Lecture 4: Filled Area Primitives

Shobhanjana Kalita,

Dept. of CSE, Tezpur University

Filled Area Primitives

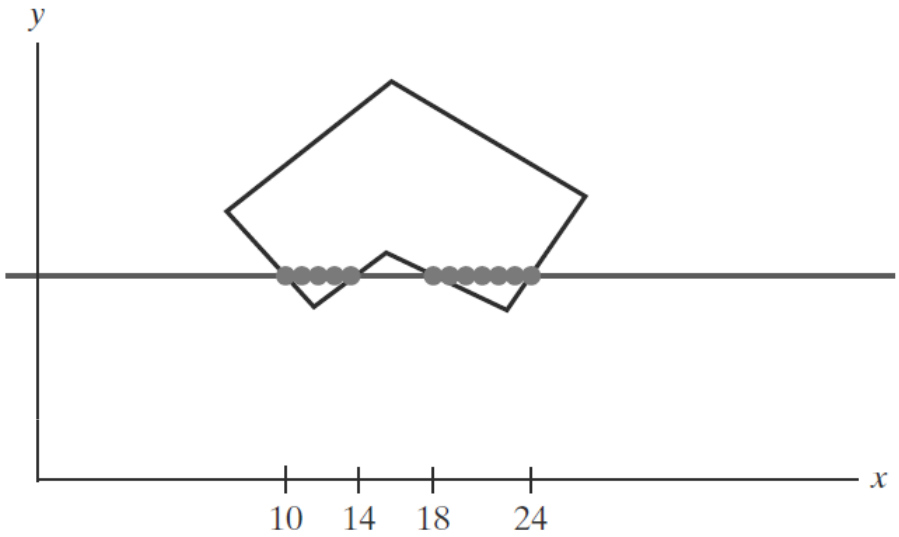
- 2 basic approaches to area filling on raster systems
- Determine overlap intervals for scan lines that cross the area
  - Typically used in general graphics packages to fill polygons, circles, ellipses, and other simple curves.
- Start from a given interior position and paint outward from this point until we encounter the specified boundary conditions
  - Useful with more complex boundaries and in interactive painting systems

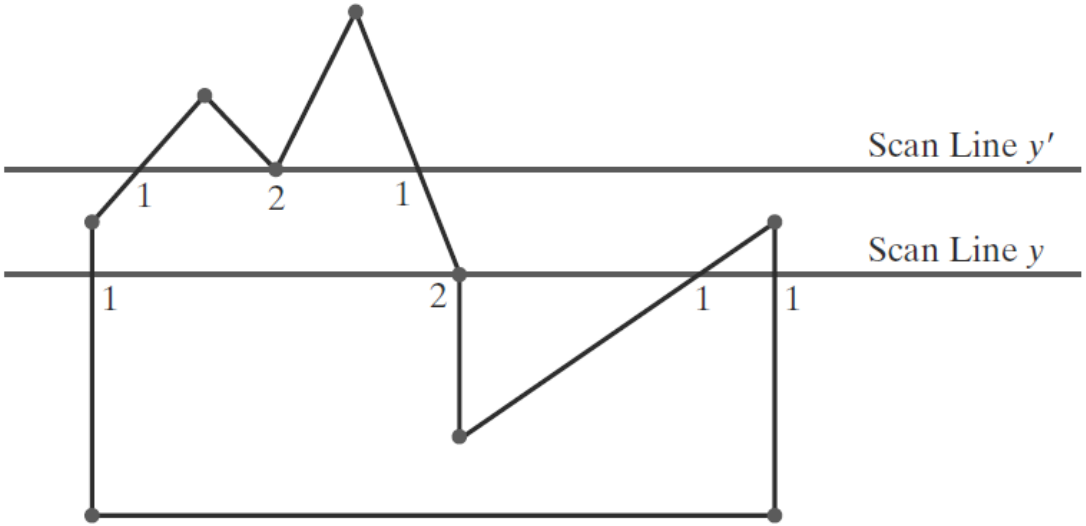
Scan Line Polygon Fill

- **Scan line** is one line, or row, in a raster scanning pattern
  - Such as a line of video on a cathode ray tube (CRT) display of a television set or computer monitor
- For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges
  - sorted from left to right, and corresponding frame-buffer positions between each intersection pair are set to fill colour

Scan Line Polygon Fill

In case of odd number of intersections, count the point which is a vertex as two points

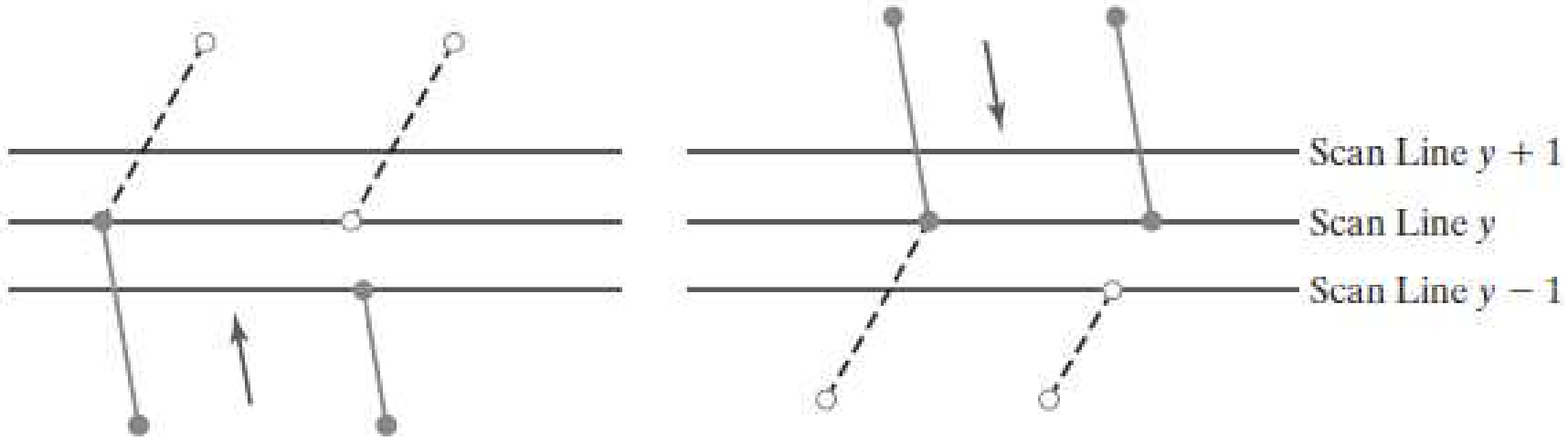




Scan Line Polygon Fill

- To resolve, shorten some polygon edges to split those vertices that should be counted as one intersection
- Process non-horizontal edges around the polygon boundary in the order specified, clockwise or counter-clockwise
- As we process each edge, check to determine whether that edge and the next non-horizontal edge have either monotonically increasing or decreasing endpoint y values
- If so, the lower edge can be shortened to ensure that only one intersection point is generated for the scan line going through the common vertex joining the two edges

Scan Line Polygon Fill



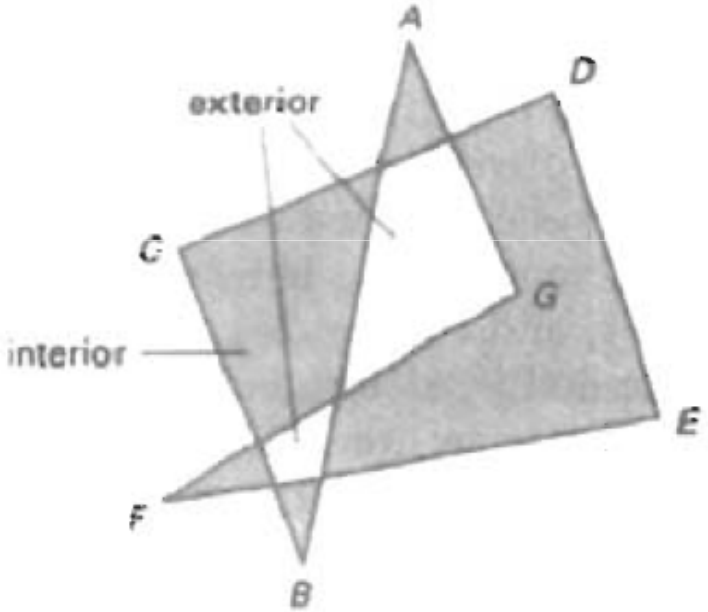
Scan-conversion take advantage of various coherence properties of a scene that is to be displayed

Coherence is simply that the properties of one part of a scene are related in some way to other parts of the scene so that the relationship can be used to reduce processing

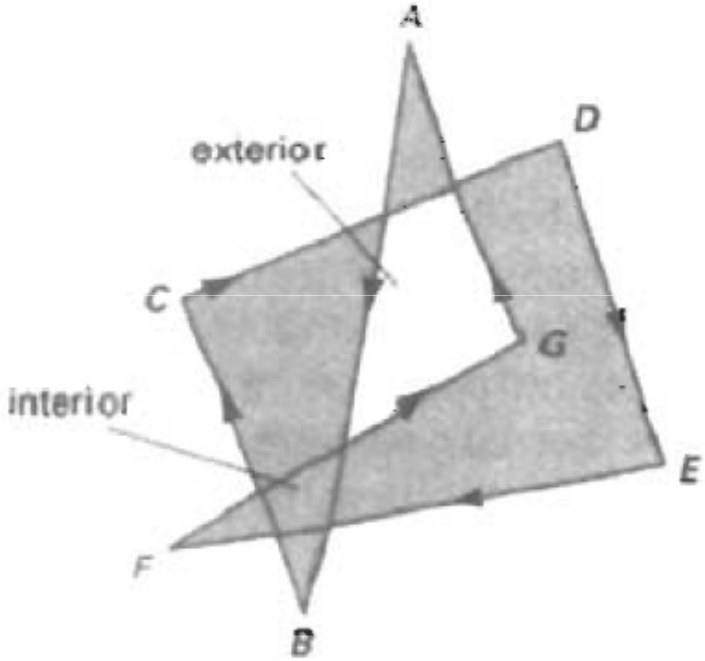
Inside Outside Tests

- Area-filling algorithms and other graphics processes often need to identify interior regions of objects
- Odd Even rule:
  - By conceptually drawing a line from any position P to a distant point outside the coordinate extents of the object and counting the number of edge crossings along the line.
  - If the number of polygon edges crossed by this line is odd, then P is an interior point. Otherwise, P is an exterior point
- Nonzero winding number:
  - Count the number of times the polygon edges wind around a particular point in the counter-clockwise direction - called the winding number
  - Interior points of a two-dimensional object are defined to be those that have a nonzero value for the winding number

Inside Outside Tests



Odd Even rule



Nonzero Winding Number Rule

Boundary Fill Algorithm

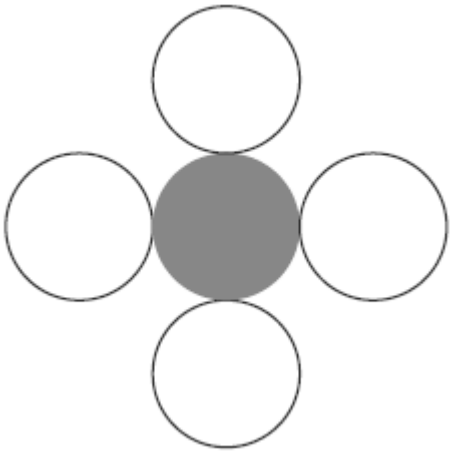
- Scan line algorithm for regions with curved boundaries require more work because intersection calculations involve non-linear boundaries
- Another approach to area filling is to start at a point inside a region and paint the interior outward toward the boundary
- If the boundary is specified in a single colour, the fill algorithm proceeds outward pixel by pixel until the boundary colour is encountered

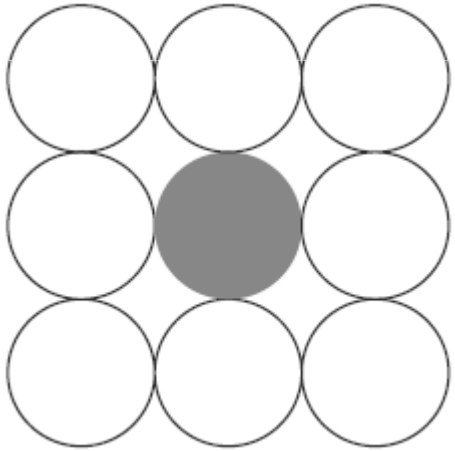
Boundary Fill Algorithm

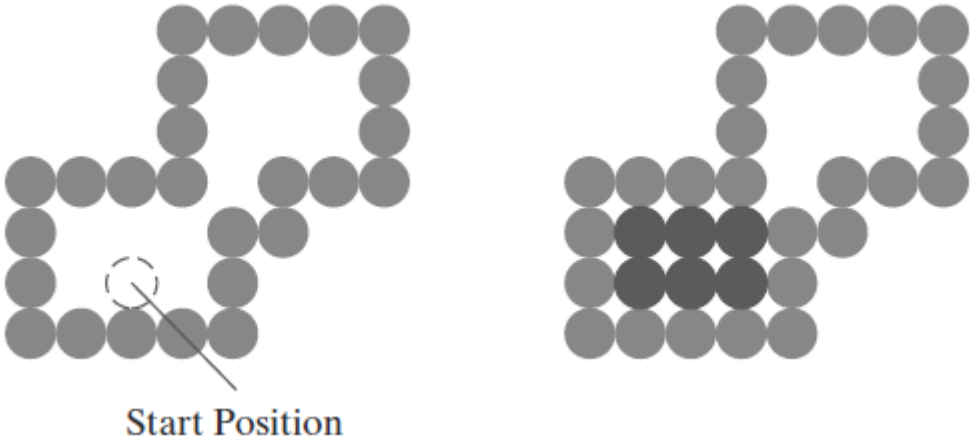
- It accepts as input the coordinates of an interior point (x, y), a fill colour, and a boundary colour
- Starting from (x, y), the procedure tests neighbouring positions to determine whether they are of the boundary colour.
- If not, they are painted with the fill colour, and their neighbours are tested.
- This process continues until all pixels up to the boundary colour for the area have been tested

Boundary Fill Algorithm

- 2 methods for processing neighbouring pixels from a current test position
  - four neighbouring points - are right, left, above, and below – **4 connected**
  - eight neighbouring points – right, left, above, below and four diagonal positions – **8 connected**







Boundary Fill Algorithm

```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
   int interiorColor;

   /* Set current color to fillColor, then perform the following operations. */
   getPixel (x, y, interiorColor);
   if ((interiorColor != borderColor) && (interiorColor != fillColor)) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      boundaryFill4 (x + 1, y , fillColor, borderColor);
      boundaryFill4 (x - 1, y , fillColor, borderColor);
      boundaryFill4 (x , y + 1, fillColor, borderColor);
      boundaryFill4 (x , y - 1, fillColor, borderColor)
   }
}
```

- Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill colour.
  - Encountering a pixel with the fill colour can cause a recursive branch to terminate, leaving other interior pixels unfilled.
  - To avoid this, first change the colour of any interior pixels that are initially set to the fill colour before applying the boundary-fill procedure.

Flood Fill Algorithm

- To fill in an area that is not defined within a single colour boundary
- Paint areas by replacing a specified interior colour instead of searching for a particular boundary colour – **flood-fill algorithm**
- Start from a specified interior point (x, y) and reassign all pixel values that are currently set to a given interior colour with the desired fill colour.
- If the area that we want to paint has more than one interior colour, we can first reassign pixel values so that all interior points have the same colour.
  - Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted

Flood Fill Algorithm

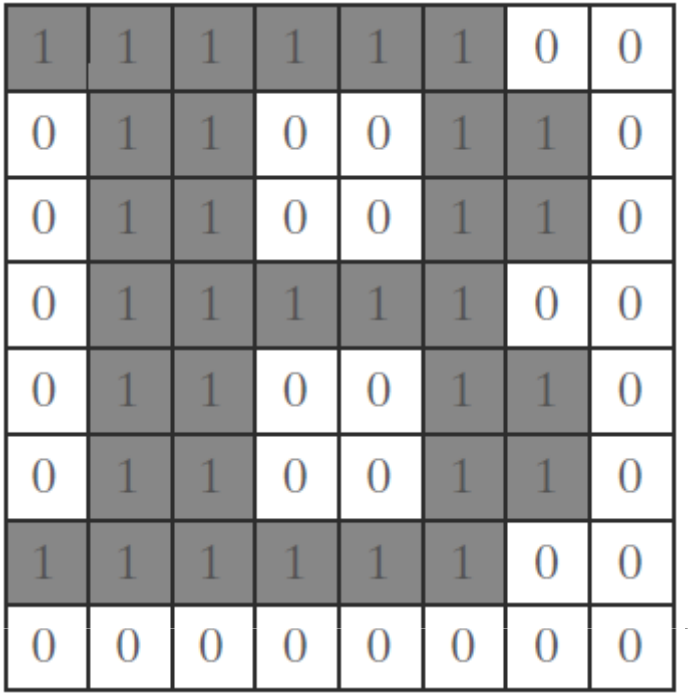
```
void floodFill4 (int x, int y, int fillColor, int interiorColor)
{
   int color;

   /* Set current color to fillColor, then perform the following operations. */
   getPixel (x, y, color);
   if (color = interiorColor) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      floodFill4 (x + 1, y, fillColor, interiorColor);
      floodFill4 (x - 1, y, fillColor, interiorColor);
      floodFill4 (x, y + 1, fillColor, interiorColor);
      floodFill4 (x, y - 1, fillColor, interiorColor)
   }
}
```

Character Generation

- Letters, numbers, and other characters can be displayed in a variety of sizes and styles
- Overall design style for a set (or family) of characters is called a **typeface (or font)**
- **Serif** vs **sans-serif**
  - Serif type has small lines or accents at the ends of the main character strokes (readable), while sans-serif type does not have accents (legible)
- **Monospace** vs **proportional**
  - Monospace font all have the same width, while in proportional font, character width varies

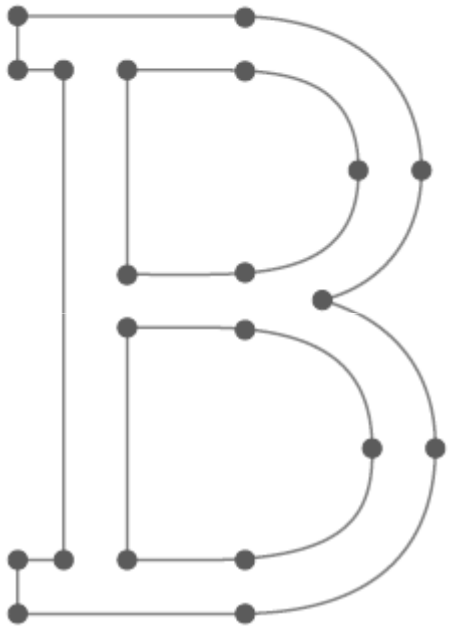
Character Generation



- Two representations are used for storing computer fonts – Bitmapped and outline
- **Bitmap font :**Set up a pattern of binary values on a rectangular grid; also referred to as a **raster font.**
  - Simple to define and display: map the character grids to a frame-buffer position.
  - Require more storage space because each variation (size and format) must be saved in a *font cache*
  - Size and other variations, such as bold and italic, if obtained from one bitmap font set, results are not good

Character Generation

- **Outline font:** Describe character shapes using straight-line and curve sections.
- Require less storage because each variation does not require a distinct font cache
- Boldface, italic, different sizes can be obtained by manipulation curve definition for the character outlines
- It does take more time to process the outline fonts because they must be scan-converted into the frame buffer



Attributes

- Any parameter that affects the way a primitive is to be displayed is referred to as an **attribute parameter**
- **Colour** and **size** – determine the fundamental characteristics
- **Depth** information for three-dimensional viewing and visibility or detectability options – display under special conditions

Attributes

- Self-study
  - Color Models – RGB, Grayscale
  - Point attributes – size
  - Line attributes – width, style