Scan Conversion

Overview of Rendering
Scan Conversion
    Drawing Lines
    Drawing Polygons

Carnegie Mellon

23 Mar. 2000

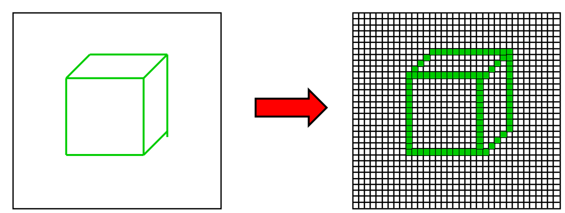
Overview of Course so Far

- We started with 2D graphics
  - Image processing
- Then moved to 3D
  - Modeling geometry (polygon meshes, implicit & parametric surfaces)
  - Illumination and shading (ambient, diffuse, specular)
  - Cameras, projection
- Then 4D: Animation
  - Traditional cartoon animation, keyframing, other techniques
- What now? 5D?
- *Rendering*
  - Creating images of things we've modeled
  - When most people talk about "computer graphics", this is the first thing that they think of

Rendering

- How to create images of our 3D models
- Why is this hard?
- Must determine what's visible
- Must simulate how light flows through the environment
- Different approaches
  - Painter's Algorithm and Z-Buffer Algorithm (OpenGL): draw objects one by one (scan conversion to determine which pixels to write)
  - Ray Tracing: shoot rays out from light source/viewer
  - Radiosity: subdivide surfaces with mesh, solve linear system
- We will start with low level, scan conversion, then move to higher level and fancier algorithms

Scan Conversion, a.k.a. Rasterization



Ideal Picture   Raster Representation

Scan Conversion: Process of converting ideal to raster

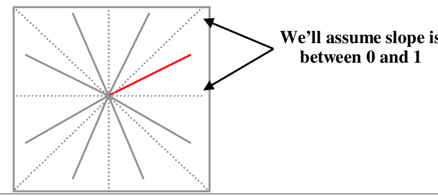
Example: http://graphics.lcs.mit.edu/classes/6.837/F98/Lecture5/Slide17.html

Scan Conversion Algorithms

- A discrete set of pixels can only approximate a continuous geometric object
- This means that scan conversion usually introduces error
- Properties of good scan conversion algorithms:
  - They should be as *accurate* as possible
  - They should be as *fast* as possible
- Challenges
  - Modify all the right pixels
  - Modify only the right pixels
  - Calculate their values correctly
  - Do it quickly
- *So, start with a correct algorithm and optimize it*

Line Drawing, Cases by Octant

- Lines come with different slopes, algorithm varies according to which octant (45 degree sector) it lies in.
- We will talk about one octant only; the algorithms generalize to the other octants easily.
- The algorithms for drawing lines need to step along one pixel at a time in the "fast" direction, but which direction this is depends on the slope of the line
- We also have to worry about reversed end point order (drawing from large to small X, for example).



A Really Simple Line Algorithm

- Equation for a line: $y(x) = mx + b$
- Fill in one pixel per column
- So, just evaluate for each x
  - This requires a choice of quadrant, so x steps evenly
- Certainly correct, but slow:
  - integer add, cast to float, floating multiply and add, plus round every step.

```
void line (int x0, int y0, int x1, int y1){
    float m = whatever;
    float b = whatever;
    int x;
    for(x=x0;x<=x1;x++) {
        float y= m*x + b;
        draw_pixel(x,Round(y));
    }
}
```


Lines: DDA Algorithm

- Optimize the previous to remove multiply from inner loop.
- Analogous to forward differencing for splines, but simpler.
- If we know y(x), we can calculate y(x+1):
  - $y(x+1) = mx + m + b = y(x) + m$

```
void line (int x0, int y0, int x1, int y1){
    float y = y0;
    float m = (y1 - y0)/ (float) (x1 - x0);
    int x;
    for(x=x0;x<=x1;x++) {
        draw_pixel(x,Round(y));
        y += m;
    }
}
```

- This is called *Differential Digital Analyzer* (DDA)
- **Problem:** Floating-point add and rounds are expensive

Bresenham's Algorithm

```
this does the right thing (same as DDA) at
a cost of only 2 or 3 integer adds per point.
(assumes sorted endpoints, 0<slope<1)

void draw_line(int x0, int y0, int x1, int y1) {
   int x, y = y0;
   int dx = 2*(x1-x0), dy = 2*(y1-y0);
   int dydx = dy-dx, F = dy-dx/2;

   for (x=x0 ; x<=x1 ; x++) {
      draw_pixel(x, y);
      if (F<0) F += dy;
      else {y++; F += dydx;}
   }
}

why does this work?
```


Implicit Function for a Line

$Line\ \mathbf{L}\ from\ [x_0, y_0]\ to\ [x_1, y_1].$

$\mathbf{P}_0 = [x_0, y_0],$

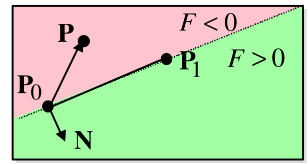
$\mathbf{P}_1 = [x_1, y_1].$

$dx = x_1 - x_0,\ dy = y_1 - y_0$

$\mathbf{N} = [dy, -dx]$

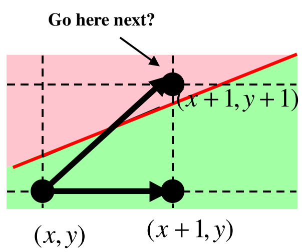
$implicit\ \ function: F(\mathbf{P}) = 2\mathbf{N} \cdot (\mathbf{P} - \mathbf{P}_0)$

$F = 0 \rightarrow \mathbf{P}\ is\ on\ \mathbf{L}$



Why the factor of 2? Because we're going to divide by 2 later.

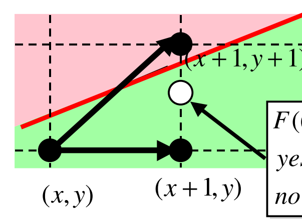
Line Drawing: Which Pixel is Next?



- **Assume:**
  - **0 < slope < 1**
  - **sorted endpoints, $x_0 < x_1$**
- **At each step:**
  - **Current point is *(x,y)***
  - **Next point is pixel (x+1,?) that's closest to the actual line**
  - **Do we increment *x* and *y* or only *x*?**
- **Use the implicit function to decide!**

Use the Implicit Function

- **Idea: Test the half-way point (x+1, y+1/2)**



$F((x+1, y+1/2)) > 0?$

$yes: \quad increment\ x\ and\ y$

$no: \quad increment\ x$

Trick: Incrementally Update *F*

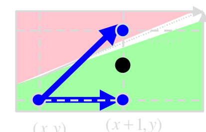
$$F(\mathbf{P}) = 2\mathbf{N}\cdot(\mathbf{P}-\mathbf{P}_0)$$
$$F(\mathbf{P}+\Delta) = 2\mathbf{N}\cdot(\mathbf{P}+\Delta-\mathbf{P}_0)$$
$$= F(\mathbf{P}) + 2\mathbf{N}\cdot\Delta$$

- **Computing *F*(P) requires a dot product:**
  - **2 multiplications and 1 add**
- **But computing *F*(P+Δ) requires only 1 add**
  - **The 2N• Δ term is constant - it only needs to be calculated once**
- **Δ is [1,0] or [1,1]**

Decision Variable F

$$F_0 = F(\mathbf{P}_0 + [1, 1/2])$$
$$= F(\mathbf{P}_0) + \mathbf{N}\cdot[2,1]$$
$$F' = F + 2\mathbf{N}\cdot\Delta$$
where
$$\Delta = [1,0] \; or \; [1,1]$$
$$\mathbf{N} = [dy, -dx]$$
so
$$F_0 = F(P_0) + 2dy - dx$$
$$F' = F + 2dy$$
or
$$F' = F + 2dy - 2dx$$

- **Initialize *x*, *y*, F**
- **Loop until end of line:**
  - **draw pixel *(x,y)***
  - **increment *x***
  - **if F>0, increment *y***
  - **increment F according to whether Δ is [1,0] or [1,1]**



Bresenham Line Algorithm

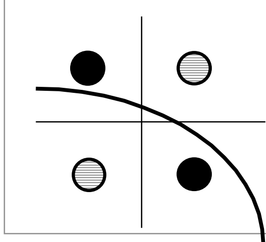
```
this does the right thing (same as DDA) at
a cost of only 2 or 3 integer adds per point.
(assumes sorted endpoints, 0<slope<1)

void draw_line(int x0, int y0, int x1, int y1) {
   int x, y = y0;
   int dx = 2*(x1-x0), dy = 2*(y1-y0);
   int dydx = dy-dx, F = dy-dx/2;

   for (x=x0; x<=x1; x++) {
      draw_pixel(x, y);
      if (F<0) F += dy;
      else {y++; F += dydx;}
   }
}
```


Bresenham Algorithm for Circles

- **Same approach as line algorithm**
  - **use a decision variable formula derived for a circle ($F = x^2 + y^2 - r^2$)**
- **Eightfold symmetry**
  - **only compute the points for one octant - use sign flips to give the rest**
- **Extends to general conics (ellipses...)**



Bresenham Circle Algorithm

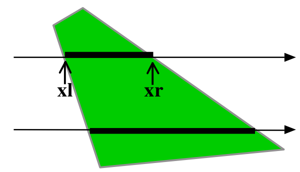
```
/* this draws a circle by calculating in one octant */
/* and re-using the resulting point 8 times */
void draw_circle(int radius) {
    int x = 0, y = radius;
    int d = 1-radius;
    while (y>x) {
        if (d<0) /* select East point next */
            d += 2*x + 3;
        else {    /* select South-East point next */
            d += 2*(x-y) + 5;
            y--;
        }
        x++;
        draw_8_pts(x,y); /* draws point in each octant */
    }
}
```


Scan Converting Filled, Convex Polygons: Basic Approach

- Find top and bottom vertices
- Make list of edges along left and right sides
- For each scanline from top to bottom
  - *There's a single span to fill*
  - Find left & right endpoints of span, xl & xr
  - Fill pixels inbetween xl & xr
  - Can use Bresenham's algorithm to update xl & xr as you step from line to line

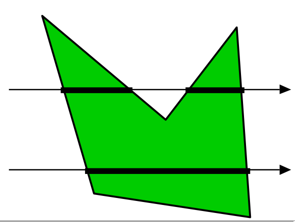
If you don't do all of the above carefully, cracks or overlaps between abutting polygons result!

BAD!



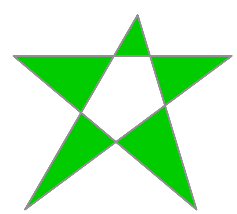
Scan Converting Filled, Concave Polygons: Basic Approach

- For each scanline
  - Find all the scanline/polygon intersections
  - Sort them left to right
  - Fill the interior *spans* between intersections
  - *Parity Rule:* odd ones are interior, even are exterior

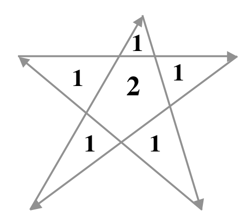


Inside/Outside Rules

- This even-odd rule is not the only option.
- An alternative would be the *winding rule:*
  - Pick a line to infinity (like the preceding part of the scan line)
  - Add up right-handed minus left-handed crossings
  - non-zero result means it's interior
- Only matters for self-intersecting primitives



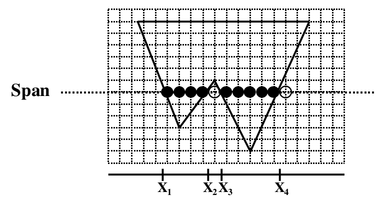
Even-odd rule



Winding rule

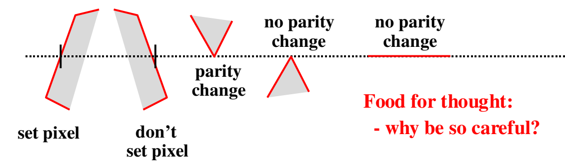
Span Filling

- Given a sorted list of scanline intersections (X1,X2,...,Xn)
  - Initialize: start at *ceiling*(X1), set *parity* = 1 (inside)
  - While x < Xn
    - » if (parity == 1), pixel = fillcolor
    - » x = x + 1
    - » if you've passed an intersection, *parity* = (1 - *parity*)



Special Cases

- For an integer intersection - exactly on a pixel
  - set the pixel if it's the beginning of a span, not if it's the end
- Shared vertices (possible grazing contact)
  - count ymin for parity, not ymax
- Horizontal edges
  - don't change parity



Food for thought:
- why be so careful?

Special Cases 2

- **Q**: Where did these inside/outside rules come from?
- **A**: Somebody made them up.

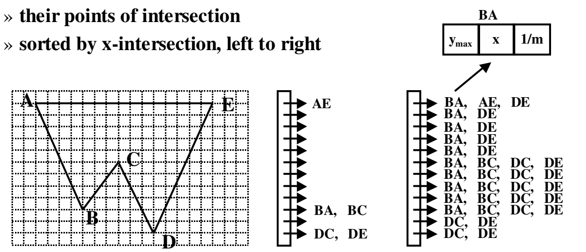
- You can resolve the special cases any way you want, as long as it's consistent:
  - Avoid drawing a pixel twice if the same edge is used for two adjacent polygons. (It's slower, and leads to errors in XOR mode)
  - Degenerate or horizontal edges should not leave seams.
  - Make sure inside regions stay inside (and vice versa) for small shifts of the polygon

Computing Edge/Scanline Intersections

- How to compute intersection points (X1, ..., Xn)?
- Brute force method
  - Calculate intersection of each scanline with each edge
- Better method
  - Exploit *coherence* between scanlines
    - » intersections do not change much between scanlines
- Cache the intersection info in a table
  - *Edge Table* with all edges sorted by *ymin*
  - *Active Edge Table* containing the edges that intersect the current scanline, sorted by x-intersection, left to right
- Process the image from the smallest *ymin* up
- When you run out of edges you're done

Caching Edge/Scanline Intersections

- Cache the intersection info in a table
  - *Edge Table* (ET) with all edges sorted by *ymin*
  - *Active Edge Table* (AET) containing
    - » the edges that intersect the current scanline
    - » their points of intersection
    - » sorted by x-intersection, left to right



Edge Table (ET) Active Edge Table (AET)