Lecture 8:

Rasterization

EEC 277, Graphics Architecture

John Owens, UC Davis, Winter 2017

No lecture Feb 14

- I'm a member of the dean's strategic planning committee!
- Feb 16: Texturing

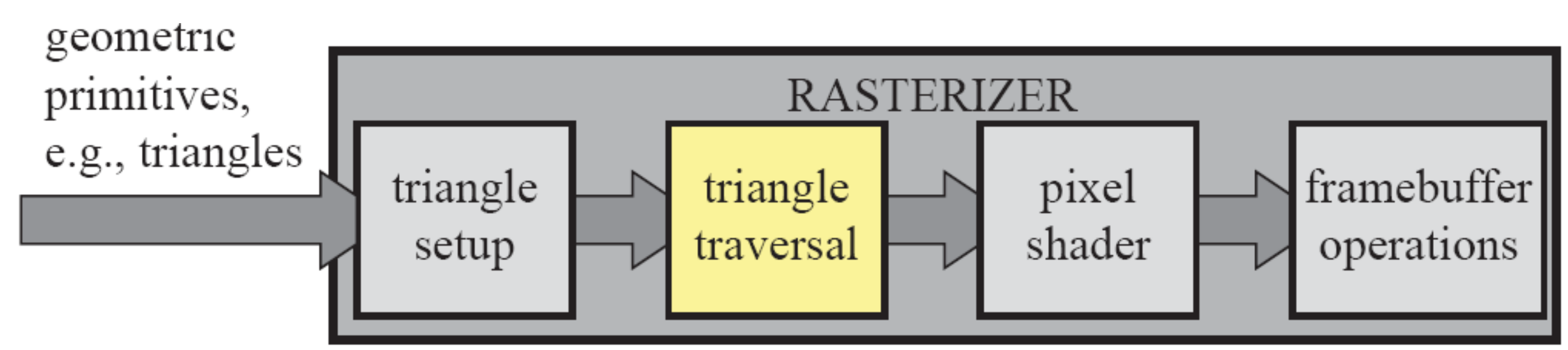
Outline

- Rasterization basics
- Pixel coverage
- Parameter determination
- Perspective correction
- Implementations

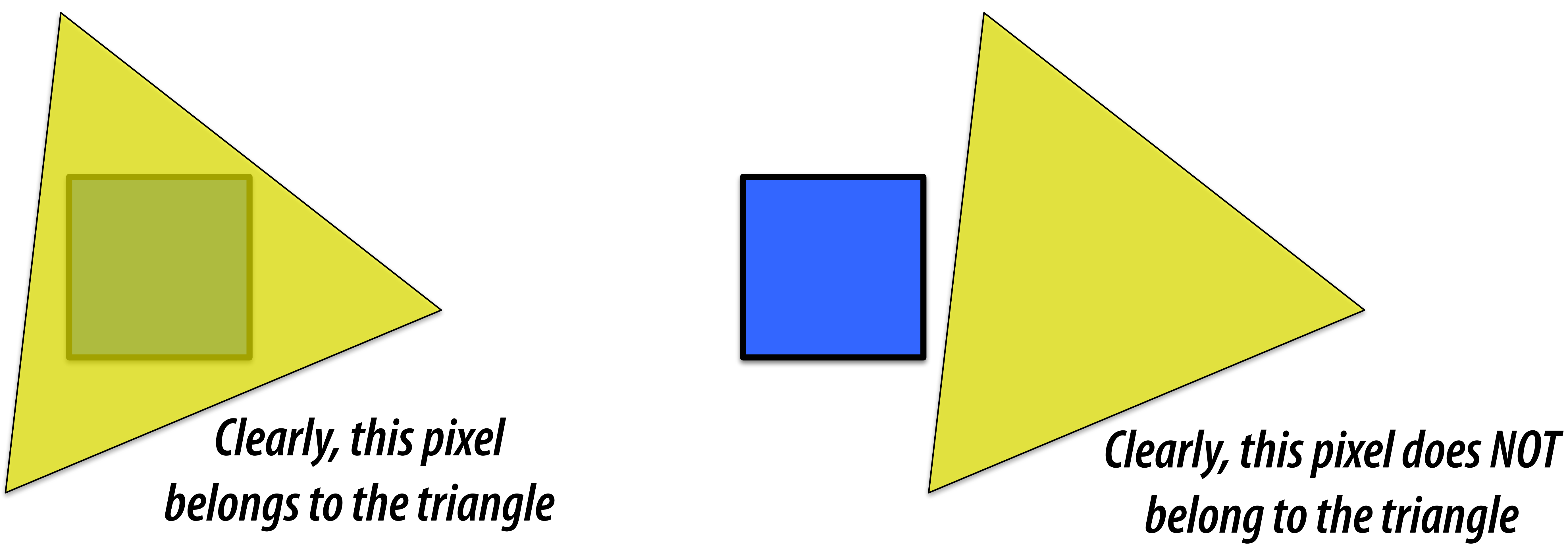
What Is Rasterization?

- Input:
  - Screen-space geometry
  - Parameters per vertex
- Output:
  - Fragments
    - Sample (pixel) location
    - Parameters per fragment
- Rasterization has two parts:
  - Pixel coverage
  - Parameter determination

Let's study triangle traversal

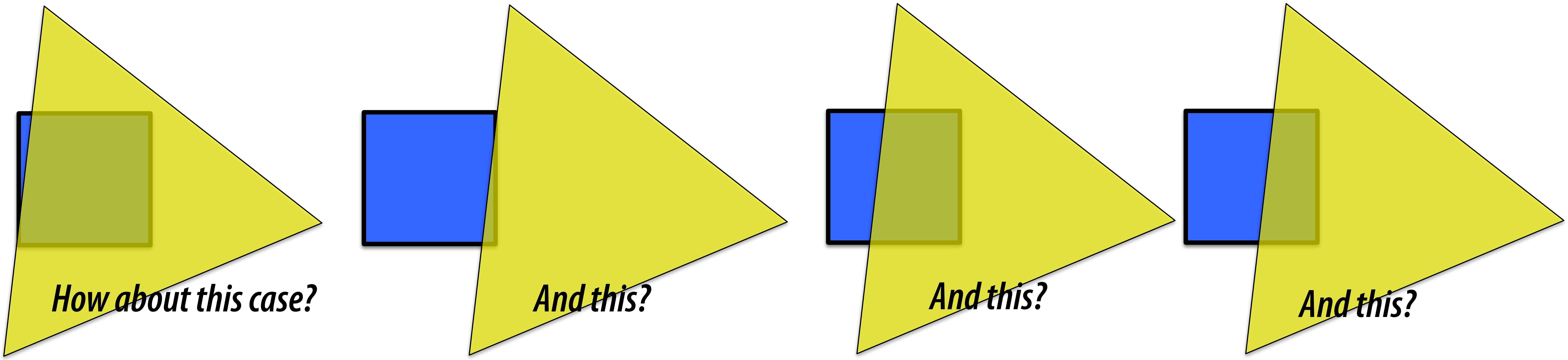


- Critical operation in rasterizer
  - without it, we have nothing
  - When we have it, we will study algorithms to increase efficiency / reduce memory accesses
- When does a pixel belong to a triangle?

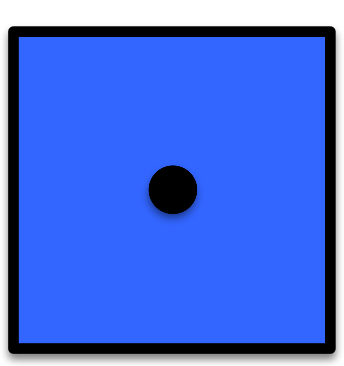


[courtesy Tomas Akenine-Möller]

When does a pixel belong to a triangle?

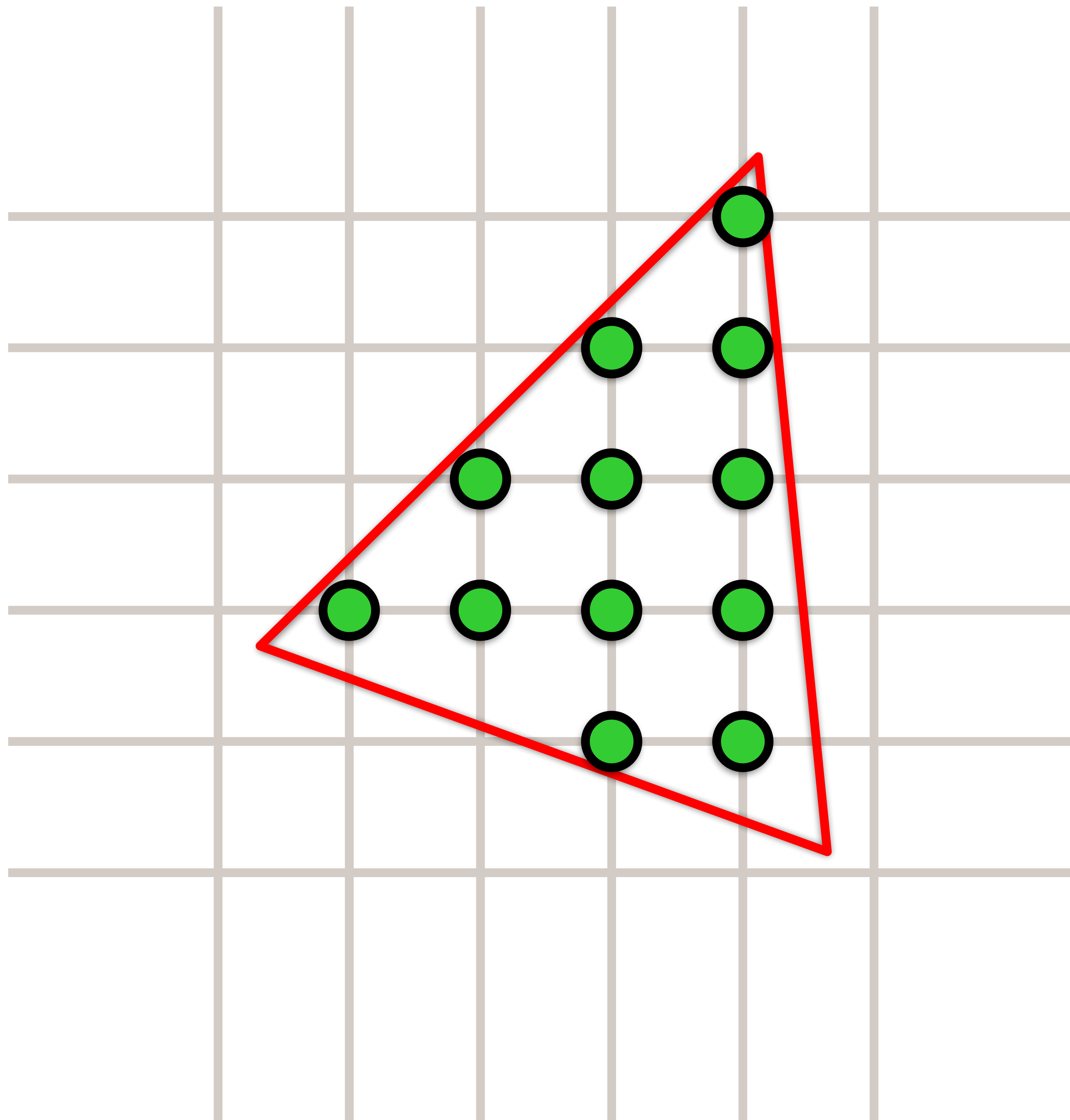


- **It all depends on where you sample!**
- **For (low-quality) normal rasterization, you sample in the center of the pixel:**
  - **If sample point is inside triangle, then pixel is inside (belongs to) the triangle.**
  - **In a later lecture, we will see how quality can be improved by using more than one sample per pixel**



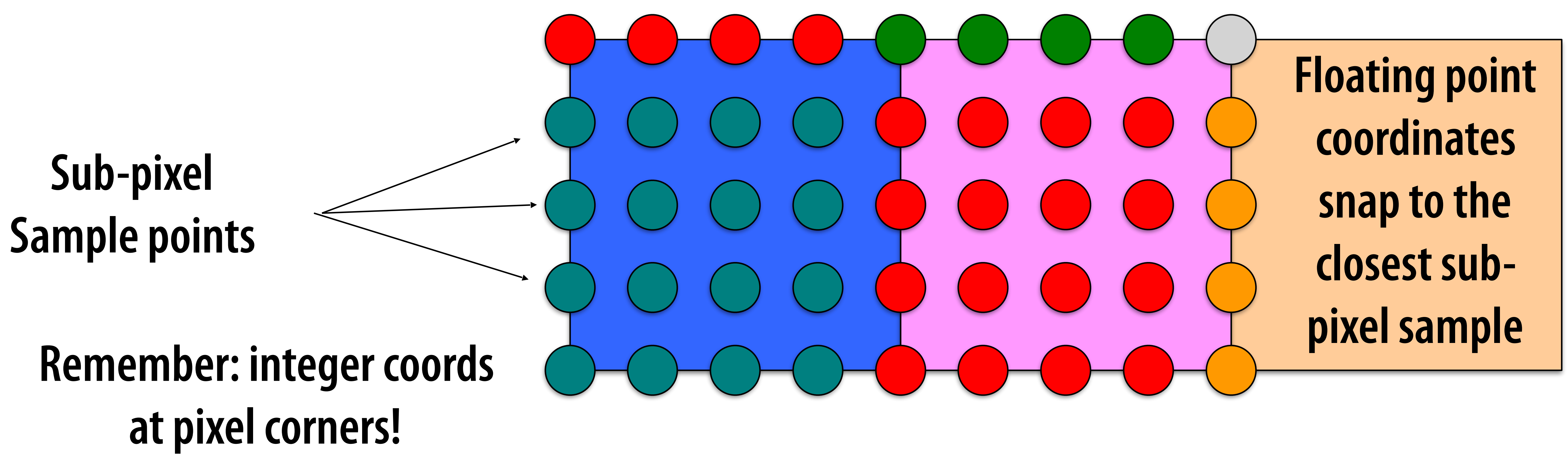
[courtesy Tomas Akenine-Möller]

Pixel Coverage



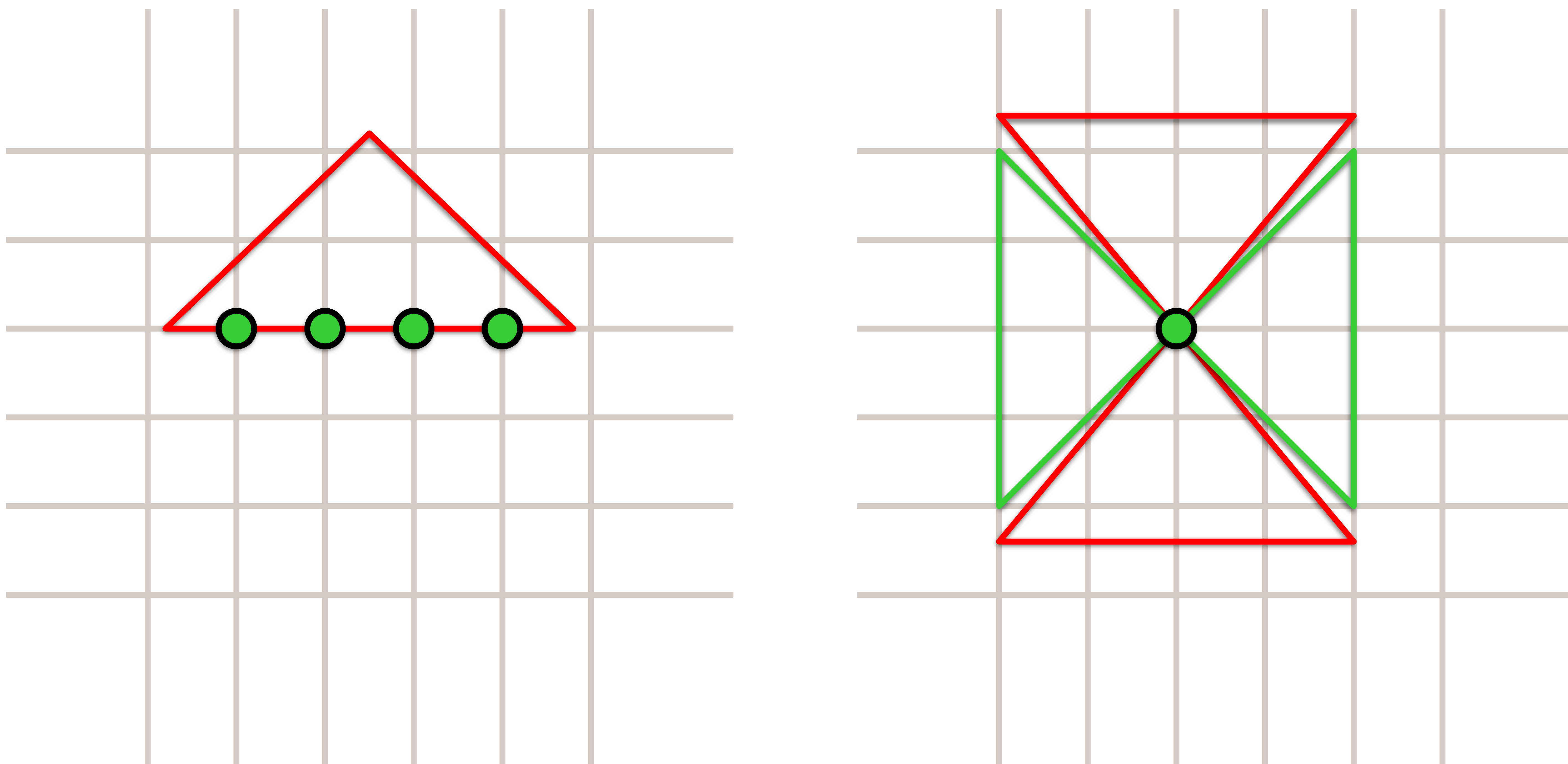
Representing sub-pixel coordinates

- Projected points are floating-point
- Due to limited range of the coordinates, we use fixed point math (integer)
- However, we cannot round off to nearest pixel center
- Instead use sub-pixel coordinates
  - Advantage: Simpler hardware (fixed-point not floating point)
  - Advantage: Many small tris (e.g., CAD applications) disappear
- With 2 subpixel fractional bits per x, and y, we get:



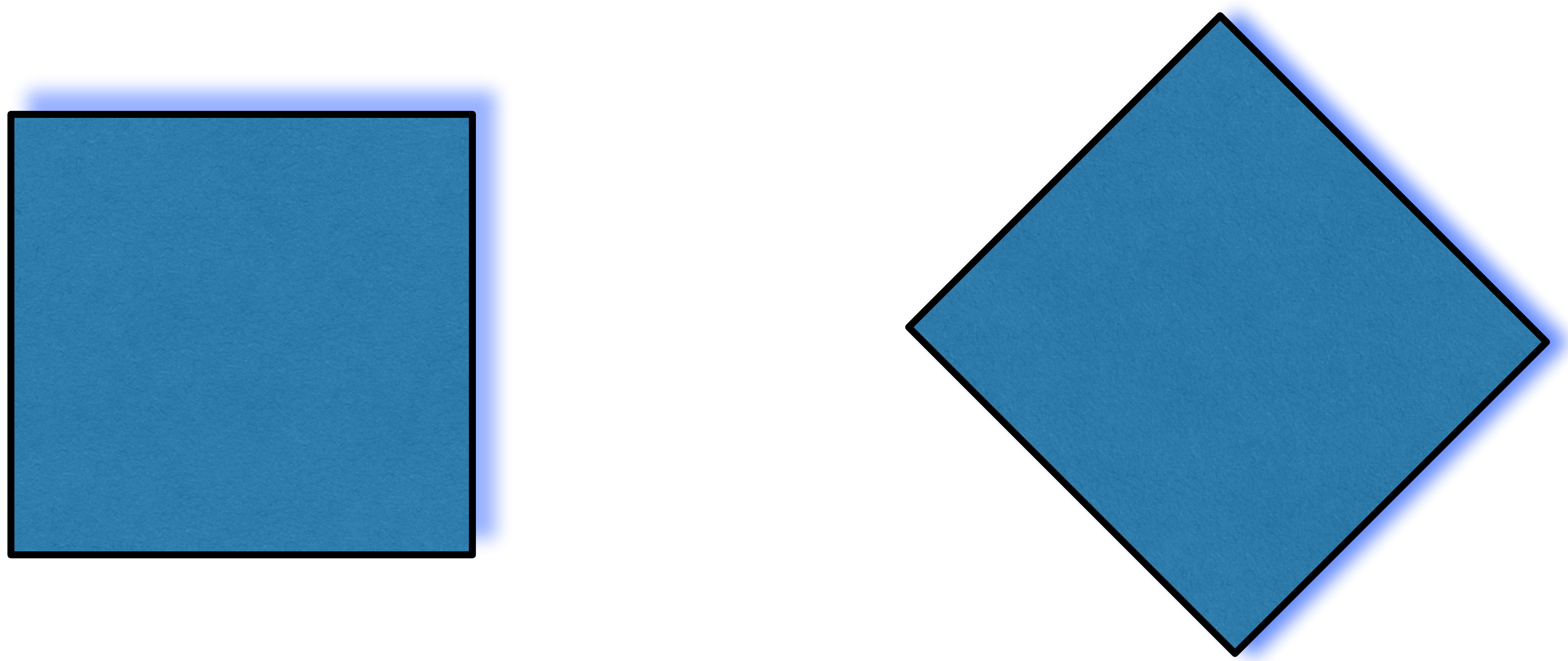
[courtesy Tomas Akenine-Möller]

Shared Pixels



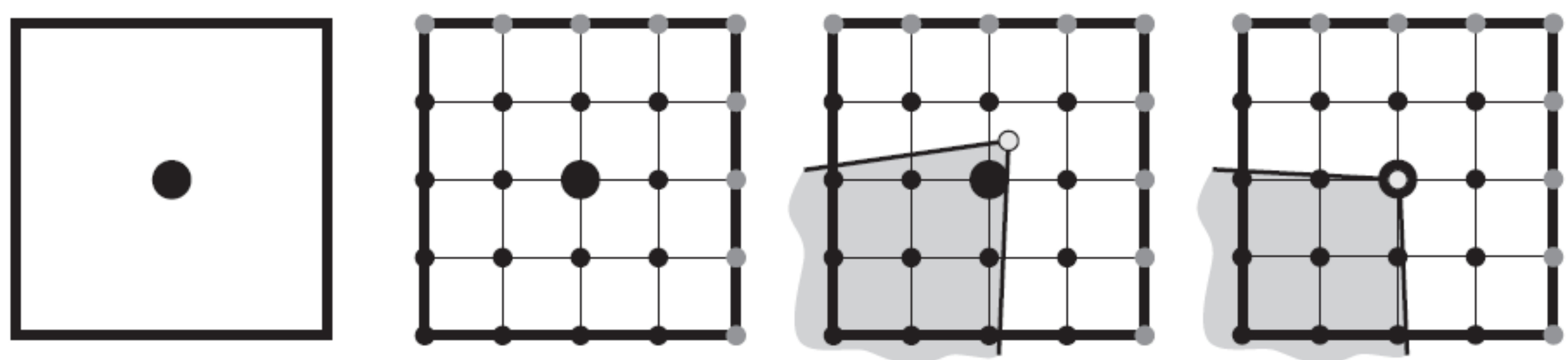
Shadow Rules

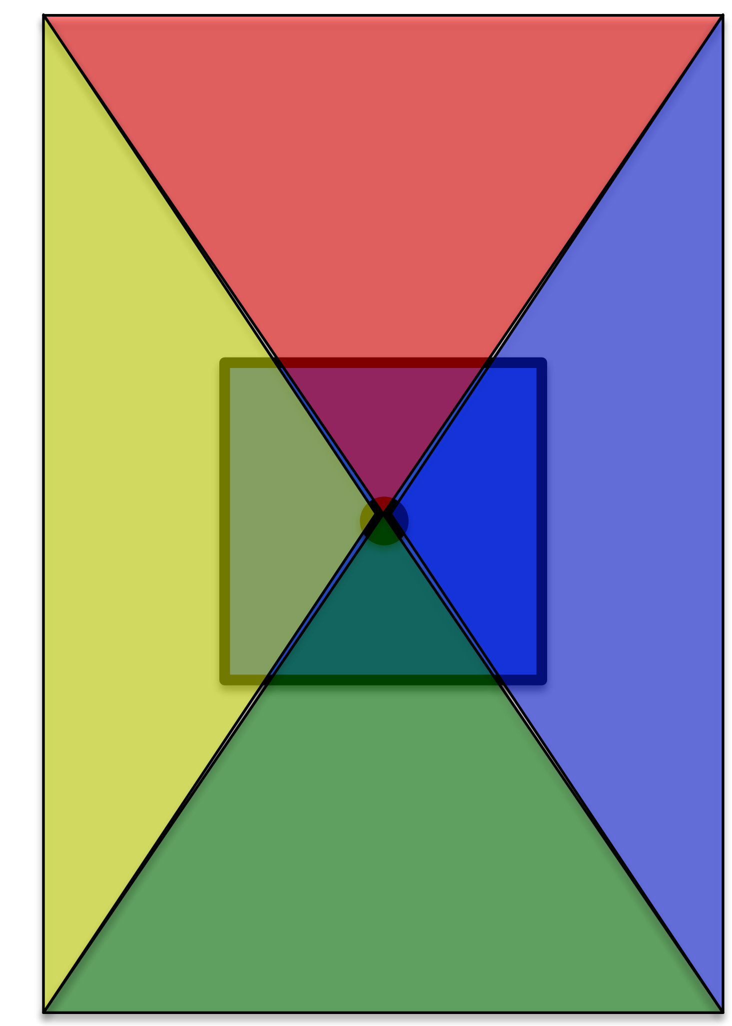
- Polygon edges are in two groups: shadowed and non-shadowed
- Facing right/left: In shadow/out of shadow
- If tie, facing up/down: In shadow/out of shadow



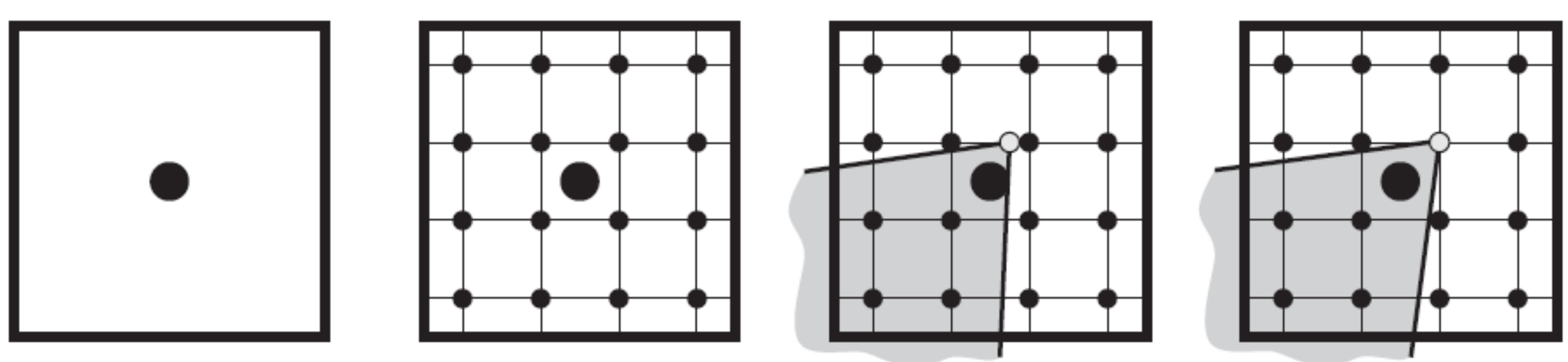
How about when a vertex coincides with the sampling point?

- You get the same kind of problems!



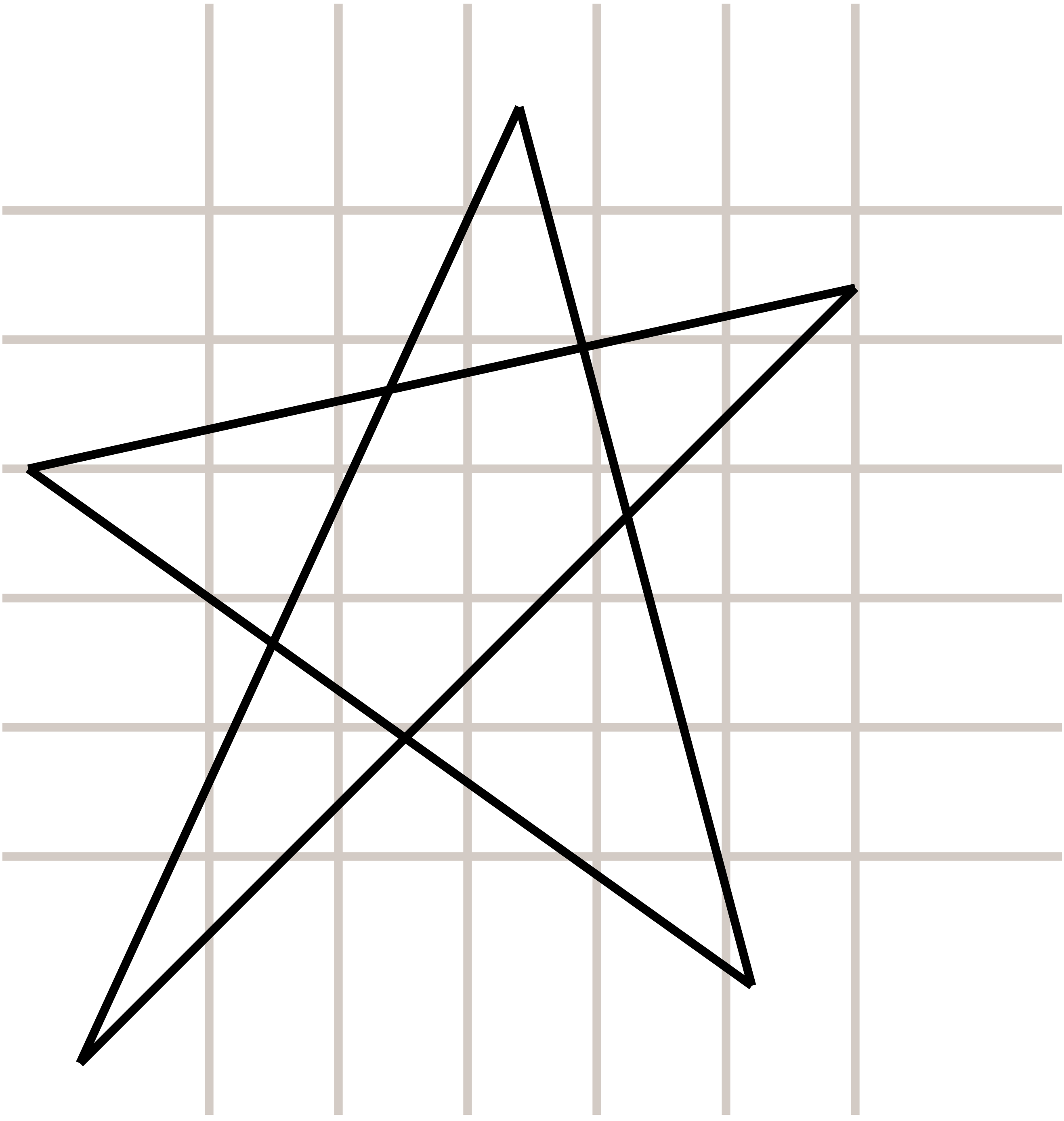


- One solution: offset the subpixel grid so that sampling points never coincides with sub-pixel grid



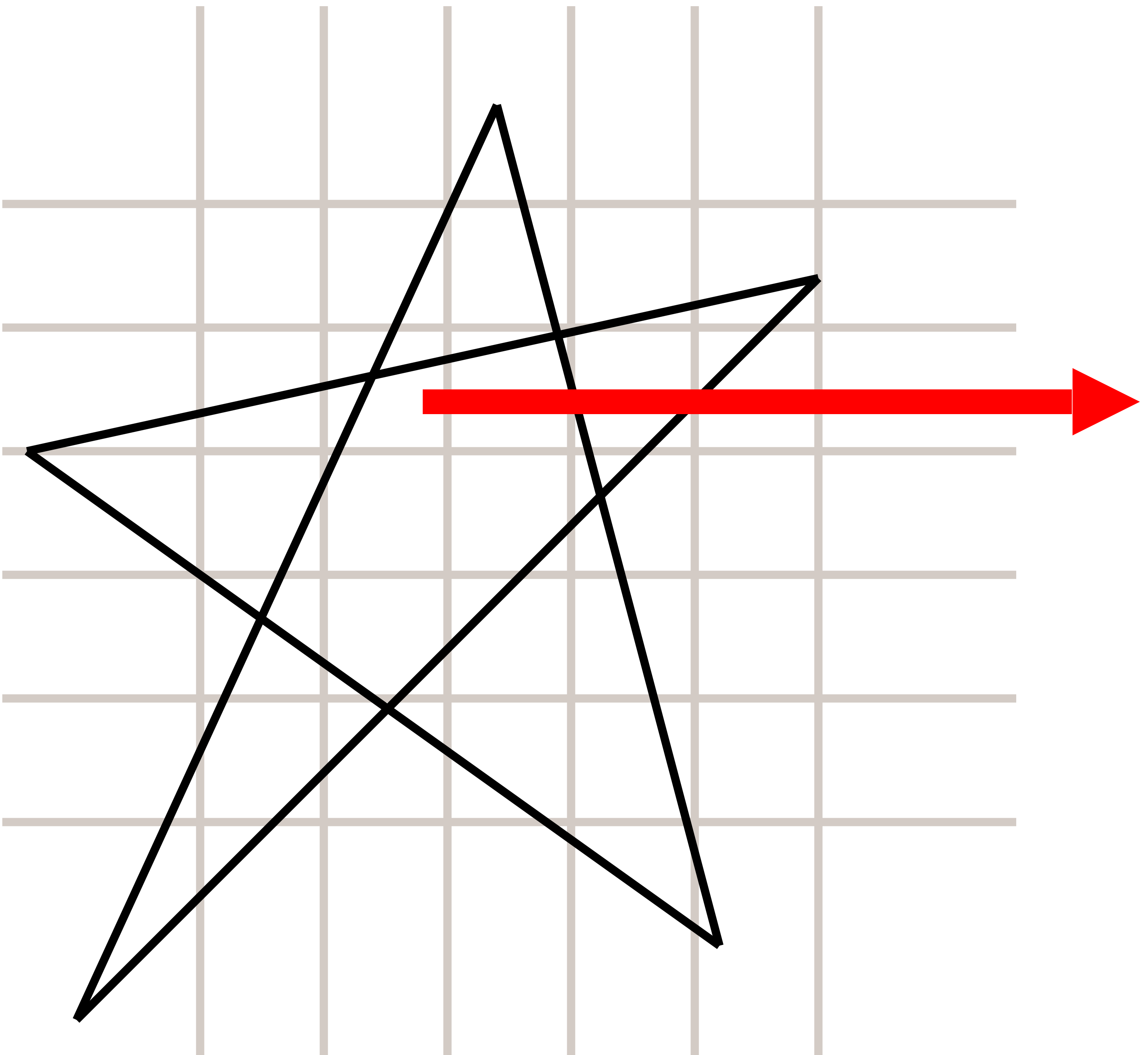
[courtesy Tomas Akenine-Möller]

Complex Polygon Insidedness



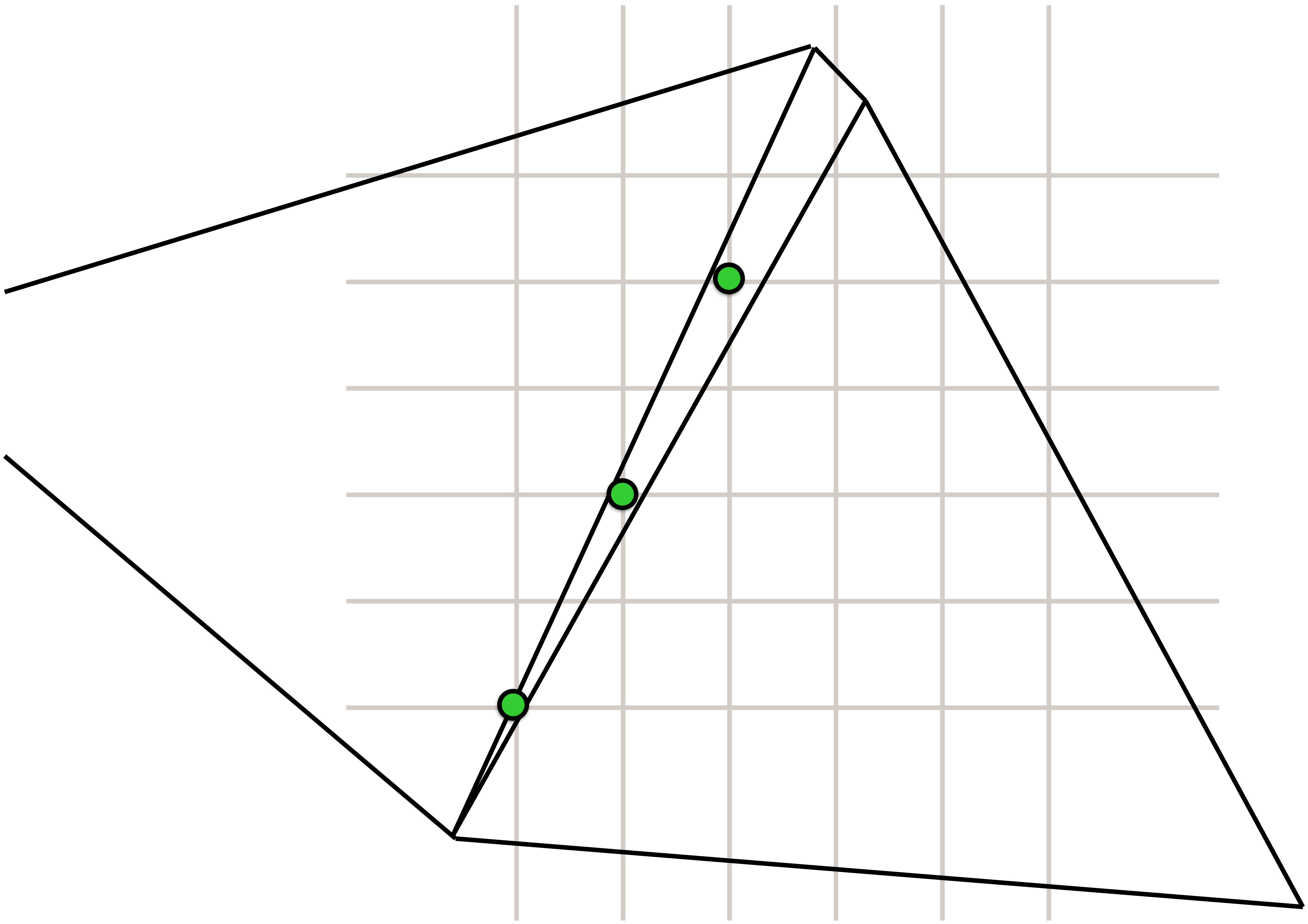
Jordan Curve Theorem

- Count the number of times a ray emitted from a point intersects a boundary.



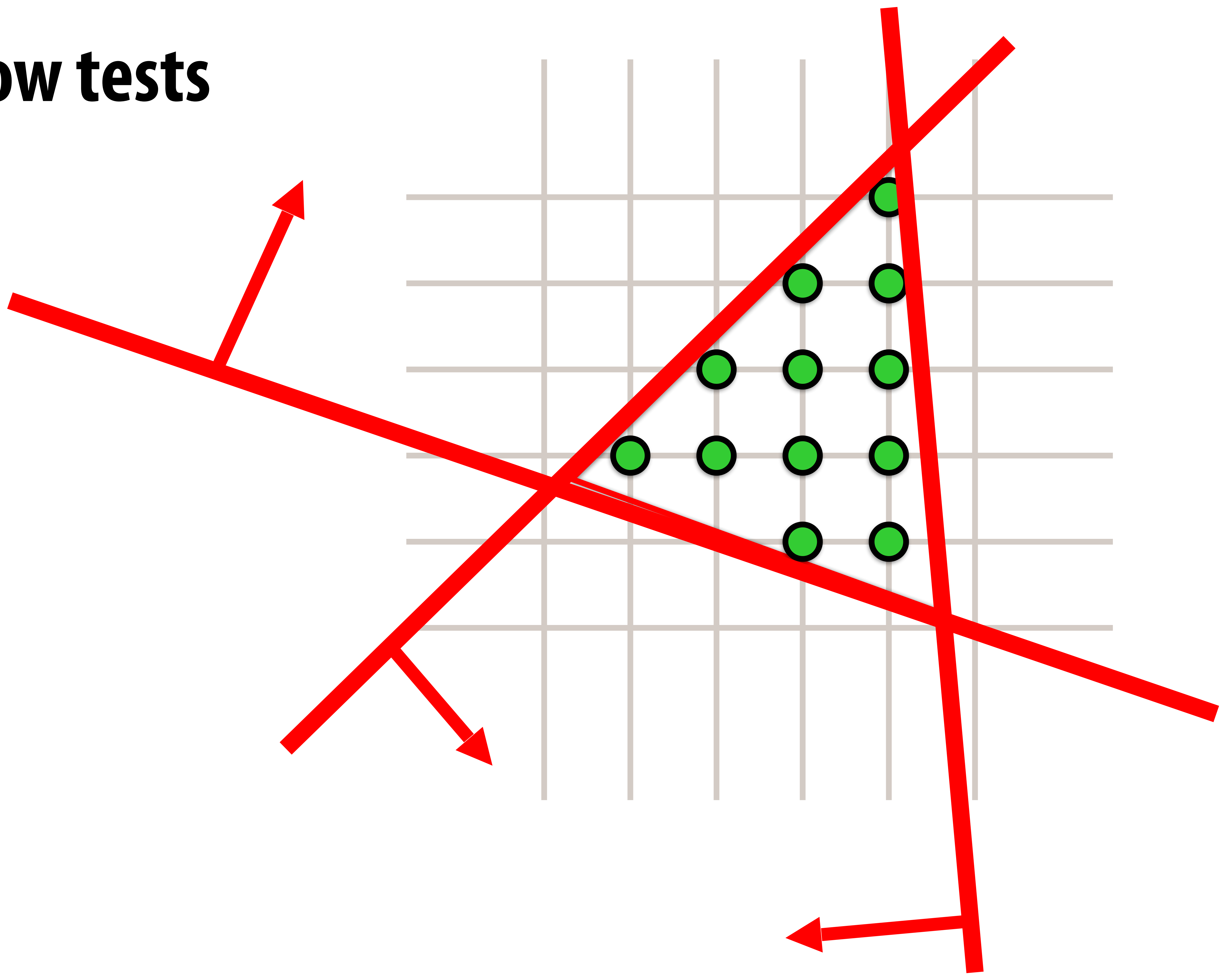
- Odd: Inside
- Even: Outside

Disconnected Fragments



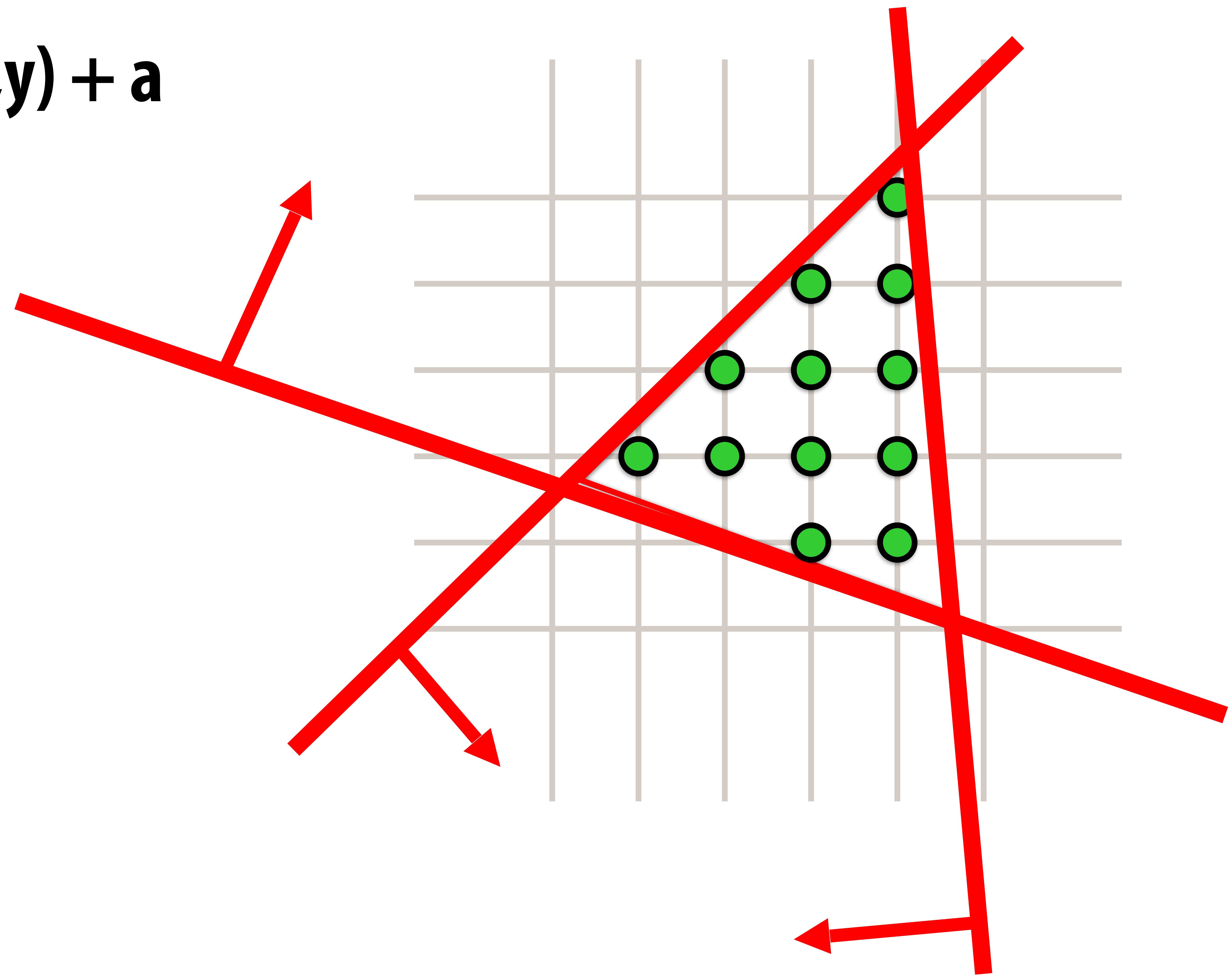
Insidedness Tests: Edge Equations

- $L = ax + by + c$
- L>0: in ; L=0: on ; L<0: out
- "on": use shadow tests



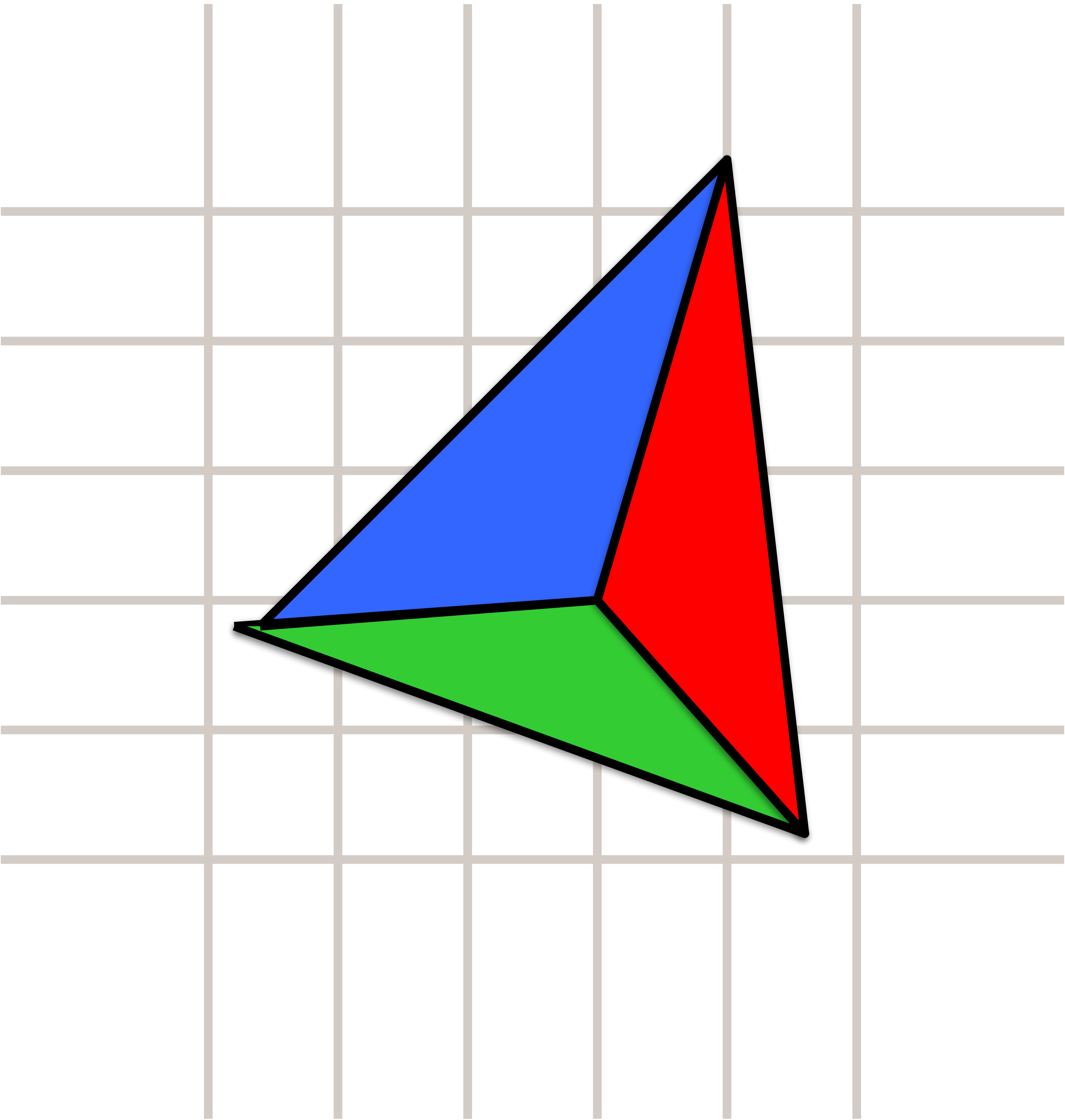
Insidedness Tests: Edge Equations

- $L = ax + by + c$
- Incremental!
- $L(x+1,y) = L(x,y) + a$



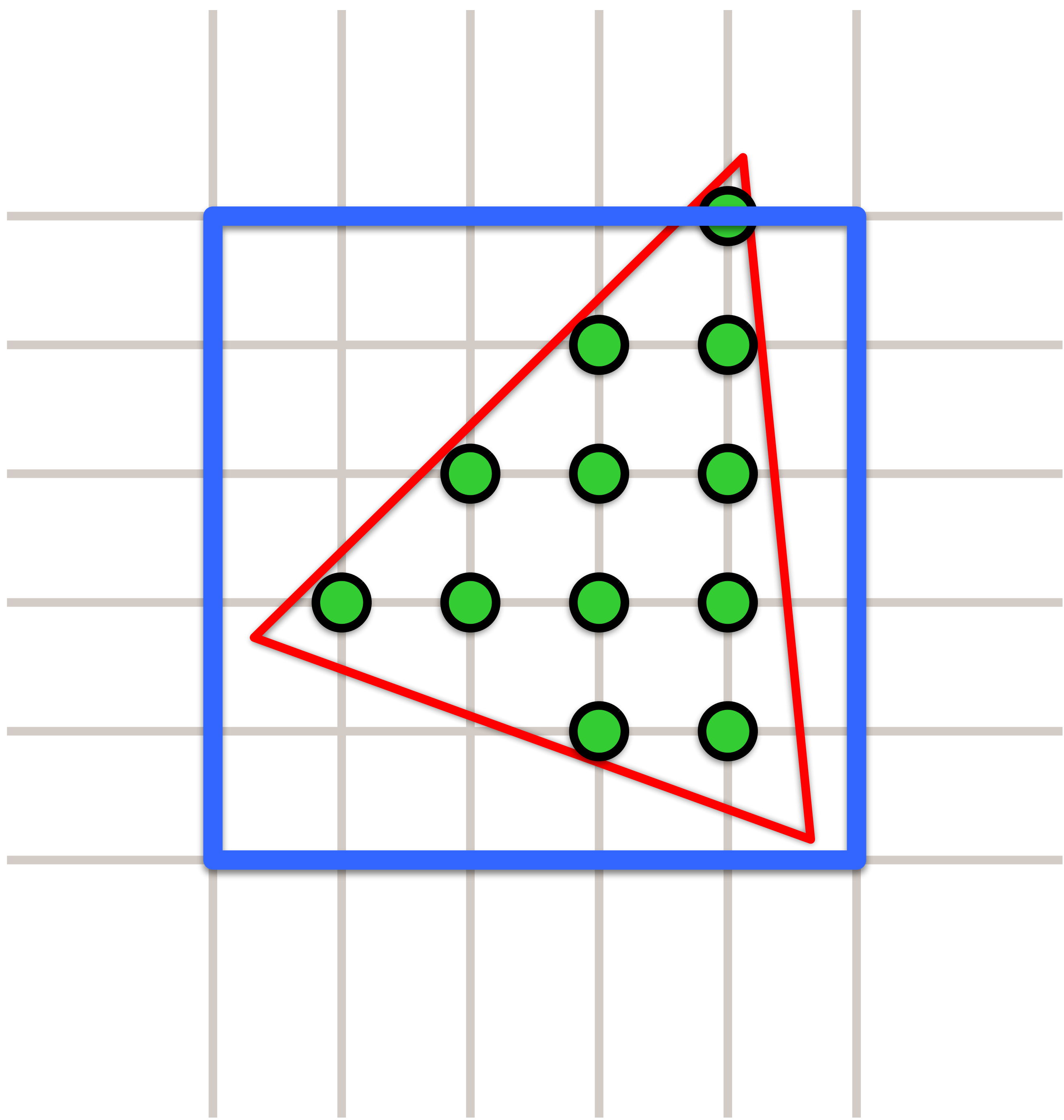
Insidedness Tests: Barycentric Formulation

- A_0: area of red triangle; A_1, blue; etc.
- $B_0 = A_0/(A_0+A_1+A_2)$; etc.
- If $B_i < 0$, pixel is outside



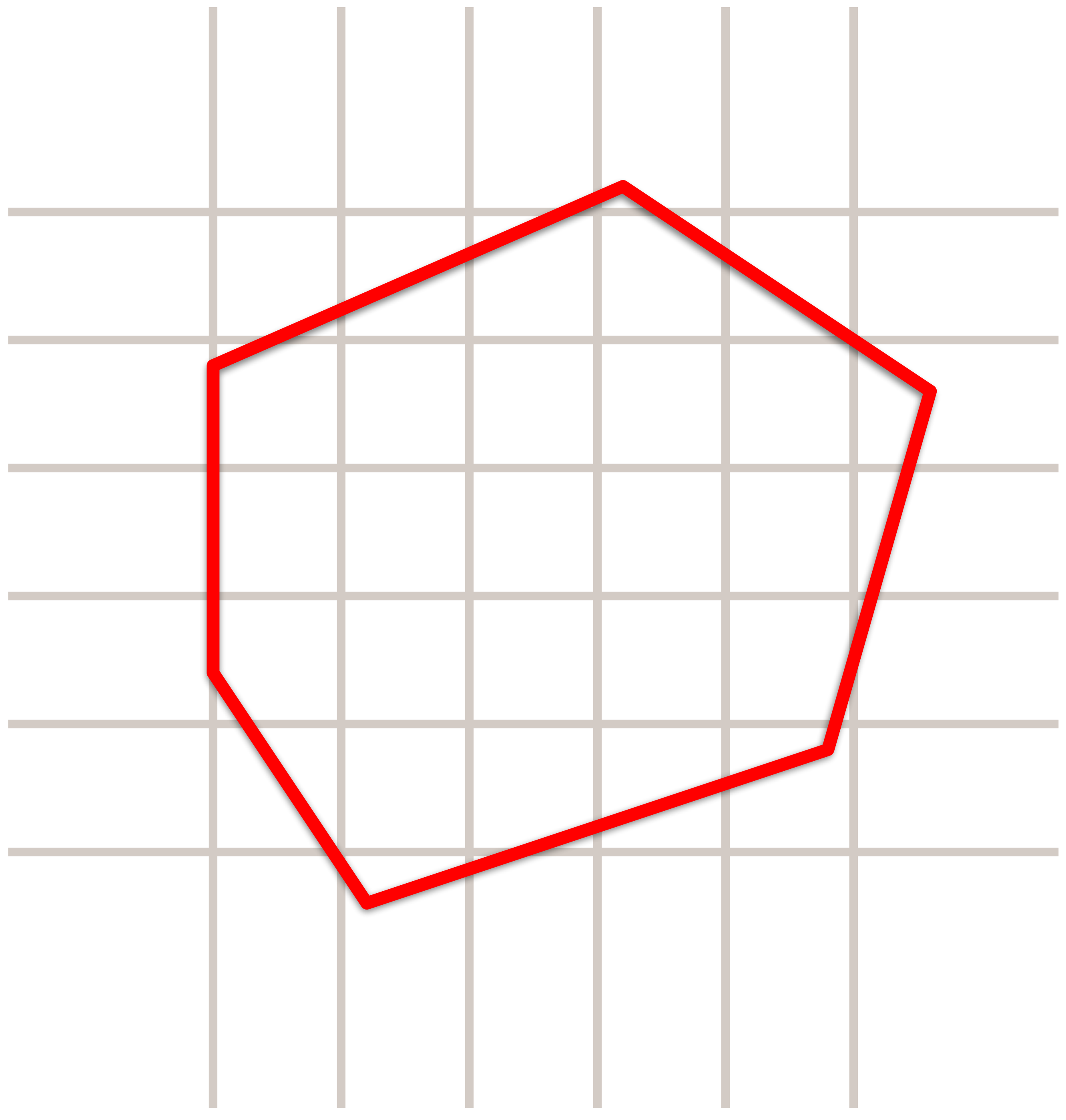
Bounding Box Traversal

- Good: Simple
- Bad: Inefficient
- How to tell inside/outside?



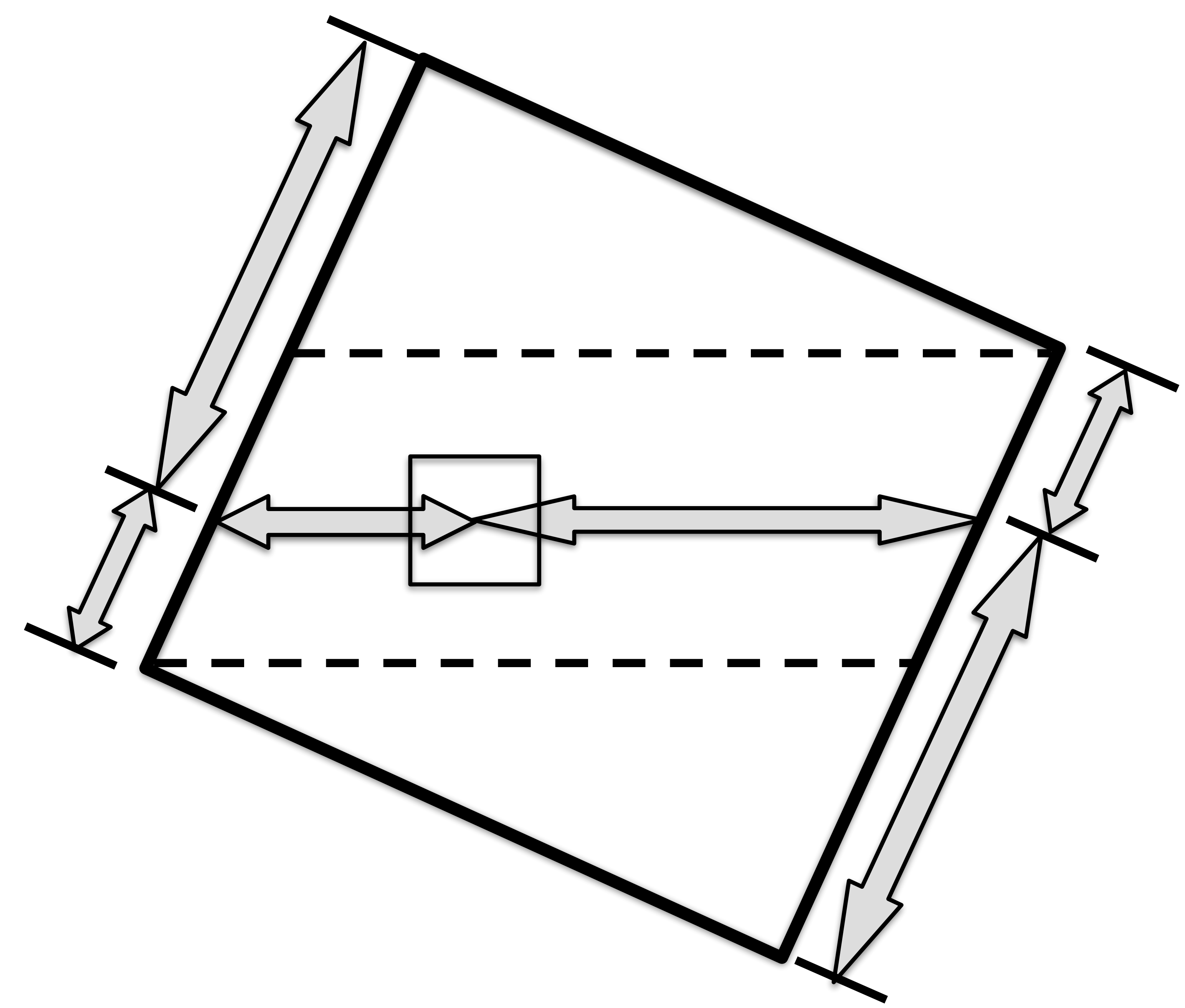
Crow's Algorithm (Scanline)

- Find lowest *y*
- Determine left and right edges
- Sweep upward, outputting "spans", until edge changes
- Replace edge and keep going
- Continue until no more edges



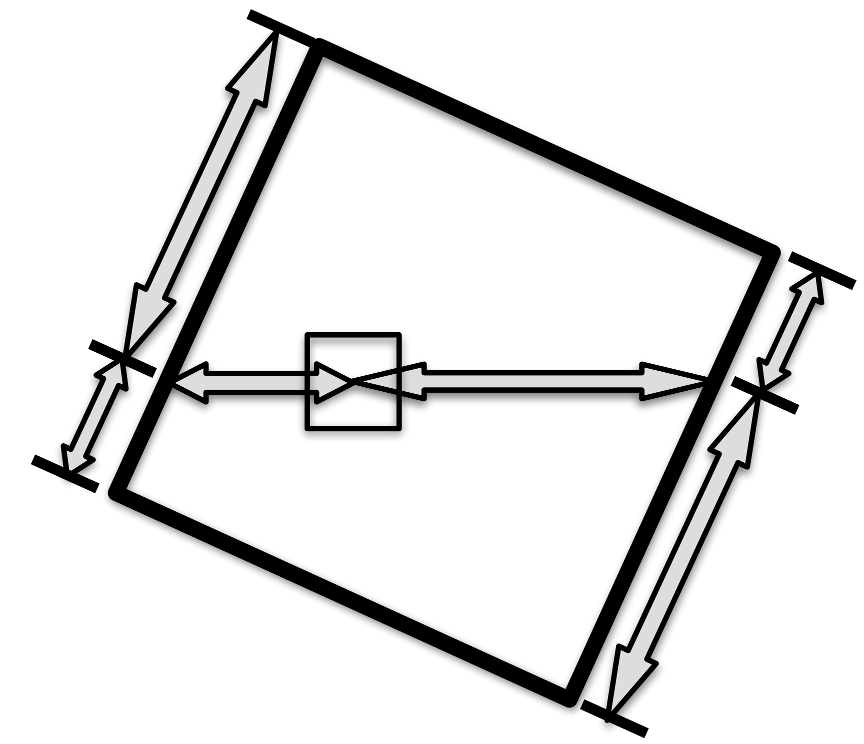
Example: Quadrilateral

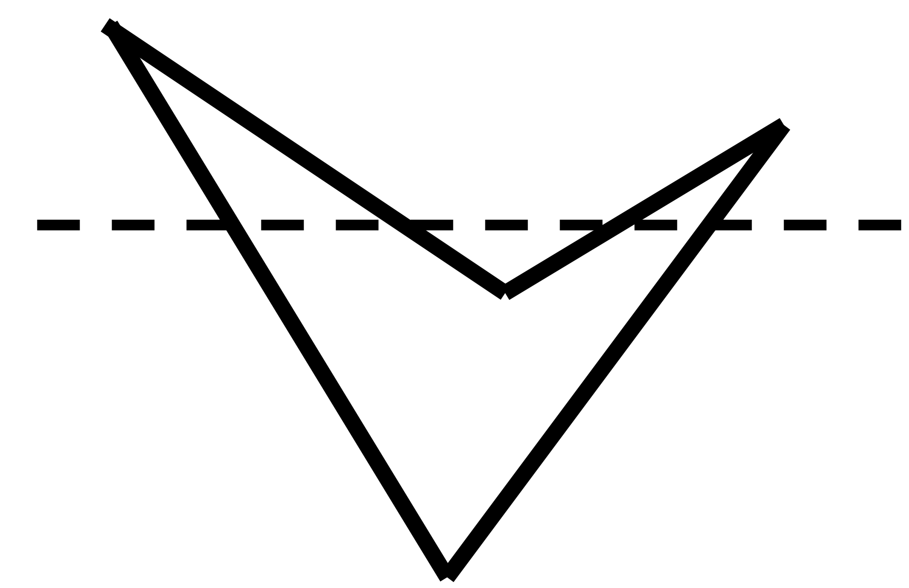
- Pixel coverage
  - Three distinct regions
    - Loop is complex
    - Must handle 1, 2, or 3 regions
  - Function of
    - Screen orientation
    - Choice of ⇔ ⇕ spans

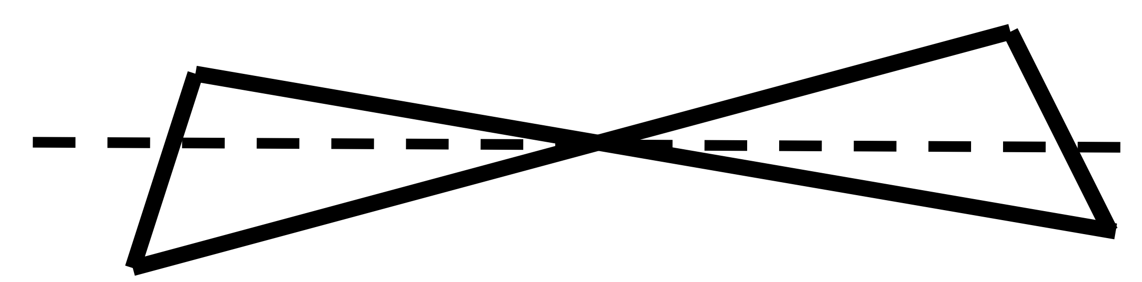


Example: Quadrilateral

- "All" projected quadrilaterals are non-planar
  - Due to discrete coordinate precision
- What if quadrilateral is concave?
  - Concave is complex (split spans—see example)
  - Non-planar → concave for some view
- What if quadrilateral intersects itself?
  - A real mess (no vertex to signal change—see example)
  - Non-planar → "bowtie" for some view







All polygons are triangles (or should be)

- Three points define a plane
  - Can treat all triangles as planar
  - Can treat all parameter surfaces as planar
- Triangle is always convex
  - Regardless of arithmetic precision
  - Simple rasterization, no special cases
- Modern GPUs decompose *n*-gons to triangles
  - SGI switched in 1990, VGX product
  - Optimal quadrilateral decomposition invented

Incremental triangle traversal

$P_i = (X_i, Y_i)$

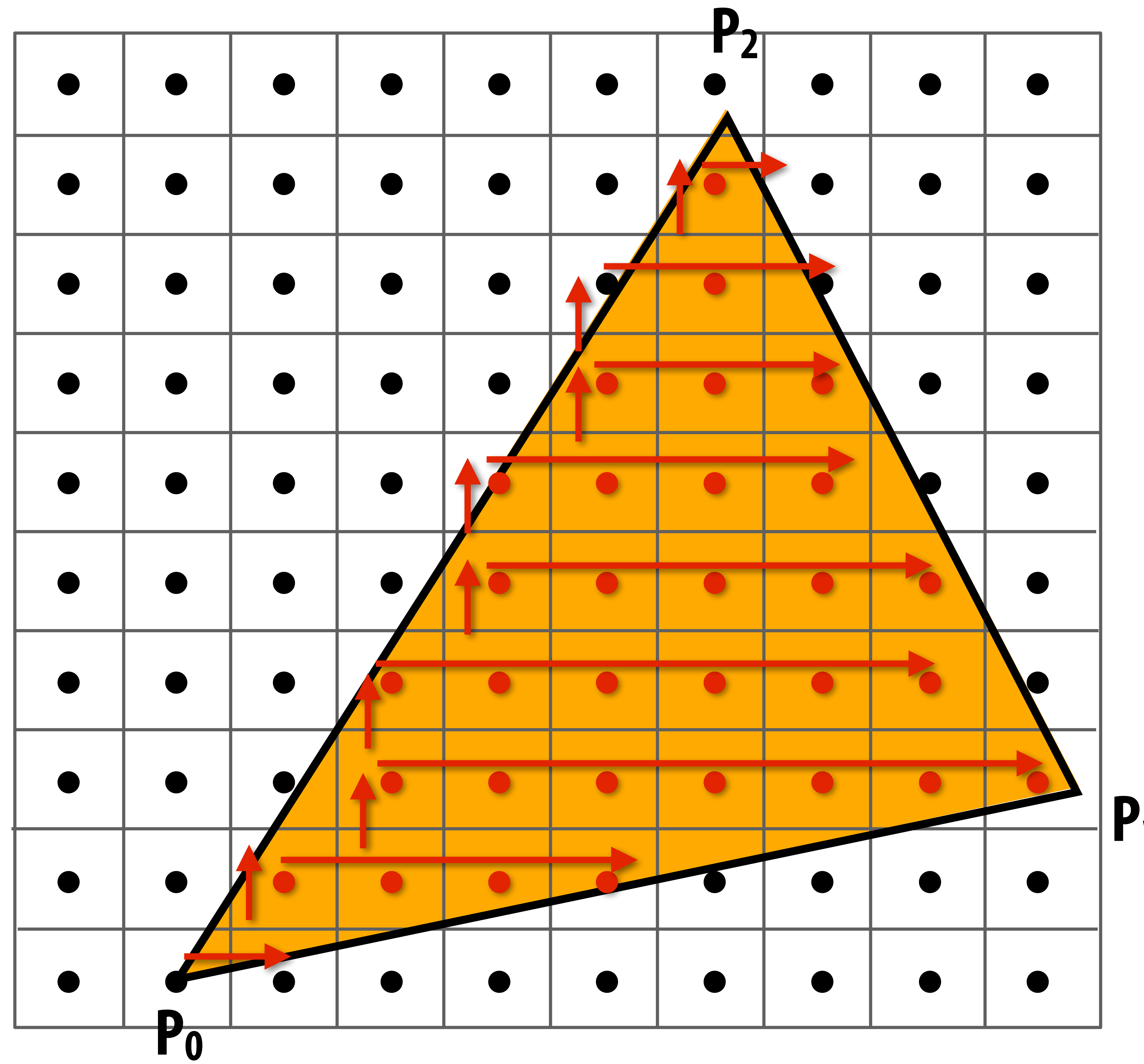
$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$$E_i(x, y) = (x - X_i)\, dY_i - (y - Y_i)\, dX_i$$
$$= A_i\, x + B_i\, y + C_i$$

$E_i(x, y) = 0$: point on edge
> 0 : outside edge
< 0 : inside edge

Efficient incremental update:

$$dE_i(x+1,y) = E_i(x,y) + dY_i = E_i(x,y) + A_i$$
$$dE_i(x,y+1) = E_i(x,y) + dX_i = E_i(x,y) + B_i$$

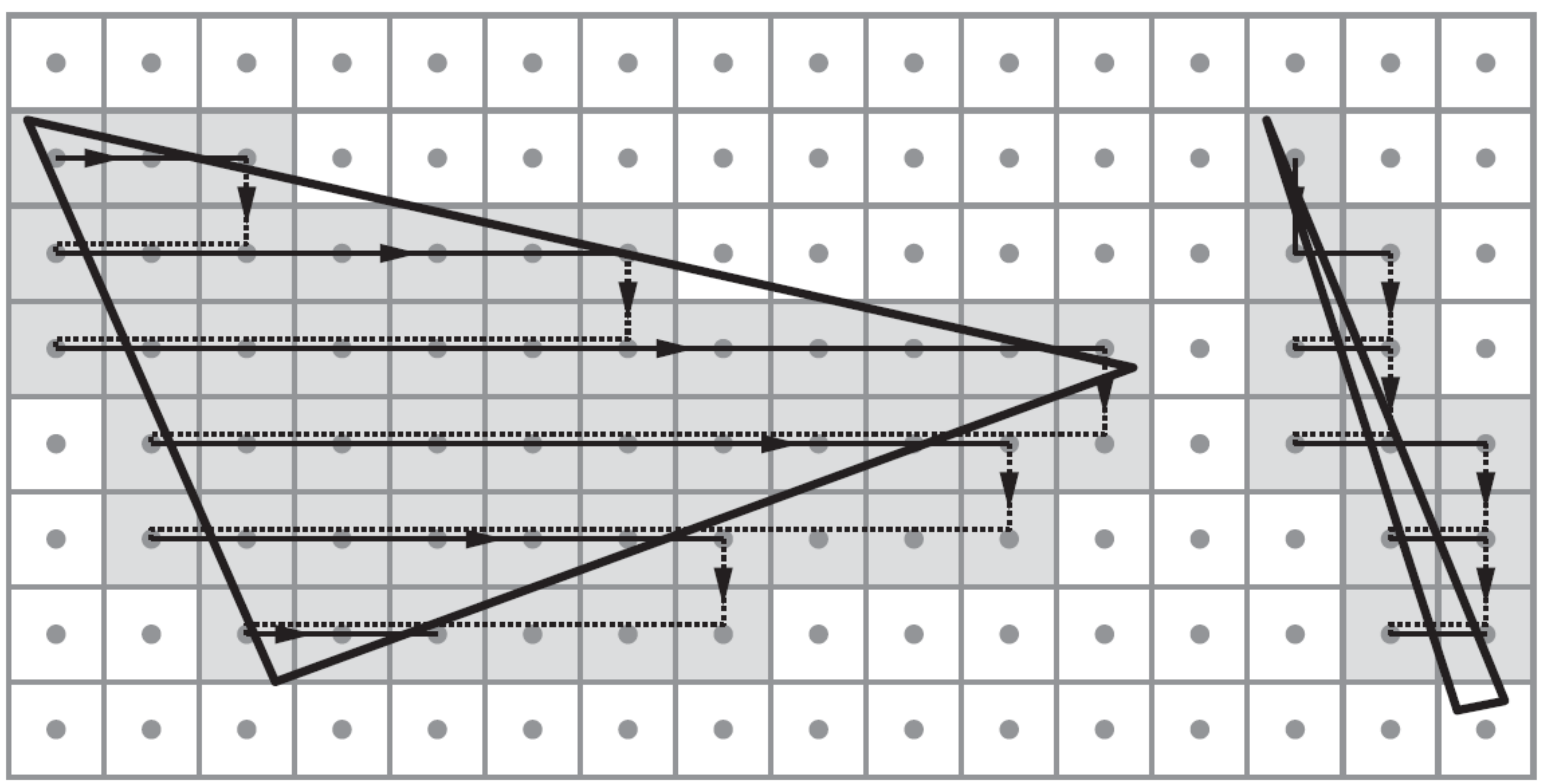


Incremental update saves computation:
Only one addition per edge, per sample test

Many traversal orders are possible: backtrack, zig-zag, Hilbert/Morton curves (locality maximizing) ***project idea?***

Backtrack traversal

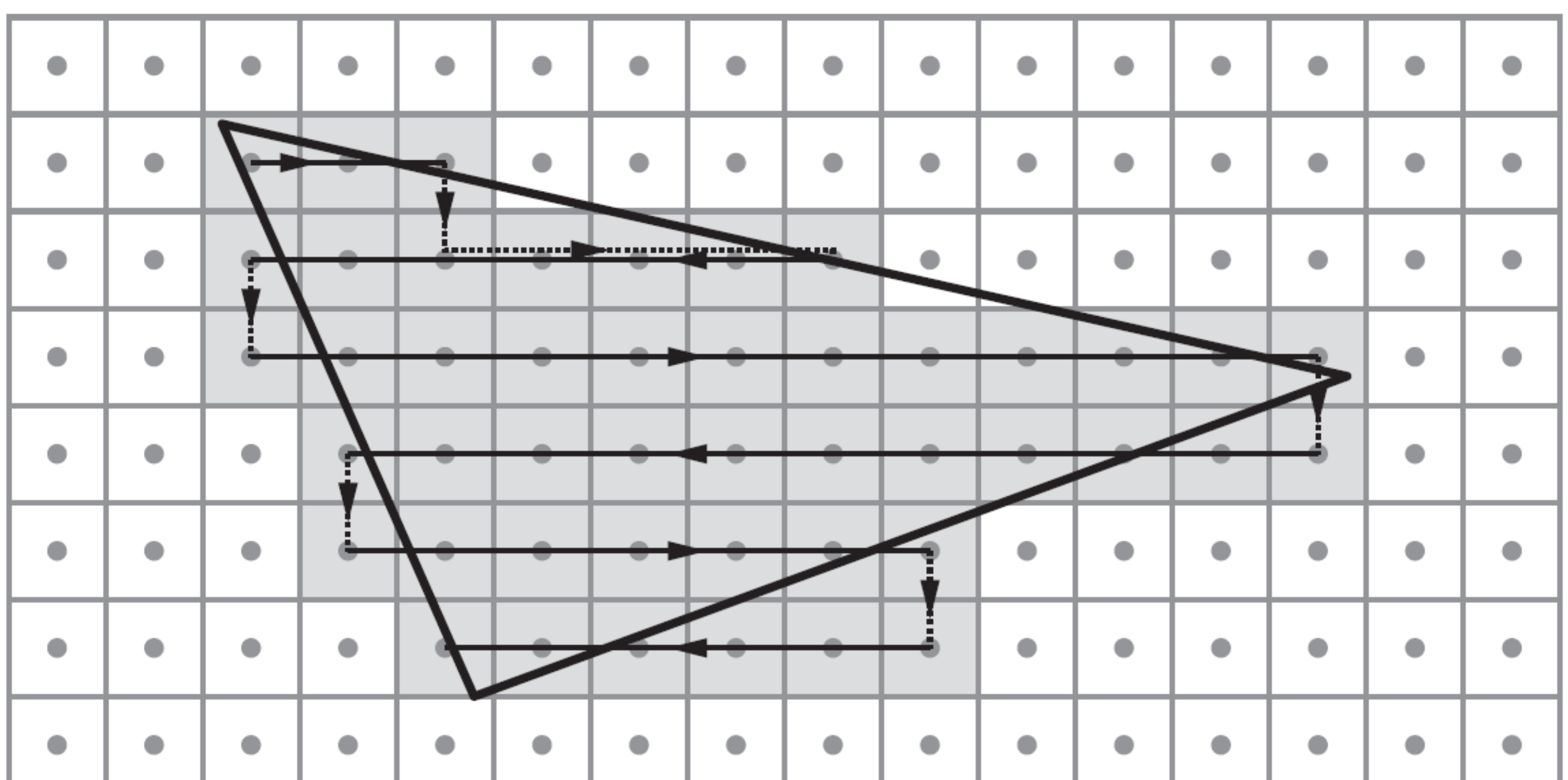
- Have been used on mobile devices (by a Korean research group)
- Advantage: only traverse from left to right
  - Could make for more efficient memory accesses
  - Could backtrack at a faster pace (because no mem acc)



[courtesy Tomas Akenine-Möller]

Zigzag traversal

- Simple technique that avoids backtracking
  - Otherwise, very similar
  - Can still visit a bunch of pixels outside (see next to most bottom scanline)
  - Can be solved, but requires more `Inside()`-testing

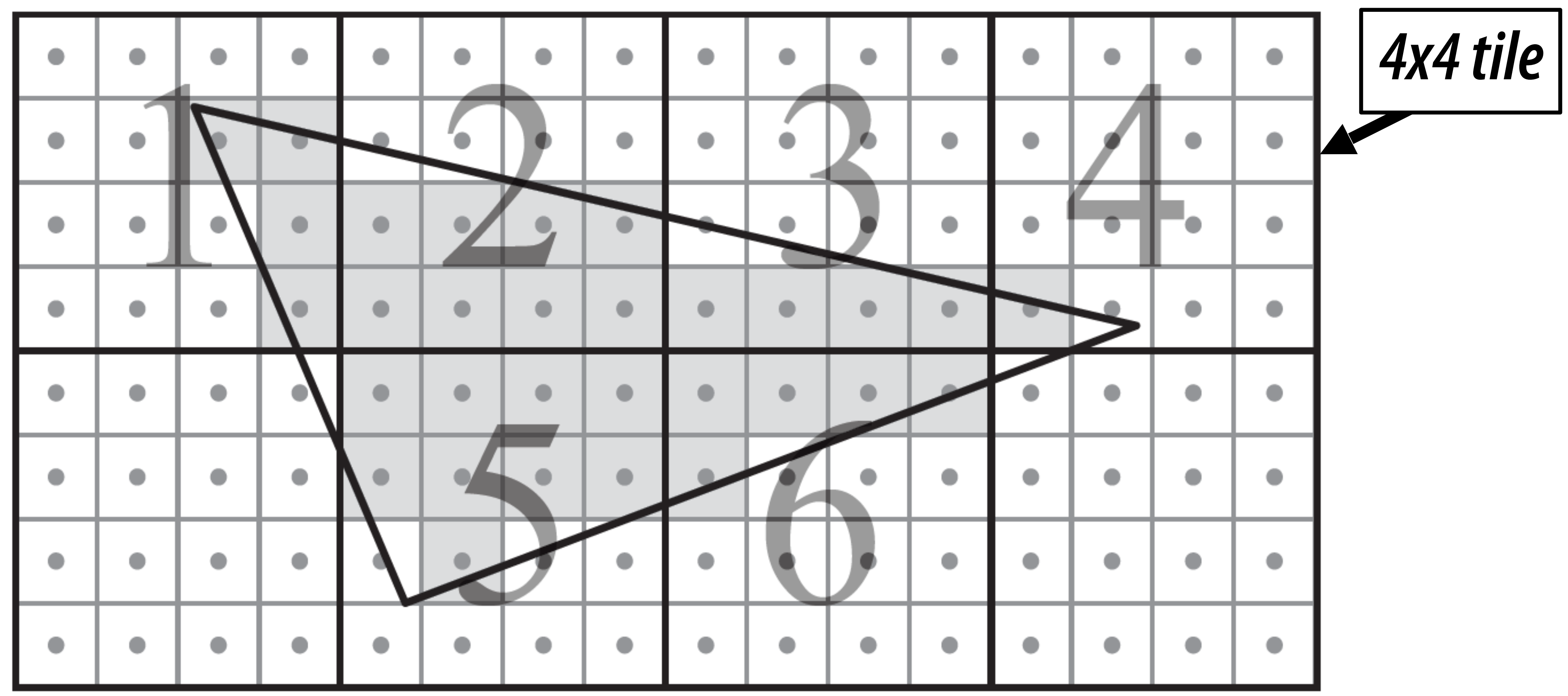


[courtesy Tomas Akenine-Möller]

Tiled traversal

- General idea: divide screen space into non-overlapping tiles (a tile is $w \times h$ pixels)
  - Traverse one tile at a time, and finish visiting pixels in tile before moving to next tile. Within a tile, test all pixels in parallel.
  - Can skip tiles if appropriate: entire block not in triangle ("early out"), entire block entirely within triangle ("early in")

8x8 tile size is common in desktop graphics cards



- Better because (all topics will be treated in later lectures):
  - Gives better texture cache performance
  - Enables simple culling (Zmin & Zmax)
  - Real-time buffer compression (color and depth)

[courtesy Tomas Akenine-Möller]

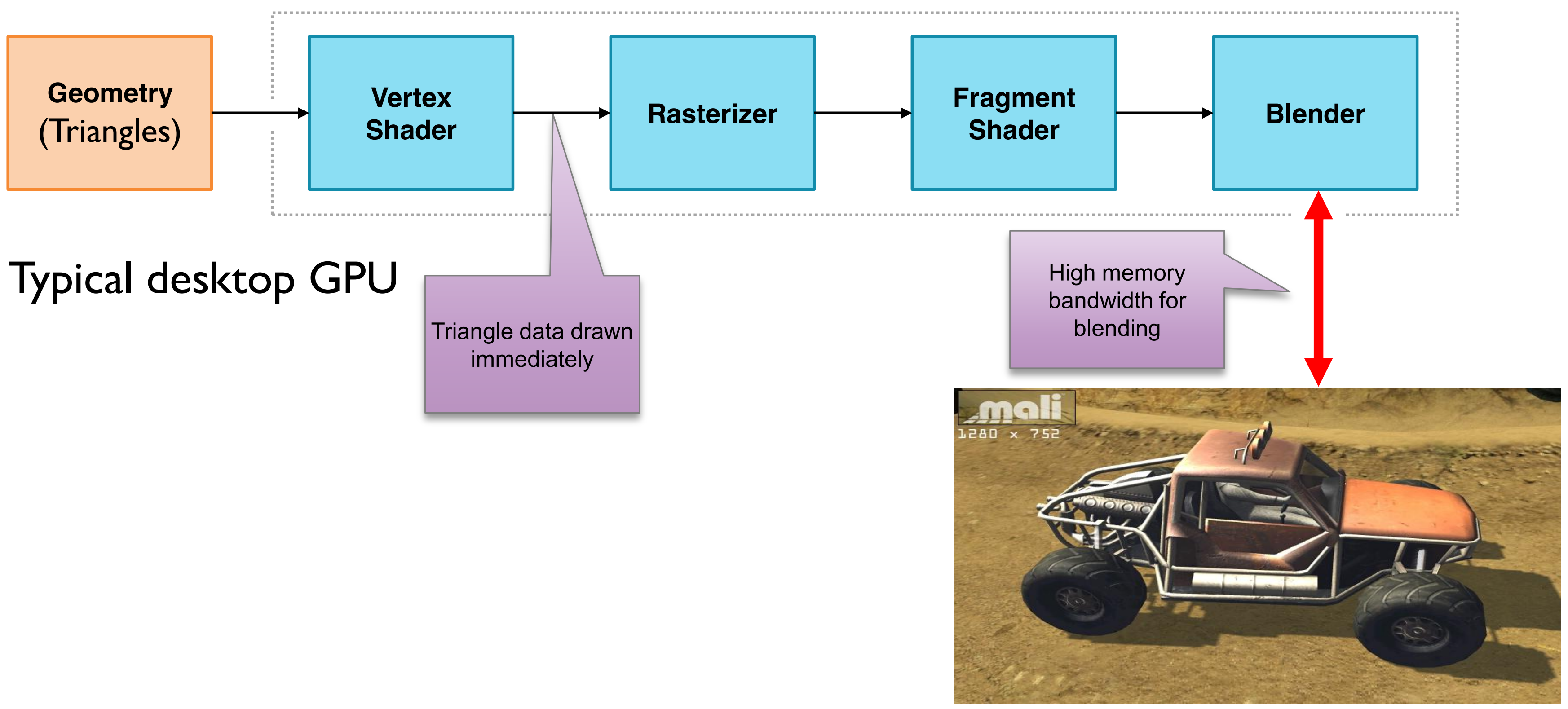
Is tiled traversal that different?

- No, not really. We need:
  - I: Traverse to tiles overlapping triangle
  - II: Test if tile overlaps with triangle
  - III: Traverse pixels inside tile
- Previous algorithms can handle I and III
- II needs to be handled
  - Easily solved using … edge functions!

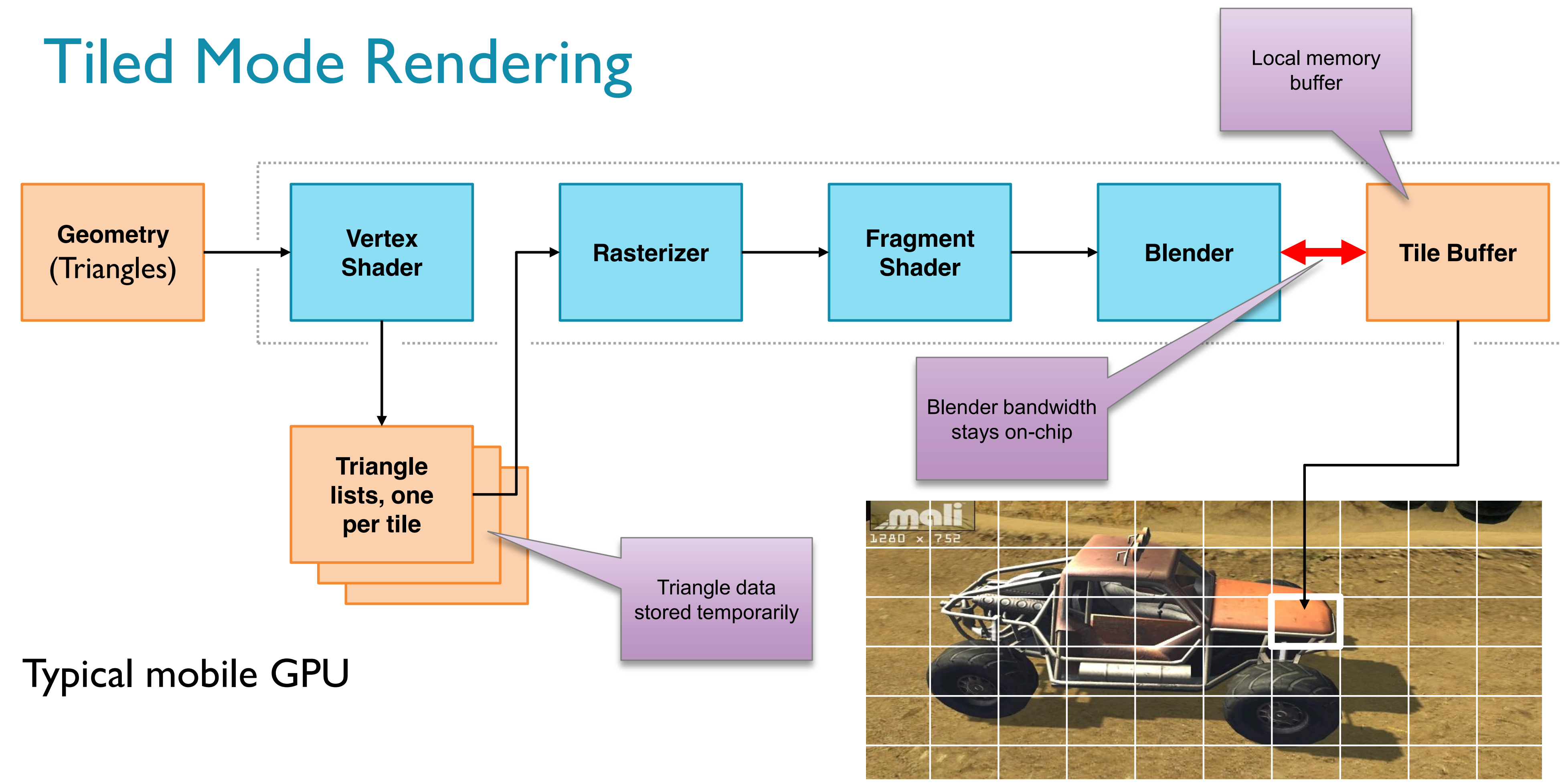
[courtesy Tomas Akenine-Möller]

Tile-based rasterization

Immediate Mode Rendering



Tiled Mode Rendering




```
foreach( tile )
     foreach( primitive in tile )
          foreach( fragment in primitive in tile )
               render fragment
```

- + : Tile memory on-chip
- – : Output of vertex stage must go to main memory
- NVIDIA Maxwell/Pascal:
  - Mobile-first architectures
  - Use tile-based rasterization
- AMD, Intel: Traditional immediate-mode rasterization

http://malideveloper.arm.com/downloads/ARM_Game_Developer_Days/PDFs/2-Mali-GPU-architecture-overview-and-tile-local-storage.pdf
https://community.arm.com/graphics/b/blog/posts/the-mali-gpu-an-abstract-machine-part-2---tile-based-rendering
http://www.anandtech.com/show/10536/nvidia-maxwell-tile-rasterization-analysis

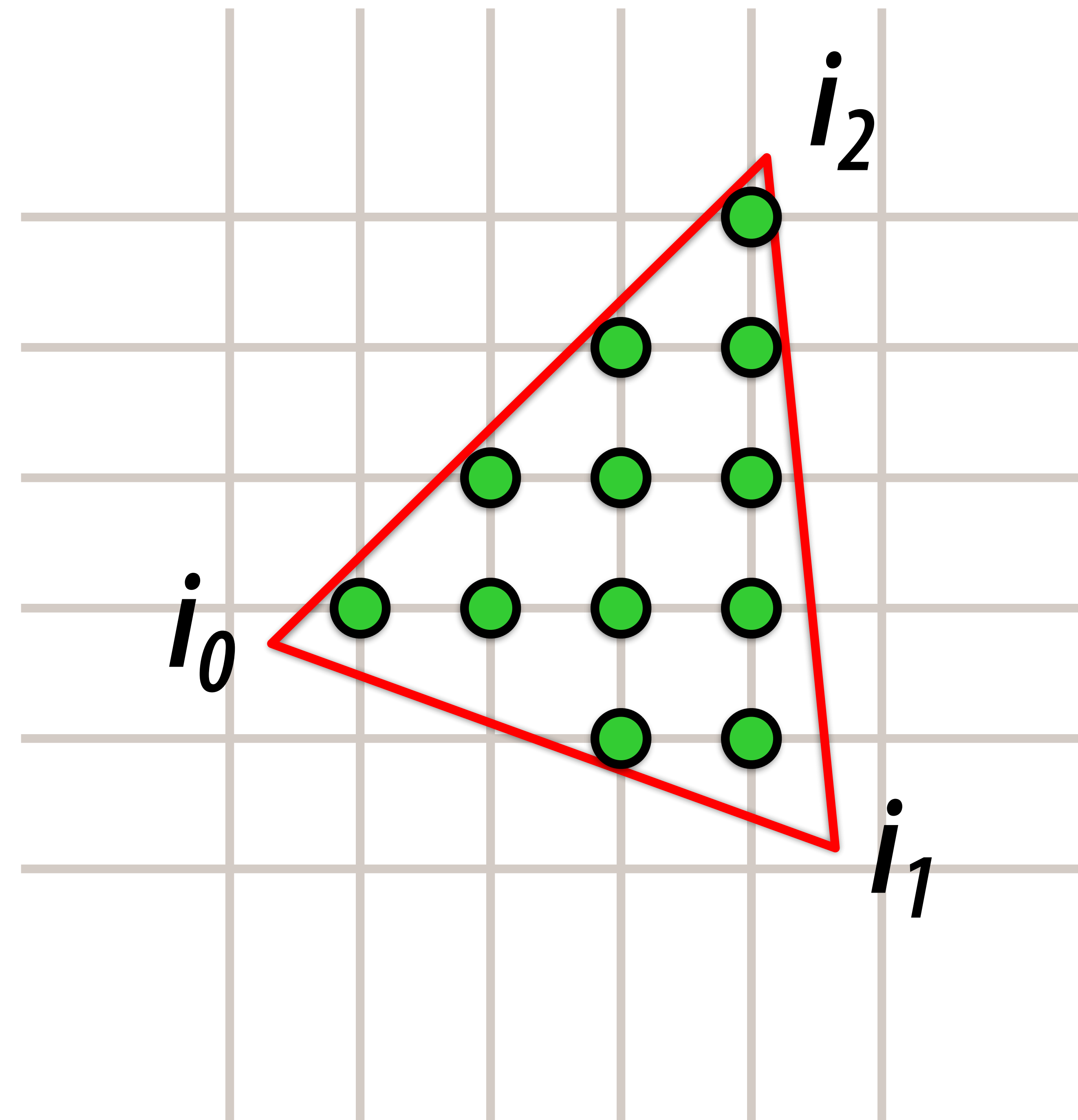
Other notes on modern rasterizers
(oral)

Outline

- Rasterization basics
- Pixel coverage
- Parameter determination
- Perspective correction
- Implementations

Parameter Determination

- Each triangle has several "interpolants"
- For each interpolant, what is the value at each pixel location?
- We want linear interpolation for *i*
- Keep in mind *setup cost* and *per-fragment* cost



Scanline Algorithm

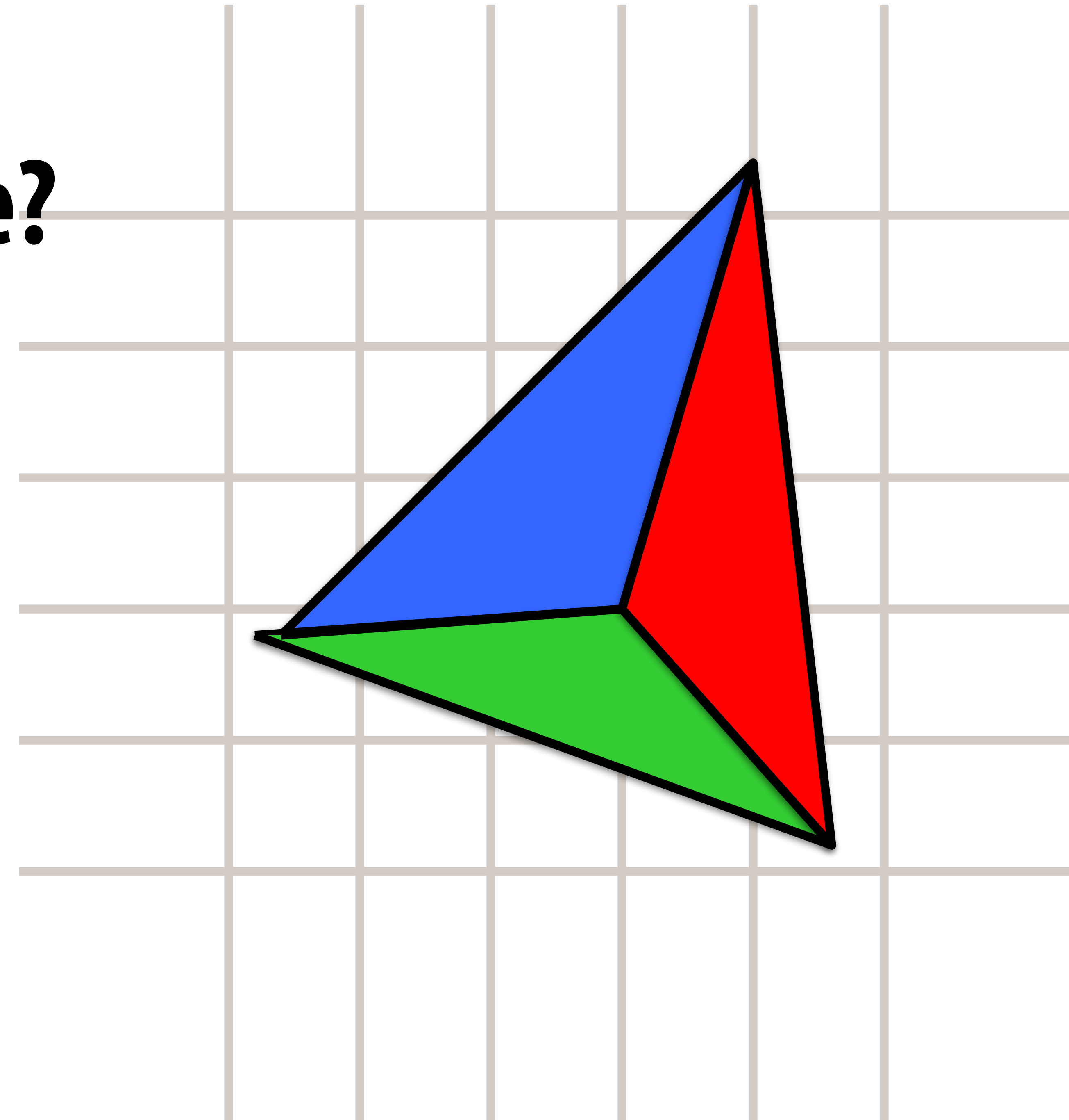
- Big picture: Interpolation at point == interp. along edges • interp. along scanline
- For edge e, given (*x0*, *y0*), (*x1*, *y1*):
  - Calculate dx/dy and every di/dx, di/dy
- Start at bottom (smallest *y*), calculate starting *x* and finishing *x* ("span") and *i0*
- Snap interpolant values to integers
- Rasterize span from xstart to xend
  - At each step, *i* += di/dx
  - Increment *y*, *x* += dx/dy, all *i0* += di/dy.
- Must handle switch in active edge for tri

Scanline Algorithm

- Merges pixel coverage/param determination
- Advantages:
  - Efficient computationally
    - Uses incremental computation
    - Little wasted work
  - Good for large triangles
- Disadvantages
  - Complex control
  - Lots of overhead for small triangles

Barycentric Formulation

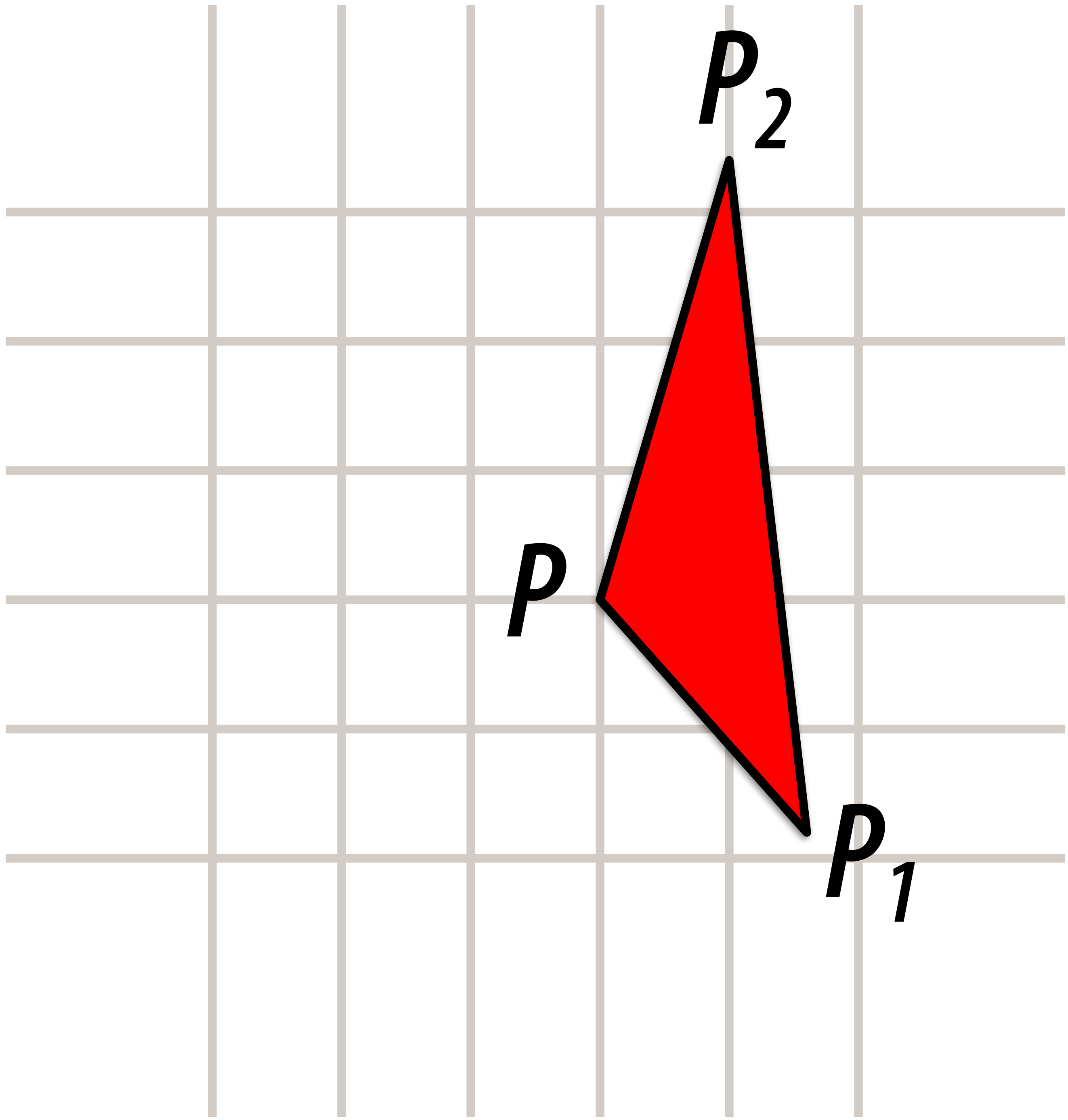
- a_0: area of red triangle; a_1, blue; etc.
- $b_0 = a_0/(a_0+a_1+a_2)$; etc.
  - Have we calculated $a_0+a_1+a_2$ before?
- $i = b_0 i_0 + b_1 i_1 + b_2 i_2$
- a_n as function of x, y?



Barycentric Formulation

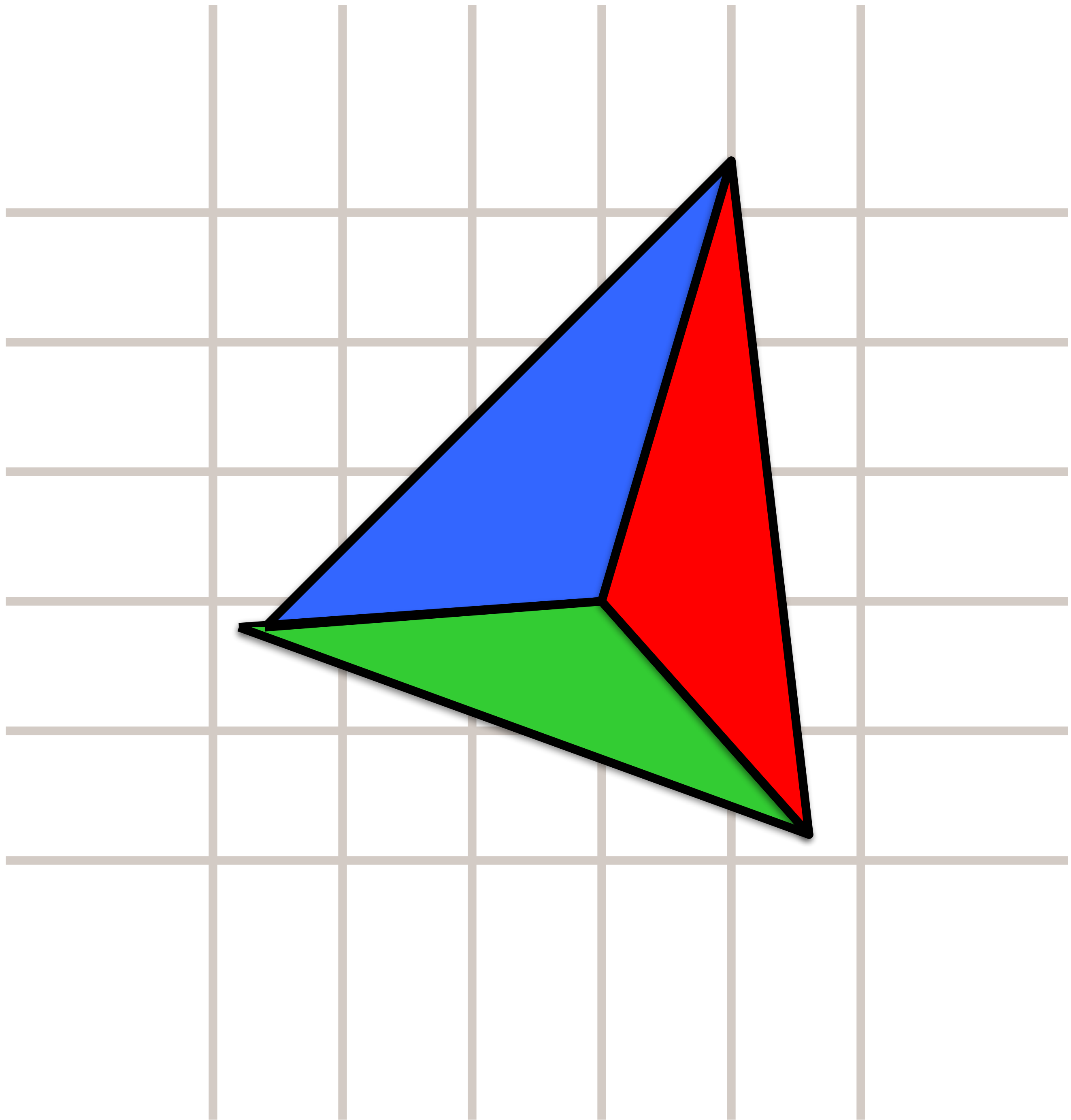
- What is area as fn(P=(x,y))?
- $a_0 = 1/2\ |PP_1 \times PP_2|$
- $= 1/2 \begin{vmatrix} x_1-x & y_1-y \\ x_2-x & y_2-y \end{vmatrix}$
- $= (y_1-y_2)x + (x_2-x_1)y + (x_1y_2-x_2y_1)$
- $= \alpha_0 x + \beta_0 y + \gamma_0$

- Must calculate a, a_0, a_1, a_2



Barycentric Formulation

- a_0: area of red triangle; a_1, blue; etc.
- $b_0 = a_0/(a_0+a_1+a_2)$; etc.
- $i = b_0 i_0 + b_1 i_1 + b_2 i_2$ $(b \cdot i)$
- a_n as function of x, y?
- $a_0 = \alpha_0 x + \beta_0 y + \gamma_0$
- So areas can be calculated linearly too!

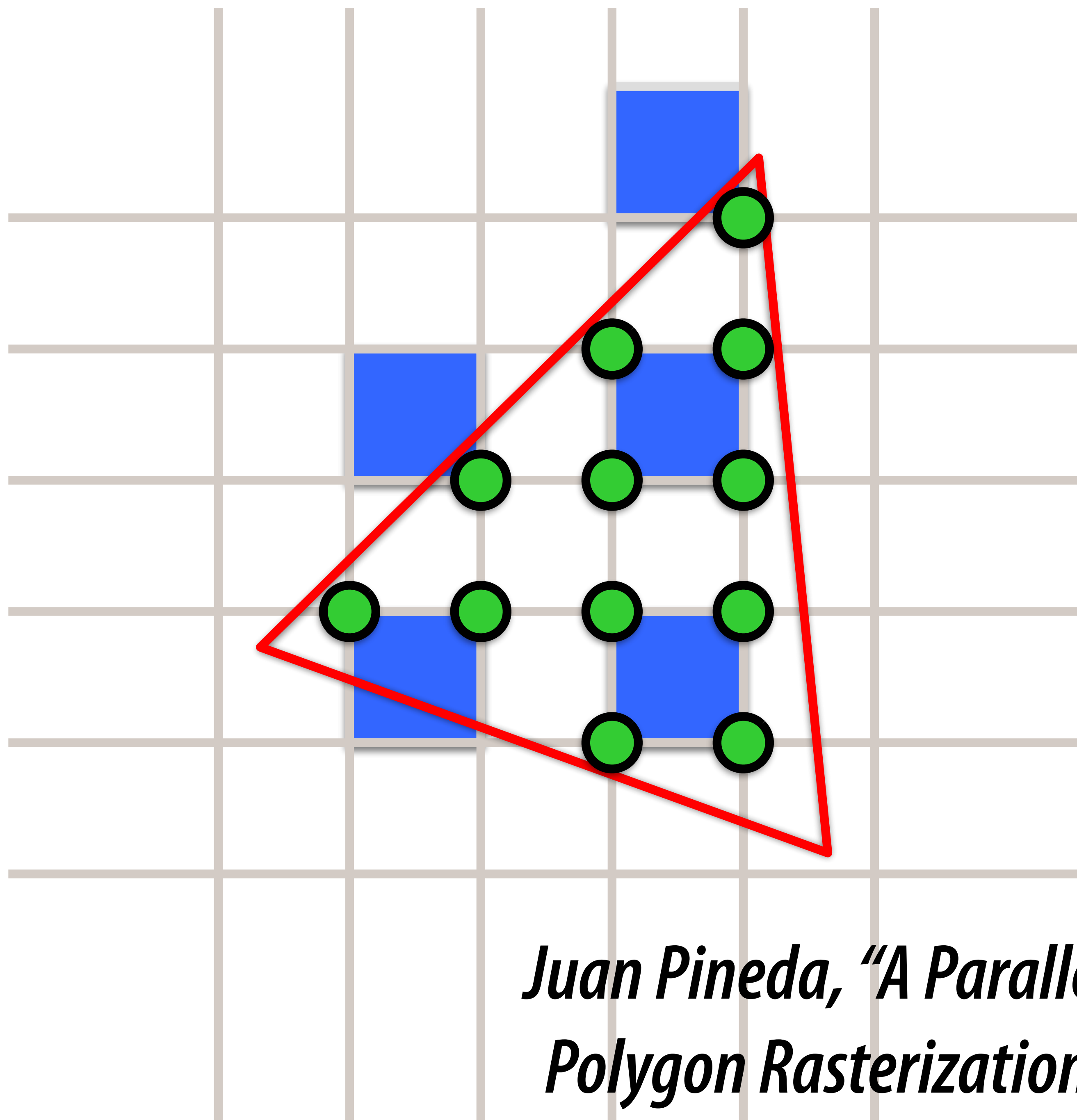


Building a Rasterizer

- Triangle setup (work per triangle)
- Scanline:
  - Edge equations (one divide/edge)
  - Find smallest *y* / sort triangle
- Barycentric:
  - Calculate A, α_n , β_n , γ_n
- Fragment generation (work per fragment)
  - Scanline: incremental evaluation
  - Barycentric: incrementally calculate b_n, dot product to find interpolants

Pineda Rasterization

- Rasterize multiple pixels at once—deltas are all linear
- Edge equations used as masks



Juan Pineda, "A Parallel Algorithm for Polygon Rasterization", Siggraph '88

GeForce Principles II

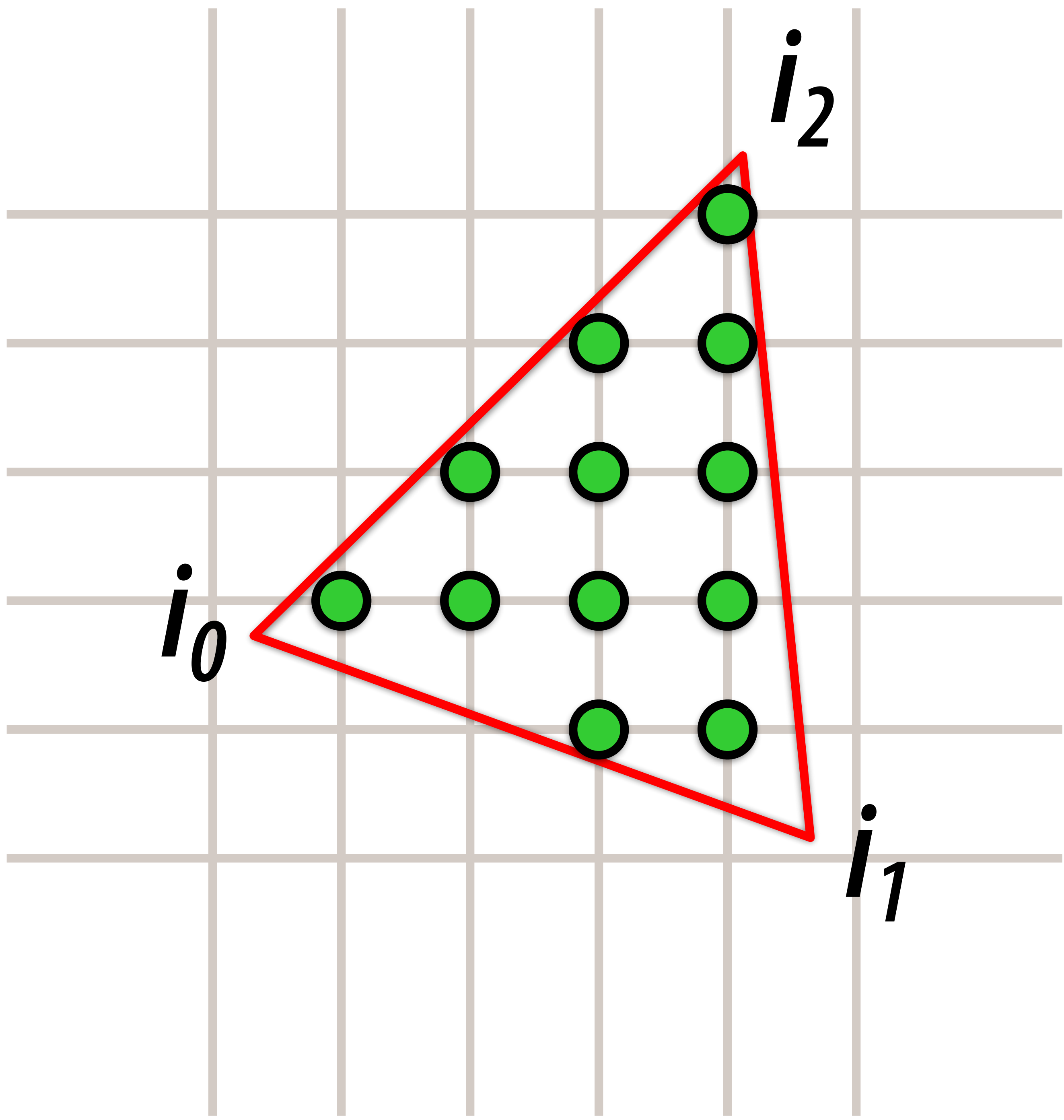
- Try to load balance the pixel engine and the vertex engine
  - Pixels are stamped out in 2x2 blocks per clock
    - Or in 2x1, or in 1x1 blocks when using heavy MultiTexture
  - Vertices are cached before TnL, after TnL and in the primitive assembly area
    - Effective use of these caches is critical for best for performance

Outline

- Rasterization basics
- Pixel coverage
- Parameter determination
- Perspective correction
- Implementations

Parameter Determination

- Each triangle has several "interpolants"
- For each interpolant, what is the value at each pixel location?
- We want linear interpolation for *i* (in object space)

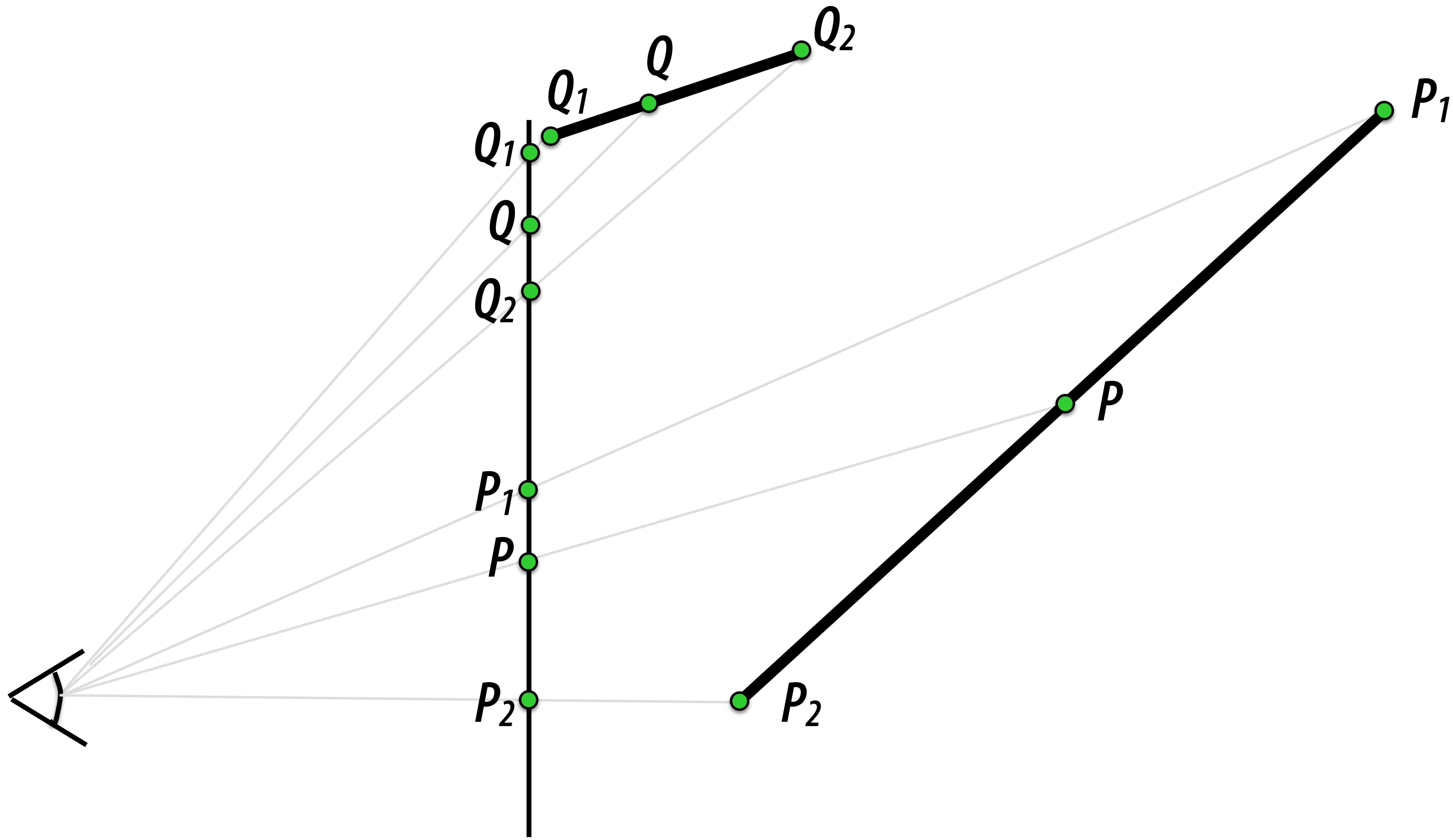


Perspective Correct Interpolation

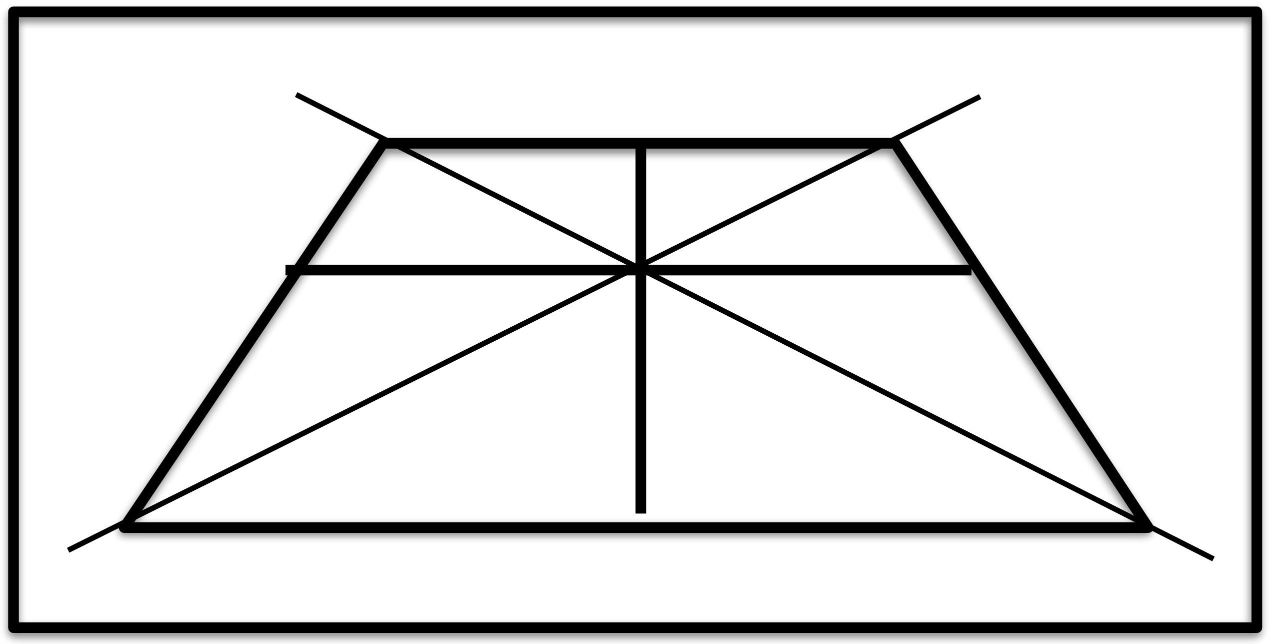
- Up to this point, we've considered linear interpolation in screen space for interpolants.

- But we can't interpolate *u*, *v* linearly. Why?

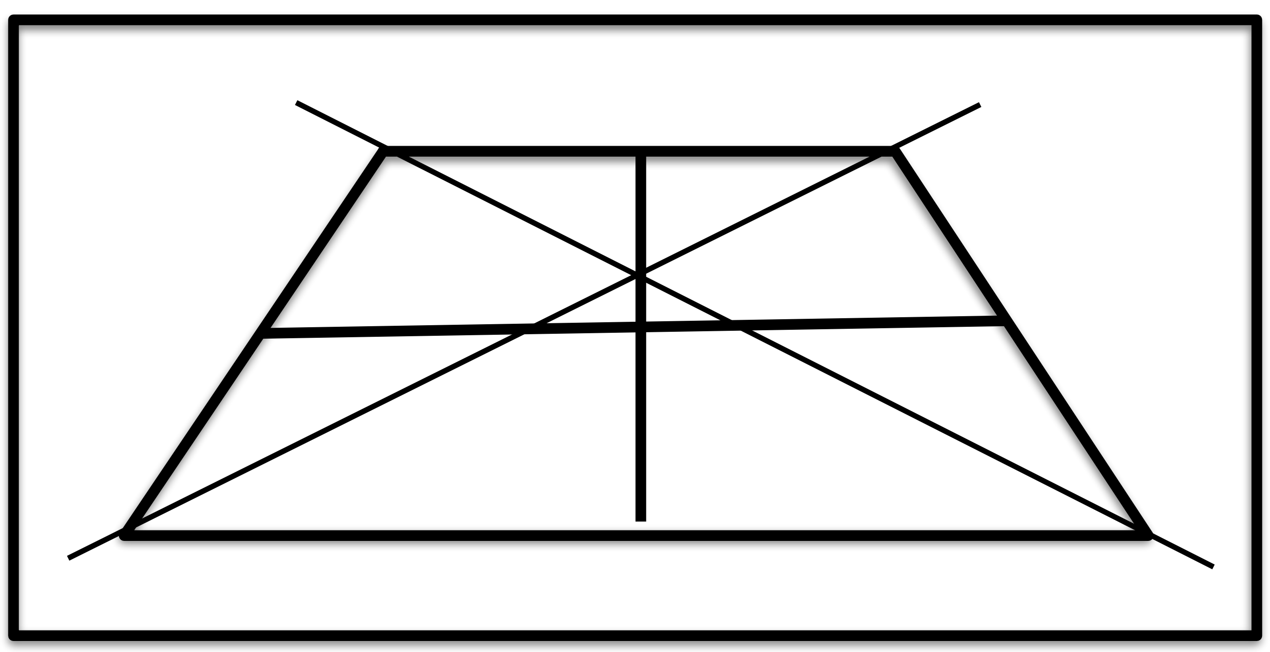
Projection to straight lines



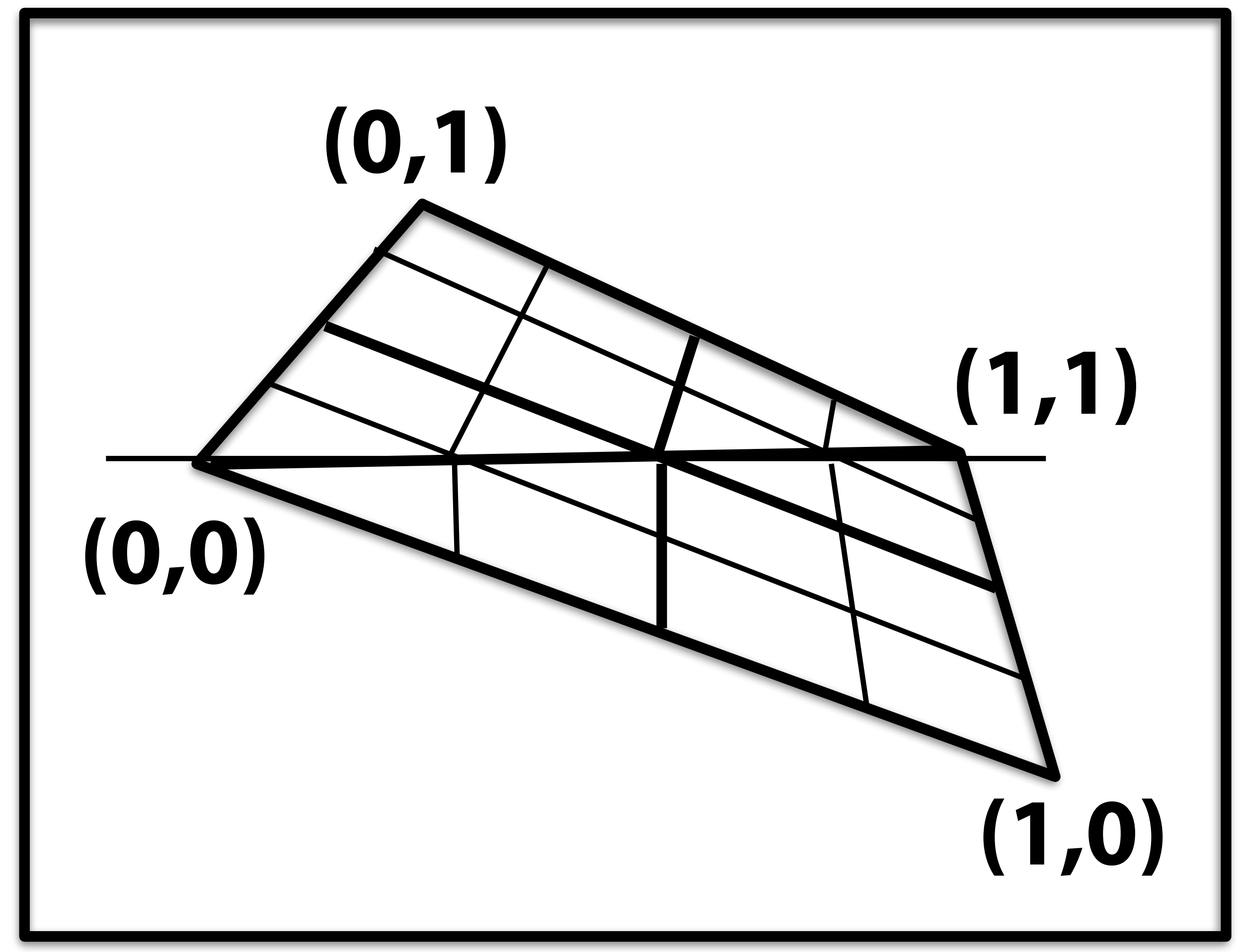
Linear Perspective



Correct Linear Perspective



Incorrect Perspective



*Linear Interpolation**, Bad***
*Perspective Interpolation**, Good***

[Akeley and Hanrahan]

Perspective Correct Interpolation

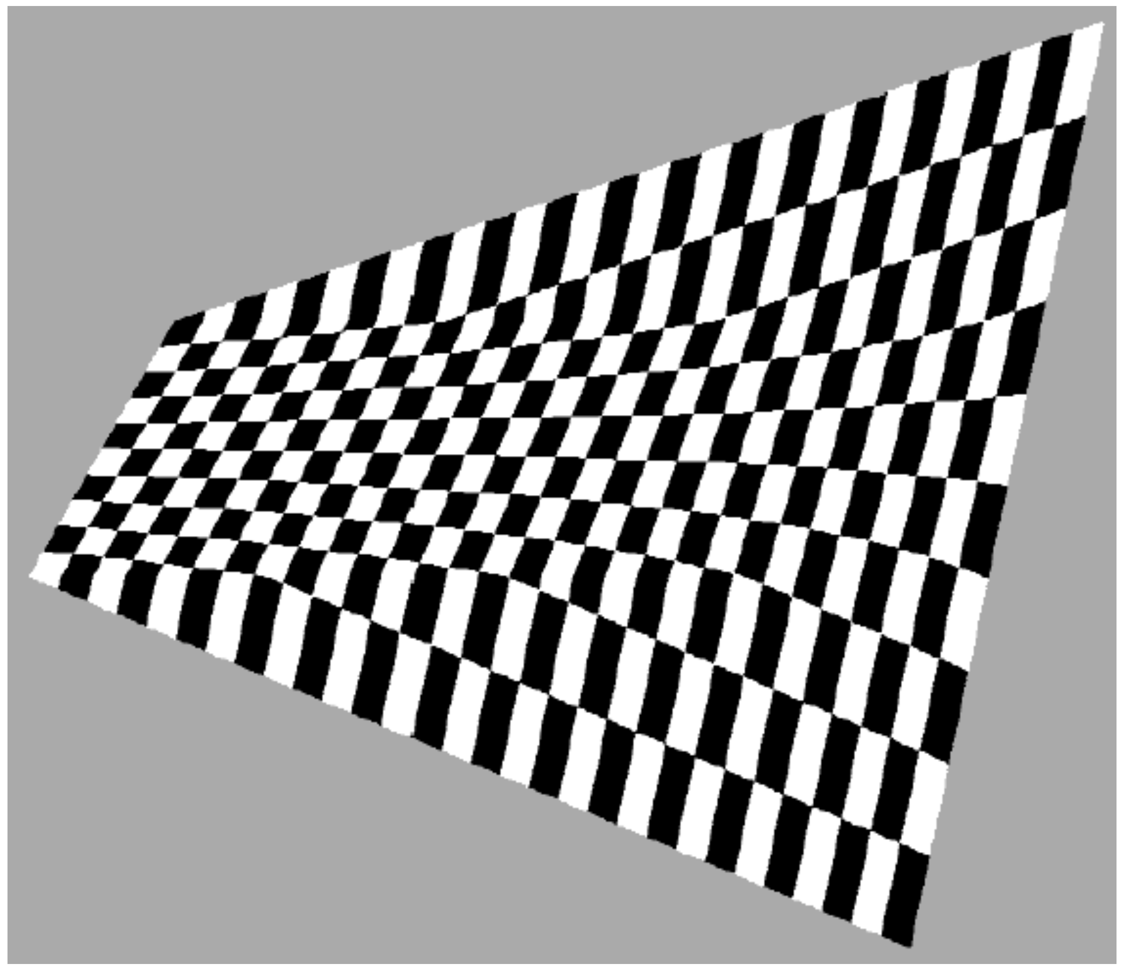
- **Terminology:**
- **Eye space: x_e, y_e**
- **Screen space before divide: x_s, y_s, w_s**
- **Pixel coordinates: $x = x_s/w_s$, $y = y_s/w_s$**
- **Want $u,v = f(x,y)$**
- **Recall we used to do affine mapping of $u(x, y) = \alpha_0 x + \beta_0 y + \gamma_0$**
- **Assume affine mapping between u,v and x_e,y_e**
- **Could do matrix transform for each pixel**

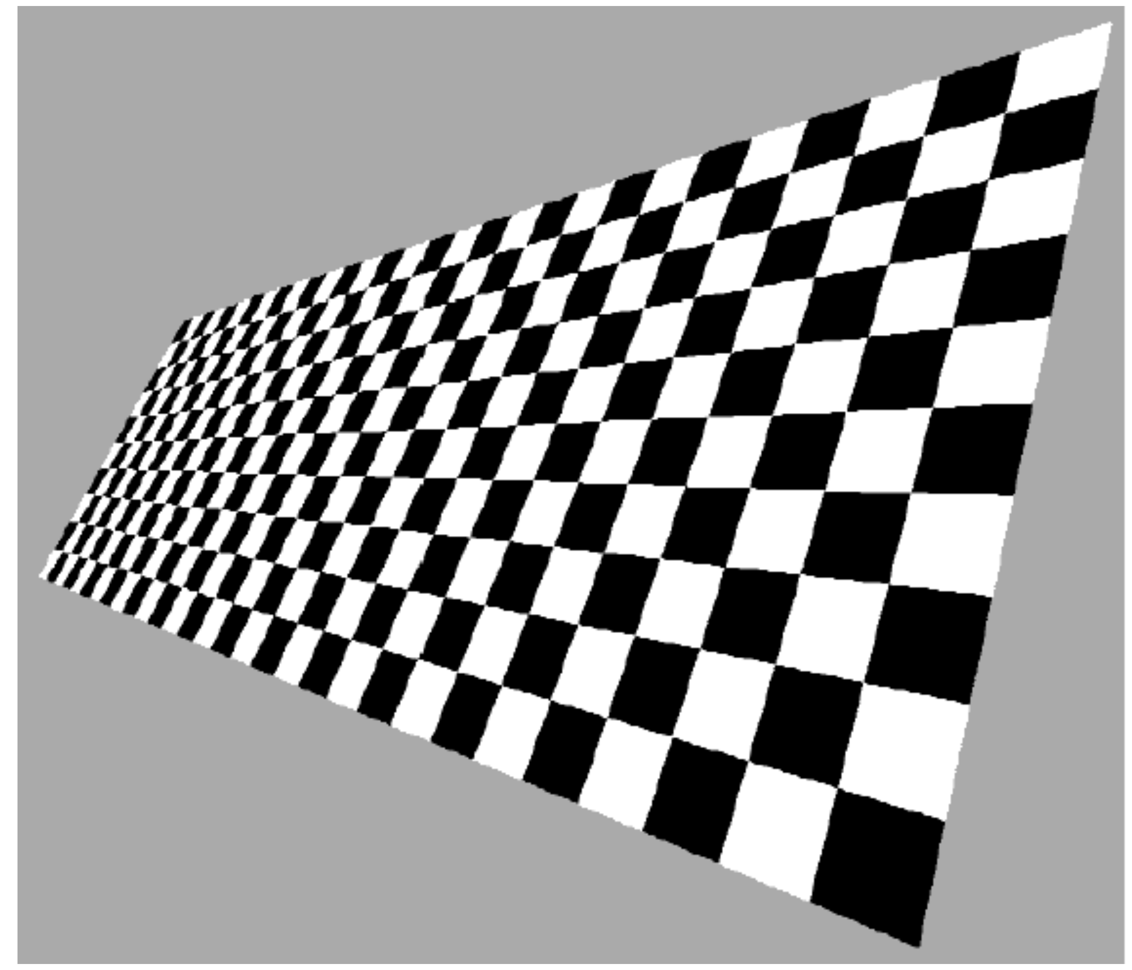
- **Blackboard!**

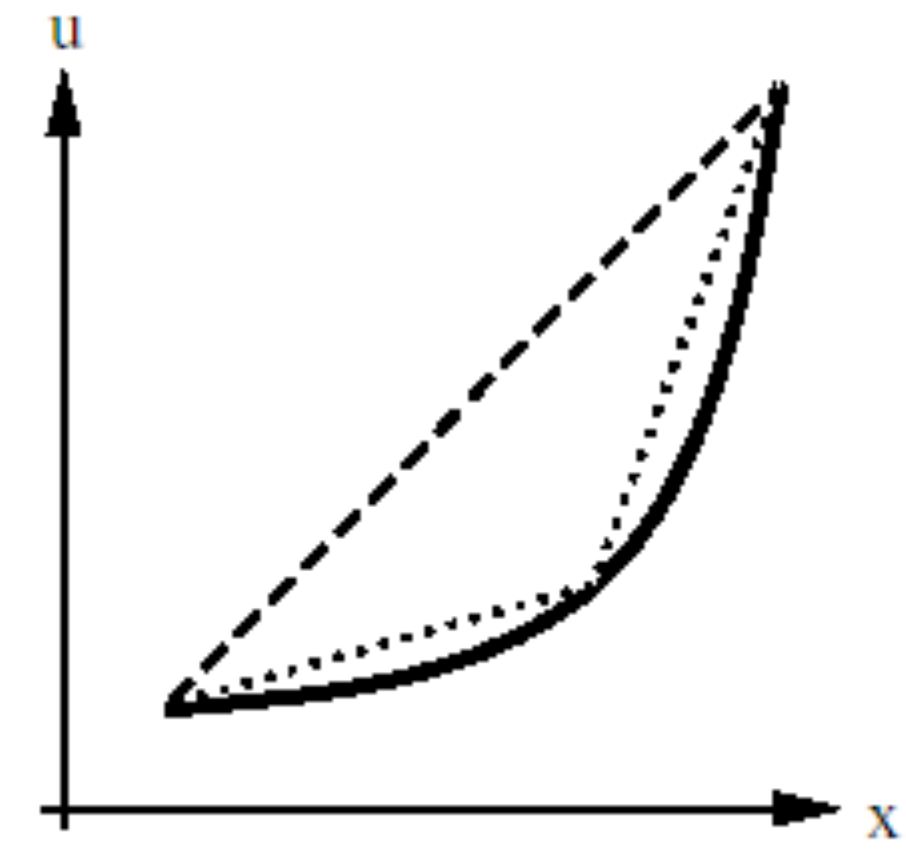
Perspective Correct Interpolation

- We can't interpolate u, v linearly.
- But we can interpolate u/w linearly and $1/w$ linearly.
- So instead of interpolating u, v, etc. ...
- We interpolate u/w, v/w, $1/w$, etc. ...
- Requires divide per vertex for setup, divide per pixel during rasterization.
- How about *rgb*, z, normals, etc.?

Alternatives







[Heckbert and Moreton]

Perspective Correct Interpolation

- **Scanline:**
  - Interpolate 1/*w* and *i*/*w*
  - At each pixel, take 1/(1/*w*) and multiply for each interpolant
- **Barycentric:**
  - $b_0 = a_0/(a_0+a_1+a_2)$ becomes
  - $b_0 = w_1w_2a_0/(w_1w_2a_0+w_2w_0a_1+w_0w_1a_2)$
  - Derivatives easily computable too

Perspective Correct Interpolation

- What if *w* was constant across a scanline?

- (Blackboard, part 2)

Doom (first full-screen textured sw renderer)

Outline

- Rasterization basics
- Pixel coverage
- Parameter determination
- Perspective correction
- Implementations (all slides, Kurt Akeley)

Triangle Rasterization Examples

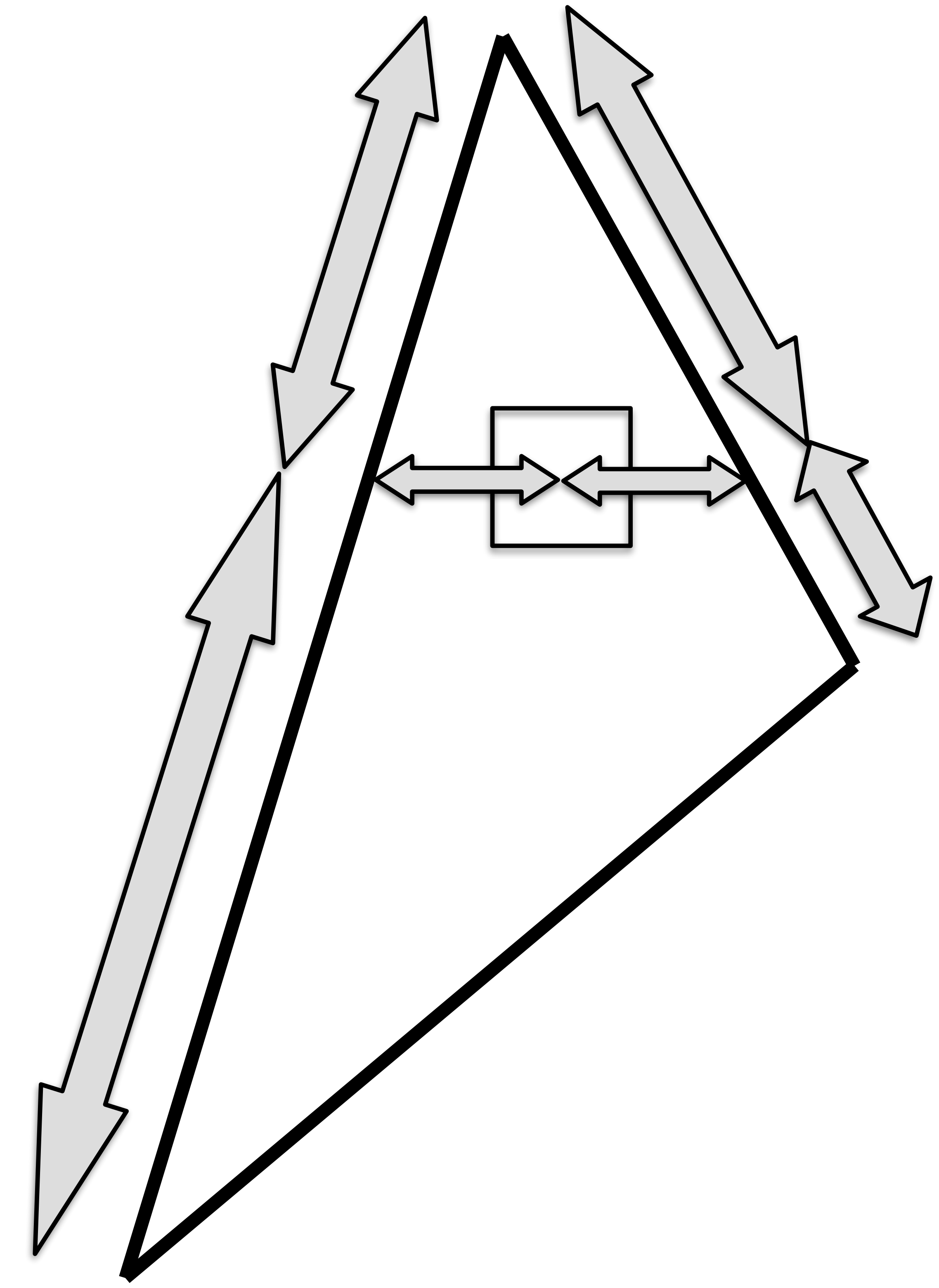
- Gouraud shaded (GTX)
- Per-pixel evaluation (Pixel Planes 4)
- Edge walk, planar parameter (VGX)
- Barycentric direct evaluation (InfiniteReality)
- Small tiles (Bali—proposed)
- Homogeneous recursive descent (NVIDIA)

Algorithm properties

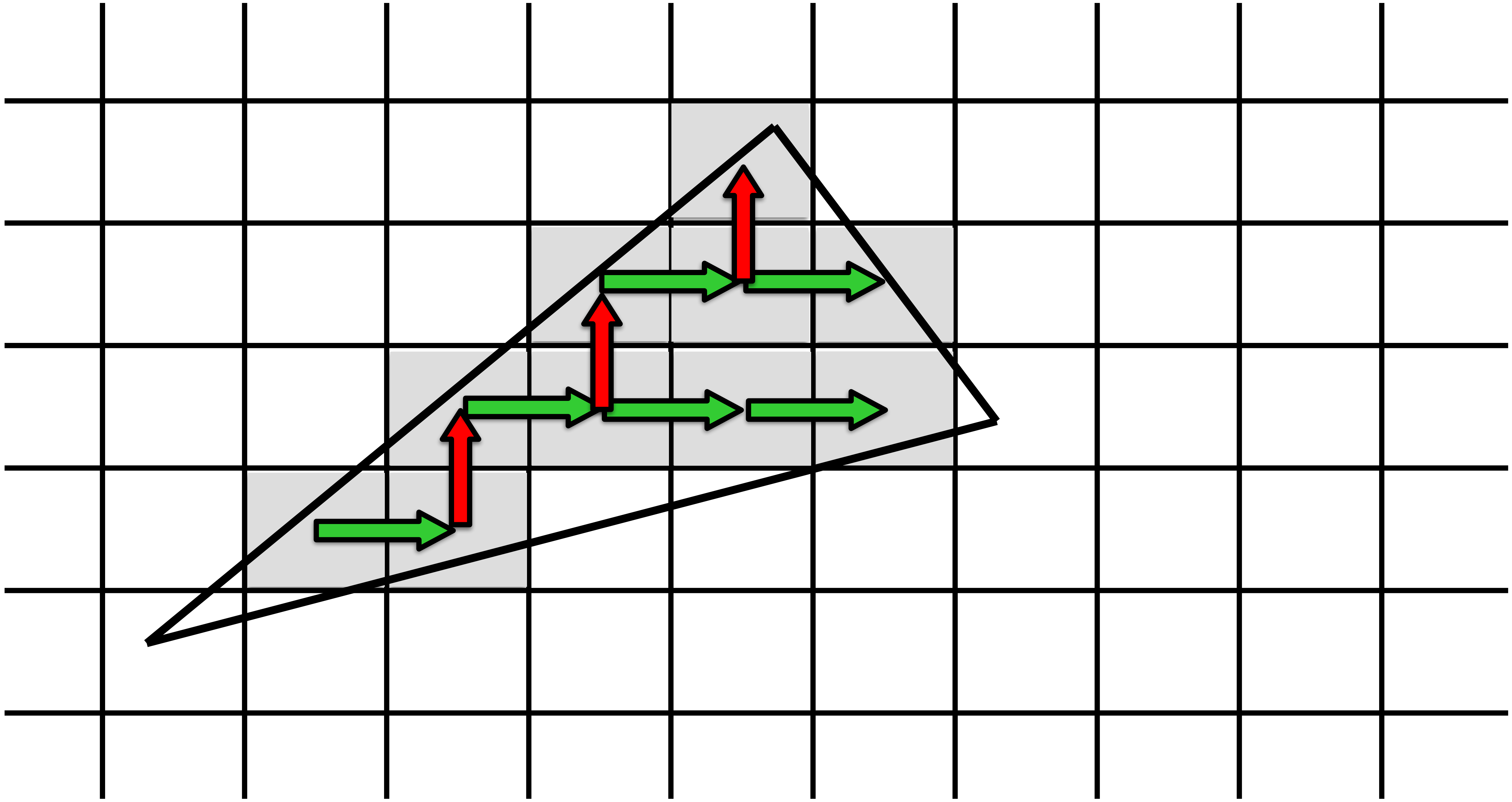
- Setup and execution cost
  - Per-triangle
  - Per-fragment
- Ability to parallelize
- Ability to cull to a rectangular screen region
  - To support tiling
  - To support “scissoring”

Gouraud shaded (GTX)

- Two stage algorithm
  - DDA edge walk
    - fragment selection
    - parameter assignment
  - DDA scan-line walk
    - parameter assignment only
- Requires expensive scan-line setup
  - Location of first sample is non-unit distance from edge
- Cannot scissor efficiently
- Works on quadrilaterals

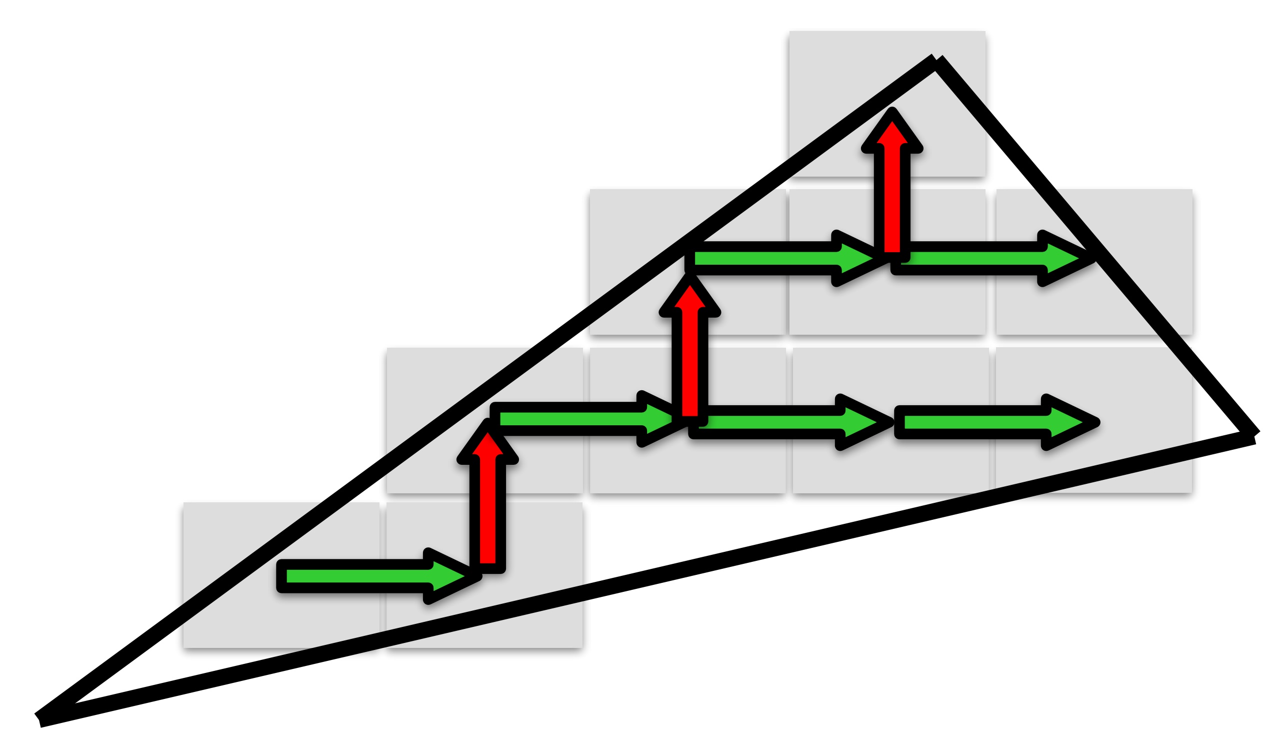


Edge walk, planar evaluation (VGX)

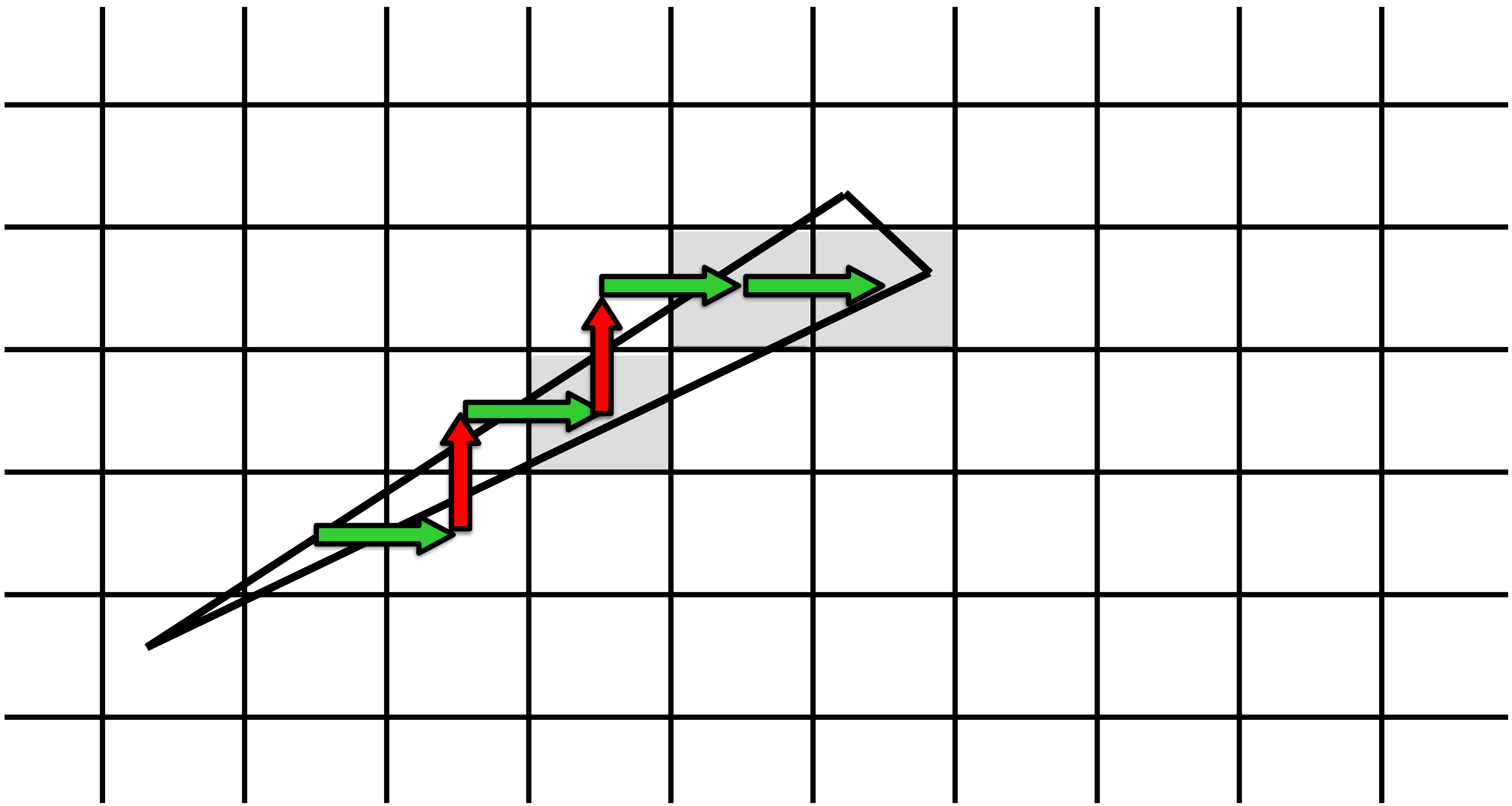


Edge walk, planar evaluation (VGX)

- Hybrid algorithm
  - Edge DDA walk for fragment selection
    - Efficient generation of conservative fragment set
  - Sample DDA walk for parameter assignment
    - Never step off sample grid, so
    - Never have to make sub-pixel adjustment
- Scissor cull possible
  - Adds complexity to edge walk
- Sample walk simplifies parallelism



Interpolation outside the triangle



Barycentric (InfiniteReality)

- Hybrid algorithm
  - Approximate edge walk for fragment selection
    - Pineda edge functions used to generate AA masks
  - Direct barycentric evaluation for parameter assignment
    - Barycentric coordinates are DDA walked on grid
    - Minimizes setup cost
    - Additional computational complexity accepted
    - Handles small triangles well
- Scissor cull implemented
  - Supports "guard band clipping"

Small tiles (Bali—proposed)

- Framebuffer tiled into $n \times n$ (16x16) regions
  - Each tile is owned by a separate engine
- Two separate rasterizations
  - Tile selection (avoid broadcast, conservative)
  - Fragment selection and parameter assignment
- Parallelizes well
- Handles small triangles well
- Scissors well
  - At tile selection stage

Homogeneous recursive descent

- Rasterizes unprojected, unclipped geometry
  - Huge improvement for geometry processing!
  - Interpolates clip-plane distances
- I have heard ~2001 NVIDIA implemented this
  - But is not well documented
  - Read Olano and Greer
    - Parameter assignment precision has many pitfalls — I have heard NVIDIA gave up on this technique for this reason
    - Watch out for infinities!
- Recursive descent
  - Scissors well
  - Drives $n \times n$ (2×2) parallel fragment generation
- Cannot generate perspective-incorrect parameter values

Next Lecture—Texturing