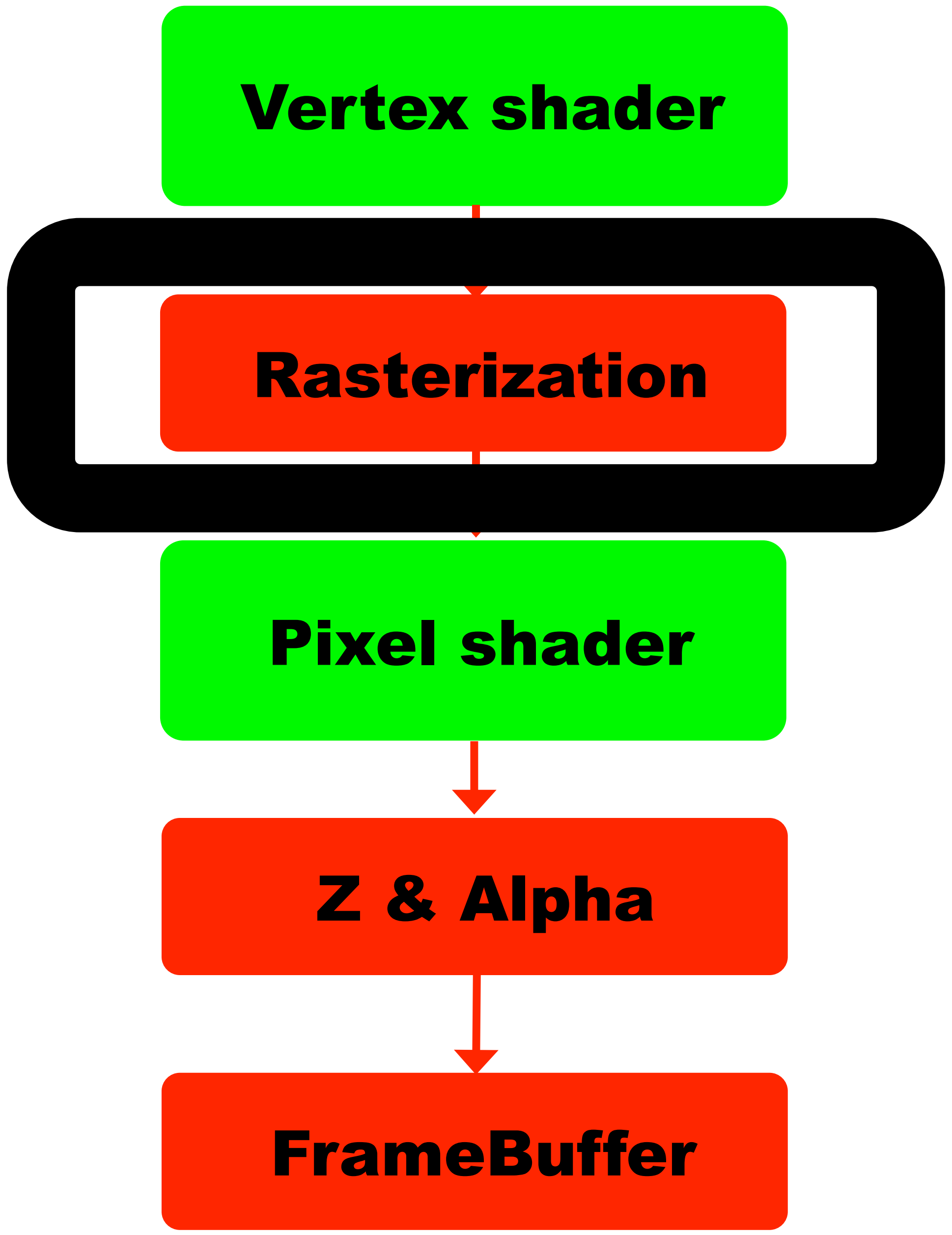
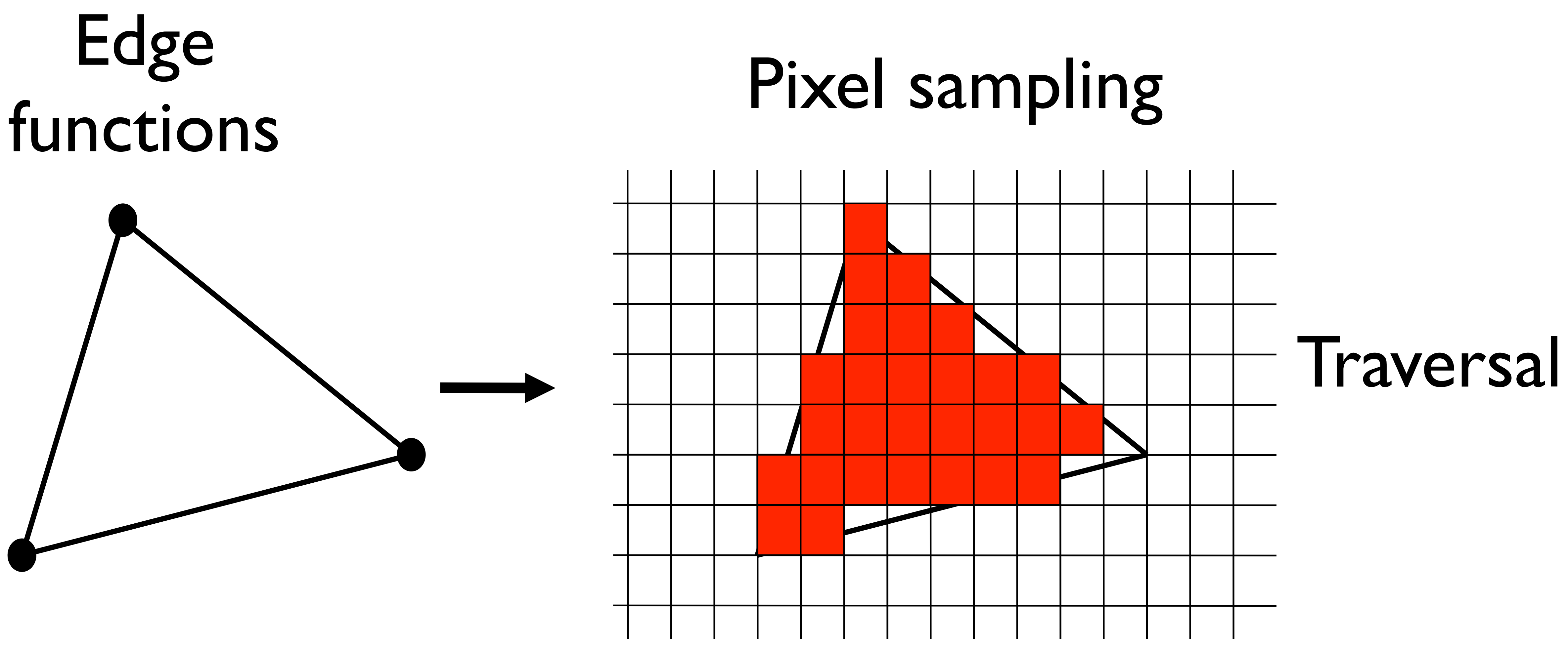
Rasterization

Michael Doggett
Department of Computer Science
Lund university

Today's stage of the Graphics Pipeline

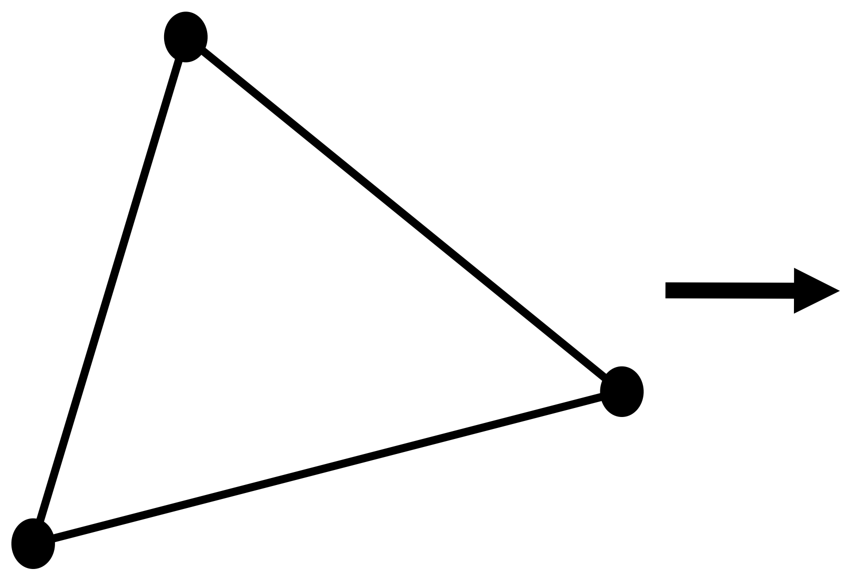


How to rasterize a triangle?

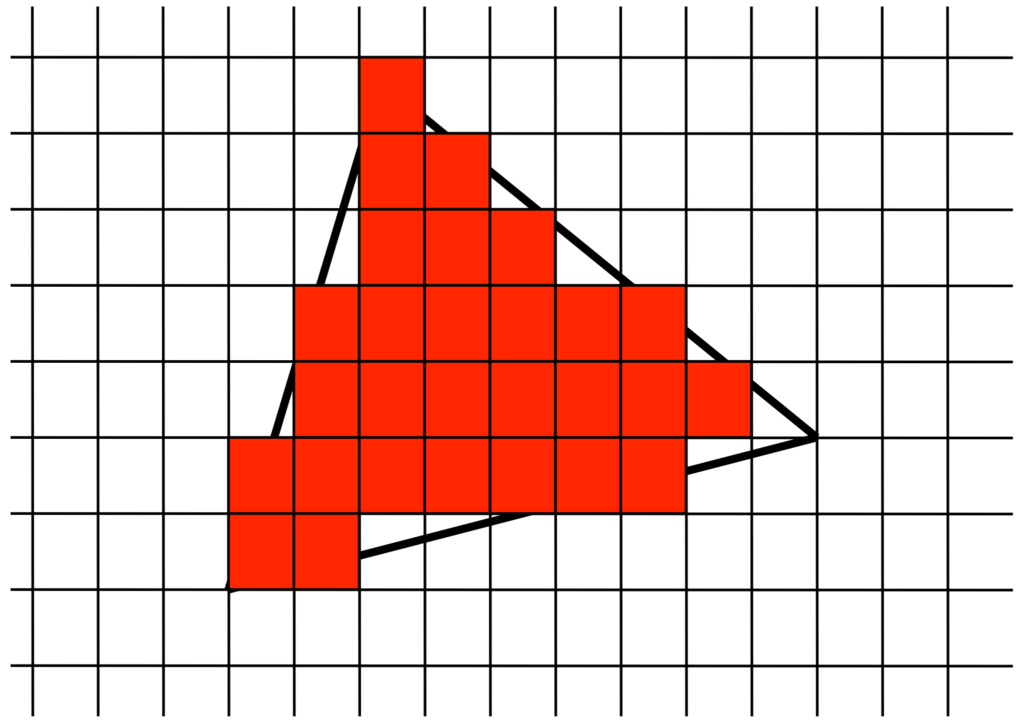


Interpolation

Edge functions



Pixel sampling

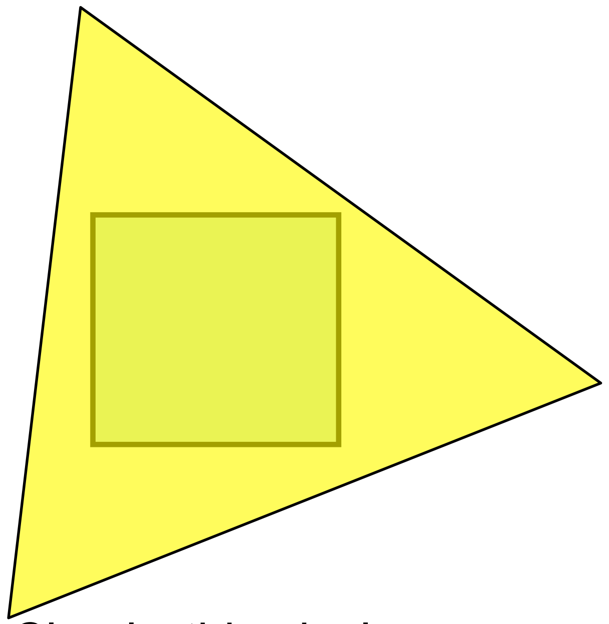


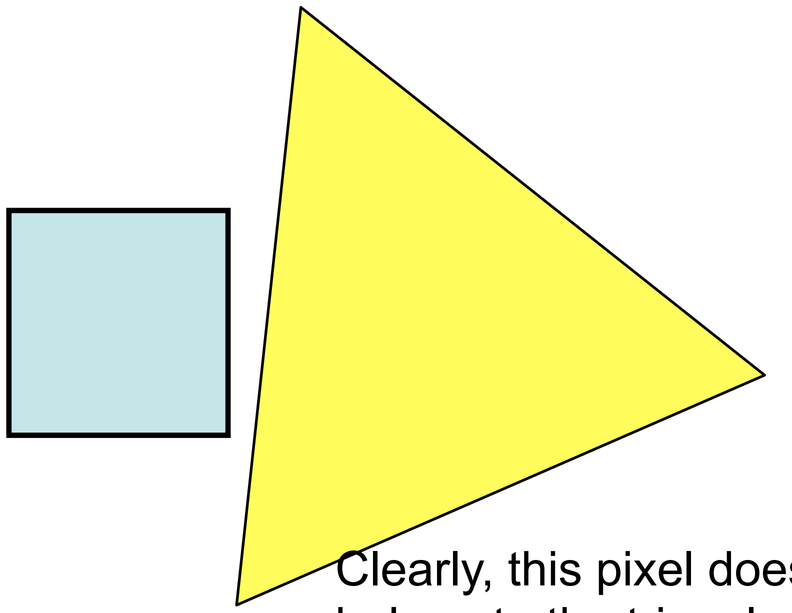
Traversal

Interpolation

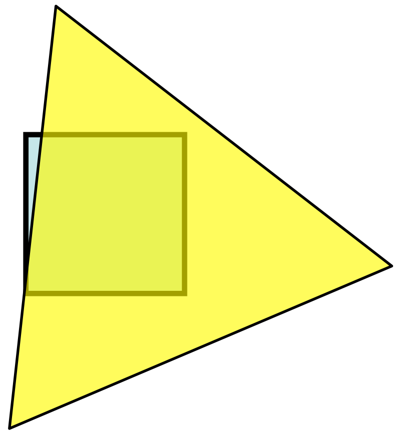
Which pixel is inside a triangle?

- Triangle traversal

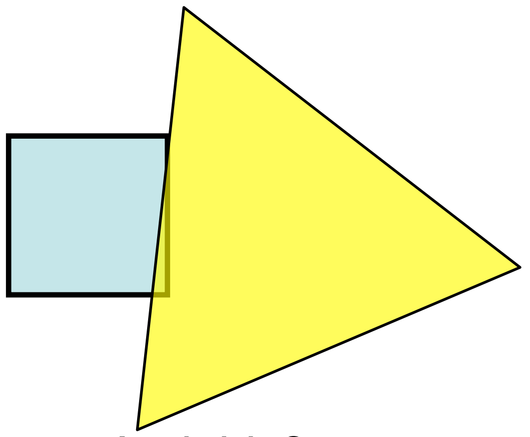

Clearly, this pixel belongs to the triangle


Clearly, this pixel does **NOT** belong to the triangle

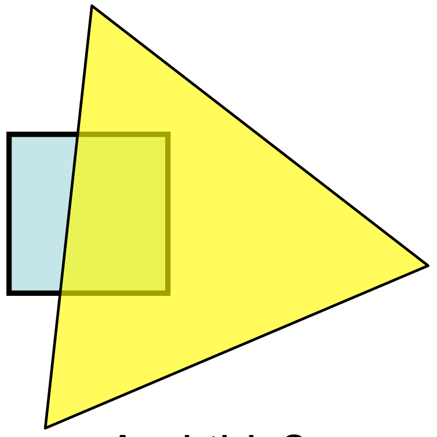
Which pixel is inside a triangle?



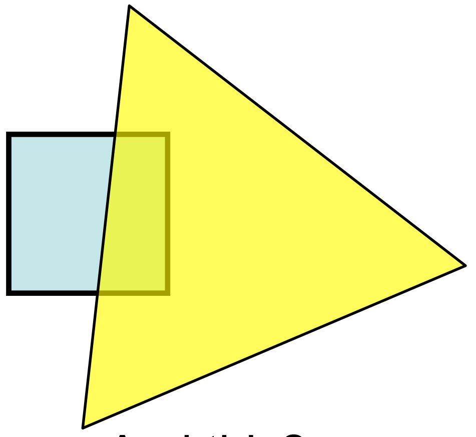
How about this case?



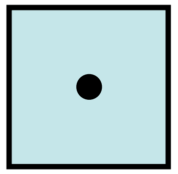
And this?



And this?

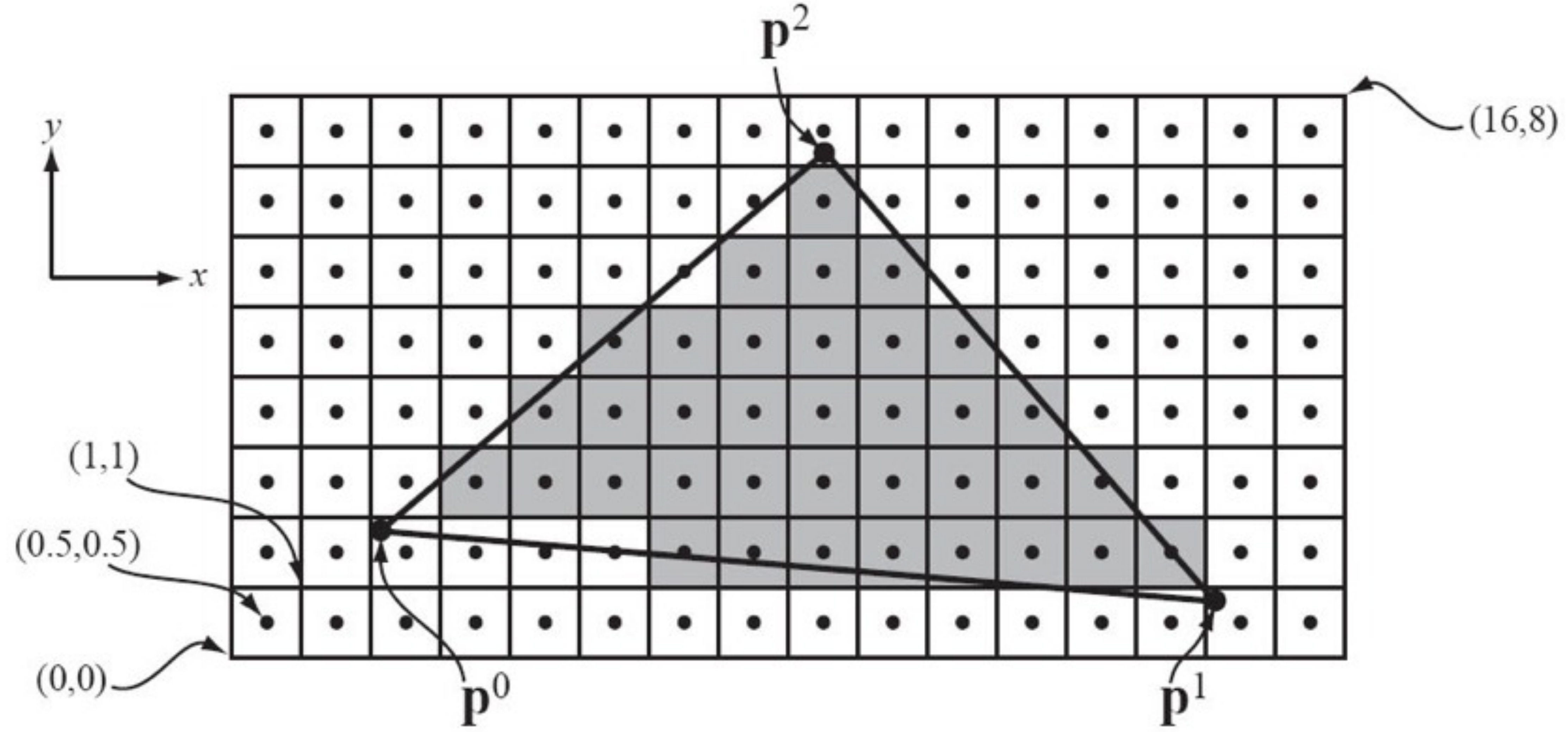


And this?



- **Sample** at the center

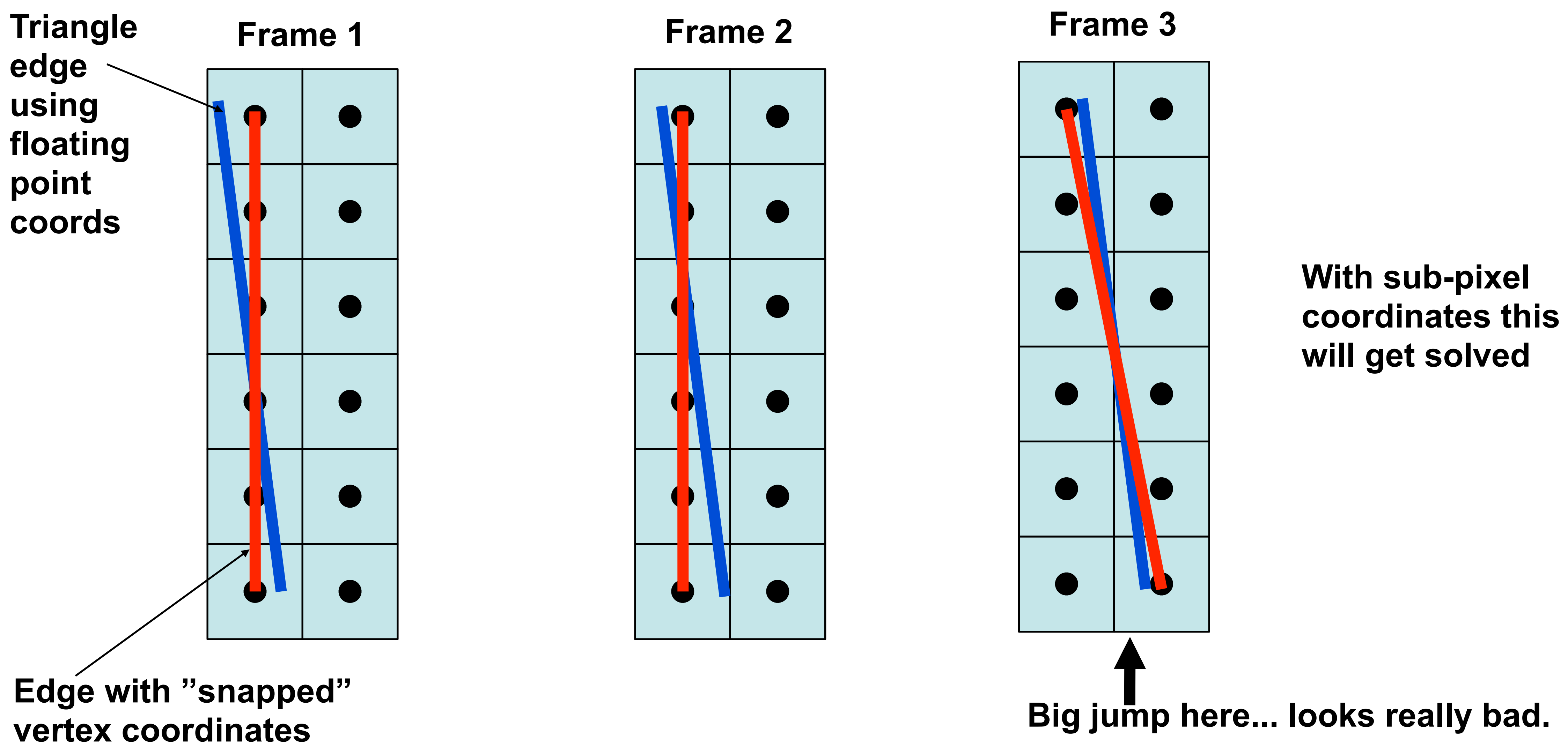
How are we computing pixel center?



Screen space coordinates

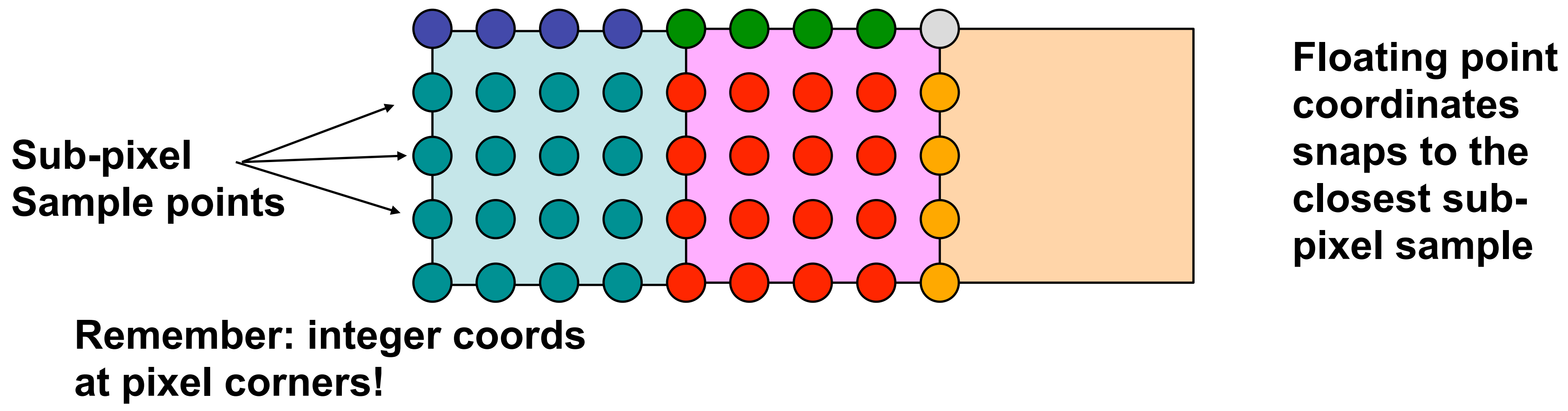
(p_x,p_y) are in $[0,w] \times [0,h]$

What happens if you round off floating point vertices to nearest pixel center?

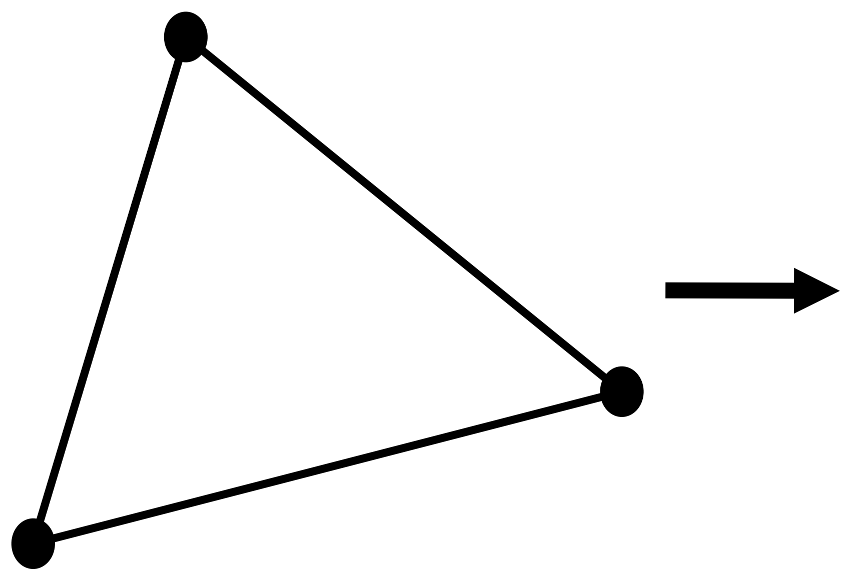

Triangle edge using floating point coords
Frame 1
Frame 2
Frame 3
With sub-pixel coordinates this will get solved
Edge with "snapped" vertex coordinates
Big jump here... looks really bad.

We need sub-pixel coordinates!

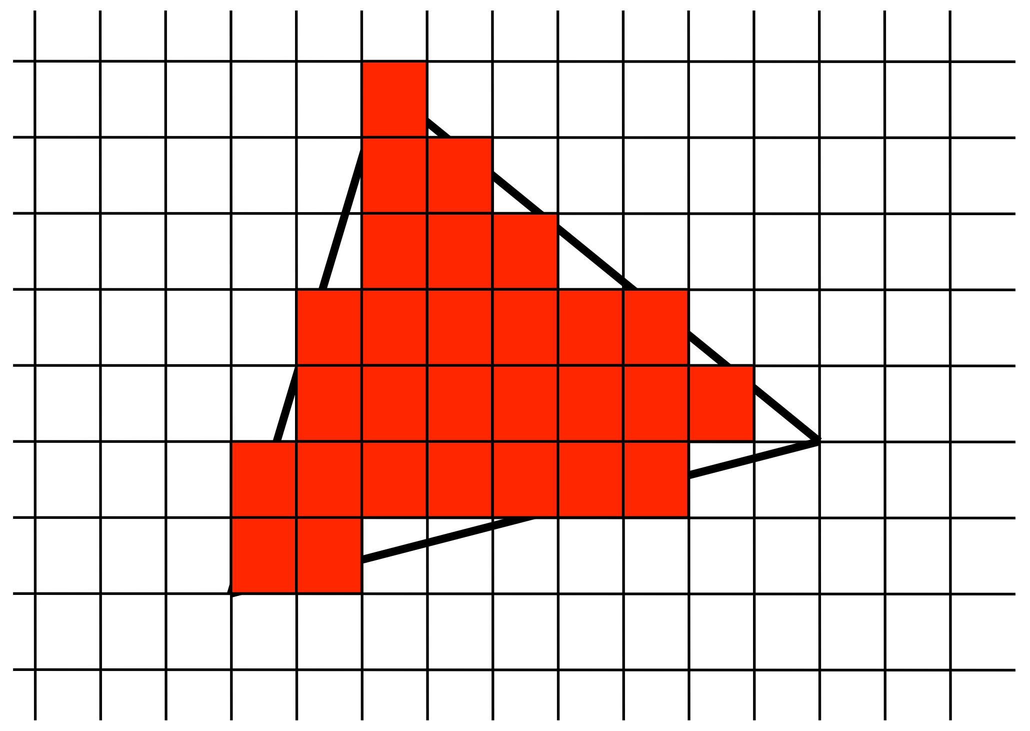
- We can use fixed point math (integer)
- Use 2 sub-pixel fractional bits per x, and y



Edge functions



Pixel sampling



Traversal

Interpolation

How do we determine if a sample is inside a triangle?

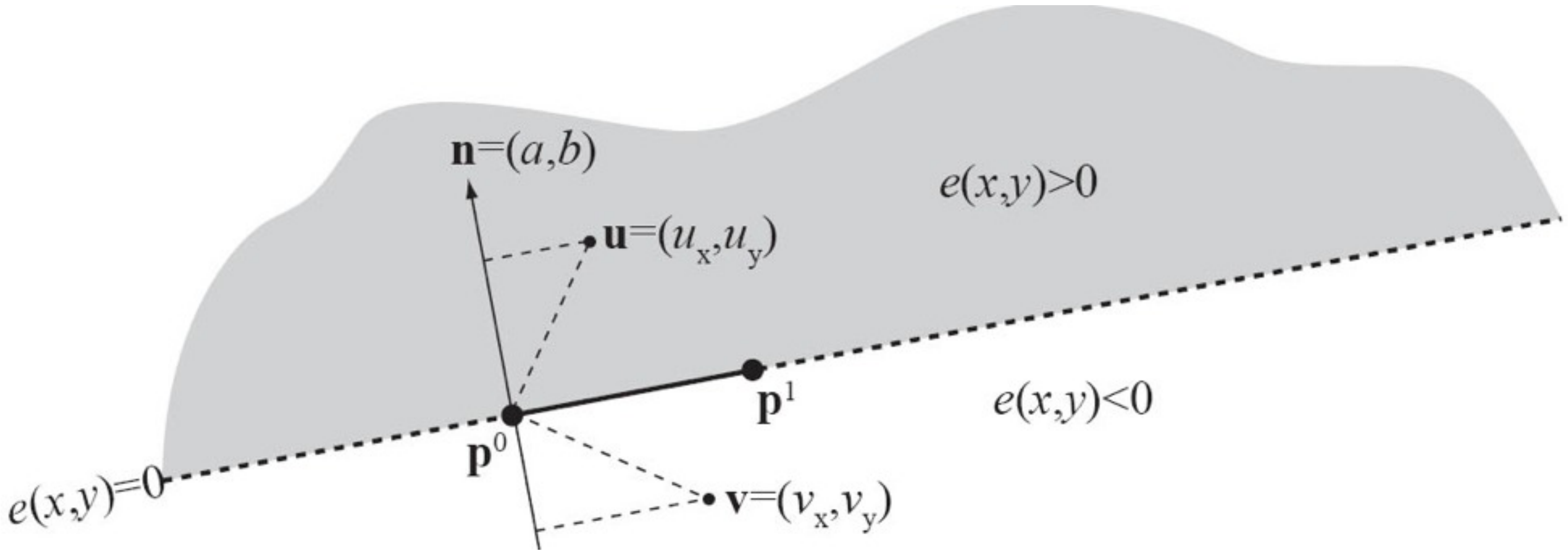
- Convert edges into functions
  - line equation $ax + by + c = 0$
- Edge function for 2 points **p**^{0} and **p**^{1} is:

$$e(x,y) = -(p_y^1 - p_y^0)(x - p_x^0) + (p_x^1 - p_x^0)(y - p_y^0)$$

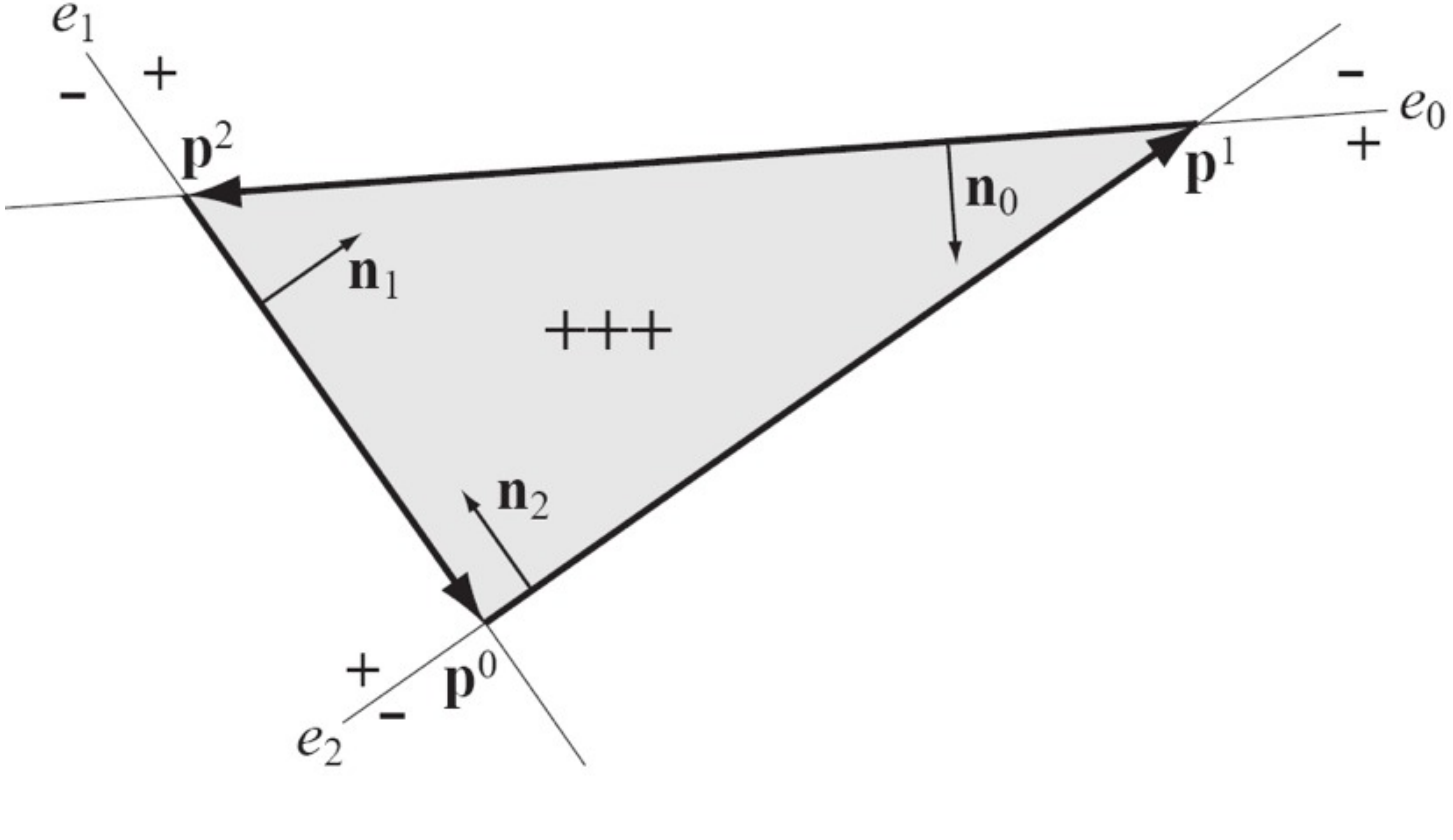
$$e(x,y) = ax + by + c = \mathbf{n} \cdot (x,y) + c.$$

Can be thought of as the 'normal' of the line

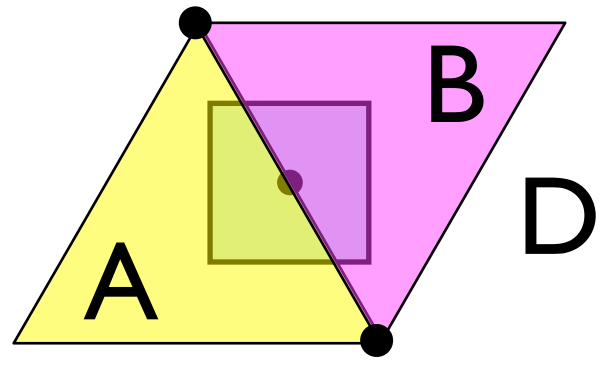
How do points relate to the edge function?



Points are inside if all edge functions are positive!



What happens to pixels exactly on an edge?



Does the pixel belong to A or B, or both ? or neither?

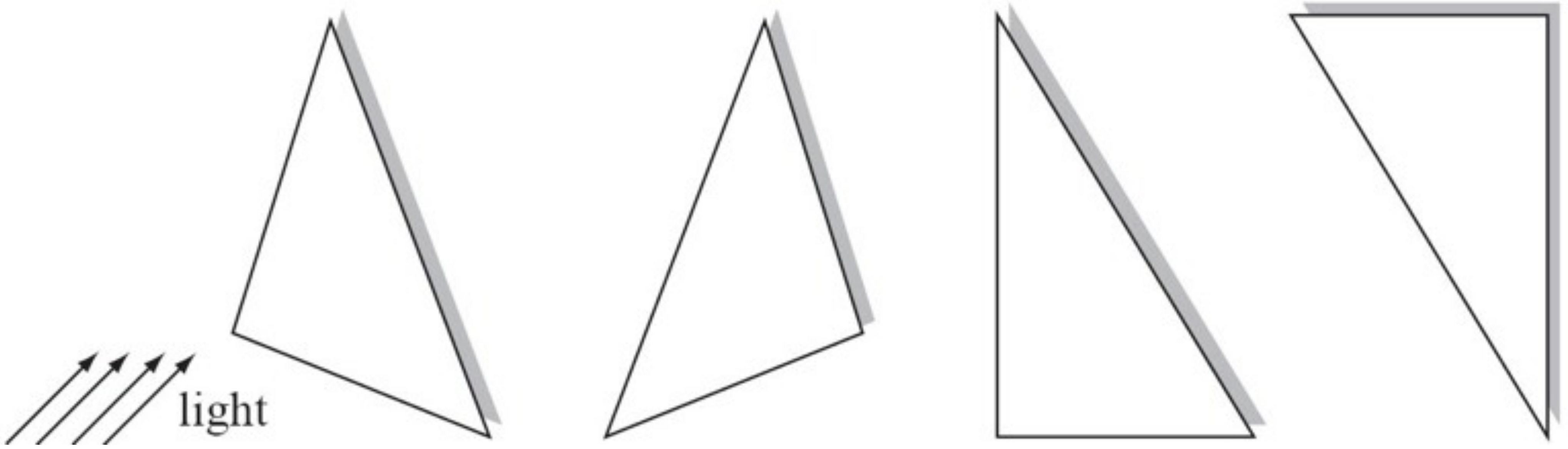
- One and only one of A or B
- Because :
  - No cracks between triangles
  - No overlapping triangles

How to decide which triangle an edge sample is in?

One solution (by McCool el at)

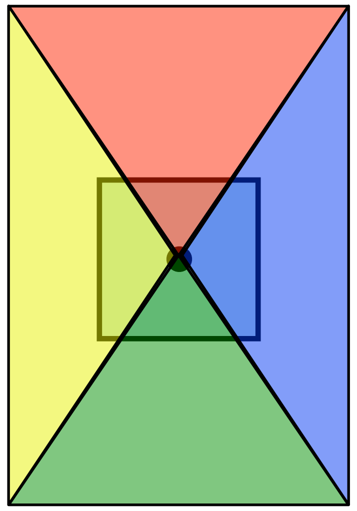
```
bool INSIDE($e, x, y$)
1   if $e(x, y) > 0$ return true;
2   if $e(x, y) < 0$ return false;
3   if $a > 0$ return true;
4   if $a < 0$ return false;
5   if $b > 0$ return true;
6   return false;
```

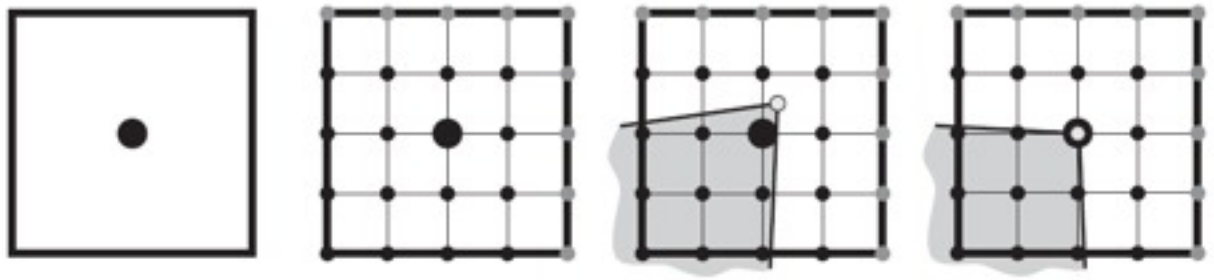
- Another way to think about it:
  - We exclude shadowed edges



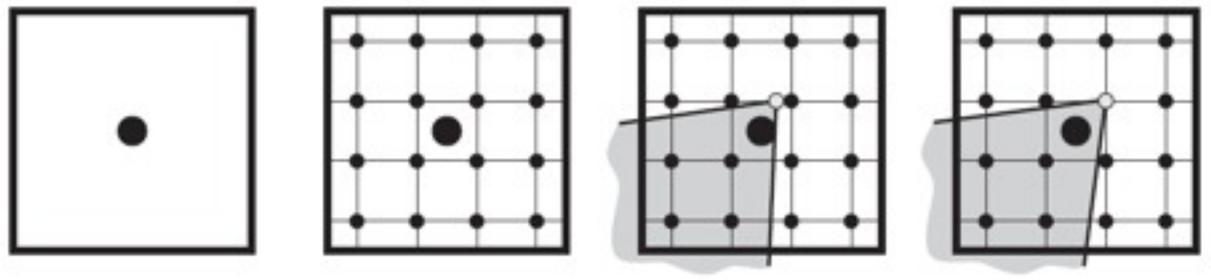
How about when a vertex coincides with the sampling point?

You get the same kind of problems!

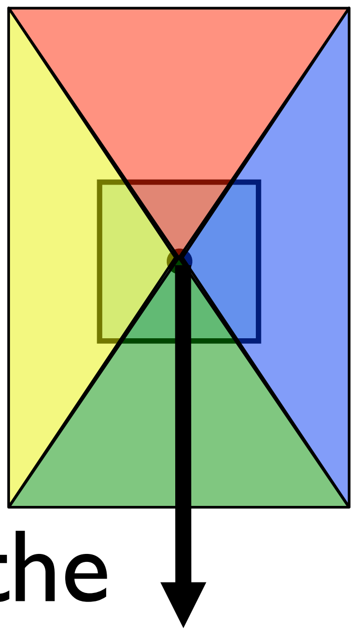




One solution: offset the subpixel grid so that sampling points never coincide with sub-pixel grid



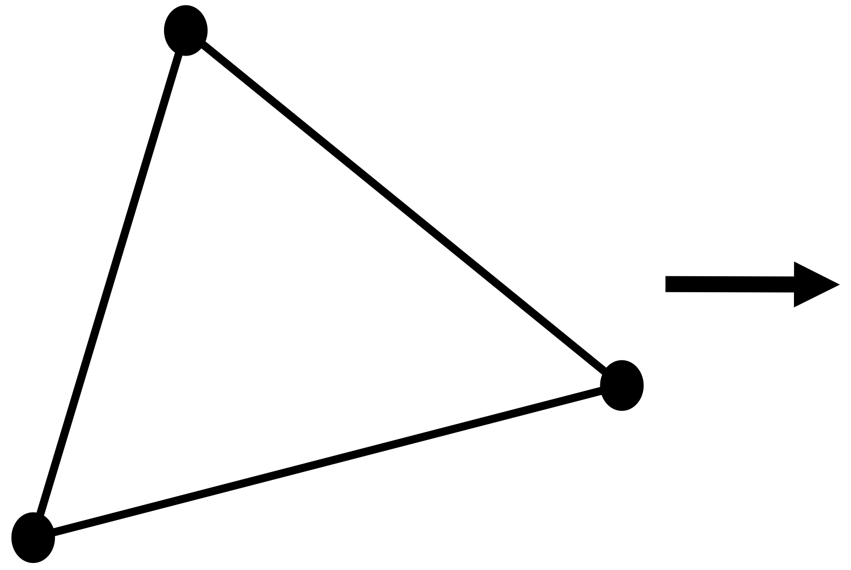
Another solution



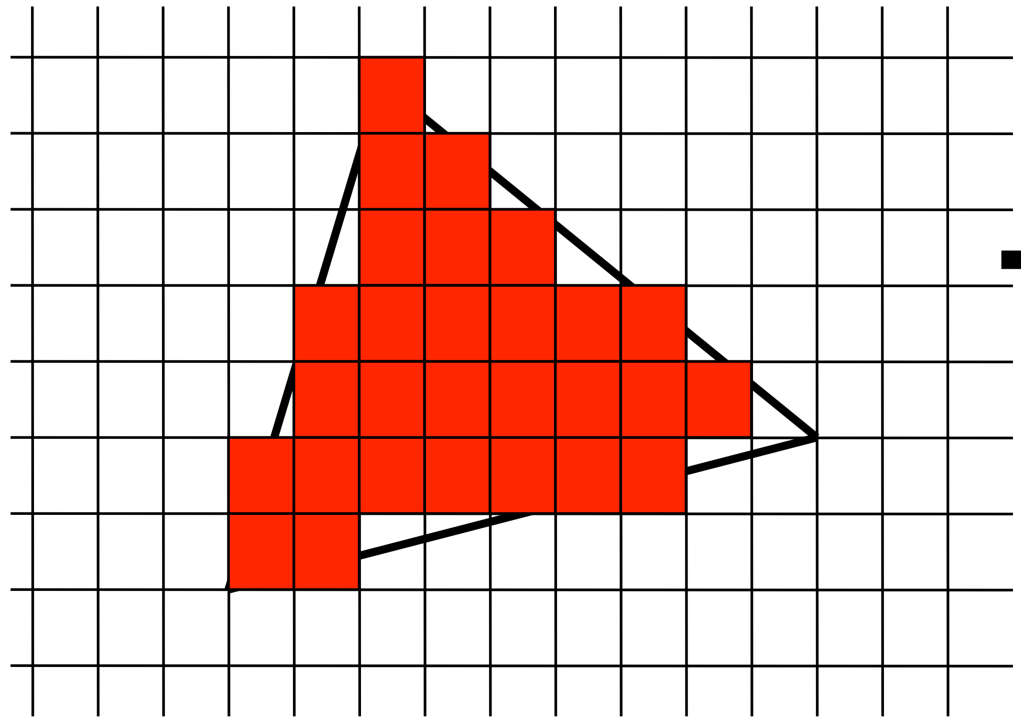
- Don’t move the grid
- Choose one direction, say southwards:
- The sampling point should only belong to the triangle that has the arrow in it
- Can be determined from looking at the ”normals” of the edge functions
- Edges sharing sample points is the most common problem, so solve that first...

[Idea by John Owens, UC Davis]

Edge functions



Pixel sampling

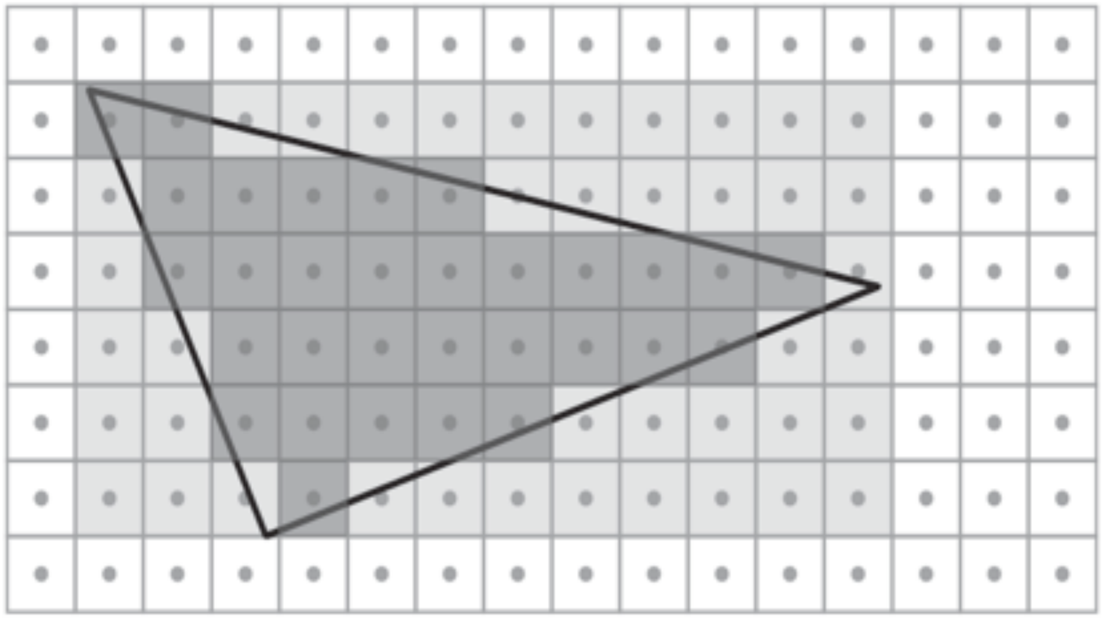


Traversal

Interpolation

Triangle traversal strategies

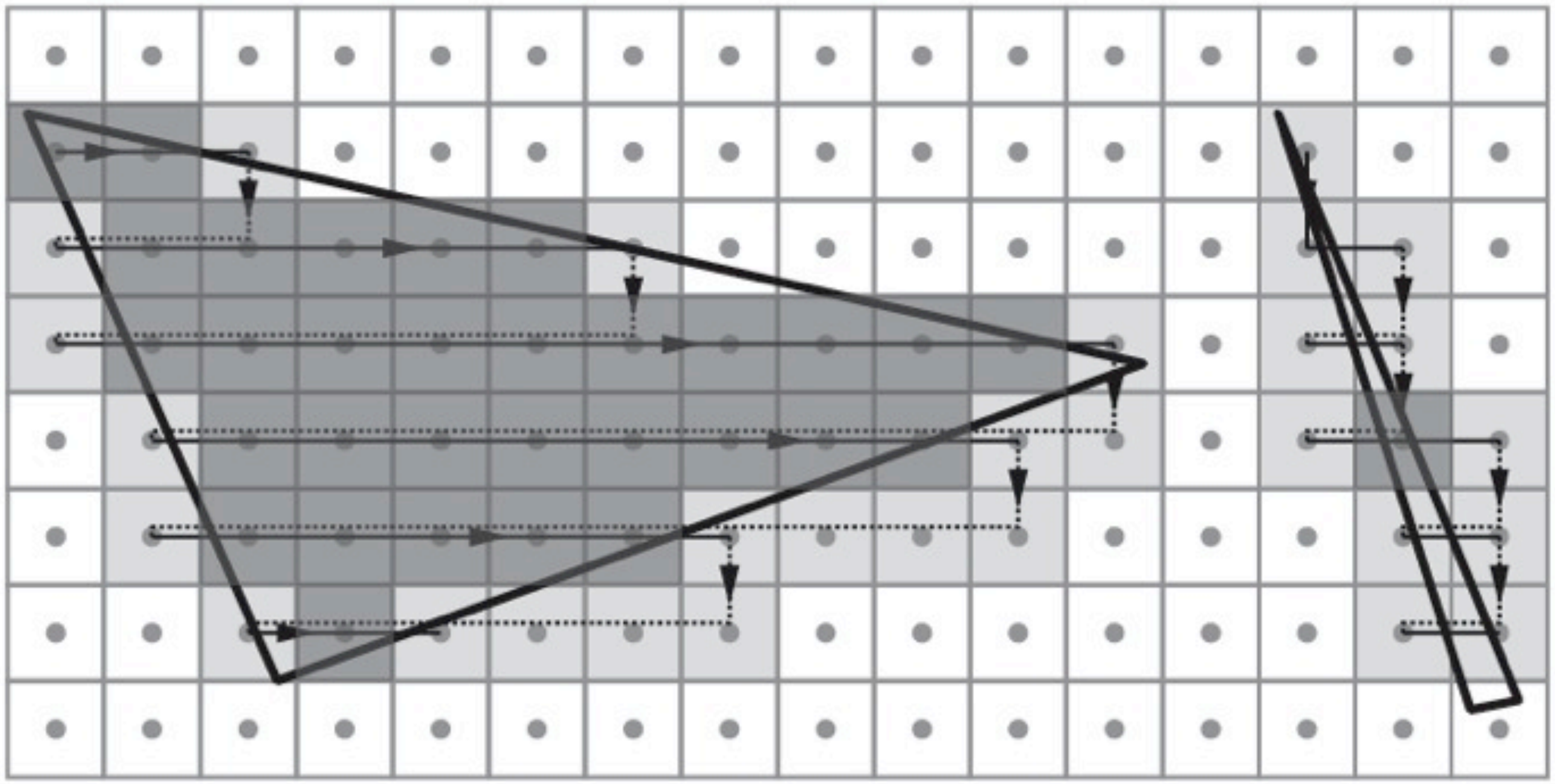
- Simple (and naive):
  - execute `Inside()` for every pixel on screen, and for every edge
- Little better: compute bounding box first
- Called "bounding box traversal"



Visits all gray pixels

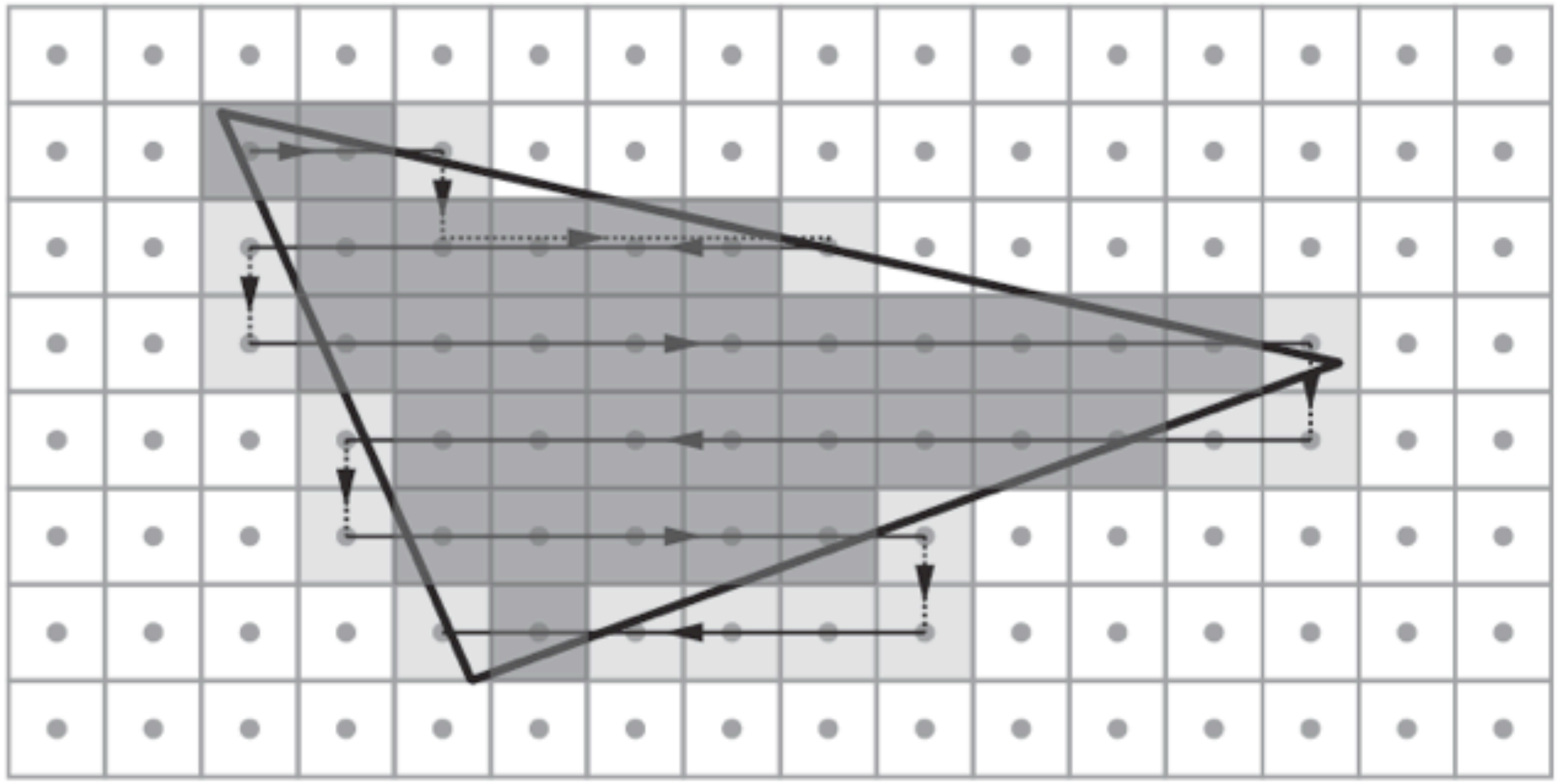
Only dark gray pixels are inside
So only keep those

Backtrack traversal



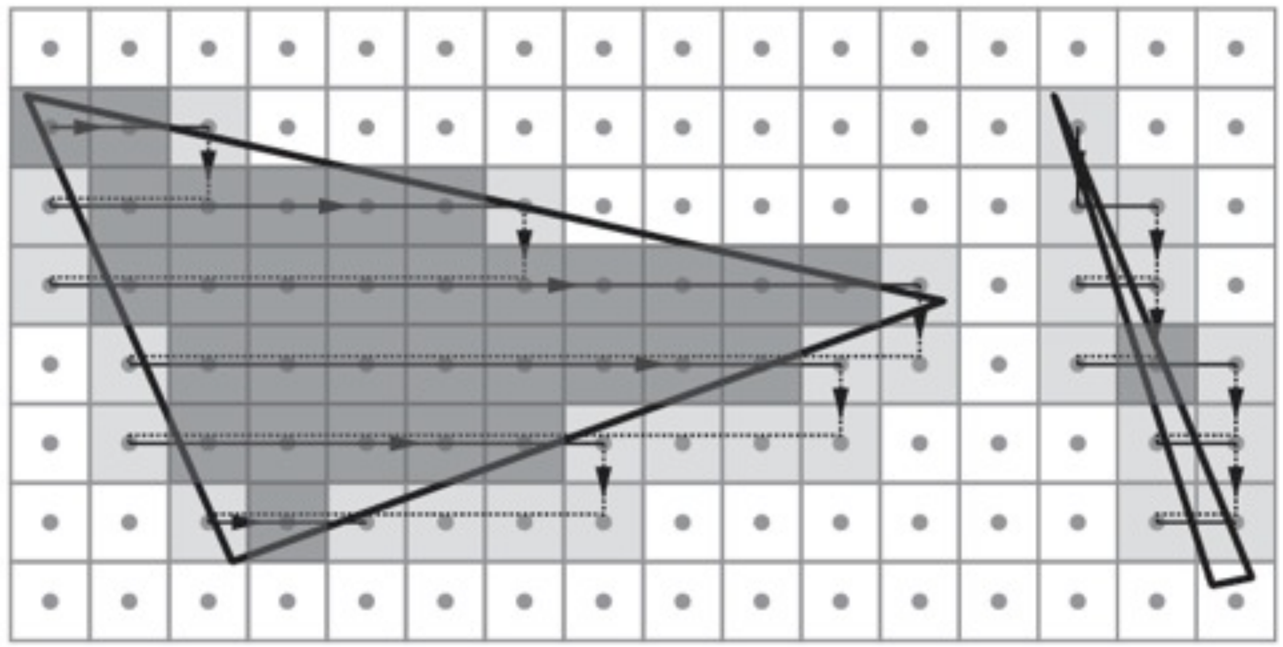
- Was used for mobile graphics chip
  - by Korean research group (KAIST)
- Advantage: only traverse from left to right
  - Could make for more efficient memory accesses
  - Could backtrack at a faster pace (because no mem acc)

Zigzag traversal

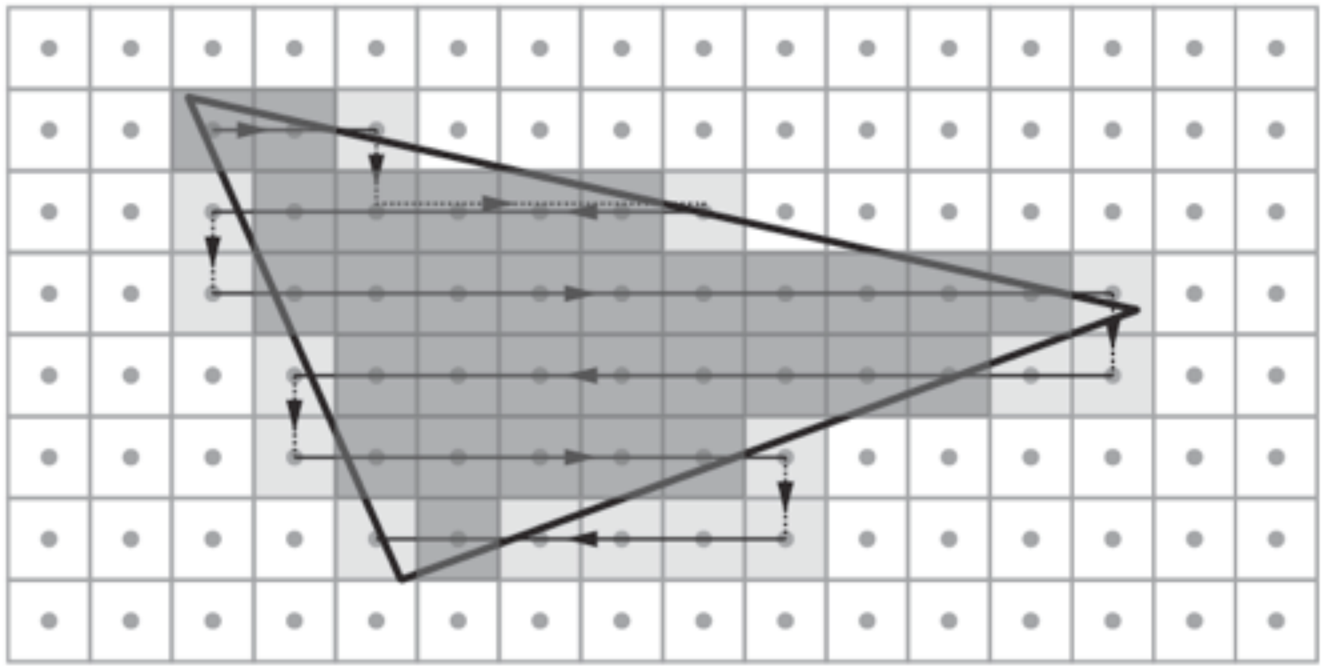


- Simple technique that avoids backtracking
- Still visits outside pixels
    - see the last scaneline

Side by side comparison Backtrack vs zigzag

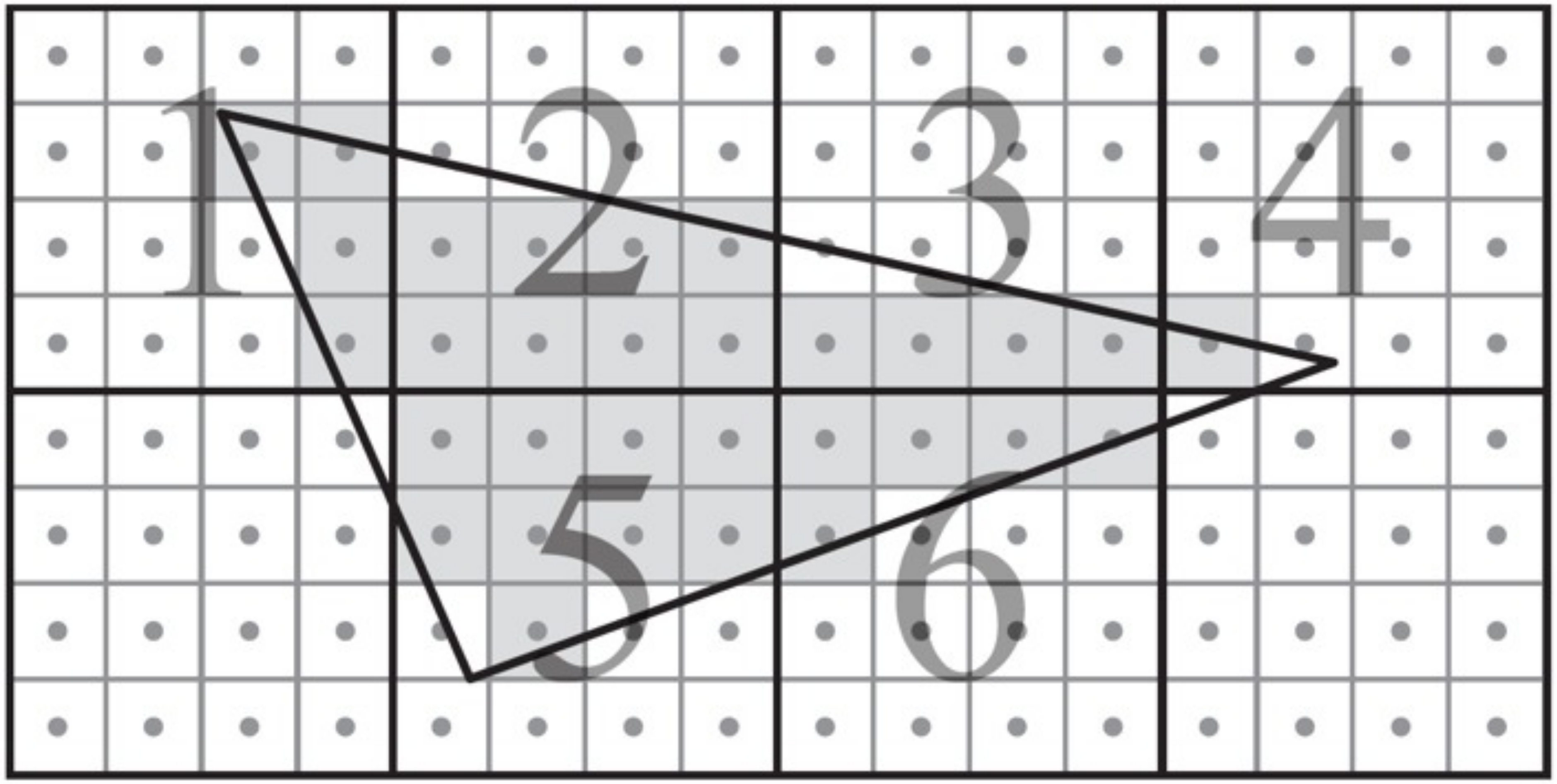


Backtrack never visits unnecessary pixels to the left



Zigzag never visits unnecessary pixels to the left on even scanlines and to the right on odd scanlines (and avoids backtracking)

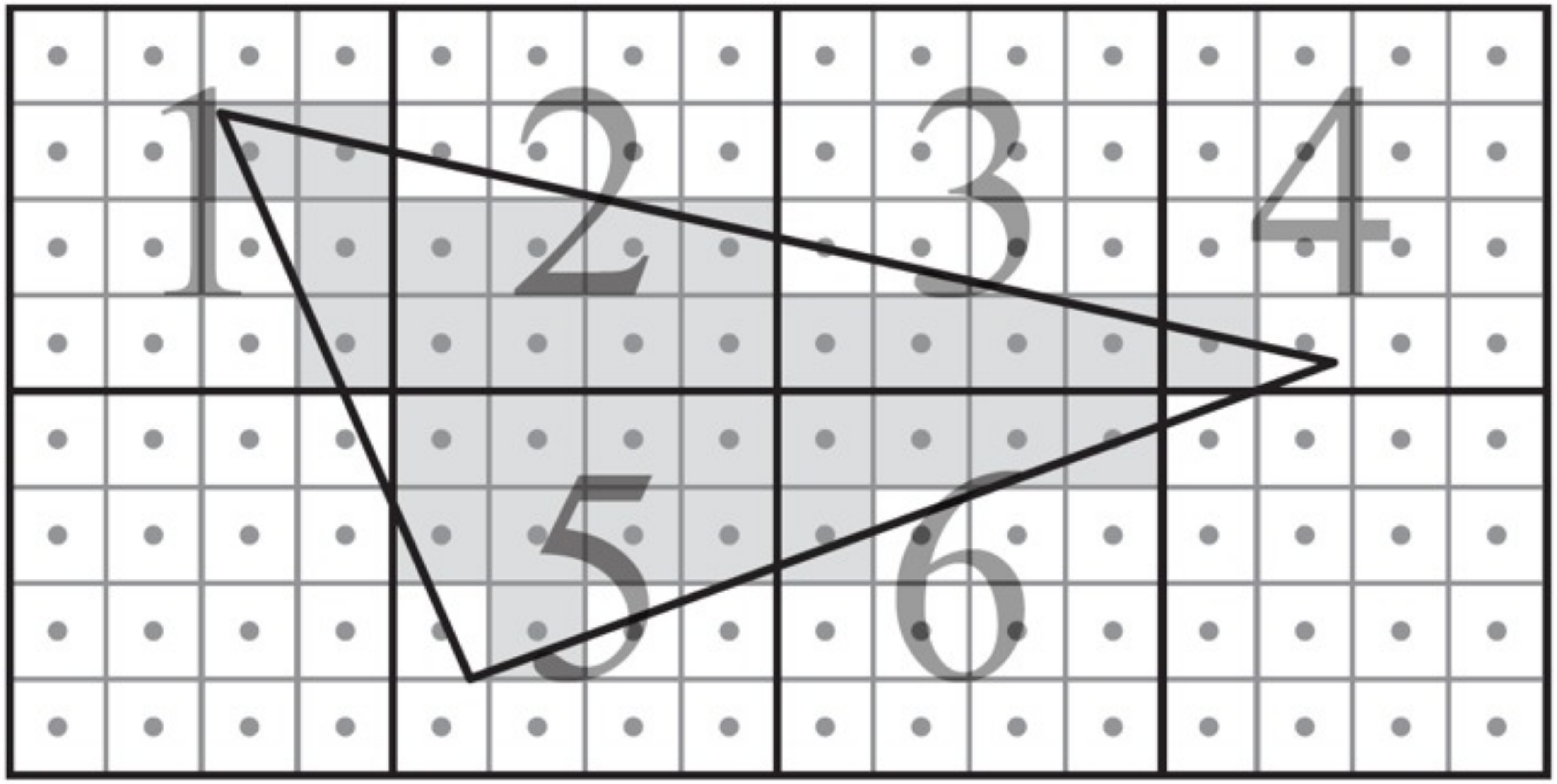
Tiled traversal



4x4 tiles

- Divide screen into tiles
    - each tile is *w* x *h* pixels
- 8x8 tile size is common in desktop GPUs

Tiled traversal



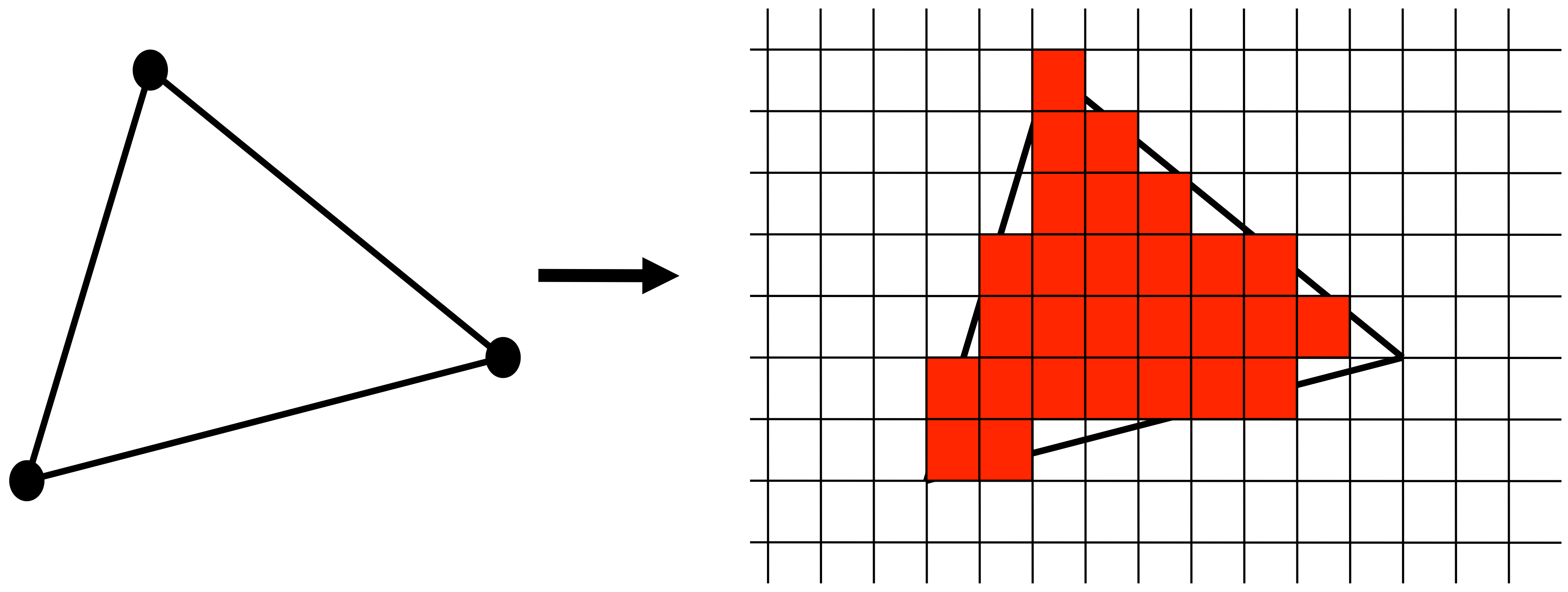
- Gives better texture cache performance
- Enables simple culling (Zmin & Zmax)
- Real-time buffer compression (color and depth)

Is tiled traversal that different?

- We need:
  - 1 : Traverse to tiles overlapping triangle
  - 2 : Test if tile overlaps with triangle
  - 3 : Traverse pixels inside tile
- We only need new algorithm for part 2
- Can use Haines and Wallace's box line intersection test (EGSR94)

Edge functions

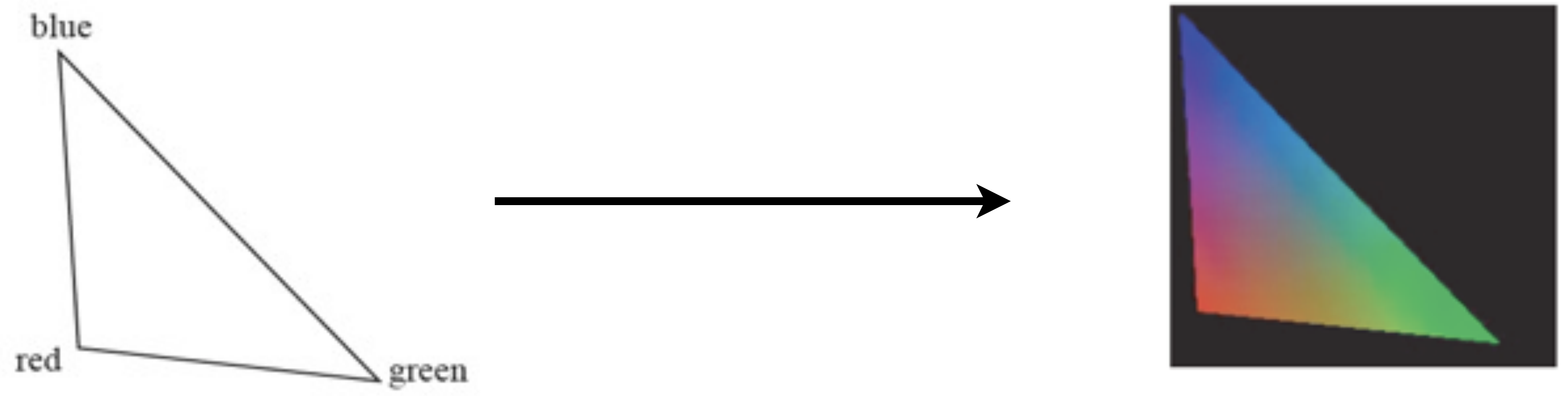
Pixel sampling

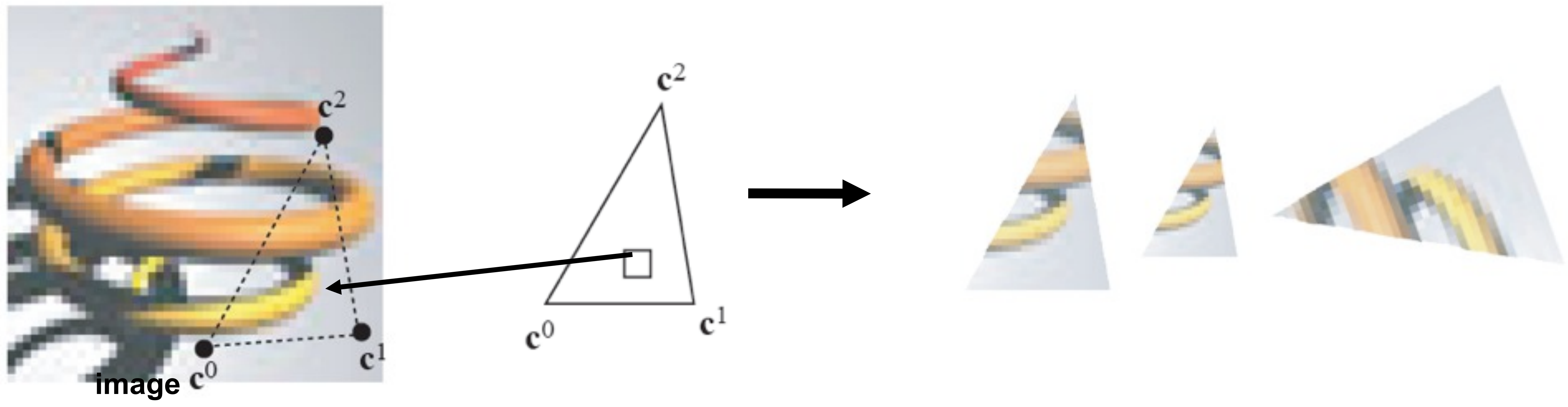


Traversal

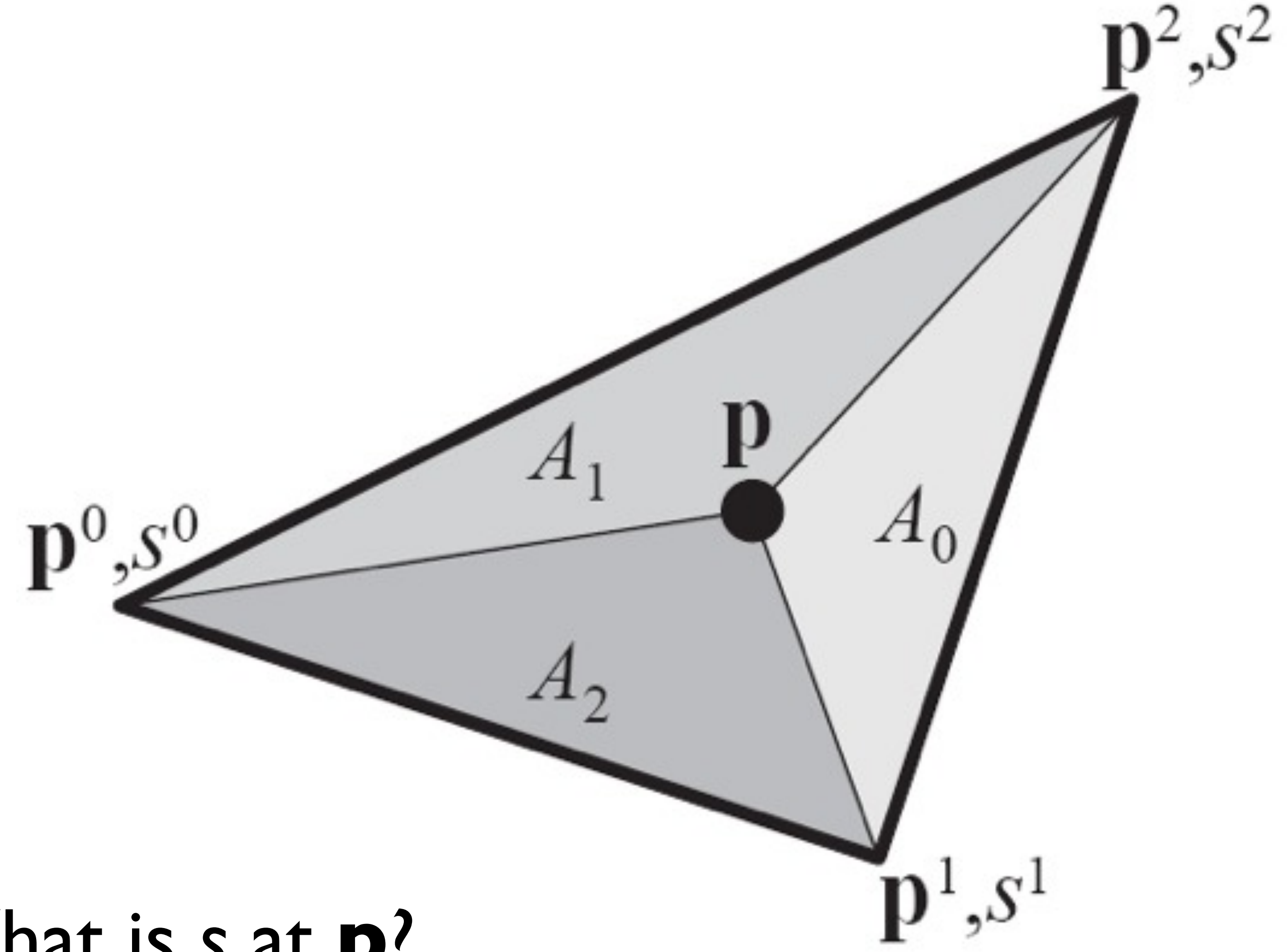
Interpolation

How can we interpolate parameters across triangles?





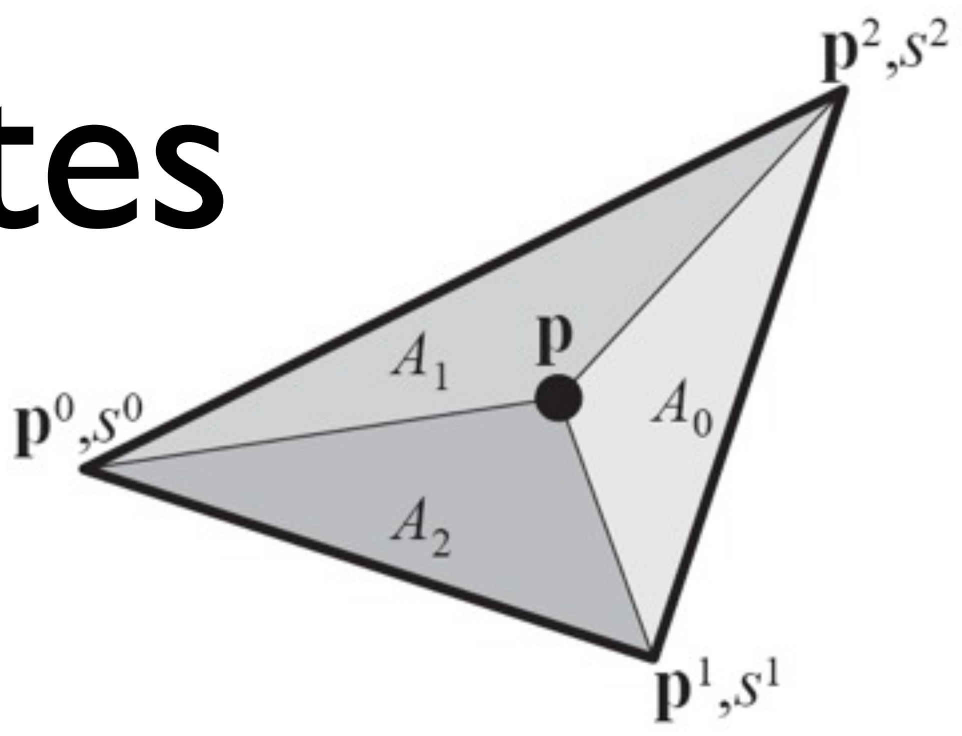
How can we interpolate parameters across triangles?



- What is *s* at **p**?
- S should vary smoothly across triangle
- Use barycentric coordinates, (*u*,*v*,*w*)

Barycentric Coordinates

Proportional to the signed areas of the subtriangles formed by p and the vertices



Area computed using cross product, e.g.:

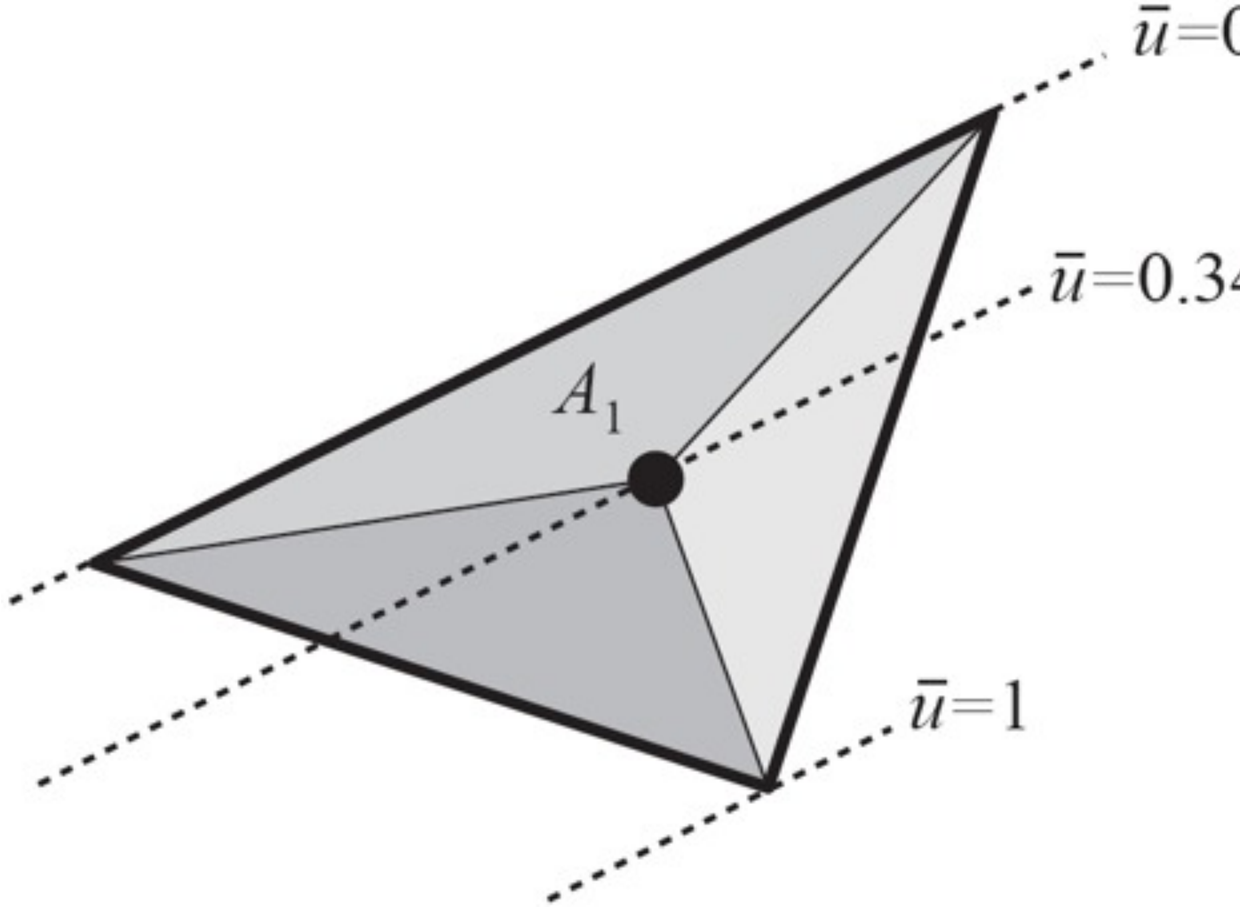
$$A_1 = \frac{1}{2}((p_x - p_x^0)(p_y^2 - p_y^0) - (p_y - p_y^0)(p_x^2 - p_x^0))$$

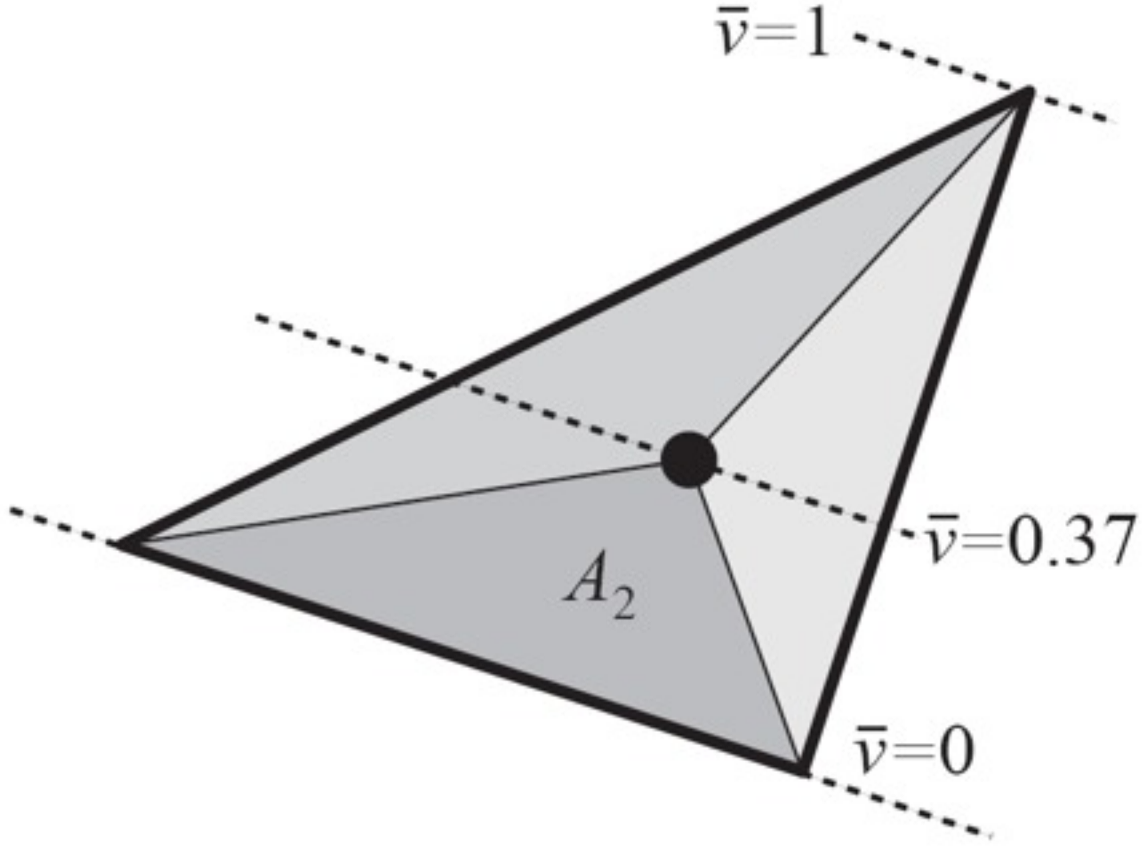
In graphics, we use barycentric coordinates normalized with respect to triangle area:

$$(\bar{u}, \bar{v}, \bar{w}) = \frac{(A_1, A_2, A_0)}{A_\Delta} \qquad A_\Delta = A_0 + A_1 + A_2$$

$$\bar{u} + \bar{v} + \bar{w} = 1 \qquad \bar{w} = 1 - \bar{u} - \bar{v}$$

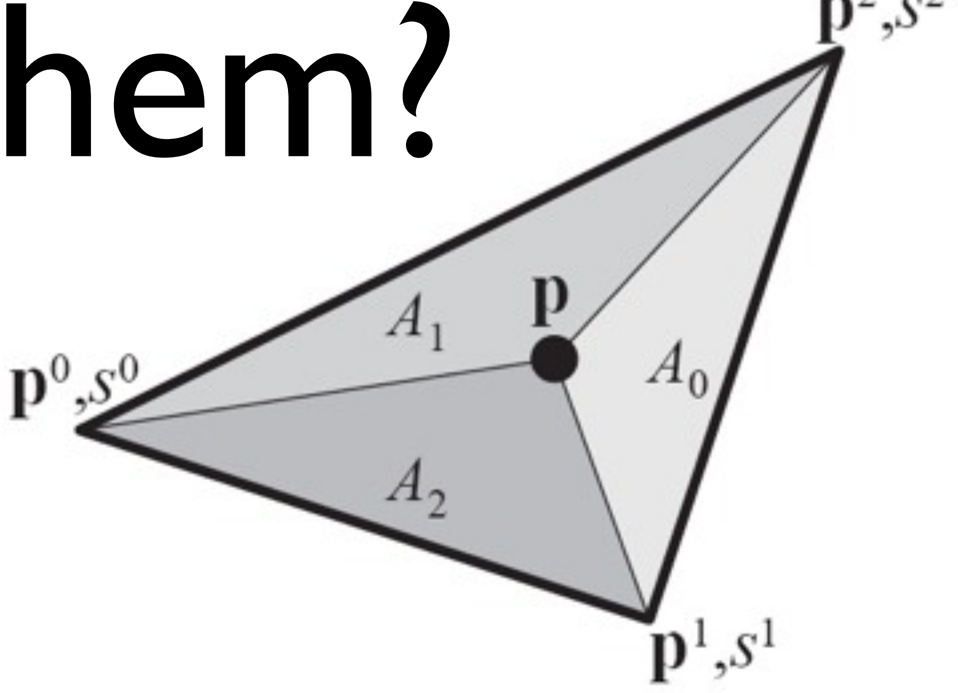
What do barycentric coordinates look like?





- Constant on lines parallel to an edge
  - because the height of the subtriangle is constant

How to use them?



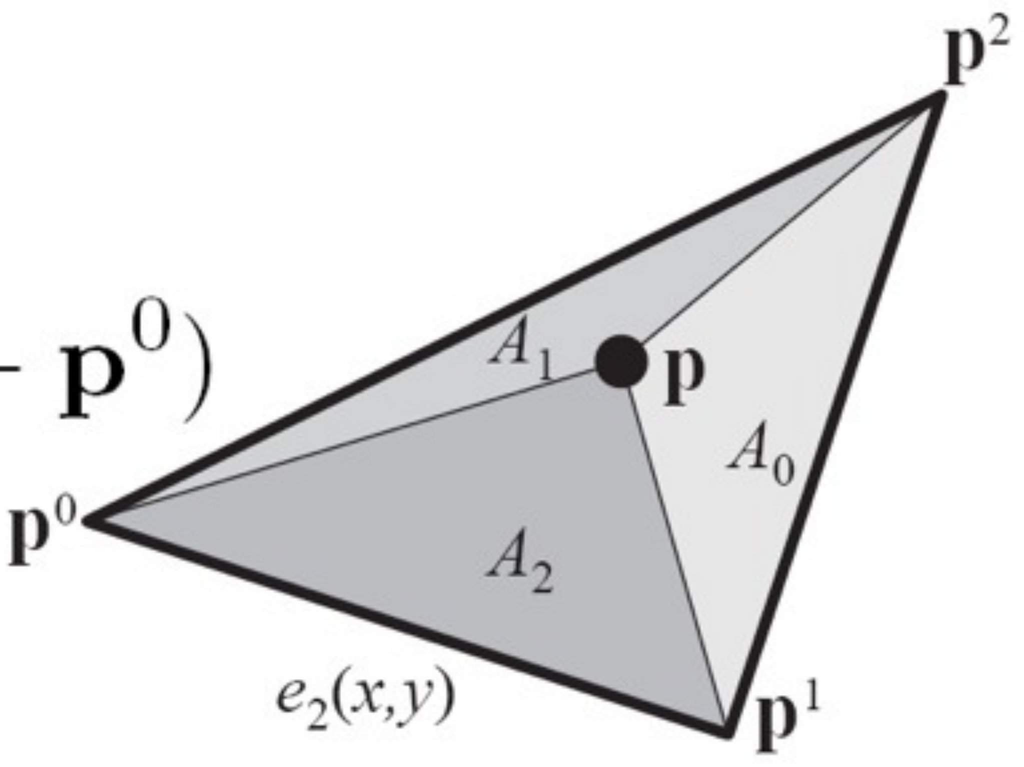
Interpolate vertex parameters s0, s1, s2

$$\begin{aligned} s &= \bar{w}s_0 + \bar{u}s_1 + \bar{v}s_2 = (1 - \bar{u} - \bar{v})s_0 + \bar{u}s_1 + \bar{v}s_2 \\ &= s_0 + \bar{u}(s_1 - s_0) + \bar{v}(s_2 - s_0). \end{aligned}$$

Barycentric coordinates from edge functions (1)

- The a and b parameters of an edge function must be proportional to the normal
  - We can use the edge functions directly to compute barycentric coordinates as well!
- Focus on edge, e_2:

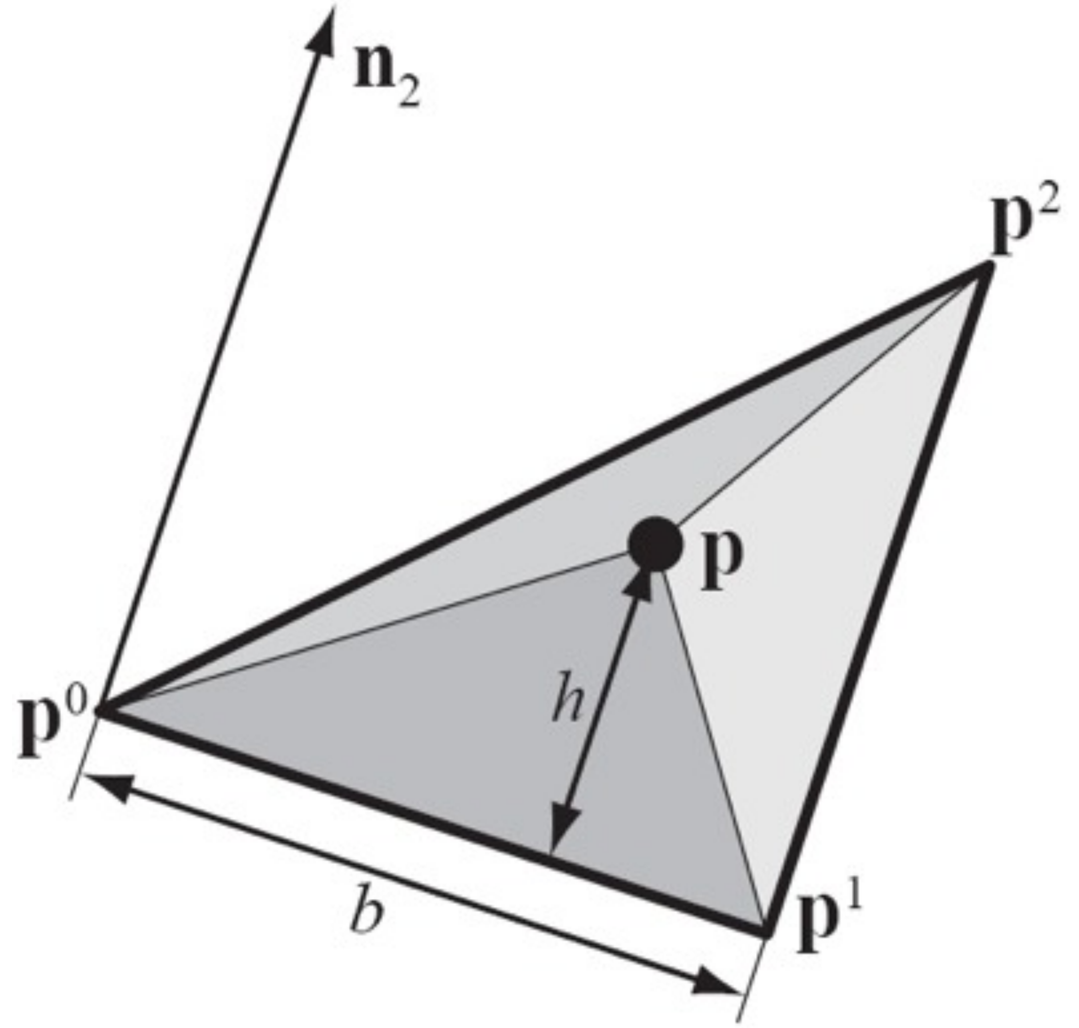
$$e_2(x, y) = e_2(\mathbf{p}) = \mathbf{n}_2 \cdot (\mathbf{p} - \mathbf{p}^0)$$



Barycentric coordinates from edge functions (2)

- From definition of dot product:

$$e_2(x, y) = e_2(\mathbf{p}) = \mathbf{n}_2 \cdot (\mathbf{p} - \mathbf{p}^0) \quad \Leftrightarrow$$
$$e_2(\mathbf{p}) = ||\mathbf{n}_2|| \, ||\mathbf{p} - \mathbf{p}^0|| \cos\alpha$$



- $||\mathbf{n}_2||$ must be exactly *b* (base of triangle)
- $||\mathbf{p}\text{-}\mathbf{p}^0||\cos\alpha$ is the length of projection of $\mathbf{p}\text{-}\mathbf{p}^0$ onto $\mathbf{n}_2$ i.e., *h* (height of triangle)

Barycentric coordinates from edge functions (3)

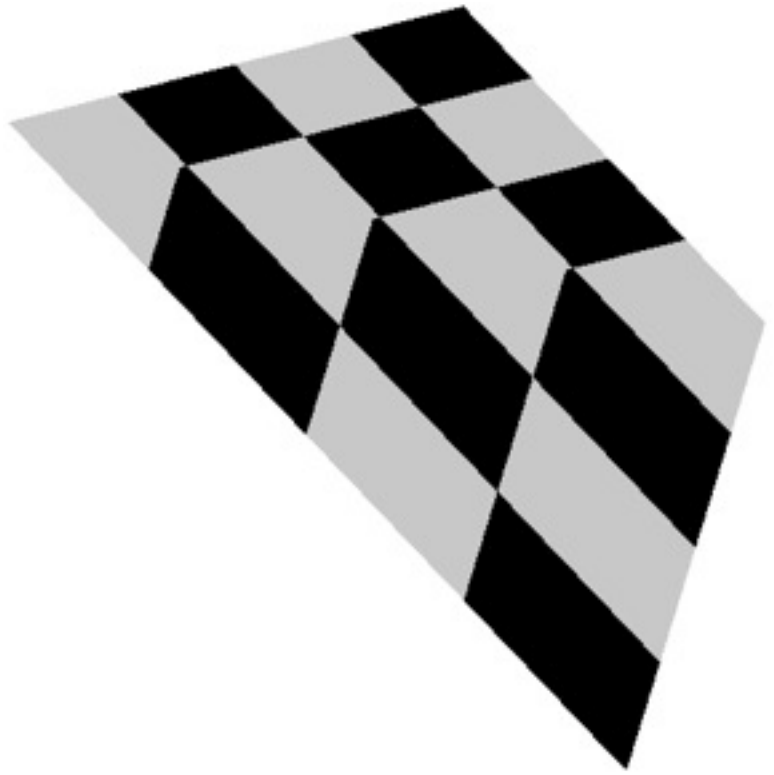
- This means:

$$\bar{u} = \frac{e_1(x,y)}{2A_\Delta}$$

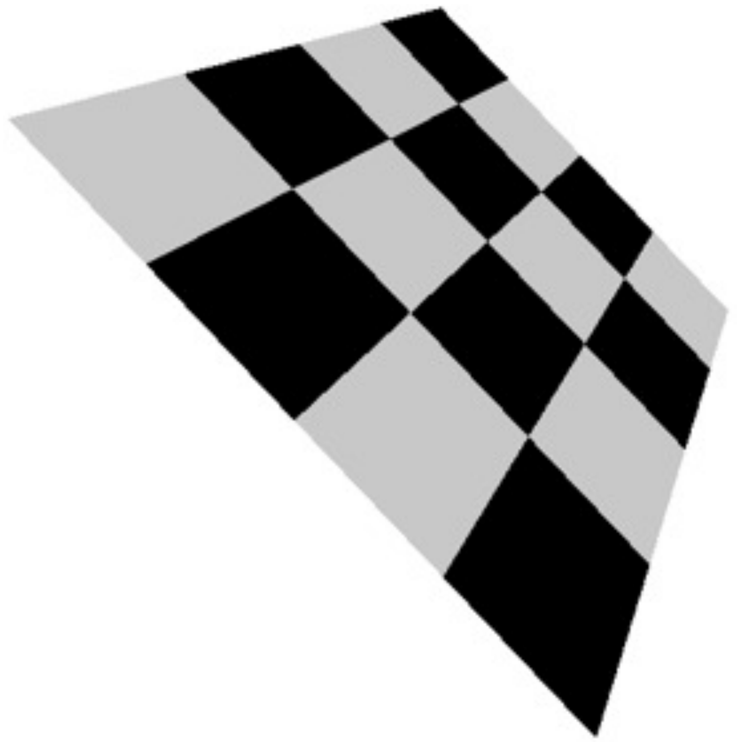
$$\bar{v} = \frac{e_2(x,y)}{2A_\Delta}$$

- And $1/(2A_\Delta)$ can be computed in the triangle setup (once per triangle)

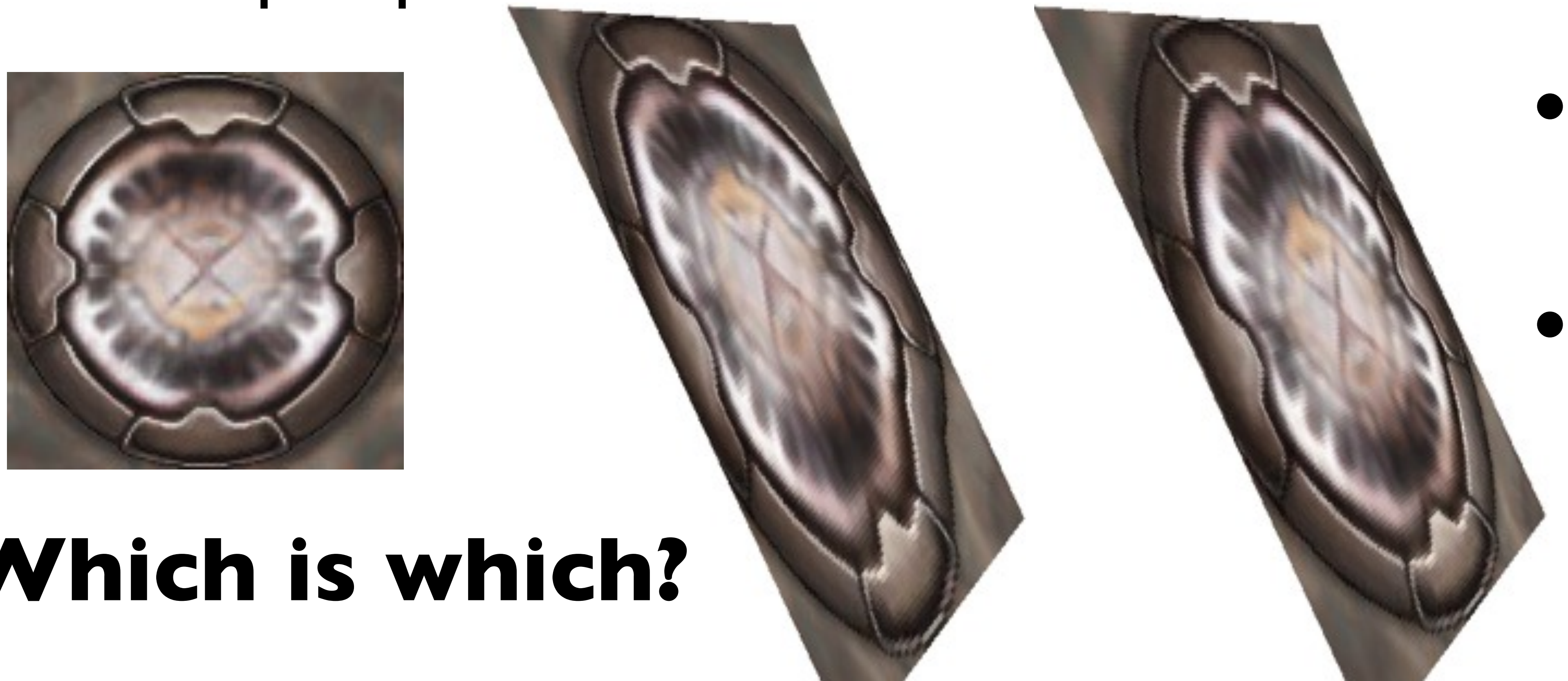
Resulting interpolation



With barycentric coordinates, i.e., without perspective correction



With perspective correction

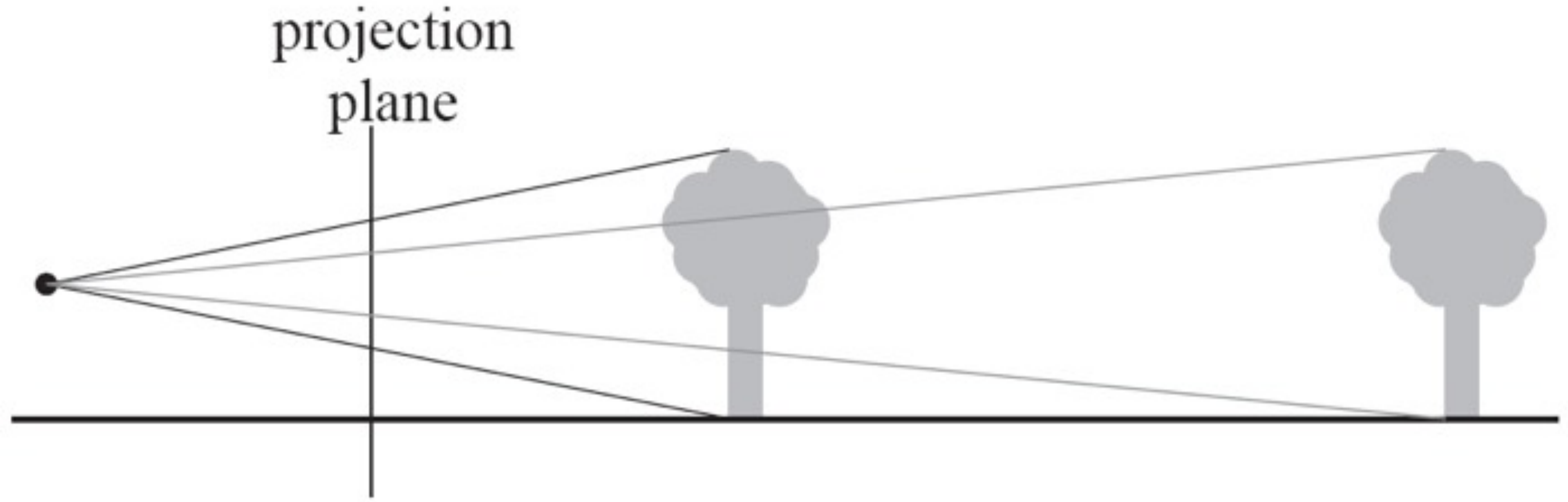


Which is which?

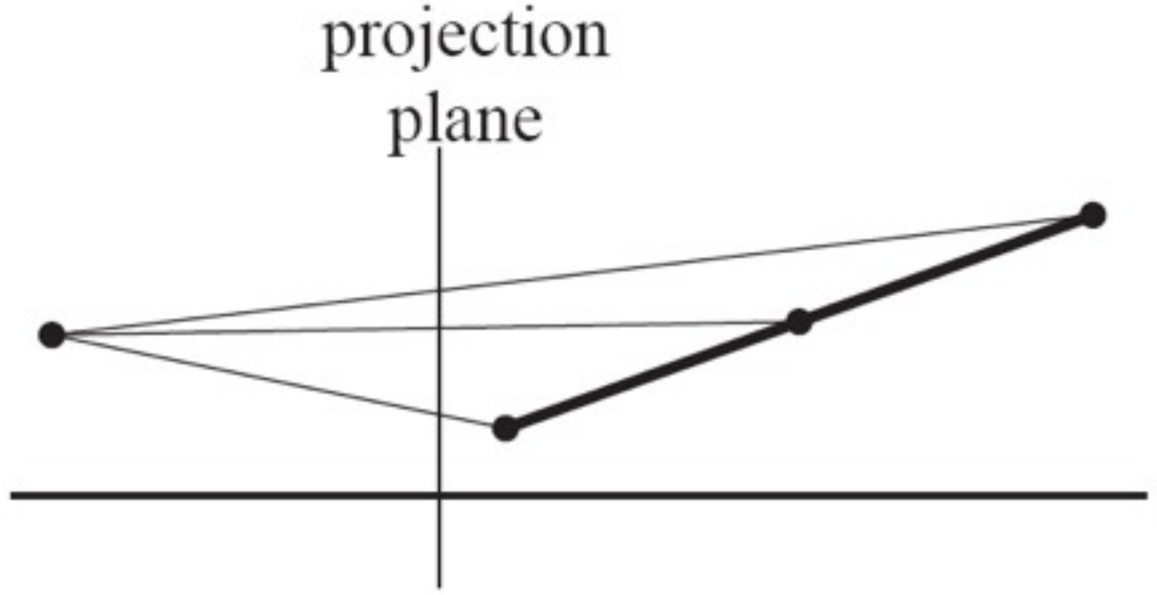
- Looks even worse when animated...
- Clearly, perspective correction is needed!

Perspective-correct interpolation

- Why?
  - Things farther away appear smaller!



- And even inside objects, of course:



Remember homogeneous coordinates

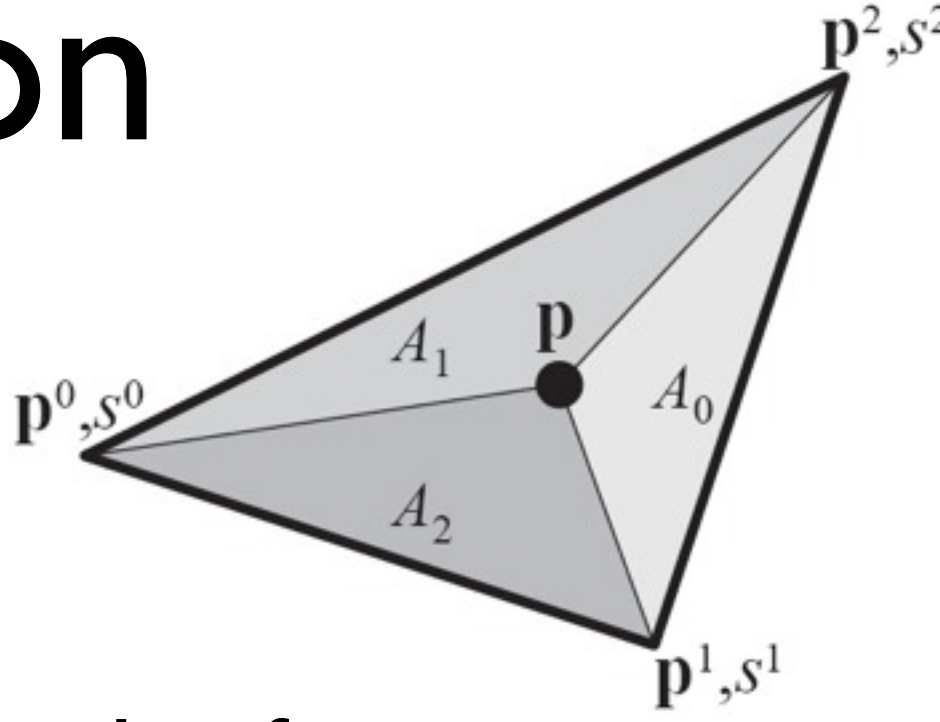
$$\mathbf{M}\mathbf{v} = \mathbf{h} = \begin{pmatrix} h_x \\ h_y \\ h_z \\ h_w \end{pmatrix} \Longrightarrow \begin{pmatrix} h_x/h_w \\ h_y/h_w \\ h_z/h_w \\ h_w/h_w \end{pmatrix} = \begin{pmatrix} h_x/h_w \\ h_y/h_w \\ h_z/h_w \\ 1 \end{pmatrix} = \mathbf{p}$$

$\mathbf{M}$ is a projection matrix

$\boldsymbol{p} = (p_x, p_y, p_z, 1)$ in screen space

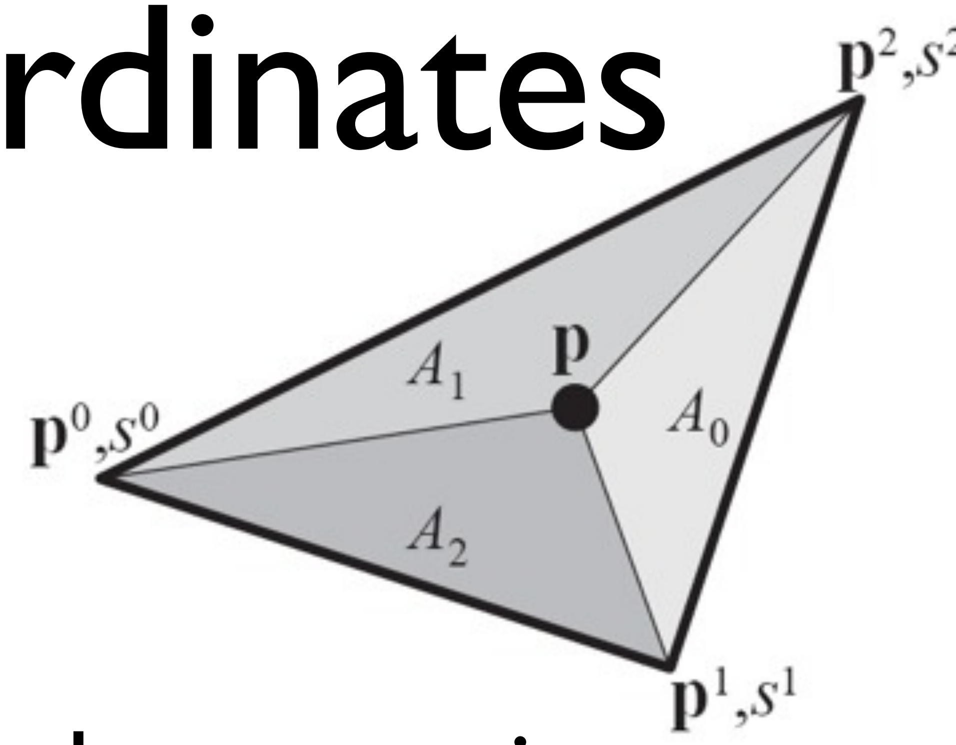
Perspective correct interpolation

$$\frac{s/w}{1/w} = \frac{sw}{w} = s$$



- An overly simplified way to think of it
- Linearly interpolate
  - s/w in screen space
  - 1/w in screen space
- Then divide

Perspective correct interpolation coordinates



- Compute perspective correct barycentric coordinates *(u,v,w)* first
- Then interpolate vertex parameters

$$s(p_x, p_y) = (1 - u - v)s^0 + us^1 + vs^2 = s^0 + u(s^1 - s^0) + v(s^2 - s^0)$$

Perspectively correct barycentric coordinates

Recall perspective correction

$$u(p_x, p_y) = \frac{\hat{s}(p_x, p_y)}{\hat{o}(p_x, p_y)}$$

$$\begin{aligned} \hat{s}(p_x, p_y) &= (1 - \bar{u} - \bar{v})\frac{0}{h_w^0} + \bar{u}\frac{1}{h_w^1} + \bar{v}\frac{0}{h_w^2} \\ \hat{o}(p_x, p_y) &= (1 - \bar{u} - \bar{v})\frac{1}{h_w^0} + \bar{u}\frac{1}{h_w^1} + \bar{v}\frac{1}{h_w^2} \end{aligned}$$

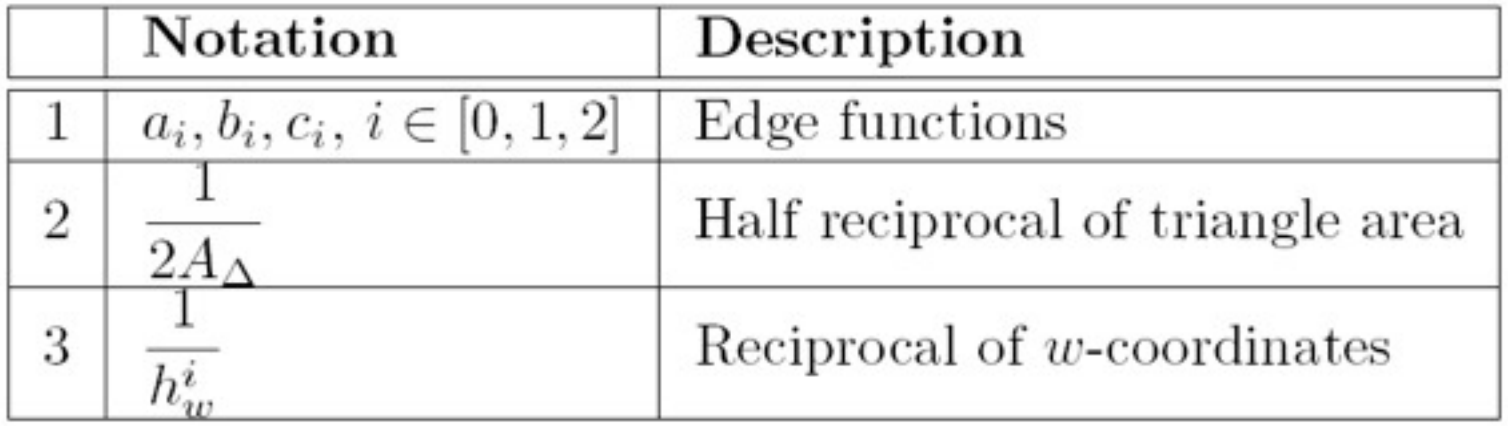
Simplify: $u(p_x, p_y) = \dfrac{\dfrac{e_1}{h_w^1}}{\dfrac{e_0}{h_w^0} + \dfrac{e_1}{h_w^1} + \dfrac{e_2}{h_w^2}}$

$$\begin{aligned} u &= \frac{f_1}{f_0 + f_1 + f_2} \\ v &= \frac{f_2}{f_0 + f_1 + f_2} \end{aligned}$$

$$f_0 = \frac{e_0(x, y)}{h_w^0}, \qquad f_1 = \frac{e_1(x, y)}{h_w^1}, \qquad f_2 = \frac{e_2(x, y)}{h_w^2}$$

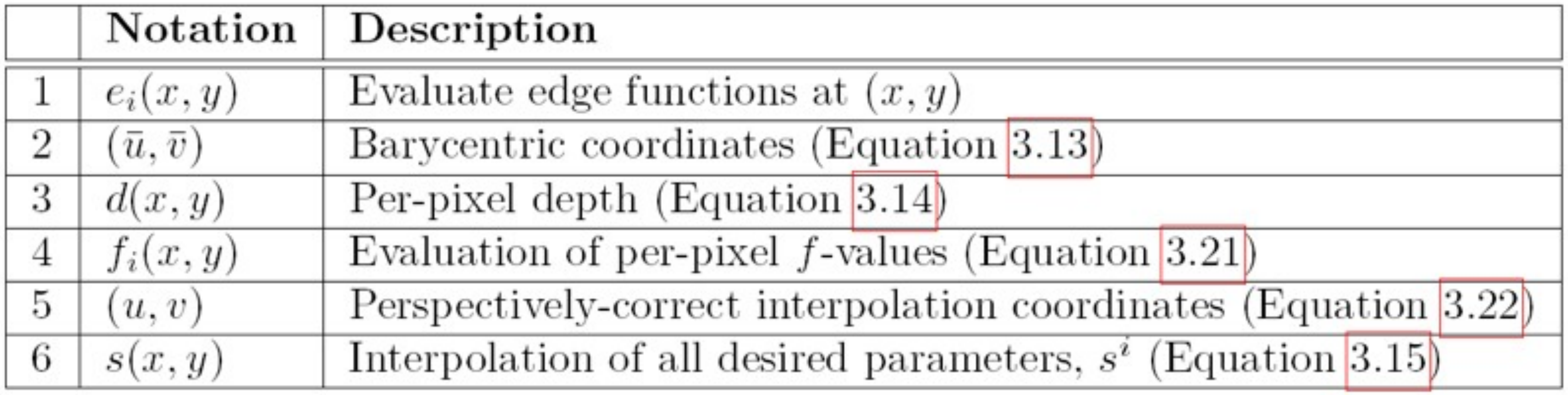
Once per triangle vs Once per pixel

Triangle setup



	Notation	Description
1	$a_i, b_i, c_i,\ i \in [0, 1, 2]$	Edge functions
2	$\frac{1}{2A_\Delta}$	Half reciprocal of triangle area
3	$\frac{1}{h_w^i}$	Reciprocal of w-coordinates

Per pixel (simple)



	Notation	Description
1	$e_i(x, y)$	Evaluate edge functions at (x, y)
2	$(\bar{u}, \bar{v})$	Barycentric coordinates (Equation 3.13)
3	$d(x, y)$	Per-pixel depth (Equation 3.14)
4	$f_i(x, y)$	Evaluation of per-pixel f-values (Equation 3.21)
5	(u, v)	Perspectively-correct interpolation coordinates (Equation 3.22)
6	$s(x, y)$	Interpolation of all desired parameters, s^i (Equation 3.15)

What's next

- Chapter 2 & 3 in Graphics Hardware notes
  - Rasterization and interpolation
- Next Week
  - Fixed point math (for the sub-pixel sampling)
  - Texturing
  - Caching
    - Read chapter 5.5 General Caching
- Assignment 1 available on the web page

The End!