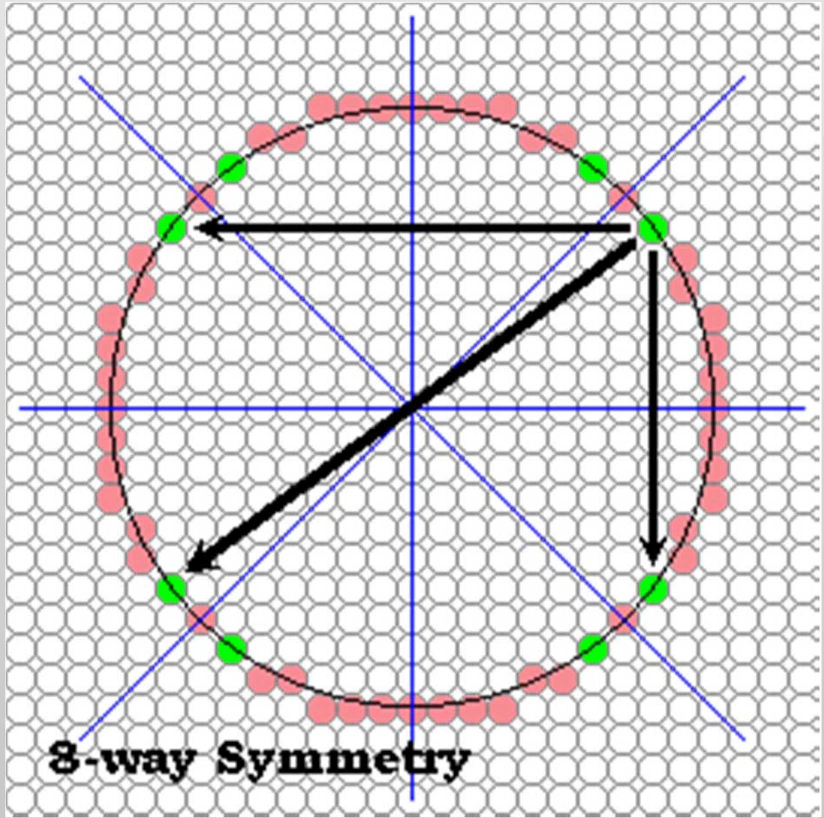
Computer Graphics

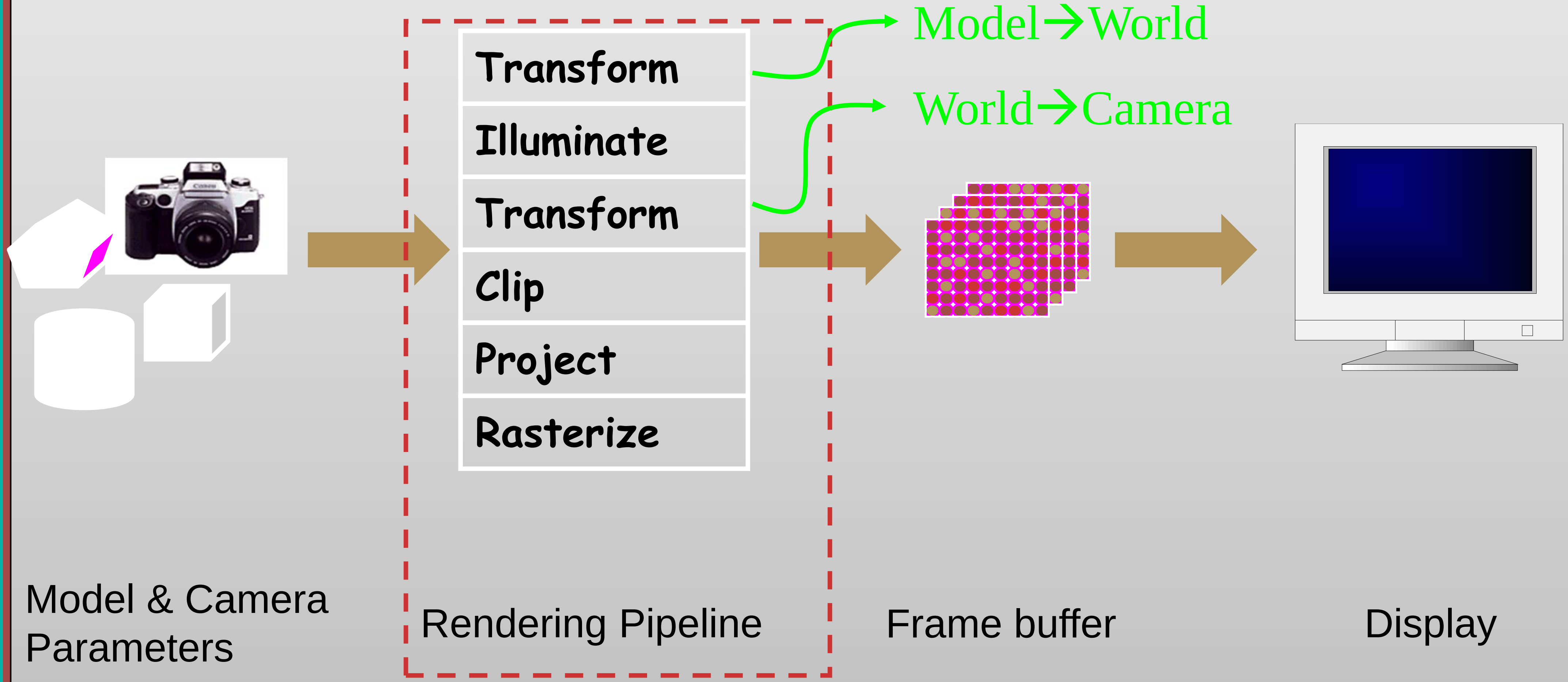


CH6
Rasterization

Instructor: Dr. MAO Aihua

ahmao@scut.edu.cn

Rendering 3D Scenes



Review: Pipeline (for direct illumination)

3D Geometric Primitives

Modeling Transformation	Transform into 3D world coordinate system
Lighting	Illuminate according to lighting and reflectance
Viewing Transformation	Transform into 3D camera coordinate system
Projection Transformation	Transform into 2D screen coordinate system
Clipping	Clip primitives outside camera’s view
Scan Conversion	Draw pixels (includes texturing, hidden surface, ...)

Image

Raster Graphics Algorithms

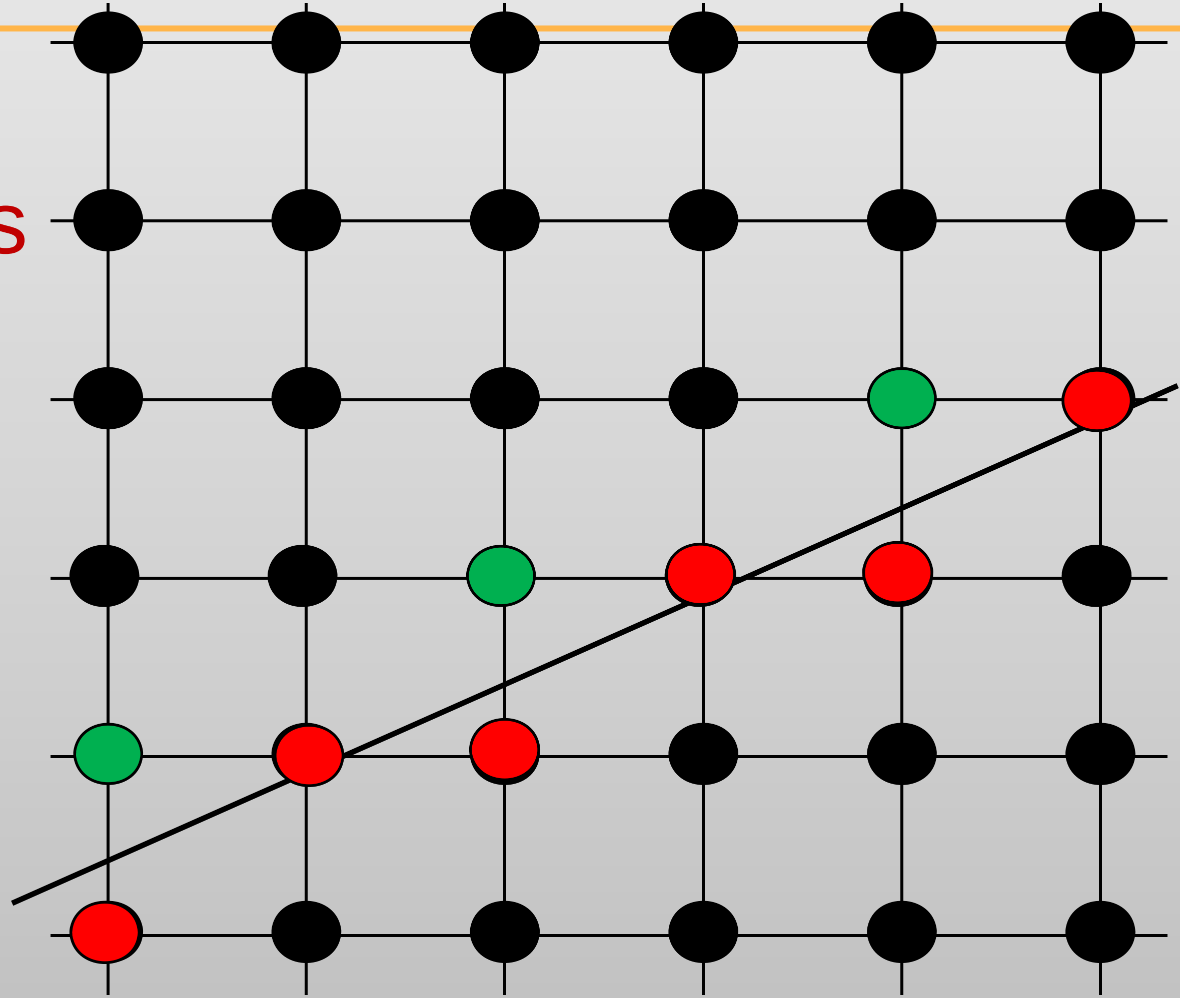
Scan conversion of a primitive (line, circle, polygon and so on)

- Find a finite point set which approximates the primitive optimally
- Converse a continuous primitive to the set of discrete pixels
- It is a sampling problem

- Also named as **scan conversion**

Example

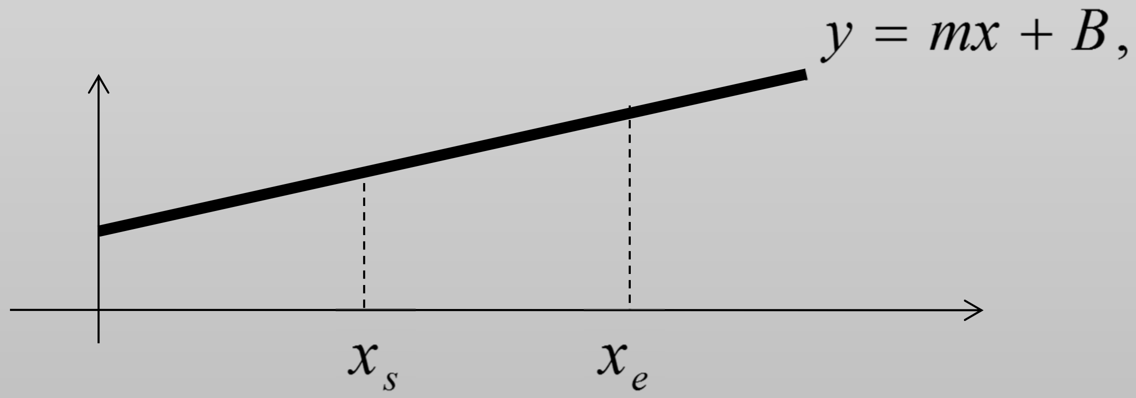
- Optimal
  - Approx. error is minimal



Line equation

$$y = mx + B, \quad x_s \le x \le x_e, |m| \le 1,$$

- $x_s = a, x_e = a + n$ are integers, m is the slope of the line



A naive method

- Let x take the following values

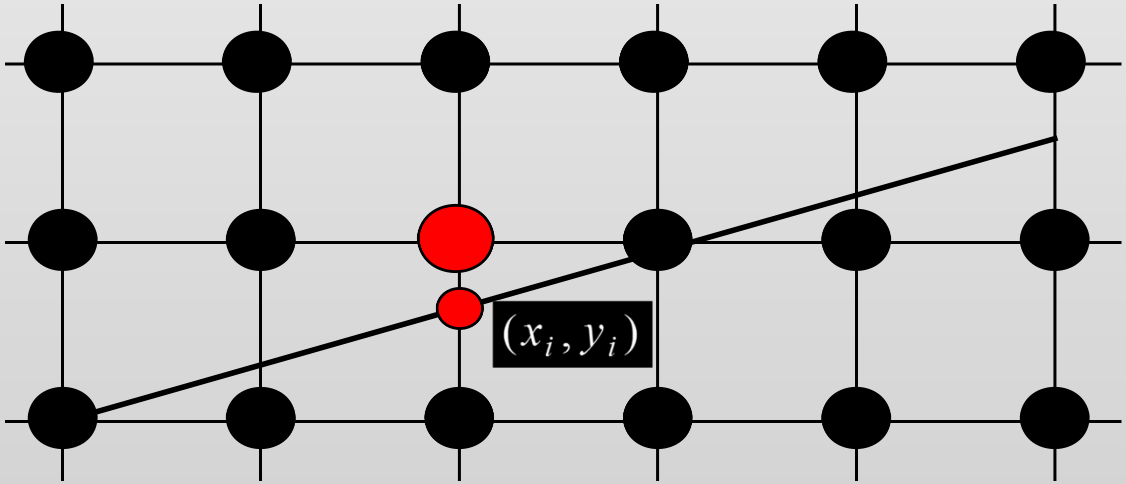
$$a, a+1, a+2, \cdots, a+n$$

- Denote the corresponding points on the line as y_i

$$x_i = a + i, \quad y_i = mx_i + B.$$

- We can use ($x = x_i$, $round(y_i)$) as the approximation of (x_i, y_i)

where $round(y_i)$ is the integer nearest to y_i

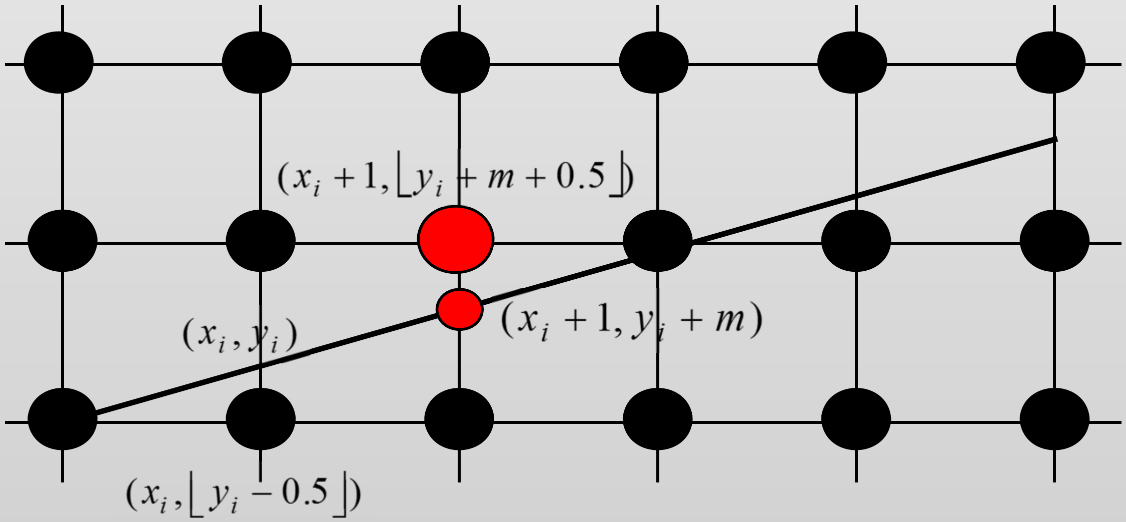


- In the above example, we select the big red circle

DDA (digital differential analyzer)

- Drawback of the naïve method: multiplication is required for evaluating $y_i = mx_i + B$
- Actually y_{i+1} can be computed from y_i

$$
\begin{aligned}
x_{i+1} &= x_i + 1 \\
y_{i+1} &= mx_{i+1} + B \\
&= m(x_i + 1) + B \\
&= m + (mx_i + B) \\
&= y_i + m.
\end{aligned}
$$



- Illustration of the aforementioned analysis

Rasterizing Polygons

In interactive graphics, polygons rule the world

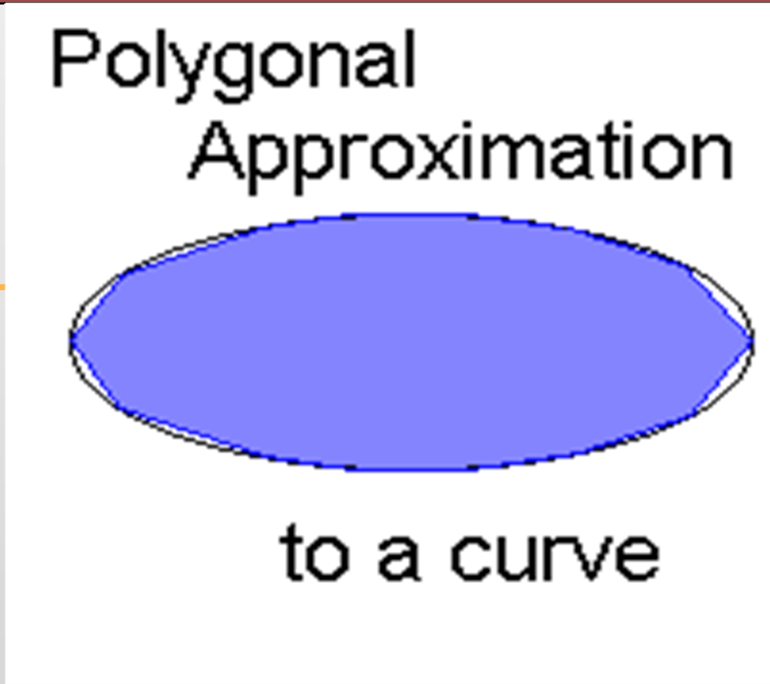
Two main reasons:

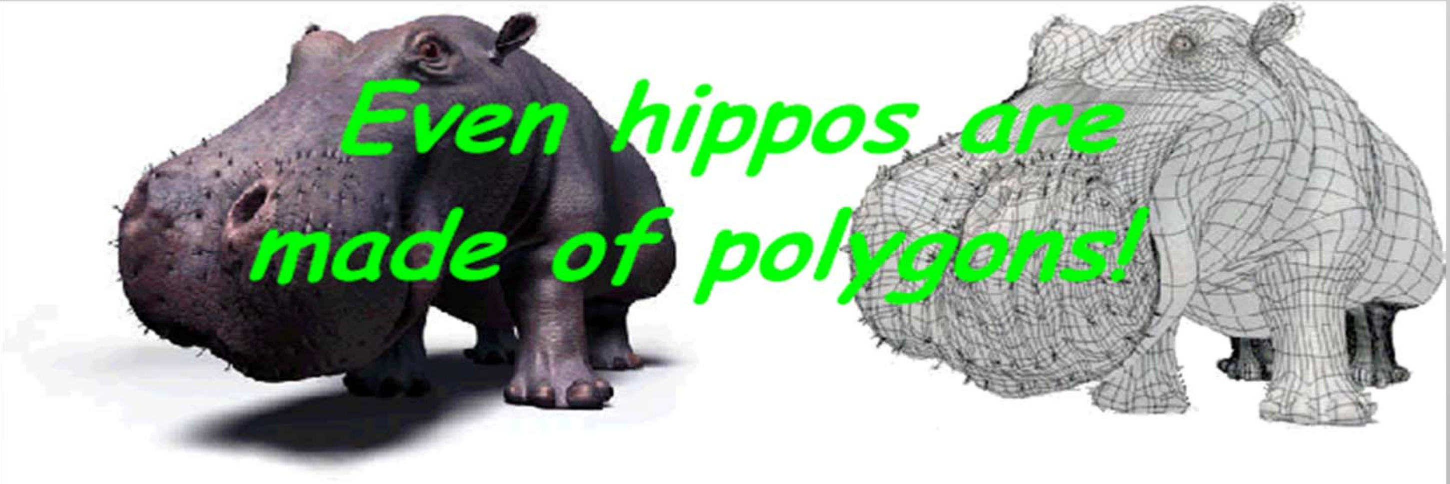
- Lowest common denominator for surfaces
  - *Can represent any surface with arbitrary accuracy*
  - *Splines, mathematical functions, volumetric isosurfaces…*
- Mathematical simplicity lends itself to simple, regular rendering algorithms

Rasterizing Polygons

Triangle is the minimal unit of a polygon

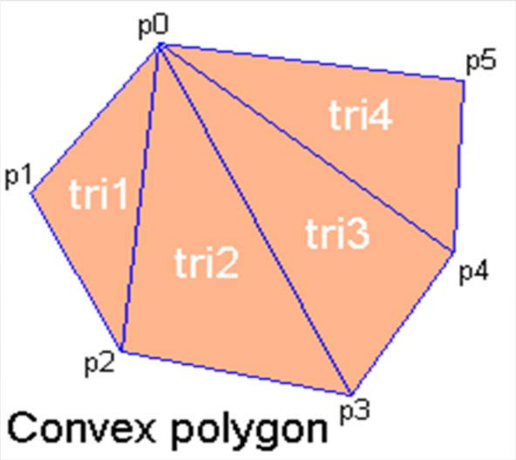
- All polygons can be broken up into triangles
- Triangles are guaranteed to be:
  - *Planar*
  - *Convex*



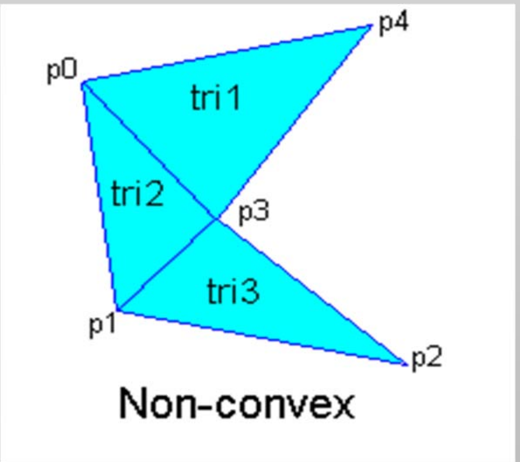


Triangulation

Convex polygons easily triangulated



Concave polygons present a challenge



Triangulation

In contrast, a triangle is always convex: no matter how a triangle is oriented, it only has one span per scan line

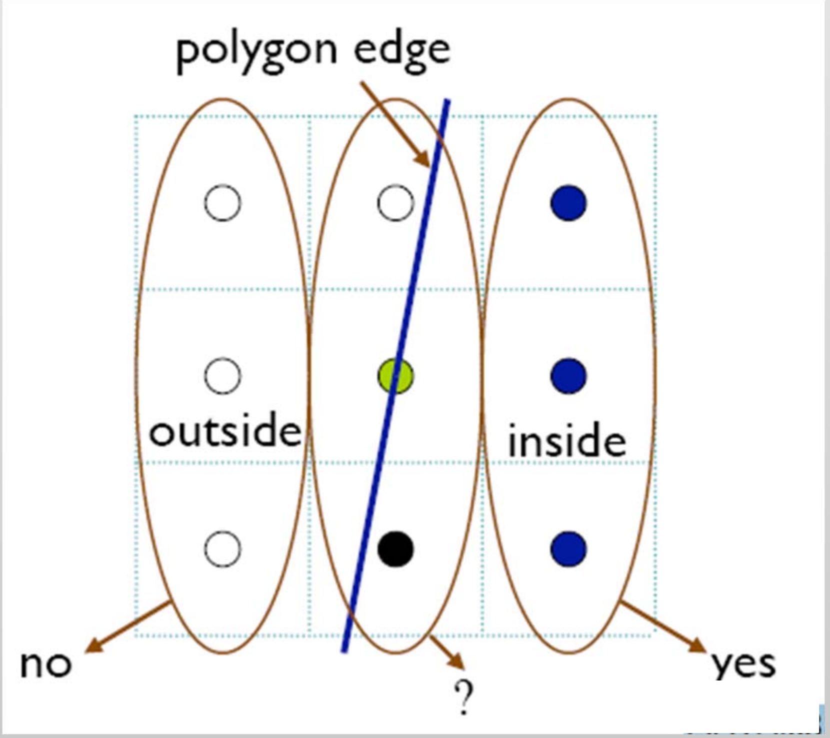
Rasterizing Triangles

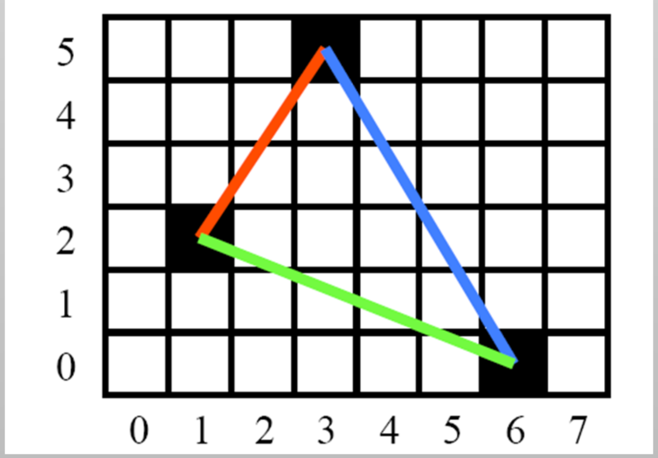
Which pixel to set?

What color to set each pixel to?

How would you

rasterize a triangle?





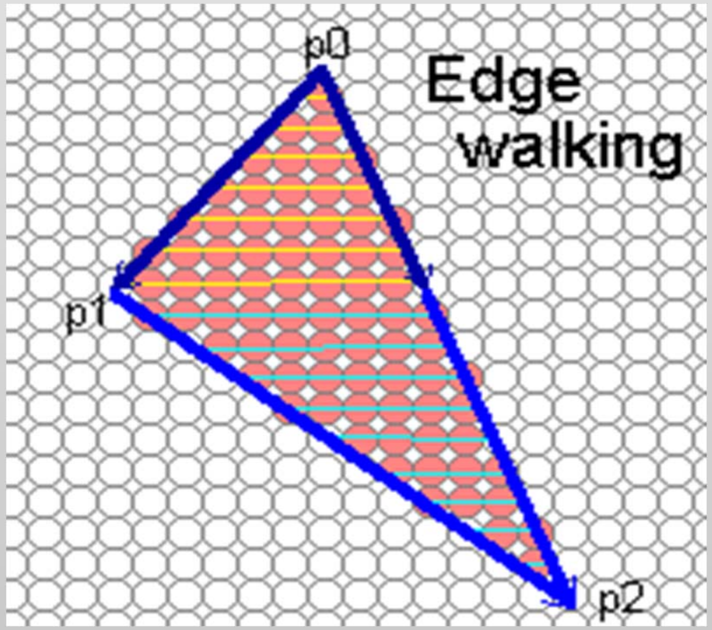
Rasterizing Triangles

- Interactive graphics hardware commonly uses edge walking or edge equation techniques for rasterizing triangles

Edge Walking

Basic idea:

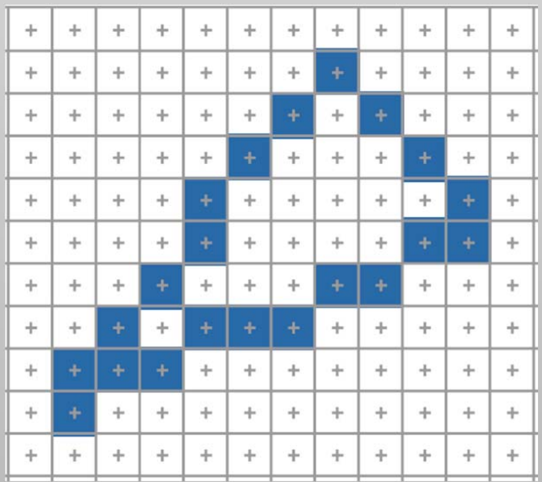
- Draw edges vertically，namely
- Fill in horizontal spans for each scanline

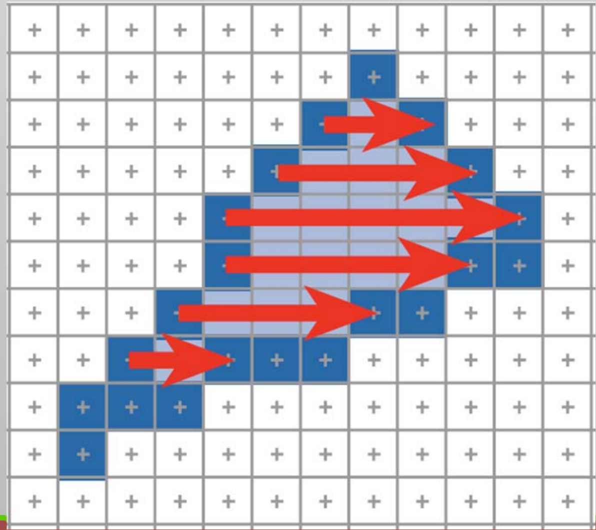


Edge Walking: Notes

Order three triangle vertices in x and y

- scan top to bottom in scan-line order
- “walk” edges: use edge slope to update coordinates incrementally
- on each scan-line, scan left to right (horizontal span), setting pixels
- stop when bottom vertex or edge is reached





Edge Walking: Codes

void edge_walking(vertices T[3])

{

for each edge pair of T {

initialize x_L, x_R;

compute dx_L/dy_L and dx_R/dy_R;

for scanline at y {

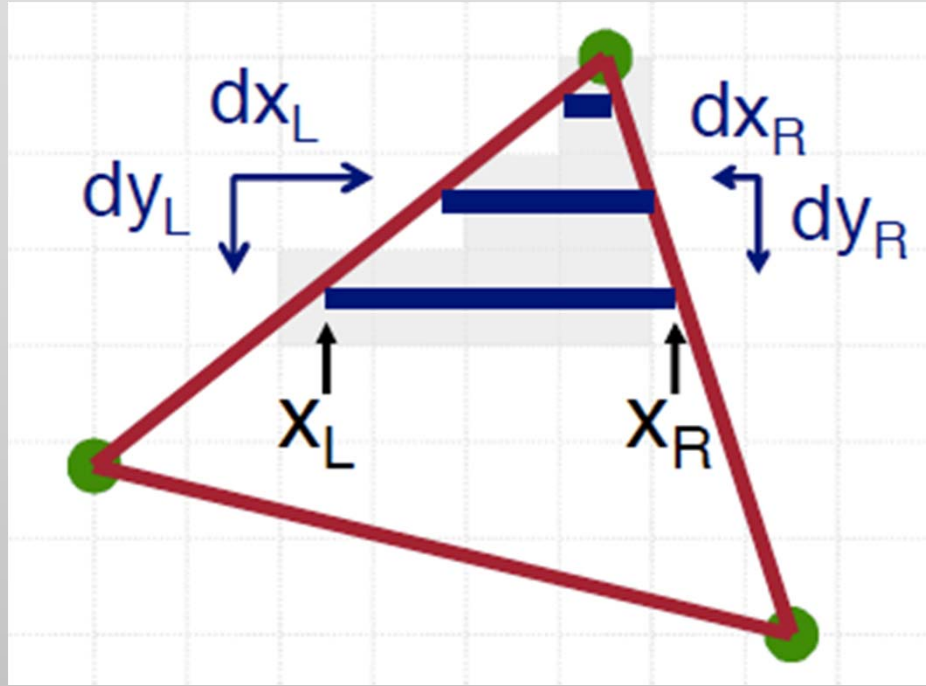
for (int x = xL; x <= xR; x++) {

set_pixel(x, y);

} }

x_L += dx_L/dy_L;

x_R += dx_R/dy_R;



Edge Walking: Disadvantages

Advantages:

- simple

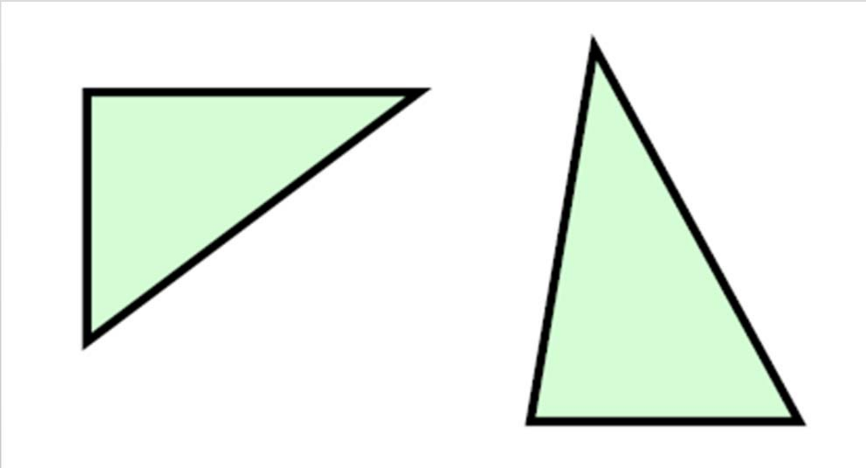
Disadvantages:

- very serial (one pixel at a time) can’t parallelize
- inner loop bottleneck if lots of computation per pixel

Edge Walking: Disadvantages

Special cases:

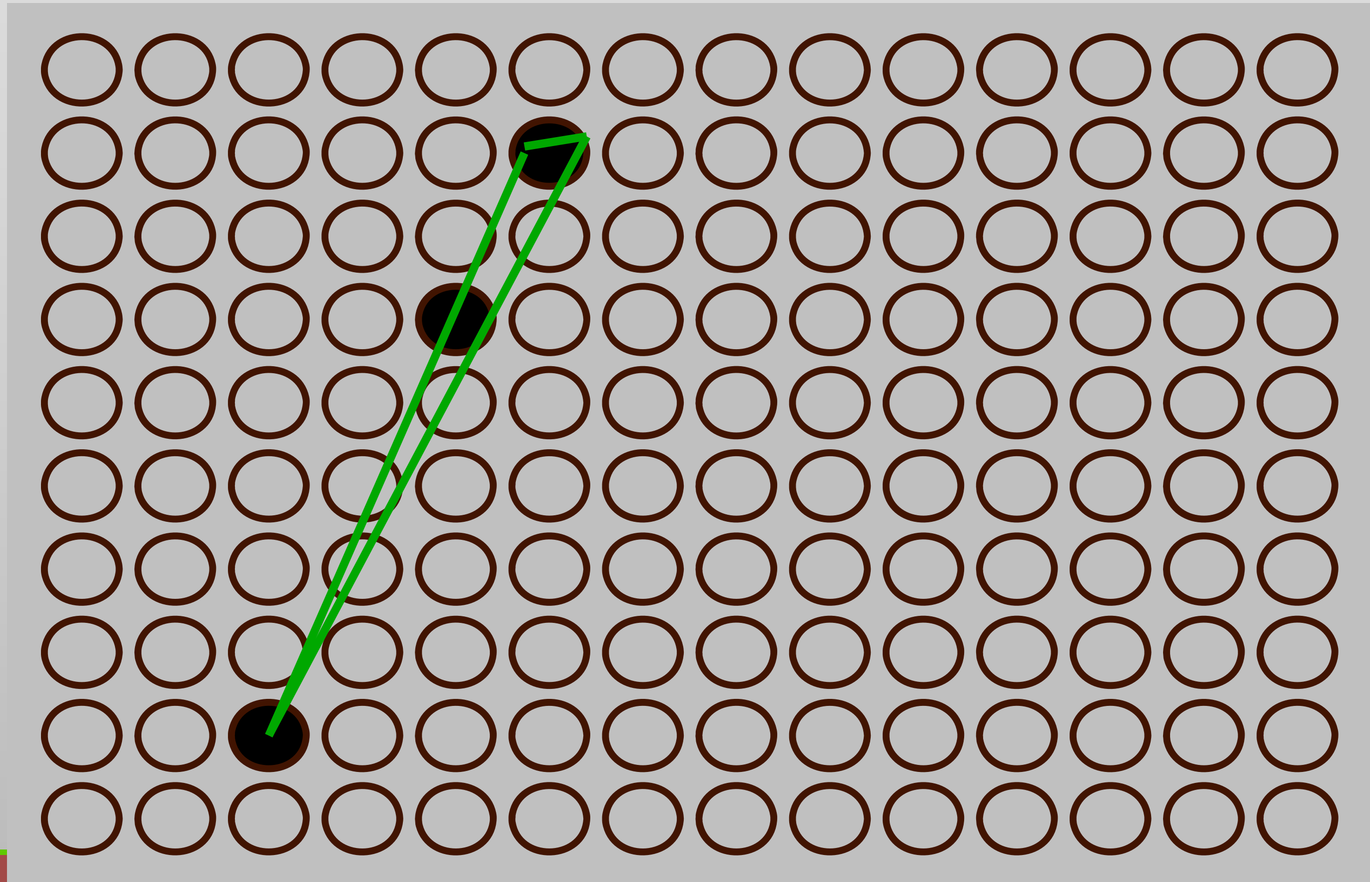
- horizontal edges: computing intersection causes divide by 0



Edge Walking: Disadvantages

Special cases:

Sliver: not even a single pixel wide



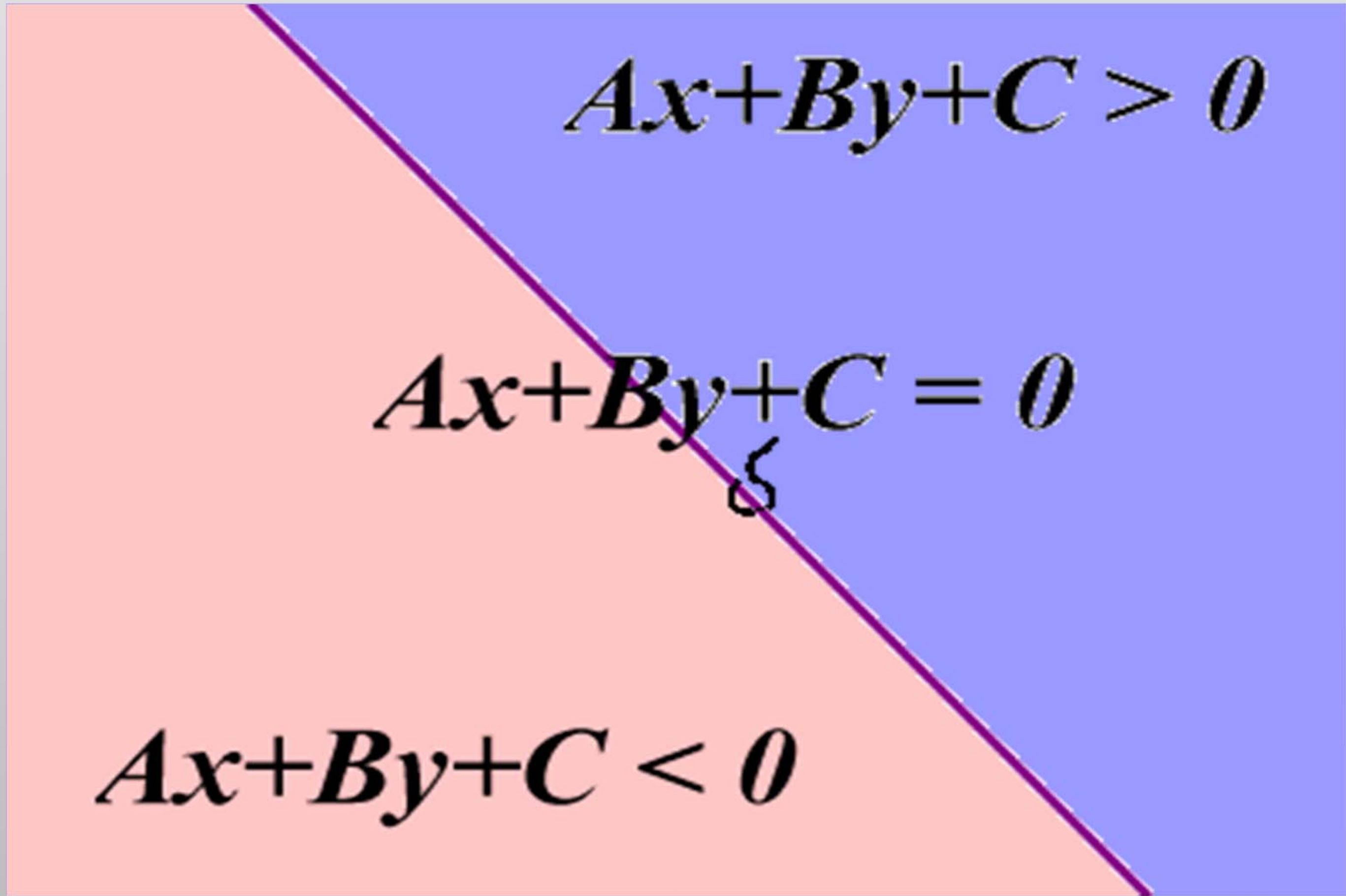
Edge Equations

An edge equation is simply the equation of the line defining that edge, can compute from vertices

- Q: *What is the implicit equation of a line?*
- A: $Ax + By + C = 0$
- Q: *Given a point (x,y), what does plugging x & y into this equation tell us?*
- A: Whether the point is:
  - *On the line:* $Ax + By + C = 0$
  - *"Above" the line:* $Ax + By + C > 0$
  - *"Below" the line:* $Ax + By + C < 0$

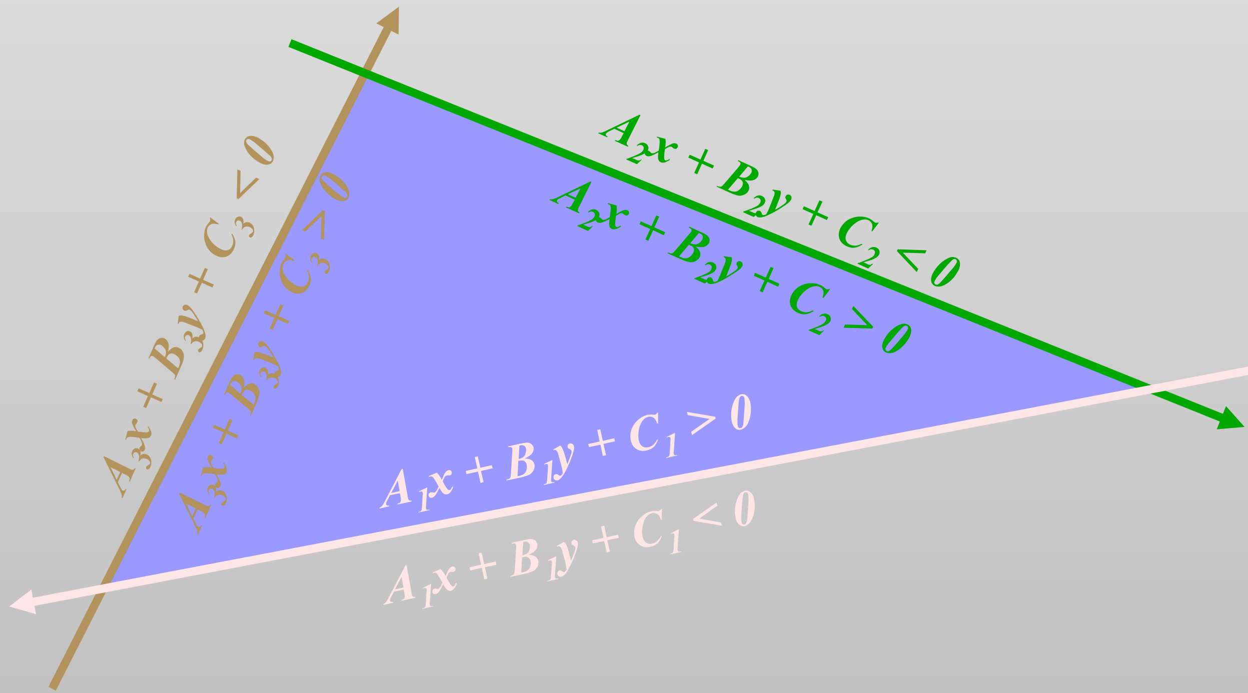
Edge Equations

Edge equations thus define two half-spaces:



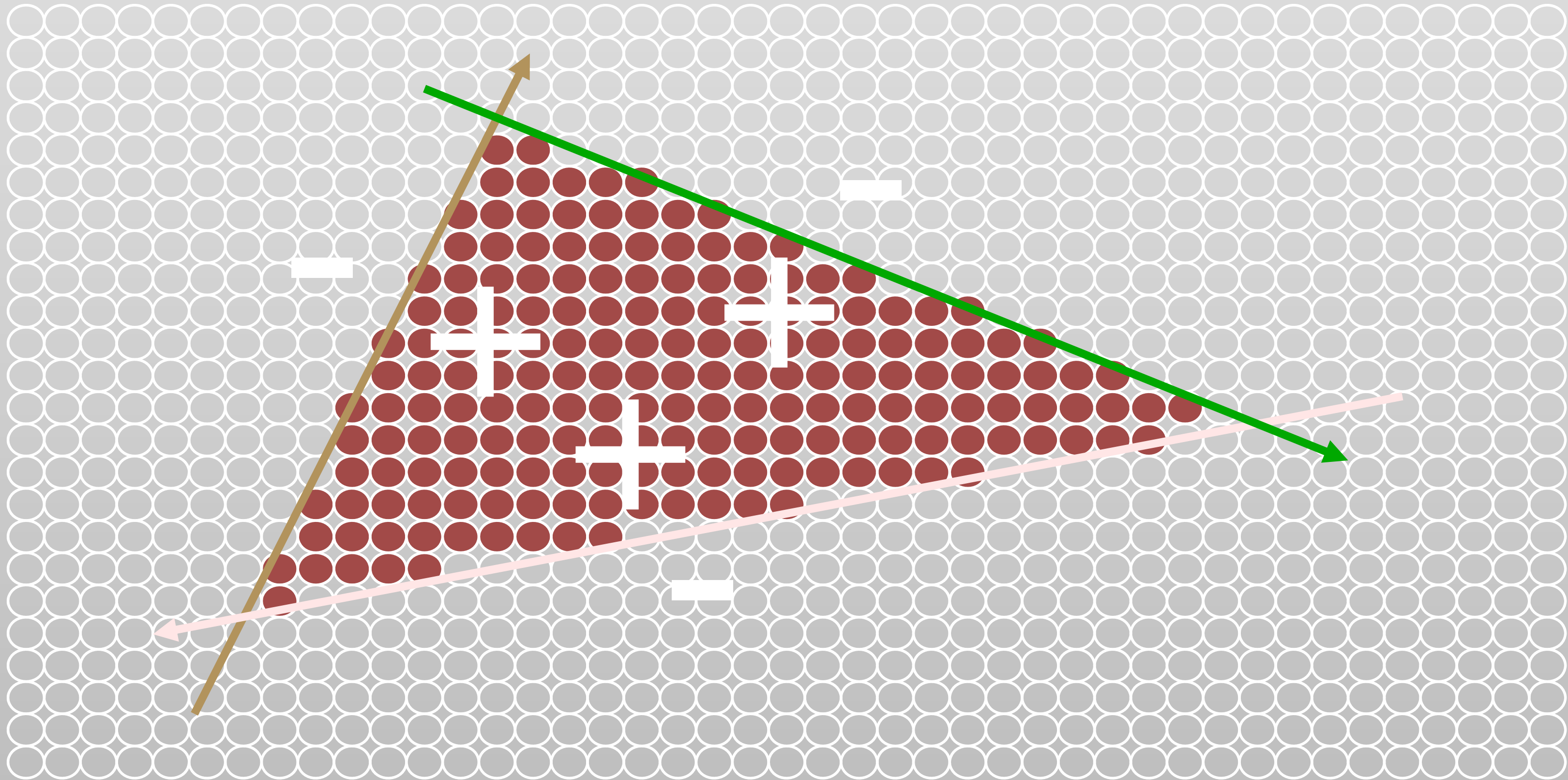
Edge Equations

And a triangle can be defined as the intersection of three positive half-spaces:



Edge Equations

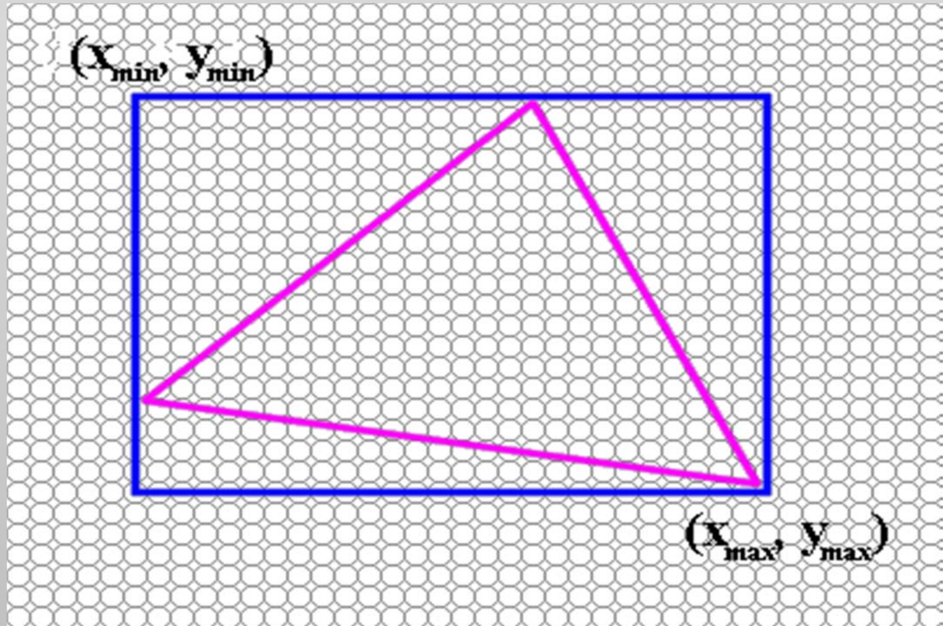
So…simply turn on those pixels for which all edge equations evaluate to > 0:



Using Edge Equations

Right now the test pixels are full, Can we reduce #pixels tested?

compute min, max bounding box



Computing Edge Equations

Want to calculate A, B, C for each edge from (x_0, y_0) and (x_1, y_1)

Treat it as a linear system:

$$Ax_0 + By_0 + C = 0$$

$$Ax_1 + By_1 + C = 0$$

Notice: two equations, three unknowns

What can we solve?

Goal: solve for A & B in terms of C

Computing Edge Equations

Set up the linear system:

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix}\begin{bmatrix} A \\ B \end{bmatrix} = -C\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Multiply both sides by matrix inverse:

$$\begin{bmatrix} A \\ B \end{bmatrix} = \frac{-C}{\begin{vmatrix} x_0, & y_0 \\ x_1, & y_1 \end{vmatrix}}\begin{bmatrix} y_1, - y_0 \\ - x_1, \ x_0 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$= \frac{-C}{x_0 y_1 - x_1 y_0}\begin{bmatrix} y_1 - y_0 \\ x_0 - x_1 \end{bmatrix}$$

Let $C = -(x_0 y_1 - x_1 y_0)$

Then $\boldsymbol{A = y_0 - y_1}$ and

$\boldsymbol{B = x_0 - x_1}$

Edge Equations

- So…we can find edge equation from two vertices.
- Given three corners P_0, P_1, P_2 of a triangle, what are our three edges?
- How do we make sure the half-spaces defined by the edge equations all share the same sign on the interior of the triangle?
  - A: Be consistent, clockwise (Ex: $[P_0\ P_1]$, $[P_1\ P_2]$, $[P_2\ P_0]$)
- How do we make sure that sign is positive?
  - A: Test, and flip if needed ($A = -A$, $B = -B$, $C = -C$)

Edge Equations: Code

Basic structure of code:

- Setup: compute edge equations, bounding box
- (Outer loop) For each scanline in bounding box...
- (Inner loop) …check each pixel on scanline, evaluating edge equations and drawing the pixel if all three are positive

Edge Equations: Code

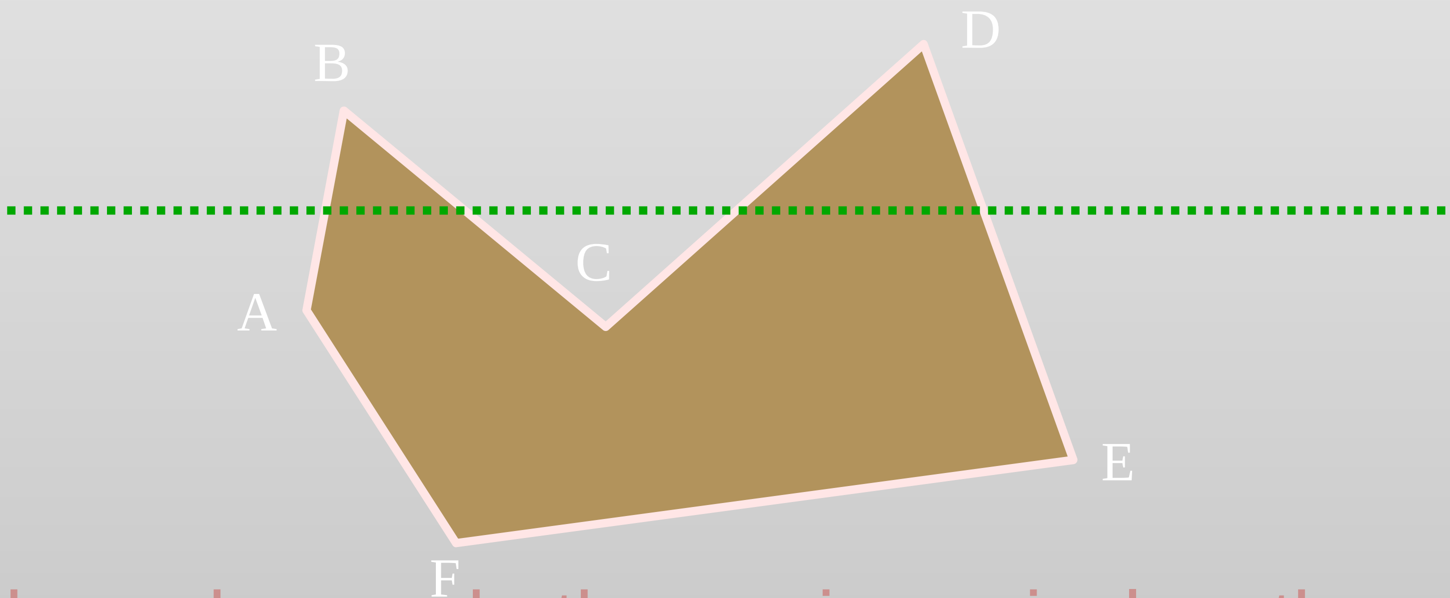
```
findBoundingBox(&xmin, &xmax, &ymin, &ymax);
setupEdges (&a0,&b0,&c0,&a1,&b1,&c1,&a2,&b2,&c2);
/* Optimize this: */
```

General Polygon Rasterization

- Now that we can rasterize triangles, what about general polygons?
- We'll take an edge-walking approach

General Polygon Rasterization

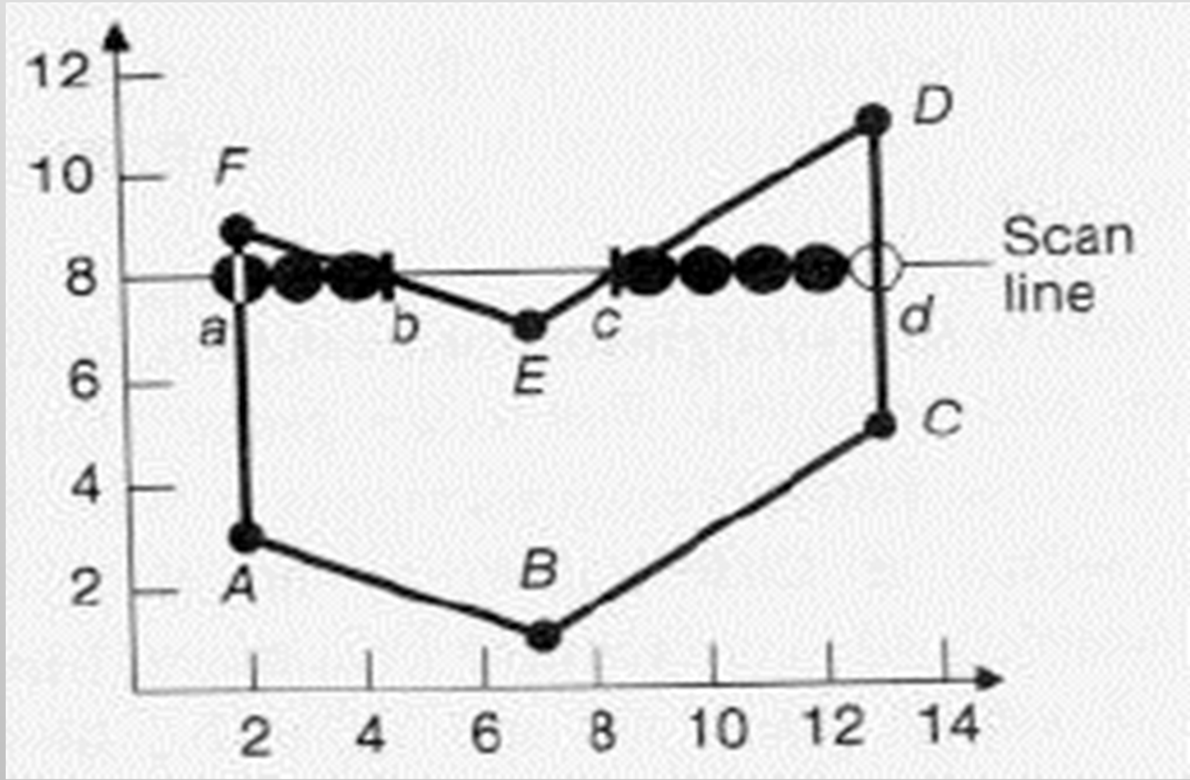
Consider the following polygon:



How do we know whether a given pixel on the scanline is inside or outside the polygon?

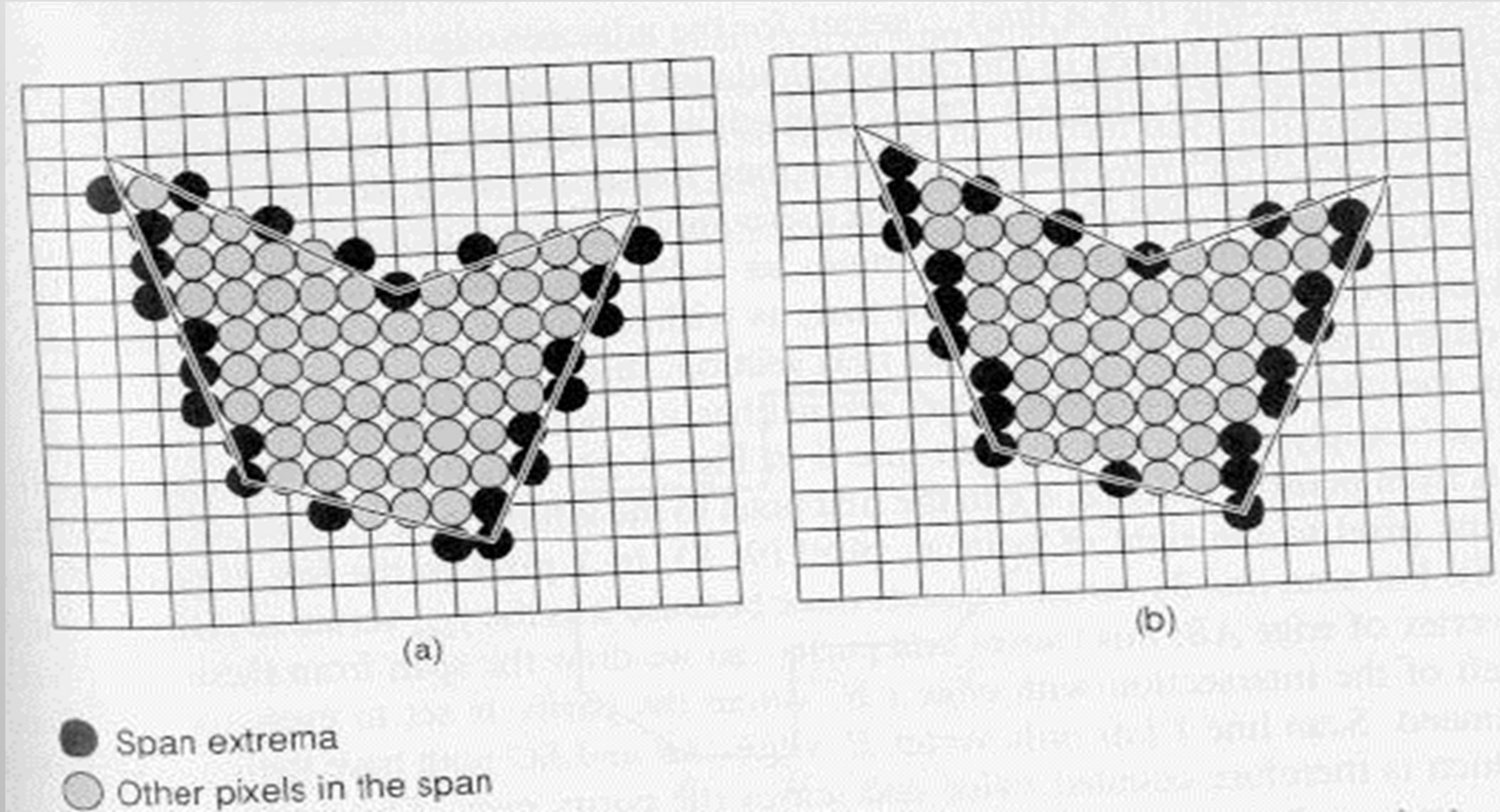
Polygon Rasterization

Inside-Outside Points



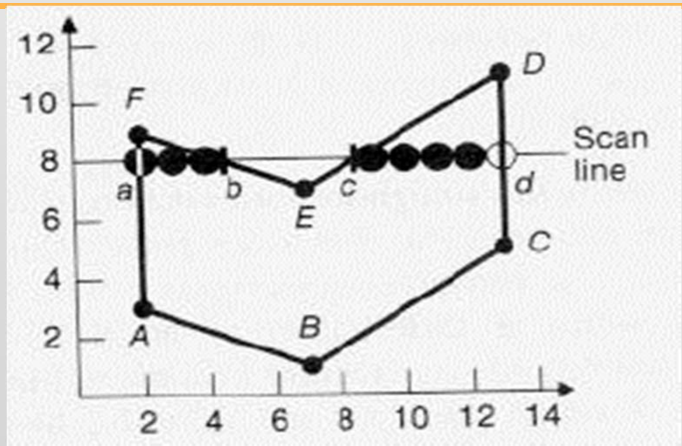
Polygon Rasterization

Inside-Outside Points



General Polygon Rasterization

Basic idea: use a parity test



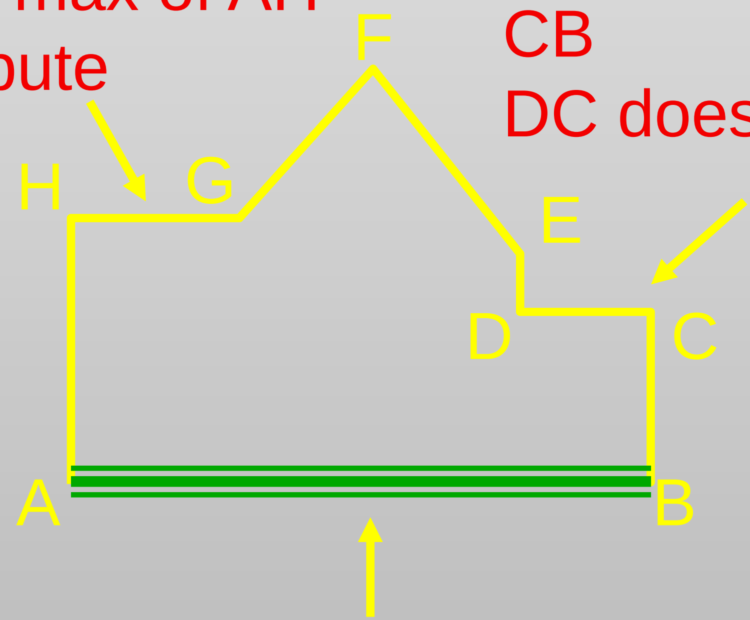

```
for each scanline
    edgeCnt = 0;
    for each pixel on scanline (l to r)
        if (oldpixel->newpixel crosses edge)
            edgeCnt ++;
        // draw the pixel if edgeCnt odd= inner point
        if (edgeCnt % 2)
            setPixel(pixel);
```

Polygon Rasterization

- Horizontal lines do not contribute to parity count
- Y_{min} endpoints do contribute to parity count
- Y_{max} endpoints do not contribute to parity count

Not drawn because H is max of AH
And HG does not contribute

Not drawn because C is max of CB
DC doesn’t contribute



Drawn Y>Ymin ,because A is min of AH. AB does not contribute

Active Edge Table

Algorithm: scanline from bottom to top…

- Sort all edges by their minimum y coordinate
- Starting at bottom, add edges with Y_{min}= 0 to AET
- For each scanline:
  - *Sort edges in AET by x intersection*
  - *Walk from left to right, setting pixels by parity rule*
  - *Increment scanline*
  - *Retire edges with $Y_{max} < Y$*
  - *Add edges with $Y_{min} < Y$*
  - *Recalculate edge intersections (how?)*
- Stop when $Y > Y_{max}$ for last edges