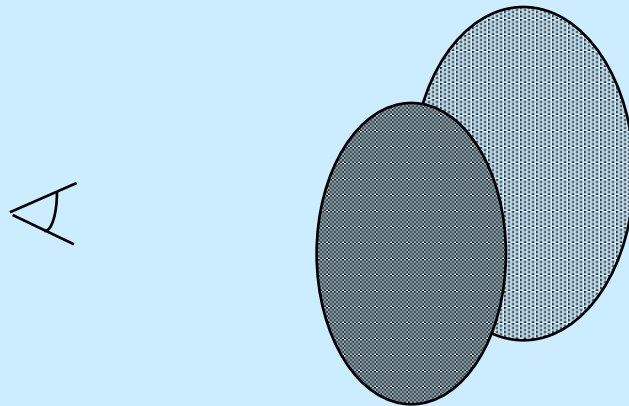


Computer Graphics: Hidden Surface Removal

**Dr. Michael Laszlo
Nova Southeastern University
School of Computer and Information Sciences
2002**

Hidden Surface Removal

- Given a 3D model to be rendered, *hidden surface removal* is the process of detecting and hiding those surfaces which are occluded or hidden by opaque surfaces nearer the viewer. The process is also called *visible surface determination*.



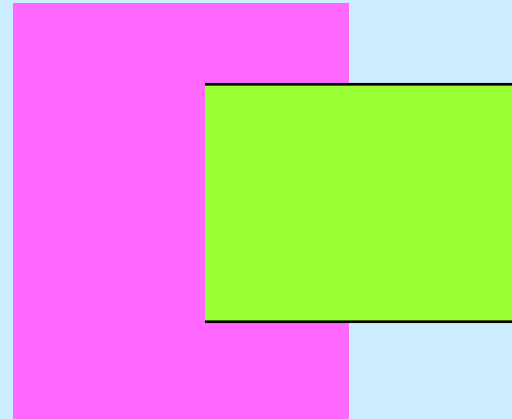
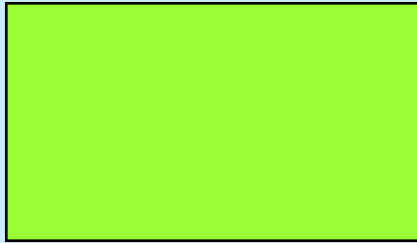
The lighter oval is partly hidden from view.

- We will examine several hidden surface removal algorithms:
 - Z-buffering, also known as the depth buffer algorithm.
 - The list priority method.
 - Recursive area subdivision.
 - Binary space partition trees.

Z-Buffering

- In addition to the frame buffer in which the image is stored, this method employs a *depth buffer* (or *z-buffer*) of the same resolution.
- Polygons are *conditionally* scan-converted into the frame buffer, based on the values stored in the depth buffer. Suppose polygon P covers pixel p_{ij} , and let p'_{ij} be the corresponding pixel in the z-buffer:
 - If the depth of P at p_{ij} is less than the depth of the last polygon painted into p_{ij} (that is, $z(P, i, j) < p'_{ij}$), then paint P into pixel p_{ij} and write the depth value $z(P, i, j)$ into pixel p'_{ij} of the frame buffer.
 - Otherwise do not change p_{ij} or p'_{ij} .
- The depth values are normalized to the range $0..2^d-1$, where the depth buffer is d bit-planes deep.

Z-Buffering



```
7 7 7 7 7 7 7 7 7 7 7
7 7 4 4 4 4 4 4 4 7 7
7 7 4 4 4 4 4 4 4 7 7
7 7 4 4 4 4 4 4 4 7 7
7 7 7 7 7 7 7 7 7 7 7
```

```
1 2 3 4 5 6 7 7 7 7 7
1 2 3 4 4 4 4 4 4 7 7
1 2 3 4 4 4 4 4 4 7 7
1 2 3 4 4 4 4 4 4 7 7
1 2 3 4 5 6 7 7 7 7 7
```

Here the z-buffer is 3 planes deep; 7 is maximum depth.

Z-buffering

```
// Here PaintColor paints into a pixel of the frame buffer,  
// and PaintZ writes a z-value into the depth buffer, and  
// ReadZ reads from the z-buffer.  
For y = 0 to Ymax  
  For x = 0 to Xmax {  
    PaintColor(x, y, BACKGROUND_COLOR);  
    PaintZ(x, y, Zmax)  
  }  
  
For each polygon P in scene  
  For each pixel (x,y) in polygon's projection {  
    pz = z-value of P at (x,y)  
    if (pz < ReadZ(x, y)) {  
      PaintZ(x, y, pz);  
      PaintColor(x, y, Color(P, X, Y));  
    }  
  }  
}
```

Z-buffering

- The depth values of a polygon can be computed efficiently while the projected polygon is scan-converted from left to right:

Plane equation of polygon: $Ax + By + Cz + D = 0$,

so $z = (-D - Ax - By) / C$.

Thus $z(x + dx, y) = (-D - A(x + dx) - By) / C$
 $= (-D - Ax - By) / C + A*dx / C$
 $= z(x, y) - (A/C) * dx$.

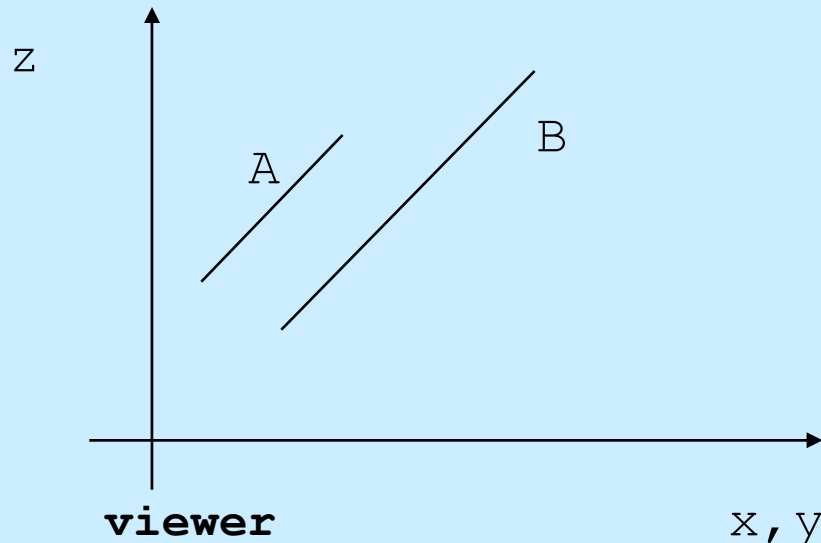
Hence knowing $z(x, y)$, **we can quickly compute** $z(x+dx, y)$.

Similarly, knowing $z(x, y)$, **we can compute** $z(x, y+dy)$:

$$z(x, y+dy) = z(x, y) - (B/C) * dy$$

List Priority Algorithm

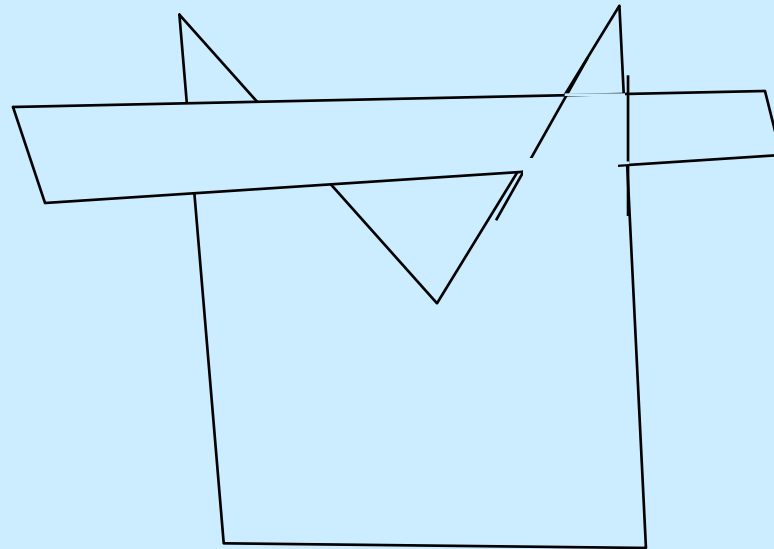
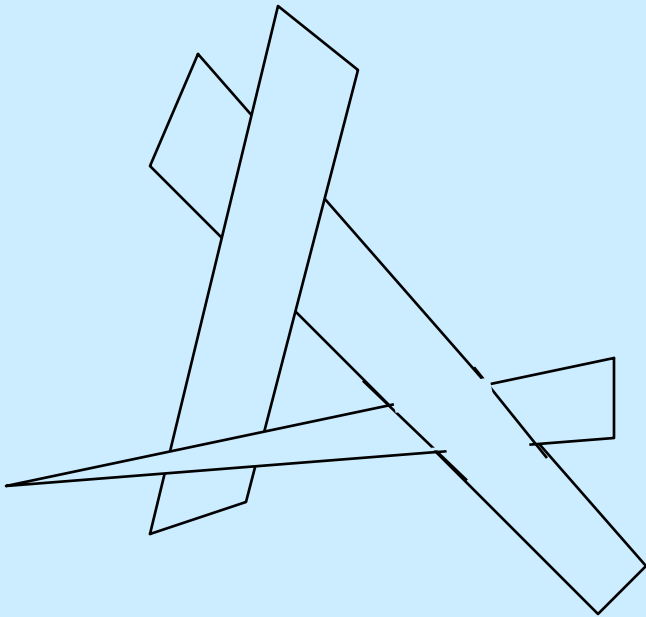
- Ideally we would like to obtain a linear ordering $<$ of polygons in space called a *depth-ordering*, and paint them in this order from far to near. In the depth-ordering $<$ below, we have $A < B$ if A does not obscure B . Thus it is safe to paint A before painting B .
- In the *painter's algorithm*, the ordering is obtained by sorting the polygons based on the deepest point z_{max} each attains. This always works if the polygons' z -extents are pairwise disjoint. Yet without this assumption, the painter's algorithm may fail, as illustrated in the following figure:



$A < B$, yet the painter's algorithm mistakenly paints B before A .

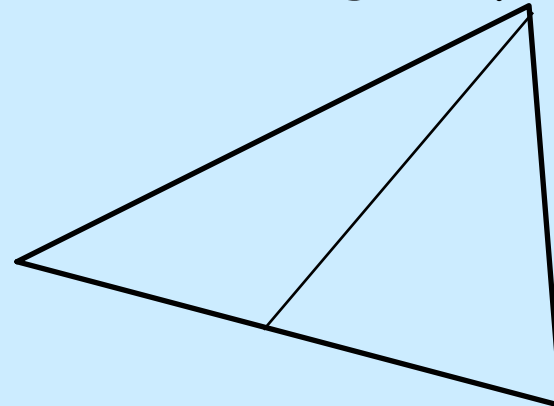
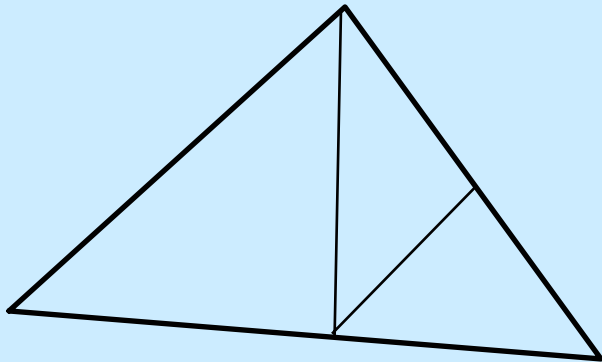
List Priority Algorithm

- In some situations, a depth-ordering does not even exist, since a cycle exists with respect to the obscuring relation. For instance, if polygon A obscures B , B obscures C , and C obscures A , then none of the three polygons can precede the other two in a valid depth-ordering, for each obscures one of the other two. Moreover, a depth-ordering is impossible if two polygons intersect.



List Priority Algorithm

- The *list-priority algorithm* of Newell, Newell, and Sancha first establishes a preliminary depth-ordering. It then iteratively performs *shuffle* and *split* operations to obtain a true depth-ordering.
- *Shuffle*: In a shuffle operation, two polygons P and Q swap places in the current depth-ordering. If P had been the candidate for the next polygon in the ordering under construction, then Q becomes the newest best candidate.
- *Split*: In a polygon split operation, a polygon Q is split by the plane of P , and the pieces of Q replace polygon Q in the current ordering. Note that no piece of Q can simultaneously obscure and be obscured by P . If we work exclusively with triangles, triangle Q is split into either two or three triangular pieces:

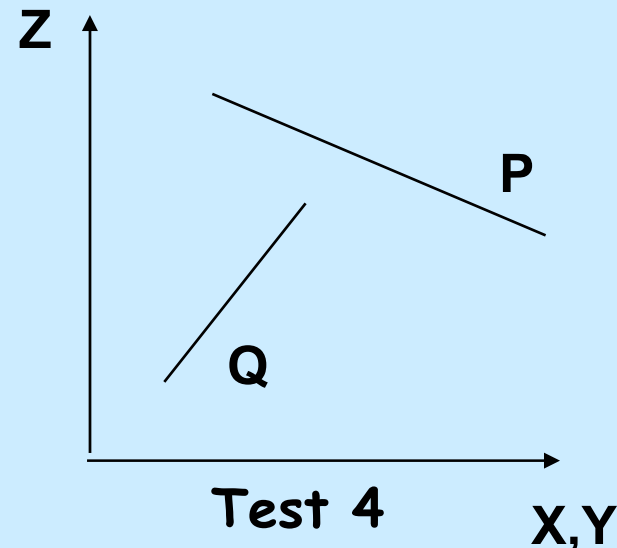
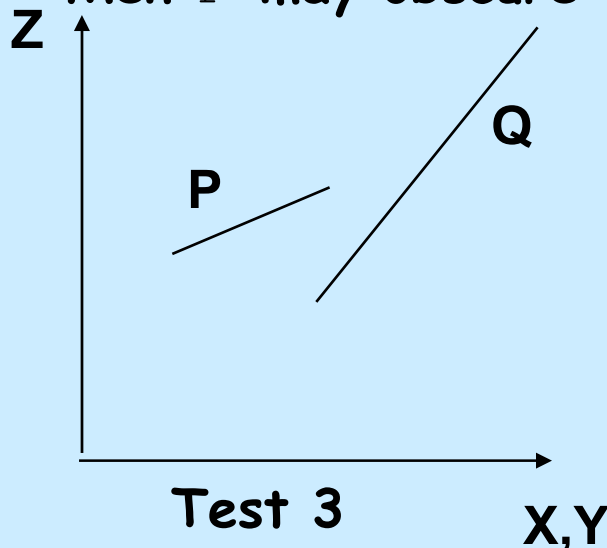


List Priority Algorithm

```
ListPriority(Set_of_polygons S) {
  Establish preliminary depth-ordering for S.
  While S is not empty {
    Let P be first polygon in S
    For each Q in S such that Z-extents of P and Q intersect {
      If MayObscure(P, Q) { // P may obscure Q
        If MayObscure(Q, P) { // Q may obscure P
          Split Q by the plane of P, replace Q by its pieces
        } else {
          Swap P and Q in S
          Go to start of While loop (Q becomes new P)
        }
      }
    }
    Paint P
  }
}
```

List Priority Algorithm

- $\text{MayObscure}(P, Q)$ returns true if P possibly obscures Q ; if the function returns false, then P does not obscure Q .
- $\text{MayObscure}(P, Q)$ is implemented by these five tests:
 1. Are the x -extents of P and Q non-overlapping?
 2. Are the y -extents of P and Q non-overlapping?
 3. Is P on the far side of Q ?
 4. Is Q on the near side of P ?
 5. Are the projections of P and Q disjoint?
- If any of the tests succeed, then P does not obscure Q (i.e., P can safely precede Q in the depth-ordering). If all 5 tests fail, then P may obscure Q .



List Priority Algorithm: An Example

$S = (A, B, C)$

$P = A.$

$\text{MayObscure}(A, B).$

$\text{Not MayObscure}(B, A):$ Shuffle A and B.

$S = (B, A, C)$

$P = B.$

$\text{Not MayObscure}(B, A).$

$\text{MayObscure}(B, C):$

$\text{Not MayObscure}(C, B):$ Shuffle B and C.

$S = (C, A, B)$

$P = C.$

$\text{MayObscure}(C, A).$

$\text{MayObscure}(A, C):$ Split A by C.

$S = (C, A1, A2, A3, B)$

$P = C.$

$\text{MayObscure}(C, A1).$

$\text{Not MayObscure}(A1, C):$ Shuffle C and A1.

$S = (A1, C, A2, A3, B)$

$P = A1.$

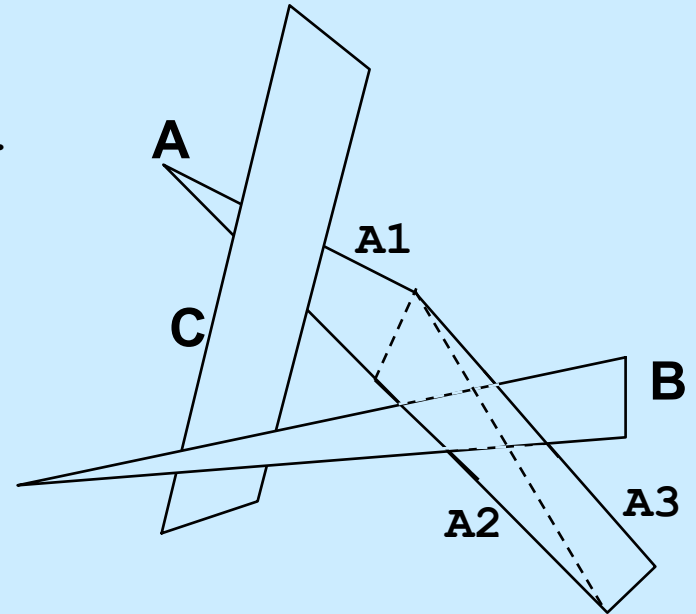
$\text{Not MayObscure}(A1, C).$

$\text{Not MayObscure}(A1, A2).$

$\text{Not MayObscure}(A1, A3).$

$\text{Not MayObscure}(A1, B).$

Paint A1.



List Priority Algorithm: An Example (cont.)

$S = (C, A2, A3, B)$

$P = C.$

Not MayObscure($C, A2$).

Not MayObscure($C, A3$).

Not MayObscure(C, B).

Paint $C.$

$S = (A2, A3, B)$

$P = A2.$

Not MayObscure($A2, A3$).

MayObscure($A2, B$).

Not MayObscure($B, A2$): Shuffle B and $A2$.

$S = (B, A3, A2)$

$P = B.$

Not MayObscure($B, A3$).

Not MayObscure($B, A2$).

Paint $B.$

$S = (A3, A2)$

$P = A3.$

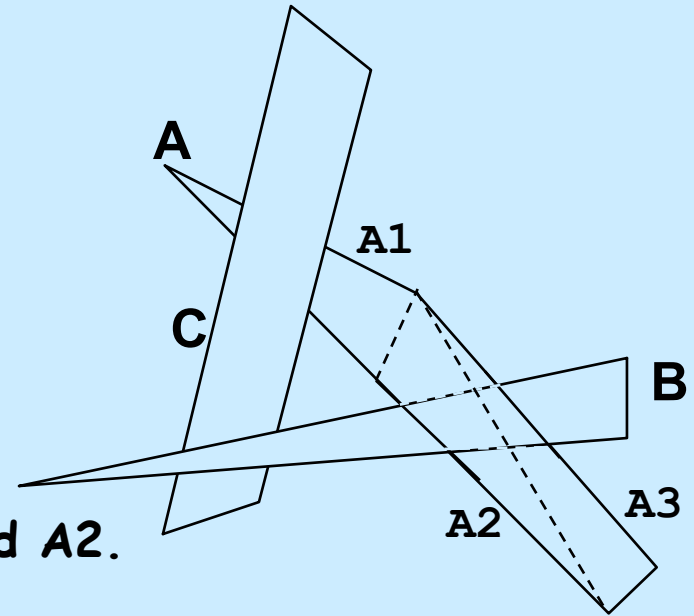
Not MayObscure($A3, A2$).

Paint $A3.$

$S = (A2)$

$P = A2.$

Paint $A2.$

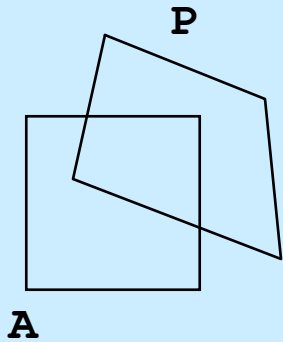


Recursive Area Subdivision

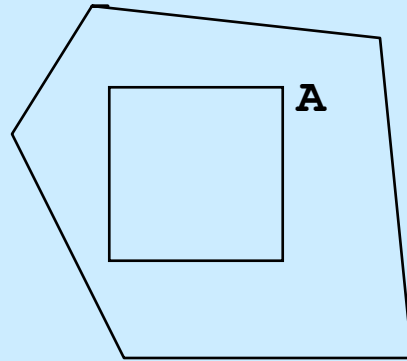
- *Recursive area subdivision*, also known as *Warnock's Algorithm*, is an image-precision algorithm. It answers the question: In each pixel, what surface (if any) is visible? We will consider the algorithm for a scene of polygons in space.
- The idea of the algorithm is to start with a square window that contains the entire image. Initially, this is the *current quadrant*. The current quadrant initially contains the projections of all the polygons in the scene.
- This is the general procedure:
 - If the contents of the current quadrant is simple enough or if the current quadrant is small enough, then display its contents.
 - Else subdivide the current quadrant into four quadrants and apply the algorithm recursively to these.
- Note that a quadtree subdivision results from this process.
- This algorithm exploits *area coherence*, the tendency for nearby areas of the image to appear the same, that is, to be spanned by the same sequence of polygons.

Recursive Area Subdivision

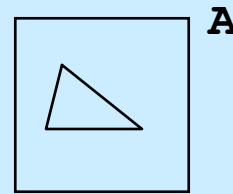
- The projection of a polygon P can have one of four possible relations to a square quadrant A of the image:



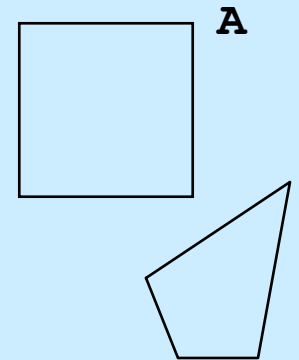
intersecting



surrounding



contained

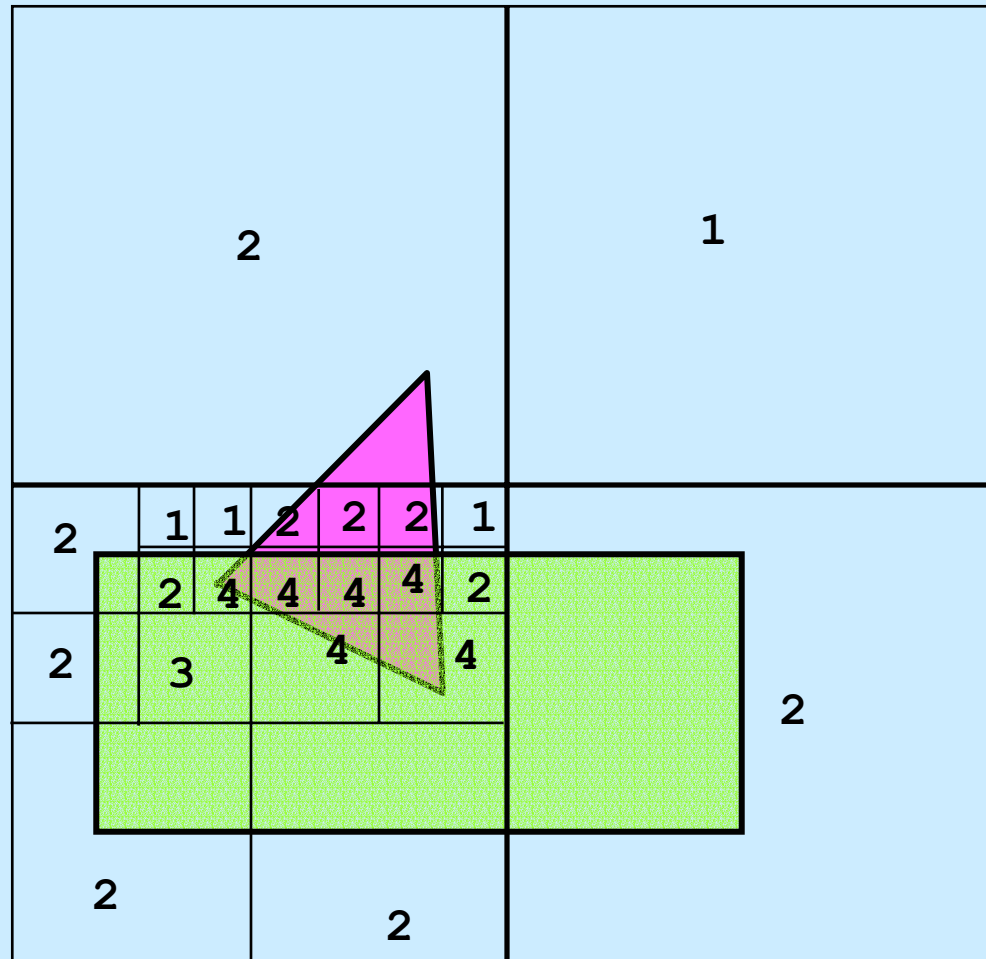


disjoint

Recursive Area Subdivision

- There are five cases in which we stop subdividing at area A :
 1. All polygons are disjoint from A .
 - Draw the background.
 2. There is exactly one intersecting or contained polygon P , and all others are disjoint.
 - Draw the background, then draw P clipped to A .
 3. There is exactly one surrounding polygon P , and all others are disjoint.
 - Draw P .
 4. There is one surrounding polygon P closer than all other interacting polygons.
 - Draw P .
 5. A is a single pixel.
 - Draw closest interacting polygon if any; else draw background.

Recursive Area Subdivision



The rectangle lies in front of the triangle. The numbers indicate which case applies in the given quadrant.

Binary Space Partition Trees

- Given a scene consisting of polygons in space, *binary space partition trees* (BSP trees) are useful for efficiently computing a depth-ordered list of the polygons from an arbitrary viewpoint.
- Given the BSP tree for a given scene, hidden surface removal for a given viewpoint is performed by traversing the BSP tree in such a way that a *depth-ordering* of the polygons is obtained. The polygons are then painted in depth-order from far to near, that is, using the painter's algorithm. An adaptive inorder (i.e., symmetric) traversal of the BSP tree is employed.
- The method is efficient enough that frames for a moving viewpoint can be generated in near-real time for relatively complex scenes. Construction of the BSP tree itself is costly, but this is a preprocessing task done once prior to rendering.

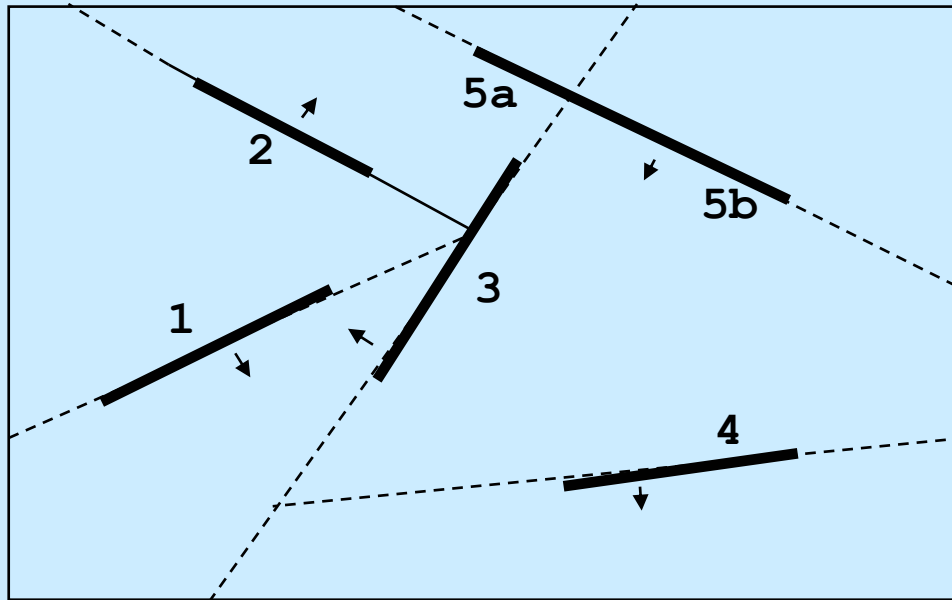
Binary Space Partition Trees

- A BSP tree is a binary tree constructed recursively over the collection of polygons in space. At the root of the tree is an arbitrary polygon P from the scene. The remaining polygons are separated into two classes:
 - class L , consisting of those polygons that lie to the left of the *plane*(P), and
 - class R , consisting of those polygons that lie to the right of the *plane*(P)
- Those polygons which are intersected by the plane of P are split by the plane into pieces, and the pieces are inserted into L and R as appropriate.
- Lastly, a BSP tree $BSP(L)$ is constructed recursively over L and a BSP tree $BSP(R)$ is constructed recursively over R . $BSP(L)$ is made the left subtree of root node P , and $BSP(R)$ is made the right subtree of P .

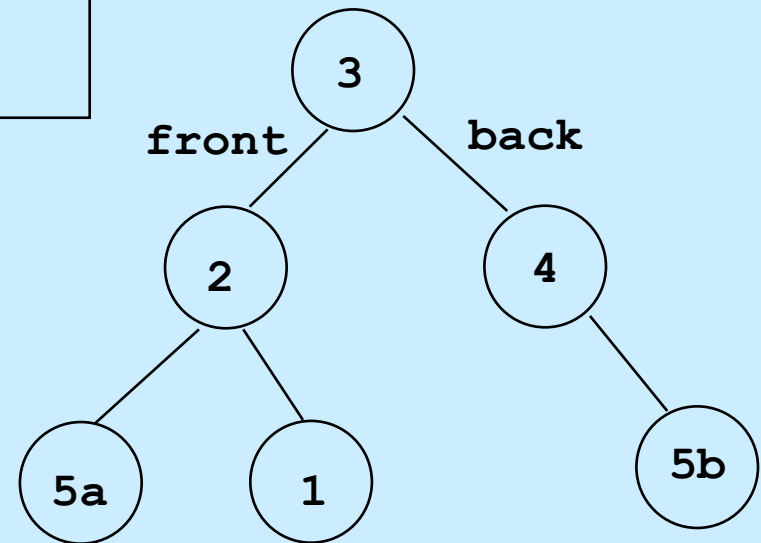
Binary Space Partition Trees

- The key to hidden surface removal: If viewpoint p is in front of $plane(n)$, then no polygon behind $plane(n)$ can obscure any polygon in front of $plane(n)$. Similarly, if viewpoint p is behind $plane(n)$, then no polygon in front of $plane(n)$ can obscure any polygon behind $plane(n)$.

Binary Space Partition Trees



This illustrates a 2D BSP tree: Line segments correspond to polygons, and lines to planes. The arrow points to the front side of a line segment.



Binary Space Partition Trees

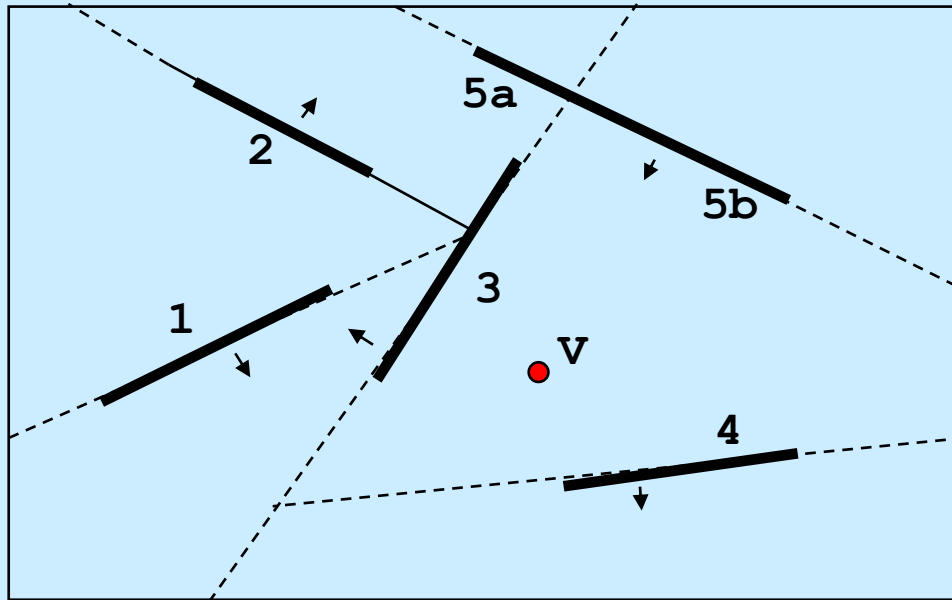
- Assume the BSP tree is built and that we wish to remove hidden surfaces for viewpoint v . We traverse the BSP tree to obtain the depth-sorted list of polygons.
- The traversal hinges on these observations. Suppose node n is the root node, and that viewpoint v is in front of $plane(n)$:
 - No polygon behind $plane(n)$ can obscure $poly(n)$ or any polygon in front of $plane(n)$. Thus it is safe to first recursively paint the polygons behind $plane(n)$.
 - $Polygon(n)$ cannot obscure any polygon in front of $plane(n)$, so next it is safe to paint $poly(n)$.
 - Finally, it is safe to recursively paint the polygons in front of $plane(n)$.
- The inorder traversal of the BSP tree proceeds symmetrically if v is behind $plane(n)$.

Binary Space Partition Trees

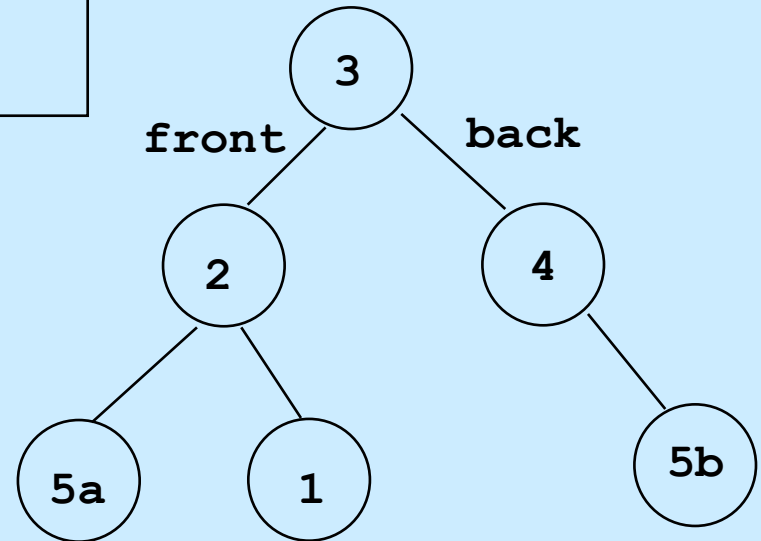
- To perform hidden surface removal from viewpoint v , we perform an adaptive inorder traversal of the BSP tree with root n . Which branch is followed first (left or right) depends on which side of $plane(n)$ the viewpoint v lies:

```
displayBSPTree(Node n, Point v) {
    if (n is not null) {
        if (infront(v, plane(n))) {
            displayBSPTree(rchild(n), v);
            paintPolygon(polygon(n));
            displayBSPTree(lchild(n), v);
        } else {
            displayBSPTree(lchild(n), v);
            paintPolygon(polygon(n));
            displayBSPTree(rchild(n), v);
        }
    }
}
```

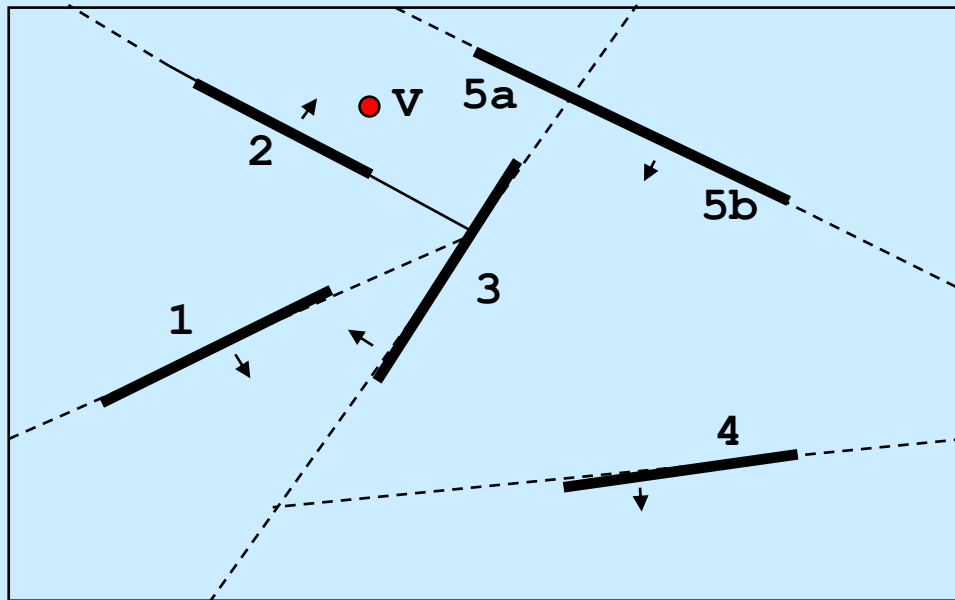
Binary Space Partition Trees



depth-ordering:
5a, 2, 1, 3, 4, 5b



Binary Space Partition Trees



depth-ordering:

4, 5b, 3, 1, 2, 5a

