2D Graphics Techniques

COSC342

Lecture 4
9 Mar 2017

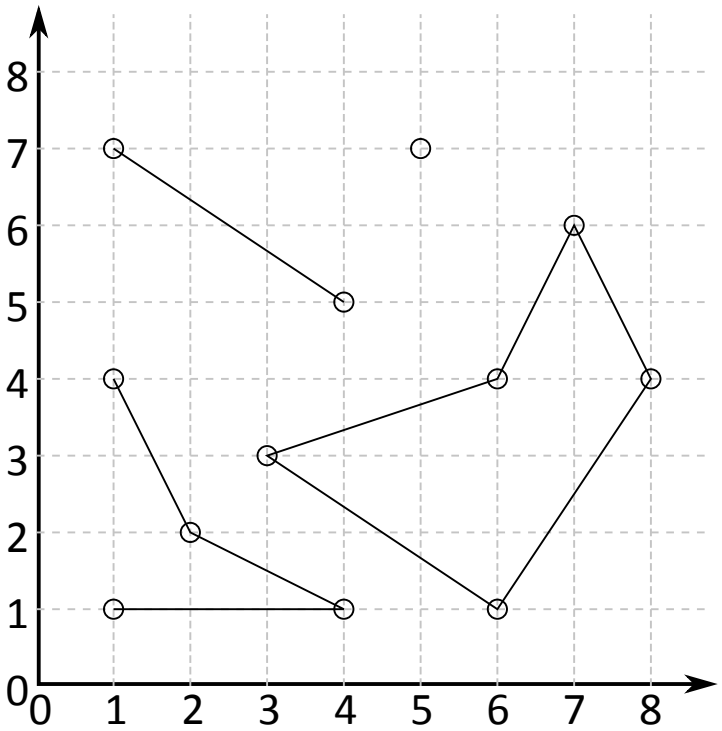
So What's This All About?

- A selection of 2D graphics techniques
- Drawing lines, circles, etc.
- Flood fill
- Intersection calculations

Points, Lines, Polylines, Polygons

- A point is a 2D location, (x, y)
- A line is defined by a pair of points, $(x_0, y_0), (x_1, y_1)$
  - Mathematicians tend to count from 1, but we usually count from 0
- A polyline with k segments is a sequence of $k + 1$ points, $(x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k)$
- A polygon is polyline where $(x_0, y_0) = (x_k, y_k)$
  - Can usually omit the duplicate point

Points, Lines, Polylines, Polygons



- A point, $(5, 7)$
- A line, $(4, 5), (1, 7)$
- A polyline,
  $(1, 4), (2, 2), (4, 1), (1, 1)$
- A polygon,
  $(7, 6), (8, 4), (6, 1), (3, 3), (6, 4)$

- We want to be able to draw lines, and fill polygons

Representing Points etc.

- A basic `Point` representation in C/C++ might be

```
struct Point {
  double x, y;
};
```

- Two possible `Triangle` representations:

```
struct Triangle1 {
  Point p1, p2, p3;
};

struct Triangle2 {
  Point *p1, *p2, *p3;
};
```

- Which is better, and why?

Drawing Lines

- Suppose we want to draw a line between two pixels
- Naïve algorithm:

```
// Line from (x0, y0) to (x1, y1)
double slope = double(y1-y0)/double(x1-x0);
for (int x = x0; x <= x1; ++x) {
  int y = int(y0 + slope*(x-x0));
  paint(x,y);
}
```

- What's wrong with this?

Bresenham's Line Algorithm (1965)

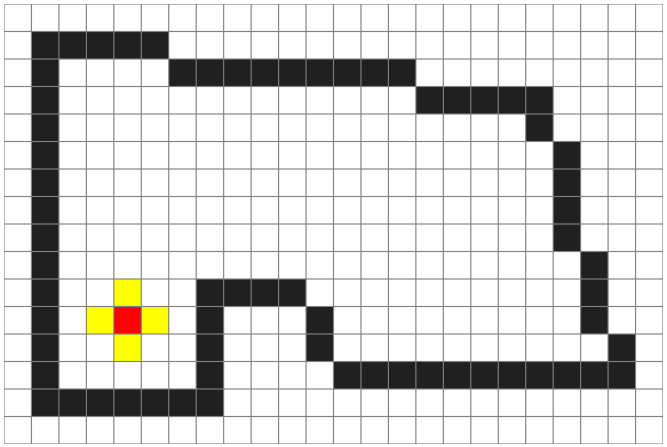
```
// Line from (x0, y0) to (x1, y1)
// This is for the case x0 < x1, |x1-x0| > |y1=y0|
int dx = x0 - x1;
int dy = y0 - y1;
double err = 0;
double derr = abs(double(dy)/double(dx)); // Note, derr < 1
for (int x = x0; x <= x1; ++x) {
  paint(x,y);
  err += derr;
  if (err > 0.5) {
    y += sign(dy);
    err -= 1;
  }
}
```

Bresenham's Line Algorithm

- How it works:
  - We loop over the possible x values
  - Track the error between the pixel locations and the ideal y value
  - If the error $\geq 1/2$ a pixel, increment y and drop the error by 1
- This code is for 'mostly horizontal' lines from left-to-right
  - To go from right-to-left, swap the points
  - For mostly vertical lines loop over y and compute errors in x
- This assumes pixels are on or off, and lines go between integer points
  - Often we want to anti-alias lines, or draw lines between arbitrary points
  - Other algorithms exist for this – Wu's Line Algorithm (1991) is one
  - Basically the error value tells you how to antialias

Filling Polygons

- Suppose we've drawn some lines
- We're given a point and want to fill until we reach the lines
- This is called 'flood fill'

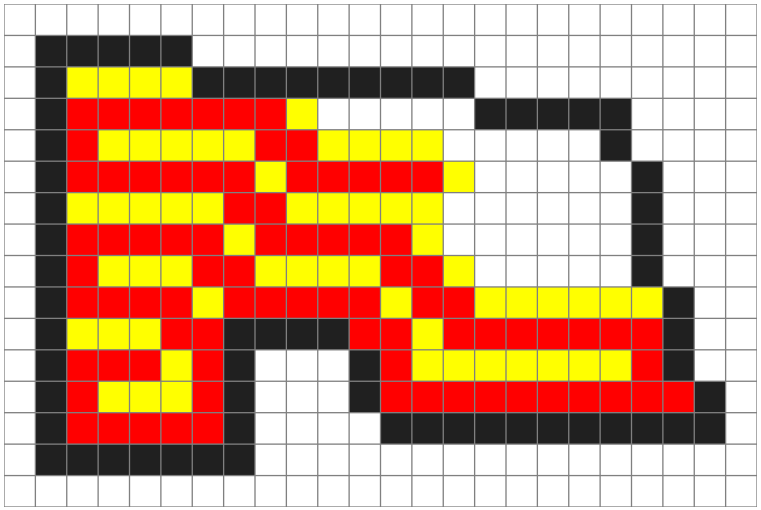


Simple algorithm:

```
floodFill(Point p) {
  paint(p);
  foreach neighbour, q, of p {
    if (!painted(q)) {
      floodFill(q);
    }
  }
}
```

Filling Polygons

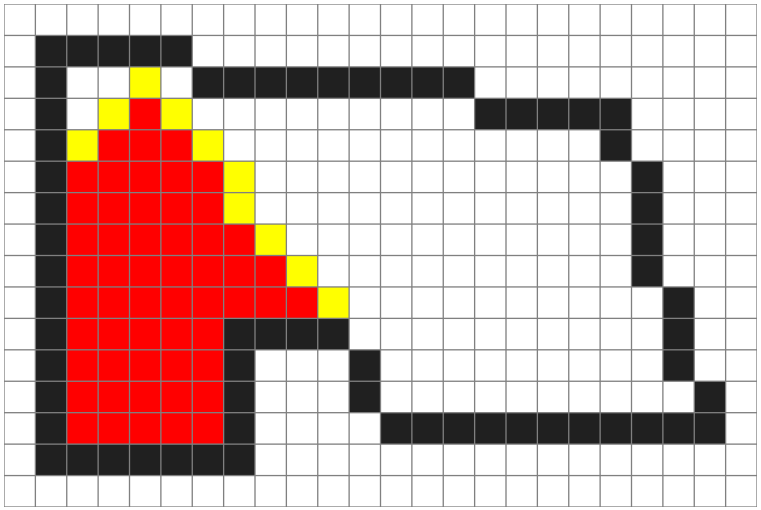
- Recursion is a concise way of representing this algorithm
- But it is a bad idea in languages like C/C++ (Why?)
- We can implement flood fill with a stack



```
floodFill(Point p) {
  stack S;
  paint(p);
  S.push(p);
  while (!S.empty()) {
    q = S.pop();
    foreach neighbour, r, of q {
      if (!painted(r)) {
        paint(r);
        S.push(r);
      }
    }
  }
}
```

Filling Polygons

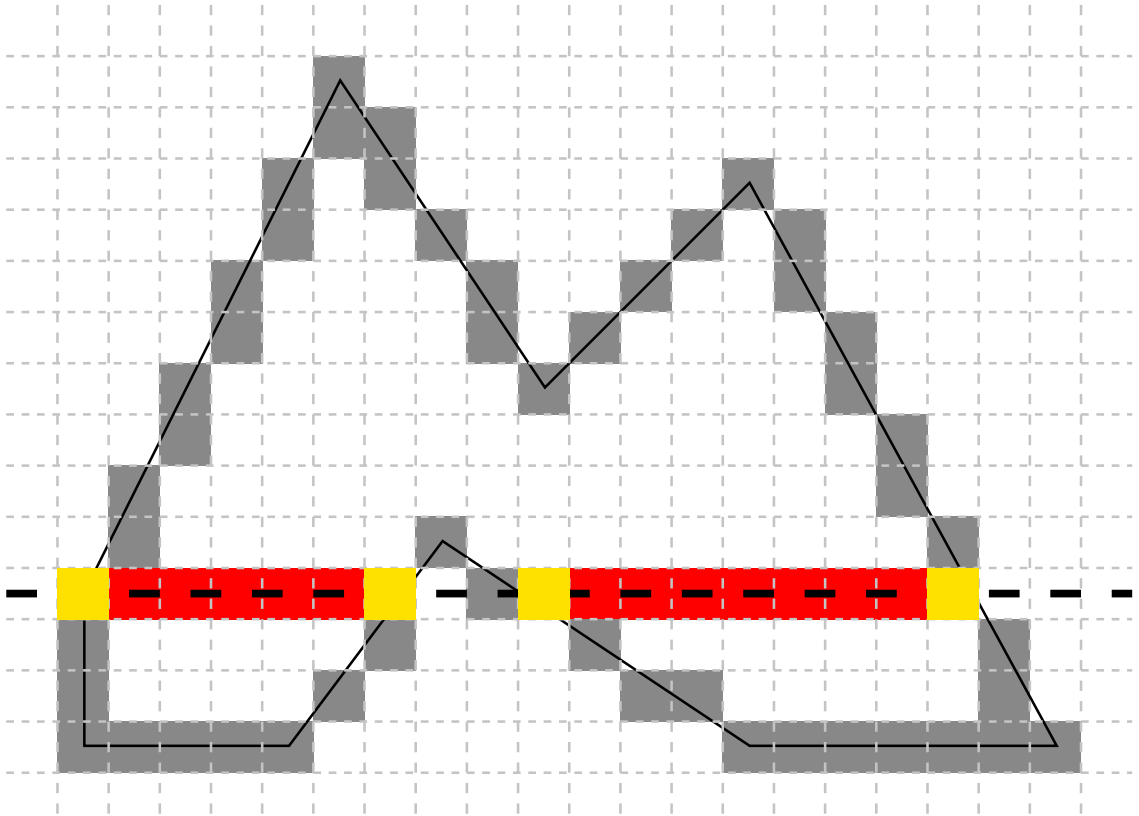
- The stack-based algorithm is a depth-first fill
- The stack can grow very quickly
- Better to use a queue, and fill breadth first



```
floodFill(Point p) {
  queue Q;
  paint(p);
  Q.enqueue(p);
  while (!Q.empty()) {
    q = Q.dequeue();
    foreach neighbour, r, of q {
      if (!painted(r)) {
        paint(r);
        Q.enqueue(r);
      }
    }
  }
}
```

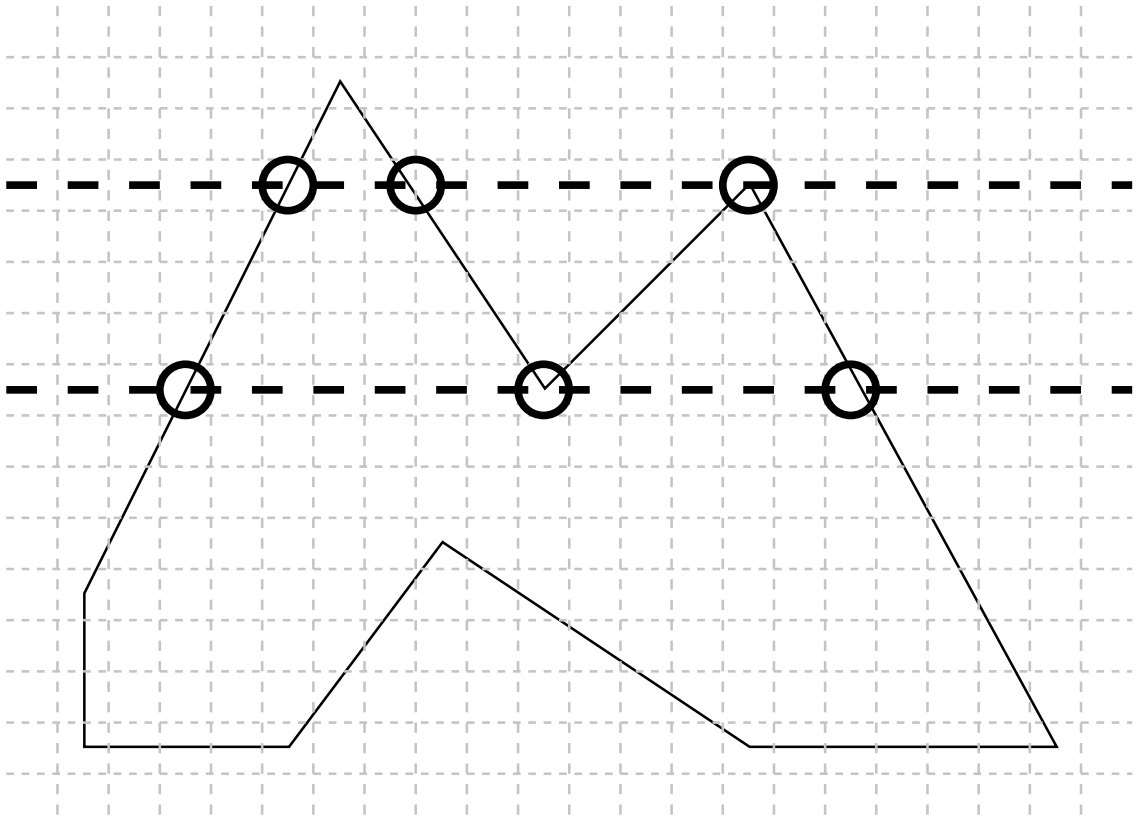
Scanline Filling

- A smarter way is to fill a polygon by scanline
- For each y value (row), find intersections with the polygon
- Fill in across the row between the intersections



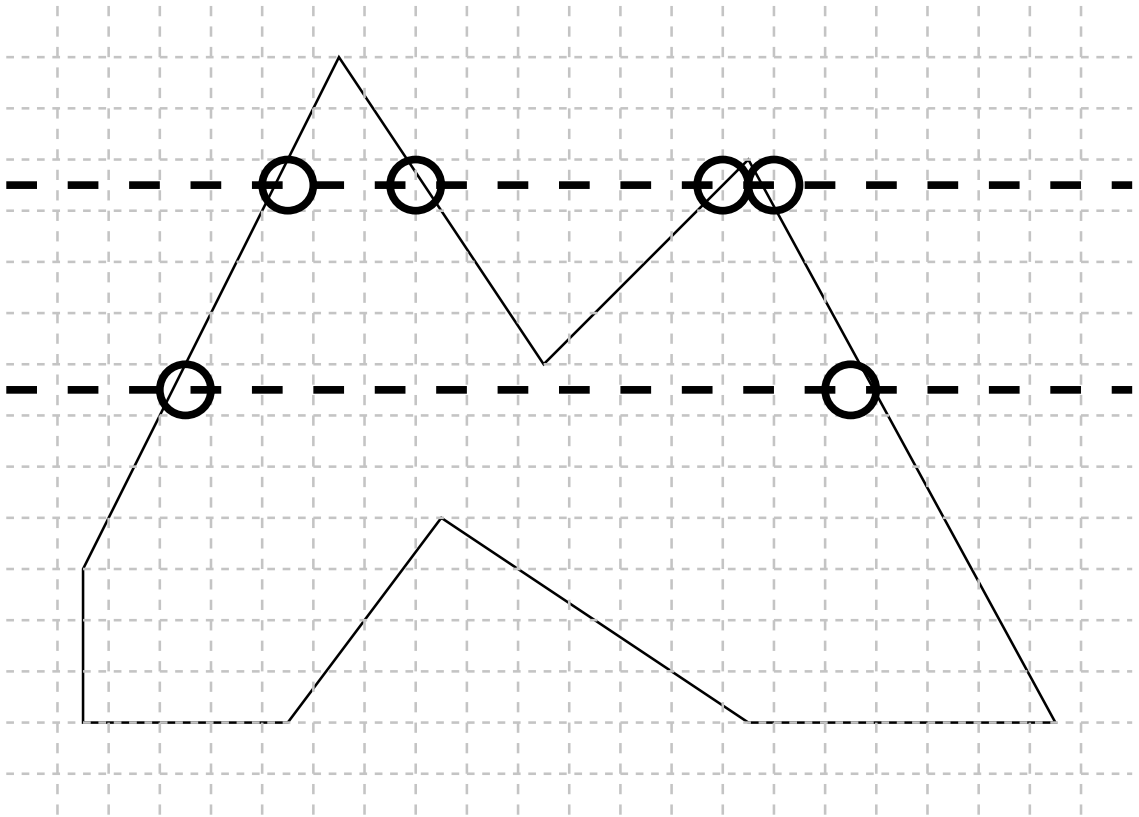
Scanline Filling

- This leads to issues with glancing intersections with the boundary
- In these cases there can be an odd number of intersections
- It is also possible to get an even number but no filling required



Scanline Filling

- This can be avoided by shifting the vertices of the polygon
- If the vertex y values are never integers, there are no issues
- Co-ords are often integers, so add/subtract a small offset



Finding Intersections

- Suppose we have a polygon edge from (x_0, y_0) to (x_1, y_1)
- Where does the scanline for some row intersect the edge?
- The line can be expressed as

$$(x, y) = (x_0, y_0) + \lambda(x_1 - x_0, y_1 - y_0)$$

- We know y so we can solve

$$y = y_0 + \lambda(y_1 - y_0)$$

  for λ, and substitute back to find x
- Note that if $\lambda < 0$ or $\lambda > 1$ there is no intersection

Tutorials and Labs

- Monday's Lab:
  - Matrices in C/C++/ OpenCV introduction
- Next week's tutorial:
  - Transformations in 2- and 3-dimensions
  - Representing transformations as matrices
  - Combining transformations