Computer Graphics

Flood Fill
Homework 1 Results

Brianna

Chris
Homework 1 Results

Emi

Elena

Eve
Homework 1 Results

Nana Aba

Tillie
Homework 1 Results
function drawLine(x1, y1, x2, y2, color) {
    graphics.fillStyle = color;
    var dx = x2-x1;
    var dy = y2-y1;
    if (Math.abs(dx)>Math.abs(dy)) {
        if (x2<x1) {
            drawLine(x2, y2, x1, y1, color);
        } else {
            var m = dy/dx; // slope
            var b = (x2*y1-x1*y2)/dx; // y intercept
            for (x = Math.floor(x1); x <= Math.floor(x2); x++) {
                y = m*x+b;
                fillPixel(x,y);
            }
        }
    } else {
        if (y2<y1) {
            drawLine(x2, y2, x1, y1, color);
        } else {
            var w = dx/dy; // inverse slope
            var c = (y2*x1-y1*x2)/dy; // x intercept
            for (y = Math.floor(y1); y <= Math.floor(y2); y++) {
                x = w*y+c;
                fillPixel(x,y);
            }
        }
    }
}
Incremental Algorithm

```javascript
function drawLine(x1, y1, x2, y2, color) {
    graphics.fillStyle = color;
    var dx = x2-x1;
    var dy = y2-y1;
    if (Math.abs(dx)>Math.abs(dy)) {
        if (x2<x1) {
            drawLine(x2, y2, x1, y1, color);
        } else {
            // precompute as much as possible:
            var x = Math.floor(x1);
            var y = Math.floor(y1);
            var dx_step = Math.sign(dx);
            var d = dy*(x+dx_step)-dx*(y+1.5)+y1*x2-x2*x1;
            var dd_step = dy*dx_step-dx;
            while (x < Math.floor(x2)) {
                fillPixel(x,y);
                if (d*dx_step < 0) {
                    x += dx_step;
                    d += dd_step;
                } else {
                    d -= dy;
                }
                x++;
            }
            fillPixel(x,y);
        }
    } else {
        // precompute as much as possible:
        var x = Math.floor(x1);
        var y = Math.floor(y1);
        var dy_step = Math.sign(dy);
        var d = dx*(y+dy_step)-dy*(x+1.5)+x1*y2-x2*y1;
        var dd_step = dx*dy_step-dy;
        while (y < Math.floor(y2)) {
            fillPixel(x,y);
            if (d*dy_step < 0) {
                y += dy_step;
                d += dd_step;
            } else {
                d -= dy;
            }
            y++;
        }
        fillPixel(x,y);
    }
}
```
Q. I'm not clear on the NSEW thing. In the example, how come, after you colored a pixel, why did you go over some pixels with lines and stuff and not others.

A. Each pixel (or recursive call) must test all four directions before returning. Sometimes the call to test a direction returns immediately, and sometimes it makes further recursive calls. Either way, it eventually returns and the next direction in the sequence is tested. The diagram represents which directions have been tested using the colored dotted lines to keep track of each pixel’s status.
Your Questions

Q. How does the recursive function keep track of what directions haven't been explored?

A. *Each time the recursive function is called it has its own set of local variables and instruction counter to keep track of how much it has completed. In the diagram this is represented by the colored dotted lines.*

Q. Is there a method that can skip going through all of the pixel's neighbors after they've all been filled?

A. *The only way to know is to check. Fortunately that is fast if they are already filled.*
Q. Could you give us an example of what multiple recursive functions would look like when coding?

A. I’m not sure what you mean. Are you referring to co-recursion? Or do you just want more examples?

Q. How do you go about making the illustrations/graphics in the videos?

A. I use Powerpoint’s animation features. It takes a while, but I’ve had practice. 😊
Q. Is recursion necessary for mask fill? How do graphics programs deal with the possibility of too many recursive calls causing a stack overflow? Is recursion the most efficient way to implement a fill operation or are there other more efficient ways?

A. There are other ways to do a mask fill.

- **Sweep fill** is a recursive algorithm that makes fewer calls by filling a whole line per call.
- The recursive fill can be made iterative with use of a stack data structure.
- Another non-recursive option is connected components.

None of these options is significantly more efficient than recursive fill.
Q. Will we be looking at coloring in diagonal directions in this class?
A. Yes! The handout for today has an example.

Q. In the slides, I saw the algorithms called "Recursive Fill" and "Sweep Fill" and in the video I saw algorithms called "Recursive Fill" and "Scanline Fill." How is "Sweep Fill" related to the ones in the video? It sounded like "Scanline Fill," but it mixed iteration and recursion, and I think that "Scanline Fill" used only iteration.

A. Sweep Fill is a variation on Recursive Fill that fills an entire row of pixels per recursive call. I decided not to teach it this year, to make room for Scanline Fill which is more capable since it can fill complex polygons. The slides still retained a reference to it – my mistake!
Q. What does recursive call depth mean in question 1 and how do you determine it?

A. Think of it like a counter. Each time you make a recursive call, the recursive depth goes up by one. When you return form a call, it goes down by one. So the maximum depth is the highest number of calls that have been made that have not yet returned. If you follow the trail of pixels traversed by the algorithm, and laid a string behind you to keep track of where you had been, the call depth is the length of the string.
Q. I'm still confused about the scanline algorithm. Can you explain what each line of the pseudocode does line by line? I'm also confused about the slide after the slide with the scanline algorithm ("scanline details").

Q. I'd love to see a simulation of the scanline fill function like you had for the recursive function since it's hard to picture how it works without one.

Q. May you explain the Scanline Algorithm in a more detailed way? I cannot really understand how it works.

Q. I'm unclear on what C is in the scanline fill algorithm. Could you walk us through the algorithm in class? Could you also walk us through the equations on the slide "Scanline Details"?

Q. Can you please explain the scan line algorithm in more detail/different words? I think I need another run through! I also don't understand the second question, I didn't realize there were multiple fill lines and why does that translate to rows? Is it just because they're horizontal?
Scanline Algorithm

Input: Sequential vertices \((x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)\), where \((x_0, y_0) = (x_n, y_n)\)
Goal: Fill complex polygon delineated by these vertices

Loop for \(y\) from \(\min(y_0, \ldots y_n)\) to \(\max(y_0, \ldots y_n)\)

\[
C = [] \quad // \text{the set of intersection coordinates for this row}
\]

Loop for \(i\) from 0 to \(n-1\)

If edge \((x_i, y_i)\) to \((x_{i+1}, y_{i+1})\) intersects scanline \(y\) at \((x_c, y_c)\):

\[
C.append((x_c, y_c)) \quad // \text{add intersection point to } C
\]

Sort \(C\) by the x coordinates

Loop for \(j\) from 0 to \(\text{Size}(C)\) step by 2

Loop for \(x\) from \(C[j]\) to \(C[j+1]\)

\[
\text{Fill}(x, y)
\]
Here’s how to find the intersection points between edges & scanlines:

- Edge intersects if one endpoint is below \( y \), the other on or above \( y \)
  \[
  ((y_i < y) \land (y_{i+1} \geq y)) \lor ((y_i \geq y) \land (y_{i+1} < y))
  \]

- Use line equation of the edge to find intersection point @ scanline \( y \):

\[
x = \frac{x_i - x_{i+1}}{y_i - y_{i+1}} y + \frac{x_{i+1}y_i - x_iy_{i+1}}{y_i - y_{i+1}}
\]
Scanline Fill Example

Polygon vertices: (3,2), (9,8), (9,5), (1,6)

Loop for y from 2 to 8: 2.5 to 7.5:
- y = 2.5: 2 intersections, fill @3
- y = 3.5: 2 intersections, fill 2-4
- y = 4.5: 2 intersections, fill 2-5
- y = 5.5: 4 intersections, fill 1-4, 6-8
- y = 6.5: 2 intersections, fill 7-8
- y = 7.5: 2 intersections, fill @8
Q. Can a scanline fill work with any equation? Like one to make curves instead of lines?

A. There might be situations where you could do something similar. But straight lines are the typical case by far.
Other Questions?
Handout 2

Practice on:
• Thinking about recursion
• Recursive fill
• Sweep fill
Handout 2

Fill with limited recursive calls:

North only

North and East only

North, East, and South only
Could the stack overflow problem be addressed by running four separate recursion operations, one for each quadrant? What problems, if any, do you see with this approach?
Handout 2