Area-Filling Algorithms

- Parity and Wrapping-Number Fills
- Boundary and Flood Fills
- YATA, YATA
- Project #2 Update

Hearn & Baker, pp 125-130 (Study)

What's the Point of Filling Algorithms?

- We've assumed that areas are filled during scan conversion
- Fill region *after* its boundary is determined.
- Irregular boundaries
-
- Freehand drawings
- Deferred fills for speed
- Allows us to recolor primitives
- Often used to select regions
-
- Unaware of any other primitives previously drawn.

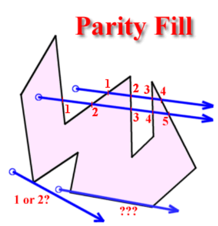
Parity Fill

Problem:

For each pixel determine if it is inside or outside of a given polygon.

Approach:

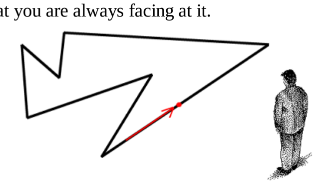
- from the point being tested cast a ray in an arbitary direction
- if the number of crossings is odd then the point is inside
- if the number of crossings is even then the point is outside



- Very fragile algorithm
- Ray crosses a vertex
- Ray is coincident with an edge
- Commonly used in ECAD
- Suitable for H/W

Winding Number

Imagine yourself watching a point traverse the boundary of the polygon in a counter-clockwise direction and pivoting so that you are always facing at it.



Your *winding number* is the number of full revolutions that you complete.

If you winding number is 0 then you are outside of the polygon, otherwise you are inside.

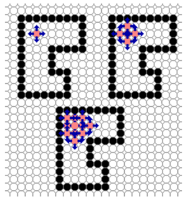
Boundary Fills

Boundary fills start from a point known to be inside of a region
and fill the region until a boundry is found.

A simple recursive algorithm can be used:

```
public void boundaryFill(int x, int y, int fill, int boundary) {
    if ((x < 0) || (x >= raster.width)) return;
    if ((y < 0)|| (y >= raster.height)) return;
    int current = raster.getPixel(x, y);


    if ((current != boundary) & (current != fill)) {
        raster.setPixel(fill, x, y);
        boundaryFill(x+1, y, fill, boundary);
        boundaryFill(x, y+1, fill, boundary);
        boundaryFill(x-1, y, fill, boundary);
        boundaryFill(x, y-1, fill, boundary);
    }
}
```



A second important class of area-filling algorithms start at a point know to be inside a figure and start filling in the figure outward from the point. Using these algorithms a graphics artist may sketch the outline of a figure and then select a color or patten from a menu with which to fill it. The actual filling process begins when a point inside of the figure is slected. These routines are like the *paint-can 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 coordinate of a starting point, a fill color, and a background color as arguments.

```
public void boundaryFill(int x, int y, int fill, int boundary) {
    if ((x < 0)|| (x >= raster.width)) return;
    if ((y < 0)|| (y >= raster.height)) return;
    int current = raster.getPixel(x, y);
    if ((current != boundary) & (current != fill)) {
        raster.setPixel(fill, x, y);
        boundaryFill(x+1, y, fill, boundary);
```


Let's Watch it in Action

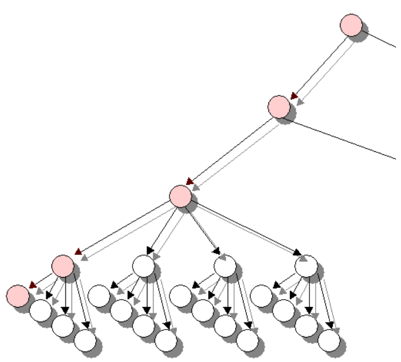
First, you should guess how you expect the algorithm to behave. Then select a point on the figure's interior. Did the algorithm act as you expected?

Serial Recursion is Depth-First

So the fill algorithm will continue in one direction until a boundary is reached.

It will then change directions momentarily and attempt to continue back in the original direction.

Will parallel execution of the algorithm behave the same way?



At Full Speed

To the right is the same algorithm operating at full speed.

Left-button click inside one of the regions to start the fill process. Click the right button to reset the image to its original state

A Flood-Fill

Sometimes we'd like a area fill algorithm that replaces all *connected* pixels of a selected color with a fill color. The *flood-fill algorithm* does exactly that.

```
public void floodFill(int x, int y, int fill, int old) {
    if ((x < 0) || (x >= raster.width)) return;
    if ((y < 0) || (y >= raster.height)) return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        floodFill(x+1, y, fill, old);
        floodFill(x, y+1, fill, old);
        floodFill(x-1, y, fill, old);
        floodFill(x, y-1, fill, old);
    }
}
```

This algorithm replaces *old* pixels with the *fill* color.

Usually, the initial pixel *(x, y)* establishes the *old* value.

Flood-Fill in Action

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.

Self-Starting Fast Flood-Fill Algorithm

The follow implementation self-starts, and is also somewhat faster.

```
public void fillFast(int x, int y, int fill)
{
    if ((x < 0) || (x >= raster.width)) return;
    if ((y < 0) || (y >= raster.height)) return;
    int old = raster.getPixel(x, y);
    if (old == fill) return;
    raster.setPixel(fill, x, y);
    fillEast(x+1, y, fill, old);
    fillSouth(x, y+1, fill, old);
    fillWest(x-1, y, fill, old);
    fillNorth(x, y-1, fill, old);
}
```


Fill East

```
private void fillEast(int x, int y, int fill, int old) {
    if (x >= raster.width) return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        fillEast(x+1, y, fill, old);
        fillSouth(x, y+1, fill, old);
        fillNorth(x, y-1, fill, old);
    }
}
```

Note:

There is only one clipping test, and only three subsequent calls.
Why?
How much faster do you expect this algorithm to be?

Working Algorithm

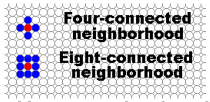
```
private void fillSouth(int x, int y, int fill, int old) {
    if (y >= raster.height) return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        fillEast(x+1, y, fill, old);
        fillSouth(x, y+1, fill, old);
        fillWest(x-1, y, fill, old);
    }
}
```

You can figure out the other routines yourself.

Previous Lecture 8 Slide 13 6.837 Fall '98 Next

4-Way and 8-Way Connectedness

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



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

Previous Lecture 8 Slide 14 6.837 Fall '98 Next

Code for an 8-Way Connected Flood Fill

```
public void floodFill8(int x, int y, int fill, int old) {
    if ((x < 0) || (x >= raster.width)) return;
    if ((y < 0) || (y >= raster.height)) return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        floodFill8(x+1, y, fill, old);
        floodFill8(x, y+1, fill, old);
        floodFill8(x-1, y, fill, old);
        floodFill8(x, y-1, fill, old);
        floodFill8(x+1, y+1, fill, old);
        floodFill8(x-1, y+1, fill, old);
        floodFill8(x-1, y-1, fill, old);
        floodFill8(x+1, y-1, fill, old);
    }
}
```

Previous Lecture 8 Slide 15 6.837 Fall '98 Next

More 8-Way Connectedness

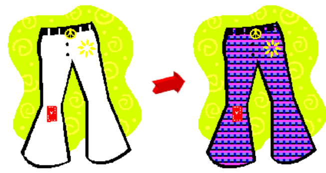
Sometimes 8-way connectedness can be too much.

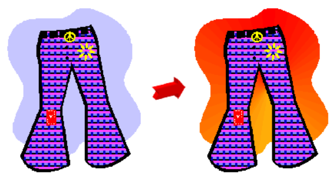
Previous Lecture 8 Slide 16 6.837 Fall '98 Next

Flood-Fill Embellishments

The flood-fill and boundary-fill algorithms can easily be modified for a wide variety of new uses:

1. Patterned fills
2. Approximate fills
3. Gradient fills
4. Region selects
5. Triangle Rendering!





YATA (Yet Another Triangle Renderer)

A simple modification of the boundary-fill algorithm produces a triangle renderer.

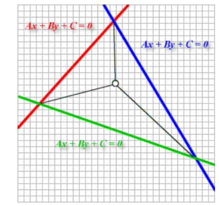
We need two things:

1. A boundary (terminating condition)

   Edge Equations

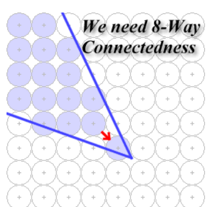
2. An interior point (initial condition)

$$\overline{p}_{mid} = \tfrac{1}{3}\left(\overline{p}_1 + \overline{p}_2 + \overline{p}_3\right)$$



More Subtle Issues

Connectedness



We can use our plane equations to generate the fill color

$$r(x, y) = A_r x + B_r y + C_r$$
$$g(x, y) = A_g x + B_g y + C_g$$
$$b(x, y) = A_b x + B_b y + C_b$$

Implementation Details

```
    public void Draw(Raster raster) {
    this.raster = raster;
    if (!triangleSetup( )) return;

    /*
        find the triangle's centroid
    */
    int x = (int) ((v[0].x + v[1].x + v[2].x)/3 + 0.5);
    int y = (int) ((v[0].y + v[1].y + v[2].y)/3 + 0.5);

    /*
        initialize parameters
    */
    int param[] = new int[7];
    param[0] = edge[0].evaluate(x, y);
    param[1] = edge[1].evaluate(x, y);
    param[2] = edge[2].evaluate(x, y);
    param[3] = alpha.evaluate(x, y) + (1 << (EdgeEqn.FRACBITS - 1));
    param[4] = red.evaluate(x, y) + (1 << (EdgeEqn.FRACBITS - 1));
    param[5] = green.evaluate(x, y) + (1 << (EdgeEqn.FRACBITS - 1));
    param[6] = blue.evaluate(x, y) + (1 << (EdgeEqn.FRACBITS - 1));
    fillTri(x, y, param);
}
```

Modified Boundary Fill

```
    private void fillTri(int x, int y, int param[]) {
        /*
            pixel inside bounding box?
        */
        if ((x > xMax) || (x < xMin)) return;
        if ((y > yMax) || (y < yMin)) return;

        /*
            pixel inside triangle?
        */
        if ((param[0]|param[1]|param[2]) < 0) return;

        /*
            compute fill color
        */
        int pixa = (param[3] >> EdgeEqn.FRACBITS);
        int pixr = (param[4] >> EdgeEqn.FRACBITS);
        int pixg = (param[5] >> EdgeEqn.FRACBITS);
        int pixb = (param[6] >> EdgeEqn.FRACBITS);
        pixa = ((pixa & ~255) == 0) ? pixa << 24 : ((pixa < 0) ? 0 : 255
        param[0] -= edge[0].A;
        param[1] -= edge[1].A;
        param[2] -= edge[2].A;
        param[3] -= alpha.A;
        param[4] -= red.A;
        param[5] -= green.A;
        param[6] -= blue.A;
        fillTri(x, y+1, param);

        param[0] -= edge[0].A;
        param[1] -= edge[1].A;
```



```
        param[2] -= edge[2].A;
        param[3] -= alpha.A;
        param[4] -= red.A;
        param[5] -= green.A;
        param[6] -= blue.A;
        fillTri(x-1, y+1, param);

        param[0] -= edge[0].B;
        param[1] -= edge[1].B;
        param[2] -= edge[2].B;
        param[3] -= alpha.B;
        param[4] -= red.B;
        param[5] -= green.B;
        param[6] -= blue.B;
        fillTri(x-1, y, param);

        param[0] -= edge[0].B;
        param[1] -= edge[1].B;
        param[2] -= edge[2].B;
        param[3] -= alpha.B;
        param[4] -= red.B;
        param[5] -= green.B;
        param[6] -= blue.B;
        fillTri(x-1, y-1, param);

        param[0] += edge[0].A;
        param[1] += edge[1].A;
        param[2] += edge[2].A;
        param[3] += alpha.A;
        param[4] += red.A;
        param[5] += green.A;
        param[6] += blue.A;
        fillTri(x, y-1, param);
```



```
        param[0] += edge[0].A;
        param[1] += edge[1].A;
        param[2] += edge[2].A;
        param[3] += alpha.A;
        param[4] += red.A;
        param[5] += green.A;
        param[6] += blue.A;
        fillTri(x+1, y-1, param);

        /*
            fix it up back to the way we found it (Why?)
        */
        param[0] += edge[0].B - edge[0].A;
        param[1] += edge[1].B - edge[1].A;
        param[2] += edge[2].B - edge[2].A;
        param[3] += alpha.B - alpha.A;
        param[4] += red.B - red.A;
        param[5] += green.B - green.A;
        param[6] += blue.B - blue.A;
    }
}
```


Test Case

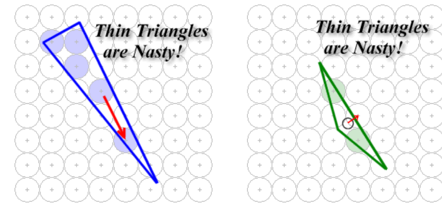
Above is our simple test case from last lecture. Using the TriTest applet from your Project #2 assignment.

Harder Cases

These two triangle databases are a little more diffficult to get right.

Discuss Problems

Triangles interiors are not necessarily 8-way connected!



The rounded coordinates of a triangle's centroid are not necessarily inside of the triangle!

Solutions?

- Redefine what it means to be inside of a triangle
- Different routines for nasty little triangles

Project #2: One Last Visit

I've never seen a purple cow
I never hoped to draw one
But I can tell you anyhow
I'd rather draw than gnaw one

with apologies to Gelett Burgess
(MIT class of 1887)

Next Time