1. Introduction

We have seen the following tasks that can be done by the display processor:
- rasterization of line segment (scan conversion of line)
- clipping of line segment
- clipping of polygon
Today we will examine some more algorithms
- area filling of polygon
- hidden-surface removal

Area-Filling Methods:
- display solid color or pattern within a designated area
- two general approaches
  1. flooding type: painting from an interior point out to boundary edges
  2. scan line: identify interior points of the area along each scan line

2. Boundary Fill Algorithm
- useful with painting programs and interactive input

Input:
(1) coordinates (x,y) of an interior point
(2) fill color
(3) boundary color

idea: from point (x,y), paint interior with fill color up to boundary

4-connected areas: (N, S, E, W)
- good for simple shapes
8-connected areas: (N, S, E, W, NE, NW, SE, SW)
- can handle regions with narrow (one pixel wide) diagonal sections
void boundaryFill4 (int x, int y, int fill, int boundary) {
    int current;
    current = getPixel (x, y);
    if ( (current != boundary) && (current != fill)) {
        setColor (fill);
        setPixel (x, y);
        boundaryFill4 (x+1, y, fill, boundary);
        boundaryFill4 (x-1, y fill, boundary);
        boundaryFill4 (x, y+1, fill, boundary);
        boundaryFill4 (x, y-1, fill, boundary);
    }
}

Note: both methods involve large stacks—can improve performance by combining boundary-fill concepts with scan line approach, stack only a few points on each scan line.

3. Flood-Fill Algorithm
   - variation on boundary-fill method  
   - useful for regions that may have multicolored boundary (e.g. adjoining patterned areas)
Input:
(1) coordinates (x,y) for interior point;
(2) fill color;
(3) current interior color (to be replaced)

void floodFill4 (int x, int y, int fillColor, int oldColor) {
    if ( getPixel(x, y) == oldColor ) {
        setColor (fillColor);
        setPixel (x, y);
        floodFill4 (x+1, y, fillColor, oldColor);
        floodFill4 (x-1, y, fillColor, oldColor);
        floodFill4 (x, y+1, fillColor, oldColor);
        floodFill4 (x, y-1, fillColor, oldColor);
    }
}
14. Scan-line Algorithm
- for filling in an area when only area boundary is specified.
- basic idea: for each scan line that intersects area, determine interior points and set raster values

Scan line processed one-at-a-time from top line to bottom line.
Procedure:
- along each scan line, determine intersections with polygon edges
- sort intersection points from left to right
- set pixel intensities between each intersection pair.

Difficulties:
- intersection at a polygon vertex can cause problems:
  - sometimes results in an odd number of intersection points on a scan line

To resolve this, note that odd number of intersections is generated when adjoining lines(edges) are on opposite sides of scan line (as in the left picture). Therefore, when y values of the two edges are monotonically increasing or decreasing, count vertex as 1 intersection.
Otherwise (local extreme case), count vertex as 2 intersection.

Implementation: an easy way to handle problem vertices is to shorten one of the adjoining lines.

Assume edges processed in clockwise order:

This ensures that vertex is counted as only one intersection point.
To efficiently process each scan line, use
(1) sorted polygon edge List
(2) coherence properties

**Sorted polygon edge list**

- specify edges in order and set up sorted list
- pointers indicate active sublist for scan line

Coherence Methods

Properties of adjacent pixels in a scene are often related. Thus can determine properties of a pixel from previously processed pixels (coherence properties).

Along a scan line, pixel intensities may be constant, linearly varying etc.

Similar “coherence” methods can be used to calculate edge intersections from one scan line to next.
Slope of polygon edge is
\[ m = \frac{(y_{k+1} - y_k)}{(x_{k+1} - x_k)} \]

and next intersection point has coordinates:
\[ y_{k+1} = y_k - 1, \quad x_{k+1} = x_k - \frac{1}{m} \]

(to avoid accumulation of roundoff error, calculate: \( x_k = x_0 - \frac{k}{m} \))

IMPLEMENTATION DETAILS OMITTED HERE.
Hidden surface removal

1. Introduction

Whenever a picture contains opaque objects, those that are closer to the viewer and in the line of sight of other objects will block or occlude those further objects from view. Blocked or hidden lines/surfaces must be removed to provide realistic images.

Possibilities for 2 polygons:

What about this??

2. Object-precision vs image-precision algorithms

Object-precision algorithms:
- Considering the objects pairwise, as seen from the center of projection
- Computation done in object coordinates at object precision level. Algorithms determine visibility of each object without regard to a particular display resolution.

For each object in the world do
  Begin
    Determine those parts of the object whose view is unobstructed by other parts of it or any other object;
    Draw those parts in the appropriate color
  End

Image-precision algorithms:
- follow our viewing and ray-casting model by considering a ray that leaves the center of projection and passes through a pixel.
- computation done only at screen resolution and determine the visibility of each pixel; more efficient in practice
- algorithm operates only on sampled data, usually with aliasing problem

For each pixel in the image do
Begin
  Determine the object closest to the viewer that is pierced by the projection through the pixel;
  Draw the pixel in the appropriate color
End

3. Back-Face Removal

For a polygonal model, we see the front of a polygon if the normal, which comes out of the front face, is pointed toward the viewer.

If \( \theta \) is the angle between the normal and the viewer, then the polygon is facing forward iff \( \theta \) is between \(-90\) and \(+90\) degree, or equivalently, \( \cos \theta \) is positive.
We can use dot product \( n \cdot v \geq 0 \) to check for positive \( \cos \theta \).

Note: back-face culling is usually applied before any other hidden-surface-removal algorithm.

4. Z-buffer algorithm (Catmull 1975)
Idea: Storing a depth value at every pixel.
(Image-precision, point sampling algorithm (per-pixel base)).

- Maintain two screen-sized arrays:
  1) color buffer Color\((x, y)\) for storing color value for each pixel
  2) depth or z-buffer \(Z(x, y)\) for storing nearest perspective z-value found so far
• Polygons (in eye coord sys), in random order, are examined to determine which pixels they project to. The pixel in both buffers are updated if the z-value is front most.

```c
for (y = 0; y < YMAX; y++) {
    for (x = 0; x < XMAX; x++) {
        WritePixel (x, y, BACKGROUND_VALUE);
        WriteZ (x, y, BACKGROUND_Z);
    }
}
```

```c
for (each polygon) {
    for (each pixel in polygon’s projection) {
        pz = polygon’s z-value at pixel coords (x, y);
        if (pz <= ReadZ(x, y)) /* new point is not further */
            WriteZ (x, y, pz);
        WritePixel (x, y, polygon’s color at (x, y));
    }
}
```

**Scanline Coherence:**
Make use of scanline coherence to obtain depth value for other pixels within the polygon:
Plane equation: \(ax+by+cz+d = 0\),
\[z = -\frac{ax + by + d}{c}\]
For a given pixel at \((x, y)\), if \(z = z_1\) then
\[z (x + \Delta x, y) = z_1 - \frac{\Delta x}{c} a/c ;
\]
\[z (x, y + \Delta y) = z_1 - \frac{\Delta y}{c} .\]

*These values \(a/c\) and \(b/c\) need to be computed only once for each polygon.*

**Z-buffer algorithm:**
Advantages:
- No pre-sorting is necessary, no explicit intersection algorithm is needed, and no object-object comparisons are required.
- Can render any object as long as a shade value (color) and a z-value can be determined for each point in the projection. (If shading computation is
expensive, perform a front-to-back depth sort of the objects to display closest object first).

- Entire process is no more than a search over each set of pairs \(\{Z(x,y), \text{Color}(x,y)\}\) for fixed \((x,y)\) to find the smallest \(Z\).

Disadvantages:
- Can handle only opaque objects.
- a single point sampling process ⇒ severe aliasing.

5. Scan-line algorithm

text 7.7.5.

Idea:
- avoid depth calculation until necessary
- do depth calculation when hitting 2 or more polygons in a scan line
- working on one scan line at a time, rather than one polygon at a time.
- need moderately sophisticated data structure for representing which edges are encountered on each scan line.

6. Depth Sort Algorithm (Newell, Newell, Sancha)

Idea: paint the polygons into the frame buffer in order of decreasing distance from the viewpoint.

Three Conceptual steps are performed:
1. Sort all polygons according to the smallest (farthest) \(z\)-coordinate of each
2. Resolve any ambiguities this may cause when the polygons’ \(z\) extents overlap, splitting polygons if necessary
3. Scan convert each polygon in ascending order of smallest \(z\) coordinate (i.e. back to front...like a painter)

Some details of step 2 are as follows.
(a) do the polygon’s \(x\) extents not overlap?
(b) do the polygons \(y\) extents not overlap?
(c) is \(P\) entirely on the opposite side of \(Q\)’s plane from the viewpoint?
(d) is \(Q\) entirely on the same side of \(P\)’s plane as the viewpoint?
(e) do the projections of the polygons onto the \((x,y)\) plane not overlap (this can be determined by comparing the edges of one polygon to the edges of the other)
Video Clips
(1) Volker Blanz and Thomas Vetter. Face reconstruction, Siggraph 99
(2) Teddy, siggraph 99 (www.alice.org)