Review: YIQ Color Space
- color model used for color TV
- Y is luminance (same as CIE)
- I & Q are color (not same I as HSI!)
- using Y backwards compatible for B/W TVs
- conversion from RGB is linear
- green is much lighter than red, and red lighter than blue

$$Y = 0.30 R + 0.59 G + 0.11 B$$

$$I = 0.60 R - 0.28 G - 0.32 B$$

$$Q = 0.21 R + 0.72 G - 0.33 B$$

Review: Luminance vs. Intensity
- luminance
- Y of YIQ
- 0.299R + 0.587G + 0.114B
- intensity/brightness
- IV/B of HSI/HSV/HSB
- 0.333R + 0.333G + 0.333B

Review: Color Constancy
- automatic "white balance" from change in illumination
- vast amount of processing behind the scenes!
- colorimetry vs. perception

Making It Fast: Reuse Computation
- midpoint if f(x+1, y+0.5) < 0 then y = y+1
- on previous step evaluated f(x-1, y-0.5) or f(x-1, y-0.05)
- f(x+1, y) = f(x,y) + f(y/2)
- f(x+1, y) = f(x,y) + (y+y+1)

$$y = y + b$$

$$\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} x \end{bmatrix} + \begin{bmatrix} y \end{bmatrix}$$

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

Midpoint Algorithm
- we're moving horizontally along x direction
- 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+0.5)
- check if midpoint above or below line
- below: top pixel
- above: pick bottom pixel
- key idea behind Bresenham
- [demo]

Making It Fast: Integer Only
- avoid dealing with non-integer values by doubling both sides

Making It Fast: Integer Only
- avoid dealing with non-integer values by doubling both sides

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()
- concave or non-simple polygons
- more effort to break into triangles
- simple approach may not work
- OpenGL can support at extra cost
  - glNewTriangle(), glTriangleArray()...
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

Making It Fast: Bounding Box
- smaller set of candidate pixels
- loop over xmin, xmax and ymin, ymax instead of all x, all y

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;

Flood Fill
- drawbacks?

Flood Fill Drawbacks
- pixels visited up to 4 times to check if already set
- need per-pixel flag indicating if set already
- must clear for every polygon!

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 (1 to r)
    if (oldpixel->newpixel)
      edgeCnt ++;
    // draw the pixel if edgeCnt odd
    if (edgeCnt % 2)
      setPixel(pixel);

Scanline Algorithms
- set pixels inside polygon boundary along horizontal lines one pixel apart vertically

Triangle Rasterization Issues
- exactly which pixels should be lit?
  - pixels with centers inside triangle edges
  - what about pixels exactly on edge?
  - move slivers
  - shared edge ordering

Triangle Rasterization Issues
- moving slivers

Interpolation
- 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

Interpolation
- bilinear interpolation
  - interpolate quantity along L and R edges, as a function of y
    - then interpolate quantity as a function of x

Interpolation
- non-orthogonal coordinate system based on triangle itself
  - origin: P_0, basis vectors: (P_2-P_1) and (P_3-P_1)
  \[ P = P_0 + \gamma (P_2-P_1) + \beta (P_3-P_1) \]

Barycentric Coordinates
- origin: P_0, basis vectors: (P_2-P_1) and (P_3-P_1)

Barycentric Coordinates
- origin: P_0, basis vectors: (P_2-P_1) and (P_3-P_1)

Barycentric Coordinates
- origin: P_0, basis vectors: (P_2-P_1) and (P_3-P_1)
Barycentric Coordinates
- non-orthogonal coordinate system based on triangle itself
  - origin: \( P_1 \), basis vectors: \( (P_2 - P_1) \) and \( (P_3 - P_1) \)

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

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

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

\[
\alpha + \beta + \gamma = 1
\]

Deriving Barycentric From Bilinear
- combining
- gives

\[
P = \frac{c_2}{c_1 + c_2} P_1 + \frac{c_1}{c_1 + c_2} P_2
\]

\[
P = \frac{d_1}{d_1 + d_2} P_1 + \frac{d_2}{d_1 + d_2} P_2
\]

\[
P = \frac{b_1}{b_1 + b_2} P_1 + \frac{b_2}{b_1 + b_2} P_2
\]

Deriving Barycentric From Bilinear
- thus \( P = \alpha P_1 + \beta P_2 + \gamma P_3 \) with

\[
\alpha = \frac{c_1 b_2}{c_1 b_2 + c_2 b_1}
\]

\[
\beta = \frac{c_2 d_2}{c_2 d_2 + c_1 d_1 + c_1 b_1 + c_2 b_2}
\]

\[
\gamma = \frac{c_1 d_1}{c_1 d_1 + c_2 d_2 + c_1 b_1 + c_2 b_2}
\]

- 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

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

\[
\alpha = A_{P_1} / A
\]

\[
\beta = A_{P_2} / A
\]

\[
\gamma = A_{P_3} / A
\]