Reading for This Module

- FCG Chap 3 Raster Algorithms (through 3.2)
- Section 2.7 Triangles
- Section 8.1 Rasterization (through 8.1.2)
Scan Conversion - Rasterization

- convert continuous rendering primitives into discrete fragments/pixels
  - lines
    - midpoint/Bresenham
  - triangles
    - flood fill
    - scanline
    - implicit formulation
  - interpolation

Scan Conversion

- given vertices in DCS, fill in the pixels
- display coordinates required to provide scale for discretization
- [demo]

Basic Line Drawing

- assume $y = mx + b$
- $y = \frac{(y_1 - y_0)}{(x_1 - x_0)}(x - x_0) + y_0$

- goals
  - integer coordinates
  - thinnest line with no gaps

  - assume $x_0 < x_1$, slope $0 < \frac{dy}{dx} < 1$
  - one octant, other cases symmetric
  - how can we do this more quickly?

Midpoint Algorithm

- we’re moving horizontally along x direction (first octant)
- only two choices: draw at current y value, or move up vertically to y +1?
  - check if midpoint between two possible pixel centers above or below line
  - candidates
    - top pixel: $(x+1, y+1)$
    - bottom pixel: $(x+1, y)$
    - midpoint: $(x+1, y+.5)$

- check if midpoint above or below line
  - below: pick top pixel
  - above: pick bottom pixel

- other octants: different tests
  - octant II: y loop, check x left/right

Line $(x_0, y_0, x_1, y_1)$
begin
float $dx, dy, x, y, slope$ ;
dx $\leftarrow x_1 - x_0$;
dy $\leftarrow y_1 - y_0$;
slope $\leftarrow \frac{dy}{dx}$.
y $\leftarrow y_0$ ;
for $x$ from $x_0$ to $x_1$ do begin
  PlotPixel ( $x$, Round ($y$) ) ;
y $\leftarrow y + slope$ ;
end ;
end ;
**Midpoint Algorithm**

- we're moving horizontally along x direction (first octant)
  - only two choices: draw at current y value, or move up vertically to y + 1?
    - check if midpoint between two possible pixel centers above or below line
      - top pixel: (x+1,y+1)
      - bottom pixel: (x+1, y)
      - midpoint: (x+1, y+.5)
- check if midpoint above or below line
  - below: pick top pixel
  - above: pick bottom pixel
- key idea behind Bresenham
  - reuse computation from previous step
  - integer arithmetic by doubling values
  - [demo]

![Midpoint Algorithm Diagram]

**Bresenham, Detailed Derivation**

- Goal: function F tells us if line is above or below some point
  - F(x,y) = 0 on line
  - F(x,y) < 0 when line under point
  - F(x,y) > 0 when line over point

\[ y = mx + b \]
\[ y = \frac{dy}{dx} x + b \]
\[ dx \cdot y = dy \cdot x + b \cdot dx \]
\[ 0 = dy \cdot x - dx \cdot y + b \cdot dx \]
\[ 2 \cdot 0 = 2 \cdot dy \cdot x - 2 \cdot dx \cdot y + 2 \cdot b \cdot dx \]
\[ 0 = 2 \cdot dy \cdot x - 2 \cdot dx \cdot y + 2 \cdot b \cdot dx \]
\[ F(x, y) = 2 \cdot dy \cdot x - 2 \cdot dx \cdot y + 2 \cdot b \cdot dx \]

**Using F with Midpoints: Initial**

\[ F(x_0, y_0) = 2 \cdot dy \cdot x_0 - 2 \cdot dx \cdot y_0 + 2 \cdot b \cdot dx \]
\[ F(x_0 + 1, y_0 + .5) \]
\[ = 2 \cdot dy \cdot (x_0 + 1) - 2 \cdot dx \cdot (y_0 + .5) + 2 \cdot b \cdot dx \]
\[ = 2 \cdot dy \cdot x_0 + 2 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

**Incremental F: Initial**

\[ F(x_0, y_0) = 2 \cdot dy \cdot x_0 - 2 \cdot dx \cdot y_0 + 2 \cdot b \cdot dx \]
\[ F(x_0 + 1, y_0 + .5) \]
\[ = 2 \cdot dy \cdot (x_0 + 1) - 2 \cdot dx \cdot (y_0 + .5) + 2 \cdot b \cdot dx \]
\[ = 2 \cdot dy \cdot x_0 + 2 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]
\[ F(x_0 + 1, y_0 + .5) - F(x_0, y_0) = 2 \cdot dy - dx = \text{diff} \]
Using F with Midpoints: No Y Change

\[ F(x_0 + 1, y_0 + .5) = 2 \cdot dy \cdot x_0 + 2 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

\[ F(x_0 + 2, y_0 + .5) = 2 \cdot dy \cdot (x_0 + 2) - 2 \cdot dx \cdot (y_0 + .5) + 2 \cdot b \cdot dx \]

\[ F(x_0 + 3, y_0 + 1.5) = 2 \cdot dy \cdot x_0 + 4 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

Next difference in F: 2*dy-2*dx (when pixel at y+1)

Using F with Midpoints: Y Increased

\[ F(x_0 + 1, y_0 + .5) = 2 \cdot dy \cdot x_0 + 2 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

\[ F(x_0 + 2, y_0 + .5) = 2 \cdot dy \cdot (x_0 + 2) - 2 \cdot dx \cdot (y_0 + .5) + 2 \cdot b \cdot dx \]

\[ F(x_0 + 3, y_0 + 1.5) = 2 \cdot dy \cdot x_0 + 4 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

Incremental F: No Y Change

\[ F(x_0 + 1, y_0 + .5) = 2 \cdot dy \cdot x_0 + 2 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

\[ F(x_0 + 2, y_0 + .5) = 2 \cdot dy \cdot (x_0 + 2) - 2 \cdot dx \cdot (y_0 + .5) + 2 \cdot b \cdot dx \]

\[ F(x_0 + 3, y_0 + 1.5) = 2 \cdot dy \cdot x_0 + 4 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

Incremental F: Y Increased

\[ F(x_0 + 1, y_0 + .5) = 2 \cdot dy \cdot x_0 + 2 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

\[ F(x_0 + 2, y_0 + .5) = 2 \cdot dy \cdot (x_0 + 2) - 2 \cdot dx \cdot (y_0 + .5) + 2 \cdot b \cdot dx \]

\[ F(x_0 + 3, y_0 + 1.5) = 2 \cdot dy \cdot x_0 + 4 \cdot dy - 2 \cdot dx \cdot y_0 - dx + 2 \cdot b \cdot dx \]

Next difference in F: 2*dy-2*dx (when pixel at y+1)
Bresenham: Reuse Computation, Integer Only

\[
y = y_0; \\
dx = x_1 - x_0; \\
dy = y_1 - y_0; \\
d = 2*dy - dx; \\\nincKeepY = 2*dy; \\
incIncreaseY = 2*dy - 2*dx; \\
for (x = x_0; x <= x_1; x++) {
\begin{align*}
    &\text{draw}(x, y) \\
    &\text{if } (d > 0) \text{ then} \\
    &\quad y = y + 1; \\
    &\quad d += incIncreaseY; \\
    &\text{else} \\
    &\quad d += incKeepY; \\
\end{align*}
\]

Rasterizing Polygons/Triangles

- basic surface representation in rendering
- why?
  - lowest common denominator
    - can approximate any surface with arbitrary accuracy
      - all polygons can be broken up into triangles
  - guaranteed to be:
    - planar
    - triangles - convex
  - simple to render
  - can implement in hardware

Triangulating Polygons

- simple convex polygons
  - trivial to break into triangles
  - pick one vertex, draw lines to all others not immediately adjacent
  - OpenGL supports automatically
    - \text{glBegin(GL\_POLYGON)} ... \text{glEnd()}
- concave or non-simple polygons
  - more effort to break into triangles
  - simple approach may not work
  - OpenGL can support at extra cost
    - \text{gluNewTess()}, \text{gluTessCallback()}, ...

Problem

- input: closed 2D polygon
- problem: fill its interior with specified color on graphics display
- assumptions
  - simple - no self intersections
  - simply connected
- solutions
  - flood fill
  - edge walking
Flood Fill

- simple algorithm
- draw edges of polygon
- use flood-fill to draw interior

Flood Fill

- start with seed point
- recursively set all neighbors until boundary is hit

Flood Fill

- draw edges
- run:
  - FloodFill(Polygon P, int x, int y, Color C)
  - if not (OnBoundary(x,y,P) or Colored(x,y,C))
  - begin
  - PlotPixel(x,y,C);
  - FloodFill(P,x + 1,y,C);
  - FloodFill(P,x,y + 1,C);
  - FloodFill(P,x,y - 1,C);
  - FloodFill(P,x - 1,y,C);
  - end ;
- drawbacks?
Scanline Algorithms

- **scanline**: a line of pixels in an image
- set pixels inside polygon boundary along horizontal lines one pixel apart vertically

General Polygon Rasterization

- how do we know whether given pixel on scanline is inside or outside polygon?

General Polygon Rasterization

- idea: use a parity test
  
  ```
  for each scanline
    edgeCnt = 0;
    for each pixel on scanline (l to r)
      if (oldpixel->newpixel crosses edge)
        edgeCnt ++;
      // draw the pixel if edgeCnt odd
      if (edgeCnt % 2)
        setPixel(pixel);
  ```

Making It Fast: Bounding Box

- smaller set of candidate pixels
- loop over xmin, xmax and ymin,ymax instead of all x, all y
Triangle Rasterization Issues

- moving slivers
- shared edge ordering

Interpolation During Scan Conversion

- drawing pixels in polygon requires interpolating many values between vertices
  - r,g,b colour components
    - use for shading
  - z values
  - u,v texture coordinates
  - $N_x, N_y, N_z$ surface normals
- equivalent methods (for triangles)
  - bilinear interpolation
  - barycentric coordinates

Triangle Rasterization Issues

- exactly which pixels should be lit?
  - pixels with centers inside triangle edges
- what about pixels exactly on edge?
  - draw them: order of triangles matters (it shouldn’t)
  - don’t draw them: gaps possible between triangles
  - need a consistent (if arbitrary) rule
    - example: draw pixels on left or top edge, but not right or bottom edge
    - example: check if triangle on same side of edge as offscreen point

Interpolation

- equivalent methods (for triangles)
  - bilinear interpolation
  - barycentric coordinates
### Bilinear Interpolation

- Interpolate quantity along $L$ and $R$ edges, as a function of $y$
  - Then interpolate quantity as a function of $x$

### Barycentric Coordinates

- Non-orthogonal coordinate system based on triangle itself
  - Origin: $P_1$, basis vectors: $(P_2-P_1)$ and $(P_3-P_1)$

\[
P = P_1 + \beta(P_2-P_1)+\gamma(P_3-P_1)
\]

### Barycentric Coordinates

- Non-orthogonal coordinate system based on triangle itself
  - Origin: $P_1$, basis vectors: $(P_2-P_1)$ and $(P_3-P_1)$

\[
P = (1-\beta-\gamma)P_1 + \beta P_2+\gamma P_3
\]

\[
P = \alpha P_1 + \beta P_2+\gamma P_3
\]
Using Barycentric Coordinates

- weighted combination of vertices
- smooth mixing
- speedup
  - compute once per triangle

\[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
\[ \alpha + \beta + \gamma = 1 \]
\[ 0 \leq \alpha, \beta, \gamma \leq 1 \] for points inside triangle

"convex combination of points"

Deriving Barycentric From Bilinear

- from bilinear interpolation of point \( P \) on scanline

\[ P_L = P_2 + \frac{d_1}{d_1 + d_2} (P_3 - P_2) \]
\[ = (1 - \frac{d_1}{d_1 + d_2}) P_2 + \frac{d_1}{d_1 + d_2} P_3 = \]
\[ = \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3 \]

Deriving Barycentric From Bilinear

- combining

\[ P = \frac{c_2}{c_1 + c_2} \cdot P_L + \frac{c_1}{c_1 + c_2} \cdot P_R \]
\[ P_L = \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3 \]
\[ P_R = \frac{b_2}{b_1 + b_2} P_2 + \frac{b_1}{b_1 + b_2} P_1 \]

- gives

\[ P = \frac{c_2}{c_1 + c_2} \left( \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3 \right) + \frac{c_1}{c_1 + c_2} \left( \frac{b_2}{b_1 + b_2} P_2 + \frac{b_1}{b_1 + b_2} P_1 \right) \]
Deriving Barycentric From Bilinear

- thus $P = \alpha P_1 + \beta P_2 + \gamma P_3$ with
  \[\alpha = \frac{c_1}{c_1 + c_2} \frac{b_1}{b_1 + b_2}\]
  \[\beta = \frac{c_2}{c_1 + c_2} \frac{d_2}{d_1 + d_2} + \frac{c_1}{c_1 + c_2} \frac{b_2}{b_1 + b_2}\]
  \[\gamma = \frac{c_2}{c_1 + c_2} \frac{d_1}{d_1 + d_2}\]
- can verify barycentric properties

\[\alpha + \beta + \gamma = 1, \quad 0 \leq \alpha, \beta, \gamma \leq 1\]

Computing Barycentric Coordinates

- 2D triangle area
- half of parallelogram area
- from cross product

\[A = \frac{A_{P_1} + A_{P_2} + A_{P_3}}{P_1 A_1 P_3 P_2}\]

\[\alpha = \frac{A_{P_1}}{A}\]
\[\beta = \frac{A_{P_2}}{A}\]
\[\gamma = \frac{A_{P_3}}{A}\]