sections on opposite sides of the curve. This is the same as generating pixel positions along the curve boundary, and we can do that with the midpoint method. Then we simply fill in the horizontal pixel spans between the boundary points on opposite sides of the curve. Symmetries between quadrants (and between octants for circles) are used to reduce the boundary calculations.

Similar methods can be used to generate a fill area for a curve section. An elliptical arc, for example, can be filled as in Fig. 3-41. The interior region is bounded by the ellipse section and a straight-line segment that closes the curve by joining the beginning and ending positions of the arc. Symmetries and incremental calculations are exploited whenever possible to reduce computations.

Figure 3-41
Interior fill of an elliptical arc.

Boundary-Fill Algorithm

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 color, the fill algorithm proceeds outward pixel by pixel until the boundary color is encountered. This method, called the **boundary-fill algorithm**, is particularly useful in interactive painting packages, where interior points are easily selected. Using a graphics tablet or other interactive device, an artist or designer can sketch a figure outline, select a fill color or pattern from a color menu, and pick an interior point. The system then paints the figure interior. To display a solid color region (with no border), the designer can choose the fill color to be the same as the boundary color.

A boundary-fill procedure accepts as input the coordinates of an interior point (x, y), a fill color, and a boundary color. Starting from (x, y), the procedure tests neighboring positions to determine whether they are of the boundary color. If not, they are painted with the fill color, and their neighbors are tested. This process continues until all pixels up to the boundary color for the area have been tested. Both inner and outer boundaries can be set up to specify an area, and some examples of defining regions for boundary fill are shown in Fig. 3-42.

Figure 3-43 shows two methods for proceeding to neighboring pixels from the current test position. In Fig. 3-43(a), four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called **4-connected**. The second method, shown in Fig. 3-43(b), is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels. Fill methods using this approach are called **8-connected**. An 8-connected boundary-fill algorithm would correctly fill the interior of the area defined in Fig. 3-44, but a 4-connected boundary-fill algorithm produces the partial fill shown.

Figure 3-42
Example color boundaries for a boundary-fill procedure.

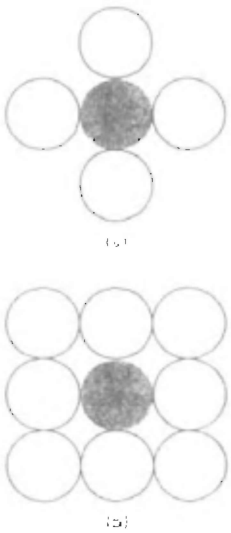

Figure 3-43
Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Open circles represent pixels to be tested from the current test position, shown as a solid color

The following procedure illustrates a recursive method for filling a 4-connected area with an intensity specified in parameter `fill` up to a boundary color specified with parameter `boundary`. We can extend this procedure to fill an 8-connected region by including four additional statements to test diagonal positions, such as $(x + 1, y + 1)$.

```
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);
  }
}
```

Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill color. This occurs because the algorithm checks next pixels both for boundary color and for fill color. Encountering a pixel with the fill color can cause a recursive branch to terminate, leaving other interior pixels unfilled. To avoid this, we can first change the color of any interior pixels that are initially set to the fill color before applying the boundary-fill procedure.

Also, since this procedure requires considerable stacking of neighboring points, more efficient methods are generally employed. These methods fill horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighboring points. Then we need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position. Starting from the initial interior point with this method, we first fill in the contiguous span of pixels on this starting scan line. Then we locate and stack starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions

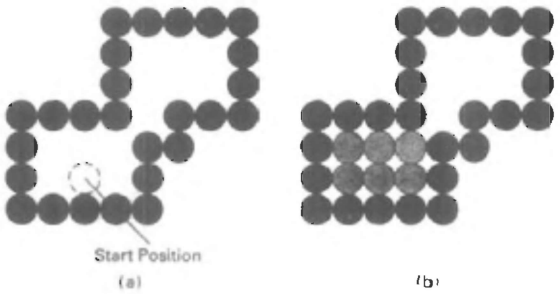

Figure 3-44
The area defined within the color boundary (a) is only partially filled in (b) using a 4-connected boundary-fill algorithm.

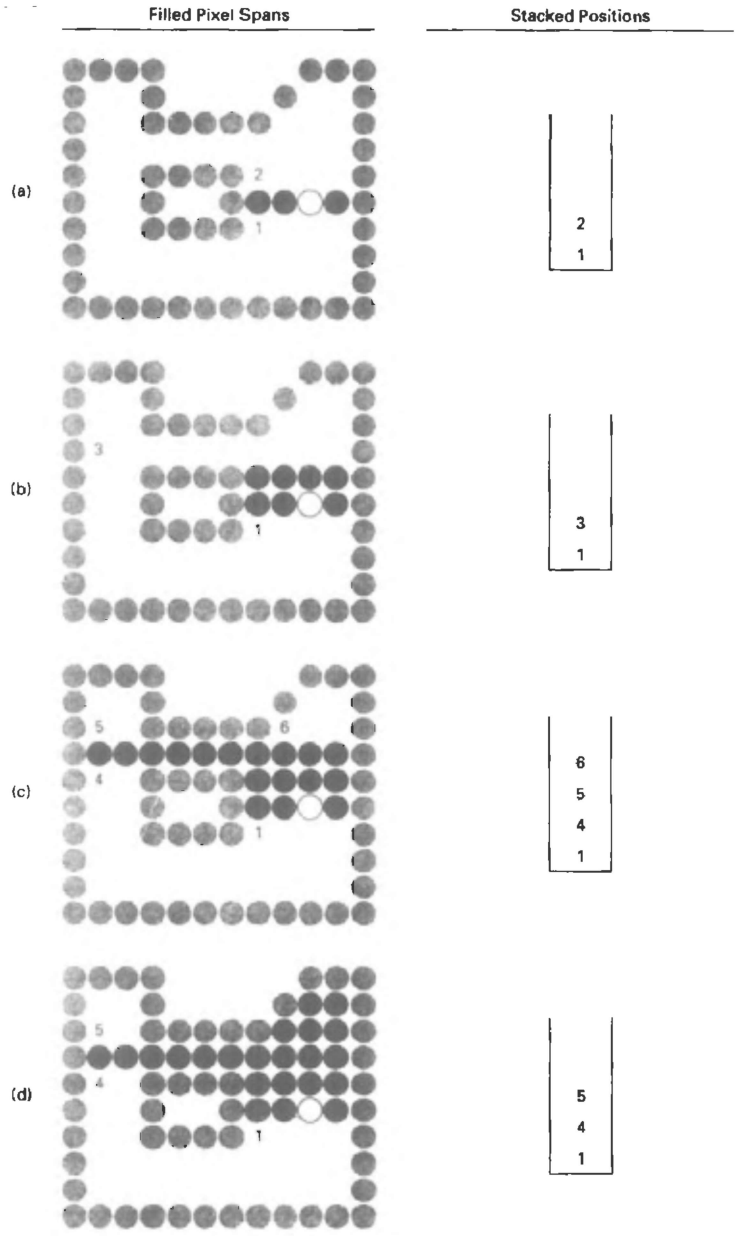

Figure 3-45
Boundary fill across pixel spans for a 4-connected area. (a) The filled initial pixel span, showing the position of the initial point (open circle) and the stacked positions for pixel spans on adjacent scan lines. (b) Filled pixel span on the first scan line above the initial scan line and the current contents of the stack. (c) Filled pixel spans on the first two scan lines above the initial scan line and the current contents of the stack. (d) Completed pixel spans for the upper-right portion of the defined region and the remaining stacked positions to be processed.

bounded by pixels displayed in the area border color. At each subsequent step, we unstack the next start position and repeat the process.

An example of how pixel spans could be filled using this approach is illustrated for the 4-connected fill region in Fig. 3-45. In this example, we first process scan lines successively from the start line to the top boundary. After all upper scan lines are processed, we fill in the pixel spans on the remaining scan lines in order down to the bottom boundary. The leftmost pixel position for each horizontal span is located and stacked, in left to right order across successive scan lines, as shown in Fig. 3-45. In (a) of this figure, the initial span has been filled, and starting positions 1 and 2 for spans on the next scan lines (below and above) are stacked. In Fig. 3-45(b), position 2 has been unstacked and processed to produce the filled span shown, and the starting pixel (position 3) for the single span on the next scan line has been stacked. After position 3 is processed, the filled spans and stacked positions are as shown in Fig. 3-45(c). And Fig. 3-45(d) shows the filled pixels after processing all spans in the upper right of the specified area. Position 5 is next processed, and spans are filled in the upper left of the region; then position 4 is picked up to continue the processing for the lower scan lines.

Figure 3-46
An area defined within multiple color boundaries.

Flood-Fill Algorithm

Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. Figure 3-46 shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a **flood-fill algorithm**. We start from a specified interior point (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color. If the area we want to paint has more than one interior color, we can first reassign pixel values so that all interior points have the same color. Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted. The following procedure flood fills a 4-connected region recursively, starting from the input position.

```
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);
  }
}
```

We can modify procedure `floodFill4` to reduce the storage requirements of the stack by filling horizontal pixel spans, as discussed for the boundary-fill algorithm. In this approach, we stack only the beginning positions for those pixel spans having the value `oldColor` . The steps in this modified flood-fill algorithm are similar to those illustrated in Fig. 3-45 for a boundary fill. Starting at the first position of each span, the pixel values are replaced until a value other than `oldColor` is encountered.