Introduction to Computer Graphics

(Lecture No 09)

Filled-Area Primitives-II

9.1 Boundary fill

Another important class of area-filling algorithms starts at a point known to be inside a figure and starts filling in the figure outward from the point. Using these algorithms a graphic artist may sketch the outline of a figure and then select a color or pattern with which to fill it. The actual filling process begins when a point inside the figure is selected. These routines are like the *paint-scan function* seen in common interactive paint packages.

The first such method that we will discuss is called the *boundary-fill algorithm*. The boundary-fill method requires the coordinates of a starting point, a fill color, and a boundary color as arguments.

Boundary fill algorithm:

The Boundary fill algorithm performs the following steps:

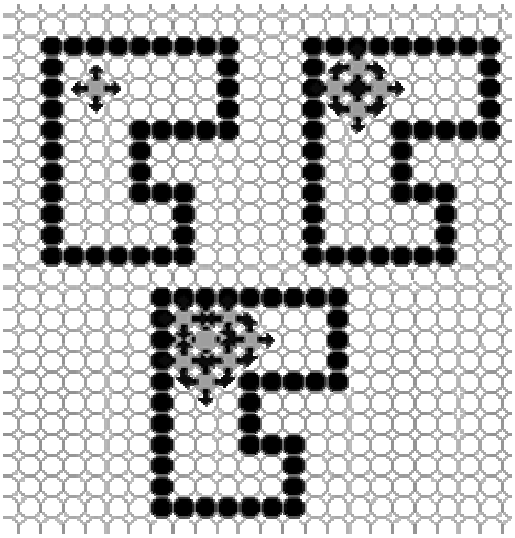
- Check the pixel for boundary color
- Check the pixel for fill color
- Set the pixel in fill color
- Run the process for neighbors

The pseudo code for Boundary fill algorithm can be written as:

```
boundaryFill (x,  y, fillColor , boundaryColor)
        if ((x < 0) || (x >= width))
                                        return
        if ((y < 0) || (y >= height))
                                        return
        current = GetPixel(x, y)
        if ((current != boundaryColor) && (current != fillColor))
        setPixel(fillColor, x, y)
        boundaryFill (x+1, y, fillColor, boundaryColor)
        boundaryFill (x, y+1, fillColor, boundaryColor)
        boundaryFill (x-1, y, fillColor, boundaryColor)
        boundaryFill (x, y-1, fillColor, boundaryColor)
```

Note that this is a **recursive routine**. Each invocation of *boundaryFill ()* may call itself four more times.

The logic of this routine is very simple. If we are not either on a boundary or already filled we first fill our point, and then tell our neighbors to fill themselves.



Process of Boundary Fill Algorithm

By the way, sometimes the boundary fill algorithm doesn't work. Can you think of such a case?

9.2 Flood Fill

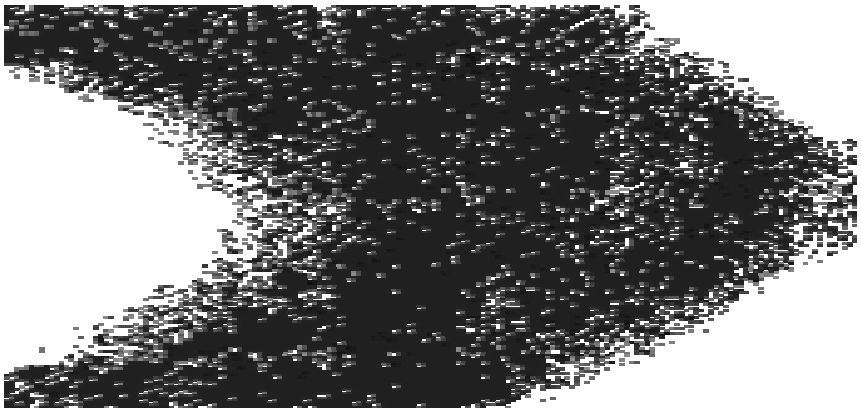
Sometimes we need an area fill algorithm that replaces all *connected* pixels of a selected color with a fill color.

The ***flood-fill algorithm*** does exactly that.

Flood-fill algorithm

An area fill algorithm that replaces all *connected* pixels of a selected color with a fill color.

Before Applying Flood-fill algorithm (Light color)



After Applying Flood-fill algorithm (Dark color)

Flood-fill algorithm in action

The pseudo code for Flood fill algorithm can be written as:

```
public void floodFill(x, y, fillColor, oldColor)

    if ((x < 0) || (x >= width))
                                    return
    if ((y < 0) || (y >= height))
                                    return
    if ( getPixel (x, y) == oldColor)

        setPixel (fillColor, x, y)
```

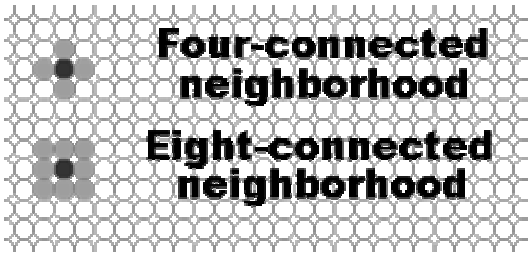
```
    floodFill (x+1, y, fillColor, oldColor)
    floodFill (x, y+1, fillColor, oldColor)
    floodFill (x-1, y, fillColor, oldColor)
    floodFill (x, y-1, fillColor, oldColor)
```

It's a little awkward to kick off a flood fill algorithm because it requires that the old color must be read before it is invoked. The following implementation overcomes this limitation, and it is also somewhat faster, a little bit longer. The additional speed comes from only pushing three directions onto the stack each time instead of four.

```
fillFast (x, y, fillColor)
   if ((x < 0) || (x >=width)) return
   if ((y < 0) || (y >=height)) return
   int oldColor = getPixel (x, y)
   if ( oldColor == fill ) return
   setPixel (fillColor, x, y)
   fillEast (x+1, y, fillColor, oldColor)
   fillSouth (x, y+1, fillColor, oldColor)
   fillWest (x-1, y, fillColor, oldColor)
   fillNorth (x, y-1, fillColor, oldColor)

fillEast (x, y, fillColor, oldColor)
   if (x >= width) return
   if ( getPixel(x, y) == oldColor)
      setPixel( fillColor, x, y)
      fillEast (x+1, y, fillColor, oldColor)
      fillSouth (x, y+1, fillColor, oldColor)
      fillNorth (x, y-1, fillColor, oldColor)

fillSouth(x, y, fillColor, oldColor)
   if (y >=height) return
   if (getPixel (x, y) == oldColor)
      setPixel (fillColor, x, y)
      fillEast (x+1, y, fillColor, oldColor)
      fillSouth (x, y+1, fillColor, oldColor)
      fillWest (x-1, y, fillColor, oldColor)

fillWest(x, y, fillColor, oldColor)
{
   if (x < 0) return
   if (getPixel (x, y) == oldColor)
      setPixel (fillColor, x, y)
```

```
        fillSouth (x, y+1, fillColor, oldColor)
        fillWest (x-1, y, fillColor, oldColor)
        fillNorth (x, y-1, fillColor, oldColor)

fillNorth (x, y, fill, old)
    if (y < 0) return
    if (getPixel (x, y) == oldColor)
        setPixel (fill, x, y)
        fillEast (x+1, y, fillColor, oldColor)
        fillWest (x-1, y, fillColor, oldColor)
        fillNorth (x, y-1, fillColor, oldColor)
```

A final consideration when writing an area-fill algorithm is the size and connectivity of the neighborhood around a given pixel.



The eight-connected neighborhood is able to get into nooks and crannies that an algorithm based on a four-connected neighborhood cannot.

Here's the code for an ***eight-connected flood fill***.

```
floodFill8 (x, y, fill, old)
    if ((x < 0) || (x >=width)) return
    if ((y < 0) || (y >=height)) return
    if (getPixel (x, y) == oldColor)
        setPixel (fill, x, y);
        floodFill8 (x+1, y, fillColor, oldColor)
        floodFill8 (x, y+1, fillColor, oldColor)
        floodFill8 (x-1, y, fillColor, oldColor)
        floodFill8 (x, y-1, fillColor, oldColor)
        floodFill8 (x+1, y+1, fillColor, oldColor)
        floodFill8 (x-1, y+1, fillColor, oldColor)
        floodFill8 (x-1, y-1, fillColor, oldColor)
        floodFill8 (x+1, y-1, fillColor, oldColor)
```