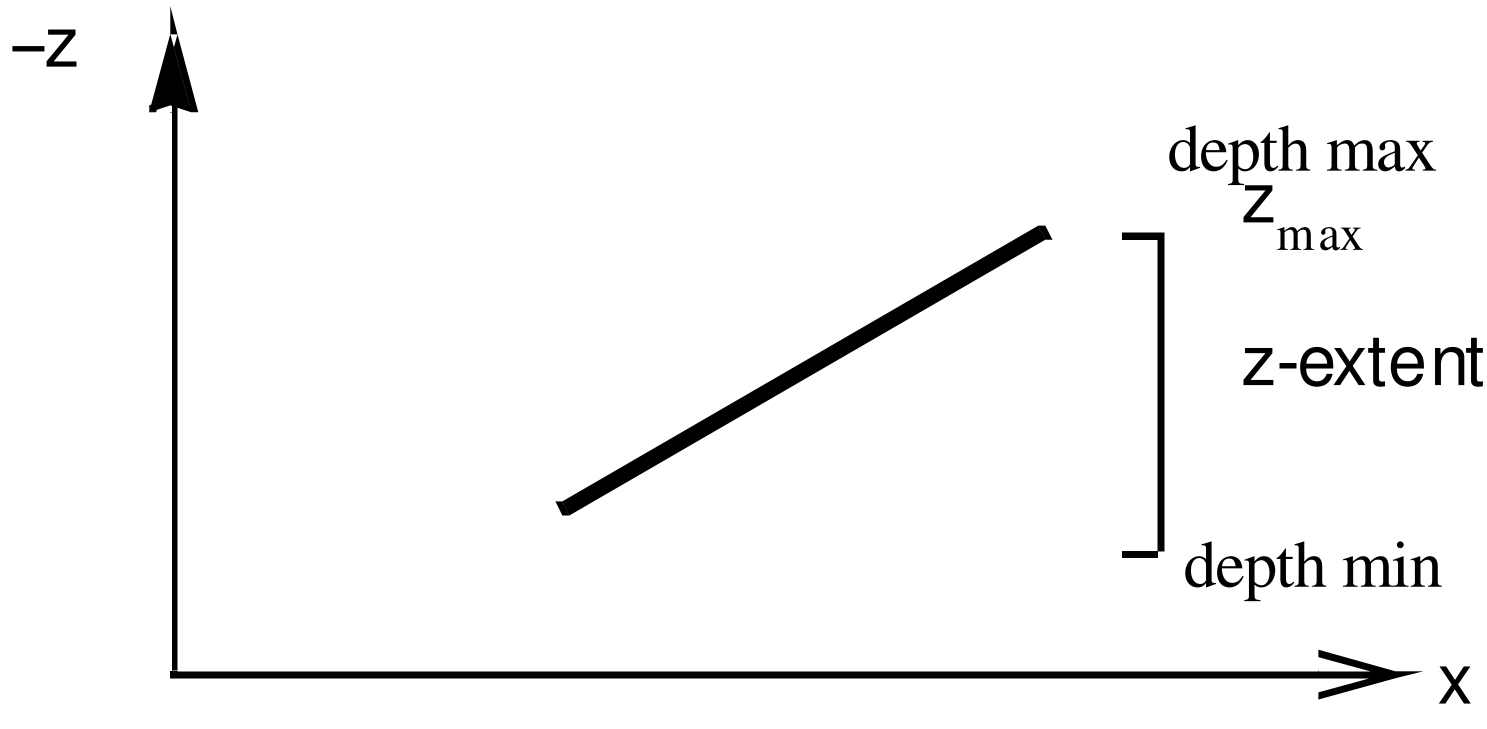


Depth-sort algorithm

- The idea here is to go back to front drawing all the objects into the frame buffer with nearer objects being drawn over top of objects that are further away.
- Simple algorithm:
  - Sort all polygons based on their farthest z coordinate
  - Resolve ambiguities
  - Draw the polygons in order from back to front
- This algorithm would be very simple if the z coordinates of the polygons were guaranteed never to overlap. Unfortunately that is usually not the case, which means that step 2 can be somewhat complex.

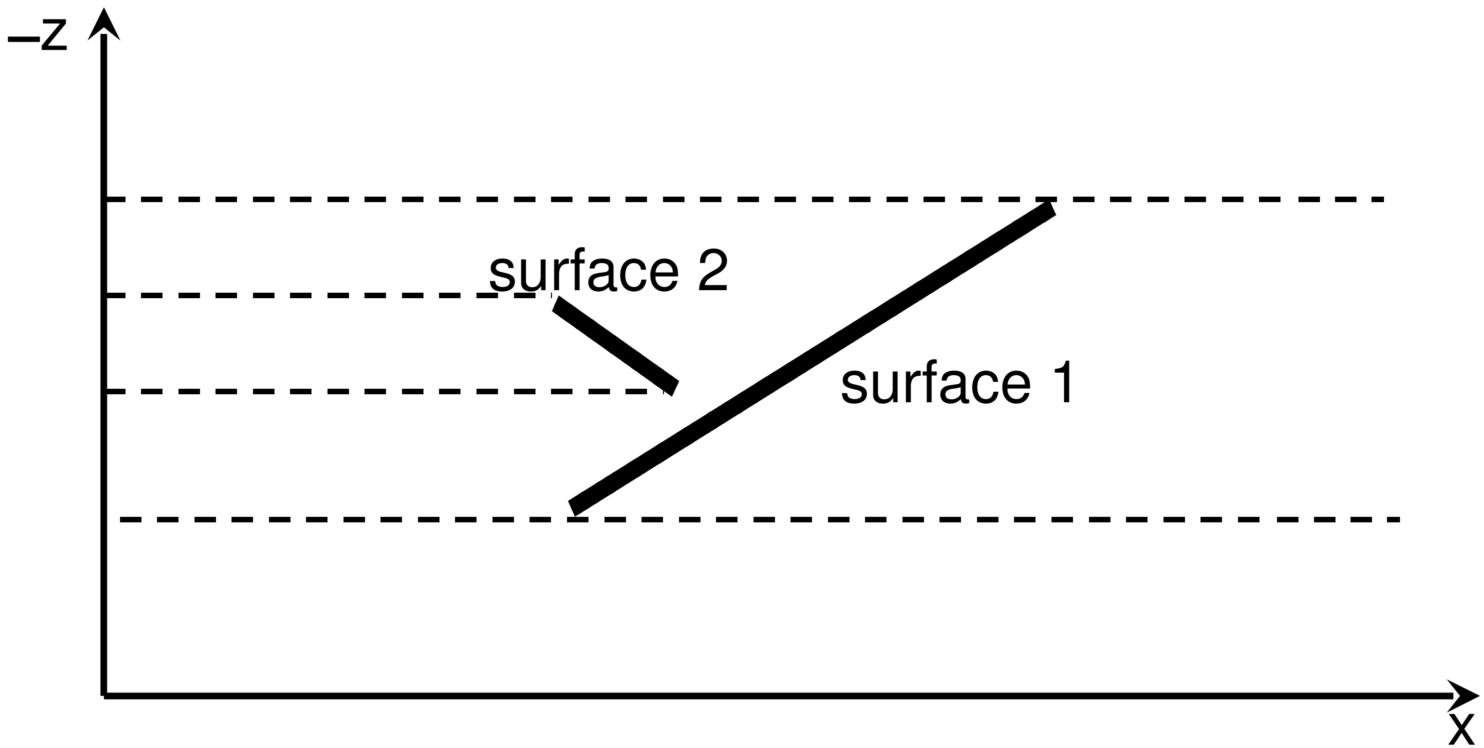
Depth-sort algorithm

- First must determine z-extent for each polygon

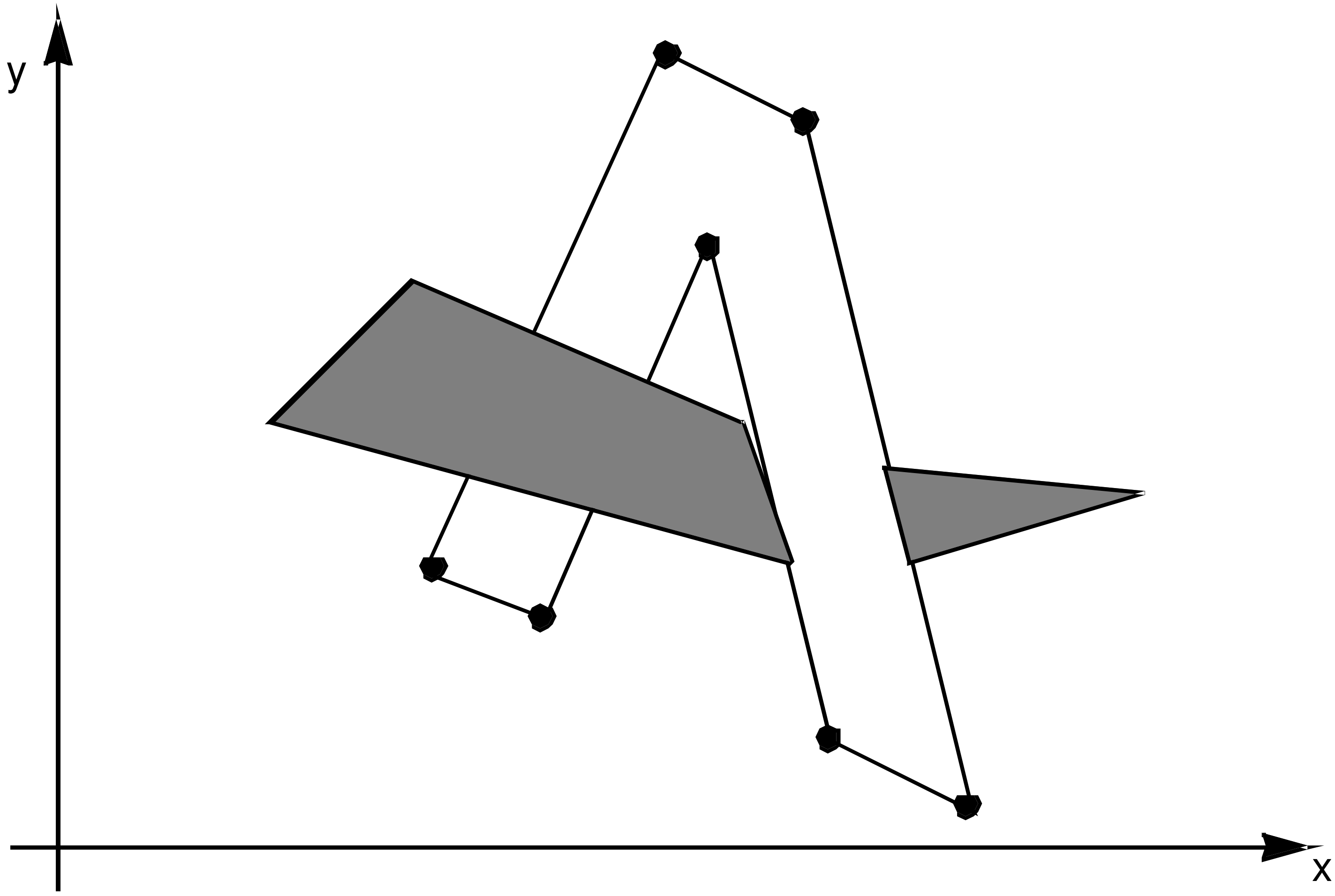


Depth-sort algorithm

- Ambiguities arise when the z-extents of two surfaces overlap.



Depth-sort algorithm



Depth-sort algorithm

- All polygons whose z extents overlap must be tested against each other.
- We start with the furthest polygon and call it P. Polygon P must be compared with every polygon Q whose z extent overlaps P's z extent. 5 comparisons are made. If any comparison is true then P can be written before Q. If at least one comparison is true for each of the Qs then P is drawn and the next polygon from the back is chosen as the new P.

Depth-sort algorithm

1. Do P and Q's x-extents not overlap?
2. Do P and Q's y-extents not overlap?
3. Is P entirely on the opposite side of Q's plane from the viewport?
4. Is Q entirely on the same side of P's plane as the viewport?
5. Do the projections of P and Q onto the (x,y) plane not overlap?

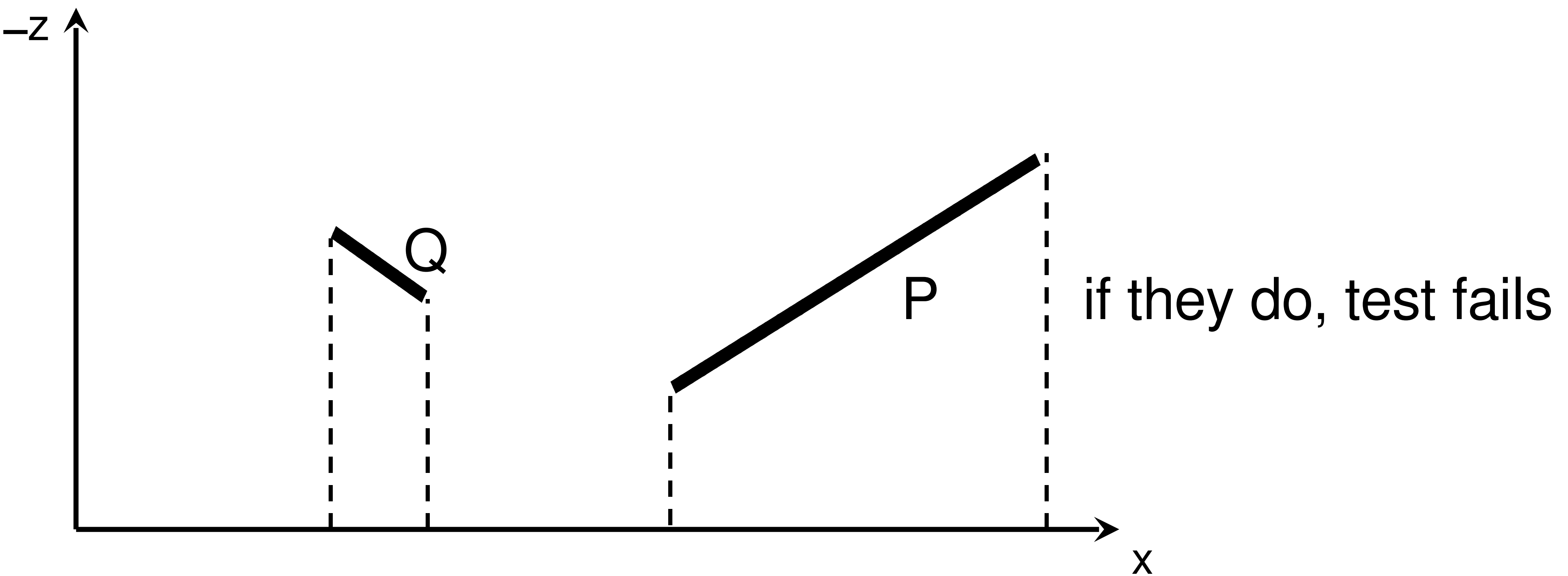
- If all 5 tests fail we quickly check to see if switching P and Q will work. Tests 1, 2, and 5 do not differentiate between P and Q but 3 and 4 do. So we rewrite 3 and 4 as:

3’. Is Q entirely on the opposite side of P's plane from the viewport?

4’. Is P entirely on the same side of Q's plane as the viewport?

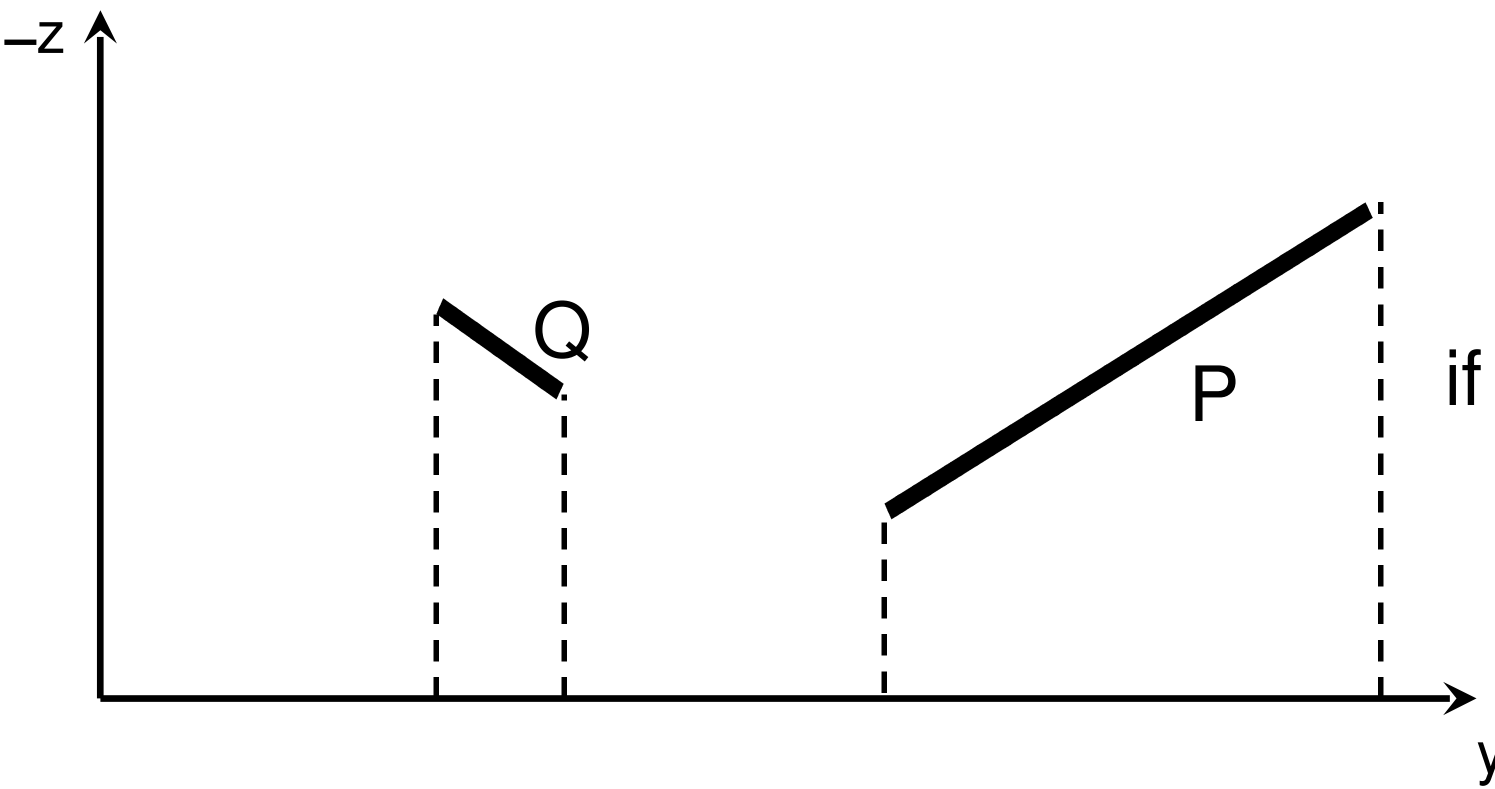
Depth-sort algorithm

x - extents not overlap?



Depth-sort algorithm

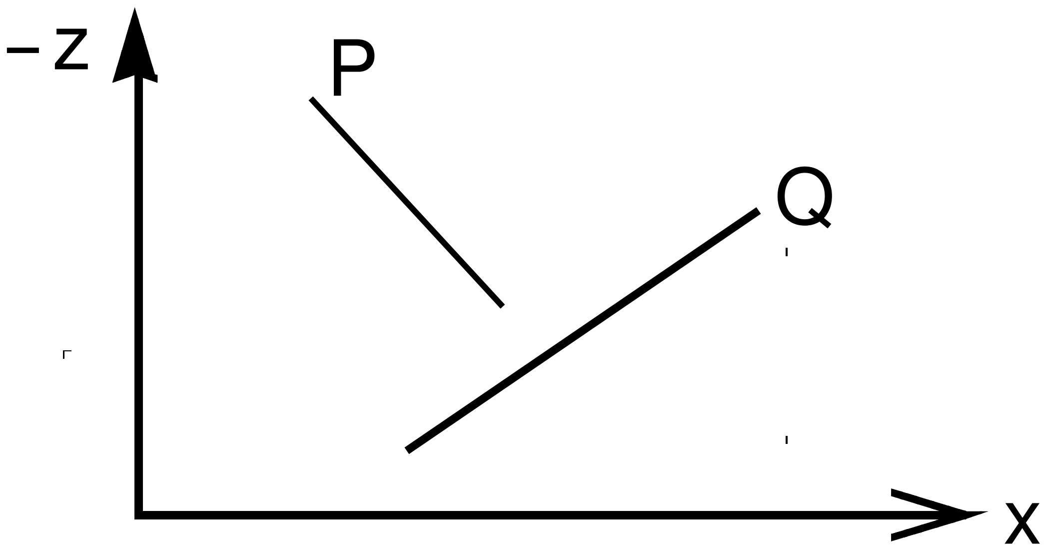
y - extents not overlap?



if they do, test fails

Depth-sort algorithm

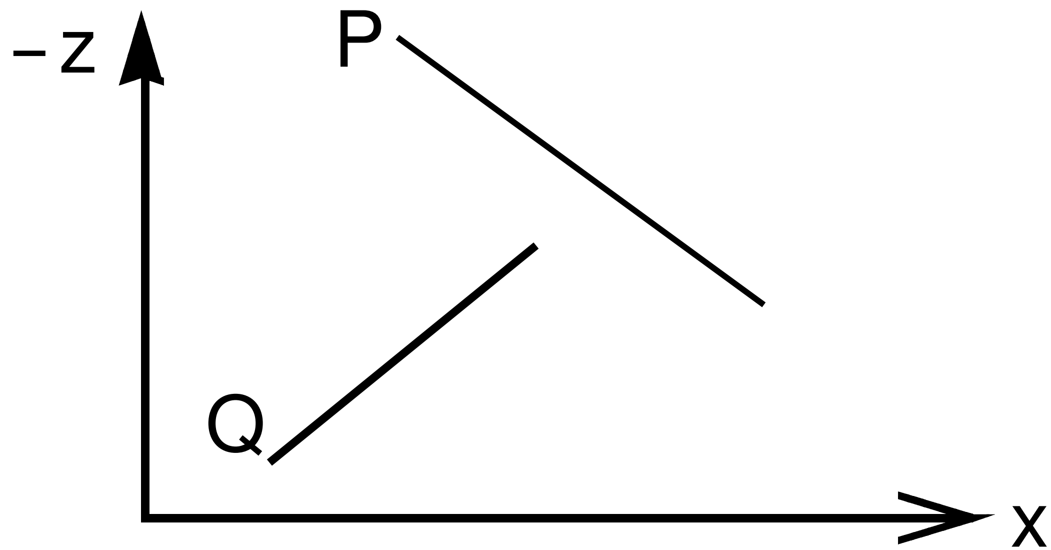
Is P entirely behind the surface Q relative to the viewing position (i.e., behind Q’s plane with respect to the viewport)?



Test is true…

Depth-sort algorithm

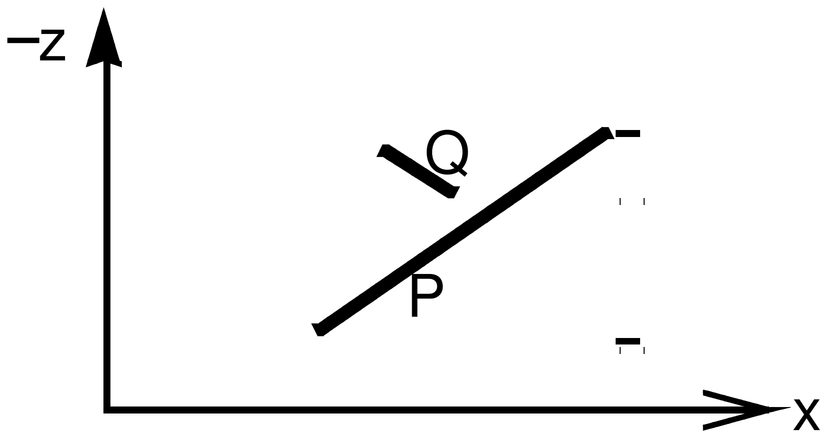
Is Q entirely in front of P's plane relative to the viewing position (i.e., the viewport)?

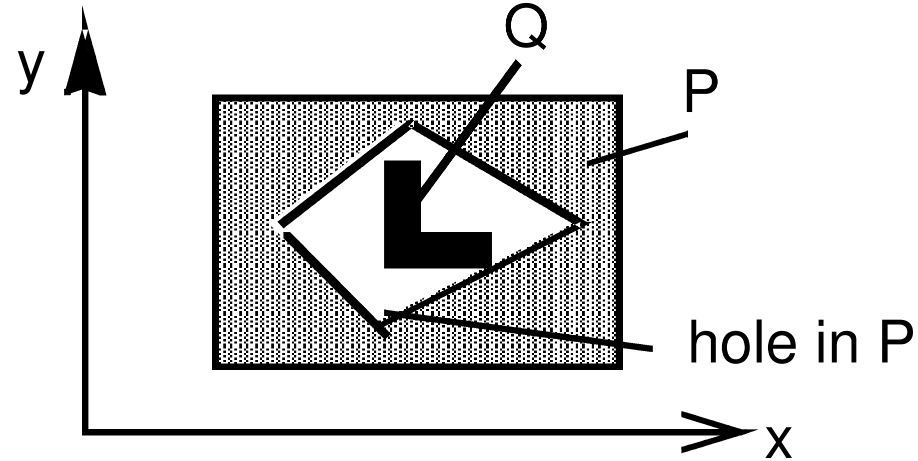


Test is true…

Depth-sort algorithm

Do the projections of P and Q onto the (x,y) plane not overlap?





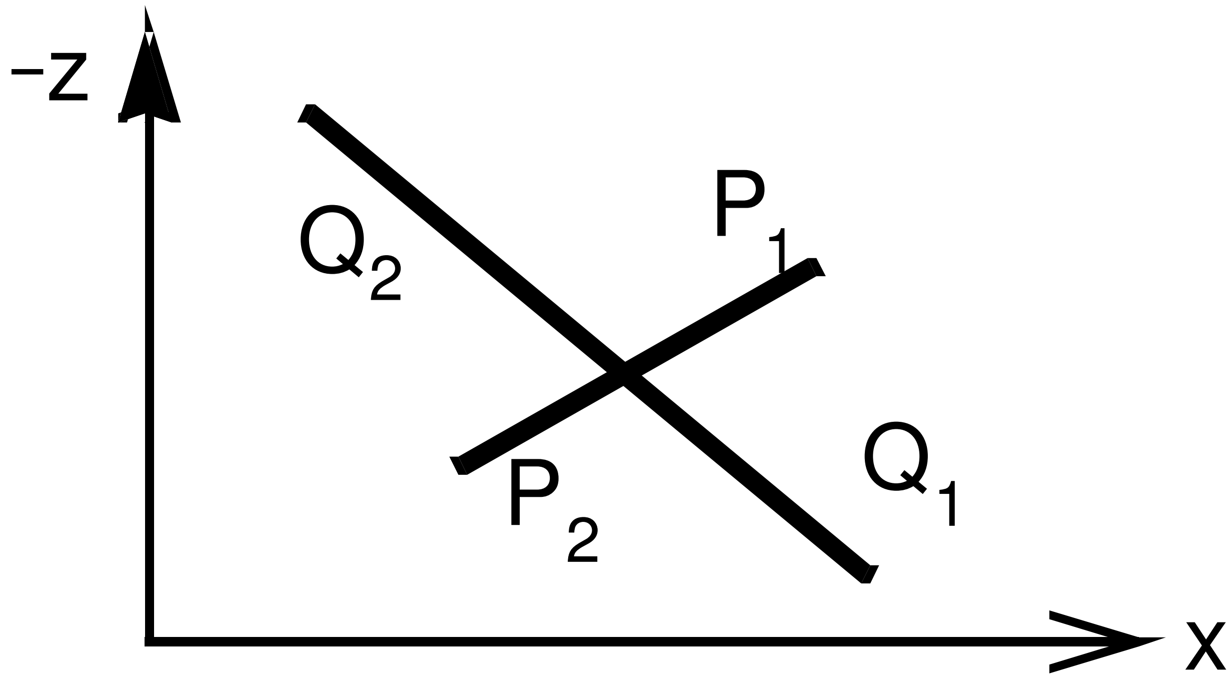
Test is true…

Depth-sort algorithm

- If all tests fail…
  - … then reverse P and Q in the list of surfaces sorted by maximum depth
  - set a flag to say that the test has been performed once.
  - If the tests fail a second time, then it is necessary to split the surfaces and repeat the algorithm on the 4 new split surfaces

Depth-sort algorithm

- Example:
  - We end up processing with order Q2,P1,P2,Q1



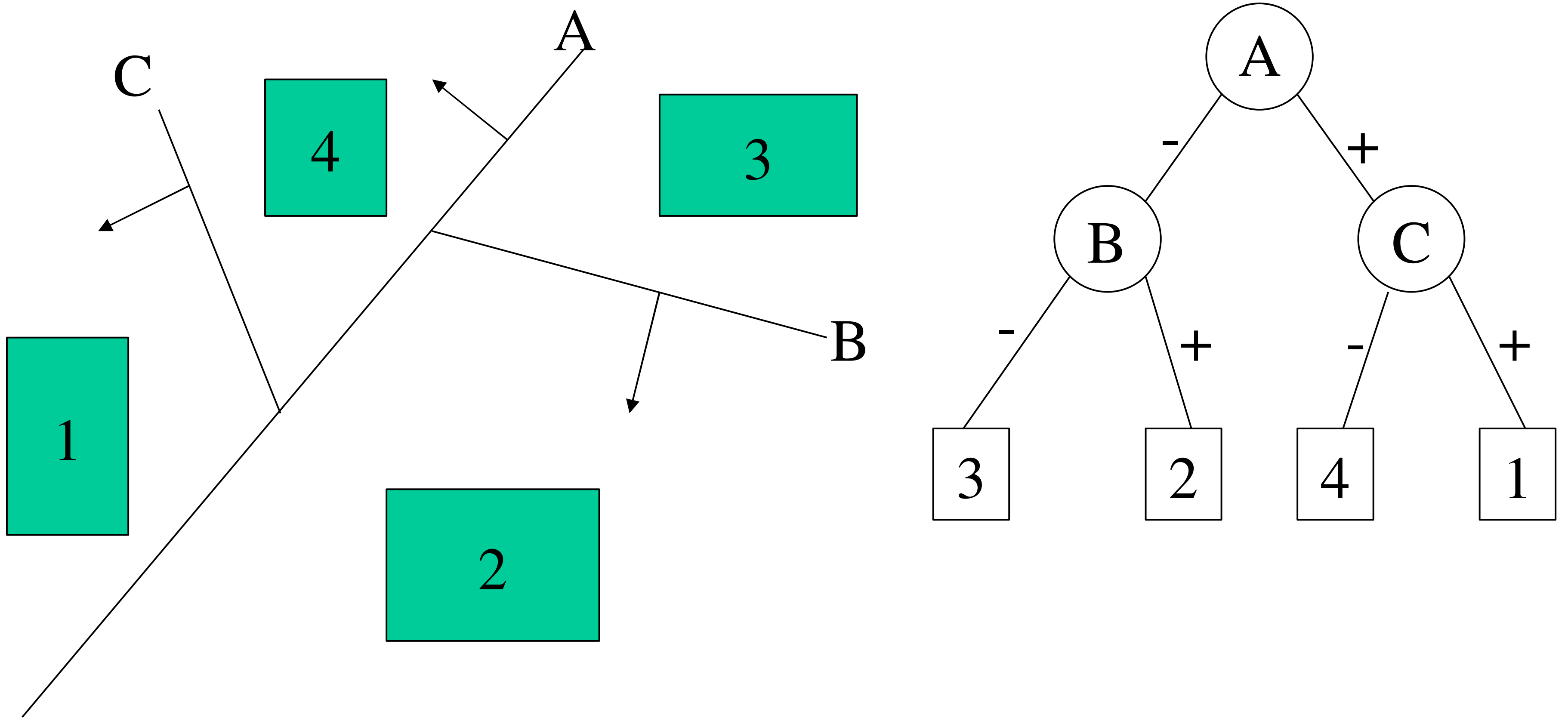
Binary Space Partitioning

- BSP tree: organize all of space (hence *partition*) into a binary tree
  - Tree gives a rendering order: correctly traversing this tree enumerates objects from back to front
- Tree splits 3D world with planes
  - The world is broken into convex cells
  - Each cell is the intersection of all the half-spaces of splitting planes on tree path to the cell
    - Splitting planes can be arbitrarily oriented

Building BSP-Trees

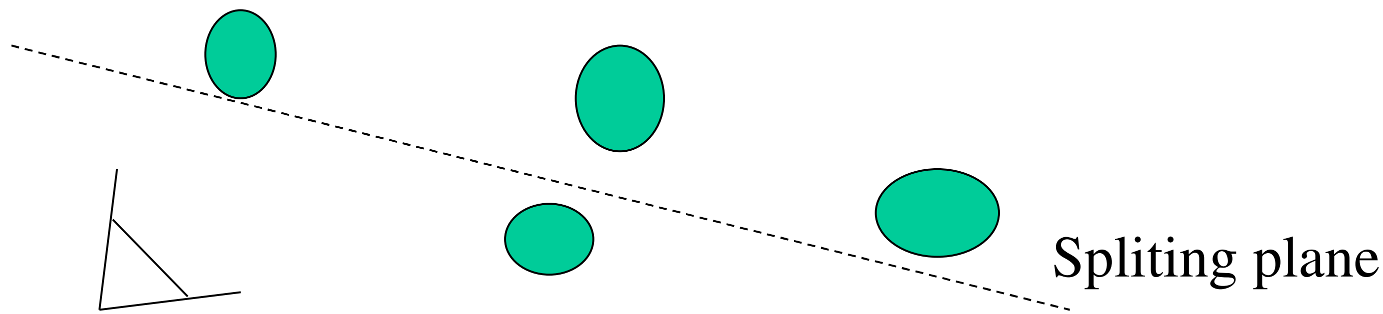
- Choose a splitting polygon (arbitrary)
- Split its cell using the plane on which the splitting polygon lies
  - May have to chop polygons in two (Clipping!)
- Continue until each cell contains only one polygon fragment (or object)

BSP-Tree Example



Using a BSP-Tree

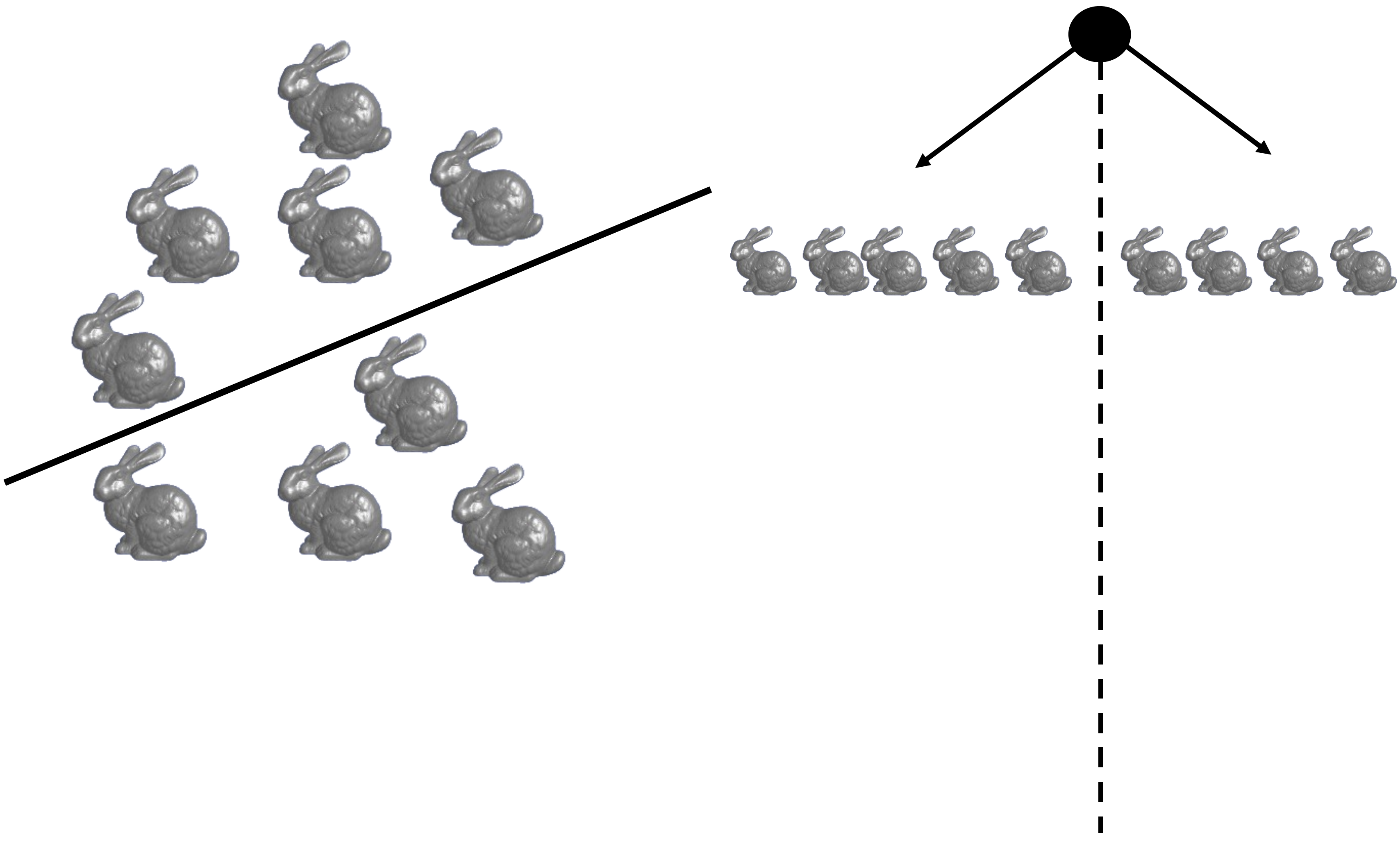
- Observation: Things on the opposite side of a splitting plane from the viewpoint cannot obscure things on the same side as the viewpoint



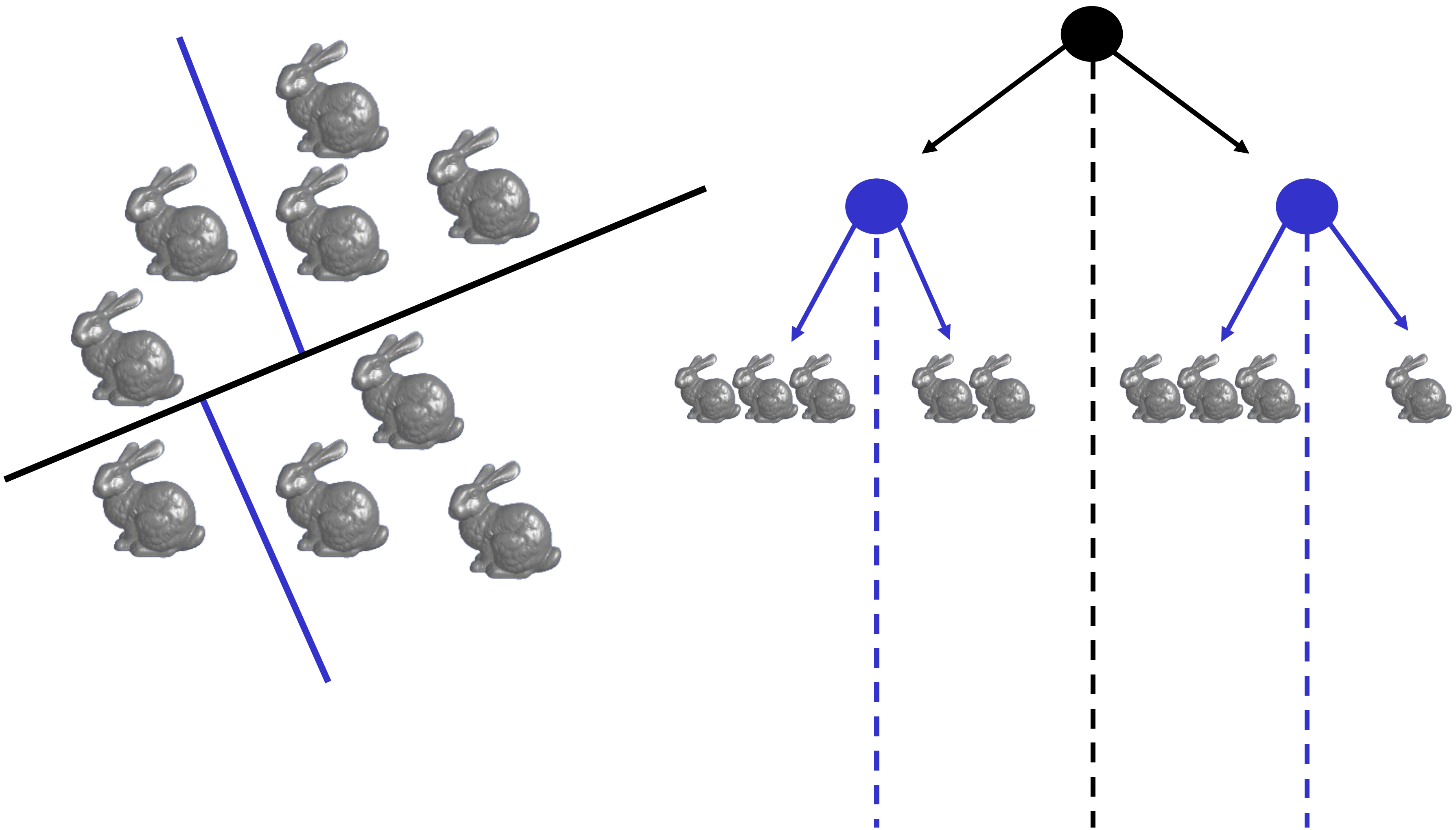
- This is a statement about rays: a ray must hit something on this side of the split plane before it hits the split plane and before it hits anything on the back side
- It is NOT a statement about distance – things on the far side of the plane can be closer than things on the near side
  - Gives a *relative* ordering of the polygons, not absolute in terms of depth or any other quantity

BSP Trees: Another example

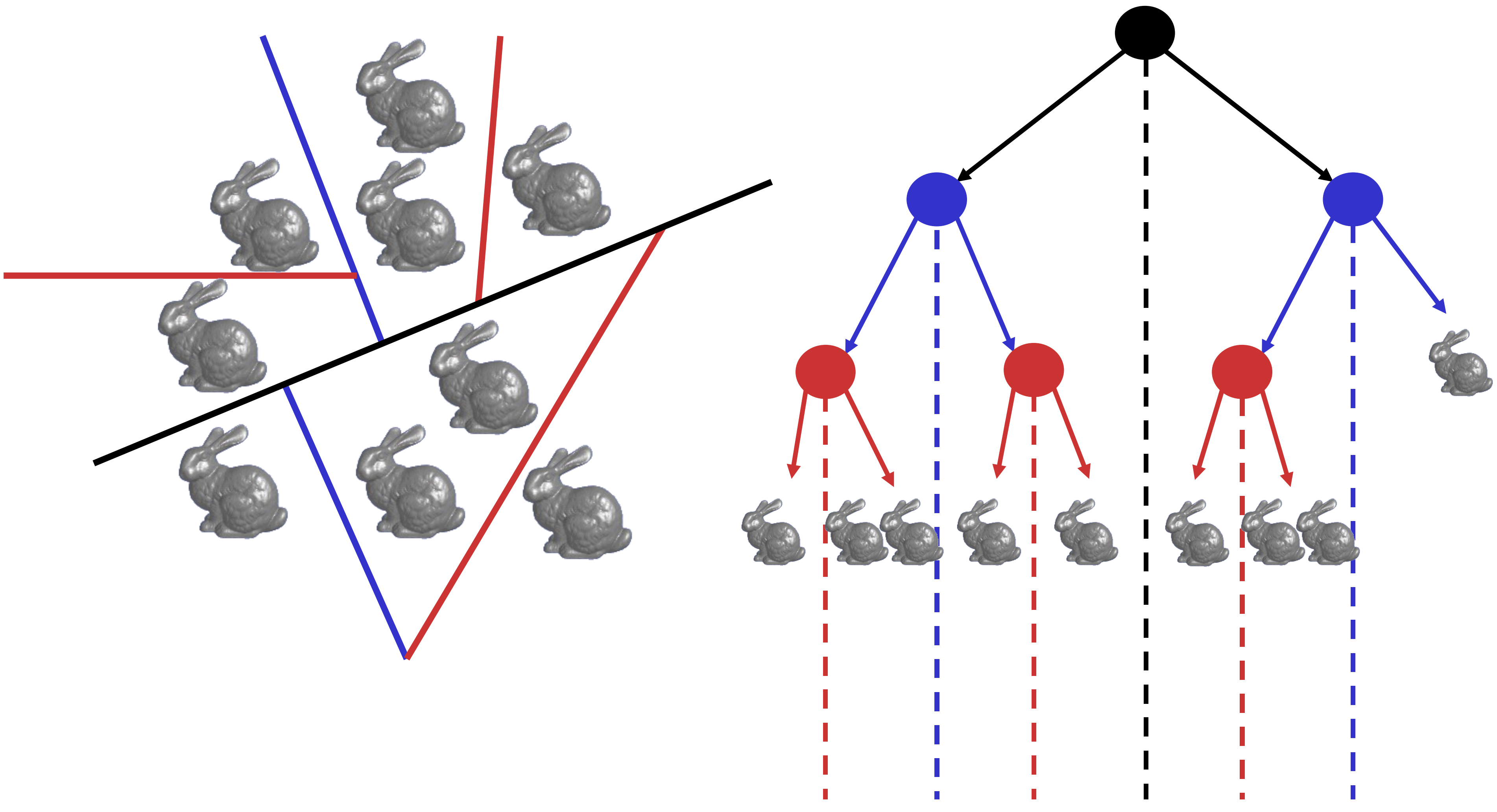
BSP Trees: Another example



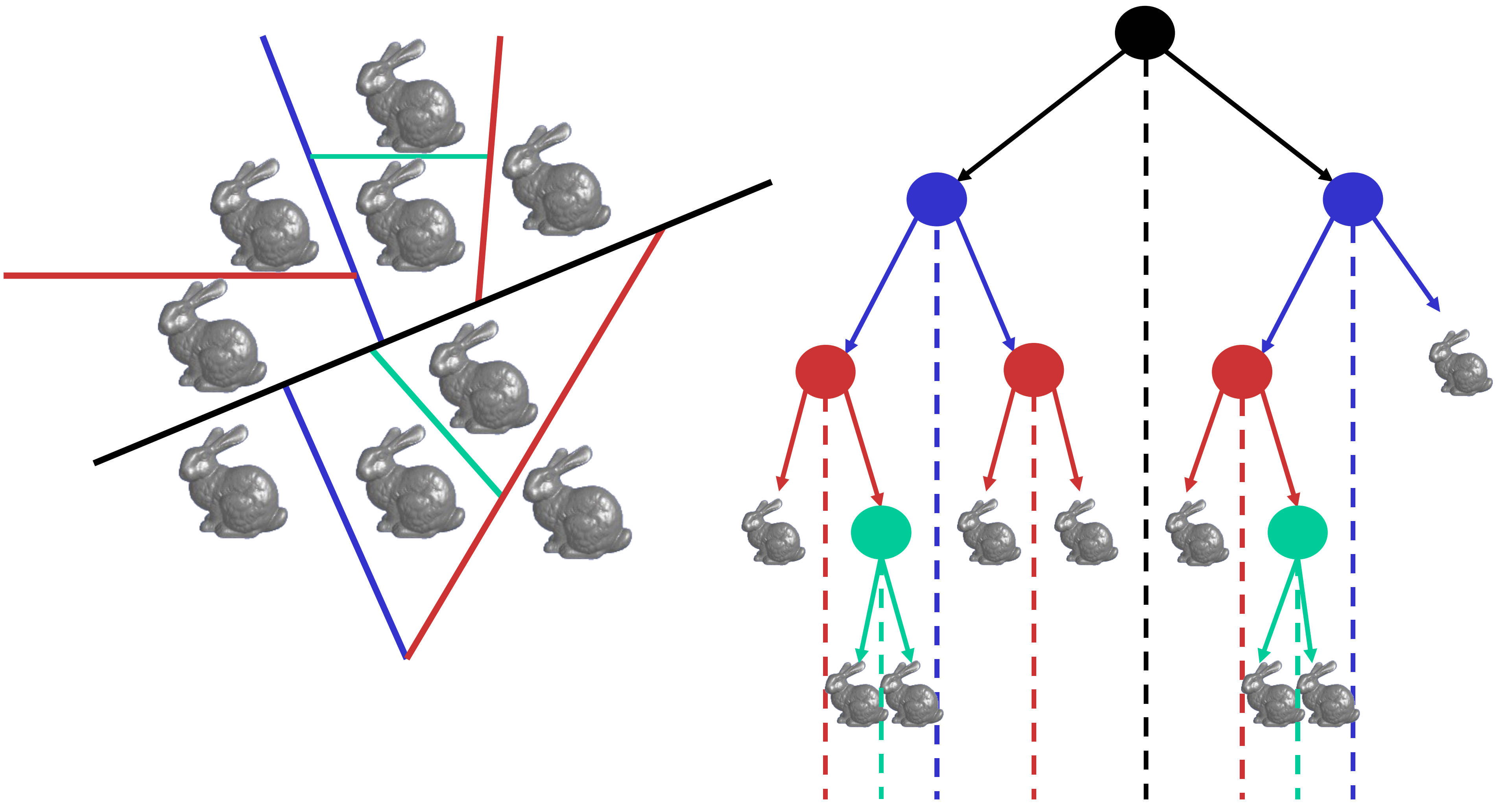
BSP Trees: Another example



BSP Trees: Another example



BSP Trees: Another example



Rendering BSP Trees

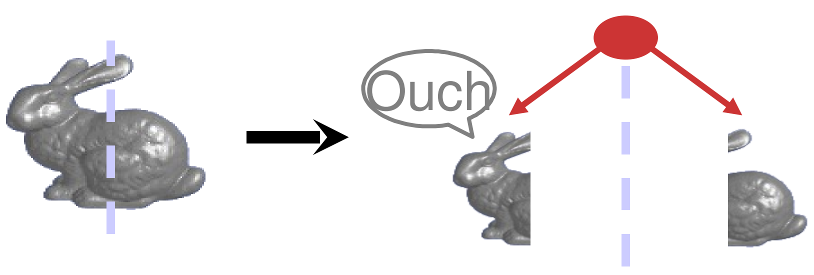
```
renderBSP(BSPtree *T)
   BSPtree *near, *far;
   if (T is a leaf node)
       { renderObject(T); return; }
   if (eye on left side of T->plane)
       near = T->left; far = T->right;
   else
       near = T->right; far = T->left;
   renderBSP(far);
  renderBSP(near);
```

BSP-Tree: Advantages

- One tree works for any viewing point

BSP-Tree: Disadvantages

- No bunnies were harmed in the example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node:



  - Worst case: can create up to $O(n^3)$ objects!

Final Topics

- You are responsible of all the topics we have covered throughout the semester.
- However, you may expect more questions on the subjects covered after the midterm:
  - Texture Mapping
  - 3D Object Representations
  - Illumination
  - Visible Surface Detection

Texture Mapping

- Texture mapping process
  - Specifying the texture image
  - Texture coordinates: (s,t) texture space
  - Magnification vs. minification: when do we need magnification/minification?

3D Object Representations

- Polygon representations
- Curved object representations:
  - Natural Cubic Splines
  - Hermite Curves
  - Bezier Curves
- Forward-difference calculations for cubic
- Scene-graph representations
- CSG
- Octrees
- Fractals

Illumination

- Light sources
  - Point, directional, spot lights
- Basic illumination model
  - Ambient, diffuse, specular components
- Surface rendering methods
  - Flat, Gouraud, and Phong shading

Visible Surface Detection

- Image Space versus Object Space methods
- Back-face detection
- Depth buffer algorithm
- Depth sorting algorithm
- Binary Space Partitioning

Format of the Final Exam

- Expect questions of similar type as in the midterm-exam