Reading for This Module

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

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, y+1)
  - bottom pixel: (x, y)
- 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

- Goal: function F tells us if line is above or below some point
  - F(x0, y0) = 0 on line
  - F(x0, y0) < 0 when line under point
  - F(x0, y0) > 0 when line over point

\[
y = \text{int} + b \\
= \frac{(y_2 - y_1)}{(x_2 - x_1)} \\
= \frac{x - x_1}{y_2 - y_1}
\]

Incremental F: Initial

\[
F(x_0, y_0) = 2^*dy + x_0 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0, y_1) = 2^*dy + x_1 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0, y_2) = 2^*dy + x_2 - 2^*dx + y_2 + 2^*b + dx \\
F(x_0 + 1, y_0) = 2^*dy + x_0 + 1 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0 + 1, y_1) = 2^*dy + x_1 + 1 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0 + 1, y_2) = 2^*dy + x_2 + 1 - 2^*dx + y_2 + 2^*b + dx \\
\]

Incremental F: Y Increased

\[
F(x_0 + 2, y_0 + 5) = 2^*dy + x_0 + 4 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0 + 2, y_1 + 5) = 2^*dy + x_1 + 4 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0 + 2, y_2 + 5) = 2^*dy + x_2 + 4 - 2^*dx + y_2 + 2^*b + dx \\
\]

Using F with Midpoints: Y Increased

\[
F(x_0 + 2, y_0 + 5) = 2^*dy + x_0 + 4 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0 + 2, y_1 + 5) = 2^*dy + x_1 + 4 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0 + 2, y_2 + 5) = 2^*dy + x_2 + 4 - 2^*dx + y_2 + 2^*b + dx \\
\]

Incremental F: Y Increased

\[
F(x_0 + 2, y_0 + 5) = 2^*dy + x_0 + 4 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0 + 2, y_1 + 5) = 2^*dy + x_1 + 4 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0 + 2, y_2 + 5) = 2^*dy + x_2 + 4 - 2^*dx + y_2 + 2^*b + dx \\
\]

Using F with Midpoints: No Y Change

\[
F(x_0, y_0 + 5) = 2^*dy + x_0 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0, y_1 + 5) = 2^*dy + x_1 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0, y_2 + 5) = 2^*dy + x_2 - 2^*dx + y_2 + 2^*b + dx \\
F(x_0 + 1, y_0 + 5) = 2^*dy + x_0 + 1 - 2^*dx + y_0 + 2^*b + dx \\
\]

Incremental F: No Y Change

\[
F(x_0, y_0 + 5) = 2^*dy + x_0 - 2^*dx + y_0 + 2^*b + dx \\
F(x_0, y_1 + 5) = 2^*dy + x_1 - 2^*dx + y_1 + 2^*b + dx \\
F(x_0, y_2 + 5) = 2^*dy + x_2 - 2^*dx + y_2 + 2^*b + dx \\
F(x_0 + 1, y_0 + 5) = 2^*dy + x_0 + 1 - 2^*dx + y_0 + 2^*b + dx \\
\]
Bresenham: Reuse Computation, Integer Only

g = y0;!
dx = x1-x0;!
dy = y1-y0;!
d = 2*dy-dx;!
incKeepY = 2*dy;!
incIncreaseY = 2*dy - 2*dx;!
for (x = x0; x <= x1; x++) {
  if (d > 0) {
    d += incIncreaseY;!
  } else {
    d += incKeepY;!
  }
}

Bresenham: Reuse Computation, Integer Only

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
    - glBegin(GL_POLYGON), glEnd()

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

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

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

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 on right or bottom edge
  - example: check if triangle on same side of edge as offscreen point

Interpolation During Scan Conversion

- drawing pixels in polygon requires interpolating many values between vertices
  - x,y,b colour components
  - use for shading
- 2 values
  - u,v texture coordinates
  - N, N1, N2 surface normals
- 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

Using Barycentric Coordinates
- weighted combination of vertices
- smooth mixing
- speedup
- compute once per triangle
  \[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
  \( \alpha + \beta + \gamma = 1 \) for points inside triangle

Deriving Barycentric From Bilinear
- from bilinear interpolation of point P on scanline

Deriving Barycentric From Bilinear
• combining

Deriving Barycentric From Bilinear
• similarly

Computing Barycentric Coordinates
• 2D triangle area
  - half of parallelogram area
  - from cross product

\[ A = A_{p1} + A_{p2} + A_{p3} \]
\[ \alpha = A_{p2} / A \]
\[ \beta = A_{p3} / A \]
\[ \gamma = A_{p1} / A \]