University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2007

Tamara Munzner

Rasterization

Week 6, Mon Feb 12

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2007

Reading for Today

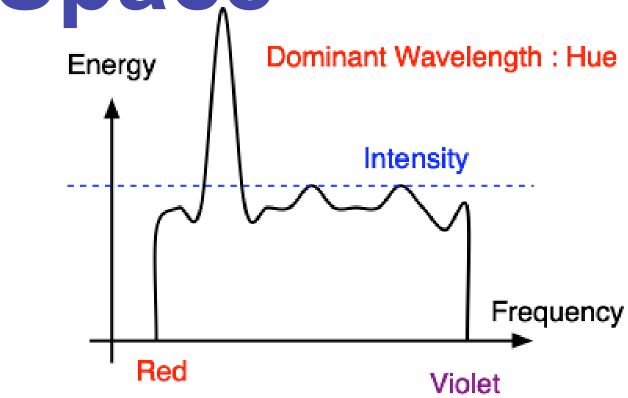
- FCG Chap 3 Raster Algorithms
  - (except 3.2-3.4, 3.8)
- FCG Section 2.11 Triangles

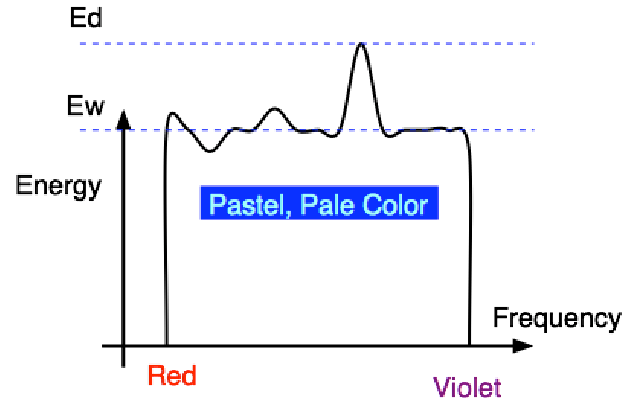
Reading for Next Three Lectures

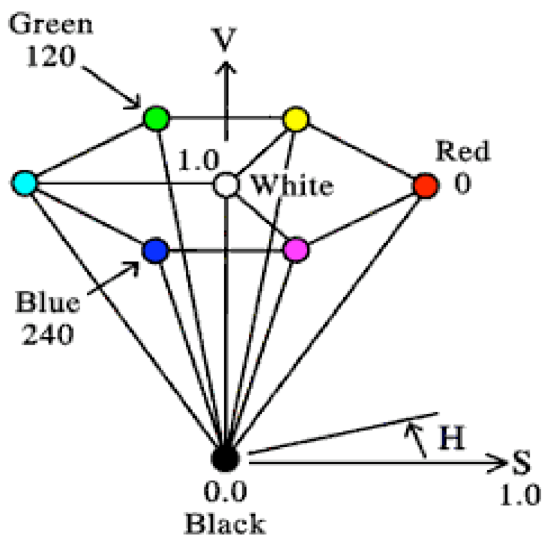
- FCG Chap 9 Surface Shading
- RB Chap Lighting

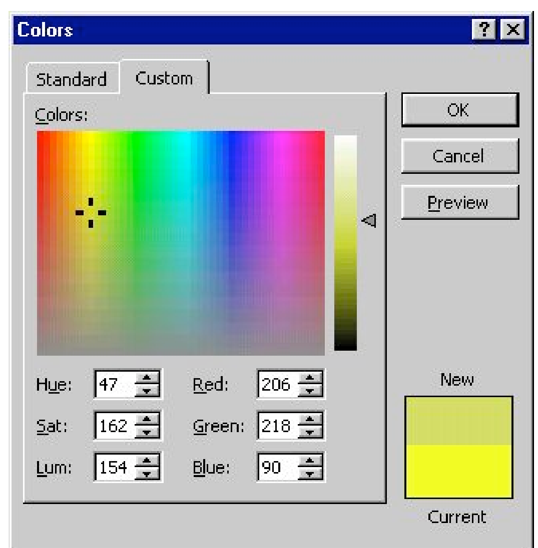
Review: HSV Color Space

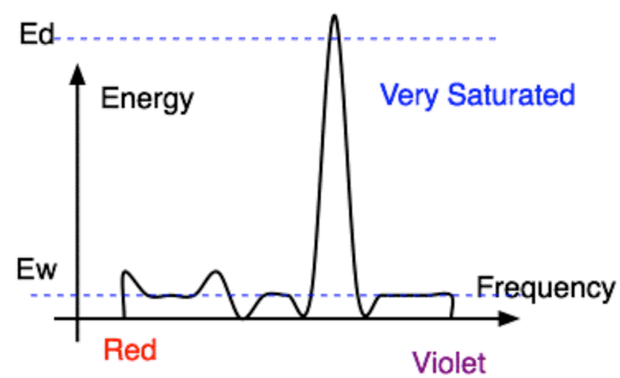
- hue: dominant wavelength, “color”
- saturation: how far from grey
- value/brightness: how far from black/white
- cannot convert to RGB with matrix alone



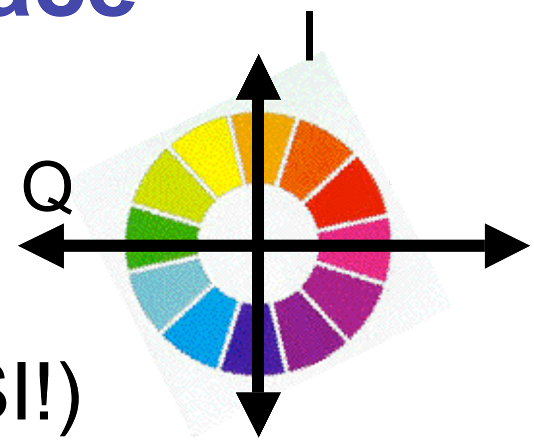








Review: YIQ Color Space



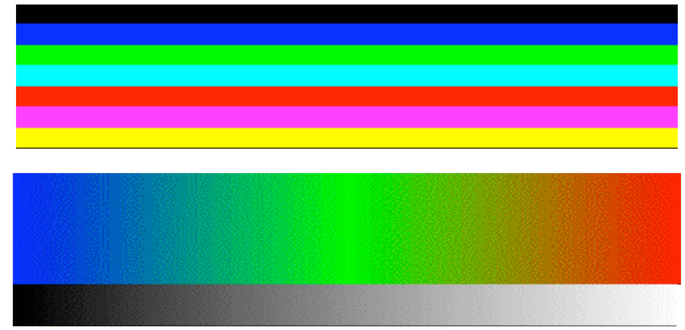
- color model used for color TV
  - Y is luminance (same as CIE)
  - I & Q are color (not same I as HSI!)
  - using Y backwards compatible for B/W TVs
  - conversion from RGB is linear

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- green is much lighter than red, and red lighter than blue

Review: Luminance vs. Intensity

- luminance
  - Y of YIQ
  - 0.299R + 0.587G + 0.114B
- intensity/brightness
  - I/V/B of HSI/HSV/HSB
  - 0.333R + 0.333G + 0.333B



(a) Colour Image

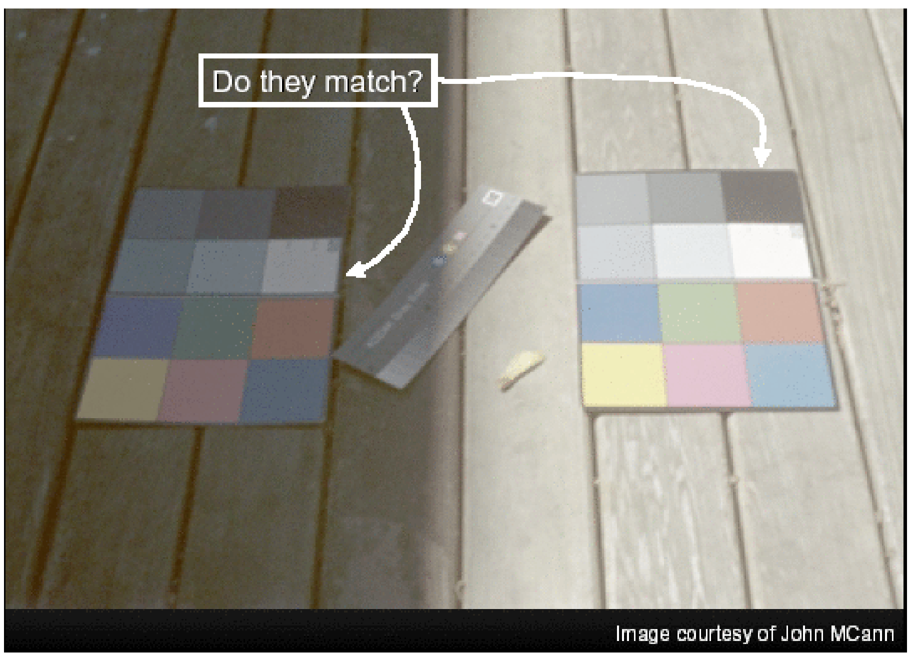
(b) Intensity Image

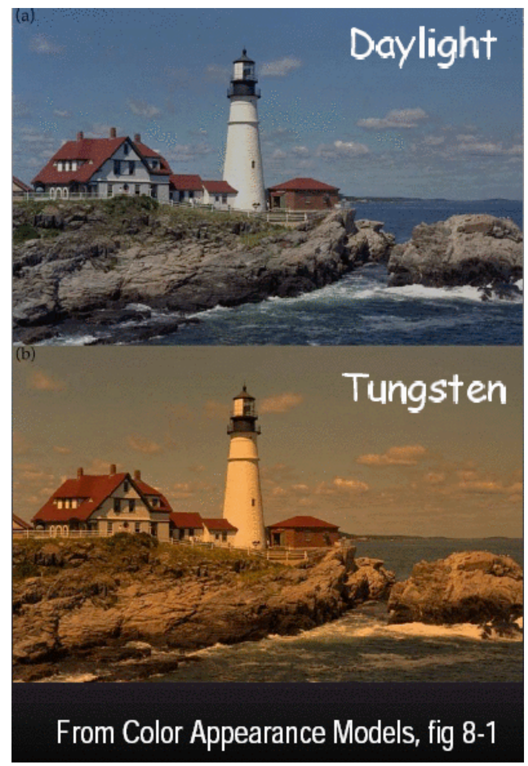
(c) Luminance Image

www.csse.uwa.edu.au/~robyn/Visioncourse/colour/lecture/node5.html

Review: Color Constancy

- automatic “white balance” from change in illumination
- vast amount of processing behind the scenes!
- colorimetry vs. perception





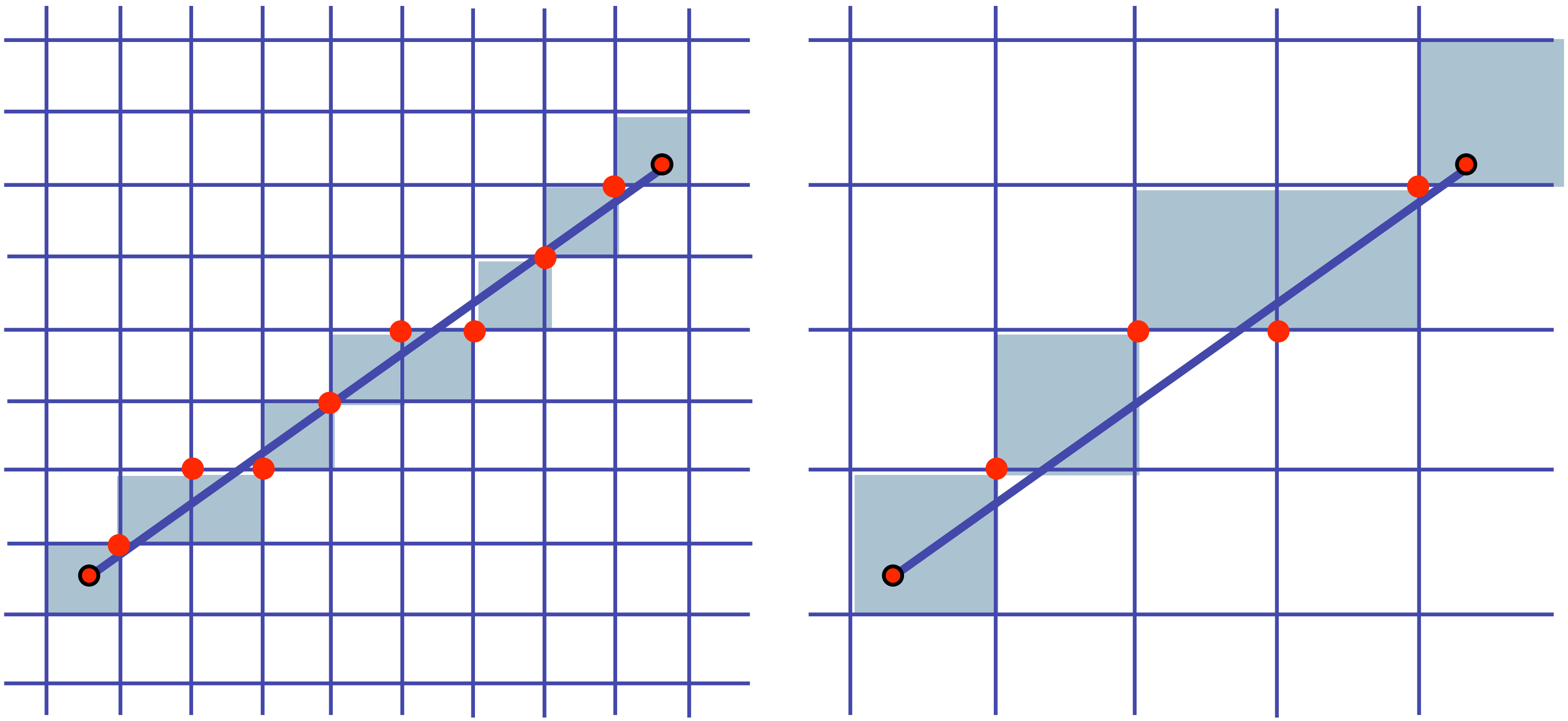
Rasterization

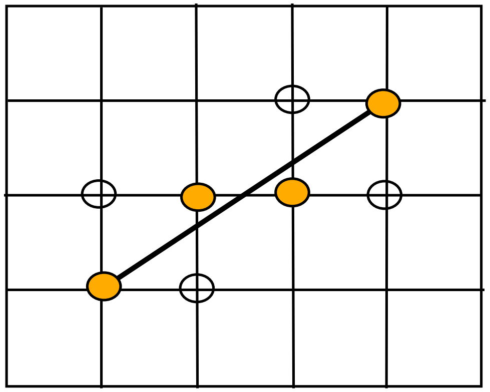
Scan Conversion - Rasterization

- 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

$$y = mx + b$$

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

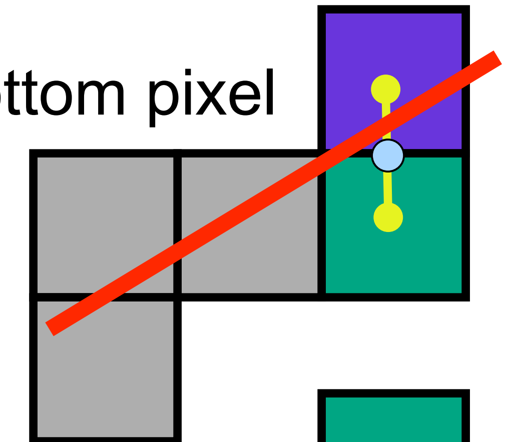
- goals
  - integer coordinates
  - thinnest line with no gaps
- assume
  - $x_0 < x_1$, slope $0 < \frac{dy}{dx} < 1$
  - one octant, other cases symmetric
- how can we do this more quickly?

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 ;

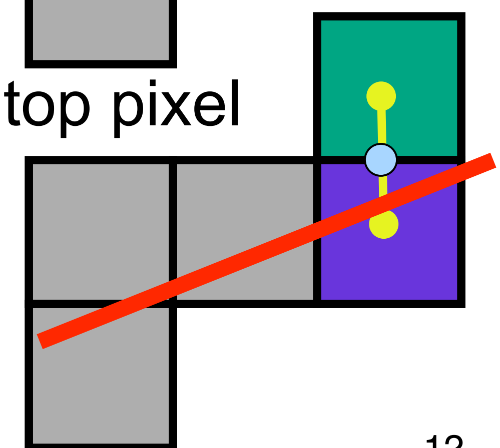
Midpoint Algorithm

- we're moving horizontally along x direction
  - 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
- key idea behind Bresenham
  - [demo]

above: bottom pixel



below: top pixel



Making It Fast: Reuse Computation

- midpoint: if f(x+1, y+.5) < 0 then y = y+1
- on previous step evaluated f(x-1, y-.5) or f(x-1, y+.05)
- $f(x+1, y) = f(x,y) + (y_0 - y_1)$
- $f(x+1, y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$

```
y=y0
d = f(x0+1, y0+.5)
for (x=x0; x <= x1; x++) {
  draw(x,y);
  if (d<0) then {
   y = y + 1;
   d = d + (x1 - x0) + (y0 - y1)
  } else {
   d = d + (y0 - y1)
}
```

Making It Fast: Integer Only

- avoid dealing with non-integer values by doubling both sides

```
y=y0
d = f(x0+1, y0+.5)
for (x=x0; x <= x1; x++)
  {
  draw(x,y);
  if (d<0) then {
   y = y + 1;
   d = d + (x1 - x0) +
           (y0 - y1)
  } else {
   d = d + (y0 - y1)
}
```

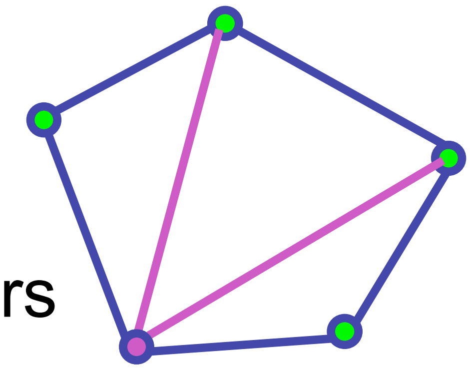
```
y=y0
2d = 2*(y0-y1)(x0+1) +
      (x1-x0)(2y0+1) +
      2x0y1 - 2x1y0
for (x=x0; x <= x1; x++) {
  draw(x,y);
  if (d<0) then {
   y = y + 1;
   d = d + 2(x1 - x0) +
           2(y0 - y1)
  } else {
   d = d + 2(y0 - y1)
}
```

Rasterizing Polygons/Triangles

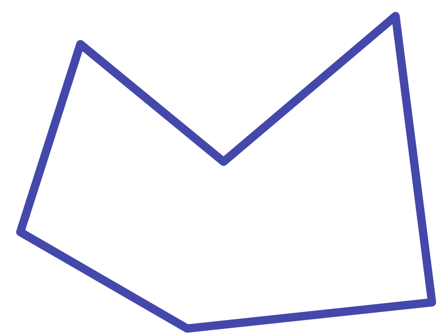
- basic surface representation in rendering
- why?
  - lowest common denominator
    - can approximate any surface with arbitrary accuracy
      - all polygons can be broken up into triangles
  - guaranteed to be:
    - planar
    - triangles - convex
  - simple to render
    - can implement in hardware

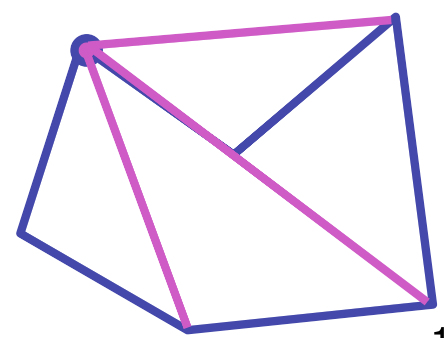
Triangulating Polygons

- 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()



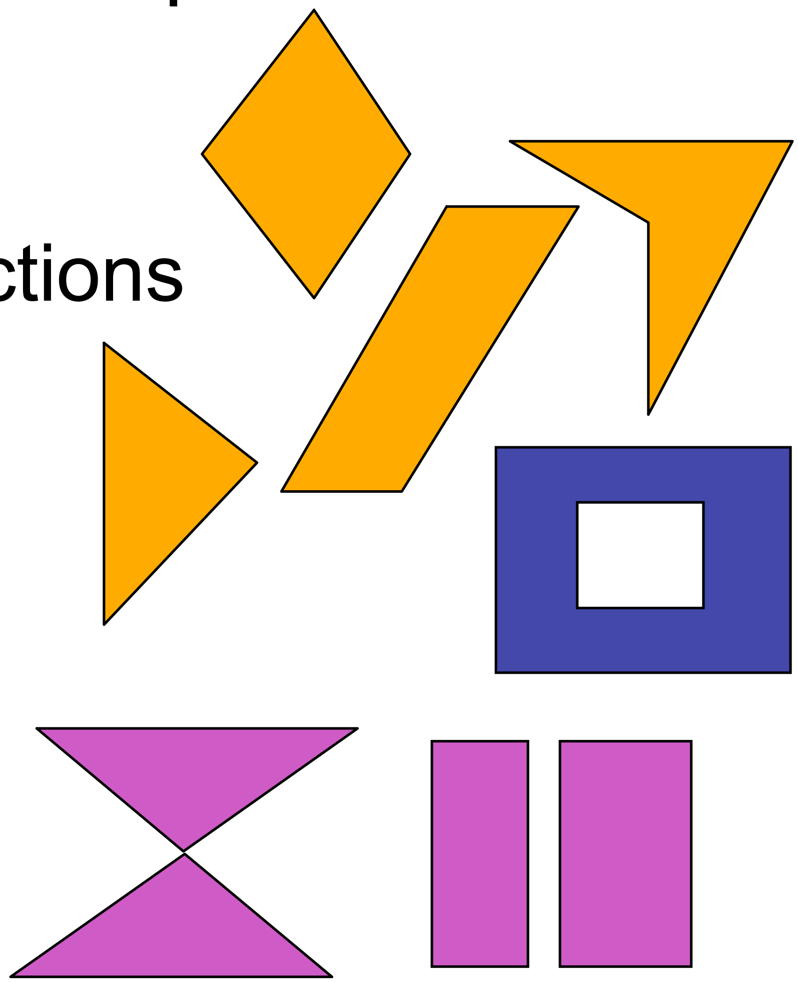
- concave or non-simple polygons
  - more effort to break into triangles
  - simple approach may not work
  - OpenGL can support at extra cost
    - gluNewTess(), gluTessCallback(), ...





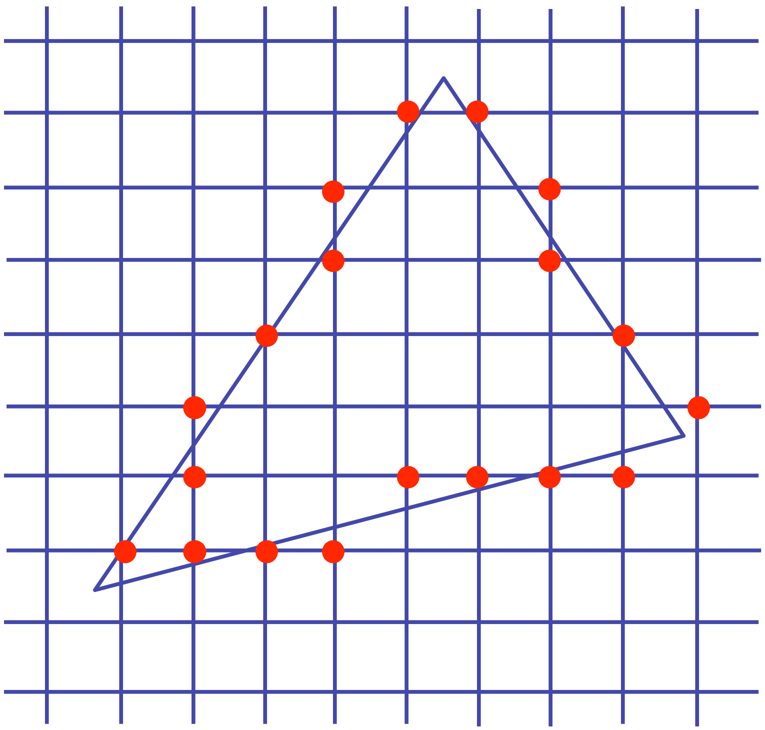
Problem

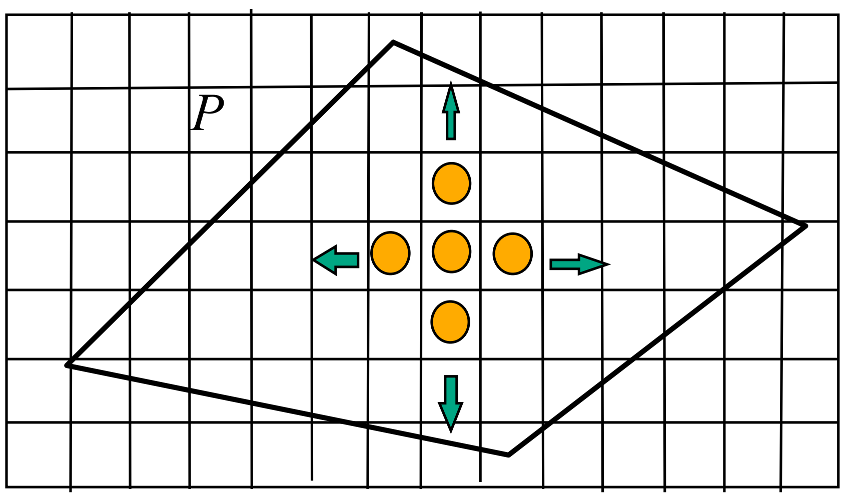
- input: closed 2D polygon
- problem: fill its interior with specified color on graphics display
- assumptions
  - simple - no self intersections
  - simply connected
- solutions
  - flood fill
  - edge walking



Flood Fill

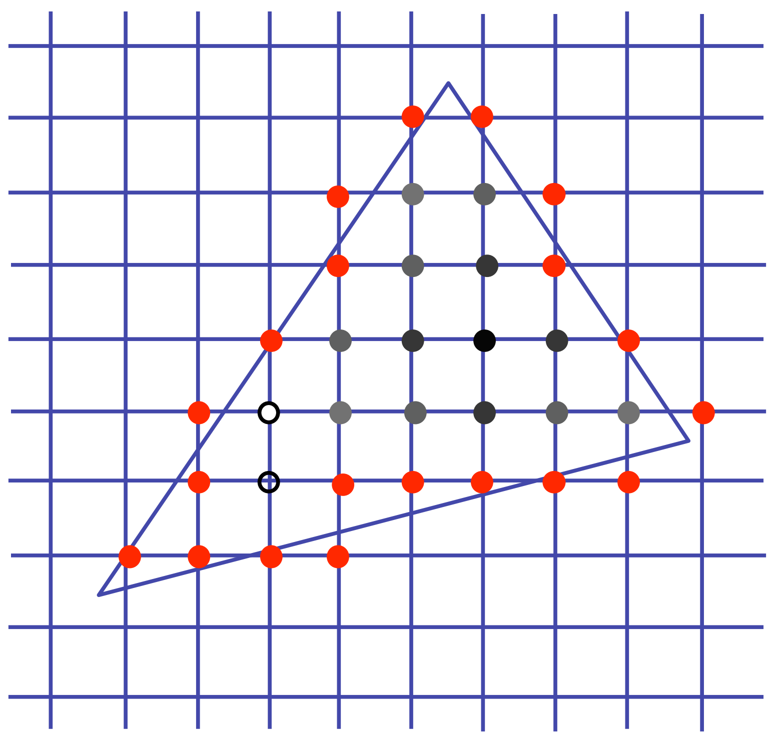
- simple algorithm
  - draw edges of polygon
  - use flood-fill to draw interior





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, $x, y-1, C$);

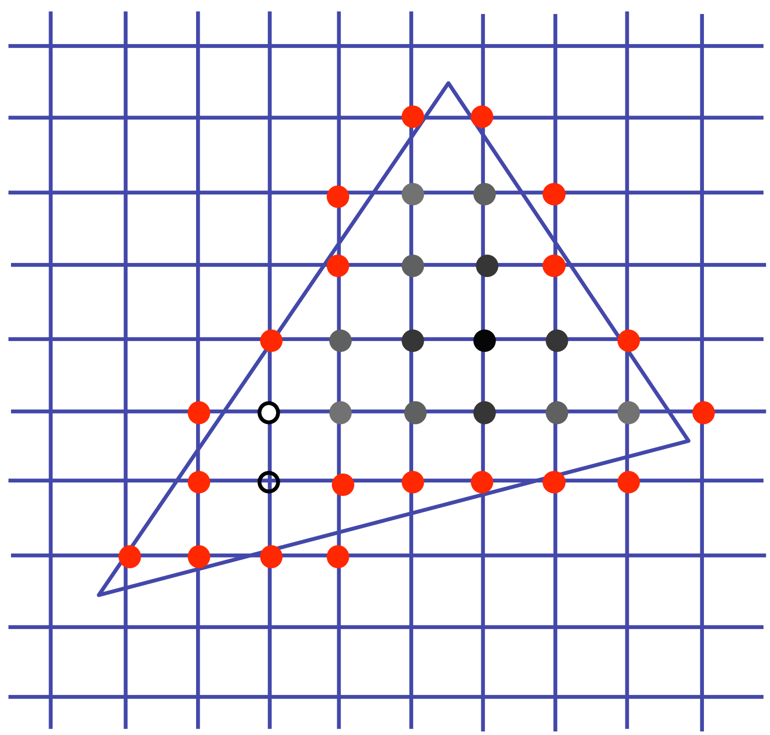
FloodFill(P, $x-1, y, C$);

end ;

- drawbacks?

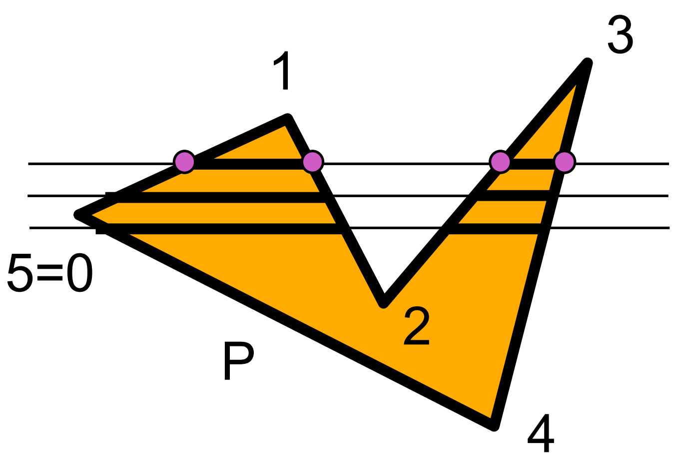
Flood Fill Drawbacks

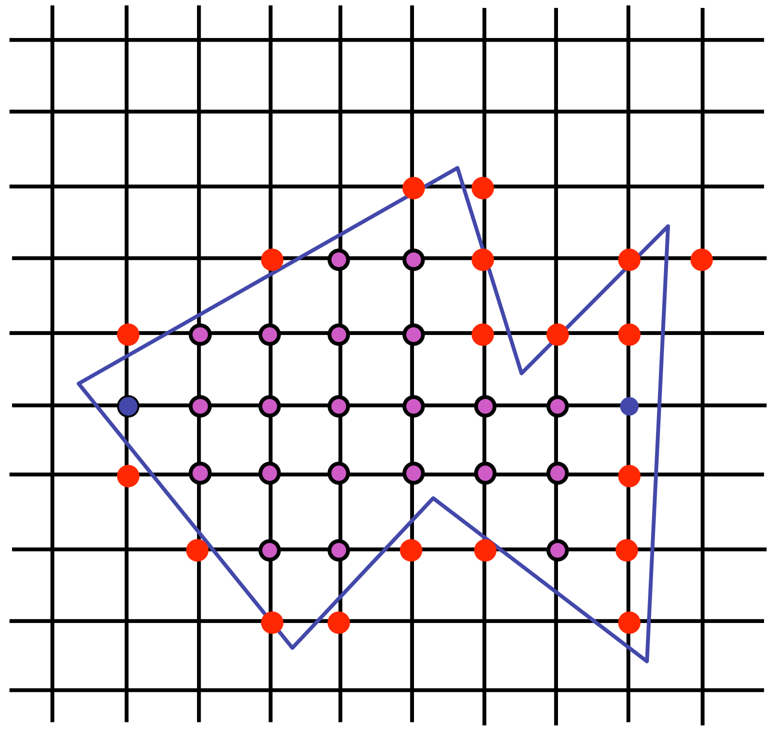
- pixels visited up to 4 times to check if already set
- need per-pixel flag indicating if set already
  - must clear for every polygon!



Scanline Algorithms

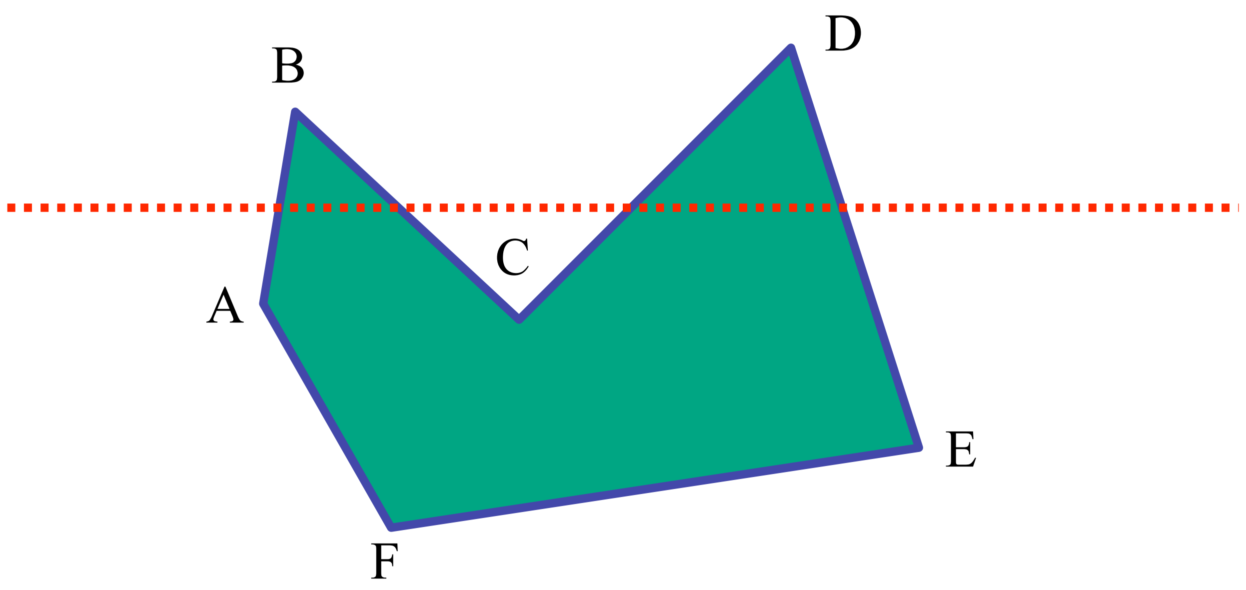
- scanline: a line of pixels in an image
  - set pixels inside polygon boundary along horizontal lines one pixel apart vertically





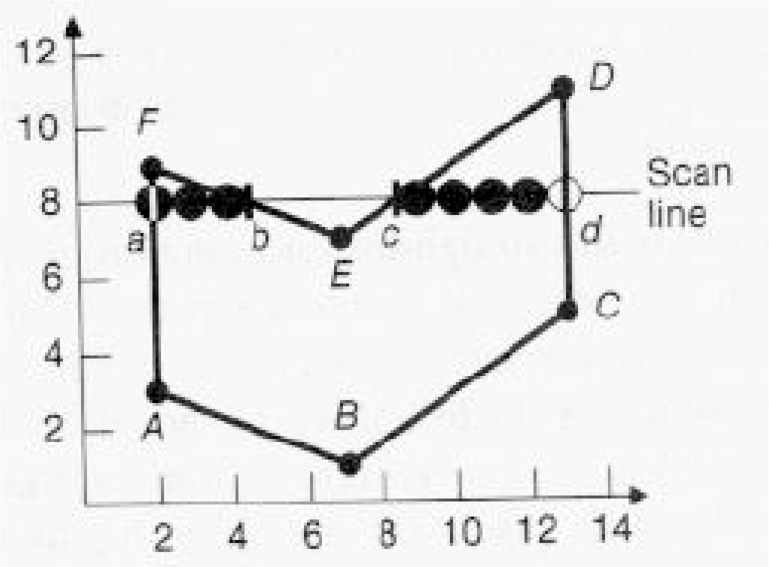
General Polygon Rasterization

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



General Polygon Rasterization

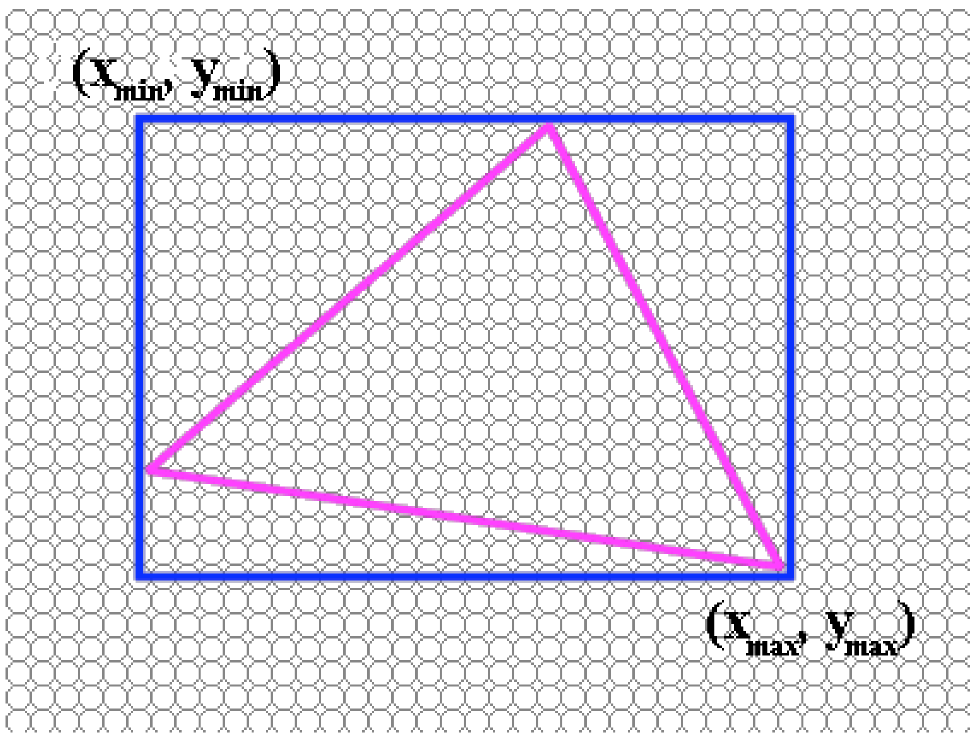
- idea: use a parity test




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

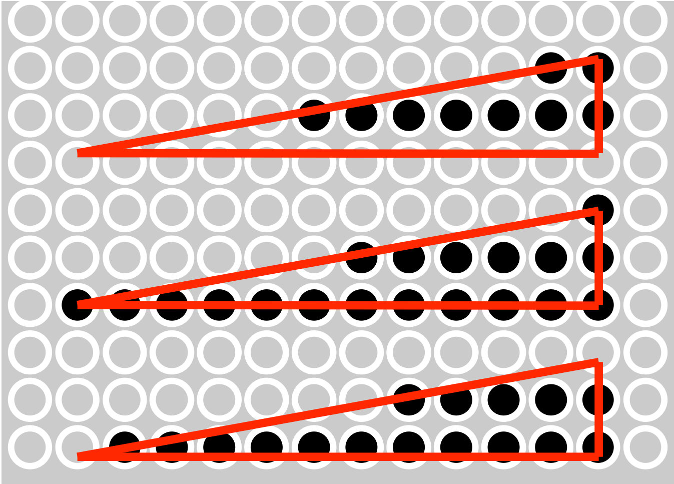
Making It Fast: Bounding Box

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

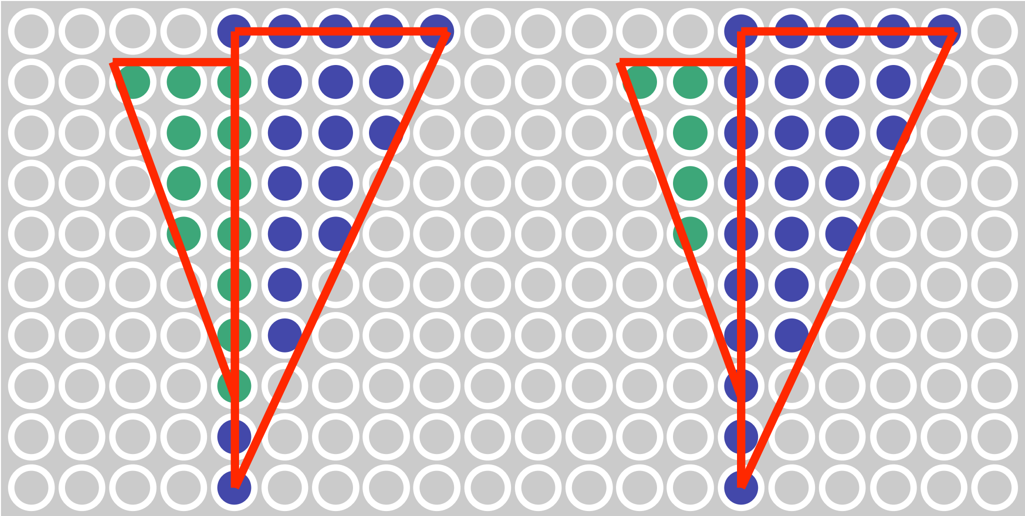


Triangle Rasterization Issues

- moving slivers



- shared edge ordering



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

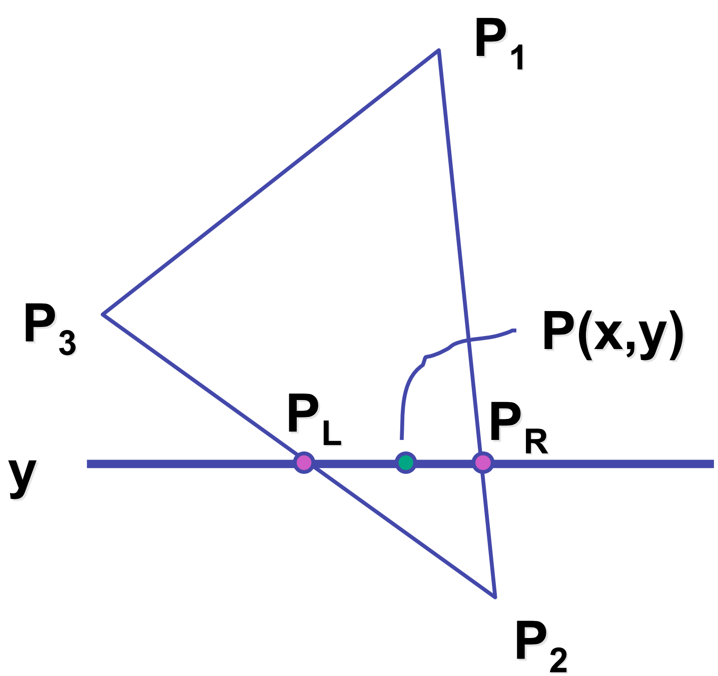
Interpolation

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

Bilinear Interpolation

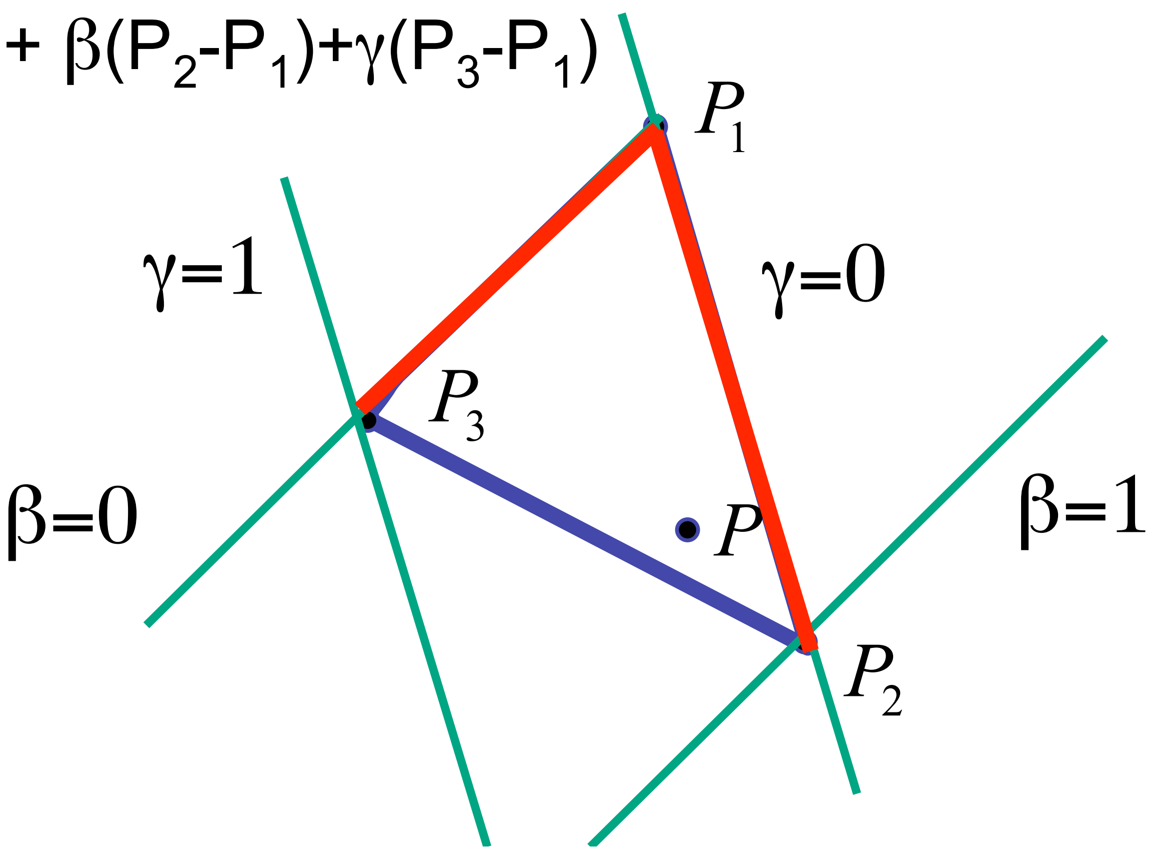
- interpolate quantity along L and R edges, as a function of y
  - then interpolate quantity as a function of x



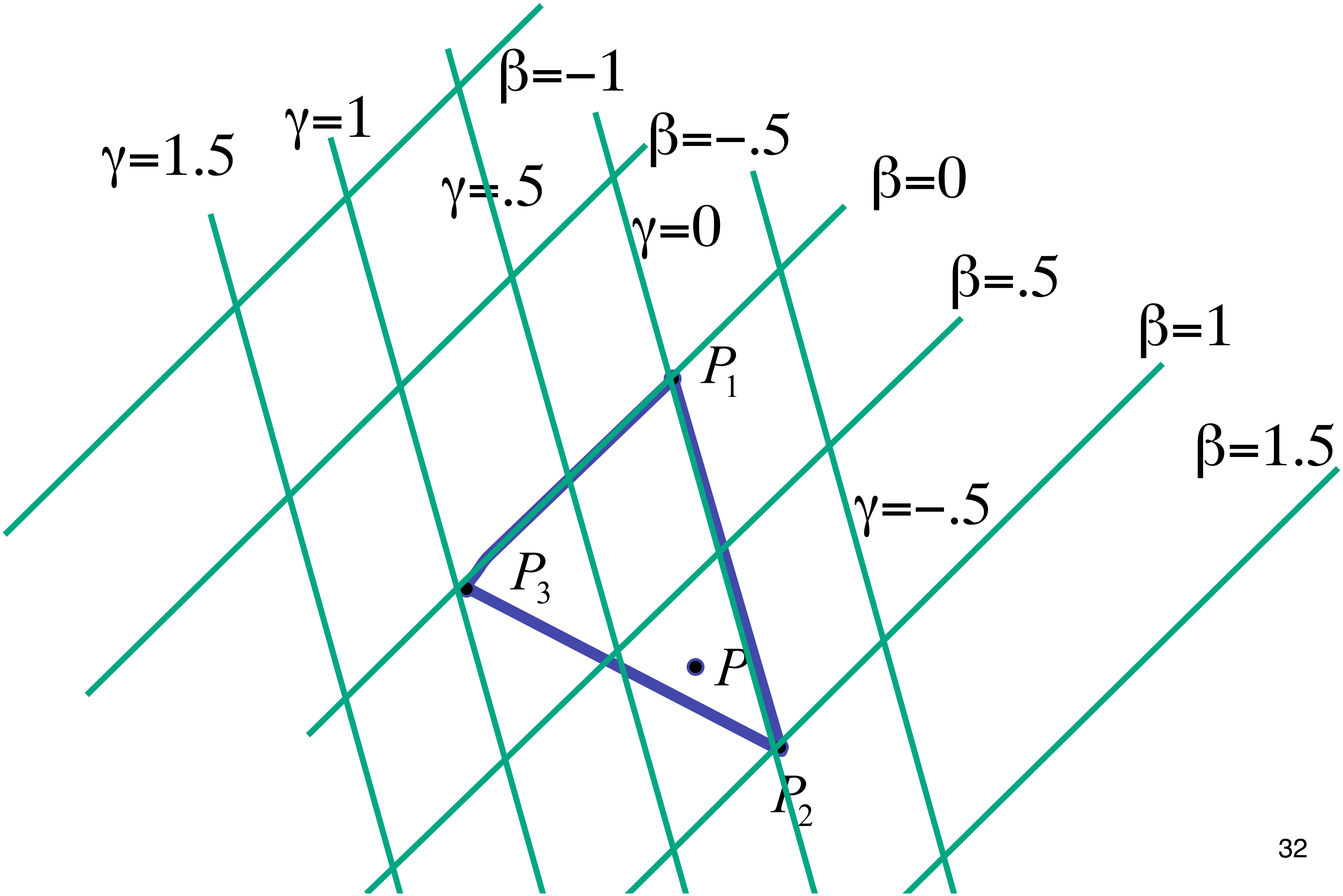
Barycentric Coordinates

- 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



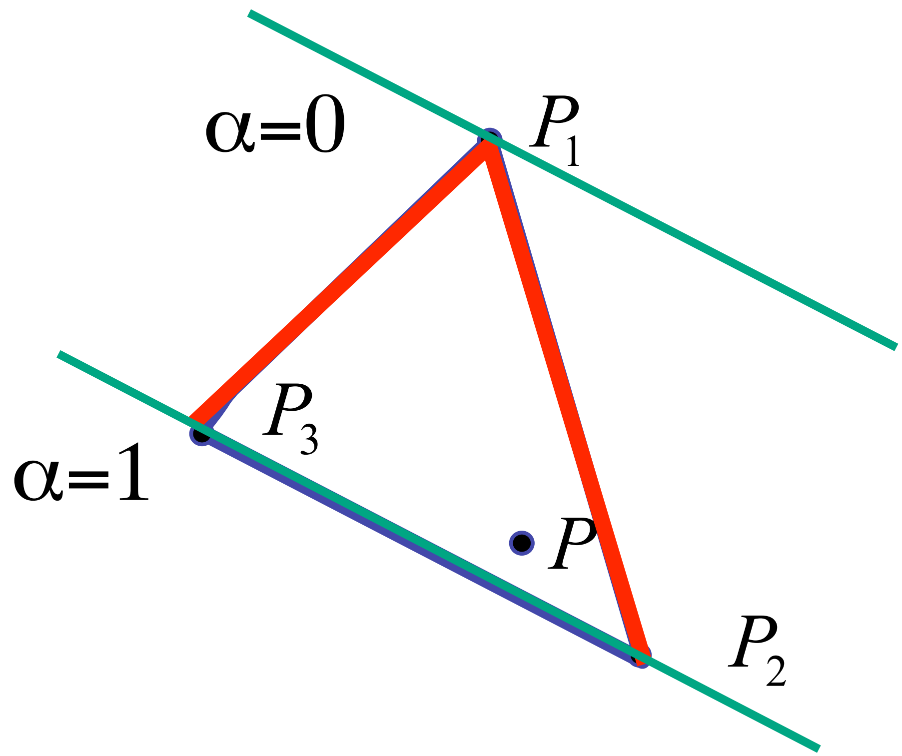
Barycentric Coordinates

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

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

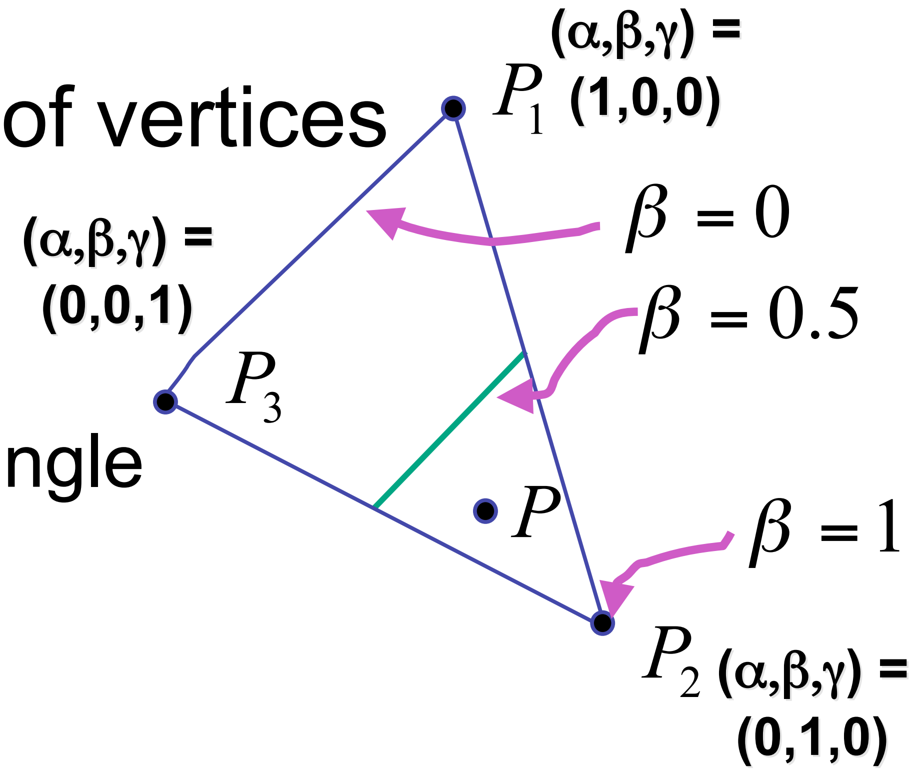
$$P = (1-\beta-\gamma)P_1 + \beta P_2+\gamma P_3$$

$$P = \alpha P_1 + \beta P_2+\gamma P_3$$



Using Barycentric Coordinates

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



$$P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3$$

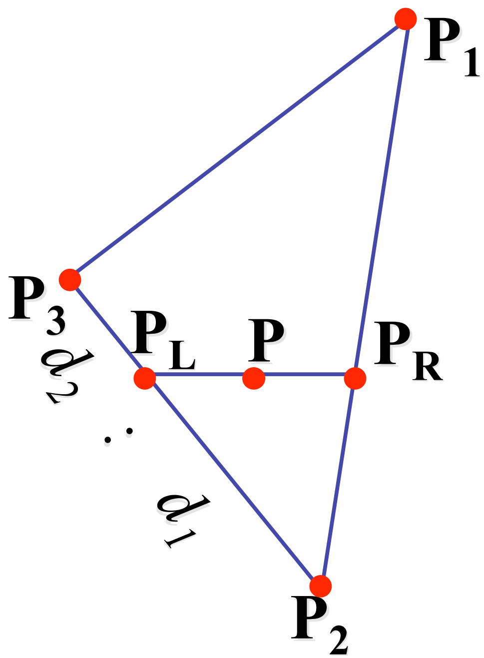
$$\alpha + \beta + \gamma = 1$$

$0 \le \alpha, \beta, \gamma \le 1$ for points inside triangle

"convex combination of points"

Deriving Barycentric From Bilinear

- from bilinear interpolation of point P on scanline



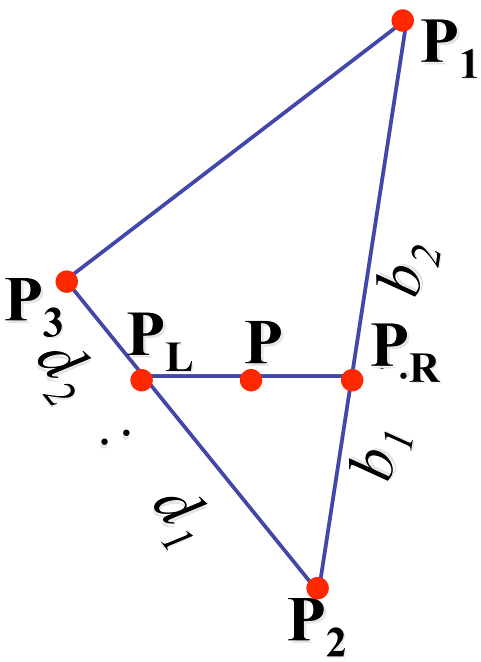
$$P_L = P_2 + \frac{d_1}{d_1 + d_2}(P_3 - P_2)$$

$$= (1 - \frac{d_1}{d_1 + d_2})P_2 + \frac{d_1}{d_1 + d_2}P_3 =$$

$$= \frac{d_2}{d_1 + d_2}P_2 + \frac{d_1}{d_1 + d_2}P_3$$

Deriving Barycentric From Bilineaer

- similarly



$$P_R = P_2 + \frac{b_1}{b_1 + b_2}(P_1 - P_2)$$

$$= (1 - \frac{b_1}{b_1 + b_2})P_2 + \frac{b_1}{b_1 + b_2}P_1 =$$

$$= \frac{b_2}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1$$

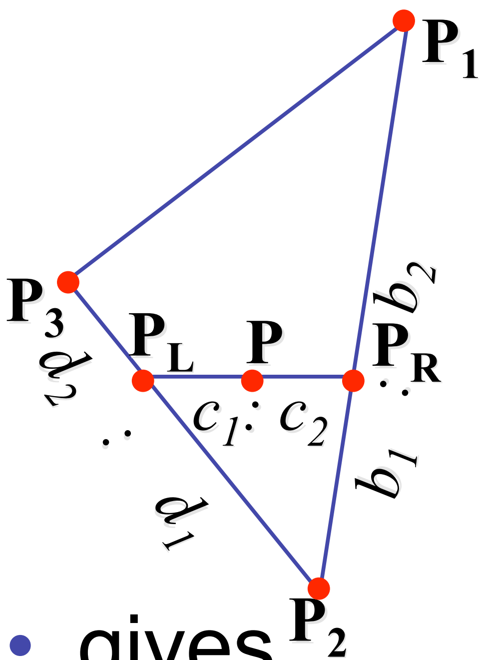
Deriving Barycentric From Bilinear

- 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

$$\alpha = \frac{c_1}{c_1 + c_2} \frac{b_1}{b_1 + b_2}$$

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

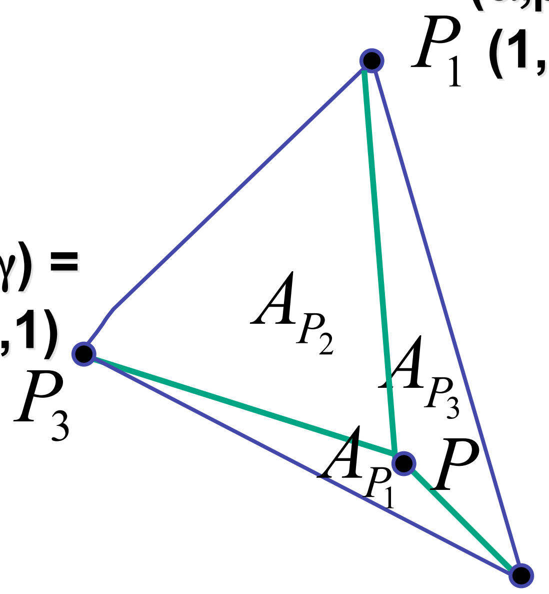
$$\gamma = \frac{c_2}{c_1 + c_2} \frac{d_1}{d_1 + d_2}$$

- can verify barycentric properties

$$\alpha + \beta + \gamma = 1, \qquad 0 \leq \alpha, \beta, \gamma \leq 1$$

Computing Barycentric Coordinates

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



$A = A_{P1} + A_{P2} + A_{P3}$

$\alpha = A_{P1} / A$

$\beta = A_{P2} / A$

$\gamma = A_{P3} / A$