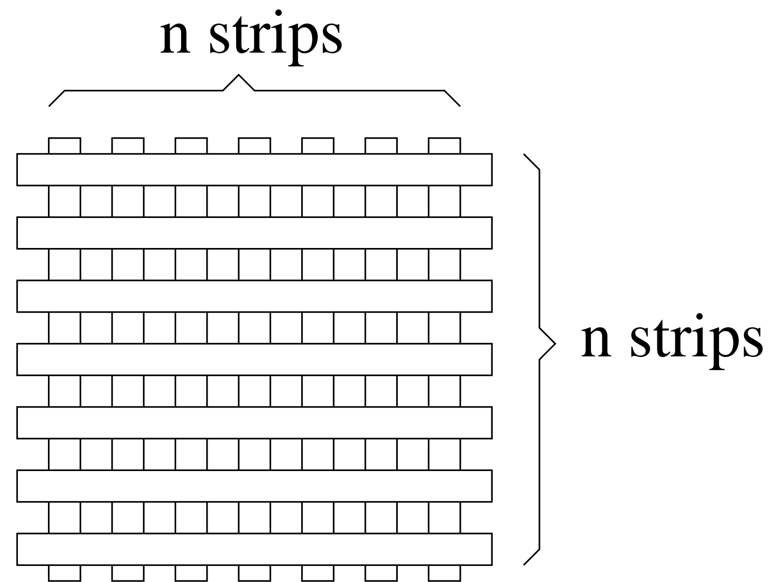
Visibility Algorithms I:

Visible Surface Determination:

We are given a collection of objects (represented, say, by a set of polygons) in 3-space, and a viewing situation, and we want to render only the visible surfaces. Each polygon face is assumed to be flat (although extensions to hidden-surface elimination of curved surfaces is an important problem). We may assume that each polygon is represented by a cyclic listing of the (x, y, z) coordinates of their vertices, so that from the "front" the vertices are enumerated in counterclockwise order.

Output:

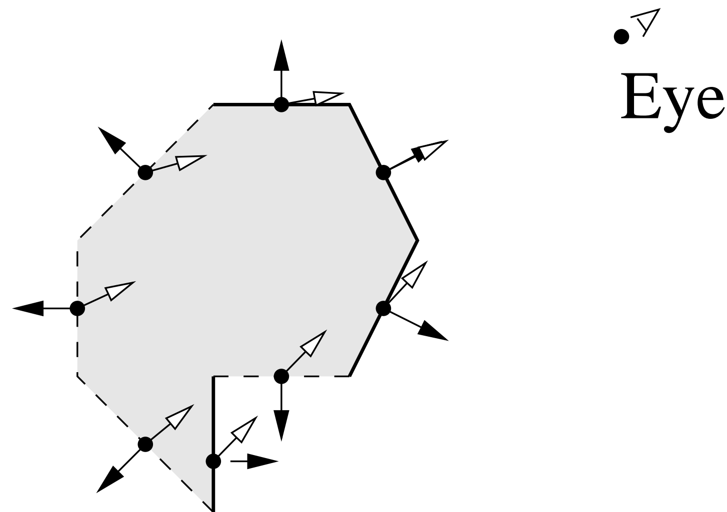
1. generate the set of pixels that form the final image.
2. generate a collection of polygons in the plane, whose interiors do not intersect, and which together form the final image.


n strips
n strips

Back-face culling

For each polygonal face, we assume an outward pointing normal can be computed. If this normal is directed away from the viewpoint, that is, if its dot product with a vector directed towards the viewer is negative, then the face can be immediately discarded from consideration.

On average this quick test can eliminate about one half of the faces from further consideration.

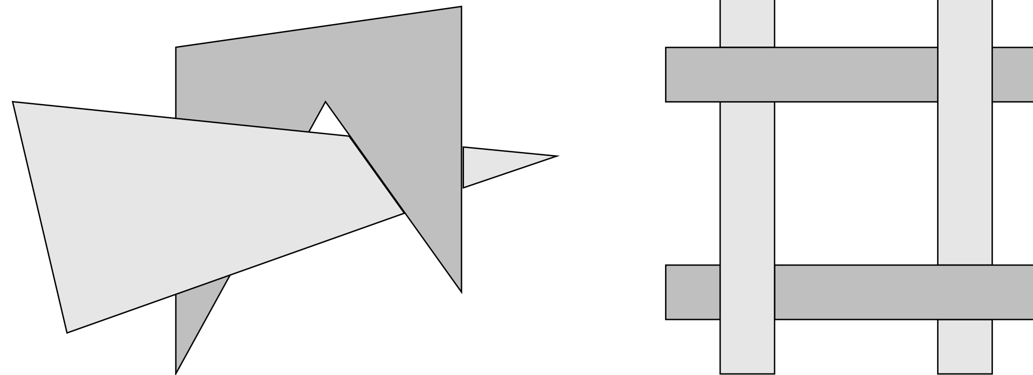


In OpenGL, the command glEnable(GL_CULL_FACE) can be used to enable this test.

Depth-Sort Algorithm

A fairly simple visible surface determination algorithm is based on the principle of painting objects from back to front, so that more distant polygons are overwritten by closer polygons. Sort all the polygons according to increasing distance from the viewpoint, and then scan convert them in reverse order (back to front). This depth-sort algorithm is sometimes called the *painter's algorithm* because it mimics the way that oil painters usually work (painting the background before the foreground).

Compute a *representative point* on each polygon (e.g. the centroid or the nearest point to the viewer), project these points saving the $-1/z$ distance information we used in our transformation, and then sort by decreasing order of distances, and draw in this order. Unfortunately, just because the centroids or nearest points are ordered, it does not imply that the entire polygons are ordered. Worse yet, it may not be possible to order polygons, as shown in the following figure.



In these cases we may need to *cut* one or more of the polygons into smaller polygons so that the depth order can be uniquely assigned. Also observe that if two polygons do not overlap in x, y space, then it does not matter much what order they are drawn in.

Depth-Sort Algorithm

We begin by sorting the polygons by depth. For example, the initial "depth" estimate of a polygon may be taken to be the closest z value of any vertex of the polygon. Let's take the polygon P at the end of the list. Consider all polygons Q whose z-extents overlaps P's. Before drawing P we make the following tests. If any test is passed, then we can assume P can be drawn before Q.

1. Do the x-extents not overlap?
2. Do the y-extents not overlap?
3. Is P entirely on the opposite side of Q's plane from the viewpoint?
4. Is Q entirely on the same side of P's plane as the viewpoint?
5. Do the projections of the polygons not overlap?

If all the tests fail then we split either P or Q using the plane of the other. The new cut polygons are inserting into the depth order and the process continues.

In theory this partitioning could generate $O(n^2)$ individual polygons, but in practice the number of polygons is much smaller.

Depth-buffer Algorithm

The depth-buffer algorithm is one of the simplest and fastest visible surface algorithms. Its main drawbacks are that it requires a lot of memory, and that it only produces a result that is accurate to pixel resolution and the resolution of the depth buffer. Thus the result cannot be scaled easily and edges appear jagged (unless some effort is made to remove these effects called "aliasing"). It is also called the **z-buffer** algorithm.

This algorithm assumes that for each pixel we store two pieces of information:

1. the color of the pixel,
2. the depth of the object that gave rise to this color.

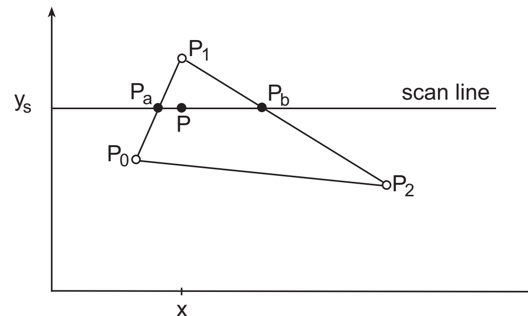
This is called the *depth-buffer* (or *z-buffer*, since z is the axis used to store depth information). Initially the depth-buffer values are set to the maximum depth value.

Suppose that we have a k-bit depth buffer, implying that we can store integer depths ranging from 0 to $D = 2^k - 1$. After applying the perspective-with-depth transformation, we know that all depth values have been scaled to the range $[-1, 1]$. If this depth is less than or

equal to the depth at this point of the buffer, then we store its RGB value in the color buffer. Otherwise we do nothing.

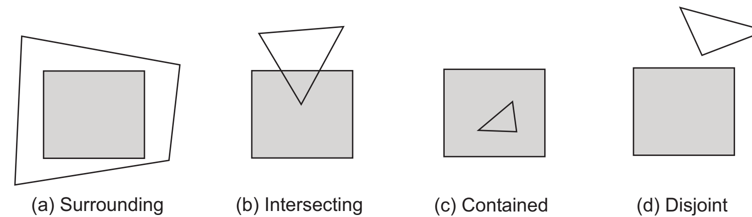
This algorithm is favored for hardware implementations because it is so simple and essentially reuses the same algorithms needed for basic scan conversion. The only problem that remains is how to interpolate the depth values for the pixels in the triangle. This is a simple exercise involving barycentric coordinates.

Let P_0, P_1, and P_2 be the vertices of the triangle after perspective-plus-depth transformation has been applied, and the points have been scaled to the screen size. Let $P_i = (x_i, y_i, z_i)$ be the coordinates of each vertex, where (x_i, y_i) are the final screen coordinates and z_i is the depth of this point.



Warnock's Algorithm

The area-subdivision algorithm developed by Warnock subdivides each area into four equal squares. In the recursive-subdivision process, four relations of polygon projections occur to a shaped square area element (see Figure):



1. *Surrounding polygons* completely contain the (shaded) area of interest.
2. *Intersecting polygons* intersect the area
3. *Contained polygons* are completely inside the area
4. *Disjoint polygons* are completely outside the area.

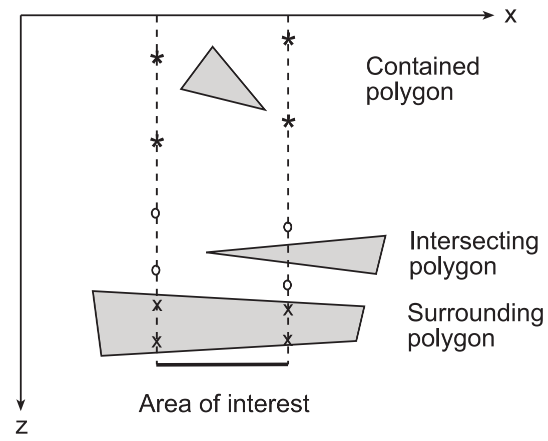
- Disjoint polygons clearly have no influence on the area of interest.

- Part of an intersecting polygon that is outside the area is also irrelevant.
- Part of an intersecting polygon that is interior to the area is the same as a contained polygon.

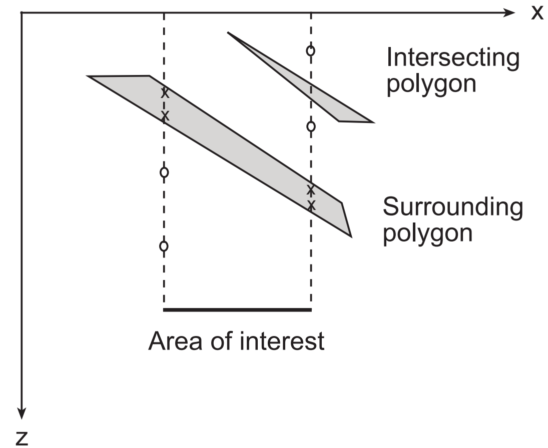
In each of the following four cases, a decision about an area can be made easily, so the area does not need to be divided further to be conquered:

1. All the polygons are disjoint from the area. The background color can be displayed in the area.
2. There is only one intersecting or only one contained polygon. The area is first filled with the background color, and then part of the polygon contained in the area is scan-converted.
3. There is a single surrounding polygon, but no intersecting or contained polygons. The area is filled with the color of the surrounding polygon.
4. More than one polygon is intersecting, contained in, or surrounding the area, but one is a surrounding polygon that is in front of all the other polygons. Determining whether a surrounding polygon is in front is done by computing the z coordinates of the planes of all surrounding, intersecting, and contained polygons. The entire area can be filled with the color of this surrounding polygon.

Cases 1, 2, and 3 are simple to understand. Case 4 is further illustrated in the following Figure.



(a)



(b)

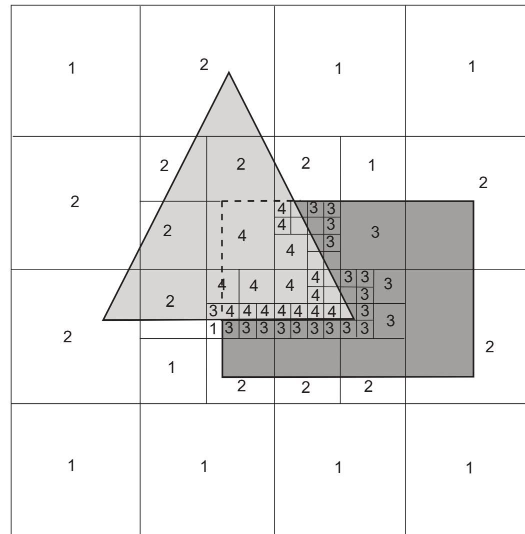
- In part (a), the four intersections of the surrounding polygon are all closer to the viewpoint (which is at infinity on the $+z$ axis) than are any of the other intersections. Consequently, the entire area is filled with the surrounding polygon's color.
- In part (b), no decision can be made, even though the surrounding polygon seems to be in front of the intersecting polygon, because on the left the plane of the intersecting polygon is in front of the plane of the surrounding polygon. After subdivision, only contained and intersecting polygons need to be reexamined: Surrounding and disjoint polygons of the original area are surrounding and disjoint polygons of each subdivided area.

Up to this point, the algorithm has operated at **object precision**, with the exception of the actual scan conversion of the background and clipped polygons in the four cases.

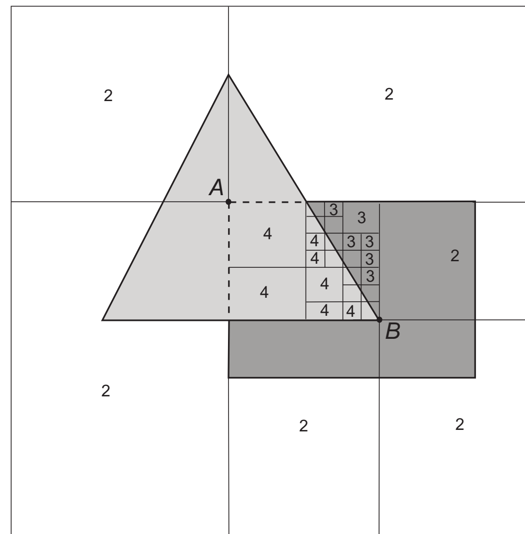
These image-precision scan-conversion operations, however, can be replaced by object-precision operations that output a precise representation of the visible surfaces: either a square of the area's size (cases 1, 3, and 4) or a single polygon clipped to the area, along with its Boolean complement relative to the area, representing the visible part of the background (case 2).

What about the cases that are not one of these four?

One approach is to stop subdividing when the resolution of the display device is reached. Thus, on a 1024 by 1024 raster display, at most 10 levels of subdivision are needed. If, after the maximum number of subdivisions, none of cases 1 to 4 have occurred, then the depth of all relevant polygons is computed at the center of this pixel-sized, indivisible area. The polygon with the closest z coordinate defines the shading of the area. Alternatively, for antialiasing, several further levels of subdivision can be used to determine a pixel's color by weighting the color of each of its subpixel-sized area by its size. It is these image-precision operations, performed when an area is not one of the simple cases, that makes this an **image-precision** approach.



The above figure shows a simple scene and the subdivisions necessary for that scene's display. The number in each subdivided area corresponds to one of the four cases; in an unnumbered area, none of the four cases are true. An alternative to equal-area subdivision, shown in the following Figure, is to (adaptively) divide about the vertex of a polygon (if there is a vertex in the area) in an attempt to avoid unnecessary subdivisions.


2
2
A
4
3
3
4
3
3
4
3
2
3
4
4
4
4
B
2
2
2

Reading Assignment and News

Pages 260 - 262, 361 - 363, of Recommended Text.

Please also track the News section of the Course Web Pages for the most recent Announcements related to this course.

(http://www.cs.utexas.edu/users/bajaj/graphics25/cs354/)